Patterns of persistence, Part 2: Increase code reuse and enhance performance

More strategies and best practices for modern ORM tools

Ryan Senior
Travis Klotz
Jim Majure

22 April 2008

Part 1 of this two-part article covers the basics of achieving a consistent, concise domain model and persistence tier with modern object-relational mapping (ORM) tools. In Part 2, the authors describe base domain entities, behavior in the domain model, and more-advanced features of a generic DAO. They also share strategies for enhancing data-retrieval performance with the domain model.

View more content in this series

Introduction

Part 1 of this two-article series covers several basic best practices for Hibernate and object-relational mapping (ORM) tools in general. Through common base domain classes and interfaces, centralized auditing, and a generic data access object (generic DAO), your applications can have a more concise and maintainable domain model and persistence tier.

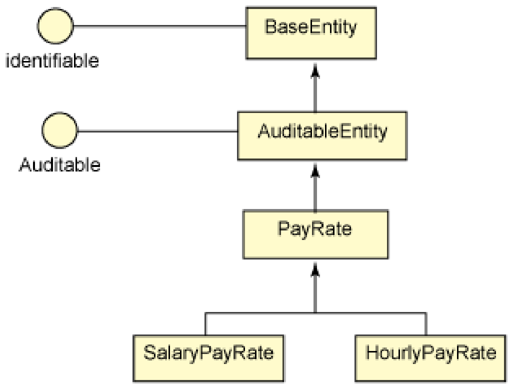
Applying the concepts in Part 1 can open new opportunities for code reuse. We'll start by presenting ways you can use Hibernate and polymorphism to incorporate behavior in the domain model. Next, we'll build on Part 1's discussion of the generic DAO. Once you incorporate and use a generic DAO in an application, you'll encounter more potential operations that are common across applications. We'll show how you can reduce code by incorporating paging of data and querying in the generic DAO. We finish with strategies for enhancing performance with the domain model. Without these strategies, incorrectly configured associations in the domain model can cause thousands of extra queries to be executed or can waste resources by retrieving records that are not needed.

Revisiting the model: Letting the ORM select behavior

Because database tables by themselves do not have behavior, it's tempting to put the behavior of domain-model entities in services or the view layer. This is undesirable because it violates a fundamental rule of object orientation: *Objects have behavior and data*. If the behavior is removed from the object and put into a service, the object then becomes a mere holder for data. And putting entities' behavior in services or the view layer scatters the entities' core logic throughout the application, causing maintenance problems. Tools like Hibernate make it much easier to put this behavior in the model along with their data, and in general gravitate toward a more domain-driven model.

Continuing the employee example used in Part 1, Figure 1 shows an object model that defines pay-rate functionality:

Figure 1. Object model for pay-rate calculations



Assume that the database has an `Employee` table that's related to a `PayRate` table. The pay-rate table has a column named `employeeType` with two possible values: `Hourly` and `Salary`. The algorithm used to calculate an employee's pay depends on that column's value, because pay calculation for hourly employees differs from pay calculation for salaried employees. Salaried employees receive the same pay from week to week regardless of the number of hours they work. Hourly workers must be paid for each hour worked, with the added possibility of an overtime calculation.

Putting the pay-calculation code in the domain entities leads to a more cohesive domain model, and the logic isn't scattered through the tiers. The pay-rate example uses single-table inheritance, for which you first need to define a *discriminator* — a column in the table that tells Hibernate the type of object to instantiate to represent the data in the database. For this example, the `employeeType` column acts as the discriminator. Next, you must define the superclass that serves as the base class for these entities — the abstract `PayRate` class in the example. Listing 1 shows the annotation and class declaration for the superclass:

Listing 1. Annotation and class decoration for superclass

```
               @Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn( name="employeeType", discriminatorType=DiscriminatorType.STRING )
public abstract class PayRate extends BaseEntity{//...}
```

Finally, you need to create the implementations for each of the possible subclasses. Listing 2 defines the subclasses for salaried employees and hourly employees:

Listing 2. Subclass definitions

```
                @Entity
@DiscriminatorValue("Salary")
public class SalaryPayRate extends PayRate {//...}

@Entity
@DiscriminatorValue("Hourly")
public class HourlyPayRate extends PayRate {//...}
```

When querying the `PayRate` table, Hibernate automatically creates an instance of `SalaryPayRate` when `employeeType="Salary"` or an instance of `HourlyPayRate` when `employeeType="Hourly"`. The application code can then call the `createPayCheck` method with the assurance that the correct algorithm for calculating pay is being used. Hibernate is acting as a kind of factory for these classes, creating the correct instance at the correct time.

