Introduction aux Design Patterns – Exercices

Pour chaque exercice, vous créerez votre propre fork, à partir du dépôt Github CERI-L3-DesignPatterns, en acceptant une affectation sur GitHub Classroom. Vous clonerez ensuite votre fork en local, et mettrez à jour le fork régulièrement en y transférant vos commits.

Pour tous les exercices, vous devez commencer par concevoir le modèle initial des classes que vous allez implémenter. Ce modèle pourra, suivant les questions, évoluer, il faudra donc en fournir les différentes versions sur votre fork. Vous fournirez aussi le code source des classes au fur et à mesure que vous les implémentez.

Exercice 1

A - On veut gérer des informations sur des *Etudiant*s, à savoir par exemple leurs nom et prénom. On veut aussi pouvoir identifier, parmi les étudiants, l'un d'entre eux, unique, qui sera le représentant de tous les étudiants.
Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours, puis implémentez les éléments de ce diagramme en Java.

B - Complétez le modèle UML précédent pour qu'il puisse aussi gérer des *Enseignants* (nom et prénom), pour lesquels on veut aussi identifier un unique représentant.

C - Transformez le modèle précédent en définissant la classe *Personne*, en factorisant les informations des deux classes précédentes.
On veut maintenant pouvoir définir un représentant unique par classe, y compris pour la classe *Personne*. Réfléchissez à un modèle possible et aux difficultés d'implémentation.
Proposez une implémentation du *Représentant* utilisant un registre associant un nom de classe, à l'instance qui en est le représentant.

Eléments techniques en Java :
`getClass()` permet, à partir d'un objet, de connaitre sa classe, et `getName()` sur cette classe retourne son identificateur.
`HashMap()` est une collection qui permet d'associer une clé à une valeur. Vous pouvez l'utiliser pour associer (par exemple) un objet *String* à un objet quelconque.

Exercice 2

On veut écrire une (très) petite application de domotique, qui doit permettre, dans un premier temps, de connecter toutes sortes de dispositifs électroniques. On ne soucie pas, pour l'instant, d'activer ces dispositifs.

A - Pour cette première partie, on ne considère que deux catégories d'objets possibles : des radios et des cafetières. L'application demande (en boucle) à l'utilisateur de saisir un entier, 1 pour connecter une radio, 2 pour une cafetière et 0 pour finir. Selon le choix de l'utilisateur, il faudra créer un objet de la classe correspondante puis l'ajouter à une collection des objets connectés : vous aurez par conséquent écrit les classes *Radio* et *Cafetière*.
Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours, puis implémentez les éléments de ce diagramme en Java.
Que faut-il faire pour ajouter une nouvelle catégorie d'objets ? Quelles sont les limites de cette solution ?

Eléments techniques en Java :
Vous pouvez utiliser une collection `ArrayList` pour stocker les objets *Connectable*s.

B - L'application précédente ne permet pas d'ajouter un nouveau dispositif sans impliquer une modification du code ; nous allons progressivement la transformer de façon à ce qu'elle puisse créer toutes sortes d'objets, à condition qu'ils soient *Connectable*s. Dans cette nouvelle version, l'application demande (en boucle jusqu'à la saisie d'une ligne vide) le nom de la classe de l'objet à créer, en crée une instance et enregistre ce nouvel objet dans la collection des objets connectés. Dans un premier temps, vous n'envisagez que de connecter des *Radio*s ou des *Cafetiere*s
Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours, puis implémentez les éléments de ce diagramme en Java.

C – On va maintenant faire fonctionner l'application de façon autonome, c'est-à-dire en dehors de l'environnement de développement : pour cela, vous suivrez les indications dans le README du répertoire exo2c : vous allez générer une archive .jar de votre application et l'exécuter en ligne de commande. Ensuite vous essayerez avec une nouvelle sorte de *Connectable*, la classe *Radiateur* qui vous est donnée dans le répertoire nouveauConnectable.

Eléments techniques en Java :
La classe *FabriqueGenerique* utilise l'API Reflection de Java. Si vous souhaitez approfondir ces mécanismes, vous pouvez consulter :
http://www.jmdoudoux.fr/java/dej/chap-introspection.htm
http://docs.oracle.com/javase/tutorial/reflect/

Exercice 3

L'application précédente va maintenant évoluer de façon à activer certains des objets connectés. Chaque objet connecté possède donc désormais une méthode *démarrer* qui correspond à sa fonctionnalité (faire couler le café pour une *Cafetière*, émettre du son pour une *Radio*, …) ; bien entendu, vous n'avez pas à gérer cette fonction principale : son implémentation consiste uniquement en l'affichage d'un message, par exemple, si une radio est démarrée, vous vous contentez d'afficher < La radio ... démarre >, avec … représentant le nom de l'objet *Radio*.
L'application propose à l'utilisateur de choisir, parmi les différents dispositifs connectés, ceux qui devront être démarrés, puis les démarre.
Elle proposera aussi de désactiver les objets qui ont été connectés, pour ne plus les démarrer.
Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours, puis implémentez les éléments de ce diagramme en Java.

Exercice 4

On vous fournit maintenant une *Imprimante* qui n'est pas directement connectable à l'application de domotique. Cette imprimante possède une fonction *imprimer* ; ici encore son implémentation ne fait qu'afficher < L'imprimante ... imprime >.
Proposez deux solutions différentes pour utiliser cette imprimante avec l'application de domotique. Réalisez les diagrammes de classes appropriés, en appliquant un pattern vu en cours, puis implémentez les éléments de ces diagrammes en Java.

