TP 3 : Base de données

Pascal GRAFFION
2014/04/11 09:29

Table des matières

TP 3 : Base de données

JDBC (Java Data Base Connectivity) est une couche logicielle standard offerte aux développeurs, en d'autres termes, une API permettant l'accès à des bases de données relationnelles à partir du langage Java. Elle se charge de trois étapes indispensables à l'accès des données :

- La création d'une connexion à la base.
- L'envoi d'instructions SQL.
- L'exploitation des résultats provenant de la base.

Cette API fait partie intégrante de la plate-forme java depuis la version 1.1 du JDK. Elle est représentée par le paquetage java.sql.

JDBC s'appuie sur des drivers (pilotes) spécifiques à un fournisseur de SGBDR ou sur des pilotes génériques en provenance de tierce partie. Les pilotes JDBC sont classés en quatre catégories :

- Les pilotes de pont JDBC/ODBC : ce type de pilote, fourni avec le J2SE (Java 2 Standard Edition), permet de convertir les appels JDBC en appels ODBC (Open Database Connectivity). ODBC est fourni en standard sur les plates-formes Windows et permet la connexion à des bases de natures diverses (Access, SQL Server, Oracle, ...).
- Les pilotes de type 2 : ils sont écrits, en partie, en Java, mais dépendent malgré tout de code natif. En termes de portage d'application Java, ces types de pilotes présentent quelques lacunes.
- Les pilotes de type 3 : ce type de pilote passe par un autre pilote JDBC intermédiaire (fréquemment de type 1 ou 2). Il s'agit certainement du type de pilote le moins utile des quatre proposés.
- Les pilotes de type 4 : leur code est écrit en 100% Java, donc portable. Ces pilotes sont fournis par presque tous les constructeurs de bases de données. Chaque pilote étant adapté à une base de données particulière. Ces drivers constituent la meilleure solution pour le portage des applications Java.

Hello PetStore !

Dans cet exemple, la classe HelloPetstore ira insérer les chaînes de caractère « Hello » et « PetStore! » dans une table en base de données. Ensuite, elle pourra accéder à cette table et en récupérer le contenu. Tout d'abord, vérifiez que la base de données MySQL est installée et fonctionne. Ensuite que la base **petstoreDB** existe ainsi que la table HELLO_PETSTORE. Si ce n'est pas le cas, créez la avec la ligne de commande suivante :

```
CREATE TABLE HELLO_PETSTORE(cle VARCHAR(10), valeur VARCHAR(50) NOT NULL);
```

Pour exécuter la classe HelloPetstore, vous devez ajouter le fichier .jar contenant le driver JDBC dans le classpath.

```
import java.sql.*;


public class HelloPetstore {


public
static void main(
String[] args) {
        // Charge la classe du driver JDBC

try {

Class.forName(
"com.mysql.jdbc.Driver");
        }
catch (ClassNotFoundException e) {
            e.printStackTrace();
        }


if (args[0].equals(
"ecrire")) {
            ecrire(); (1)
        }
else {
```

```
            lire();
        }
    }


private
static void ecrire() { (2)
        Connection connection =
null;
        Statement statement =
null;


try {
            // Récupère une connexion à la base de données
            connection = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/petstoreDB",
"root",
""); (3)
            statement = connection.createStatement();

            // Insert une ligne en base
            statement.executeUpdate(
"INSERT INTO HELLO_PETSTORE (cle, valeur) VALUES ('Hello', 'PetStore !!!')"); (4)

        }
catch (Exception e) {
            e.printStackTrace();
        }
finally {
            // Ferme la connexion

try {

if (statement !=
null) statement.close();

if (connection !=
null) connection.close();
            }
catch (Exception e) {
                e.printStackTrace();
            }
        }
    }


private
static void lire() { (5)
        Connection connection =
null;
        Statement statement =
null;
        ResultSet resultSet =
null;


try {
            // Récupère une connection à la base de données
            connection = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/petstore",
"root",
""); (6)
            statement = connection.createStatement();

            // Récupère une ligne de la base
            resultSet = statement.executeQuery(
"SELECT valeur FROM HELLO_PETSTORE WHERE cle = 'Hello' "); (7)

if (!resultSet.next())

System.out.println(
"Non trouvé !!!");

            // Affiche le résultat

System.out.println(
```

```
"Hello " + resultSet.getString(1));

        }
catch (Exception e) {
            e.printStackTrace();
        }
finally {
            // Ferme la connexion

try {

if (resultSet !=
null) resultSet.close();

if (statement !=
null) statement.close();

if (connection !=
null) connection.close();
            }
catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Si vous exécutez ce programme en passant le paramètre « ecrire » (1) la méthode privée ecrire est appelée (2). Après avoir obtenu une connexion à la base de données (3), elle insère une chaîne de caractère (4) dans la table HELLO_PETSTORE. La méthode lire (5) quant à elle, se sert d'une connexion à la base (6) pour lire les données de la table (7).

Outils

Pour limiter les frais de l'entreprise YAPS, la base de données open source MySQL est utilisée. Après installation du logiciel, il faut créer la base PetStore :

Dans le répertoire %MYSQL_HOME%/bin tapez la commande suivante :