Hibernate is also aware of polymorphism when writing queries. When it writes a query to find all salaried employees, Hibernate appends the `employeeType="Salary"` line to the query to achieve the desired results. Listing 3 shows an example of a polymorphic query; note that there's nothing special or different about it:

Listing 3. Polymorphic query

```
Criteria crit = session.createCriteria(PayRate.class);
crit.add(Restictions.eq("jobRole", "Programmer");
...
Criteria salaryCrit = session.createCriteria(SalaryPayRate.class);
salaryCrit.add(Restrictions.eq("yearlyRate",50000.00));
```

Sometimes an entity's discriminator isn't as straightforward as a single column. Perhaps more than one column should be factored into the logic of the discriminator, or the logic is different from the basic *when column A equals X* logic. It's also possible to define a formula for determining the instance to create. One example is to create an instance of one class if a value in a column is null and to create a different instance if it has some value. These types of strategies are generally signs that the data model should be refactored, but that may not be an option for legacy data models.

Polymorphism for information hiding

Using Hibernate to select the behavior of an entity in the model is good, but often you also need different sets of data for different entities. If fields make sense for one subclass but not the other, you can remove them from the superclass and push them down into the subclass they belong to. This could prevent future maintainers of the code from inadvertently using a field that doesn't belong to that instance.

For example, there's a piece of information needed for calculating an hourly employee's pay that isn't needed for calculating a salaried employee's pay. Overtime pay doesn't make sense in the context of a salaried employee, so it can be pushed down from `PayRate` into the `HourlyRate` class. Likewise, the `yearlyRate` field does not belong in the `HourlyRate` class and can be pushed down into the `SalaryRate` class.

`PayRate` defines fields common to both hourly and salary employees, as shown in Listing 4:

Listing 4. The `PayRate` class

```
public abstract class PayRate extends BaseDomainEntity{

public String getJobRole(){//...}

@OneToMany(cascade = CascadeType.ALL)
@OrderBy("payPeriodBeginDate")
public List<PayCheck> getPayChecks(){//...}

@OneToOne(cascade = CascadeType.ALL)
public Employee getAssociatedEmployee(){//...}

}
```

Then the `SalaryRate` class inherits those fields and adds a `yearlyRate` field, as shown in Listing 5:

Listing 5. The `SalaryRate` class

```
@Entity
@DiscriminatorValue("Salary")
public class SalaryPayRate extends PayRate {
    private double yearlyRate;
    //...
}
```

Advanced generic DAO

As more and more applications use your generic DAO, you'll identify more commonality in the data access tier. You can reduce code duplication by abstracting the common behavior and including it in the generic DAO, where all users of the class will have access to the functionality.

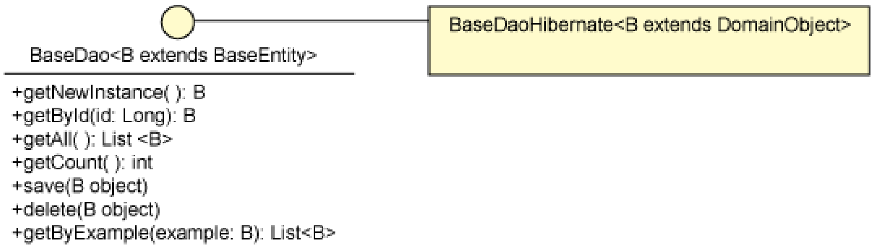
Two common operations that occur frequently are querying based on search parameters and database-level pagination.

