



Framework Corporatiu J2EE

Servei de Persistència

Versió 1.1

Barcelona, 3 / octubre / 2006



Històric de modificacions

Data	Autor	Comentaris	Versió
09/12/2005	Atos Origin, sae openTrends	Versió inicial del document	1.0
02/10/2006	Atos Origin	Versió 1.1 openFrame	1.1

Llegenda de Marcadors



Índex

1.	INTRODUCCIÓ	4
1.1.	PROPÒSIT	4
1.2.	CONTEXT I ESCENARIS D'ÚS	4
1.3.	VERSIONS I DEPENDÈNCIES	4
1.3.1.	<i>Versions</i>	4
1.3.2.	<i>Dependències Bàsiques</i>	5
1.4.	A QUI VA DIRIGIT	5
1.5.	DOCUMENTS I FONTS DE REFERÈNCIA	5
1.6.	GLOSSARI	5
2.	DESCRIPCIÓ DETALLADA	7
2.1.	ARQUITECTURA I COMPONENTS	7
2.1.1.	<i>Interfícies i components genèrics</i>	7
2.1.2.	<i>Implementació de les interfícies</i>	8
2.2.	INSTAL·LACIÓ I CONFIGURACIÓ	10
2.2.1.	<i>Instal·lació</i>	10
2.2.2.	<i>Configuració</i>	10
2.3.	UTILITZACIÓ DEL SERVEI	16
2.3.1.	<i>Actors</i>	16
2.3.2.	<i>Transaccionalitat</i>	17
2.4.	EINES DE SUPORT	19
2.4.1.	<i>Hibernate Synchronizer</i>	19
2.4.2.	<i>Hibernate Tools</i>	19
2.5.	INTEGRACIÓ AMB ALTRES SERVEIS	19
2.6.	PREGUNTES FREQUÈNTS	19
3.	EXEMPLES	20
3.1.	TEST UNITARIS	20
4.	ANNEXOS	28

1. Introducció

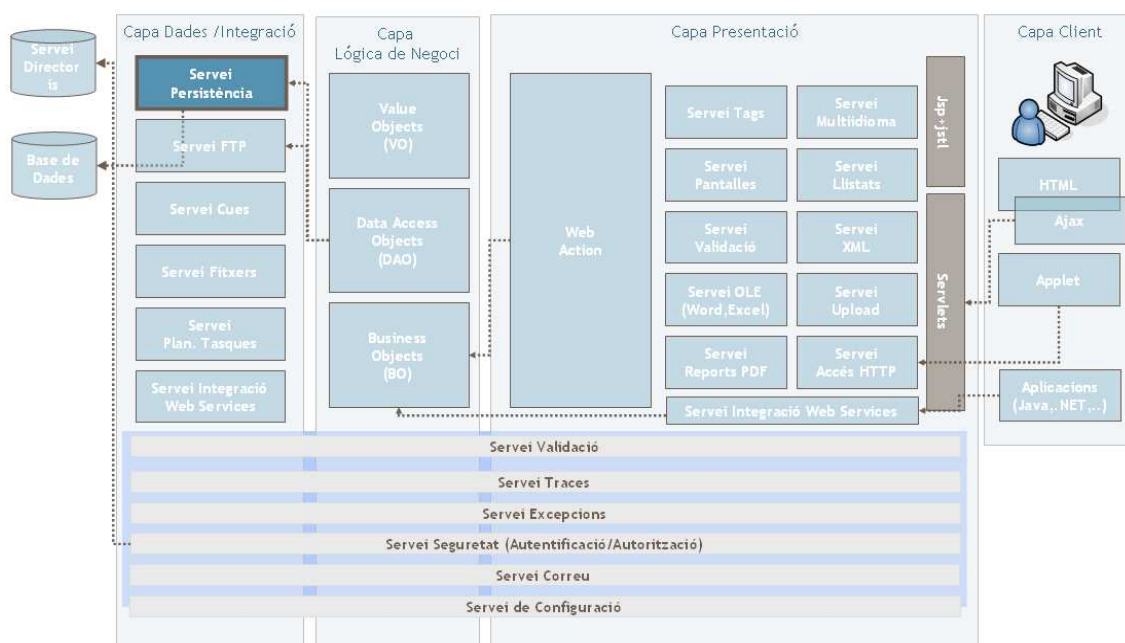
1.1. Propòsit

El Servei de Persistència permet persistir i recuperar dades entre l'aplicatiu i el motor de base de dades. La base tecnològica està basada en Spring i Hibernate, i entre d'altres característiques ofereix:

1. Suport per a DAO (Data Access Object) orientat principalment a facilitar la feina amb les tecnologies d'accés a dades a dades.
2. Traducció d'excepcions específiques de la tecnologia utilitzada a una jerarquia d'excepcions genèrica
3. Transaccionalitat declarativa

1.2. Context i Escenaris d'Ús

El Servei de Persistència es troba dins dels serveis de Propòsit General de openFrame.



1.3. Versions i Dependències

1.3.1. Versions

En la versió 1.1 es potencia l'ús del DAOUniversal, es deixa de donar suport amb HibernateSynchronizer, i les transaccions s'ha de configurar a nivell de BO.