```
mysql -u root
```

Vous entrez alors en mode script et pouvez saisir différentes commandes. Entre autres, si vous tapez la commande help vous aurez l'aide du langage.

En tapant **show databases;** (n'oubliez pas le ';' à la fin de chaque commande) vous aurez la liste des bases de données gérées par MySQL. Pour créer la base petstoreDB il vous suffit de saisir : **create database petstoreDB ;** ou si elle existe déjà **drop database if exists petstoreDB** ; pour la supprimer.

Expression des besoins

La couche de persistance est remise en question, il serait plus simple d'utiliser une base de données. La solution de facilité serait d'avoir une base de données sur le poste de Bill et une autre sur le poste de John. Malheureusement, ces deux postes ne sont pas très puissants et utilisent d'autres logiciels gourmands en mémoire. YAPS décide d'investir dans un petit réseau local et d'avoir un poste distant accueillant la base de données. Ainsi, les deux applications PetStore Customer et PetStore Catalog utiliseront la même base de données distante.

Les besoins utilisateurs restent inchangés puisque cette nouvelle évolution est purement technique et non fonctionnelle. Les applications Petstore Customer et Petstore Catalog doivent continuer à fonctionner comme elles le faisaient auparavant ainsi que les tests de recette.

Par contre, Bill voudrait qu'on apporte une évolution fonctionnelle sur l'application Petstore Customer : afficher la liste de tous les clients.

Vue Utilisateur

Diagramme de cas d'utilisation

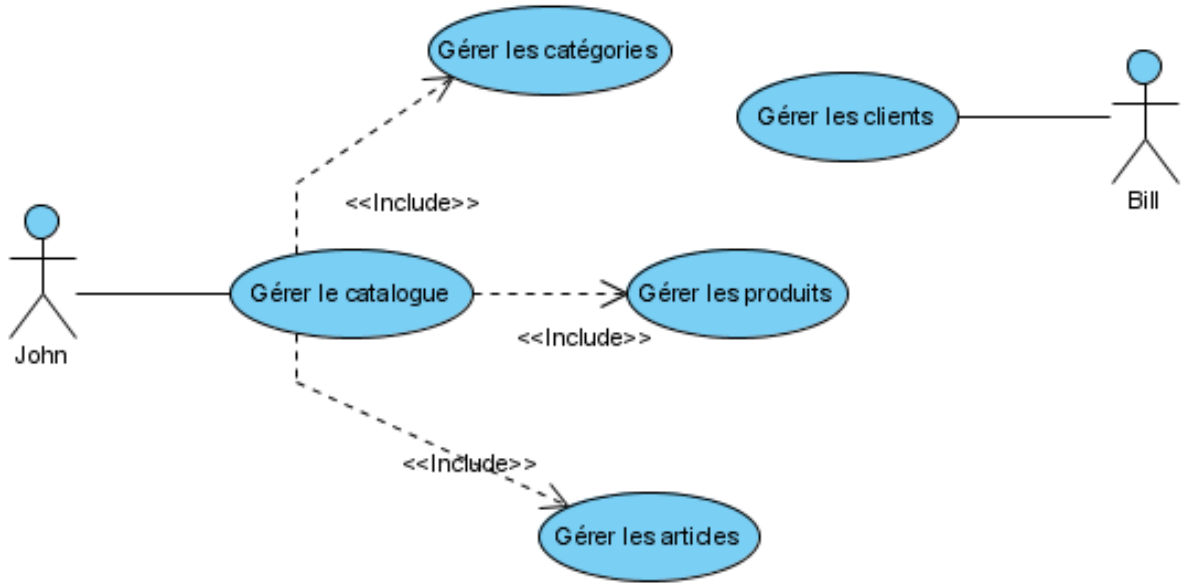

Figure 1 - Diagramme de cas d'utilisation inchangé

Le cas d'utilisation « Gérer les clients » regroupe les quatre cas d'utilisation « Créer un client », « Supprimer un client », « Mettre à jour les informations d'un client » et « Rechercher un client ». La granularité des cas d'utilisation est différente de celle de la première version de l'application bien que son contenu soit identique. Ce regroupement est souhaitable pour des cas d'utilisations simples.

Analyse et conception

Vue logique

La livraison de cette nouvelle version de PetStore nous permet de faire quelques remaniements au niveau des objets métier. Comme on peut le voir sur le diagramme de classe de la figure 2, l'attribut identifiant (id), présent dans chaque classe métier précédemment, a été déplacé dans la super-classe DomainObject.

