Programmation orientée objet en Java

Persistance des données

Dominique Blouin
Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

Objectifs d’apprentissage

- Qu’est-ce que la persistance des données
- La persistance des données en OO
- Framework de mapping objet-relationnel
- La norme JPA

Qu’est-ce que la persistance des données?

- Il arrive très souvent que les données des applications logicielles doivent être sauvegardées **d’une exécution à l’autre** de l’application.
- C’est ce qu’on appelle la **persistance des données** d’une exécution à l’autre du programme.
- Cela est vrai pour presque toutes les applications et ce peu importe leur langage de programmation.
- Cette sauvegarde se fait typiquement dans une mémoire permanente telle qu’un disque SSD ou disque dur par exemple.
- Elle peut prendre la forme d’un système de fichiers ou d’une base de données relationnelle (ou à base de graphe) par exemple.
- Cela implique souvent un **codage** / **décodage** des données.
- Cela peut être réalisé à l’aides des classes d’**entrées-sorties** introduites lors du cours précédent.

Comment implémenter la persistance des données dans un programme OO ?

- Exemple pour un modèle de labyrinthe :
- Question : est-ce une bonne pratique de programmation OO ?

```java
public final class Maze implements Graph {

    public final void initFromTextFile(final String fileName) {

    /**
     * enregistre un labyrinthe dans un fichier texte deja créé
     * @param fileName le nom du fichier dans lequel le labyritnhe sera enregistré
     * @throws FileNotFoundException
     */
    public final void saveToTextFile(final String fileName)
    throws FileNotFoundException {
        FileOutputStream fos = new FileOutputStream(fileName);
        PrintWriter pw = new PrintWriter(fos);

        try {
            for (int i=0;i<lines;i++) {
                for (int j =0;j<columns;j++) {
                    MazeBox boite = mazeboxList.get(columns*i+j);

                    if (boite.isEmptyBox()) {
                        pw.print("E");
                    }
                    else if (boite.isArrivalBox()) {
                        pw.print("A");
                    }
                    else if (boite.isDepartureBox()) {
                        pw.print("D");
                    }
                    else if (boite.isWallBox()) {
                        pw.print("W");
                    }
                }

                if (i!=lines-1) {
                    pw.println();
                }
            }
        }
        finally {
            pw.close();
        }
    }
```

Rappel : le principe responsabilité unique

- En programmation orientée objet, Robert C. Martin exprime le **principe de responsabilité unique** comme suit :
  - « une classe ne doit changer que pour une seule raison » (a class should have only one reason to change).

- Une classe ne devrait avoir qu'une seule responsabilité, clairement identifiée par le **nom** de la classe.

Data mapper pattern

- Problème avec la persistance codée dans les classes du modèle : il sera difficile de changer de type de stockage des données (fichier binaire, textuel, base de données, serveur distant, etc.) sans changer les classes du modèle.

- Solution: le data mapper pattern (Martin Fowler)

- Couche d'accès aux données qui **gère le transfert bidirectionnel** des données entre un **stockage permanent** (souvent une base de données relationnelle) et une **représentation en mémoire du programme** des données.

Responsabilité de persistance des données

- La sauvegarde des objets, que ce soit dans un fichier ou dans une base de données, est une **responsabilité en tant que telle**.
- On va donc utiliser une classe **dédiée** à cette responsabilité.
- Toutes les informations techniques du système de stockage de données utilisé seront gérées par cette classe.
- Si on change de type stockage, on ne changera que la classe de persistance, sans devoir modifier les classes du modèle de données.
- Exemple :

```
Maze myMaze = new Maze(...);
EntityPersistenceManager myPersistenceManager = ...;
myPersistenceManager.persist(myMaze); // Sauvegarde des
données
```

Un des touts premier framework de persistance des données

- Enterprise Java Beans (**EJB**) : composants fournissant simultanément la logique métier et les capacités de persistance.
- Problèmes : lourd et compliqué (configuration basée sur des documents XML compliqués nécessitant beaucoup de code de configuration).
- Déploiement doit se faire uniquement dans un serveur d'applications Java EE.

