



Framework Corporatiu J2EE

Servei de Persistència

Versió 1.0

Barcelona, 21 / febrer / 2006



Històric de modificacions

Data	Autor	Comentaris	Versió
09/12/2005	Atos Origin, sae openTrends	Versió inicial del document	1.0

Llegenda de Marcadors



Índex

1.	INTRODUCCIÓ	4
1.1.	PROPÓSIT	4
1.2.	CONTEXT I ESCENARIS D'ÚS	4
1.3.	VERSIONS I DEPENDÈNCIES	4
1.3.1.	<i>Dependències Bàsiques</i>	4
1.4.	A QUI VA DIRIGIT	5
1.5.	DOCUMENTS I FONTS DE REFERÈNCIA	5
1.6.	GLOSSARI	5
2.	DESCRIPCIÓ DETALLADA	6
2.1.	ARQUITECTURA I COMPONENTS	6
2.1.1.	<i>Interfícies i components genèrics</i>	6
2.1.2.	<i>Implementació de les interfícies</i>	6
2.2.	INSTAL·LACIÓ I CONFIGURACIÓ	8
2.2.1.	<i>Instal·lació</i>	8
2.2.2.	<i>Configuració</i>	8
2.3.	UTILITZACIÓ DEL SERVEI	12
2.3.1.	<i>Actors</i>	12
2.3.2.	<i>Transaccionalitat</i>	14
2.4.	EINES DE SUPORT	14
2.4.1.	<i>Hibernate Synchronizer</i>	14
2.5.	INTEGRACIÓ AMB ALTRES SERVEIS	27
2.6.	PREGUNTES FREQUÈNTS	27
3.	EXEMPLES	28
3.1.	TEST UNITARIS	28
4.	ANNEXOS	36

1. Introducció

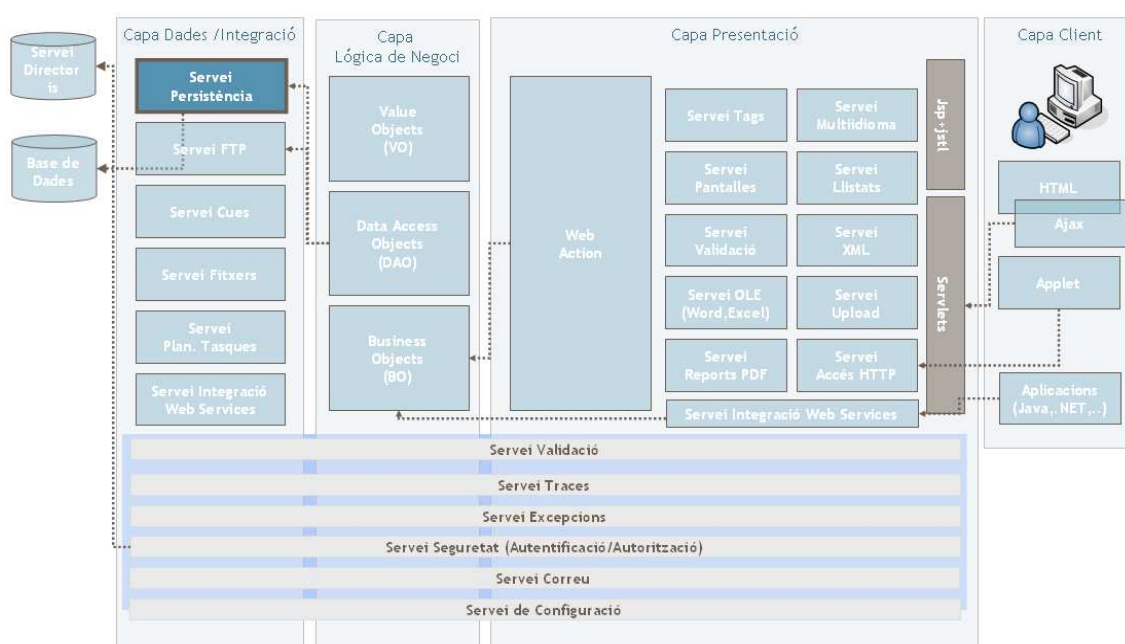
1.1. Propòsit

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

En el present apartat es mostren quines són les versions i dependències necessàries per fer ús del Servei.

1.3.1. Dependències Bàsiques

Nom	Tipus	Versió	Descripció
-----	-------	--------	------------



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	1.0-SNAPSHOT	
openframe-services-exceptions	jar	1.0-SNAPSHOT	

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 dona 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

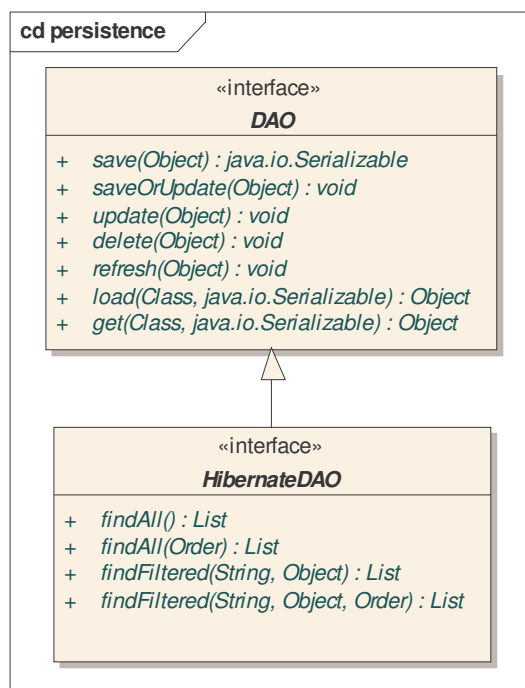
2.1. Arquitectura i Components

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

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

2.1.1. Interfícies i components genèrics

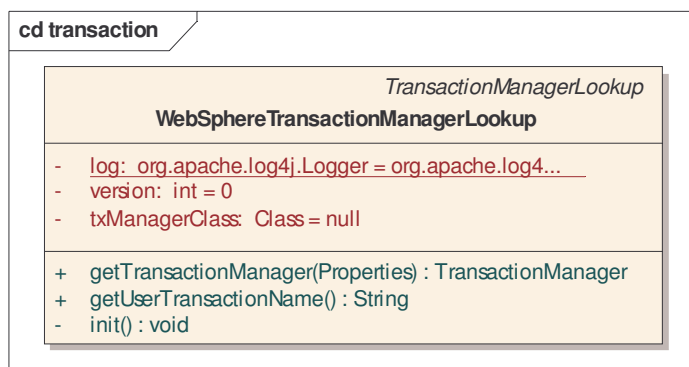
El servei de llistats defineix les següents interfícies:



Component	Package	Descripció
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

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.

