TP HIBERNATE

Résumé

L'objectif de ce TP est de mettre en place et d'utiliser le framework Hibernate dans une application Java de **gestion de location de véhicules**,voiture, camion ou moto. Il s'agit de gérer des individus et un parc de véhicules. L'application devra permettre, entre autres, de :

— de lister les véhicules du parc
— de lister les individus ayant loué un véhicule
— de permettre à des individus d'emprunter et de rendre des véhicules

1 Pré-requis

1.1 L'environnement de développement

Pour ce TP, l'environnement de développement utilisé est Netbeans, entre autre pour le fait que l'IDE intègre le moteur de base de données Derby. Cet environnement est disponible dans les salles de TP de Galilée. Pour le chapitre JDBC, il est nécessaire d'utiliser la base Derby intégrée à Glassfish. Dans tous les cas, le lecteur pourra trouver une version Linux de ces applications à cette adresse : http ://www-lipn.univ-paris13.fr/ fortier/Enseignement/Applications/

1.2 Les librairies nécessaires

Les librairies nécessaires sont téléchargeables à cette adresse :

http ://www-lipn.univ-paris13.fr/ fortier/Enseignement/Hibernate/TP/

Pour utiliser les outils de génération (configuration, Reverse Engineering) via l'IDE, il faut que le plugin "Hibernate" soit installé dans Netbeans. L'installation se fait simplement par le menu "Tools → Plugins".

1.3 La base de données

On utilise une base de données derby, installée avec Glassfish. Avant tout, il faut créer la base de nom "location", l'utilisateur "location" et comme mot de passe "mdp" (par exemple) Voila les requêtes SQL de base à exécuter pour créer les schéma et la première table :

base.sql

```
create schema location;
create table location.personne (
        personne_id integer generated always as identity primary key,
        nom varchar(20),
        age numeric
);
insert into location.personne (nom,age) values ('michael', 25);
insert into location.personne (nom,age) values ('christophe', 30);
```

1.4 Tests unitaires

Les exercices peuvent être testés via des classes Main, mais on préfèrera utiliser des tests unitaires, déjà intégrés dans l'environnement Netbeans (Cf. Annexes).

2 Premiers pas

Dans ce paragraphe, nous créons une **application Java** sous Netbeans qui se connecte à une **base de données Derby**.

Nous manipulons du **HQL** pour toutes les opérations effectuées sur de sobjets persistants.

2.1 Création et initialisation du projet

— Créer la base de données Derby
— Créer un projet Java, sans utiliser les librairies Hibernate de Netbeans
— Dans les propriétés du projet, ajouter toutes les librairies Hibernate nécessaires
— Créer le fichier de configuration d'Hibernate (News → Other → Hibernate Configuration)
— Ajouter l'option de visualisation du SQL dans les propriétés
— Vérifier et comparer la configuration du cours

2.2 Génération des mappings

— Créer le fichier de Reverse Engineering hibernate.reveng avec l'assistant Netbeans (sélectionner la table personne)
— Créer le fichier de mapping à partir de la base de données en choisissant hibernate.reveng.xml et hibernate.cfg.xml
— Vérifier le fichier de mapping et le fichier de classe créés

2.3 Génération des requêtes HQL

— Compiler le projet
— Ouvrir l'éditeur HQL via click droit sur hibernate.cfg.xml
— Entrer la requête : from entite.Personne
— Vérifier le résultat, analyser le SQL
— Tester les clauses "where", "order by"
— Ecrire les requêtes de modification et de suppression de personnes

2.4 Tests d'implémentation

— Créer une classe de tests JUnit (à défaut une classe Main.java)
— Pour chaque type de requête HQL précédente, créer une méthode de tests type :

TP/HIBERNATE/test/HibernateTest.java

```java
@Test
public void q2_test_select(){
    String hql = "from entite.Personne";
    List resultList=null;
    SessionFactory sf = new Configuration().configure("q2/hibernate.cfg.xml").
                                buildSessionFactory();
    Session session = sf.openSession();
    try {
        Transaction transaction=session.beginTransaction();
        Query q = session.createQuery(hql);
        resultList = q.list();
        transaction.commit();
    } catch (HibernateException he) {
        he.printStackTrace();
    }
    // Affichage des resultats
    for (Object o : resultList) {
        Personne personne = (Personne) o;
        System.out.println("nom : " + personne.getNom()+"   age : " + personne.getAge());
    }
}
```

— Créer une méthode permettant de tester l'insertion d'une personne

3 Relation One-to-One

Nous ré-utilisons dans cet exercice la table personne précédemment ainsi qu'une table adresse :

base.sql

```
create table location.adresse (
        personne_id integer,
        rue varchar(50),
        cp numeric,
        ville varchar(30),
        CONSTRAINT pkk primary key(personne_id)
);
```

— Créer les mappings d'objets (personne et adresse)
— Modifier la classe Personne pour déclarer la relation one-to-one avec la classe Adresse
— Créer une classe de test permettant :
  1. d'assurer le bon fonctionnement du lien bidirectionnel en fournissant une référence de l'objet Personne à l'instance de l'adresse.
     Ceci doit être fait manuellement car Hibernate ne prend pas en charge automatiquement les liens bidirectionnels.
  2. de sauvegarder la personne dans la base de données
— Vérifier les insertions et les correspondances entre les clés primaires

