TD n°6 : Patterns Adapter et Façade

Exercice 1 (Adapter 1 : Base de données de chimie)

Vous écrivez un programme de calcul de saturation moléculaire pour une entreprise pharmaceutique. Jusqu'à aujourd'hui, cette entreprise mettait à disposition une base de données des molécules qu'elle utilise, mais elle a décidé de mettre ses nouvelles molécules dans une toute nouvelle base de données, tout en gardant les molécules de base dans l'ancienne base de données.

Bref, ça vous complique la tâche : votre code existant (qui ne fera qu'afficher les éléments de la BDD) doit toujours fonctionner avec l'ancienne BDD mais également avec la nouvelle, et bien entendu ces deux BDDs n'ont pas la même interface. Il semble que le pattern Adapter s'impose ici.

Voici ce que votre code dans l'état actuel :

- Une classe `Molecule` qui a un `nom` et une `formule` (en string). Cette classe a une méthode `affiche` affichant le nom et la formule. On récupère la molécule via une classe fournit par l'entreprise, à savoir `OldDB` qui a une méthode `Molecule oldMolecule(String name)` prenant le nom d'une molécule et renvoyant son objet correspondant.
- Une classe principale avec juste une méthode principale instanciant et affichant une molécule de base `eau`.

Ce que vous avez maintenant :

- Une nouvelle classe `NewDB` contenant l'interface de la nouvelle base de données, à savoir une méthode `Molecule newMolecule(String name)`.

1. Faire le diagramme UML du programme utilisant le pattern Adapter pour prendre en compte l'interface de cette nouvelle BDD.
2. Coder le pattern en Java. Vous ne devez pas modifier votre classe `Molecule`, et bien sûr vous n'avez pas accès aux classes `OldDB` et `NewDB`. Modifier la méthode principale de manière à afficher en plus la molécule `ethanol` se trouvant dans la nouvelle BDD.

Exercice 2 (Adapter 2 : Bibliothèques et rétro-compatibilité)

Ce qui marche avec les BDD marche aussi avec les bibliothèques : Imaginez que vous codiez un programme s'appuyant sur des bibliothèques (ce qui est très vraisemblable). Votre code est open-source mais vous ne fournissez pas le code des bibliothèques que vous utilisez. Par exemple, vous faites un code en C++ utilisant Boost et vous écrivez dans votre fichier README "pour compiler et exécuter ce programme, installez d'abord la bibliothèque Boost".

Maintenant vous décidez de changer de bibliothèque pour une raison X (par exemple, Boost est bien mais trop lourd). Cependant vous souhaitez que votre nouveau code soit rétro-compatible, c'est-à-dire que quelqu'un qui continue à utiliser Boost puisse toujours compiler et exécuter votre programme. Là encore, Adapter est votre sauveur. Gloire à lui.

Exercice libre et créatif : imaginez un code existant faisant des appels à une ancienne bibliothèque, et, en considérant l'utilisation d'une nouvelle bibliothèque avec une interface différente, utilisez le pattern Adapter afin d'adapter votre code à cette nouvelle bibliothèque sans changer l'interface de votre code pour quelle reste compatible avec l'ancienne bibliothèque.

Exercice 3 (Façade)

Quand un prof arrive en salle de classe pour son cours, il a plein de trucs à faire qui l'agace :

- allumer les lumières de la salle,
- brancher la multi-prises au secteur,
- brancher le vidéo-projecteur sur la multi-prises,
- brancher l'ordinateur sur la multi-prises,
- relier le vidéo-projecteur à l'ordinateur ,
- allumer le vidéo-projecteur,
- allumer l'ordinateur.

Et l'inverse à la fin du cours. Vous avez un code existant avec les classes :

- `Salle` contenant les méthodes :
  - `allumerLumiere()`
  - `eteindreLumiere()`
- `Multiprises` avec :
  - `brancheMulti()`
  - `débrancherMulti()`
- `Ordi` avec :
  - `brancherOrdi(Multiprises mp)`
  - `debrancherOrdi(Multiprises mp)`
  - `allumerOrdi()`
  - `eteindreOrdi()`
- et enfin `VP` avec :
  - `brancheVP(Multiprises mp)`
  - `debrancherVP(Multiprises mp)`
  - `relierVP(Ordi ordinateur)`
  - `allumerVP()`
  - `eteindreVP()`

Or, ce qu'un prof aimerai, c'est d'avoir juste deux méthodes `debutCours` et `finCours` à appeler. Pour ça, Façade est votre amie.

1. UML.
2. Java.