# Don't repeat the DAO!

## Build a generic typesafe DAO with Hibernate and Spring AOP

Per Mellqvist (per@mellqvist.name)
System architect
自由职业者

May 12, 2006

With the adoption of Java™ 5 generics, the idea of a generic typesafe Data Access Object (DAO) implementation has become feasible. In this article, system architect Per Mellqvist presents a generic DAO implementation class based on Hibernate. He then shows you how to use Spring AOP introductions to add a typesafe interface to the class for query execution.

For most developers, writing almost the same code for every DAO in a system has by now become a habit. While anyone would identify the repetition as a "code smell," most of us have learned to live with it. And there are workarounds. You can use many ORM tools to avoid code repetition. With Hibernate, for example, you can simply use session operations directly for all of your persistent domain objects. The downside of this approach is the loss of typesafety.

Why would you want a typesafe interface for your data access code? I would argue that it reduces programming mistakes and increases productivity when used together with modern IDE tools. First of all, a typesafe interface clearly indicates which domain objects have persistent storage available. Secondly, it removes the need for error-prone type casts. (A problem more common with query operations than CRUDs.) Finally, it leverages the auto-completion features found in most IDEs today. Using auto completion is a fast way to remember what queries are available for a certain domain class.

In this article, I show you how to avoid repeating DAO code over and over again, while still retaining the benefits of a typesafe interface. In fact, all you need to write for each new DAO is a Hibernate mapping file, a plain old Java interface, and 10 lines in your Spring configuration file.

## The DAO implementation

The DAO pattern should be well known to any enterprise Java developer. Implementations of the pattern vary considerably, however, so let's clarify the assumptions behind the DAO implementation I present in this article:

- All database access in the system is made through a DAO to achieve encapsulation.

- Each DAO instance is responsible for one primary domain object or entity. If a domain object has an independent lifecycle, it should have its own DAO.
- The DAO is responsible for creations, reads (by primary key), updates, and deletions -- that is, CRUD -- on the domain object.
- The DAO may allow queries based on criteria other than the primary key. I refer to these as *finder methods* or *finders*. The return value of a finder is normally a collection of the domain object for which the DAO is responsible.
- The DAO is not responsible for handling transactions, sessions, or connections. These are handled outside the DAO to achieve flexibility.

## A generic DAO interface

The foundation for a generic DAO is its CRUD operations. The following interface defines the methods for a generic DAO:

### Listing 1. A generic DAO interface

```
public interface GenericDao <T, PK extends Serializable> {

    /** Persist the newInstance object into database */
    PK create(T newInstance);

    /** Retrieve an object that was previously persisted to the database using
     *   the indicated id as primary key
     */
    T read(PK id);

    /** Save changes made to a persistent object.  */
    void update(T transientObject);

    /** Remove an object from persistent storage in the database */
    void delete(T persistentObject);
}
```

### Implementing the interface

Implementing the interface in Listing 1 is trivial with Hibernate, as shown in Listing 2. It's just a matter of calling the underlying Hibernate methods and adding the casts. Spring takes care of the session and transaction management. (Of course, I assume that these functions have been properly set up, but that subject is well covered by the Hibernate and Spring framework manuals.)

### Listing 2. First generic DAO implementation

```
public class GenericDaoHibernateImpl <T, PK extends Serializable>
    implements GenericDao<T, PK>, FinderExecutor {
    private Class<T> type;

    public GenericDaoHibernateImpl(Class<T> type) {
        this.type = type;
    }

    public PK create(T o) {
        return (PK) getSession().save(o);
    }

    public T read(PK id) {
```

```
        return (T) getSession().get(type, id);
    }

    public void update(T o) {
        getSession().update(o);
    }

    public void delete(T o) {
        getSession().delete(o);
    }

    // Not showing implementations of getSession() and setSessionFactory()
            }
```

### The Spring configuration

Finally, in the Spring configuration, I create an instance of `GenericDaoHibernateImpl`. The constructor of `GenericDaoHibernateImpl` must be told which domain class the DAO instance will be responsible for. This is necessary for Hibernate to have runtime knowledge of the type of objects managed by the DAO. In Listing 3, I pass the domain class `Person` from the sample application to the constructor and set a previously configured Hibernate session factory as a parameter to the instantiated DAO:

### Listing 3. Configuring the DAO

```
<bean id="personDao" class="genericdao.impl.GenericDaoHibernateImpl">
        <constructor-arg>
            <value>genericdaotest.domain.Person</value>
        </constructor-arg>
        <property name="sessionFactory">
            <ref bean="sessionFactory"/>
        </property>
</bean>
```

## A usable generic DAO

I'm not done yet, but what I have is certainly usable already. In Listing 4, you can see an example of using the generic DAO as it is:

### Listing 4. Using the DAO

```
public void someMethodCreatingAPerson() {
    ...
    GenericDao dao = (GenericDao)
     beanFactory.getBean("personDao"); // This should normally be injected

    Person p = new Person("Per", 90);
    dao.create(p);
}
```

At this point, I have a generic DAO capable of typesafe CRUD operations. It would be perfectly reasonable to subclass `GenericDaoHibernateImpl` to add query capabilities for each domain object. Because the purpose of this article is to display how you can achieve this *without* writing explicit Java code for each query, however, I'll use two more tools to introduce queries to the DAO, namely Spring AOP and Hibernate named queries.