Querying

Almost all persistent entities require some type of query functionality. The query is typically specified by entering search text (sometimes with wildcard functionality) for one or more of the fields in the object to be searched. One approach to querying is to create an instance of the persistent object and populate the fields on which to search with the search criteria. Hibernate supports this approach through *example* queries.

Figure 2 shows the `BaseDao` interface, which has been extended to support query-by-example functionality:

Figure 2. The extended `BaseDao` interface



Before we show you an implementation of this interface, let's consider how such a DAO would be used. Consider the following code, which illustrates the process of executing a query for a particular persistent entity:

```
Employee example = employeeDao.getNewInstance();
example.setLastName("Smith");
List<Employee> results = employeeDao.getByExample(example);
```

In a Web application example, the search parameters would probably be the backing object for a search form, and the query's results query would be presented in a user interface. Listing 6 shows the way the `getAllByExample()` method might be implemented using Hibernate:

Listing 6. `getAllByExample` implementation

```
                List<B> getAllByExample(B example) {
    Criteria criteria = getCurrentSession().createCriteria(getQueryClass());
    Example hibExample = Example.create(example);
    return criteria.add(hibExample).list();
}
```

The power of this approach is that it enables considerable code reduction. Not only is the DAO code reduced through the use of the generic DAO, but it is also possible to create generic UI plumbing code that is agnostic about the type of data being queried and displayed. It simply knows that it needs to collect some data from the UI, use it to execute a query, and present the results to the user.

Pagination

When database queries that produce large result sets are executed, it's common to break the result set into pages and then require the user to navigate among the pages. There are two approaches to accomplishing pagination. The first is to execute the query, retrieve the entire result set from the database, and then simply display the results to the user one page at a time. This approach incurs the cost of transferring the full result set and storing it in the user's session.

An alternative approach is to push the paging all the way to the database. This is effective in reducing resource consumption but requires a somewhat more complicated programming model. Luckily, you can make much of this functionality generic by extending the generic DAO. The `BaseDao` interface is extended once more in Figure 3:

Figure 3. The `BaseDao` interface, extended for pagination

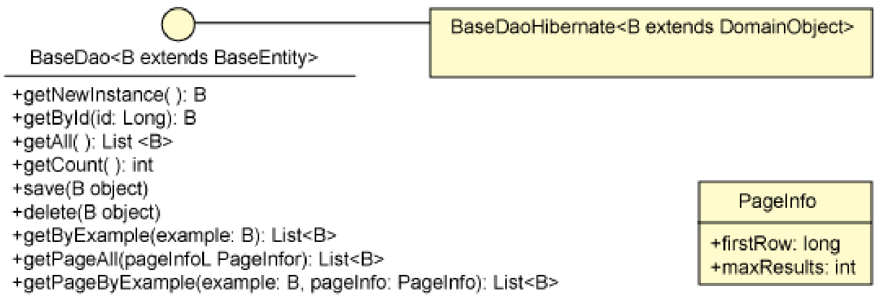

BaseDaoHibernate<B extends DomainObject>
BaseDao<B extends BaseEntity>
+getNewInstance( ): B
+getById(id: Long): B
+getAll( ): List <B>
+getCount( ): int
+save(B object)
+delete(B object)
+getByExample(example: B): List<B>
+getPageAll(pageInfoL PageInfor): List<B>
+getPageByExample(example: B, pageInfo: PageInfo): List<B>
PageInfo
+firstRow: long
+maxResults: int

Figure 3 introduces the `PageInfo` class, which identifies the page of data that should be retrieved. The implementation of the `getPageAll` method is shown in Listing 7:

Listing 7. `getPageAll` implementation

```
public List<B> getPageAll(PageInfo pageinfo) {
    return getCurrentSession().createCriteria(getQueryClass()).
        setFirstResult(pageinfo.getFirstRow()).
        setMaxResults(pageinfo.getMaxResults()).list();
}
```

The example code included with this article shows you yet more functionality that can be added to the generic DAO (see Download).

Improving data-retrieval performance

When you performance-tune Hibernate applications, you'll spend the most development time tweaking how Hibernate deals with entity relationships. You'll want to minimize the amount of data pulled into the application and at the same time minimize the number of queries that need to be executed. Hibernate gives you two main ways of dealing with relationships: *lazy fetching* and *eager fetching*.