1.3.2. Dependències Bàsiques

Nom	Tipus	Versió	Descripció
log4j	jar	1.2.12	http://logging.apache.org/log4j
commons-logging	jar	1.0.4	http://jakarta.apache.org/commons
spring	jar	1.2.5	http://www.springframework.org
hibernate	jar	3.0	http://www.hibernate.org/
openframe-services-logging	jar	1.1	
openframe-services-exceptions	jar	1.1	

Cal destacar que dintre de la dependència "Hibernate" s'inclouen totes les llibreries necessàries per al funcionament correcte d'aquesta capa de persistència. Per a veure quines són es pot consultar la web <http://www.hibernate.org>

1.4. A qui va dirigit

Aquest document va dirigit als següents perfils:

- Programador. Per conèixer l'ús del servei
- Arquitecte. Per conèixer quins són els components i la configuració del servei
- Administrador. Per conèixer com configurar el servei en cadascun dels entorns en cas de necessitat

1.5. Documents i Fonts de Referència

- [1] Hibernate <http://www.hibernate.org>

1.6. Glossari

DAO

Data Access Object, permet persistir les dades d'un objecte Java (normalment un POJO) a una base de dades

POJO

Plain Old Java Object, nom que se li dóna a les classes Java normals que no són de cap tipus especial (EJB's, etc) i que segueixen la convenció JavaBean.

BO

Business Object, objecte que implementa la lògica de negoci.



VO

Value Object, objecte que fa de transport de les dades necessàries per a la lògica de negoci.

2. Descripció Detallada

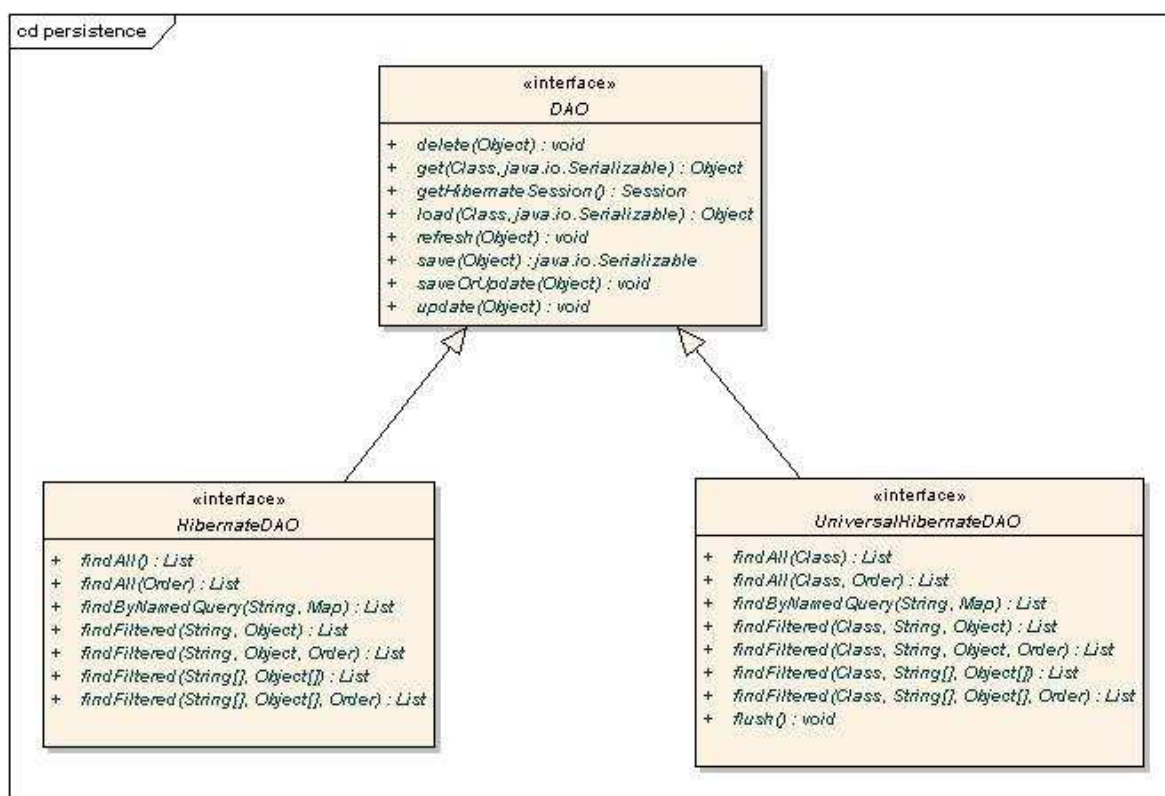
2.1. Arquitectura i Components

Existeixen tres tipus de components. Podem classificar-los en:

- Interfícies i components genèrics.
- Implementació de les interfícies.

2.1.1. Interfícies i components genèrics

El servei de llistats defineix les següents interfícies:



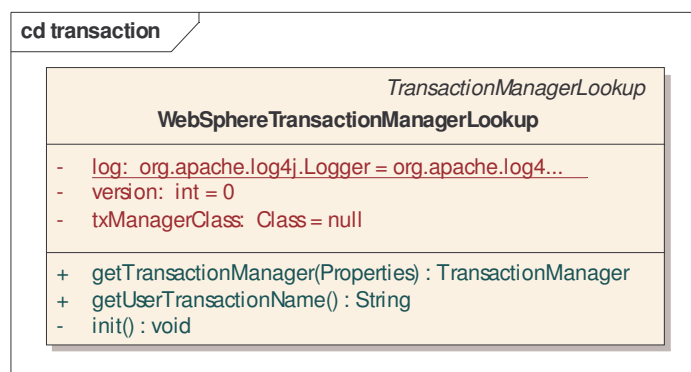
Component	Package	Descripció
-----------	---------	------------