Les classes du domaine possèdent toutes un identifiant (id). Le remaniement Extraire Super Classe (Extract Superclass) permet de créer des classes mères et d'y déplacer le code des classes filles.

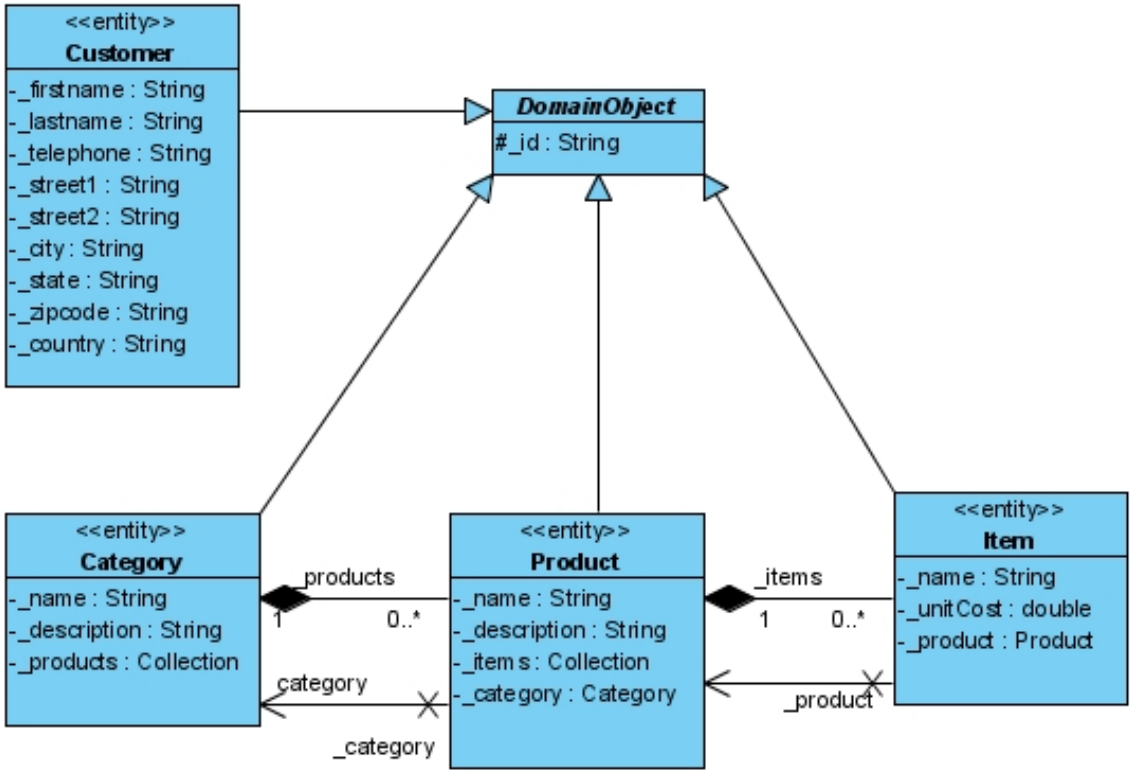

Figure 2 - Diagramme de classes des objets métiers

La couche de persistance doit être complètement remodelée puisque la sérialisation a été abandonnée au profit d'une base de données. Pour ce faire, nous allons utiliser le design pattern Data Access Object (DAO).

Le design pattern Data Access Object permet de centraliser dans un objet dédié le lien entre la couche d'accès aux données et la couche métier (constituée alors de POJOs). Le DAO peut utiliser une base de données, un fichier texte, une base objet ou même un serveur LDAP. Les applications PetStore Customer et Catalog persisteront leurs données dans une base de données relationnelle uniquement.

Ci-dessous, le diagramme de classes représentant les liens entre les objets métiers (Customer, Category, Product et Item) et leurs DAO respectifs (CustomerDAO, CategoryDAO, ProductDAO et ItemDAO). La superclasse AbstractDataAccessObject possède aussi une relation d'utilisation avec la superclasse DomainObject.

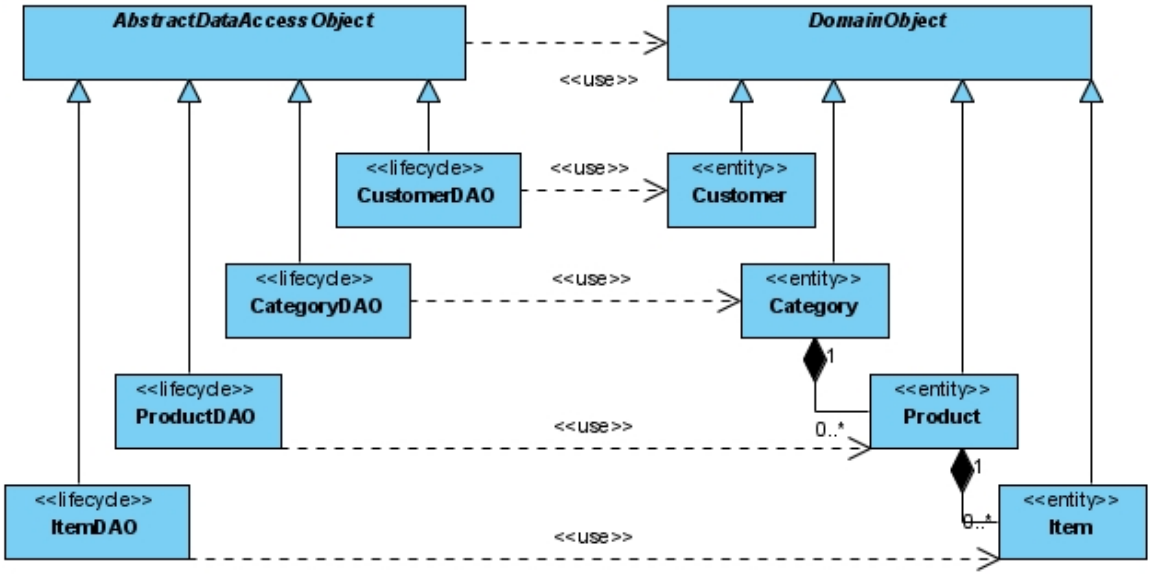

Figure 3 - Diagramme de classe représentant les liens entre objets du domaine et DAO

Le code SQL n'est bien sûr présent que dans les DAO concrets (CustomerDAO, CategoryDAO, ...) ou leur superclasse AbstractDataAccessObject.

Puiqu'à présent nous utilisons une base de données, il est plus approprié de nommer la méthode de recherche select ou findByPrimaryKey au lieu de tout juste find. Pour cela nous utilisons le remaniement Renommer Méthode (Rename Method).

Vue Processus

Le diagramme de séquence ci-dessous liste les différents appels aux objets pour pouvoir afficher un produit (Product). Dans les versions précédentes de l'application, nous utilisions la sérialisation. Bien que celle ci puisse poser des problèmes, elle présente quand même quelques avantages. Le plus important est que lorsqu'on sérialisait un produit lié à sa catégorie, la grappe entière d'objets était sérialisée et donc, par la suite, facilement utilisable. Il suffisait de charger l'objet produit et, lors de son affichage, les informations de la catégorie étaient automatiquement affichées.

