Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

ATL – Ateliers logiciels

Accès aux données

Repository pattern

> **Consignes**
> Ce TD va vous apprendre à implémenter le design pattern `Repository` en l'appliquant au cas concret de gestion d'un fichier. Le temps de travail estimé se compose de 2h de travail en classe et de 2h de travail à domicile. Les ressources nécessaires sont disponibles sur PoESI.

1 Fichier des étudiants

L'objectif de ce TD est de lire le contenu du fichier `subscribed.txt` via un programme java en respectant le pattern `Repository`.

Ce fichier contient une liste d'étudiants inscrits à l'ESI. Chaque étudiant est représenté par un matricule, un nom et un prénom.

```
64931,Olsen,Maggy
73780,Frost,Phoebe
94853,Ortega,Skyler
93371,Blankenship,Byron
82227,Cote,Molly
```

Les données de ce fichier ont été générées aléatoirement via https://generatedata.com/.

Pour pouvoir accéder au fichier `subscribed.txt` il faut renseigner son chemin à notre programme. On peut penser à :

- ▷ entrer son chemin via l'interface graphique ;
- ▷ déclarer une constante dans notre programme ;
- ▷ utiliser un fichier qui contient les paramètres de l'application.

C'est cette dernière option que nous allons mettre en place.

2 Accès à une ressource

Au sein du dépôt git du cours, créez un nouveau projet Java avec Maven appelé `Mentoring`. Ajoutez une classe principale qui contient le code suivant et exécutez cette classe :

```
    System.out.println("Chemin courant \t" + new File(".").getAbsolutePath());
    System.out.println("Chemin classe \t" +
↪ this.getClass().getResource(".").getPath());
    System.out.println("Chemin jar \t" + new
↪ File(getClass().getClassLoader().getResource(".").getFile()));
```


Version du document
18 février 2020

Comme vous pouvez le constater ce programme peut accéder facilement à différents répertoires :

- le répertoire de votre projet sur la machine ;
- le répertoire de la classe en cours d'exécution ;
- le répertoire de l'exécutable.

Consultez la documentation de la méthode `getRessource()` [1] pour en savoir plus.

3 Maven et ses ressources

Maven utilise pour gérer ses fichiers une structure de dossier particulière.

Premièrement il faut distinguer :

- les fichiers qui sont copiés lors de la compilation pour faire partie du programme ;
- les fichiers extérieurs au programme.

Le contenu des deux dossiers ci-dessous est **toujours** copié tel quel dans le dossier `target` après compilation :

- `/src/main/resources` : contient les données paramétrant le programme ;
- `/src/test/resources` : contient les données utiles pour les tests ;

> Créez les deux dossiers de gestion des ressources et placez-y les fichiers textes
>
> - `/src/main/resources` : `resourceMain.txt` ;
> - `/src/test/resources` : `resourceTest.txt` ;
>
> Compilez votre code et vérifiez l'existence des fichiers :
>
> - `/target/classes/resourceMain.txt` ;
> - `/target/test-classes/resourceTest.txt` ;

Concernant les données extérieures au programme, placez le fichier `subscribed.txt` à la racine du programme, en dehors des sources dans un dossier nommé `data`. Ce dossier ne sera pas copié dans le dossier `target` lors de la compilation.

4 Fichier Properties

Créez le dossier `/src/main/resources/config` et ajoutez un nouveau fichier appelé `config.properties` dont le contenu est le suivant :

```
#Properties

#Application description
 app.name=Mentoring
 app.version=1.0
 app.author=G12345
```

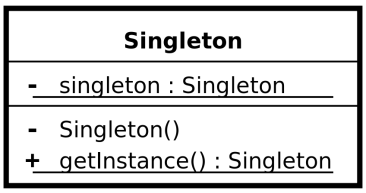
Les fichiers `properties` sont des fichiers clés-valeurs faciles à utiliser en java. On peut lire dans le fichier `config.properties` que la clé `app.author` est associée à la valeur `G12345`.

1. https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Class.html

Le JDK met à disposition la classe `Properties`[2] qui prévoit différentes méthodes pour accéder aux données contenues dans ces fichiers `properties` :

- ▷ `load(InputStream inStream)` : charge le contenu de tout le fichier en mémoire ;
- ▷ `getProperty(String key)` : retourne la valeur de la clé correspondante.

Afin d'avoir accès au contenu de ce fichier de configuration dans toutes les classes, créez une classe Singleton[3] via le menu de création de fichier de Netbeans[4].



