

Adapter et Façade

Exercice 1 (Adapter : 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 fournie 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.
 3. Écrivez une méthode principale qui instancie une molécule `eau` se trouvant dans l'ancienne base de données et une molécule `ethanol` se trouvant dans la nouvelle. Rappels : la formule de l'eau est H_2O et celle de l'éthanol est CH_3CH_2OH .

Exercice 2 (Façade : Cours d'un prof)

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 aimerait, c'est d'avoir juste deux méthodes `debutCours` et `finCours` à appeler. Pour ça, Façade est votre amie.

On va procéder en deux temps : considérons d'abord l'époque où tous les profs avaient leur cours sur transparents afficher via un rétroprojecteur. Votre sous-système sera donc d'abord composé des classes **Salle**, **Multiprises**, et **Retroprojecteur**, ce dernier contenant les méthodes `allumer` et `eteindre`.

1. Écrivez ces classes en Java, ainsi qu'une classe **Prof** et une **Façade** permettant au prof de juste avoir à appeler deux méthodes qui font tout : `debutCours` et `finCours`.
2. Changez votre sous-système en laissant tomber le rétroprojecteur et en utilisant cette fois les classes **Ordi** et **Videoprojecteur** du TD. Modifiez le fonctionnement de votre façade considérant ce nouveau sous-système. Constatez que l'interface de votre façade ne change pas d'un poil, ainsi que votre classe cliente **Prof**.