Lazy fetching

Lazy fetching minimizes the amount of data queried from the database. Instead of querying all data from an association when an object is loaded, queries for associations marked as lazy are deferred until that association is actually used. For example, the `Employee` object has an `Address` object associated with it. Whenever an `Employee` is loaded, Hibernate does not automatically load the `Address` because the association between these objects is lazy. Instead, when the `Employee` object is loaded, Hibernate creates a proxy instance of the `Address`. When accessed for the first time, the proxy asks the Hibernate session to query the `Address` object and then defer all future calls directly to the queried instance. If the `Address` object is never used during the `Employee` object life cycle, then the `Address` query never needs to execute. Using this feature ensures that Hibernate loads the data only when it is needed.

Lazy fetching should almost always be used as your default association-fetching strategy. Real-world object models have large numbers of highly complex object associations; not using lazy evaluation could easily result in the entire database being loaded into the session any time a single object is loaded. Enabling lazy evaluation is simple but varies slightly depending on the association type. Collection-based associations (`OneToMany` and `ManyToMany`) are set to lazy evaluation by default, so no configuration is necessary to benefit from lazy evaluation. Single-object associations (`OneToOne` and `ManyToOne`) are not lazy by default. To use lazy evaluation with these associations, specify it on the association's annotation:

```
@ManyToOne(fetch=FetchType.LAZY)
public void getAddress() {
```

Lazy mapping and HBM format

Implementing lazy mapping is slightly different when you use Hibernate's HBM format, instead of annotations, to define your mappings. Since version 3.0, for any type of association defined using HBM files, the default fetching strategy is lazy fetching. No additional configuration is necessary even for single-object associations.

Issues with lazy fetching

Although lazy fetching should be the primary fetching strategy for most applications, it does add a few complications. The most serious is the `LazyInitializationException`. A lazy association can be traversed (and the associated entity retrieved) long after its parent object was initially loaded. However, to query the association's data, the Hibernate session needs to be available and attached to a database connection from the pool. This error can occur if the Hibernate session and associated database connection is closed after the object was retrieved, but before the association was accessed. Hibernate throws a `LazyInitializationException` because the query is executed for the lazy association, and no connection to the database is associated with it. Many strategies for dealing with this issue are available, but that topic is beyond this article's scope (the "Open Sessions in View" entry in Resources links to a relevant discussion).

Another issue to consider is how many queries are executed when traversing lazy associations. The first time an instance of a lazy association is traversed, a query is executed. This isn't much of an issue with single instances of objects, but iterating over lists of objects can easily cause a large number of queries to be executed. Consider an example of a program that prints out the address associated with 10 employees. By default, if lazy fetching is used, 11 queries will be executed — one for the list of employees and one for each employee's address. This obviously can lead to serious performance issues. The problem is common enough to have been given its own name. Hibernate calls this the *n+1 Selects Problem*. (Martin Fowler calls it *ripple loading*.).

Eager fetching

Eager fetching is essentially the opposite of lazy fetching. When an object is loaded, any associations marked as eager are immediately loaded as well. When a single object is loaded, no fewer queries are run than if lazy association is used. But because the query runs immediately, there's no chance of a `LazyInitializationException`. Eager evaluation can also help with the *n +1 Selects Problem*. If the `Employee` object's `Address` association is eager, and a query is executed to retrieve ten `Employee` instances, two queries will fire: one for the list of `Employee` instances and one for a list of `Address` instances associated with the `Employee` objects. Hibernate then seamlessly merges these lists together.

