

DAOMagikk with Proxies

2008-09-15 19:07:12

This article describes some Proxy-Techniques to automatically generate reusable DAOs only by defining their interfaces.

Proxies

Java Proxies are well known, especially if you work with APIs like Hibernate or Spring. Usually proxies are used to extend the functionality of applications without including non-business logic code into the business logic code. For example logging can be handled by proxies, which log every method invocation of some target object.

Hibernate extensively uses proxies around domain objects to trace their state (if they are dirty or not). In Spring proxies can be used to handle transactions over some business methods, or cache method values or anything else. [Mock Objects](#) use proxies, to implement interfaces and return pre-defined objects for testcases.

In most cases proxies are used to add functionality to existing code. So as in thousands of examples, some logging proxy would look like [in this example](#):

```
package org.apache.examples.impl; import java.lang.reflect.InvocationHandler; import
java.lang.reflect.InvocationTargetException; import java.lang.reflect.Method; import
org.apache.commons.logging.Log; import org.apache.hivemind.service.impl.LoggingUtils; public class
ProxyLoggingInvocationHandler implements InvocationHandler{ private Log _log; private Object
_delegate; public ProxyLoggingInvocationHandler(Log log, Object delegate){ _log = log;
_delegate = delegate; } public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable{ boolean debug = _log.isDebugEnabled(); if (debug)
LoggingUtils.entry(_log, method.getName(), args); try{ Object result =
method.invoke(_delegate, args); if (debug){ if (method.getReturnType() == void.class)
LoggingUtils.exit(_log, method.getName()); else
LoggingUtils.exit(_log, method.getName(), result); } return result; } catch
(InvocationTargetException ex){ Throwable targetException = ex.getTargetException(); if
(debug) LoggingUtils.exception(_log, method.getName(), targetException); throw
targetException; } } }
```

As seen in the example code, there is a delegate object, that is being proxied (also called “target”). Every call to the proxy results in the invocation of the “invoke”-method. This method first does the proxy-specific work, here logging, and then delegates the method call to the main object, the target.

But the delegation to a target object is not a must.

DAOs

Some times ago I tried to write some generic DAO-Code (especially for Hibernate) that I could easily reuse in several projects.

It looks somehow like [the one I posted earlier in my blog](#).

Then I met Grails, with their mind blowing metaprogramming concept, that handles, with the help of the GORM API, method to Hibernate criteria translation.

The GORM approach

As on the examplepage for [GORM in Grails](#), you can query your domain objects with methods like:

```
results = Book.findByTitleLike("Harry Pot%")
results = Book.findByReleaseDateBetween( firstDate, secondDate )
results = Book.findByReleaseDateGreaterThan( someDate )
results = Book.findByTitleLikeOrReleaseDateLessThan( "%Something%", someDate )
```

In the background Grails analyses (or better: *parses*) the method name and generates a query that fits to the method name. You don't have to implement them.

Never the less, in Java, you can't do this, since you have to provide at least an Interface, to be able to call some method like "findByTitleLike".

To Java

But do you have to implement that methods? At this place, Proxies come into the game.

I wrote a method handler, that doesn't need to delegate to any other class but analyse the method name and try to create a appropriate action. So the "daobeauty"-API came to light.

In my API (DaoBeauty), you provide a interface for your DAO. Of course, the Proxy needs much more information than the one from the logging example. For example the proxy needs a Hibernate SessionFactory to be able to work with hibernate. As well as the domain class this DAO is made for.

As a result, lots of mehtods you can define in your DAO interface will be handled automatically. Here some examples (sourcecode available in the distribution jar).

Let's create some domainclass called D1:

```
package daobeauty.test.domain; public class D1 { private Long id; private String name; private String
subject; private String firstAndLastName; /** * @return the firstAndLastName */ public String
getFirstAndLastName() { return firstAndLastName; } /** * @param firstAndLastName the
firstAndLastName to set */ public void setFirstAndLastName(String firstAndLastName) {
this.firstAndLastName = firstAndLastName; } /** * @return the subject */ public String getSubject() {
return subject; } /** * @param subject the subject to set */ public void setSubject(String subject) {
this.subject = subject; } /** * @return the id */ public Long getId() { return id; } /** * @param id the
id to set */ public void setId(Long id) { this.id = id; } /** * @return the name */ public String
getName() { return name; } /** * @param name the name to set */ public void setName(String name)
{ this.name = name; } /* (non-Javadoc) * @see java.lang.Object#hashCode() */ @Override public
int hashCode() { final int PRIME = 31; int result = 1; result = PRIME * result + ((id == null) ? 0 :
id.hashCode()); return result; } /* (non-Javadoc) * @see java.lang.Object#equals(java.lang.Object) */
@Override public boolean equals(Object obj) { if (this == obj) return true; if (obj == null) return false;
if (getClass() != obj.getClass()) return false; final D1 other = (D1) obj; if (id == null) { if (other.id != null)
return false; } else if (!id.equals(other.id)) return false; return true; } public String toString(){ return
```

```
id+&quot; &quot;+name+&quot; &quot;+subject; } }
```