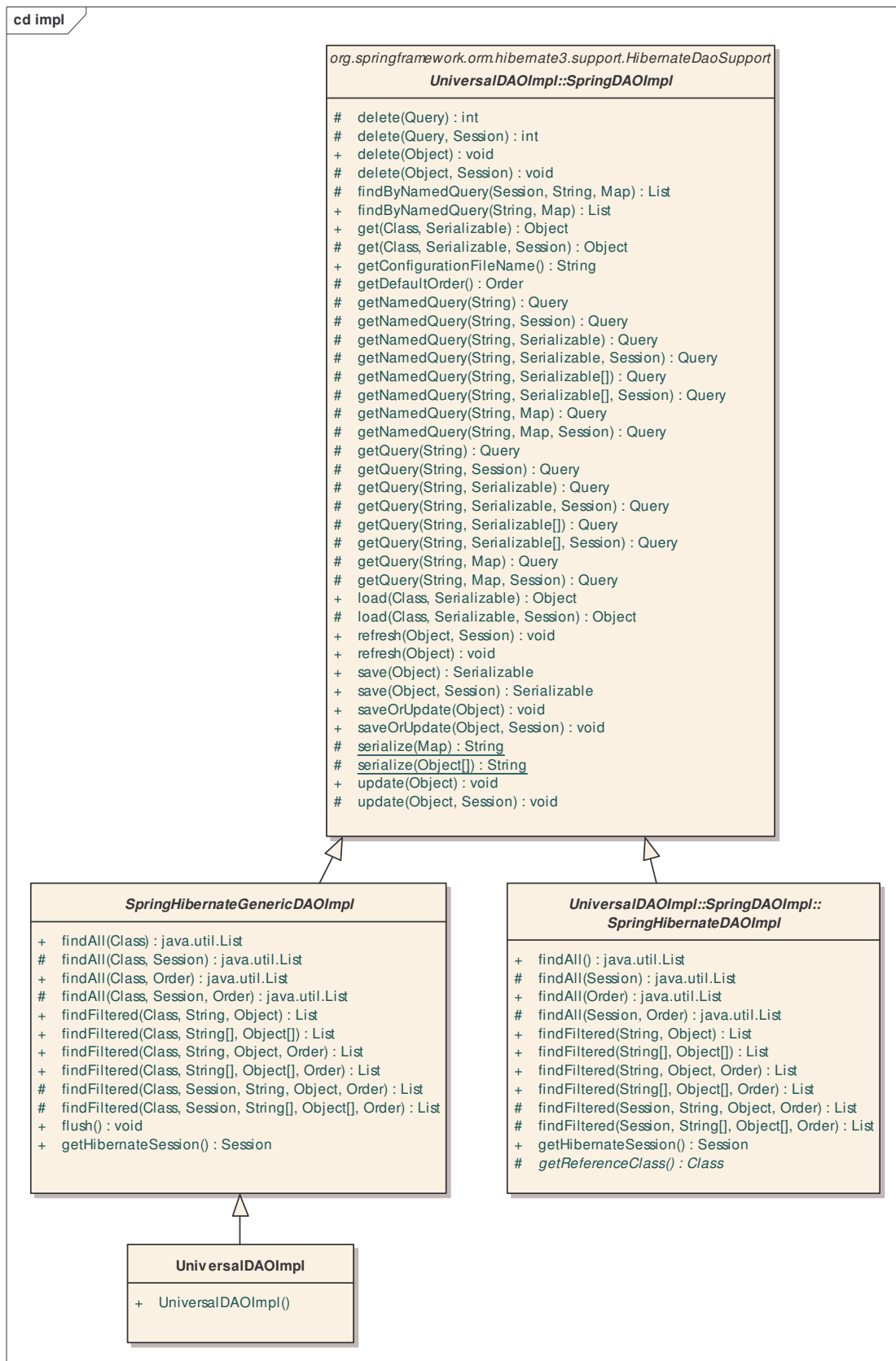
Component	Package	Descripció
DAO	net.opentrends.openframe.services.persistence	Interfície per a les implementacions dels DAO's
HibernateDAO	net.opentrends.openframe.services.persistence	Interfície per a les implementacions dels DAO's amb Hibernate
UniversalHibernateDAO	net.opentrends.openframe.services.persistence	Interfície per a les implementacions del DAO Universal.

2.1.2. Implementació de les interfícies

El servei de dades defineix els següents components:



Component	Package	Descripció
WebSphereTransactionManagerLookup	net.opentrends.openframe.services.persistence.transaction	Delegat que busca al JNDI la transacció JTA corresponent





Component	Package	Descripció
SpringHibernateDAOImpl	net.opentrends.openframe.services.persistence.spring.dao.impl	Implementació del DAO per a Hibernate basada en Spring.
UniversalDAOImpl	net.opentrends.openframe.services.persistence.spring.dao.impl	Implementació de la interfície DAOUniversal.

2.2. Instal·lació i Configuració

2.2.1. Instal·lació

La instal·lació del servei requereix de la utilització de la llibreria 'openFrame-services-persistence' i les dependències indicades a l'apartat 'Introducció-Versions i Dependències'.

