Patterns of persistence, Part 1: Strategies and best practices for modern ORM tools

Apply object-oriented principles to a domain model using Hibernate features

Skill Level: Intermediate

Ryan Senior (ryan@sourceallies.com)
Consultant
Source Allies, Inc.

Travis Klotz (travisklotz@sourceallies.com)
Consultant
Source Allies, Inc.

Jim Majure (jim@sourceallies.com)
Consultant
Source Allies, Inc.

22 Apr 2008

Although many developers use object-relational mapping (ORM) tools for their applications' persistence tier, some are confused about how to use them and duplicate code unnecessarily. The authors' experience constructing many persistence tiers has given them a clear understanding of persistence patterns and best practices. The first part of this two-part article covers the basics of a consistent, concise domain model and persistence tier. Part 2 builds and expands on the concepts covered in this article.

Introduction

The past five to ten years have seen a fundamental shift in the way that developers represent persistent entities in enterprise applications. Early enterprise applications used database tables and foreign-key relationships between them to model entities. Applications were seen as a way to view and query the database's underlying model. The trend in recent years is to move away from modeling entities in the database toward modeling them in the application's object model. The database is

now perceived simply as a mechanism for storing the persistent information defined by the object structure. Overall, this shift from modeling in the database to the object model has many advantages, including:

- Tighter integration of the persistent entities with the operations performed on them
- Increased ability to create loosely coupled application components
- Richer relationships allowed by object orientation that aren't supported by relational databases
- Higher degrees of independence from a particular database platform

The major factor behind this transition is the advent of highly capable object-relational mapping (ORM) systems that allow access to persistent objects in a manner consistent with the idiomatic use of the target language. Tools such as Hibernate and TopLink have made it much easier to map an object model into a relational database schema.

Since the advent of these tools, their use has evolved. Initially, many developers approached ORM tools just as they approached database tables. The entities were mapped one-to-one with database tables. Variables for fields such as primary keys were duplicated from entity to entity. Because databases do not support behavior associated with entities, the domain model ended up having simple variables with getter and setter methods for them. Behavior for these entities would end up being pushed up to the service or the view layer.

Experience in using ORM tools in many projects reveals better approaches to these same problems. Just as business domains differ, so do their domain models, persistent or not. This article covers best practices that apply to many domain models across industries. The best practices we present here can lead to a more consistent, reusable, and maintainable domain model. We use Hibernate to demonstrate these best practices, but you can apply many of the concepts to other ORM tools.

This article is separated into two parts. This first part covers some basic concepts in:

- Implementing common functionality in the domain
- Reducing code duplication in the data access tier
- Consistently handling auditing of entity changes

Part 2 focuses in more depth on some of the concepts we discuss here and also covers performance tuning in the domain model.

From the ground up: Starting with the object model

Defining an object model to support persistent objects is the same process as

defining any object model. First, look for common elements that all objects will share. Two such elements are common in persistent information: a unique way to identify a persistent object (that spans application executions) and audit information about object instances. Figure 1 illustrates how these two concepts can be defined using interfaces and base classes:

Figure 1. Common interfaces and base classes

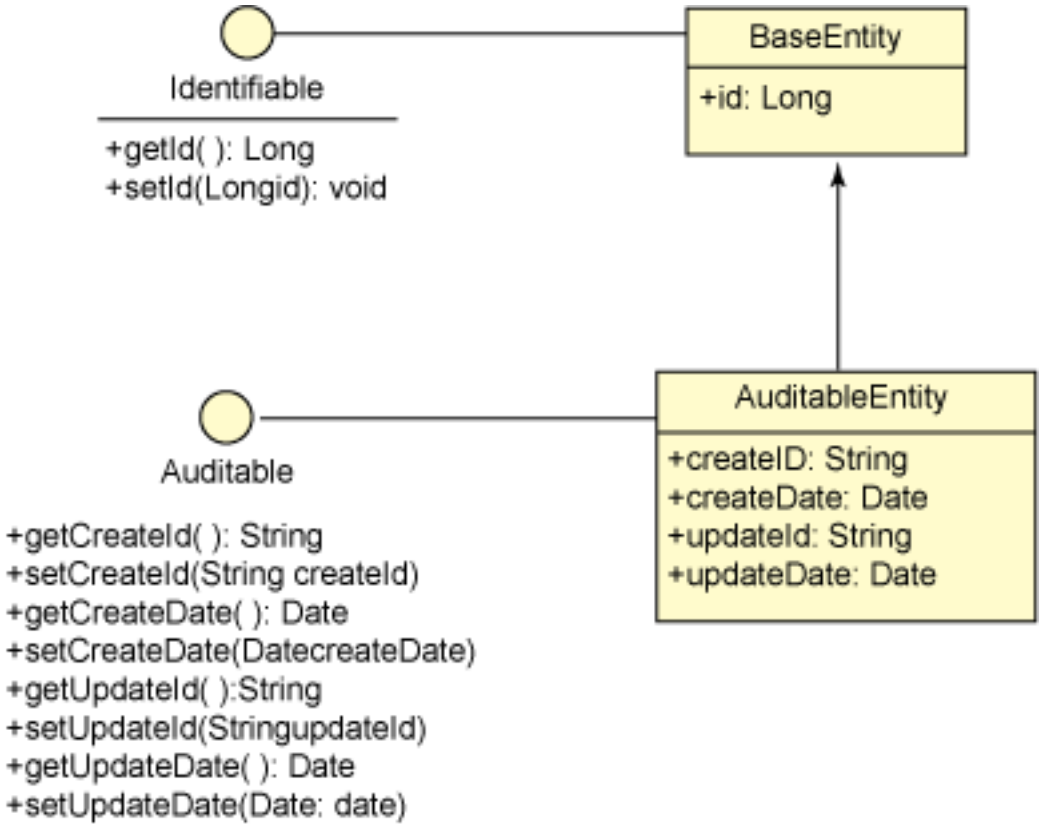


Figure 1 introduces the `Identifiable` and `Auditable` interfaces, which define the API for identifying object instances and setting audit information on object instances. It also introduces the `BaseEntity` and `AuditableEntity` base classes, from which concrete persistent classes can be extended, depending on whether or not auditing information is required for an object.

Defining persistent objects in terms of these interfaces provides a surprising number of opportunities to create abstract behavior that you can apply to all concrete object types. This includes the UI tier — for identifying objects on which create, read, update, and delete (CRUD) operations will take place — as well as the service and data tiers. This article's code examples (see Download to obtain the full package) show how to use these interfaces to facilitate auditing and reduce code duplication in data access objects (DAOs).

Common base entities

Database tables, unlike objects, do not have a concept of inheritance. Fields that are common to many tables, such as audit fields, must be redefined for each table. With this in mind, you can use inheritance in Java™ code to ensure this duplication doesn't spread to the code. Although ORM tools have supported this feature for a

long time, the Java Persistence API annotations have made it much easier, further reducing code duplication (see Resources).

Class-level annotations let you embed database mapping directly in class source code using Java 5 annotations. The Java Persistence API defines a suite of standard annotations for this purpose. Hibernate and other tools now provide support for these annotations. The `@MappedSuperclass` annotation lets you use mappings defined in base classes. As long as all database tables use the same column types and names for the common fields, the mappings can be written once in the base class and reused across all subclasses. Listing 1 is an example for the `BaseEntity` class:

Listing 1. Defining BaseEntity as a @MappedSuperclass

```
@MappedSuperclass
public class BaseEntity implements Identifiable{
    private Long id;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
}
```

The mapping in Listing 1 has the effect of mapping the ID field to the default column name (`id`) and specifying that the ID will be generated automatically (with the implementation being database-specific).

Note that it's possible to use these common base fields even if the column name differs for each table. A typical example: Even if the primary key is always a `Long` across all database tables, the column names might be different. By redefining the column that is assigned to an attribute for the specific subclass, you can still reuse the common code around the `id` attribute. Listing 2 illustrates this by redefining the column associated with the `id` attribute:

Listing 2. Redefining the column associated with the id attribute

```
@Entity
@AttributeOverride( name="id", column = @Column(name="EMPLOYEE_ID") )
public class Employee extends BaseEntity {//...}
```

You can also reuse code in these base classes if you are not using Hibernate annotations. However, you must map the fields for each concrete class. Hibernate then uses the inherited fields in the Java code automatically.