There is nothing really special with this domain object. Just Eclipse generated setters, getters, hashCode and equals.

A corresponding hibernate mapping file ensures, that it's mapped to a database:

```
<?xml version=&quot;1.0&quot;?> <!DOCTYPE hibernate-mapping PUBLIC
&quot;://Hibernate/Hibernate Mapping DTD 3.0//EN&quot;
&quot;http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd&quot;> <hibernate-mapping
default-access=&quot;property&quot; default-lazy=&quot;false&quot; auto-import=&quot;true&quot;>
<class name=&quot;daobeauty.test.domain.D1&quot; table=&quot;d1&quot;> <id name=&quot;id&quot;
column=&quot;id&quot;> <generator class=&quot;native&quot;/> </id> <property
name=&quot;name&quot; column=&quot;name&quot; not-null=&quot;true&quot; unique=&quot;true&quot;
length=&quot;20&quot;/> <property name=&quot;subject&quot; column=&quot;subject&quot;
not-null=&quot;false&quot; length=&quot;50&quot;/> <property name=&quot;firstAndLastName&quot;
column=&quot;frstlastname&quot; not-null=&quot;false&quot; length=&quot;50&quot;/> <query
name=&quot;findAll&quot;>from D1 d</query> </class> </hibernate-mapping>
```

Now we will need some DAO that will handle the DB stuff. Here we just define our interface:

```
package daobeauty.test.dao; import java.util.List; import daobeauty.test.domain.D1; public interface
D1Dao{ public void saveOrUpdate(D1 d1); public void save(D1 d1); public void delete(D1 d1); public
void update(D1 d1); public List<D1> findAll(); public List<D1> findByExample(D1 example); public D1
findById(Long id); public D1 findByPrimaryKey(Long id); public List<D1> findByNameAndSubject(String
name, String subject); public List<D1> findByNameLikeAndSubjectIsNull(String name); public List<D1>
findByFirstAndLastName(String name); }
```

As seen, the methodnames are quite like the ones in Grails/GORM. And now, let's implement the whole DAO:

```
// get some sessionFactory final sessionFactory sessionFactory =
HibernateSessionFactoryUtil.getSessionFactory(); final ProxyFactory proxyFactory = new ProxyFactory();
// creation final D1Dao d1dao = (D1Dao) proxyFactory.generateProxy(new Class[]{D1Dao.class},
sessionFactory, D1.class, null, null);
That's it. Really.
```

For a simple test, we can handle the transaction and session stuff manually. DaoBeauty doesn't take care of transactions, so you can define them manually, or use some proxy within Spring.

```
Session session = null; Transaction transaction = null; try{ // preparation final sessionFactory
sessionFactory = HibernateSessionFactoryUtil.getSessionFactory(); session =
sessionFactory.getCurrentSession(); transaction = session.beginTransaction(); final ProxyFactory
proxyFactory = new ProxyFactory(); // creation final D1Dao d1dao = (D1Dao)
proxyFactory.generateProxy(new Class[]{D1Dao.class}, sessionFactory, D1.class, null, null); D1 test =
new D1(); test.setName(&quot;a&quot;); d1dao.save(test); test = new D1();
test.setName(&quot;b&quot;); d1dao.save(test); List<D1> result = d1dao.findAll(); for (D1 d:result){
System.out.println(d.toString()); } final D1 exampleD1 = new D1();
exampleD1.setName(&quot;a&quot;); result = d1dao.findByExample(exampleD1); for (D1 d:result){
System.out.println(d.toString()); } System.out.println(&quot;by primarykey&quot;); final D1 byPk =
```

```
d1dao.findByPrimaryKey(new Long(2)); System.out.println(byPk.toString()); result =
d1dao.findByNameAndSubject("a", null); if (result.size()<1){ System.out.println("empty
result"); } else{ for (D1 d:result){ System.out.println(d.toString()); } }
```

All works like charming. The ProxyFactory of DaoBeauty is intended to encapsulate different Proxy APIs, like Reflection, Javassist or CGLIB. The last both are needed in a future version, when adding generation of proxy Daos for existing Dao implementations or abstract base classes that already implement some of the Dao-interface methods.