De meilleures solutions ont émergées depuis

- Mapping Objet-Relationnel (Object Relational Mapping, ORM)
  - Couche d'accès aux données réalisant l'adaptation entre les objets (OO) et leur representation en base de donnée (tables relationnelles).
  - E.g., Hibernate et Oracle Toplink devenu EclipseLink.
- Basé sur des Plain Old Java Objects (POJO).
  - Ne contiennent que des données et de la logique métier, **pas de persistance dans le modèle**.
- Puis apparition de la norme JPA (Java Persistence API) intégrée à EJB 3.0.
- Devenu un projet indépendant en 2019 appelé Jakarta Persistence et intégrée à Jakarta EE.
- La dernière version est JPA 3.2

Annoter une classe pour JPA

```
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Student implements Serializable {

   @id
   private int number; // database primary key

   private String firstName;

   private String lastName;
   ...
}
```

Fichier de configuration de la persistance (persistence.xml)

```
<?xml version="1.0"    encoding="UTF-8"?>
  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

  <persistence-unit name= "myStudentsApplicationDB">

<provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>mypackage.Student</class>
  </persistence-unit>
</persistence>
```

Créer la table correspondante dans le modèle de la base de données

```
CREATE TABLE Student {
    Id INTEGER PRIMARY KEY;
    FirstName CHAR(256);
    LastName CHAR(256);
}
```

Compléter les annotations de la classe

```
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Student implements Serializable {

   @id
   @column(name=“Id" table="Student")
   private int number; // database primary key

   @column(name=“FirstName" table="Student")
   private String firstName;

   @column(name=“LastName" table="Student")
   private String lastName;
   ...
}
```

Gérer la persistance des objets

```
EntityManagerFactory entManagerFactory =
Persistence.createEntityManagerFactory("myStudentsApplicationDB");

EntityManager entityManager =
entManagerFactory.createEntityManager();

EntityTransaction entityTransaction =
entityManager.getTransaction();

entityTransaction.begin();

Student myStudent = new Student(12345, "Elton", "John");
entityManager.persist(myStudent);

entityTransaction.commit();

entityManager.close();

entManagerFactory.close();
```

Lire un objet en base

```
EntityManagerFactory entManagerFactory =
Persistence.createEntityManagerFactory("myStudentsApplicationDB");

EntityManager entityManager =
entManagerFactory.createEntityManager();

Student myStudent = entityManager.find(Student.class, 12345);

entityManager.close();

entManagerFactory.close();
```

Lire un objet en base en exécutant une requête SQL

```
EntityManagerFactory entManagerFactory =
Persistence.createEntityManagerFactory("myStudentsApplicationDB");
EntityManager entityManager = entManagerFactory.createEntityManager();

Query query = entityManager.createQuery("select s from Student p
where ...");

Student myStudent = (Student) query.getSingleResult();

List<Student> myStudents = (List<Student>) query.getResultList();

entityManager.close();
entManagerFactory.close();
```

Une interface de persistance simple fournie par l'interface graphique *Canvas Viewer*

Package fr.tp.inf112.projects.canvas.model

Interface CanvasPersistenceManager

All Known Implementing Classes:

`AbstractCanvasPersistenceManager`

`public interface CanvasPersistenceManager`

Interface providing methods to manage the persistence of canvas models into a data store such as files, databases or any other permanent storage. Required by the canvas viewer when the canvas must be saved or read from the storage.

Author:

Dominique Blouin

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method	Description
`boolean`	`delete(Canvas canvasModel)`	Deletes the specified canvas model from the data store of this persistence manager.
`CanvasChooser`	`getCanvasChooser()`	Provides an object responsible for choosing a canvas to be retrieved from the data store by the canvas persistence manager.
`void`	`persist(Canvas canvasModel)`	Persists the specified canvas model into the data store of this persistence manager.
`Canvas`	`read(String canvasId)`	Reads a canvas model from the data store of this persistence manager from the ID of the required canvas model.

Conclusion

- La persistance des données est un ingrédient essentiel des applications logicielles.
- Plusieurs architectures de persistance existent, et il est préférable d'**isoler** la persistance des données de leurs données (couche de persistance).
  - Le model est indépendant de la technologie de persistance employée.
  - Principe de responsabilité unique des classes.
  - La technologie de persistance peut être changée sans modifier les classes du modèle.
- JPA permet cette architecture mais demeure une technologie complexe couvrant les nombreux aspects des bases de données.
- JPA s'utilise dans des serveurs d'application qui permettent de tout automatiser :
  - Déclarations du modèle de données.
  - Créations de tables.
  - Génération des classes.
  - ...