Départ. Informatique
LP DEVOPS - Module Conception d'Architectures Logicielles
Fév 2023

V. Deslandres

TD 3 - Design Patterns (1)

EXERCICE 1 – Location de biens immobiliers

Au sein de votre ESN, vous devez modéliser la location de logements pour l'agence BESTLOC, qui possède un ensemble de biens (appartements ou maisons) à louer. Les personnes peuvent directement réserver un bien (ce qui conduit à créer un pré-contrat de location avec virement d'un dépôt d'un certain montant, le pré-contrat étant ensuite validé par signature à l'agence), ou demander à résilier leur contrat.

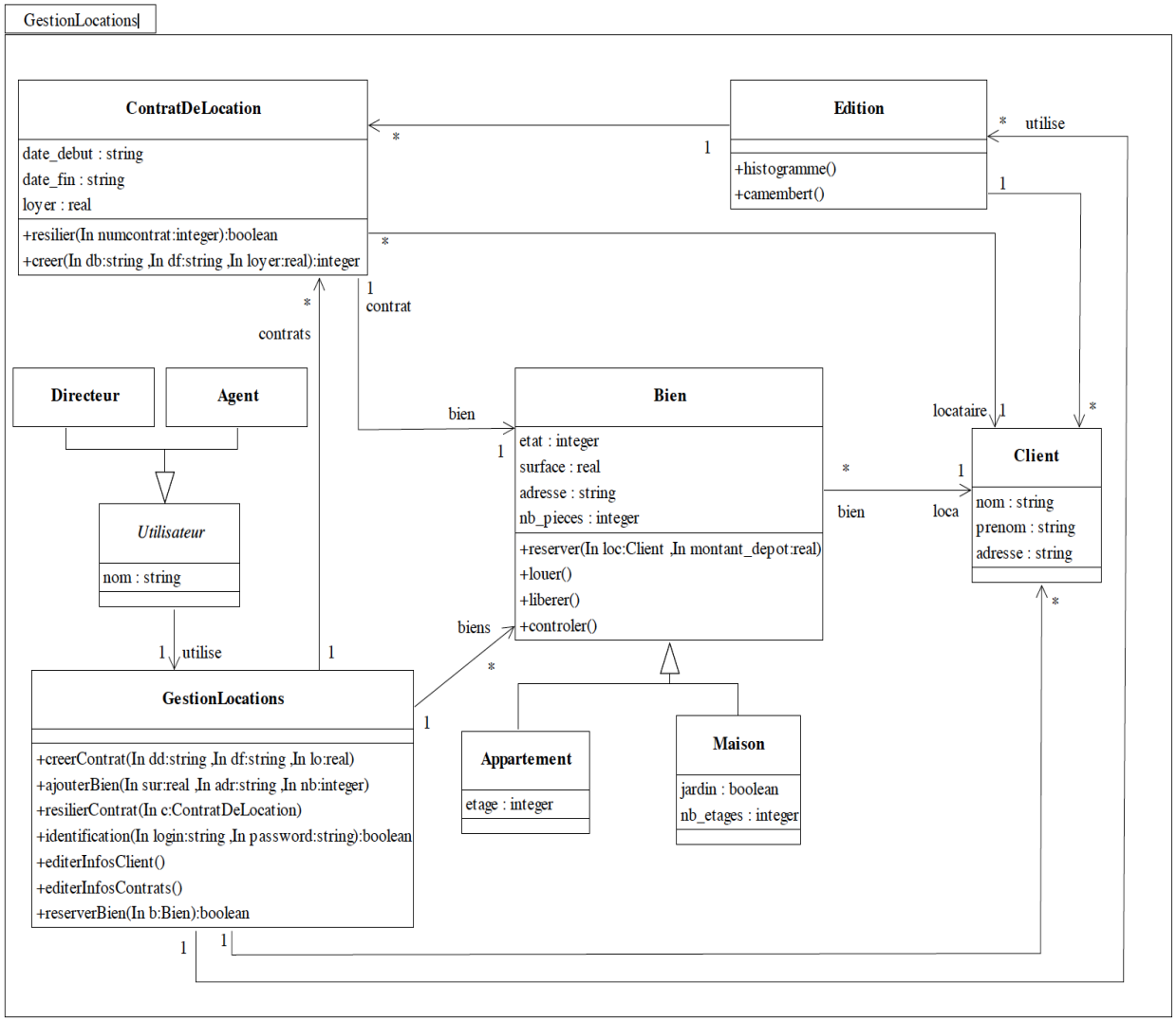
Dans cette agence, on considère qu'un bien passe par plusieurs états : libre, loué, réservé, qui conditionnent ce qu'on peut faire du Bien. Ainsi seul un bien 'libre' peut être réservé par un client ; un contrat de location est créé lorsqu'un bien 'réservé' devient 'loué'. Un bien 'réservé' ou 'loué' peut être à tout moment libéré par le client et le bien devient alors à nouveau 'libre'. L'état d'un bien est modélisé par un attribut *état* de type entier (voir la classe Bien) qui peut prendre trois valeurs correspondant aux trois états possibles du bien : 1 pour 'libre', 2 pour 'réservé' et 3 pour 'loué'.