Inside DaoBeauty

Internally the Proxy has a 5-step strategy to create queries and return results.

1. The called method is searched on a given target object (can be supplied to the proxy-factory). When the method is found here, it is executed as a delegate.
2. The called method is searched on any of the supplied delegate classes. This proxy allows you to provide any amount of delegate classes, that have a method with same name and signature as in the DAO interface, without implementing ALL of them. An example of this will follow up.
3. If the called method is some CRUD method (save, update, delete, saveOrUpdate), it is executed by a implementation within the proxy. The CRUD operations can have a single domain object as parameter or a whole collection (in which case the CRUD operation is applied to all elements of the collection). It's also planned to support arrays of objects instead of collections.
4. If there is a named query with the same name as the method, the named query is used. In the above example, the method "findAll" is executed by running the "findAll" named query of the D1-hibernate mapping. If some parameters are provided here, they are set as parameters to the query.
5. If the methodname starts with "findBy", the method name is parsed and converted to some hibernate criteria. At this point, also invocations for methods "findByExample(Object example)", "Criteria createCriteria()", "findAll(String hql)", "findAll(String hql, Object[] paramters)", "findAll(String hql, Map namedParameters)", "findByPrimaryKey(Object pk)" and "findById(Object pk)" are handled.

In all cases the method interceptor analyses the returntype of the method as defined in the DAO interface and invokes the "list()" for java.util.Collection or "setMaxResults(1).uniqueResult()" method for single objects of the appropriate query.

Delegates

For more complex queries, that can't be handled by the standard "findBy"-Syntax, some classes that implement the appropriate method can be provided (the delegates).

For example, we want to determine the highest used id in the domain table. Our delegate looks like this:

```
package daobeauty.test.dao; import org.hibernate.Session; import org.hibernate.SessionFactory; import
org.hibernate.criterion.Order; import daobeauty.dao.SessionFactoryAware; import
daobeauty.test.domain.D1; public class D1Delegate implements SessionFactoryAware { private
SessionFactory sessionFactory; public Session getCurrentSession() { return
sessionFactory.getCurrentSession(); } /* (non-Javadoc) * @see
daobeauty.dao.hibernate.SessionFactoryAware#getSessionFactory() */ public SessionFactory
```

```
getSessionFactory() { return sessionFactory; } /* (non-Javadoc) * @see
daobeauty.dao.hibernate.SessionFactoryAware#setSessionFactory(org.hibernate.SessionFactory) */
public void setSessionFactory(SessionFactory sessionFactory) { this.sessionFactory = sessionFactory; }
    public D1 findByIdHighestId(){ return (D1) getCurrentSession().createCriteria(D1.class).
addOrder(Order.desc("id")).setMaxResults(1).uniqueResult(); } }
This delegate is given to the proxy when creating our extended DAO:
```

```
public interface D1Dao{ ... public D1 findByIdHighestId(); ... }
```

```
final D1Delegate delegate = new D1Delegate(); final ProxyFactory proxyFactory = new ProxyFactory(); //
creation final D1Dao d1dao = (D1Dao) proxyFactory.generateProxy(new Class[]{D1Dao.class},
sessionFactory, D1.class, null, new Object[]{delegate}); D1 highestId = d1dao.findByIdHighestId();
The interface SessionFactoryAware is used to tell the proxy, that the delegate needs access to a
SessionFactory. The proxy will ensure, that his SessionFactory will be set to the delegate. This is not a
must. You can give the delegates a SessionFactory as you like.
```

TODO

At this point, the DaoBeauty-API is just a proof of concept and still misses many functionality and is not well tested. The parser for the methodnames is not 100% safe and tested for now (just a simple descending parser). Also some support for sorting/ordering/offset/maxresults is missing for now. The API parses findBy-methods always, so there is no caching to speed up. And last but not least: There is a proxy-factory for spring-integration in work, that is not ready for now.

[Download proof of concept with dependencies and sourcecode](#)