Une base de données relationnelle ne gère pas les liens entre objets de la même façon. Elle utilise le système de clé étrangère. Lorsqu'on revient dans le monde objet, une clé étrangère est inappropriée. Il faut trouver un mécanisme qui puisse charger un objet à partir de cette clé étrangère.

Dans le diagramme suivant, on se rend compte que la classe MenuCatalog doit faire plusieurs appels pour afficher le contenu d'un produit. Ainsi, un simple select (ou findByPrimareyKey) sur le ProductDAO ne ramènera pas l'objet catégorie lié, mais simplement sa clé étrangère. Il faudra donc faire un autre select sur le CategoryDAO pour obtenir toutes les attributs de la catégorie à laquelle appartient le produit sélectionné.

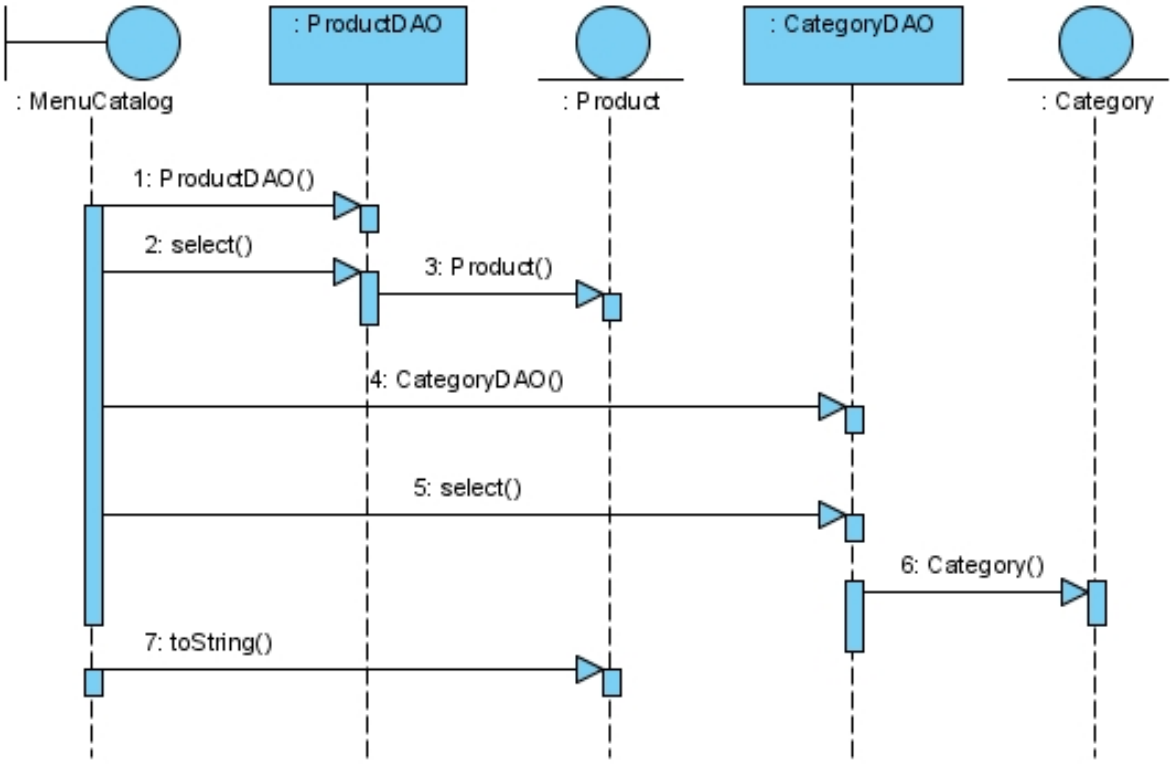

Figure 4 - Diagramme de séquence pour afficher un produit et sa catégorie liée

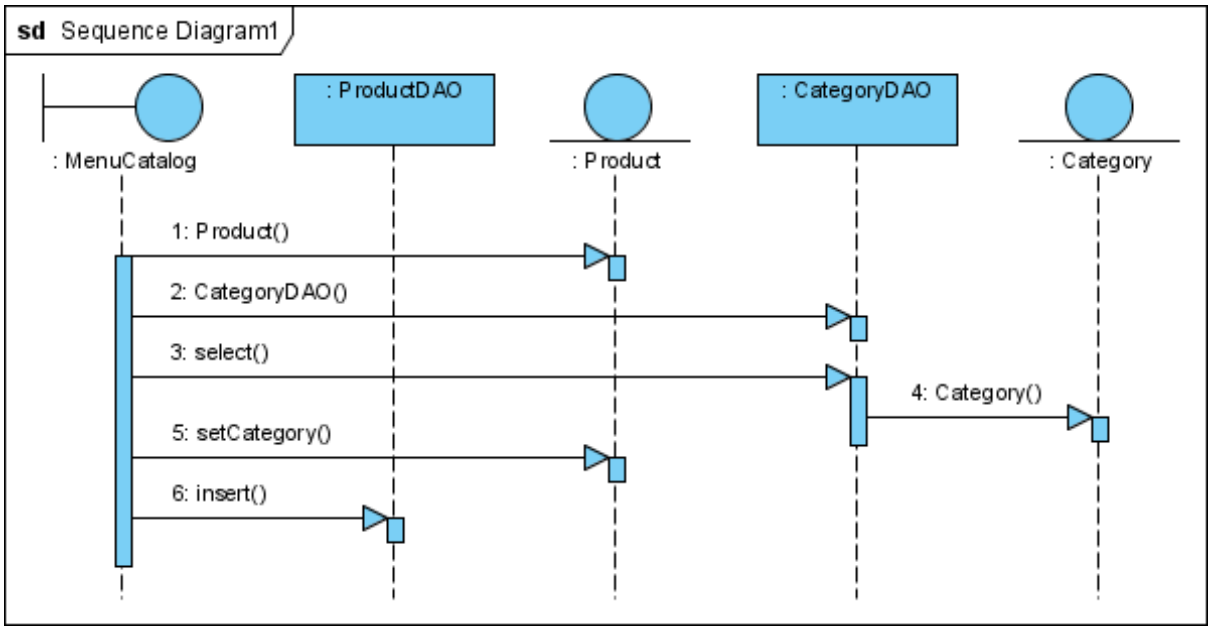

Figure 5 - Diagramme de séquence pour créer un produit

Vue implementation

L'application PetStore continue à augmenter son nombre de classes. En réutilisant les remaniements « Extraire Paquetage » et « Déplacer Classe » on se retrouve avec la structure suivante :

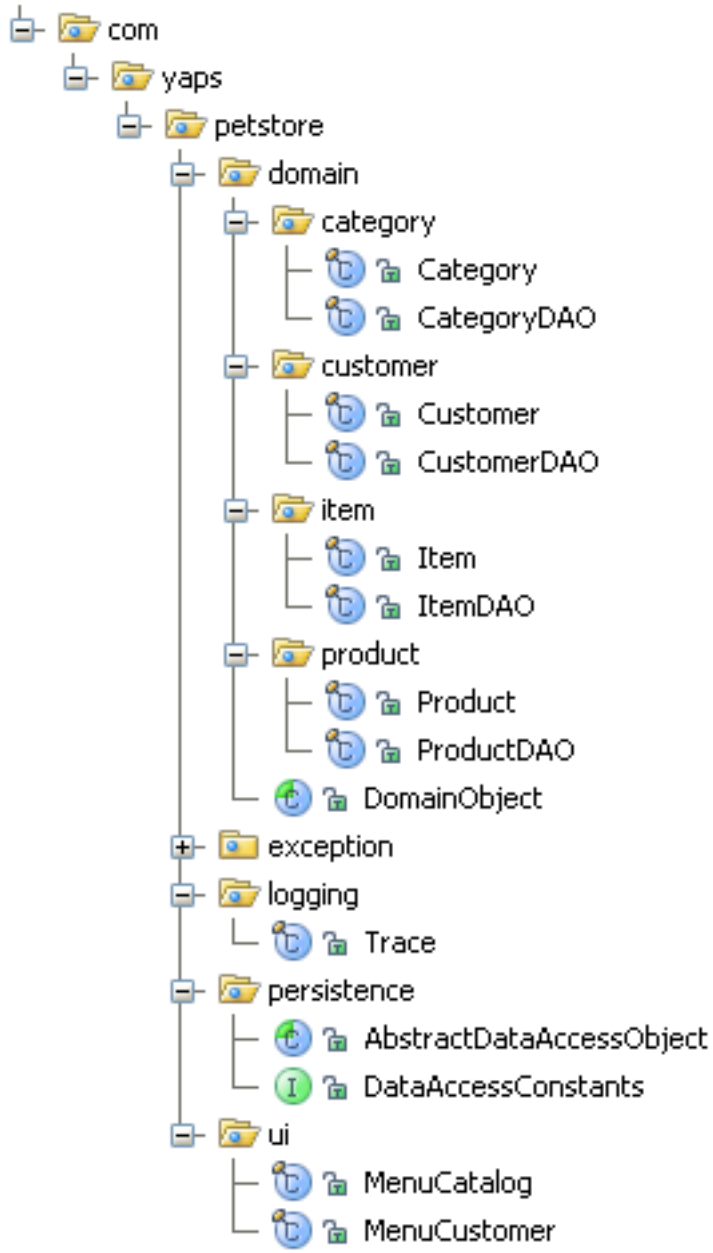

Figure 6 - Paquetages de l'application

Depuis le JDK 1.4 il existe une API de logging (paquetage java.util.logging). Cette API consiste à garder les traces des évènements survenus dans un système ou dans une application. Un ou plusieurs fichiers de log au format prédéfini sont générés en cours d'exécution et conservent des messages informant sur la date et l'heure de l'événement, la nature de l'événement et sa gravité par un code ou une description sémantique, éventuellement d'autres informations : utilisateur, classe, etc...

Pour loguer un message msg, il faut utiliser la fonction log() de l'objet Logger. L'argument Level définit le niveau de criticité du message msg passé en paramètre. Si ce niveau est géré, le message sera redirigé vers tous les flux de sortie associés au journal. Il existe plusieurs niveaux :

- SEVERE Niveau le plus élevé
- WARNING Avertissement
- INFO Information
- CONFIG Configuration
- FINE Niveau faible
- FINER Niveau encore plus faible
- FINEST Niveau le plus faible

Exemple :