Exercice 5

A - On veut maintenant pouvoir répartir la consommation totale de l'application de domotique sur chacun des objets connectés. On imagine pour le moment qu'on ne peut faire que deux types de répartition : la répartition du temps de fonctionnement ou la répartition de la puissance consommée.
Par exemple, pour chacun des objets connectés, on pourra afficher le pourcentage du temps total pendant lequel il a fonctionné, ou bien le pourcentage de la puissance totale qu'il a consommé.

Le problème, dans cet exercice, n'est pas de calculer les temps de fonctionnement de l'application ou de chacun des objets connectés : ces données seront fixées "en dur" dans votre code (ou, si vous le souhaitez, demandées à l'utilisateur). Il ne s'agit pas non plus de calculer les puissances consommées. Ainsi, si vous décidez que l'application a fonctionné pendant 60 minutes et qu'une *Radio* a été activée pendant 30 minutes, vous devez juste déduire, par simple calcul, que cette *Radio* a consommé 50% du temps total.

Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours, puis implémentez les éléments de ce diagramme en Java.

B – On souhaite maintenant répartir la consommation en fonction de données fixes, qui dépendent du mode de fonctionnement de l'appareil connecté.
Par exemple, un radiateur pourrait fonctionner en mode veille, ou en mode normal ou en mode haute consommation, en fonction du fait qu'on aurait besoin de plus ou moins de chaleur.
A chaque mode est affecté un pourcentage fixe.
Intégrez cette nouvelle fonctionnalité à l'application de domotique existante.

Exercice 6

Dans cet exercice, vous ne devez faire que la modélisation afin de produire un diagramme UML, l'implémentation n'est pas demandée.
On veut connecter une télévision à l'application de domotique, il s'agit d'un dispositif comme les autres, qui peut être démarré au moment souhaité.
Il s'agit ici d'ajouter, du côté de l'application, une possibilité de contrôle parental, qui, lorsqu'on veut programmer le démarrage de la télé, demanderait de saisir un code de sécurité.

Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours.

Exercice 7

Dans cet exercice, vous ne devez faire que la modélisation afin de produire un diagramme UML, l'implémentation n'est pas demandée.
Toujours à partir de la même application, on peut allumer une guirlande décorative, qui comme tous les dispositifs précédents, doit être connectée et dont on programme le moment d'activation. Cette guirlande est équipée de leds de couleurs différentes pouvant être activées indépendamment (sans être exclusives).
Il s'agit ici d'ajouter dynamiquement, du côté de l'application domotique, de nouvelles fonctionnalités à cette guirlande.
On ajoutera tout d'abord une possibilité de faire varier et de combiner les couleurs.
On ajoutera ensuite la possibilité de faire clignoter cette guirlande, en paramétrant une temporisation de quelques secondes, de façon à maintenir pendant ce temps la guirlande dans chaque état allumé ou éteint. Par exemple, si on paramètre la guirlande avec une valeur d'une seconde, elle restera 1 seconde allumée et 1 seconde éteinte.
On ajoutera enfin la possibilité de faire clignoter la guirlande en transmettant une séquence de plusieurs intermittences de temps. Ainsi si on active avec une séquence de [2 secondes, 1 seconde, 2 secondes], la guirlande s'allume pendant 2 secondes, s'éteint pendant 2 secondes, puis s'allume pendant 1 seconde, s'éteint pendant 1 seconde , puis s'allume pendant 2 secondes, s'éteint pendant 2 secondes, pour recommencer en boucle au début de la séquence.

Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours.

Exercice 8

A – On souhaite utiliser l'application de domotique pour gérer les dates de maintenance des dispositifs que l'on connecte.
Le type et la fréquence de la maintenance varie selon le type de dispositif : par exemple, il faudrait détartrer une cafetière tous les 3 mois, tandis qu'un radiateur doit être entretenu une fois par an, et pour une radio, on peut envisager de synchroniser les fréquences tous les deux mois.
On suppose désormais que tous les dispositifs que l'on connecte possède une date de dernière opération de maintenance , ou, à défaut, une date de première mise en service.
L'application propose de nouvelles fonctionnalités :
- afficher, pour chaque dispositif, leur date de prochaine maintenance, accompagnée d'un message personnalisé. Par exemple, elle affichera les messages suivants : "Il faudrait détartrer la cafetière ... avant le …" ou bien "Les fréquences de la radio ... seront synchronisées avant le …".
- afficher, pour chaque dispositif, la liste des dates des 3 prochaines maintenances à prévoir.

Proposez un diagramme de classes approprié, en appliquant un pattern vu en cours, puis implémentez les éléments de ce diagramme en Java.

B – Que se passe-t-il si l'application doit gérer une nouvelle catégorie de dispositif à connecter, par exemple un volet roulant électrique dont il faut contrôler la motorisation ? Dans cette partie, vous ne devez pas d'implémentation mais réfléchir à la modélisation afin de produire un diagramme UML, et tirer une conclusion sur les caractéristiques du pattern mis en œuvre.

Eléments techniques en Java :
Vous pouvez utiliser la classe LocalDate et ses méthodes of, now, plusMonths. Voir pour cela :
https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html