On souhaite aussi pouvoir éditer les informations Métier (Client, Contrat) sous forme de camemberts ou d'histogrammes en fonction des durées de contrat, par ex., ou de nombre de contrats signés / résiliés sur une période pour les Clients. On apprend d'autre part que seul le Directeur peut ajouter ou supprimer des biens à louer.

1-1- L'équipe d'analyse propose une première version du DCL (ci-après). Discuter de la valeur de ce diagramme et reprener ce qui est incorrect.

1.2 – Votre chef trouve que dans la modélisation actuelle « GestionLocations », l'utilisation de l'attribut *etat* engendre une forte utilisation des tests du type « If-Then-Else » dans les opérations de la classe Bien (un *bien* ne peut par exemple être loué que s'il a été réservé auparavant). Il propose donc d'utiliser une approche plus élégante basée sur le **design pattern State.**

a – On va dans un 1er temps construire le DCL de State pour Gestion Location sous l'AGL : créer une **classe abstraite** *EtatBien* représentant les différents états du bien, qui possède le comportement (abstrait) d'un Bien : réserver, louer, libérer (3 méthodes). Définir qu'un Bien possède un état de type *EtatBien*.



b – Créer **3 classes concrètes** correspondant aux différents états du Bien, et initialiser un nouveau Bien crée et l'initialiser à *Libre*. Placer ensuite correctement les méthodes (par ex. *louer()* sera accessible uniquement dans la classe état *Réservé*). Écrire le corps des méthodes dans des notes explicatives : par exemple, *bien.setEtat(new Loué())* pour la méthode louer() de la classe état *Réservé.$*

c – **Implémenter ce diagramme en Java** et faire tourner avec un scénario nominal et un scénario d'exception (par ex. on tente de *réserver* un bien déjà *loué*) et vérifier que tout fonctionne correctement.

1.3 – Considérons maintenant qu'un bien passe obligatoirement par **un nouvel état** '*Acquis*', qui correspond à l'état d'un bien tout juste acquis par l'agence immobilière. Ce bien ne deviendra '*libre*' qu'après avoir été contrôlé (nouvelle méthode *contrôler()* de la classe *Bien*). Modifiez la solution précédente en conséquence.

1.4 – Discutez des **avantages** et des **inconvénients** de l'utilisation du pattern State par rapport à la solution initiale basée sur un attribut *état*.

Exercice 2 – Reverse Engineering

Étudier le code Java suivant :

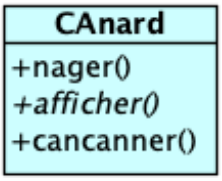
```
public class ClasseA {
    public void congeler() {
        System.out.println("Methode congeler() de la classe ClasseA");
    }

    public void rechauffer() {
        System.out.println("Methode rechauffer() de la classe ClasseA");
    }
}

public class ClasseB {
    public void mixer() {
        System.out.println("Methode mixer() de la classe ClasseB");
    }
      public void hacher() {
        System.out.println("Methode hacher() de la classe ClasseB");
    }
}
public class ClasseC {
    private ClasseA maClasseA ;
    private ClasseB maClasseB ;

    public ClasseC() {
      maClasseA = new ClasseA();
      maClasseB = new ClasseB();
    }

    public void rechauffer() {
      System.out.println("-> Méthode rechauffer() de la classe ClasseC: ");
      maClasseA.rechauffer();
    }

    public void conserver() {
      System.out.println("-> Méthode conserver() de la classe ClasseC : ");
      maClasseB.hacher();
      maClasseA.congeler();
    }
}

public class TestMain {

    public static void main(String[] args) {

        ClasseC unObjetC = new ClasseC();
        unObjetC.rechauffer();
        unObjetC.conserver();

    }
}
```

2.1- Sans faire tourner ce code, dites ce qui sera affiché en sortie.

