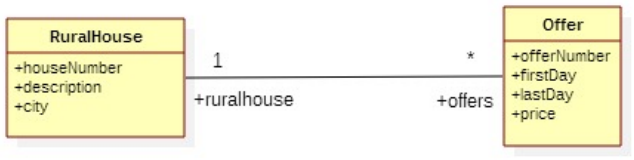
Hay que crear entonces los ficheros de enlace o mapping, la configuración e implementar la clase DAO (HibernateDataAccess).

Ayuda: creación de los ficheros de enlace o mapping

Para crear correctamente los ficheros de enlace o mapping, usando las facilidades explicadas en el laboratorio (New => Other => Hibernate => Hibernate XML Mapping File), hay que definir correctamente las siguientes clases del dominio: RuralHouse y Offer. Son prácticamente las mismas clases que las del proyecto wsRuralHouse-OCW. La representación en UML de las mismas es la siguiente:



En el enunciado ya se ha dicho que hay que quitar las siguientes anotaciones de las clases del dominio (Offer y RuralHouse):

```
@Entity
@Id
@GeneratedValue
```

Al definir las clases del dominio hay que tener en cuenta que Hibernate no funciona bien dando persistencia a atributos o propiedades que son de tipo Vector. En vez de Vector hay que usar colecciones Java (como Set, List,…). Ver: **https://forum.hibernate.org/viewtopic.php?f=6&t=932647**

Hay que tener en cuenta que la utilidades que generan el mapping de manera automática, consideran que el primer atributo de la clase es el atributo clave, y lo toma como clave primaria de la tabla correspondiente en la BD.

Hay que asegurarse de que todas las clases POJO tienen constructores vacíos, puesto que Hibernate los invoca, y lanzaría errores si no los encuentra.

Se incluye el inicio de la clase RuralHouse de wsRuralHouse-OCW

```
package domain;
import ...;
@XmlAccessorType(XmlAccessType.FIELD)
@Entity
public class RuralHouse implements Serializable {
	private static final long serialVersionUID = 1L;
	@XmlID
	@XmlJavaTypeAdapter(IntegerAdapter.class)
	@Id
	@GeneratedValue
	private Integer houseNumber;
	private String description;
	private String city;
	private Vector<Offer> offers;
...
```

Y el inicio de la nueva clase RuralHouse para ser mapeada por Hibernate (se han quitado las anotaciones de objectDB, el primer atributo de la clase (***serialVersionUID***), que no se necesita para que quede como atributo clave `houseNumber`, y se ha sustituido `Vector` por `Set`.

```
package domain;
import ...;
@XmlAccessorType(XmlAccessType.FIELD)
public class RuralHouse implements Serializable {
	@XmlID
	@XmlJavaTypeAdapter(IntegerAdapter.class)
	private Integer houseNumber;
	private String description;
	private String city;
	private Set<Offer> offers;
	...
```

Ayuda: definir la configuración de Hibernate

Tal y como se ha visto en el laboratorio de Hibernate, hay que crear el fichero de configuración hibernate.cfg.xml con el nombre de la base de datos MySQL (en este caso "ruralHouses", que debe ser creada; Hibernate no la crea de manera automática). Debe incluir los ficheros de mapping *domain/Offer.hbm.xml* y *domain/RuralHouse.hbm.xml*. Y también hay que declarar correctamente el nombre de usuario y password, y resto de opciones.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory name="">
  <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost/ruralHouses</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">admin</property>
  <property name="hibernate.connection.pool_size">1</property>
  <property name="hibernate.connection.autocommit">false</property>
  <property name="hibernate.format_sql">true</property>
  <property name="hibernate.show_sql">true</property>
  <property name="hibernate.hbm2ddl.auto">update</property>
  <property name="hibernate.current_session_context_class">thread</property>
  <mapping resource="domain/Offer.hbm.xml"/>
  <mapping resource="domain/RuralHouse.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

Se necesita también la clase `HibernateUtil` que configura y crea la instancia de `SessionFactory`, la cual permitirá trabajar con sesiones de la BD.

```
package dataAccess;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
   // Crea una instancia de SessionFactory con los datos de configuración de hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        }
        catch (Throwable ex) {
            System.err.println("Fallo creando el SessionFactory." + ex);
            throw new ExceptionInInitializerError(ex); }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory; }
}
```

Ayuda: probar que el enlace o mapping está bien

Antes de programar la clase HibernateDataAccess, se puede comprobar si los mappings de las clases del dominio y la configuración de la BD son correctos si se programa un método de inicialización de la BD. Además, nos permitiría introducir las instancias de RuralHouse que estaban en la BD objectDB anterior.