2.2.2. Configuració

La configuració del servei implica:

- 1) Definir el servei i injectar-li les seves dependències
- 2) Enllaç d'una acció amb el seu "DAO"

Definició del servei i de les seves dependències

```
<bean id="sessionFactory"
...>
```

Fitxer de configuració: openFrame-services-persistence.xml

Ubicació proposada: <PROJECT_ROOT>/src/main/resources/spring

Factoria de sessions d'Hibernate

Atributs:

Atribut	Requerit	Descripció
class	Sí	Implementació de la factoria de sessions Opcions: <ul style="list-style-type: none">• org.springframework.orm.hibernate3.LocalSessionFactoryBean

També es necessari configurar les següents propietats:

Propietat	Requerit	Descripció
configLocation	Sí	Ubicació del fitxer de propietats d'Hibernate



Propietat	Requerit	Descripció
hibernateProperties	No	Propietats d'Hibernate opcionals per a sobre escriure en funció del host

Exemple:

```
...  
    <bean id="sessionFactory"  
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">  
        <property name="configLocation"  
            value="classpath:${sessionFactory.configLocation}" />  
        <property name="hibernateProperties">  
            <props>  
                <prop  
key="hibernate.connection.datasource">${dataSource.jndiName}</prop>  
            </props>  
        </property>  
    </bean>  
...
```

```
<bean id="transactionManager"  
    ...>
```

Fitxer de configuració: openFrame-services-persistence.xml

Ubicació proposada: <PROJECT_ROOT>/src/main/resources/spring

Delegat transaccional

Atributs:

Atribut	Requerit	Descripció
class	Sí	Implementació del DAO universal. Opcions: <ul style="list-style-type: none">net.opentrends.openframe.services.persistence.spring.dao.impl.UniversalDAOImpl

També es necessari configurar les següents propietats:

Propietat	Requerit	Descripció
sessionFactory	Sí	Referència a la factoria de sessions

Exemple:



```
<bean id="universalHibernateDAO"  
      class="net.opentrends.openframe.services.persistence.spring.dao.impl.UniversalDAOImpl">  
  <property name="sessionFactory" ref="sessionFactory" />  
</bean>
```

Fitxer de configuració: openFrame-services-persistence.xml

Ubicació proposada: <PROJECT_ROOT>/src/main/resources/spring

Delegat transaccional

Atributs:

Atribut	Requerit	Descripció
class	Sí	Implementació del delegat transaccional Opcions: <ul style="list-style-type: none">org.springframework.orm.hibernate3.HibernateTransactionManager

També es necessari configurar les següents propietats:

Propietat	Requerit	Descripció
sessionFactory	Sí	Referència a la factoria de sessions

Exemple:

```
...  
<!-- Transaction manager for a single Hibernate SessionFactory -->  
<bean id="transactionManager"  
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
  <property name="sessionFactory" ref="sessionFactory" />  
</bean>  
...
```

```
<bean id="baseDaoProxy"  
      ...>
```

Fitxer de configuració: openFrame-services-persistence.xml

Ubicació proposada: <PROJECT_ROOT>/src/main/resources/spring

Proxy per a la definició declarativa de la transaccionalitat dels nostres DAO's. Servirà de bean "pare" d'altres beans.

Atributs:

Atribut	Requerit	Descripció
class	Sí	Implementació proxy Opcions: <ul style="list-style-type: none"> org.springframework.transaction.interceptor.TransactionProxyFactoryBean

També és necessari configurar les següents propietats:

Propietat	Requerit	Descripció
target	Sí	Implementació del DAO sobre la que actuarà el proxy. Com estem en la definició "pare" del proxy, el valor serà "java.lang.Object"
transactionAttributes	Si	Atributs transaccionals dels diferents mètodes dels DAO's. Per a més informació sobre les diferents opcions consultar http://www.springframework.org/docs/api/index.html?org/springframework/transaction/interceptor/TransactionProxyFactoryBean.html

Exemple:

```
...
<bean id="baseDaoProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref local="transactionManager" />
    </property>
    <property name="target">
        <bean class="java.lang.Object" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="store*">PROPAGATION_REQUIRED</prop>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
            <prop key="delete*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
...
```



Enllaç d'una acció amb el seu DAO

```
<property name="dao">
  <bean parent="baseDaoProxy">
    ...>
  </bean>
</property>
```

Fitxer de configuració: action-servlet-XXX.xml

Ubicació proposada: <PROJECT_ROOT>/src/main/resources/spring

Cada acció que implementi una cerca necessita la definició d'un bean que hereti del bean “pare” del servei i que tingui els següents atributs:

Atributs:

Atribut	Requerit	Descripció
parent	Sí	Bean “pare”. Opcions: <ul style="list-style-type: none">baseDaoProxy

A més és necessari configurar les següents propietats

Propietat	Requerida	Descripció
target	Sí	Implementació del DAO sobre la que es definirà la transaccionalitat

Exemple:

```
...
<property name="accountTxBO" >
  <bean parent="baseDaoProxy">
    <property name="target" ref="productDao"/>
  </bean>
</property>
...
```

Fitxer de configuració: action-servlet-XXX.xml

Ubicació proposada: <PROJECT_ROOT>/src/main/resources/spring

En el cas que s'hagi de construir un DAO concret per una entitat, s'ha de fer de la següent manera.

S'aconsella l'ús del DAOUniversal enlloc d'un DAO concret, excepte si aquest té alguna particularitat que no dona suport el DAOUniversal.

Cada acció que implementi una cerca necessita la definició d'un bean que hereti de la implementació DAO "pare" del servei (Spring amb Hibernate) i que tingui els següents atributs:

Atributs:

Atribut	Requerit	Descripció
class	Sí	Implementació particular del DAO.