As a default, eager fetching is probably not a good idea. If you use it too frequently, it's easy to load large trees of unneeded data. But sometimes an association is always needed, in which case working with eager evaluation can make sense. Enabling eager fetching is essentially the same process as enabling lazy fetching. Simply specify it on the annotation:

```
@ManyToOne(fetch=FetchType.EAGER)
public void getAddress() {//...}
```

Best practices and querying associations

Hibernate does have an elegant solution to addressing the issues with lazy fetching while still maintaining lazy fetching's positive attributes. Hibernate lets the definition of a query override the default fetching strategy for associations. You can use lazy fetching during normal traversal of the object tree, but in special cases — when the association is known to be needed at query time — you can set a query parameter that flags the association as eager. For example, the Employee example program might need to write a query that retrieves all employees who worked a certain number of hours during a pay period, in order to generate mailing labels. The query in Listing 8 can accomplish this:

Listing 8. Overriding the default fetching strategy

```
Criteria addressCrit = empCrit.createCriteria("payChecks")
    .add(Restrictions.eq("hoursWorked", hoursParam));
empCrit.setFetchMode("address",FetchMode.JOIN);
return empCrit.list();
```

The query in Listing 8 starts out by building a normal criteria query, but the interesting part is the call to `setFetchMode`. This code overrides the default lazy fetching strategy for the employee's address association. `FetchMode.JOIN` retrieves the `Address` instances as part of the search query, using joins to pull all of the data back in a single query. In many cases, this is exactly what you want and will outperform any other strategy.

However, `FetchMode.JOIN` does have some caveats. When used with collection-based associations, the query can return a different number of rows than would be returned in the default fetching strategy. The reason is that a single returned parent object can have multiple returned child objects. Hibernate seamlessly hides all of this at run time, properly parsing the result set and returning the proper list of objects. But this behavior can be an issue if `setFirstResult` or `setMaxResults` is also used in the query. Normally Hibernate uses a database-specific SQL statement to implement these features, but because the raw SQL query is returning the incorrect number of rows, database-specific techniques cannot work. Instead the entire data set is pulled from the database, and Hibernate manually extracts the window of data it needs to fulfill the request. As a result, a simple performance tweak intended to fix the *n+1 Selects Problem* ends up causing thousands of unused rows to be pulled into the application layer.

Hibernate also provides a second fetch mode setting you can use in queries. `FetchMode.SELECT` overrides eager evaluation on an association and instead uses lazy evaluation. But there's no way at query time to override an association to use the default eager evaluation techniques (second but immediate query).

Batching lazy loads

Another solution to the *n+1 Selects Problem* is a hybrid approach of lazy and eager fetching through the batching of lazy load requests. This approach still lazily loads the data, but instead of lazily loading one association at a time, it loads multiple associations. Many of the ORM frameworks have implementations of this basic idea. Hibernate calls this setting `BatchSize`, and TopLink calls it *batch reading*.

Using the preceding example, suppose a query retrieves ten `Employee` instances. The `Address` instance associated with the `Employee` is lazy loaded and has a batch size of five. One query would be used to retrieve the `Employee` instances, and no query would be issued for `Address` instances.

Later when one of the `Address` instances is accessed, Hibernate retrieves that `Address` and the next four that are marked to be lazy loaded. Assuming all of the `Address` instances associated to the `Employee` instances are accessed, this results in only two queries to retrieve all of the `Address` instances, rather than ten.

To use this strategy, the `Address` class needs the `@BatchSize` annotation, as shown in Listing 9:

Listing 9. The `@BatchSize` annotation

```
@Entity
@Table(name="ADDRESS")
@BatchSize(size=5)
public class Address extends AuditableEntity {//...}
```

Note in Listing 9 that the `Address` is being annotated, not the `Employee` class. Once the annotation is in place, `Address` instances automatically get pulled five at a time (or less if fewer than five records are available for lazy loading). You can also batch load collections of entities by annotating the collection with `@BatchSize`.

Conclusion

This series has outlined many approaches to common problems within the persistence tier. Through solutions as simple as refactoring primary keys into a base class and changing fetching strategies in the model, adapting Hibernate and a domain model to the task at hand makes for a more maintainable application. A more expressive and reusable domain model can emerge when you transfer common object-oriented concepts such as inheritance and polymorphism to the database via a framework like Hibernate. We hope many of the best practices we've outlined here can be molded to fit into the context and domain in which you work.

Downloadable resources

Description	Name	Size
Sample code for this article	PatternsOfPersistenceCode.zip	16KB