- ▷ package : `g12345.mentoring.config`
- ▷ classe : `ConfigManager.java`

2. https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Properties.html
3. https://en.wikipedia.org/wiki/Singleton_pattern
4. Si vous créez un nouveau fichier vous pouvez sélectionner dans le menu java l'option Java Singleton Class.

```java
public class ConfigManager {

   private ConfigManager() {
      prop = new Properties();
      url = getClass().getClassLoader().getResource(FILE).getPath();
   }

   private static final String FILE = "./config/config.properties";

   private final Properties prop;

   private final String url ;

   public void load() throws IOException {
      try (InputStream input = new FileInputStream(url)) {
         prop.load(input);
      } catch (IOException ex) {
         throw new IOException("Chargement configuration impossible " +
    ↪ ex.getMessage());
      }
   }

   public String getProperties(String name) {
      return prop.getProperty(name);
   }

   public static ConfigManager getInstance() {
      return ConfigManagerHolder.INSTANCE;
   }

   private static class ConfigManagerHolder {

      private static final ConfigManager INSTANCE = new ConfigManager();
   }
}
```

Chargez les propriétés de votre programme dans votre classe principale et affichez-les.

Une fois cette étape réalisée, ajoutez au fichier `properties` l'endroit où est déposé le fichier de données via la clé `file.url`.

```
#Properties

#Application description
 app.name=Mentoring
 app.version=1.0
 app.author=G12345

#File description
 file.url=data/subscribed.txt
```

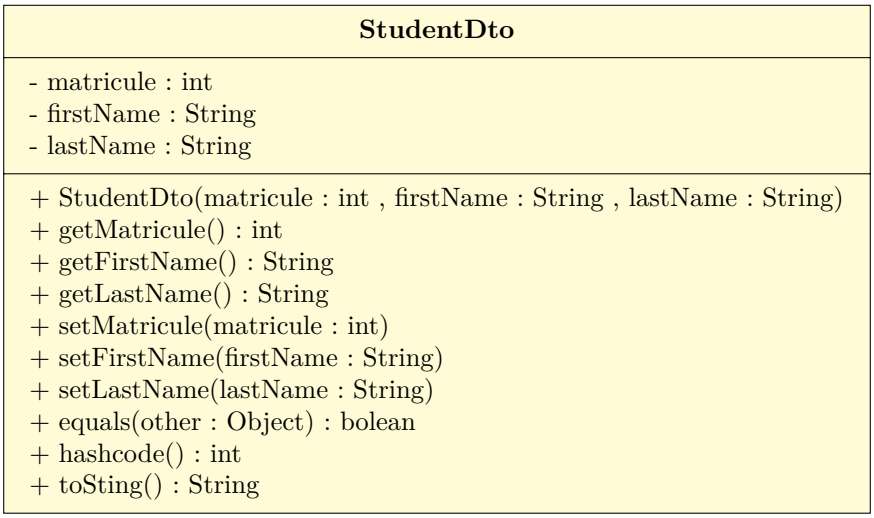
5 DTO : Data Transfer Object

Afin de lire le contenu du fichier nous allons placer chacune de ses lignes dans une instance d'un objet qui a pour unique responsabilité de transporter les données de la ligne. Ce type particulier d'objet qui ne contient aucune logique est appelé `Data Transfer Object`. Ce DTO possède :

- ▷ les attributs qui représentent chaque colonne d'une ligne du fichier ;
- ▷ un constructeur qui prend en paramètre les différents attributs ;
- ▷ des accesseurs et des mutateurs pour chacun des attributs ;

- une méthode `equals` qui utilise la clé de la donnée pour l'identifier ;
- une méthode `toString` qui affiche les valeurs des données.

Dans le package `g12345.mentoring.dto` créez la classe `StudentDto` comme présentée ci-dessous.



6 Le repository

L'idée derrière le `Repository` pattern est d'avoir au sein de l'application, une classe qui centralise les accès à un type de données.

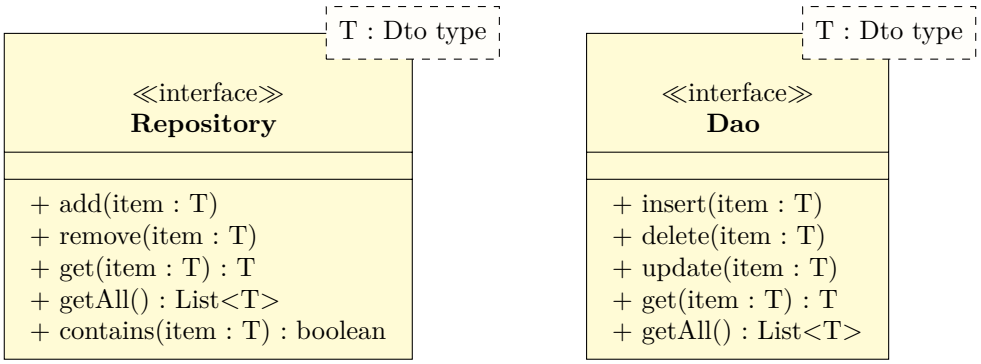
Par exemple nous allons créer une classe `StudentRepository` qui va être le point d'accès des données concernant les étudiants. Si nous avions besoin d'accéder aux informations concernant les enseignants nous créerions une classe `TeacherRepository`.

Lors de l'accès aux données il faut distinguer deux types d'actions :

- la logique des accès aux données : avant d'ajouter une donnée, il faut vérifier si elle n'existe pas déjà, avant de lire une donnée il faut vérifier si on ne l'a pas déjà en mémoire,...
- l'accès à la ressource : faut-il se connecter à une base de données, comment ouvrirle fichier à consulter,...