2.2- **Identifier le Design Pattern utilisé** avec ClasseC.

EXERCICE 3 – SuperCanard

Vous êtes développeur dans une société de jeux, où le gros succès repose sur des jeux de pédagogie et de simulation, basés sur des canards (avec un héros SuperCanard). On a donc des applications où on doit représenter et manipuler des canards de toute sorte. Les concepteurs du jeu ont créé une super classe Canard, dont héritent différentes sous-classes de canards avec une hiérarchie qui permet d'envisager la réutilisation pour les applications de la société :



3.1- Tous les canards nagent et une grande majorité cancanent. Une amélioration du jeu est prévue qui prévoit de **faire voler** les canards en plus de nager. Vous envisagez dans un 1er temps d'ajouter une méthode *voler()* à la super classe Canard. Mais vous aviez oublié qu'une des sous-classes était :

Voir des canards en plastique traverser l'écran ne satisfait pas votre chef de projet… De plus il s'avère qu'un canard en plastique *couine* : il ne cancane pas…

Quelles solutions de conception permettraient d'éviter que des canards en plastique volent et cancanent ? Comment gérer tous ces canards qui partagent des éléments tout en ayant leurs propres caractéristiques ? Réfléchir à leurs avantages/inconvénients.

*Indice : **identifier ce qui varie** de **ce qui est stable** d'un canard à l'autre.*

Avec l'AGL, vous allez élaborer **une conception** qui permette d'autre part de :

- Affecter **dynamiquement** un comportement à une instance ;
- Décrire le comportement **par un ensemble d'attributs**, par ex. *pour le vol* : nombre de battements d'ailes par minute, altitude max, vitesse max, etc.; pour le *cri* : plage de décibels par ex.) ;
- **Ajouter un nouveau type de comportement**, par exemple de vol : *vol à propulsion* pour une classe *Canardator…*
- Pouvoir **réutiliser** le cancannement pour une autre application demandée par le CPT (Chasse Pêche et Traditions), où ils ont une classe Appeau (= instrument qui imite le cri du canard utilisé par les chasseurs).

3.2- Une fois le modèle défini avec l'AGL, **générer le code java**. Pour cela :

- Compléter avec les éléments de conception nécessaire (identifiants, types),
- Migrer toutes les classes dans un package (Clic droit sur le DCL - Convertir en package)
- Ajouter les attributs qui vous semblent compléter l'application (attributs et méthodes propres aux sous-classes),
- Puis générer le code Java de l'application : menu **Langage – Générer le code Java**.
- Modifier le code sous un IDE et tester l'application : ça doit tourner !

3.3.- Quel Design Pattern avons-nous utilisé ici ?

EXERCICE 4 – Recettes de cuisine

4.1 - Préparation

Modéliser une classe "Ingrédient" qui possède les méthodes :

- *toString()* qui renvoie le nom de l'ingrédient et les informations suivantes
- *estSucre(), estSale(), estSucreSale(),*
- le *nombreDeCalories()*
- *estDietetique()* : retourne *true* si le nombre de kcalories est inférieur à 200.

Créer les classes Sucre, Sel, Poire, Pomme, Framboise, Veau, Bœuf, Chocolat, Carotte, HaricotVert, qui sont des ingrédients.

4.2 - Plat composé

On souhaite représenter un plat composé (un bœuf carotte par exemple, ou une poire Belle Hélène).

Dans ce cas, la méthode *toString()* renvoie la liste des ingrédients du plat composé. On dira qu'un plat composé est *sucré* si au moins un de ses éléments est sucré, *salé* si un au moins est salé et *sucré-salé* quand au moins un de ses éléments est sucré et un autre salé.

Comment procéder ? Modéliser une solution en UML puis générer le code Java. Tester votre implémentation avec quelques objets.

Indice : vous pouvez renommer le nom des classes proposées dans l'énoncé pour 'coller' au pattern.

EXERCICE 5 – Jeu d'aventures

Imaginez que vous ayez à concevoir un jeu d'aventure, avec des personnages : roi, reine, chevalier, troll, etc. Chaque personnage peut faire usage d'armes variées : arc et flèches, poignard, hache, épée, mais une seule à la fois. Par contre le personnage peut en changer au cours de son aventure.

Quel design pattern serait utile ici ? Proposez une modélisation sous l'AGL (sans l'implémenter en Java).