Data access core

The DAO pattern, as its name implies, encapsulates logic for accessing data in an object or set of related objects. DAOs in Hibernate contain `Criteria` queries and

Hibernate Query Language queries, and they have a Hibernate `SessionFactory`. The intent is that all database-oriented logic should be contained in the DAO, meaning that plain old Java objects (POJOs) and other primitive values should go in and come out of the DAO. The DAO is a fairly typical pattern in enterprise Java tiered architectures, and typically the DAO is accessed via a service. Close examination of DAOs reveals that their operations are often similar.

To find the commonality between DAOs, let's first look at some examples. Listing 3 shows two methods that query for an `Address` and an `Employee` based on the ID from the `Identifiable` interface defined in Figure 1:

Listing 3. Typical methods in Data Access Objects

```
public Address findById(Long id){
    return (Address) getSession().get(Address.class, id);
}

public Employee findById(Long id){
    return (Employee) getSession().get(Employee.class, id);
}
```

The major difference between these operations on `Employee` and `Address` is simply the classes used in the operations. The query is the same, and the result is just cast to a different class. Other similar operations such as deleting a given entity in the model or retrieving all of the instances of an entity from the model are common in a DAO and are also similar from entity to entity. With this in mind, it's possible to leverage Java 1.5's generics facilities to create a reusable DAO that can form the core of your data access tier.

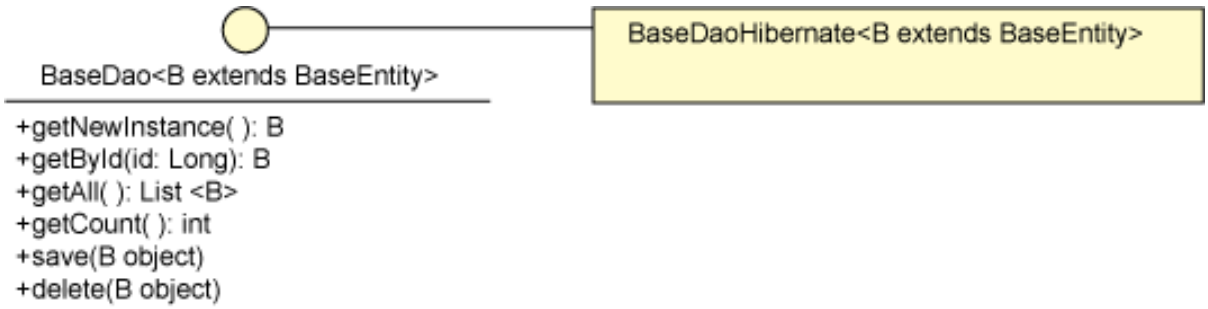
Generic DAO

The generic DAO pattern (also called Typesafe DAO) is critical to reducing code duplication in the data access tier. If you use Java 1.4, you can also benefit from common base DAO; the implementation won't be type safe or as clean, but it can still reduce code duplication.

Several approaches to the generic DAO are possible, many of which depend on your environment. This example uses a dependency-injection style, whereby the implementation does not worry about how to configure itself and instead assumes that everything it needs to function will be injected before it is used. Some other approaches don't use dependency injection (see the link in Resources to the Hibernate wiki for more information). The key to the dependency-injection approach is that you query data by injecting the `Class` of the entity that the DAO will be querying, along with defining the generic type.