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
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 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 es 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>
    </props>
  </property>
</bean>
```



```
        <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 búsqueda 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="dao" >
  <bean parent="baseDaoProxy">
    <property name="target" ref="productDaoTarget"/>
  </bean>
</property>
...
```



```
<bean name="productDaoTarget">  
    ...>
```

Fitxer de configuració: action-servlet-XXX.xml

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

Cada acció que implementi una búsqueda 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.jpstore.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 dels nostres DAO's amb les 'BeanFactory' que proporciona Spring i les 'SessionFactory' que proporciona Hibernate. Aquests aspectes són bàsicament tres:

1. *D'on treiem les instàncies dels DAO's.*

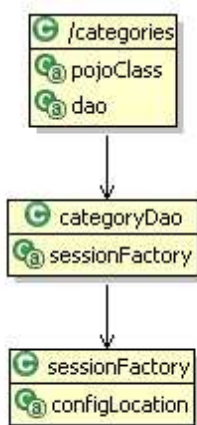
Una vegada tenim els DAO's ja els podem començar a utilitzar a qualsevol classe. Se'ns ocòrrer, per exemple, una classe 'Action' d'Struts a on tinguem una variable d'instància que sigui un dels nostres DAO's i que guardi un objecte de negoci:

```
...  
public class CategoryAction extends ActionExtendedSupport {  
    private CategoryDAO dao = null;  
...  
    public ActionForward save(Category vo, StrutsContext context){  
        ...  
        dao.saveOrUpdate(vo);  
        ...  
        return context.getActionMapping().findForward("success");  
    }  
    ...  
}
```

A l'exemple, la instància del CategoryDAO no la hem creat nosaltres si no que és **Spring** qui la **injecta** al nostre 'Action'. Això és **aplicable sempre** i és l'**escenari correcte** d'utilització dels DAO's, es a dir, no serem nosaltres qui instanciaran els DAO's 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 gerarquia de dependències dels 'beans' d'Spring per aquest cas senzill (sense introduir cap concepte de transaccionalitat):

2. *Com es relacionen els nostres DAO's amb la 'SessionFactory' d'Hibernate que han de fer servir.*

A la figura anterior veiem que el nostre DAO es relaciona amb la 'SessionFactory' d'Hibernate a partir de la propietat 'sessionFactory'. Això es així perquè els nostres DAO's hereten tots de 'net.opentrends.openframe.services.persistence.spring.dao.impl.SpringHibernateDAOImpl'. Aquesta herència la definim a nivell de configuració de projecte d'Hibernate Synchronizer. A l'apartat Configuració s'explica com.





3. *Com saben els nostres DAO's 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 complexa, a on les operacions de base de dades es realitzen en bloc (transacció) i a on el resultat pot ser que volguem fer enrera tota la transacció si es produeix un error.

Hem de definir per tant la transaccionalitat de les nostres operacions. Això implica, mirant l'exemple anterior, que el nostre *dao.saveOrUpdate(...)* ha de fer rollback o commit en funció de si es produeix o no un error. Així doncs definirem que el nostre mètode en qüestió és transaccional o no i quines característiques s'apliquen (segons s'ha introduït en l'apartat primer d'aquest document). La següent figura mostra la gerarquia de beans que es faria servir en aquest cas (típicament al fitxer *applicationContext.xml*), i que aprofitarem per anar explicant el procediment:

El primer que destaca és l'aparició en escena d'un 'categoryDaoProxy' i un 'transactionManager'. El que abans era el 'categoryDao' ara és el 'categoryDaoTarget'. Ambdós beans són classes que ens proporciona Spring. L'explicació de com configurar-les es troba a l'apartat Configuració. La idea que hi ha darrere de tot això és ben senzilla. En lloc d'utilitzar directament el nostre DAO, utilitzem un proxy que ens intercepta les crides als nostres mètodes i que afegeix la transaccionalitat amb el manager que li indiquem, en aquest cas un manager d'Hibernate que fa ús de la 'sessionFactory' configurada. A nivell de codi tot queda igual, és a dir, farem ús de les nostres interfícies DAO però per sota s'apliquen els conceptes necessaris per garantir la coherència de les nostres dades. Tot això és possible perquè és Spring qui ens proporciona les instàncies de les implementacions dels nostres DAO's, tal i com s'ha remarcat en l'apartat anterior.