A més és necessari configurar les següents propietats

Propietat	Requerida	Descripció
sessionFactory	Sí	Referència a la factoria de sessions d'Hibernate

Exemple:

```
...  
    <bean id="productDaoTarget"  
    class="net.opentrends.openframe.samples.jpetest.model.dao.hibernate.impl.Hiberna  
    teProductDAOImpl">  
        <property name="sessionFactory" ref="sessionFactory" />  
    </bean>  
...
```



2.3. Utilització del Servei

2.3.1. Actors

En un escenari típic d'utilització d'OpenFrame hi han aspectes importants a considerar. En concret són els derivats de la relació del DAOUniversal amb les 'BeanFactory' que proporciona Spring i les 'SessionFactory' que proporciona Hibernate. Aquests aspectes són bàsicament tres:

1. *D'on traiem el DAOUniversal.*

Una vegada tenim el DAOUniversal ja els podem començar a utilitzar a qualsevol classe. Pensem, per exemple, una classe 'BO's on tinguem una variable d'instància que sigui el DAO universal i que guardi un objecte de negoci:

```
...
public class CategoryBOImpl extends CategoryBO {
    private UniversalDAO dao = null;
    ...
    public void save(Category vo) {
        ...
        dao.saveOrUpdate(vo);
        ...
    }
    ...
}
```

A l'exemple, la instància del UniversalDAO no la hem creat nosaltres si no que és **Spring** qui la **injecta** al nostre BO. Això és **aplicable sempre** i és l'**escenari correcte** d'utilització dels DAO's, es a dir, no serem nosaltres qui instanciem el DAOUniversal si no que deixarem a Spring aquesta tasca mitjançant els fitxers de configuració (típicament applicationContext.xml; a l'apartat Configuració s'explica com fer-ho). La següent figura mostra la jerarquia de dependències dels 'beans' d'Spring per aquest cas senzill (sense introduir cap concepte de transaccionalitat):

2. *Com es relacionen els nostres BO's amb el DAOUniversal mitjançant Spring.*

El nostres BOs mitjançant la injecció d'Spring tindran el DAOUniversal. L'exemple a continuació mostra el citat anteriorment.



```
<bean id="categoryBO"
class="net.opentrends.openframe.samples.jpetsstore.model.bo.impl.CategoryBOImpl">
  <property name="dao" ref="universalHibernateDAO"/>
</bean>
```

3. *Com saben la transaccionalitat dels nostres BO's i .les característiques de transaccionalitat dels seus mètodes.*

Aquest punt requereix un apartat separat que analitzem a continuació.

2.3.2. Transaccionalitat

L'escenari plantejat en l'apartat anterior es correspon amb un exemple senzill d'utilització d'un DAO qualsevol sense introduir en cap moment el concepte de transaccionalitat. Això no es correspon amb una aplicació complex, a on les operacions de base de dades es realitzen en bloc (transacció) i a on el resultat pot ser que vulguem fer enrera tota la transacció si es produeix un error.

```
<!-- Category BO Target -->
<bean id="categoryBOTarget"
class="net.opentrends.openframe.samples.jpetsstore.model.bo.impl.CategoryBOImpl">
  <property name="dao" ref="universalHibernateDAO"/>
</bean>

<!-- categories -->

<bean name="/categories"

class="net.opentrends.openframe.samples.jpetsstore.struts.action.CategoryAction"
>
  <property name="logService" ref="loggingService"/>
  <property name="pojoClass"

value="net.opentrends.openframe.samples.jpetsstore.model.Category" />
  <property name="bo" >
    <bean parent="baseDaoProxy">
      <property name="target"><ref
bean="categoryBOTarget"/></property>
    </bean>
  </property>
  ..
</bean>
```

Una de les diferències amb la versió 1.0 es que es defineix la transaccionalitat a nivell de DAO envers a nivell de BO. Això ha deixat de tenir sentit ja que una operació de negoci pot realitzar diferents accions contra base de dades i es vol que totes operin en la mateixa transacció. Per tant el més correcte és definir les transaccions a nivell de operacions de negoci que és el que conceptualment és vol que compleixi l'especificació ACID.

A més en la versió 1.1 no s'aconsella fer servir DAO per a cada entitat de negoci per l'alt cos de manteniment i no té cap utilitat, excepte algun cas on es vulgui alguna acció pròpia d'alguna entitat de negoci. En els casos més habituals s'ha de fer d'una única instància del DAOUniversal, que s'ha injectar en tots el BO.

La diferència principal amb un BO transaccional amb un que no ho és, es que l'action que fa servir aquest BO, no el té injectat directament sinó amb un bean amb pare baseDAOProxy.

Aquest proxy crea una classe al vol que compleix la interfície del BO. Aquesta classe s'encarrega de gestionar les transaccions abans d'invocar el BO real. Les transaccions es configuren en la classe baseDAOProxy definida en openFrame-service-persistence.xml.

