UE Conception Orientée Objet

Devoir Surveillé

Jeudi 19 décembre 2013 – 14h-17h
Copies des diapositives de cours autorisées
Fiches « design pattern » du cours
Une feuille recto-verso de notes personnelles
Dictionnaire de langue (papier ou électronique "dédié") autorisé
Tout autre document interdit

Les exercices sont indépendants.
La clarté des réponses sera prise en compte dans l'évaluation, en particulier pour les diagrammes UML dans lesquels apparaitront toujours :

- *les liens d'héritage/implémentation entre types*
- *les noms et types de tous les attributs, ainsi que leur visibilité,*
- *les méthodes et constructeurs avec leurs paramètres et leurs types ainsi que les types des valeurs de retour*

Soignez la présentation de ces diagrammes en évitant autant que possible les ratures !
La javadoc et les tests ne sont à fournir que lorsque cela est explicitement demandé.

A titre purement indicatif, durée estimée et suggérée de traitement des exercices :
lecture du sujet : 10mn – exercice 1 : 20mn – exercice 2 : 55mn – exercice 3 : 55mn – exercice 4 : 40mn

Répondre sur deux copies (et leurs intercalaires) séparées,
copie jaune pour les exercices 1 et 2 et copie verte pour les exercices 3 et 4.

Exercice 1 : A propos de programmation objet. *(sur la copie jaune)*
A l'aide de réponses claires, courtes et précises répondez aux questions suivantes :

Q 1 . Définissez `this` et `super` et illustrez ces définitions à l'aide d'un exemple.

Q 2 .

Q 2.1. Définissez ce qu'est le mécanisme de « *liaison tardive* ».

Q 2.2. Donnez un exemple simple et clair mettant en évidence son fonctionnement.

Q 2.3. Qu'apporte ce mécanisme à la construction de programmes ?

Q 3 . *(bonus)* Quel résultat renvoie un appel à la méthode :

```
public boolean foo() {
    return super.getClass().equals(this.getClass());
}
```

Exercice 2 : Biens immobiliers *(sur la copie jaune)*
On s'intéresse ici à la modélisation de biens immobiliers[1]. Un bien `Immobilier` est caractérisé par une surface totale du bien (en nombre entier de m^2), une adresse (de type `Adresse` supposée définie) et un propriétaire (voir ci-dessous). Un objet représentant un bien immobilier dispose d'une méthode `surfaceImposable` qui fournit la surface du bien soumise à impôt, par défaut il s'agit de la surface totale.
On distingue parmi ces biens les `Habitation`s des bâtiments à usage professionel (`BatimentProfessionnel`).

Les habitations sont caractérisées par un nombre de pièces. Parmi les habitations on trouve :

- les `Appartement`s qui sont caractérisés par le numéro de l'étage où se trouve l'appartement.
  Les `Studio`s sont des appartements particuliers n'ayant qu'une seule pièce. Un coefficient de 0,9 est alors appliqué par rapport à la surface imposable retenue pour un appartement.
- les `Maison`s qui sont caractérisées par la surface du terrain sur lequel est bâtie la maison.

Les bâtiments à usage professionnel sont divisés entre :

[1] Le sujet ne se veut absolument pas exhaustif sur ce thème, ni même réaliste, et certains choix très arbitraires sont faits pour les besoins de l'exercice

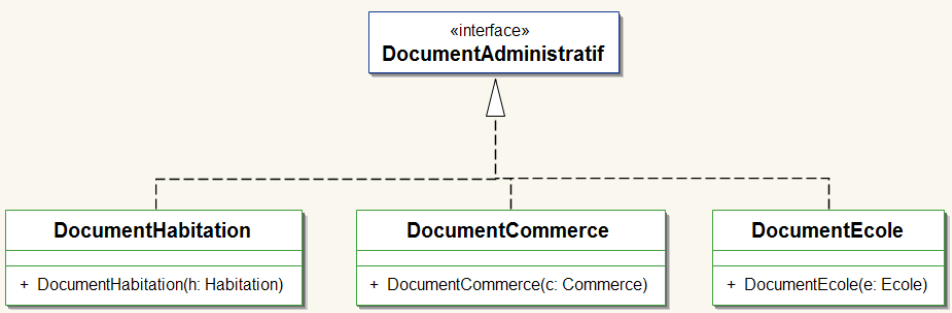
- les `Commerces` qui sont caractérisés par un entrepôt d'une certaine surface. La surface de cet entrepôt est déduite de la surface totale pour obtenir la surface imposable d'un commerce.
- les `Ecoles` caractérisées par un nombre de salles de classe et un `Niveau` (maternelle, élémentaire, collège ou lycée). Les écoles ne sont pas imposables.