2.4. Eines de Suport

2.4.1. Hibernate Synchronizer

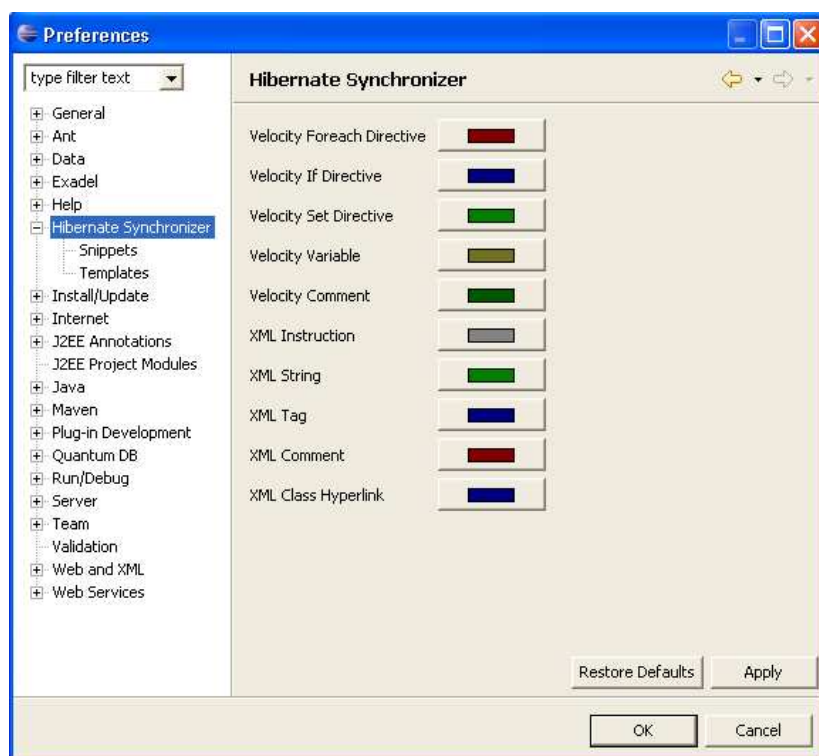
Important

La versió que es fa servir d'Hibernate Synchronizer, 3.1.1, només funciona correctament en un Eclipse 3.1 que s'executi sota una JDK 5. Per tant, cal arrencar eclipse amb una definició de la variable d'entorn *JAVA_HOME* que apunti a un directori d'instal·lació d'aquest JDK. Alternativament, es pot crear un .bat o .sh que arrenqui l'eclipse amb aquesta variable correctament assignada.

Això no implica que el codi que es generi posteriorment es compili amb JDK 5, ja que es pot configurar en el menú 'Window/Preferences/Java/Installed JREs' tal.

La versió d'Hibernate Synchronizer que s'utilitza amb OpenFrame no és directament la que es pot baixar de la web, si no que és una versió modificada amb la correcció d'alguns bugs proporcionada amb el propi framework.

Per a configurar aquest plugin, s'ha d'accedir a 'Window/Preferences/Hibernate Synchronizer':

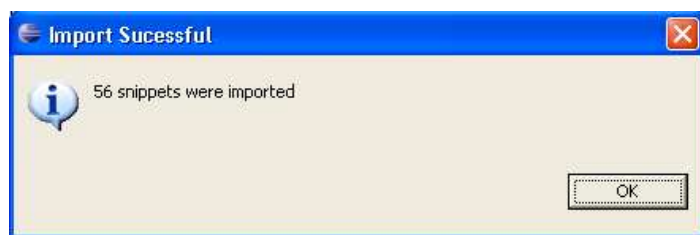


Els dos ítems a configurar són:

- Snippets
- Templates

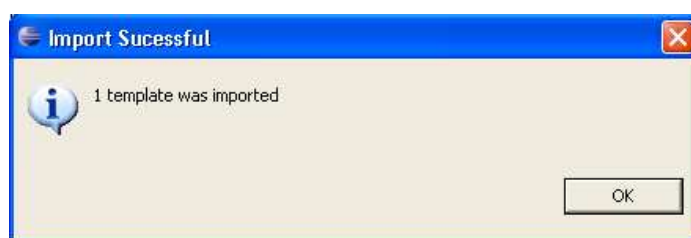
Els 'Snippets' són troços de codi que es faran servir als 'Templates'. Per a poder utilitzar el framework correctament s'han d'importar els 'Snippets' que es proporcionen amb el mateix (típicament 'openFrame-hibernate_sync-snippets.zip') que són les versions modificades dels 'Snippets' originals de l'Hib.Sync. modificats per funcionar amb JDK 1.4.2 o superior i amb Spring, seguint un patró de treball DAO + interface DAO.

Per tant, en la vista 'Snippets' fem clic al botó 'Import' y seleccionem el fitxer comprimit amb els continguts correctes. Si tot va bé obtindrem una finestra a on ens indica el número de 'Snippets' importats:

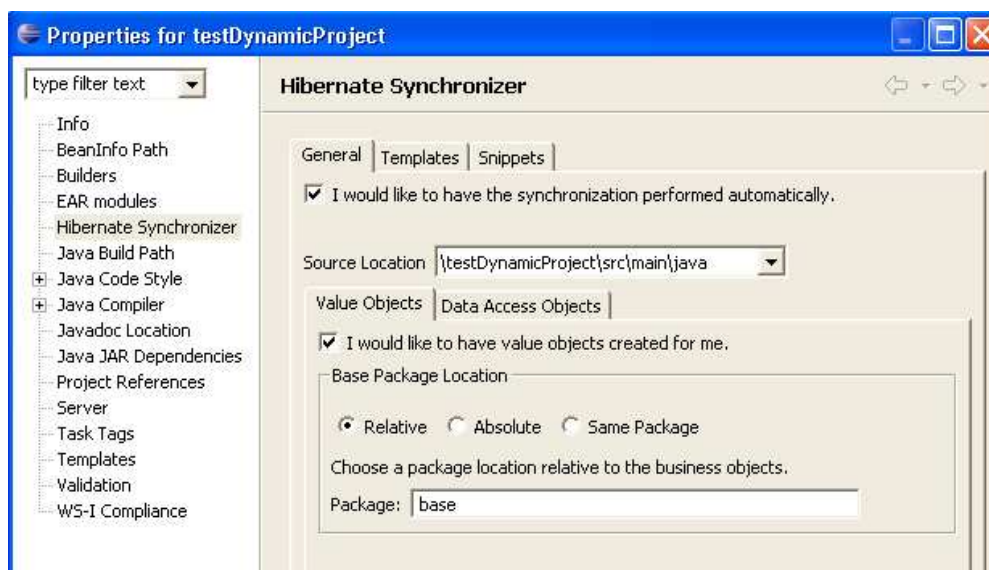


Els 'Templates' o plantilles són els que generen les classes que després farem servir. Per a poder utilitzar el framework correctament s'han d'importar els 'Templates' que es proporcionen amb el mateix (típicament 'OpenFrame-hibernate_sync-templates.zip') que són templates nous per funcionar seguint un patró de treball DAO + interface DAO.

Per tant, en la vista 'Templates' fem clic al botó 'Import' y seleccionem el fitxer comprimit amb els continguts correctes. Si tot va bé obtindrem una finestra a on ens indica el número de 'Templates' importats:

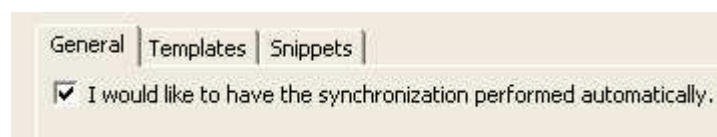


Una vegada fet això ja tenim configurat l'Hibernate Syncn. a nivell de 'Workspace'. Per a poder fer-lo servir correctament encara ens manca un altre pas, i és configurar-lo a nivell de projecte a on definirem, entre d'altres, la gerarquia i noms de les classes. Obrim les propietats del nostre projecte amb el menú contextual 'Project/Properties/Hibernate Synchronizer':

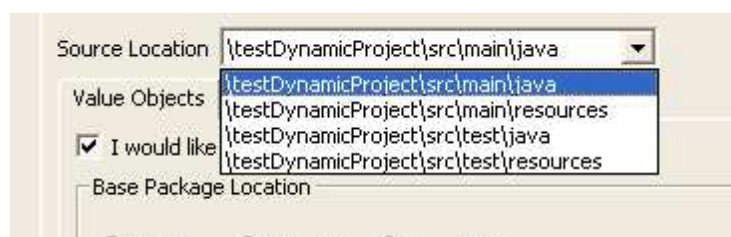


Apareixen diverses pestanyes per a configurar la generació automàtica de classes. A continuació es detalla la informació que conté cadascuna d'elles:

- Pestanya '**General**': Informació relativa als noms i ubicacions de les classes generades.
 1. **Activem** 'I would like to have the synchronization performed automatically.' per tal de fer la sincronització automàtica amb la base de dades.

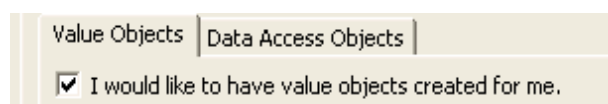


2. **Seleccionem** el 'Source Location': en una estructura de projecte amb suport 'Maven', hauríem de ser capaços de seleccionar entre les següents quatre alternatives:

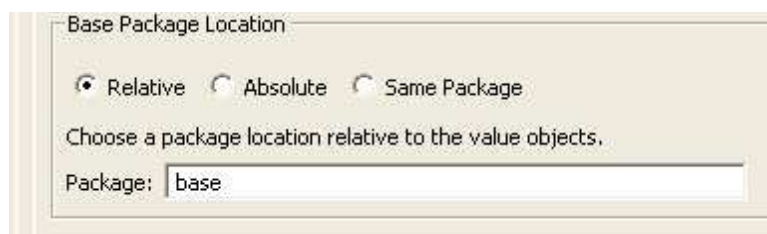


segons la configuració del projecte. La que ens interessa és la que fa referència a '/src/main/java'

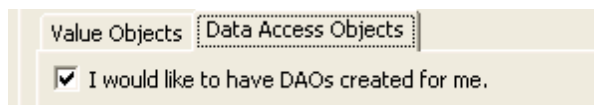
3. **Activem** 'I would like to have value objects created for me.' per tal de crear el 'Value Objects'. Aquests objectes són els POJO's (*Java Plain Object*) que es passaran als DAO's per guardar els registres a la base de dades.