The first step to creating the generic DAO is to define some of its common operations. Figure 2 presents a possible interface for a generic DAO:

Figure 2. Base DAO interface and implementation



Listing 4 shows some example code for the generic DAO:

Listing 4. Example code the for generic DAO pattern

```
public interface BaseDao<B extends BaseEntity> {
    B getById(Long id);
    // ...other methods
}

public class BaseDaoHibernate<B extends BaseEntity>
        implements BaseDao<B extends BaseEntity> {

    private Class<B> queryClass;

    public B getById(Long id) {
        return (B) getSession().get(getQueryClass(), id);
    }
    // ...other methods
}
```

The methods in Listing 4 form the core of the data access tier. This generic DAO can be used directly or subclassed depending on the needs of the entity that is being queried. The generic DAO could be used directly via a service, as shown in Listing 5:

Listing 5. Using the generic DAO via a service

```
BaseDao<Employee> dao = new BaseDao<Employee>();
dao.setQueryClass(Employee.class);
dao.setSessionFactory(sessionFactory);
...
dao.getById(-1L);
```

If a more complex set of data is required with more-customized queries, you can subclass the generic DAO. For example, suppose you want to find all full-time employees who live in Iowa. To do this, you may want to define an `Employee`-specific DAO method called `findIowaEmployees`. If you create a new `EmployeeDAO` that extends `BaseDAO`, the `EmployeeDAO` will get the specific full-time employees query along with all of the basic queries provided by the generic DAO, as shown in Listing 6:

Listing 6. Subclassing the generic DAO

```
public class EmployeeDao extends BaseDaoImpl<Employee> {
    public EmployeeDao() {
        setQueryClass(Employee.class);
    }
```

```
    List<Employee> findIowaEmployees() {
        Criteria crit = getCurrentSession().createCriteria(getQueryClass());
        crit.createCriteria("address").add(Restrictions.eq("state", "IA"));
        return crit.list();
    }
}
```

Note that Listing 6 uses the `createCriteria()` method. It is another method that's often duplicated from DAO to DAO in enterprise applications. As you use the generic DAO, you'll start to see new common operations that you can add to the generic DAO to increase reusability and decrease duplication. Other examples of common methods, such as enabling true pagination and dealing with search parameters, are detailed in Part 2.

Auditing

Basic auditing is a common feature in database-centric applications. Most applications want to keep track of when some or all objects are created and who created them, as well as when an object is modified and who modified it. Modeling these features is not difficult. Any object that requires auditing needs just four additional fields, one for each piece of auditing information. Figure 1 presents a base class for persistent entities that contains the necessary auditing fields. The more difficult part of this feature is figuring out where in the code to set the auditing data. For this you have a few choices:

- **Brute force**: You can simply make sure that audit information is populated with the correct values anywhere an object requiring audit information is modified. This has many obvious drawbacks. The most obvious is that the audit logic is duplicated throughout the application. Even if this logic is centralized into a utility class, you must still remember to use the utility everywhere the object is modified — not only when the system is initially developed, but also as it is maintained. In a system of any significant size, any developer will forget sooner or later.
- **Putting the logic in the DAO**: Another possibility is to centralize the logic in a common DAO. By adding the auditing logic to the save method, any object saved using the DAO will automatically have the auditing fields filled in. This works well in many situations, but it still leaves several gaps. One is that it assumes that the common DAO save method will always be used to save data. That may not always be the case, and then the issue of needing to remember to add the auditing logic surfaces again. The other, larger issue is that this solution ignores one of ORM tools' most useful features: *transitive persistence*. The DAO can be used to save an `Employee` object explicitly, but Hibernate also automatically takes care of persisting any changes to the `Address` associated with it. In this case, the `Employee` would have its auditing fields filled in, but the `Address` would not.
- **Hibernate `Interceptors`**: To solve this problem, an extension point

built into Hibernate is needed. Whenever the framework saves an object, there needs to be a place to populate these auditing fields. Thankfully, Hibernate provides this exact feature with its `Interceptor` interface. This interface provides callback methods for many different Hibernate events, including object creation, modification, and deletion. Building the auditing logic into a Hibernate `Interceptor` eliminates repeated logic, and you needn't worry about making sure the logic is executed. As long as Hibernate is in charge of saving the data, the auditing always occurs.