L'exemple següent mostra la configuració d'aquest baseDaoProxy.

```
<bean id="baseDaoProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref local="transactionManager" />
  </property>
  <property name="target">
    <bean class="java.lang.Object" />
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="store*">PROPAGATION_REQUIRED</prop>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
      <prop key="delete*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Aquest exemple el baseDaoProxy, gestiona les transaccions dels mètodes que comencen per get,find, load,store,save,delete.

Els atributs per gestió de les transaccions són

VALOR	DESCRIPCIÓ
PROPAGATION_MANDATORY	Fa ús de la transacció actual. Sino n'hi ha llença un excepció.
PROPAGATION_NESTED	S'executa com a transacció anidada si existeix una transacció, sino es comporta com PROPAGATION_REQUIRED.
PROPAGATION_NEVER	S'executa en un entorn no transaccional. Si existeix transacció llença una excepció.
PROPAGATION_NOT_SUPPORTED	S'executa en entorn no transaccional. Si



	existeix transacció la suspèn fins la finalització del mètode
PROPAGATION_REQUIRED	Fa servir la transacció actual i sino existeix en crea una de nova.
PROPAGATION_REQUIRES_NEW	Crea una nova transacció, i si existeix una d'oberta l'atura fins la finalització del mètode.
PROPAGATION_SUPPORTS	Fa servir la transacció existent, i sinó n'hi ha cap en el mètode s'executa en un entorn no transaccional.

L'atribut readOnly es una optimització que indica que aquesta transacció no farà modificacions a base de dades.

2.4. Eines de Suport

2.4.1. *Hibernate Synchronizer*

En la versió 1.1 no s'aconsella l'ús d'aquesta eina a causa de la quantitat de DAOs que es generaven i la rigidesa de les plantilles.

En la versió 1.0 aquesta eina era d'ús obligatori.

2.4.2. *Hibernate Tools*

L'Hibernate Tools és una eina opensource del projecte Hibernate que ajuda en la tasca de la generació de mappings, pojos, i creació d'esquemes de base de dades.

Per tant, es recomana fer servir aquesta eina enlloc del Hibernate Synchronizer.

2.5. Integració amb Altres Serveis

2.6. Preguntes Freqüents

3. Exemples

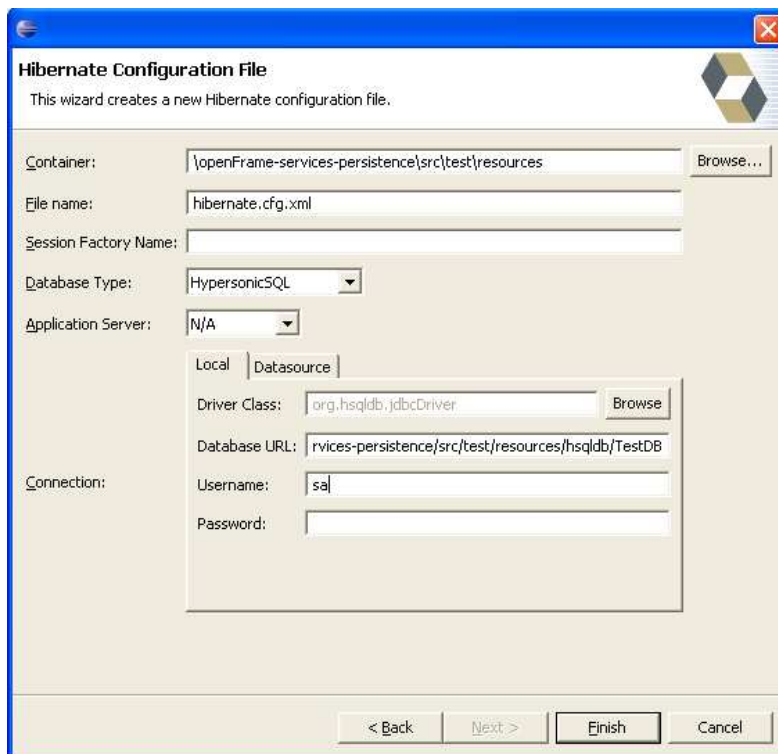
3.1. Test unitaris

Com a exemple d'utilització implementarem un test unitari que faci recull de tots els conceptes implicats. Resumint, necessitem:

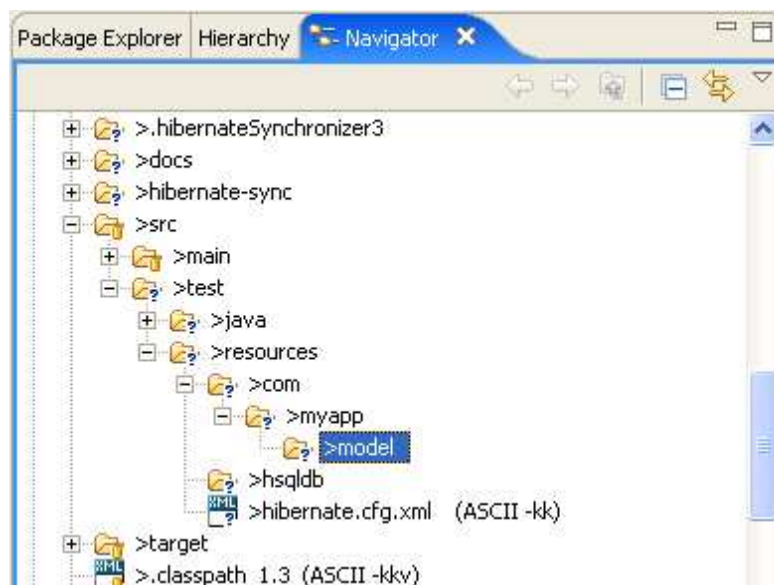
- Fitxer de configuració d'Hibernate (típicament 'hibernate.cfg.xml')
- Fitxers de mapeig de les taules que volem manejar (típicament '*.hbm.xml')
- Interfícies i implementacions dels DAO's.
- Objectes de negoci (també coneguts com 'Value Objects')
- Fitxer de configuració d'Spring (típicament applicationContext.xml)
- Classe principal. En el nostre cas estendrà 'TestCase' i farà les vegades de 'Action' en una aplicació d'Struts.

Partim doncs d'una base de dades Hypersonic senzilla amb dues taules: CATEGORY i PRODUCT i una relació '1..n' entre Category i Product. També obviem la part de configuració de l'Hibernate Sync. ja que s'explica en detall en apartats posteriors. Així doncs els passos són:

1. Crear el fitxer de propietats d'Hibernate. Menú 'File/New/Other', seleccionem 'Hibernate/Hibernate Configuration File' i omplim totes les propietats correctament:

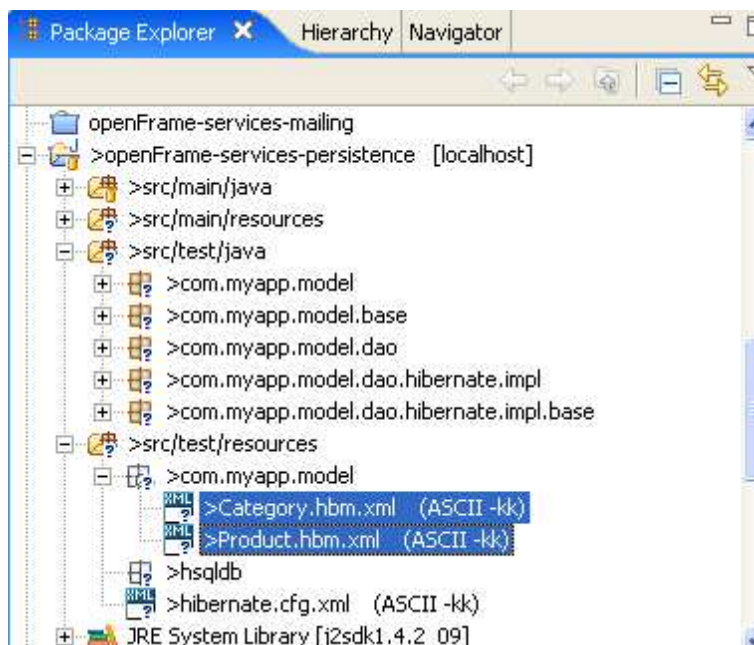


2. Crear els directoris a on situarem els fitxers de mapeig:
'/src/test/resources/com/myapp/model'



3. Crear el fitxer de mapeig de les taules seleccionades. Menú 'File/New/Other', seleccionem 'Hibernate/Hibernate Mapping File' i omplim totes les propietats correctament:

4. Sincronitzem les fitxers de mapeig per crear les classes Java i obtenim el següent resultat:



5. Afegim el fitxer 'applicationContext.xml' al directori '/src/test/resources' per a configurar les nostres classes com a beans d'Spring. Farem servir un escenari transaccional per a veure tots els conceptes:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
  