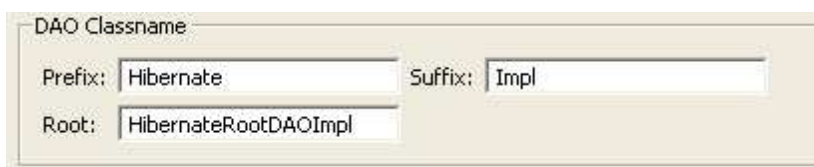
4. **Seleccionem** 'Base Package Location' amb valor 'Relative' i **introduïm** a 'Choose a package location relative to the value objects.' el literal 'base'. Això fa que les classes base dels POJO's es guardin en un 'package' diferent i així tenir una estructura de directoris més neta.



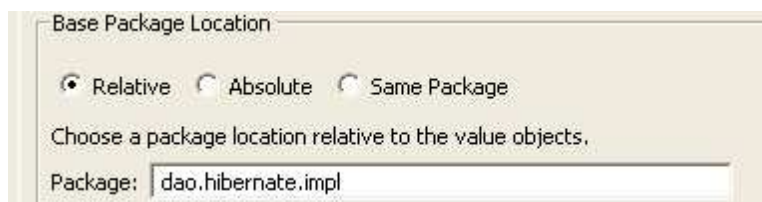
5. **Activem** a la pestanya 'Data Access Objects' 'I would like to have DAO's created for me.' per tal de crear els DAO's (interfícies i implementacions)



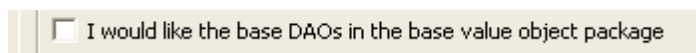
6. **Introduïm** els prefix i sufix dels noms de les classes de les implementacions dels DAO's generats i el valor del nome de la superclasse de totes les implementacions ('Root'). Els valors són el literal 'Hibernate' pel prefix, i el literal 'Impl' pel sufix i 'HibernateRootDAOImpl'. Aquests valors són aquests i no uns altres per la tecnologia utilitzada per fer el mapeig d'un model relacional a objectes.



7. **Seleccionem** 'Base Package Location' amb valor 'Relative' i **introduïm** a 'Choose a package location relative to the value objects.' el literal 'dao.hibernate.impl'. Això fa que les classes de les implementacions dels DAO's es guardin en un 'package' diferent i així tenir una estructura de directoris més neta.



8. **Desactivem** 'I would like the base DAOs in the base value objects package' per tal de no barrejar els 'Value Objects' amb els DAO's.



9. **Seleccionem** 'Base DAO Location' amb valor 'Relative' i **introduïm** a 'Package:' el literal 'base'



El resultat d'aquesta ruta relativa és que ens generarà les classes base de les implementacions dels DAO's en el package 'dao.hibernate.impl.base'

10. Per últim només ens queda **seleccionar** la classe base dels nostres DAO's que serà la que ens proporciona el framework. Per això, seleccionem 'I would like to use a custom DAO root' i **introduïm** 'net.opentrends.openframe.services.persistence.spring.dao.impl.SpringHibernateDAOImpl' com a 'DAO class' i 'net.opentrends.openframe.services.persistence.exception.PersistenceServiceException' com a 'DAO Exception'



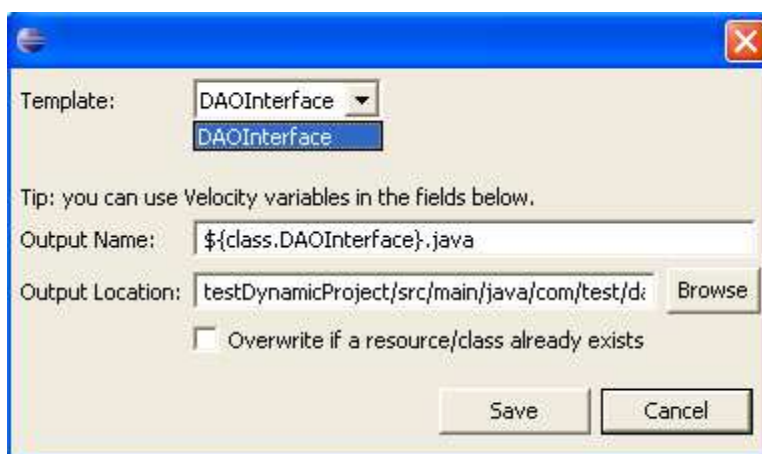
☒ I would like to use a custom DAO root.

DAO Class:

DAO Exception:

Cal remarcar que per poder utilitzar el 'DAO root' y 'DAO Exception' especificades, cal tenir ven configurada la dependència del servei d'excepcions en el nostre projecte tant a nivell de maven com de classpath de projecte.

- Pestanya '**Templates**': Informació relativa a les plantilles 'extres' que s'utilitzaran per generar les nostres classes. Per defecte el plugin genera les implementacions dels DAOs que farem servir en el nostre projecte, però manca la generació de les interfícies correctes per aquestes implementacions. Fem clic a 'New' per afegir la nostra plantilla i ens apareix una finestra a on introduïrem els següents valors:



Template:

Tip: you can use Velocity variables in the fields below.

Output Name:

Output Location:

☐ Overwrite if a resource/class already exists



- Template: la plantilla que farem servir. En aquest cas només tenim una opció: 'DAOInterface'
- Output Name: és el nom del fitxer generat que es correspon amb el nom de la classe de la interfície. En aquest cas: '\$ {class.DAOInterface}.java'
- Output Location: és el directori a on es crearà la interfície. Fent clic en el botó 'Browse' **hem de seleccionar** el subdirectori 'dao' a partir del directori de creació dels 'Value Objects'. Per exemple, suposem que els nostres objectes de negoci els guardem en un directori que es diu 'com.myapp.model'. El 'Output Location' en aquest cas seria 'com.myapp.model.dao'.

Ara ja tenim el Hibernate Synchronizer correctament configurat per a la generació automàtica de les nostres classes segons el patró de treball DAO+Interfície DAO.

La utilització del servei de dades comporta el coneixement de l'eina Hibernate Synchronizer, els conceptes que es manegen i els passos a seguir per a poder persistir les dades, més enllà de les classes disponibles en el propi framework. Els passos a seguir en aquest ordre són:

1. Configuració d'Hibernate Synchronizer (com s'ha vist adalt)
2. Creació dels fitxers propis d'Hibernate: **hibernate.cfg.xml** i **fitxers de mappings (*.hbm.xml)**
3. Sincronització dels fitxers de mappings per a la creació dels DAO's i objectes de negoci.
4. Definició dels actors que intervenen en el servei de dades
5. Definició de la transaccionalitat de les nostres operacions de negoci
6. Utilització dels DAO's en la nostra aplicació

A continuació es detalla cadascun dels passos a seguir per a completar amb èxit una operació de persistència (el pas de configuració d'Hibernate Sync. el passarem per alt ja que es detalla ampliament en apartats posteriors)

Lo primer que tenim que fer és definir les propietats necessàries per al motor de persistència que farem servir: Hibernate 3. Típicament aquestes propietats es defineixen en un fitxer que es diu '**hibernate.cfg.xml**'.

Per tant, en el menú 'File/New/Other' seleccionem 'Hibernate/Hibernate Configuration File'. Aquesta opció està disponible gràcies a que ens em instal·lat l'Hibernate Synchronizer.

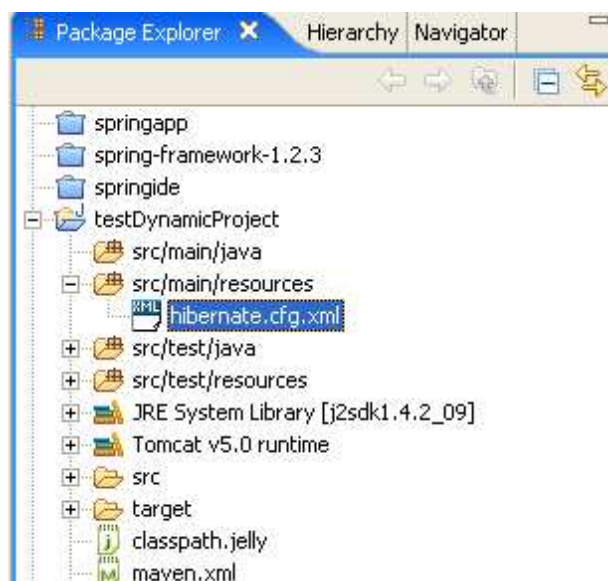
Se'ns obrirà una finestra a on tenim que informar:

- Container: ubicació a on situarem el fitxer que es crearà. Aquesta ubicació tindrà que ser el directori de 'resources' del nostre projecte. Mitjançant el botó 'Browse' podem seleccionar aquesta destinació amb facilitat. Típicament serà '*\project_name\src\main\resources*'
- File name: nom del fitxer. El valor per defecte, 'hibernate.cfg.xml', ja ens està bé.
- Session Factory Name: nom sota el que podem trobar la factoria de sessions hibernate en un arbre JNDI.



- Database Type: tipus de base de dades que farem servir.
- Application Server: nom del servidor d'aplicacions sota el que despleguem l'aplicació. En entorns de desenvolupament no fa falta informar aquest valor. Això afecta a la transaccionalitat de les nostres operacions, ja que hibernate ens proporciona diferents classes per a buscar les transaccions en el contenidor J2EE. El valor 'N/A' ja ens està bé, la qual cosa implica que és el propi driver JDBC qui implementa la transaccionalitat.
- Local: si estem en entorns de desenvolupament lo més normal és obtenir connexions a la BD mitjançant una cadena de connexió i un driver adequat. En aquest cas les dades a informar són:
 - Driver Class: nom de la classe que fa de 'driver' per a la connexió amb la BD. Depèn de la BD que fem servir. Per exemple, per MySQL seria 'com.mysql.jdbc.Driver'. Mitjançant el botó 'Browse' podem localitzar amb facilitat aquesta classe. És important però que les llibreries que contenen els 'drivers' estiguin en el classpath del nostre projecte.
 - Database URL: cadena de connexió del a BD. Depèn de la BD que fem servir. Per exemple, per MySQL seria algo com 'jdbc:mysql://localhost/test'
 - Username: usuari per la BD
 - Password: contrasenya per la BD
- Datasource: es possible que les connexions a la BD les obtenim d'un datasource. En aquest cas les dades a informar són:
 - Name: nom JNDI a on podem trobar el 'datasource'
 - JNDI URL: no es necessari especificar cap valor
 - JNDI Class: no es necessari especificar cap valor
 - Username: no es necessari especificar cap valor
 - Password: no es necessari especificar cap valor

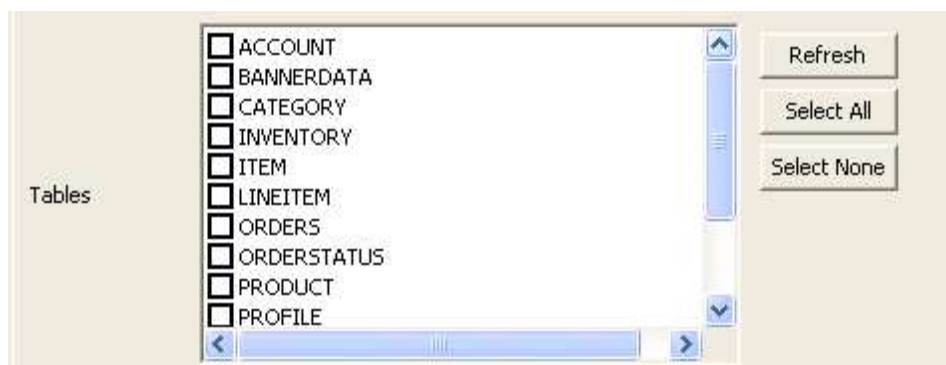
Cliquem a 'Finish' i ja tenim el fitxer 'hibernate.cfg.xml' al directori que li hem dit:



Aquest és només el primer pas en la configuració del nostre motor de persistència. Realment falta afegir la informació relevant: els 'mappings' dels nostres objectes de negoci. Cada taula presenta (en la majoria de casos) un objecte de negoci i té associat un fitxer de mapping. Aquest fitxer es diu igual que el nom de la taula (però amb la convenció Java) i té extensió '.hbm.xml'. Anem a crear doncs aquests fitxers. En el menú 'File/New/Other' seleccionem 'Hibernate/Hibernate Mapping File'. Aquesta opció està disponible gràcies a que ens em instal·lat l'Hibernate Synchronizer. Se'ns obrirà una finestra a on hem d'informar dues pestanyes: 'Configuration' i 'Properties'. En la primera informem:

- Container: ubicació a on situarem els fitxers que es crearan. Aquesta ubicació tindrà que ser el directori de 'resources' del nostre projecte i subdirectori el nom del package a on crearem els objectes de negoci però en format 'folders'. Per exemple, imaginem que volem crear els objectes de negoci en un package que es dirà 'com.myapp.model'. El directori a on desarem els fitxers de mappings serà doncs '/project_name/src/main/resources/com/myapp/model'. Mitjançant el botó 'Browse' podem seleccionar aquesta destinació amb facilitat. Cal remarcar que aquest directori ha d'existir. Per tant, en el cas que no el tinguem, **el crearem abans de crear els fitxers de mappings**.
- Driver: nom de la classe que fa de 'driver' per a la connexió amb la BD. Depèn de la BD que fem servir. Per exemple, per MySQL seria 'com.mysql.jdbc.Driver'. Mitjançant el botó 'Browse' podem localitzar amb facilitat aquesta classe. És important però que les llibreries que contenen els 'drivers' estiguin en el classpath del nostre projecte. Per defecte ens apareix el que havíem informat al crear el fitxer de configuració 'hibernate.cfg.xml'
- Database URL: cadena de connexió del a BD. Depèn de la BD que fem servir. Per exemple, per MySQL seria algo com 'jdbc:mysql://localhost/test'. Per defecte ens apareix el que havíem informat al crear el fitxer de configuració 'hibernate.cfg.xml'

- Username: usuari de connexió a la BD. Per defecte ens apareix el que havíem informat al crear el fitxer de configuració 'hibernate.cfg.xml'
- Password: contrasenya de connexió a la BD. Per defecte ens apareix el que havíem informat al crear el fitxer de configuració 'hibernate.cfg.xml'
- Table pattern: patró de les taules que volem visualitzar. Això ens permet filtrar en el cas que hi haguessin moltes taules (per defecte blanc ja ens està bé)
- Schema pattern: patró del schema que volem visualitzar (per defecte blanc ja ens està bé)
- Package: nom del package a on volem desar els objectes de negoci. Aquest valor és important ja moltes classes es crearan a partir d'aquest valor. Es recomana un valor similar a 'net.myapp.model'
- Fent clic ara al botó 'Refresh' ens apareixeran (si no em comés cap error) les taules per a les quals podem generar mappings:

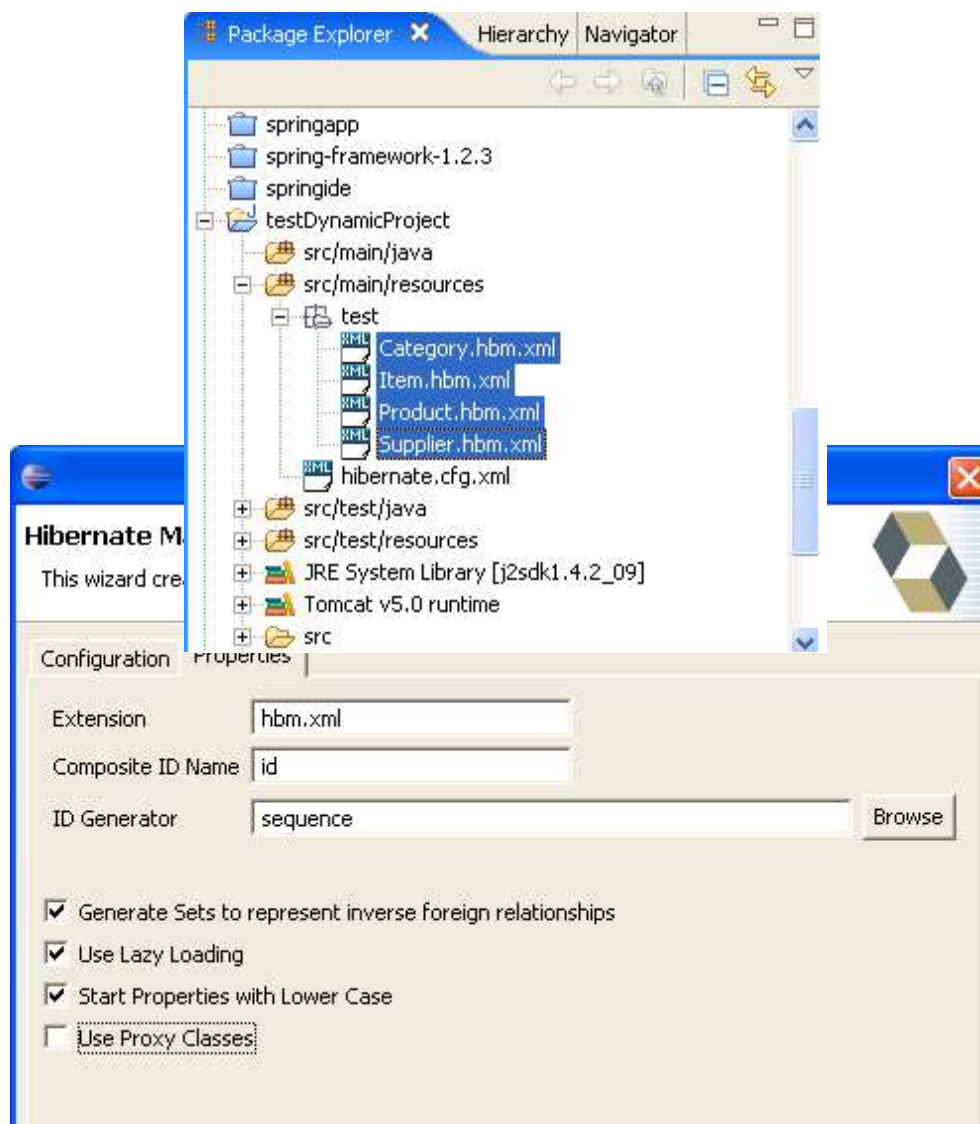


Seleccionem les que ens interessin.

En la segona pestanya, 'Properties', informem:

- Extension: extensió per defecte dels fitxers generats. Ja ens està bé 'hbm.xml'
- Composite ID Name: nom de la clau primària de les taules
- ID Generator: tipus de generador de claus primàries. Això està molt lligat a l'API d'hibernate. Podem consultar la següent URL en cas de fer ús d'un generador de claus primàries proporcionat per aquesta API http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#mapping-declaration-id o introduir la nostra classe que ens genera les claus primàries.
- Generate Sets to represent inverse foreign relationships: ho activem
- Use Lazy Loading: ho activem
- Start Properties with Lower Case: ho activem
- Use Proxy Classes: ho desactivem

Ara ja estem en condicions de generar els fitxers. Fem clic a 'Finish' i sens crearan els fitxers indicats per les taules que hem seleccionat:



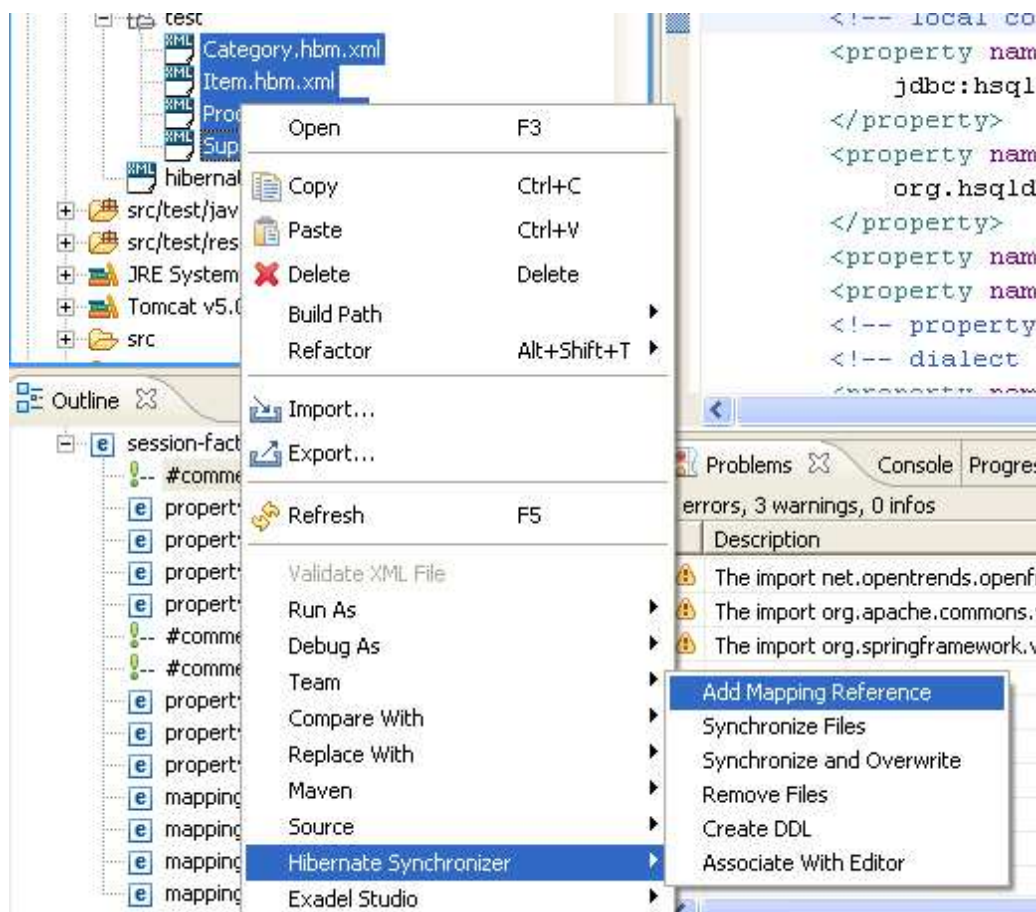
Ara tenim per una banda els fitxers de mapeig i per altra el fitxer de configuració d'hibernate. Però com els lliguem? És tan fàcil com seleccionar els fitxers de mapeig que volem incloure en el fitxer de configuració, obrir el menú contextual i seleccionar l'opció 'Hibernate Synchronizer/Add mapping reference'

Amb això aconseguim incloure al 'hibernate.cfg.xml' els fitxers seleccionats:

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
...
  <mapping resource="test/Category.hbm.xml" />
  <mapping resource="test/Item.hbm.xml" />
  <mapping resource="test/Product.hbm.xml" />
  <mapping resource="test/Supplier.hbm.xml" />
</session-factory>
</hibernate-configuration>
  