Ces responsabilité sont attribuées à deux types d'objet différents :

- le `Repository` qui gère la logique de l'accès aux données ;
- le Data Access Object qui gère l'accès "physique" aux données.

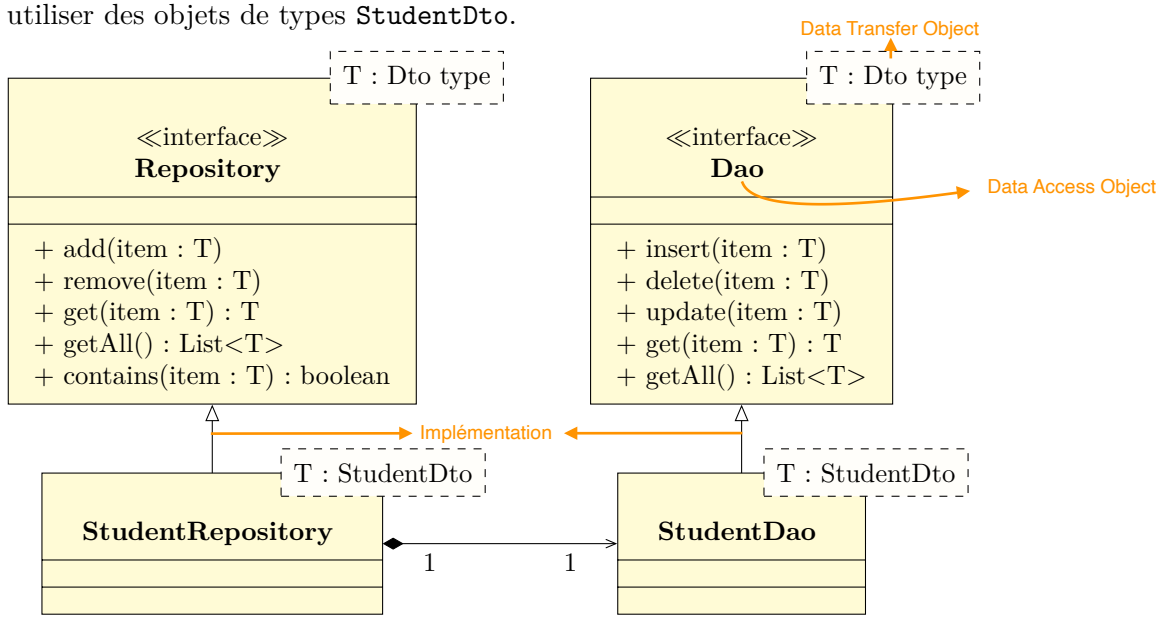


Remarques

Prenez note de l'utilisation de génériques dans ces interfaces via les templates `T`. Remarquez également que la lecture d'une donnée via la méthode `get(T item)` retourne un objet de `T`.
Ce qui signifie dans notre cas qu'un objet de type `StudentDto` est donné en paramètre. Ce paramètre contient uniquement la valeur concernant le matricule de l'étudiant, ses autres attributs étant nulls. La méthode `get` retourne alors un objet complet avec son nom et prénom. La signature de la méthode devrait recevoir uniquement un paramètre entier. Nous améliorons cet aspect dans le prochain TD.

Créez dans le package `g12345.mentoring.repository` les interfaces `Repository` et `Dao`.

Ensuite afin d'accéder à des données représentées par des `StudentDto`, créez les implémentation `StudentRepository` et `StudentDao` de vos interfaces en spécifiant les classes pour utiliser des objets de types `StudentDto`.



Pour vous aider dans votre implémentation voici quelques pistes :

- remarquez que le `StudentRepository` est composé d'un objet `StudentDao` ;
- ajouter un élément dans le fichier via le `Repository` demande de vérifier si cet élément existe déjà afin de distinguer les insertions des mises à jour ;
- vous avez accès à l'emplacement du fichier via la propriété `file.url` du fichier de configuration de l'application ;
- n'hésitez pas à utiliser la classe `java.nio.file.Files`[5] pour accéder aux fichiers, pensez en particulier aux méthodes :
  - `Files.lines(Path path)`
  - `Files.write(Path path, byte[] bytes, OpenOption... options)` ;
- l'utilisation d'un Singleton pour votre `Dao` est sans doute pertinente.

5. https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

7 Validation de votre repository

Dans le prochain TD vous apprendrez à effectuer des tests unitaires sur votre `Repository` via `Mockito`[6]. Pour l'instant validez votre travail via votre classe principale qui effectuera les opérations suivantes :

1. afficher tous les éléments présents dans le fichier via le `Repository` ;
2. ajouter un élément nouveau dans le `Repository` ;
3. afficher tous les éléments présents dans le fichier via le `Repository` ;
4. modifier un élément existant dans le fichier ;
5. afficher tous les éléments présents dans le fichier via le `Repository` ;
6. supprimer un élément du fichier ;
7. afficher tous les éléments présents dans le fichier via le `Repository` ;

6. https://site.mockito.org/