```





```
<beans>
```

```
<!-- Hibernate SessionFactory -->
<bean name="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
</bean>

<!-- Transaction manager for a single Hibernate SessionFactory (alternative to
JTA) -->
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<!-- Category DAO Proxy -->
<bean id="categoryDaoProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
>
    <property name="transactionManager">
        <ref local="transactionManager" />
    </property>
    <property name="target">
        <ref local="categoryDaoTarget" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="store*">PROPAGATION_REQUIRED</prop>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
<!-- Category DAO Target -->
<bean id="categoryDaoTarget"
class="com.myapp.model.dao.hibernate.impl.HibernateCategoryDAOImpl">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
<!-- Category Instances -->
<bean id="category" class="com.myapp.model.Category" singleton="false"/>
<!-- Product Instances -->
<bean id="product" class="com.myapp.model.Product" singleton="false"/>
<!-- Logging Service -->
<bean id="loggingConfigurator"
class="net.opentrends.openframe.services.logging.log4j.xml.HostDOMConfigurator"
init-method="init">
    <property
name="configFileName"><value>D:\users\manelix\workspace\openFrame-services-
persistence\src\test\resources\log4j-test.xml</value></property>
</bean>
<bean id="loggingService"
class="net.opentrends.openframe.services.logging.log4j.Log4JServiceImpl" init-
method="init">
    <property name="configurator"><ref
local="loggingConfigurator"/></property>
</bean>
</beans>
```



6. Ara només queda implementar el TestCase.

```
...
public class DAOTest extends TestCase {
...
    protected void setUp() throws Exception {
        super.setUp();
        ClassPathXmlApplicationContext appContext = new
        ClassPathXmlApplicationContext("applicationContext.xml");
        this.factory = (BeanFactory) appContext;
        /**
         * Logging service
         */
        LoggingService logService =
        (LoggingService) factory.getBean("loggingService");
        this.log = logService.getLog(this.getClass());
    }
    public void testSave(){
        /**
         * Request the DAO to manipulate category instances
         */
        CategoryDAO categoryDAO =
        (CategoryDAO) factory.getBean("categoryDaoProxy");
        /**
         * Request a new category instance
         */
        Category newCategory = (Category) factory.getBean("category");
        newCategory.setName("name at "+System.currentTimeMillis());
        newCategory.setDescn("descn at "+System.currentTimeMillis());
        try {
            categoryDAO.save(newCategory);
        }
        catch(PersistenceServiceException ex){
            log.warn("Se ha producido un error:
            "+ex.getLocalizedMessage());
        }
        log.debug("El ID de la nueva category es:
        "+newCategory.getId());
        assertTrue("La category no se ha guardado
        correctamente!", newCategory.getId() != null);
    }
    ...
}
```

A l'exemple anterior cal destacar:

- La utilització dels DAO's al nostre codi es a través de les interfícies i no les implementacions, encara que al fitxer de configuració d'Spring que el bean retorna la implementació.
- Només hi ha una operació de base de dades. Com tot va bé s'executa un 'commit' totalment transparent per l'usuari que consolida les dades a la BD. Això poder no reflexa massa una transacció real, en la que en la mateixa transacció es poden executar múltiples operacions de BD. Anem per tant a crear una situació fictícia d'aquest comportament.