## Spring AOP introductions

You can use introductions in Spring AOP to add functionality to an existing object by wrapping it in a proxy, defining what new interface it should implement, and delegating all previously unsupported methods to a single handler. In my DAO implementation, I use introductions to add a number of finder methods to my existing generic DAO class. Because the finder methods are specific to each domain object, they are applied to the typed interfaces of the generic DAO.

The Spring configuration for this is shown in Listing 5:

### Listing 5. Spring configuration for FinderIntroductionAdvisor

```
<bean id="finderIntroductionAdvisor" class="genericdao.impl.FinderIntroductionAdvisor"/>

<bean id="abstractDaoTarget"
        class="genericdao.impl.GenericDaoHibernateImpl" abstract="true">
        <property name="sessionFactory">
            <ref bean="sessionFactory"/>
        </property>
</bean>

<bean id="abstractDao"
        class="org.springframework.aop.framework.ProxyFactoryBean" abstract="true">
        <property name="interceptorNames">
            <list>
                <value>finderIntroductionAdvisor</value>
            </list>
        </property>
</bean>
```

In the configuration file in Listing 5, I define three Spring beans. The first bean, FinderIntroductionAdvisor, handles all the methods introduced to the DAO that are not available in the `GenericDaoHibernateImpl` class. I explain the Advisor bean in detail in a moment.

The second bean is "abstract." In Spring, this means that the bean can be reused in other bean definitions but it is not instantiated. Other than the abstract property, the bean definition simply points out that I want an instance of `GenericDaoHibernateImpl` and that it needs a reference to a `SessionFactory`. Note that the `GenericDaoHibernateImpl` class defines only one constructor that takes a domain class as its argument. Because this bean definition is abstract, I can reuse the definition a number of times later and set the constructor argument to a suitable domain class.

Finally, the third and most interesting bean wraps the vanilla instance of `GenericDaoHibernateImpl` in a proxy, giving it the ability to execute finder methods. This bean definition is also abstract and does not specify the interface that I would like to introduce to my vanilla DAO. The interface is different for each concrete instance. Note that the entire configuration shown in Listing 5 is done just once.

## Extending GenericDAO

The interface for each DAO is, of course, based on the `GenericDao` interface. I just need to adapt the interface to a specific domain class and extend it to include my finder methods. In Listing 6, you can see an example of the `GenericDao` interface extended for a specific purpose:

### Listing 6. The PersonDao interface

```
public interface PersonDao extends GenericDao<Person, Long> {
    List<Person> findByName(String name);
}
```

Clearly the purpose of the method defined in Listing 6 is to look up a `Person` by name. The necessary Java implementation code is all generic code that does not require any updates when adding more DAOs.

### Configuring PersonDao

Because the Spring configuration relies on the "abstract" beans defined earlier, it becomes reasonably compact. I need to point out which domain class my DAO is responsible for, and I need to tell Spring which interface the DAO should implement (some methods directly and some by using introductions). Listing 7 shows the Spring configuration file for `PersonDAO`:

### Listing 7. Spring config for PersonDao

```
<bean id="personDao" parent="abstractDao">
    <property name="proxyInterfaces">
        <value>genericdaotest.dao.PersonDao</value>
    </property>
    <property name="target">
        <bean parent="abstractDaoTarget">
            <constructor-arg>
                <value>genericdaotest.domain.Person</value>
            </constructor-arg>
        </bean>
    </property>
</bean>
```

In Listing 8, you can see this updated version of the DAO in use:

### Listing 8. Using the typesafe interface

```
public void someMethodCreatingAPerson() {
    ...
    PersonDao dao = (PersonDao)
     beanFactory.getBean("personDao"); // This should normally be injected

    Person p = new Person("Per", 90);
    dao.create(p);

    List<Person> result = dao.findByName("Per"); // Runtime exception
}
```

While the code in Listing 8 is a correct way of using the typesafe `PersonDao` interface, the implementation of the DAO is not complete. Calling `findByName()` causes a runtime exception. The problem is that I have not yet implemented the query necessary to call `findByName()`. All that remains is to specify the query. To correct this, I use a Hibernate named query.