```



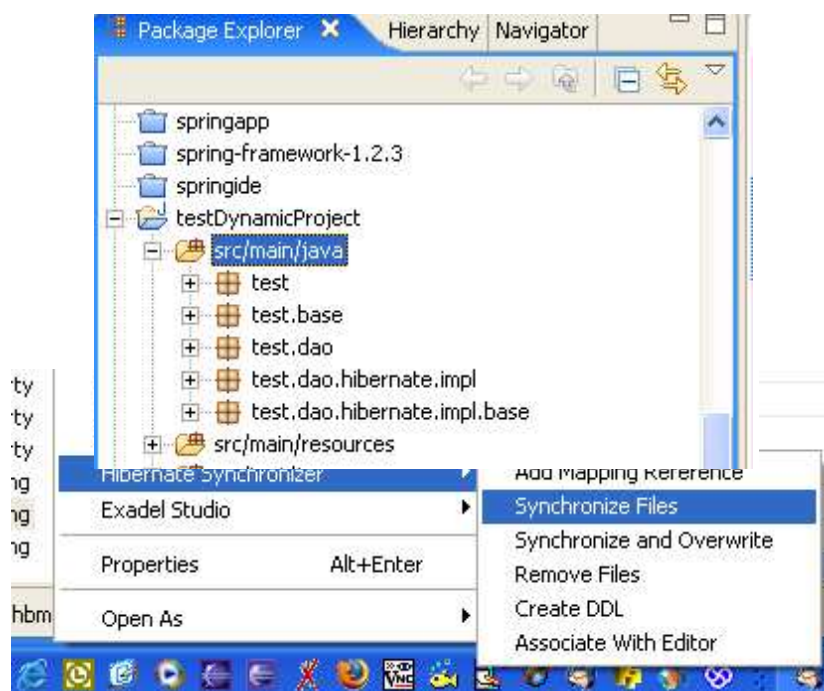
El següent pas en l'utilització del servei de dades es crear les següents classes:

- Interfícies DAO
- Implementacions DAO

- Objectes de negoci

Per a aquesta creació seleccionem les fitxers de mapeig i en el menú contextual seleccionem 'Hibernate Synchronizer/Sincronize Files':

Si tot va bé apareixan cinc directoris i l'estructura dels mateixos ha de ser algo similar a la figura següent:



A on veiem:

- *test*: package dels objectes de negoci
- *test.base*: package base dels objectes de negoci
- *test.dao*: interfícies dels DAO's
- *test.dao.hibernate.impl*: implementacions dels DAO's
- *test.dao.hibernate.impl.base*: implementacions base dels DAO's

Qualsevol canvi a la base de dades només implicaria una resincronització dels fitxers de mapeig per tornar a generar les classes implicades amb l'opció 'Synchronize And Overwrite'. S'ha de tenir en compte que una resincronització fa que es sobreescriguin els fitxers que ja tenim amb els nous.