```
logger.log(Level.WARNING,
"argument out of limit");
```

Cet exemple trace le message "argument out of limit" de niveau Level.WARNING.

Cette API permet aussi d'éviter l'utilisation de la méthode printStackTrace d'une exception. Bien que très utile, cette méthode affiche la trace d'une exception à l'écran, sans la garder dans un fichier. En utilisant la méthode throwing de l'API de logging, la trace de l'exception pourra être répertoriée comme un message, c'est-à-dire dans un fichier ou sur la console.

Les propriétés par défaut de logging peuvent être changées dans le fichier de configuration suivant : lib/logging.properties du répertoire du JRE de la JVM.

L'API de logging a été encapsulée dans la classe com.yaps.petstore.logging.Trace.

Architecture

Le diagramme de composants ci-dessous nous montre les trois grands sous-systèmes de l'application : l'interface utilisateur (composée de MenuCatalog et MenuCustomer) qui dialogue avec les objets du domaine qui, eux-mêmes, délèguent la persistance à une couche de DAO puis à la persistance en tant que telle, ici un SGBDR.

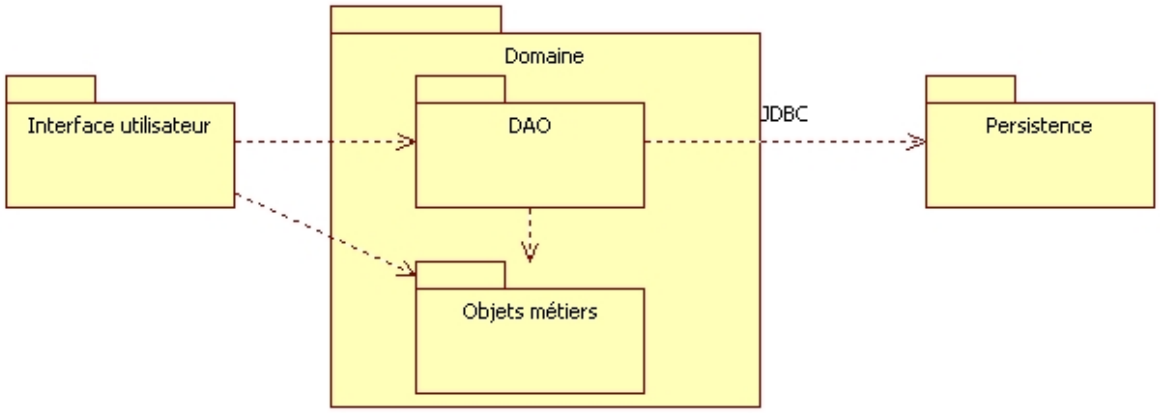

Figure 7 - Diagramme de composants montrant les sous-systèmes

Schéma de la base de données

Chaque classe métier devra persister ses données dans une table relationnelle :

Classe	Table
Customer	t_customer
Category	t_category
Product	t_product
Item	t_item

Remarquez ci-dessous la présence de clés étrangères. La relation 1..n entre les catégories et les produits se fait dans la table fille, c'est-à-dire que la table t_product possède une clé categoryid pointant vers la table t_category. Idem pour les tables t_product et t_item.

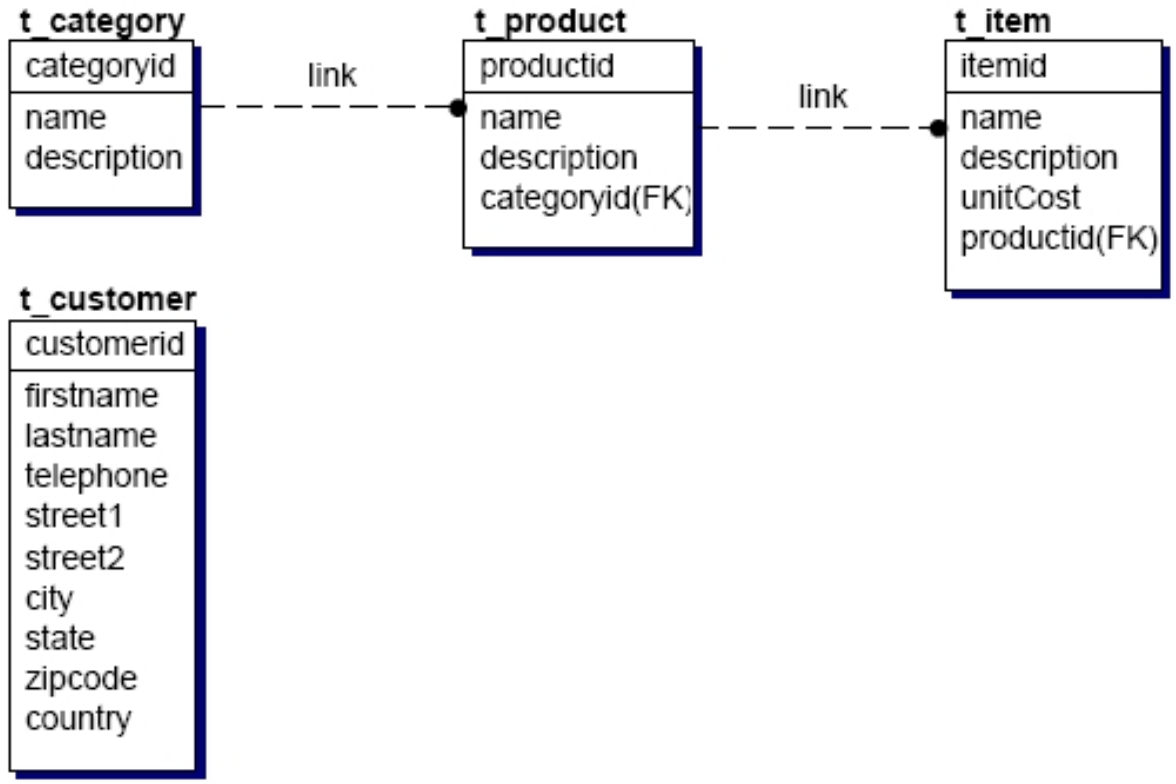

Figure 8 - Schéma de la base de données

Vue déploiement

Les applications PetStore Customer et PetStore Catalog, respectivement déployées sur les postes de Bill et John, utilisent toutes deux la base de données distante MySQL via JDBC.