Les différentes informations définissant les biens immobiliers (habitations ou commerces) sont fournies à la création des objets et des accesseurs (« getters ») sont disponibles.

Un propriétaire de bien immobilier dispose d'un nom. On distingue parmi les propriétaires, les personnes physiques (`PersonnePhysique`) et les personnes morales (`PersonneMorale`). Une personne physique est identifiée par un numéro de sécurité sociale (représenté par une classe `NumSecu` supposée définie) et une personne morale par un numéro de SIRET (représenté par une classe `NumSIRET` supposée définie).
Le propriétaire d'une habitation est nécessairement une personne physique et celui d'un bâtiment à usage professionnel une personne morale. Le propriétaire peut changer.

Les appartements, maisons et commerces sont `Louables` mais pas les écoles. Ils proposent donc des méthodes (`estLoue`/`setLoue`) permettant de gérer cet aspect.

Enfin, à chaque bien immobilier correspond un document administratif règlementaire propre au bien. Le type de document peut différer selon la nature du bien immobilier, on a ainsi un type de document pour les habitations, un autre pour les commerces et un troisième pour les écoles. Les types de ces documents appartiennent au paquetage `immobilier.document` et sont présentés dans le diagramme suivant :



Les méthodes disponibles pour ces documents ne sont pas détaillées car elles n'ont pas d'importance pour ce sujet.
Chaque bien immobilier doit disposer d'une méthode qui lui permette de fournir un exemplaire du document administratif qui lui est attaché. Dans la mesure où les informations sur le bien peuvent changer (le propriétaire par exemple) mais aussi le contenu règlementaire du document (soumis aux évolutions de la loi), pour garantir son exactitude il est nécessaire qu'un nouveau document administratif soit généré à chaque invocation de cette méthode.

Q 1 . Faites une proposition de modélisation pour les données présentées ci-dessus. Vous présenterez votre solution sous la forme d'un diagramme UML **clair** et **détaillé** dans lequel apparaissent :

- les liens d'héritage/implémentation entre types
- les noms et types de tous les attributs,
- les méthodes et constructeurs avec leurs paramètres et leurs types ainsi que les types des valeurs de retour,

Vous devez faire apparaître dans votre diagramme **au minimum** tous les éléments dans `cette police` du texte précédent, à l'exception des classes `Adresse`, `NumSecu` et `NumSIRET` que l'on supposera définies.

Si besoin présentez, pour une meilleure lisibilité, votre diagramme en "format paysage" sur votre copie ou sur les deux pages intérieures de la copie.

Q 2 . Donnez un code java pour le type permettant de représenter les studios.

Q 3 . Donnez un code java pour le type permettant de représenter les écoles.

Exercice 3 : Commandes et fournisseurs *(sur la copie verte)*
On va s'intéresser dans cet exercice à un marché (`Marche`) qui regroupe des `Commandes`. Ce `Marche` propose ces `Commandes` à des `Fournisseurs` qui peuvent les accepter selon leurs critères.

Les commandes.

Une commande porte sur une certaine quantité de produits d'une qualité minimale exigée. Une commande est également caractérisée par le prix qui sera payé lors de la livraison de la commande. Une commande est représentée par une instance de la classe `Commande` ci-contre.

marche::Commande
...
+ Commande(qualite : int, qte : int, prix : double) + getQualiteMin():int + getQuantite() : int + getPrixPaye() : float + toString() : String