## Hibernate named queries

With Hibernate, you can define an HQL query in a Hibernate mapping file (hbm.xml) and give it a name. You can use this query later in your Java code by simply referring to the given name. One of

the benefits of this approach is the ability to tune queries at deployment time without making code changes. As you will soon see, another benefit is the potential to implement a "complete" DAO without writing any new Java implementation code. Listing 9 is an example of a mapping file with a named query:

### Listing 9. A Hibernate mapping file with a named query

```
<hibernate-mapping package="genericdaotest.domain">
    <class name="Person">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name" />
        <property name="weight" />
    </class>

    <query name="Person.findByName">
        <![CDATA[select p from Person p where p.name = ? ]]>
    </query>
</hibernate-mapping>
```

Listing 9 defines the Hibernate mapping of a domain class, `Person`, with two properties: `name` and `weight`. `Person` is a simple POJO with the mentioned properties. The file also contains a query that finds all instances of `Person` in the database, where the "name" equals the provided parameter. Hibernate provides no real namespace functionality for the named queries. For the purposes of this discussion, I prefix all query names with the short (unqualified) name of the domain class. In a real-world situation, using the full class name, including package name, is probably a better idea.

## A step-by-step overview

You've seen all the steps necessary to create and configure a new DAO for any domain object. The three simple steps are:

1. Define an interface that extends `GenericDao` and contains any finder methods you need.
2. Add a named query for each finder method to the hbm.xml mapping file for the domain object.
3. Add the 10-line Spring configuration file for the DAO.

I conclude my discussion with a look at the code (written just once!) that executes my finder methods.

## A reusable DAO class

The Spring advisor and interceptor used are trivial and their job is in fact to refer back to the `GenericDaoHibernateImplClass`. All invocations where the method name starts with "find" are passed on to the DAO and the single method `executeFinder()`.

### Listing 10. Implementation of FinderIntroductionAdvisor

```
public class FinderIntroductionAdvisor extends DefaultIntroductionAdvisor {
    public FinderIntroductionAdvisor() {
        super(new FinderIntroductionInterceptor());
```

```
    }
}

public class FinderIntroductionInterceptor implements IntroductionInterceptor {

    public Object invoke(MethodInvocation methodInvocation) throws Throwable {

        FinderExecutor genericDao = (FinderExecutor) methodInvocation.getThis();

        String methodName = methodInvocation.getMethod().getName();
        if (methodName.startsWith("find")) {
            Object[] arguments = methodInvocation.getArguments();
            return genericDao.executeFinder(methodInvocation.getMethod(), arguments);
        } else {
            return methodInvocation.proceed();
        }
    }

    public boolean implementsInterface(Class intf) {
        return intf.isInterface() && FinderExecutor.class.isAssignableFrom(intf);
    }
}
```

### The executeFinder() method

The only thing missing from the implementation in Listing 10 is the `executeFinder()` implementation. This code looks at the name of the invoked class and method and matches these to the name of a Hibernate query using convention over configuration. You could also use a `FinderNamingStrategy` to enable other ways of naming queries. The default implementation looks for a query called "`ClassName.methodName`" where `ClassName` is the short name without packages. Listing 11 completes my generic typesafe DAO implementation:

### Listing 11. Implementation of executeFinder()

```
public List<T> executeFinder(Method method, final Object[] queryArgs) {
     final String queryName = queryNameFromMethod(method);
     final Query namedQuery = getSession().getNamedQuery(queryName);
     String[] namedParameters = namedQuery.getNamedParameters();
     for(int i = 0; i < queryArgs.length; i++) {
             Object arg = queryArgs[i];
             Type argType =  namedQuery.setParameter(i, arg);
      }
      return (List<T>) namedQuery.list();
 }

 public String queryNameFromMethod(Method finderMethod) {
     return type.getSimpleName() + "." + finderMethod.getName();
 }
```

## In conclusion

Prior to Java 5, the language did not support writing code that was both typesafe *and* generic; you had to choose one or the other. In this article, you've seen just one example of using Java 5 generics in combination with tools such as Spring and Hibernate (and AOP) to improve productivity. A generic typesafe DAO class is relatively easy to write -- all you need is a single interface, some named queries, and a 10-line addition to your Spring configuration -- and can greatly reduce errors, as well as save time.

Almost all the code in this article is reusable. While your DAO classes may contain types of queries or operations not implemented here (bulk operations, perhaps), you should be able to implement at least some of them using the techniques I've demonstrated.

## Acknowledgments

The concept of a single generic typesafe DAO had been a topic since the appearance of generics in the Java language. I briefly discussed the feasibility of a generic DAO with Don Smith at JavaOne 2004. The DAO implementation class used in this article is intended as an example implementation; other implementations do exist. For example, Christian Bauer has published an implementation with CRUD operations and criteria searches, and Eric Burke has also done work in this area. I'm sure many more implementations will surface. I owe additional credit to Christian for looking at my first attempt at writing a generic typesafe DAO and suggesting improvements. Finally, I thank Ramnivas Laddad for his invaluable help in reviewing this article.

Related topics Advanced DAO programming Introduction to generic types in JDK 5.0 Java theory and practice: Generics gotchas

# Downloads

| Description | Name | Size |
| --- | --- | --- |
| Full source code | j-genericdao.zip | 12KB |

## About the author

**Per Mellqvist**

Per Mellqvist is a system architect based in Stockholm, Sweden. He enjoys all aspects of enterprise Java, and appreciates any framework or tool that does a single thing well.