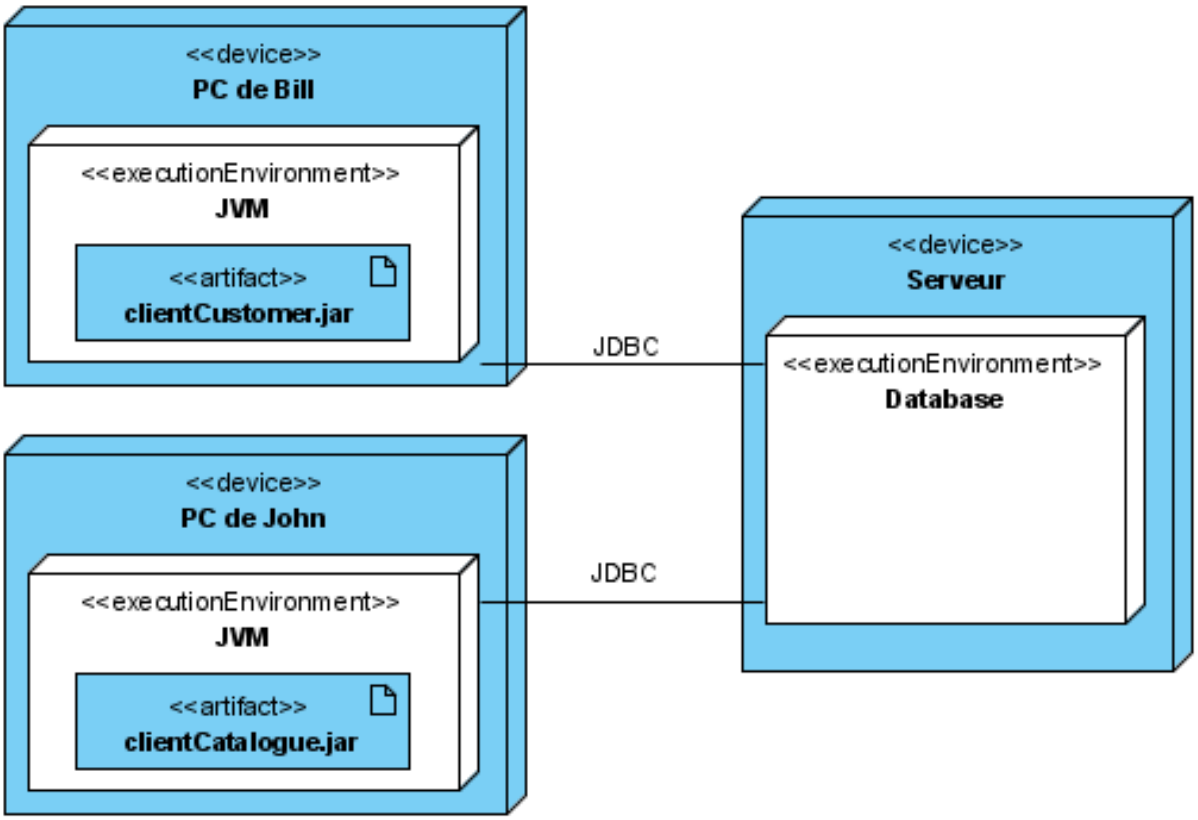

Figure 9 - Diagramme de déploiement des deux postes et de la base de données

Comme dans la version précédente, chaque application est déployée dans un fichier .jar différent : clientCatalog.jar et clientCustomer.jar se trouvant sous le répertoire %YAPS_PETSTORE%/build/. Pour que les applications clientes puissent fonctionner, elles doivent avoir les drivers JDBC de MySQL dans leur classpath.

Implémentation

Vous pouvez maintenant développer l'application à partir de la version précédente, ou télécharger la liste des classes fournies pour commencer votre développement. Dans les documents fournis, il y a la nouvelle classe MenuCustomer qui utilise la méthode findAll() pour afficher la liste des clients. Les hiérarchies de classe DomainObject, Customer et AbstractDataAccessObject, CustomerDAO vous sont données pour vous permettre de reproduire la même technique de développement pour les autres classes métiers. L'interface DataAccessConstant possède toutes les constantes d'accès à la base de donnée telles que le host, le port, le nom d'utilisateur… La classe utilitaire Trace vous permettra de rajouter des traces dans votre code si vous le souhaitez.

Dans le fichier build.xml vous trouverez de nouvelles taches Ant : yaps-create-db crée la structure des quatre tables et yaps-insert-data y insère des données. Une fois ces taches exécutées, retournez dans MySQL et utilisez la commande use petstoreDB qui vous permettra de sélectionner la base de données. Ensuite, vous pourrez en consulter le contenu (show tables ;) et effectuer n'importe quelle requête SQL (select * from t_category).

La commande desc vous permet de connaître la structure d'une table. Ainsi en tapant desc t_product ; vous obtenez le résultat suivant :