Les marchés. Un marché est le lieu où sont déposées des commandes. Ces commandes sont rangées par ordre d'urgence décroissante (on ne se préoccupe pas de la manière dont ce critère d'urgence est géré) et proposées aux fournisseurs qui opèrent sur ce marché. Ces fournisseurs choisissent la commande qu'ils vont accepter d'honorer. On parle de contrat lorsqu'une commande est acceptée par un fournisseur. Chaque fournisseur ne peut accepter qu'une commande à la fois et donc n'a qu'un seul contrat actif à la fois. Les commandes sont "livrées" via le marché qui mémorise les contrats achevés.
On donne le code suivant pour cette classe `Marche` :

```
package marche;
import java.util.*;
public class Marche {
    /* les commandes rangees par ordre "d urgence" */
    private List<Commande> cmdeDisponibles;
    private List<Fournisseur> fournisseurs;
    private Map<Fournisseur, Commande> contratsActifs;
    private Map<Fournisseur, Commande> contratsFinis;
    /** initialise les attributs */
    public Marche() {   ...    }
    /** Propose les commandes disponibles a chacun des fournisseurs (les uns apres les autres)
     * qui n'a pas  encore de contrat actif.
     * Si un fournisseur choisit une commande, un nouveau contrat "actif" est enregistre.
     * La commande est retiree des commandes disponibles.
     */
    public void proposeCommandes() {    ...     }
    /** Pour chacun des contrats actifs, on demande au fournisseur s'il est prêt
     * à livrer la commande. La commande est alors livree. Le fournisseur est
     * credite du prix de la commande. Le fournisseur est a nouveau libre.
     */
    public void recupereCommandes() {
       for (Fournisseur fournisseur : this.contratsActifs.keySet()) {
            if (fournisseur.estPret()) {
                this.livre(this.contratsActifs.get(fournisseur), fournisseur);
                this.contratsActifs.remove(fournisseur);
            }
        }
    }
    private void livre(Commande commande, Fournisseur fournisseur) {
        System.out.println(fournisseur + "␣livre␣"+ commande); // pour  faire simple
        fournisseur.credite(this.contratsActifs.get(fournisseur).getPrixPaye());
        // on informe le fournisseur que sa commande actuelle ("en cours") est finie
        fournisseur.commandeTerminee();
        this.contratsFinis.put(fournisseur, commande);
    }
    /** ajoute une nouvelle commande, les commandes sont rangees par ordre "d'urgence"
     * @param commande une nouvelle commande
     */
    public void nouvelleCommande(Commande commande) {
        // gestion de "l'urgence" et ajout a la liste 'cmdeDisponibles' "dans l'ordre"
       ...
    }
    public void nouveauFournisseur(Fournisseur fournisseur) {
        this.fournisseurs.add(fournisseur);
    }
}
```

Les fournisseurs. Un fournisseur est un objet du type `marche.Fournisseur`. Un tel objet est caractérisé par son nom (une chaîne de caractères) supposé unique, son niveau de qualité de production (un entier) et son coût unitaire de production (un *double*).
Le type `Fournisseur` dispose (au moins) de la méthode `choisitCommande(List<Commande> l)` dont le paramètre est la liste des commandes disponibles transmise par le marché. Dans cette liste certaines commandes sont *acceptables* par le fournisseur, d'autres non. Pour être acceptable, il est indispensable que le niveau de qualité de production de ce fournisseur soit au moins du niveau exigé dans la commande et que son coût unitaire de production lui permette d'être en dessous du prix de la commande, pour la quantité demandée. La différence entre le prix de la commande et son coût global de production est appelée la *marge* du fournisseur.
Le résultat de la méthode `choisitCommande` est la commande choisie par le fournisseur dans cette liste ou `null` si aucune n'est acceptable.
Une exception `CommandeEnCoursException` est levée si le fournisseur a déjà une commande en cours.

Un fournisseur dispose également d'une méthode `public boolean estPret()`, dont le résultat est `true` si et seulement si la commande en cours de ce fournisseur est prête à être livrée, et d'une méthode `commandeTerminee()` qui permet de réinitialiser la commande en cours.