2.5. Integració amb Altres Serveis

2.6. Preguntes Freqüents

3. Exemples

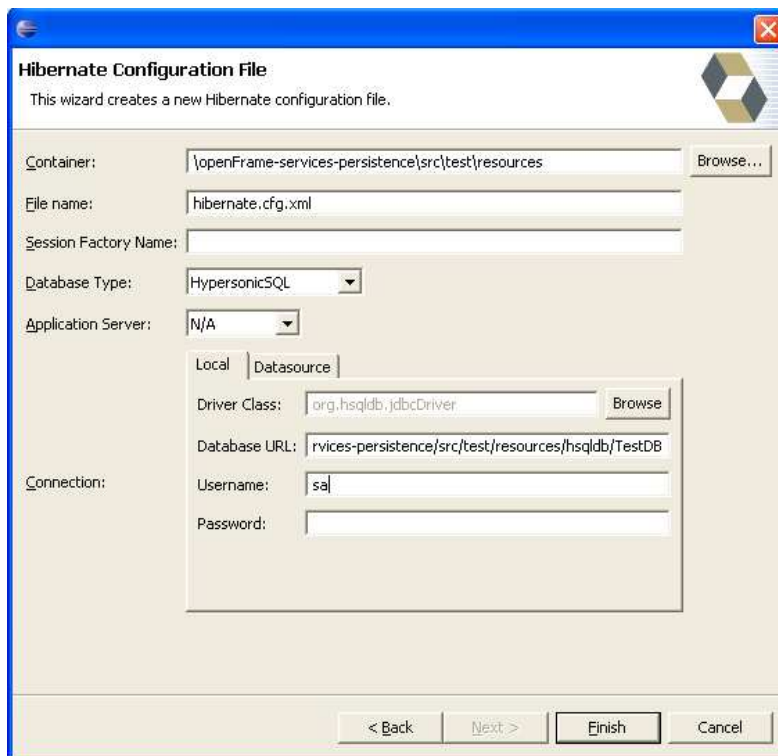
3.1. Test unitaris

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

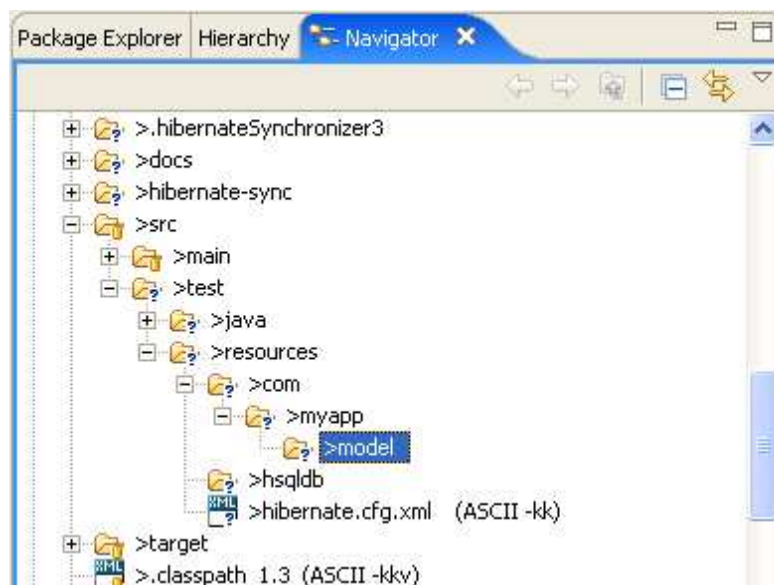
- Fitxer de configuració d'Hibernate (típicament 'hibernate.cfg.xml')
- Fitxers de mapeig de les taules que volem manejar (típicament '*.hbm.xml')
- Interfícies i implementacions dels DAO's.
- Objectes de negoci (també coneguts com 'Value Objects')
- Fitxer de configuració d'Spring (típicament applicationContext.xml)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



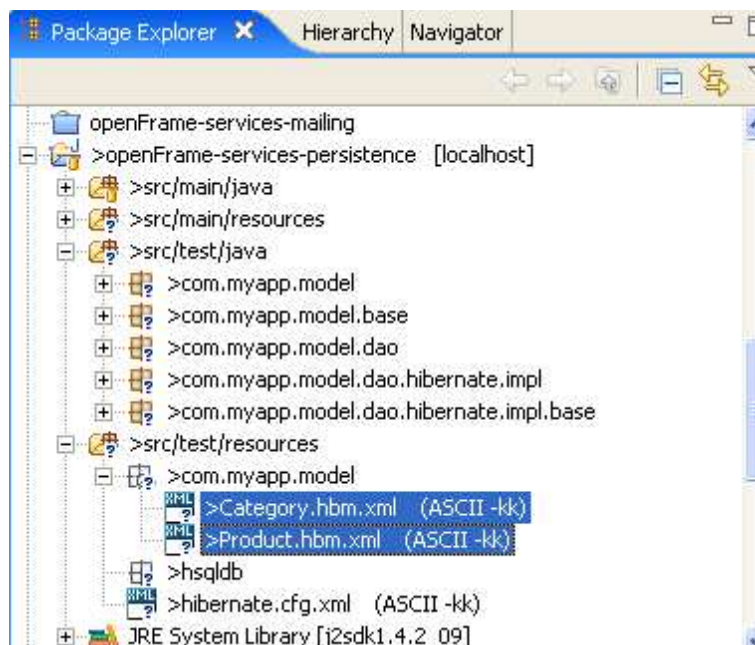
2. Crear els directoris a on situarem els fitxers de mapeig:
'/src/test/resources/com/myapp/model'



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">
```

Configuration Properties

Container: /openFrame-services-persistence/src/test/resources Browse

Driver: org.hsqldb.jdbcDriver Browse

Database URL: vices-persistence/src/test/resources/hsqldb/TestDB

Username: sa

Password:

Table pattern:

Schema pattern:

Tables

☒ CATEGORY

☒ PRODUCT

Refresh

Select All

Select None

Package: com.myapp.model Browse



```
<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 fictifi que fagi 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 tenim que 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 previament 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