```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id          | varchar(10)  |      | PRI |         |       |
| name        | varchar(50)  |      |     |         |       |
```

```
| description | varchar(255) | YES  |     | NULL    |       |
| category_fk | varchar(10)  |      | MUL |         |       |
+-------------+--------------+------+-----+---------+-------+
```

Avertissement: Tous les liens représentés dans le diagramme de classe dans la partie Analyse ne sont pas nécessairement à implémenter. Ainsi lorsque le DAO retourne une Category, cette instance n'est pas peuplée pour l'instant avec tous ses Products. C'est un choix technique car cette opération a été jugée trop coûteuse … et inutile (pour implémenter les cas d'utilisation recensés)!

Recette utilisateur

La classe AllTests appelle les autres classes, c'est-à-dire CustomerTest, CustomerDAOTest, CategoryDAOTest, ProductDAOTest et ItemDAOTest.

Les tests de recette ont été enrichis pour prendre en compte la persistance en base de données. Par exemple pour la classe CategoryDAOTest, la méthode permettant de créer une catégorie se décompose comme suit : la méthode récupère un identifiant unique (1) et s'assure que la recherche d'une catégorie possédant cet identifiant échoue (2). Ensuite elle crée une nouvelle catégorie avec cet identifiant unique (3) et s'assure qu'elle peut la retrouver (4) par l'application mais aussi directement dans la base de données (5). La méthode s'assure ensuite que c'est bien la même catégorie (6). La méthode recrée une nouvelle catégorie avec le même identifiant et s'assure que le système renvoie bien une exception (7). Enfin, elle supprime la catégorie (8) et vérifie qu'il n'existe plus dans le système (9) ni dans la base de données (10).