7. Afegim un objecte fictici que faci varies operacions de 'save' i que contingui el nostre DAO:

```
<bean id="multiSaveTarget" class="com.myapp.test.MultiSave">
    <property name="categoryDAO" ref="categoryDaoProxy" />
    <property name="logService" ref="loggingService" />
</bean>
```

i afegim un gestor transaccional sobre aquest objecte per indicar que volem una transacció sobre el mètode que inicia la transacció i llença totes les operacions de base de dades:

```
<bean id="multiSaveProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref local="transactionManager" />
    </property>
    <property name="target">
        <ref local="multiSaveTarget" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

8. Ara només hem de crear la classe 'MultiSave' :

```
...
public class MultiSave {
...
    public void saveMultiple(List categories) {
        Iterator it = categories.iterator();
        int i = 0;
        while(it.hasNext()){
            Category c = (Category)it.next();
            this.logService.getLog(this.getClass()).debug("Trying to
save category...");
            if(i==categories.size()-1){
                this.categoryDAO.update(c);
            }
            else {
                this.categoryDAO.save(c);
            }
            this.logService.getLog(this.getClass()).debug("Category
saved with id "+c.getId());
            i++;
        }
    }
...
}
```



```
}
```

i afegir un altre cas de test a la nostra classe DAOTest:

```
...
public class DAOTest extends TestCase {
...
    public void testMultipleSave() {
        /**
         * Request a the 'multisave object'
         */
        MultiSave multiSave =
        (MultiSave) factory.getBean("multiSaveProxy");
        ArrayList list = new ArrayList();
        for(int i=0; i<3; i++) {
            /**
             * Request a new category instance
             */
            Category newCategory =
            (Category) factory.getBean("category");
            newCategory.setName("name at "+System.currentTimeMillis());
            newCategory.setDescn("descn at
            "+System.currentTimeMillis());
            list.add(newCategory);
        }
        try {
            multiSave.saveMultiple(list);
        }
        catch(PersistenceServiceException ex) {
            log.warn("Se ha producido un error:
            "+ex.getLocalizedMessage());
        }
        Iterator it = list.iterator();
        while(it.hasNext()) {
            Category c = (Category) it.next();
            if(c.getId() != null) {
                log.debug("Loading category with ID="+c.getId());
                Category cLoaded = multiSave.loadCategory(c.getId());
                log.debug("Loaded from BD a category "+cLoaded+" with
                Id="+c.getId()+"?" + (cLoaded != null));
                assertTrue("The category has been saved into DB and
                should not!", cLoaded == null);
            }
        }
        log.debug("End.");
    }
...
}
```

Si ens fixem, en el mètode *saveMultiple(...)* el que estem fent es provocar un error a l'hora de guardar la última Category, ja que estem cridant a *update(...)* quan la instància és nova i hauríem de cridar a *save(...)*. Això provocarà un 'rollback' de totes les instàncies prèviament

guardades. Com ho comprovem? Posteriorment en el codi intentem fem un *load(...)* per a veure si s'ha desat algun registre. Això mateix ho podem mirar veient els logs generats:

```
DEBUG [main] org.hibernate.transaction.JDBCTransaction - begin
DEBUG [main] org.hibernate.transaction.JDBCTransaction - current autocommit
status: false
DEBUG [main] com.myapp.test.MultiSave - Trying to save category...
DEBUG [main] com.myapp.test.MultiSave - Category saved with id 1
DEBUG [main] com.myapp.test.MultiSave - Trying to save category...
DEBUG [main] com.myapp.test.MultiSave - Category saved with id 2
DEBUG [main] com.myapp.test.MultiSave - Trying to save category...
DEBUG [main] com.myapp.test.MultiSave - Persistence error will occur...
DEBUG [main] org.hibernate.transaction.JDBCTransaction - rollback
DEBUG [main] org.hibernate.transaction.JDBCTransaction - rolled back JDBC
Connection
WARN [main] com.myapp.test.DAOTest - Se ha producido un error:
org.springframework.dao.InvalidDataAccessApiUsageException: The given object has a
null identifier: com.myapp.model.Category; nested exception is
org.hibernate.TransientObjectException: The given object has a null identifier:
com.myapp.model.Category
DEBUG [main] com.myapp.test.DAOTest - Loading category with ID=1
DEBUG [main] org.hibernate.transaction.JDBCTransaction - begin
DEBUG [main] org.hibernate.transaction.JDBCTransaction - current autocommit
status: false
DEBUG [main] org.hibernate.transaction.JDBCTransaction - commit
DEBUG [main] org.hibernate.transaction.JDBCTransaction - committed JDBC Connection
DEBUG [main] com.myapp.test.DAOTest - Loaded from BD a category null with
Id=1?false
DEBUG [main] com.myapp.test.DAOTest - Loading category with ID=2
DEBUG [main] org.hibernate.transaction.JDBCTransaction - begin
DEBUG [main] org.hibernate.transaction.JDBCTransaction - current autocommit
status: false
DEBUG [main] org.hibernate.transaction.JDBCTransaction - commit
DEBUG [main] org.hibernate.transaction.JDBCTransaction - committed JDBC Connection
DEBUG [main] com.myapp.test.DAOTest - Loaded from BD a category null with
Id=2?false
DEBUG [main] com.myapp.test.DAOTest - End.
```

La seqüència és la següent:

1. S'obre una transacció (missatge: *org.hibernate.transaction.JDBCTransaction - begin*)
1. S'intenten desar els dues primeres 'Category'. Tot correcte
2. S'intenta actualitzar la tercera 'Category'. Error. El framework fa un rollback de tot i no es desar res (missatge: *org.hibernate.transaction.JDBCTransaction - rollback*). Tanquem transacció.
3. Obrim transacció (missatge: *org.hibernate.transaction.JDBCTransaction - begin*)
4. Intentem carregar de la base de dades els suposats registres que s'havien desat sense problemes (ID=1 i ID=2; missatge: *Loading category with ID=1 i ID=2*)
5. Tanquem transacció (missatge: *org.hibernate.transaction.JDBCTransaction - commit*)
6. Les dades no existeixen. Tot és correcte (missatge: *Loaded from BD a category null with Id=1 i Id=2?false*)



4. Annexos