Implementation details

Hibernate has an `EmptyInterceptor` class that provides blank implementations of the ten-plus callback methods found in the `Interceptor` interface. This works perfectly for adding audit information. In the implementation shown in Listing 7, only two methods are relevant: `onSave`, which is called whenever a new object is flushed to the database, and `onFlushDirty`, which is called whenever Hibernate flushes an updated (or dirty) object to the database:

Listing 7. Extending EmptyInterceptor

```
public class AuditInterceptor extends EmptyInterceptor {

    @Override
    public boolean onSave(Object entity, Serializable id, Object[] currentState,
            String[] propertyNames, Type[] types) {
        ...
    }

    @Override
    public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState,
                Object[] previousState, String[] propertyNames, Type[] types) {
        ...
    }
}
```

Both of these events are called after Hibernate has determined which of the object's properties will be saved and what those properties' values are. These are passed into the methods as a `String` array of property names and an `Object` array of property values. Because Hibernate has already determined the values it's interested in, updating the object directly will not have the desired effect. In fact, updating that object will have no effect on the values that will ultimately be sent to the database. Instead, the elements in the property-value array need to be updated. Even with this somewhat awkward array approach, the implementation is still fairly simple and straightforward. Just iterate over the array of property names looking for the audit fields. When they're found, update the value array with the corresponding index. One final detail remains: these callback methods return a boolean value. If the object's state is modified, the method needs to return `true`. It should return `false` if no changes are made. The example code in Listing 8 shows this logic:

Listing 8. Interceptor callback methods updating audit fields

```
for(int i = 0; i < propertyNames.length; i++) {
    if(propertyNames[i].equals("createdOn")) {
        currentState[i] = new Date();
        updated = true;
    }
```

```
    if(propertyNames[i].equals("createdBy")) {
        currentState[i] = username;
        updated = true;
    }
}
```

The final step is a way to help enforce the names of the audit fields and to ensure that the database entities have those fields. The `Auditable` interface ends up being the easiest and best route for this purpose, but it can look a little strange. Although getter and setter methods are provided for these fields, they are never actually used in the auditing code. However, by having the entities implement the `Auditable` interface, you'll greatly reduce the number of typos created when you develop the persistent classes.

Conclusion to Part 1

This article has focused on applying basic object-oriented principles to a domain model using Hibernate features. As with all patterns and best practices, you should evaluate the solutions we've outlined here in the context you'll use them in, adding or removing pieces accordingly. The generic DAO is a great example of this flexibility. Different database structures, technologies, and business requirements have different common functionality that can be moved into the common code of a generic DAO. Different auditing information might be needed for different applications. (Obviously, auditing information for ATM machine domain entity transactions is different from auditing employee address information.) Regardless of the context, the best practices remain the same.

Part 2 of this series covers more best practices in constructing a data model by leveraging polymorphism with Hibernate, other useful features of a generic DAO, and performance tuning the data model.

Downloads

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

Information about download methods

Resources

Learn

- "Don't repeat the DAO!" (Per Mellqvist, developerWorks, May 2006): Build a generic typesafe DAO with Hibernate and Spring AOP.
- "Generic Data Access Objects" (Hibernate Community Wiki): Creating a generic DAO without Spring.
- Hibernate Reference Documentation: Hibernate from A to Z.
- Hibernate Annotations Reference Guide: Learn more about Hibernate annotations.
- "Design enterprise applications with the EJB 3.0 Java Persistence API" (Borys Burnayev, developerWorks, March 2006): An introduction to EJB 3.0's Java Persistence API.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Hibernate: Download Hibernate.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the authors

Ryan Senior
Ryan Senior, a consultant at Source Allies, Inc., has seven years of experience in the insurance, finance, manufacturing, and health-care domains. He specializes in designing and developing enterprise Java software.

Travis Klotz
Travis Klotz is a consultant with Source Allies, Inc. in Des Moines, IA. He has 10 years of experience developing software for the education, insurance, and engineering industries. He specializes in developing enterprise software using open source frameworks and helping organizations develop and deploy automated build management facilities.

Jim Majure
Jim Majure is a consultant with Source Allies, Inc. in Des Moines, IA. He has more than 20 years of experience developing software in the scientific, financial, and insurance domains. Currently, he specializes in designing and developing enterprise Java software.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.