4 Many-to-One

Pour cet exercice, nous modifions :
— la table personne en ajoutant le champ adresse_id (référence à une adresse)
— la table adresse en modifiant la clé primaire personne_id en adresse_id

base.sql

```
create table location.adresse (
        adresse_id integer generated always as identity primary key,
        rue varchar(50),
        cp numeric,
        ville varchar(30)
);
```

— Modifier le mapping Personne précédent pour déclarer la relation many-to-one avec la classe Adresse
— (Penser à modifier le mapping Adresse précédent)
— Créer une classe de test permettant de vérifier que lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi

5 Many-to-Many

Pour cet exercice, nous intégrons la table vehicule ainsi que la table de jointure avec celle des personnes :

base.sql

```
create table location.vehicule (
        vehicule_id integer generated always as identity primary key,
        marque varchar(20),
        kilometrage numeric,
        modele varchar(20)
);
create table location.personne_vehicule (
        vehicule_id integer not null,
        personne_id integer not null,
        CONSTRAINT pk primary key(vehicule_id,personne_id)
);
```

— Modifier le mapping Personne précédent pour déclarer la relation many-to-many avec la classe Vehicule
— Générer le mapping Vehicule et la classe associée
— Créer une classe de test permettant de vérifier qu'une personne peut louer plusieurs véhicules simultanément

6 Application

— Ecrire les requêtes HQL permettant de gérer les locations
— Récupérer et lancer le projet contenu dans l'archive Application.tar.gz
— Fusionner les deux projets
— Ecrire les classes de gestion des locations (emprunts, retours...)

7 Annexes

7.1 Tests unitaires - JUnits

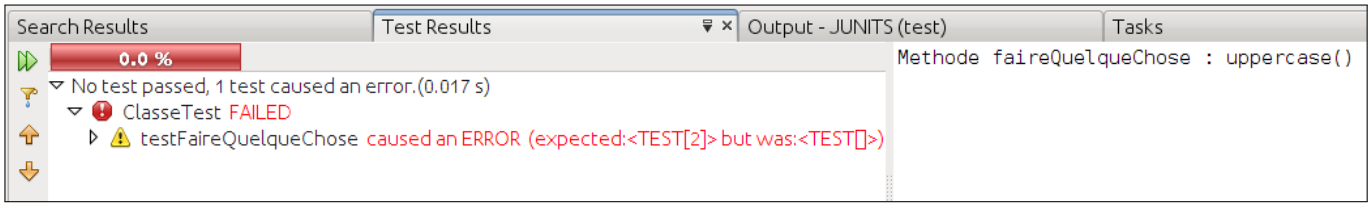
Un test unitaire est une méthode permettant de tester une partie code (appelé unité). Le test consiste a vérifier que le retour de l'exécution de la portion de code correspond a ce que l'on attend.
Il est exécuté indépendamment du programme en créant un environnement d'exécution spécifique à la portion de code. Cela permet d'eviter de devoir recompiler tout un logiciel pour vérifier la correction d'un bug des fois assez difficile à déclencher "manuellement" ou pour tester des fonctionnalités profondes dans le logiciel.

JUnits est un framework pour générer des tests unitaires qui est intégré à Netbeans.
Pour générer un test, il suffit de cliquer "Outils" → "Create Junits Tests" (sélectionnez Junit 4.x), puis de choisir la classe contenant les méthodes à tester.
Tous les tests de chaque méthode sont créés. Il suffit à présent de modifier ces tests pour vérifier la validité des méthodes.
«TP/JUNITS/test/ClasseTest.java -write -chunk hide»= import org.junit.After ; import org.junit.AfterClass ; import org.junit.Before ; import org.junit.BeforeClass ; import org.junit.Test ; import static org.junit.Assert.* ;
public class ClasseTest
public ClasseTest()
@BeforeClass public static void setUpClass() throws Exception
@AfterClass public static void tearDownClass() throws Exception
@Before public void setUp()
@After public void tearDown() @

TP/JUNITS/test/ClasseTest.java

```
<<TP/JUNITS/test/ClasseTest.java -write -chunk verb>>=
    @Test
    public void testFaireQuelqueChose() {
        System.out.println("Methode faireQuelqueChose : uppercase()");
                // Creation d'une instance de la classe testee
        Classe instance = new Classe();
        // Initialisation du resultat attendu
        String expResultat="TEST2";
        // Lancement du test de la methode
        String resultat = instance.faireQuelqueChose("test");
        // Resultat du test de la methode faireQuelqueChose
        assertEquals(expResultat,resultat);
    }
@
```

«TP/JUNITS/test/ClasseTest.java -write -chunk hide»= @



7.2 Installer Netbeans et Glassfish

Dans le cas où l'on se trouve sur une machine ne disposant pas de Netbeans ou Glassfish, il peut suivre les procédures suivantes qui lui permettront d'effectuer le TP.
Pour Netbeans, décompiler l'archive située à :

http ://www-lipn.univ-paris13.fr/ fortier/Enseignement/Applications/netbeans-6.7.1_fr.tar.gz

L'application se lance avec le commande bin/netbeans (pensez à régler les droits d'execution). Pour Glassfish, charger

l'archive disponible :

http ://www-lipn.univ-paris13.fr/ fortier/Enseignement/Applications/glassfish2.tar.gz

Une fois l'archive décompilée, entrez les commandes pour effectuer l'installation :
- cd glassfish2
- chmod 755 install.sh
- ./install.sh