Ese método podría ser `initializeDB()`, se podría incluir en la propia clase HibernateDataAccess, y se podría ejecutar una vez invocándolo desde su método main, por ejemplo.

```
        private void initializeDB(){

                Session session = HibernateUtil.getSessionFactory().getCurrentSession();
                session.beginTransaction();

                List<Offer> resultO = session.createQuery("from Offer").list();
                for (Offer o: resultO){
                session.delete(o);
                }
                List<RuralHouse> resultRH = session.createQuery("from RuralHouse").list();
                for (RuralHouse rh: resultRH){
                session.delete(rh);

                session.getTransaction().commit()
                System.out.println("BD borrada");

                session = HibernateUtil.getSessionFactory().getCurrentSession();
                session.beginTransaction();

                RuralHouse rh1= new RuralHouse("Ezkioko etxea","Ezkio");
                session.save(rh1);

                RuralHouse rh2= new RuralHouse("Eskiatzeko etxea","Jaca");
                session.save(rh2);

                RuralHouse rh3= new RuralHouse("Udaletxea","Bilbo");
                session.save(rh3);

                RuralHouse rh4= new RuralHouse("Gaztetxea","Renteria");
                session.save(rh4);

                session.getTransaction().commit();

                System.out.println("BD inicializada");
        }
```

Ayuda: implementar la clase DAO (HibernateDataAccess)

Hay que implementar los métodos de `HibernateDataAccess`, que antes `DataAccess` implementaba utilizando objectDB, pues ahora utilizando Hibernate.

Entre los métodos que hay que implementar se encuentran los de la interfaz `ApplicationFacadeInterfaceWS`, que es la lógica del negocio. Suele ser habitual que para cada método de la lógica del negocio haya un método en una clase DAO de acceso a datos que lo implemente, pero puede que haya métodos en la lógica del negocio que no tengan su correspondiente en el acceso a datos, y viceversa.

A continuación se muestran la implementación de tres métodos de `HibernateDataAccess` donde se puede comprobar que se utilizan sentencias HQL (`session.createQuery`) o instrucciones de inserción, borrado, actualización o recuperación de objetos dado su identificador. En este caso una de inserción (`session.save`)

```
public Offer createOffer(RuralHouse ruralHouse, Date firstDay, Date lastDay, float price) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    Offer o= new Offer(firstDay, lastDay, price);
    o.setRuralHouse(ruralHouse);
    session.save(o);
    session.getTransaction().commit();
    return o;
}

public Vector<RuralHouse> getAllRuralHouses() {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    List<RuralHouse> resultRH = session.createQuery("from RuralHouse").list();
    Vector<RuralHouse> v = new Vector(resultRH);
    session.getTransaction().commit();
    return v;
}

public Vector<Offer> getOffers(RuralHouse rh, Date firstDay,  Date lastDay) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    SimpleDateFormat formateador = new SimpleDateFormat("yy-MM-dd");
    List<Offer> resultOf = session.createQuery("from Offer where firstDay>='"+
        formateador.format(firstDay)+"' and lastDay<='"+formateador.format(lastDay)+"'").list();
    Vector<Offer> v = new Vector(resultOf);
    session.getTransaction().commit();
    return v;
}
```

Ayuda: consultar bibliografía y diapositivas de Hibernate del curso

Hay que asegurarse de que se entiende bien cómo funciona algunas características y atributos de los mapping. En particular los que corresponden a relaciones entre clases (one-to-many, many-to-many, one-to-one) y los atributos “lazy”, “fetch”,

“cascade” e “inverse”. Se puede consultar las diapositivas de este curso OCW, en el apartado de Hibernate, y también internet donde se puede encontrar abundante información como por ejemplo:

- **http://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example/**
- **http://www.adictosaltrabajo.com/tutoriales/hibernate-join/#4.1.%20Lazy|outline**
- **http://cursohibernate.es/doku.php?id=unidades:03_relaciones:06_cascade**
- **http://www.mkyong.com/hibernate/inverse-true-example-and-explanation/**