On identifie plusieurs types de fournisseurs qui se distinguent dans leur manière de choisir et d'accepter les commandes qu'ils reçoivent :

- Certains fournisseurs donnent priorité à l'urgence et choisissent la première des commandes de la liste `disponibles` qui est acceptable pour eux (on rappelle que dans la liste `disponibles` les commandes sont par hypothèse fournies par ordre d'urgence décroissante).
- Parmi les fournisseurs qui adoptent ce critère on en trouve qui nuancent un peu ce choix. Soucieux de leur réputation, ils n'acceptent pas des commandes pour lesquelles l'écart entre le niveau de qualité exigé dans la commande et leur propre niveau de qualité de production est supérieur à un seuil fixé à la construction de ces fournisseurs.
- D'autres fournisseurs choisissent parmi la liste de toutes les commandes disponibles celle acceptable pour laquelle ils réalisent la marge la plus importante. Ils ne se préoccupent de l'urgence qu'en cas d'égalité de marge entre différentes commandes et choisissent alors la plus urgente.

Q 1 . Donnez sous la forme de diagrammes UML clairs et détaillés (liens d'héritage, méthodes, types des attributs, des paramètres, des valeurs de retour) une proposition de modélisation pour représenter les différents types de fournisseur décrits ci-dessus.

La proposition devra être compatible avec le code fourni pour la classe `Marche`.

Q 2 . Donnez le code Java complet correspondant à cette proposition.

Pour simplifier, la méthode `estPret` renverra toujours simplement `true` et le corps de la méthode `credite`, qui apparaît dans le code de `Marche`, pourra être laissé vide.

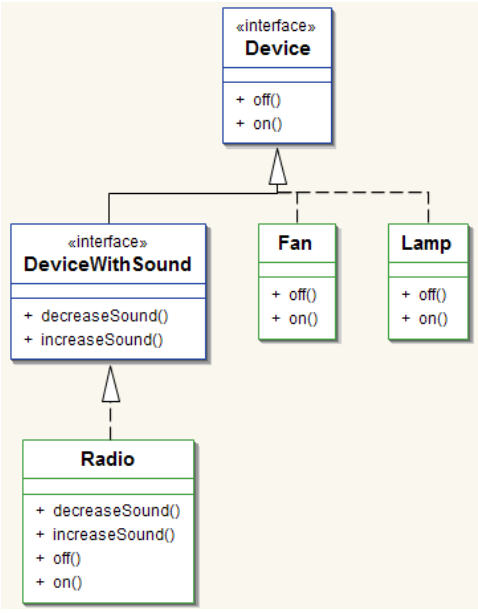
Q 3 . Pour l'une des implémentations (vous préciserez laquelle) de la méthode `choisitCommande` vous donnerez la javadoc et le code des tests (au sens JUnit) nécessaires.

Q 4 . Donnez le code du type `CommandeEnCoursException`

Exercice 4 : Télécommande programmable *(sur la copie verte)*

On souhaite modéliser des télécommandes « programmables » telles que celle présentée ci-contre. Ces télécommandes disposent d'un certain nombre de boutons dont on peut choisir et modifier l'action. Il peut s'agir d'allumer ou d'éteindre un appareil électrique ou encore d'en monter le son ou bien de passer à la chaîne suivante d'un téléviseur, etc. On souhaite donc disposer d'une classe `Telecommande` dont le constructeur sera paramétré par le nombre de boutons de la télécommande. Les appareils électriques seront du type `Device` présenté dans le diagramme ci-contre.

Chaque bouton est numéroté et la classe `Telecommande` dispose d'une méthode `public void pushButton(int i)` dont l'invocation correspond à l'appui sur le bouton numéro `i` de la télécommande. Elle doit donc déclencher l'action associée.
Il doit donc être possible d'associer à chaque bouton "`i`" l'action à déclencher.



Q 1 . Sous la forme d'un diagramme UML faites une proposition de conception pour répondre à ce problème.

La solution proposée doit permettre de programmer un bouton de la télécommande pour qu'il permette d'allumer ou d'éteindre un « `Device` » et d'augmenter ou diminuer le son d'un « `DeviceWithSound` ».

Il doit être possible d'ajouter facilement d'autres types de « `Device` » et d'autres actions sur ceux-ci doivent être possible.

Q 2 . Donnez le code de la méthode `pushButton` de `Telecommande`.

Q 3 . Donnez des lignes de code qui créent une lampe, une radio et une télécommande à 6 boutons qui permettent d'allumer et d'éteindre la lampe et la radio et de monter et baisser le son de la radio.

NB : *fan*= ventilateur