```
public void testCreateCategory()
throws Exception {

final
int id = _dao.getUniqueId(); (1)
    Category category =
null;

    // Ensures that the object doesn't exist

try {
        category = findCategory(id);
        fail(
"
Object has not been created yet it shouldn't be found");
    }
catch (ObjectNotFoundException e) { (2)
    }

    // Creates an object
    createCategory(id); (3)

    // Ensures that the object exists

try {
        category = findCategory(id); (4)
    }
catch (ObjectNotFoundException e) {
        fail(
"
Object has been created it should be found");
    }

    // Ensures that the object exists in the database by executing a sql statement

try {
        findCategorySql(id); (5)
    }
catch (ObjectNotFoundException e) {
        fail(
"
Object has been created it should be found in the database");
    }
```

```
    // Checks that it's the right object
    checkCategory(category, id); (6)

    // Creates an object with the same identifier. An exception has to be thrown

try {
        createCategory(id);
        fail(
"An object with the same id has already been created");
    }
catch (DuplicateKeyException e) { (7)
    }

    // Cleans the test environment
    removeCategory(id); (8)


try {
        findCategory(id); (9)
        fail(
"
Object has been deleted it shouldn't be found");
    }
catch (ObjectNotFoundException e) {
    }


try {
        findCategorySql(id); (10)
        fail(
"
Object has been deleted it shouldn't be found in the database");
    }
catch (ObjectNotFoundException e) {
    }
}
```

Les méthodes findCategory, createCategory, updateCategory, … sont en l'occurence des méthodes privées de la classe de test; elles utilisent toutes une instance de CategoryDAO, la classe que l'on teste :

```
private void createCategory(
final
int id)
throws CreateException, CheckException {

final Category category =
new Category(
"cat" + id,
"name" + id,
"description" + id);
        _dao.insert(category);
    }


private void updateCategory(
final Category category,
final
int id)
throws UpdateException, ObjectNotFoundException {
        category.setName(
"name" + id);
        category.setDescription(
"description" + id);
        _dao.update(category);
    }
```

Résumé

Dans cette version, la couche de persistance a été complètement remodelée. De la sérialisation d'objets, YAPS est passée à une base de données relationnelle distante. Grâce à la couche d'accès aux données (DAO) on se rend compte que le code des interfaces utilisateurs n'a pas dû être modifié entre la précédente version et celle-ci.

Il existe d'autres technologies de persistance. Dans le monde de l'objet, il y a, bien sûr, les bases de données objets. Elles permettent non seulement de faciliter l'intégration avec les langages de programmation objet, mais surtout d'être bien plus rapides que leur homologue relationnel. Il faut cependant bien reconnaître que les bases de données relationnelles sont encore, de nos jours, la technologie de stockage prédominante. Le problème revient donc à construire une sorte de « pont » entre ces deux logiques objet et relationnel. C'est là qu'intervient JDBC et c'est pour cela que nous l'avons utilisé : pour persister des objets dans une base de données relationnelle.

Références

JDBC Technology http://java.sun.com/products/jdbc/

JDBC FAQ Home Page http://www.jguru.com/faq/JDBC

JDBC drivers in the wild http://www.javaworld.com/javaworld/jw-07-2000/jw-0707-jdbc.html

Database Programming with JDBC and Java George Reese. O'Reilly. 2000.

SQL Tutorial http://www.w3schools.com/sql/default.asp

MySQL Cookbook Paul DuBois. O'Reilly. 2002.

MySQL Home Page http://www.mysql.com/

Refactoring : Extract Superclass http://www.refactoring.com/catalog/extractSuperclass.html

Refactoring : Rename Method http://www.refactoring.com/catalog/renameMethod.html

Core J2EE Patterns - Data Access Object http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Tutoriel d'utilisation de l'API de logging java.util.logging http://www.programmez.com/tutoriels.php?tutoriel=79&titre=Logging-dans-Java