IUT Lyon 1
l'excellence technologique

Départ. Informatique
LP DEVOPS - Module Conception, Modélisation et Méthodes Agiles

V. Deslandres

TD 2 - Design Patterns

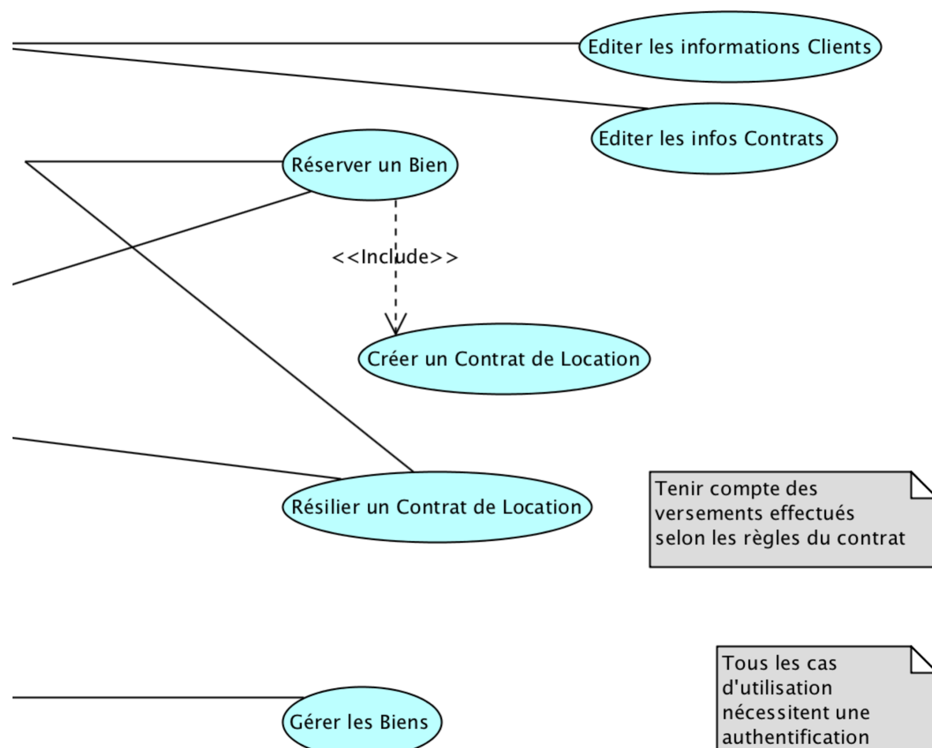
EXERCICE 1 – Location de biens immobiliers

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.

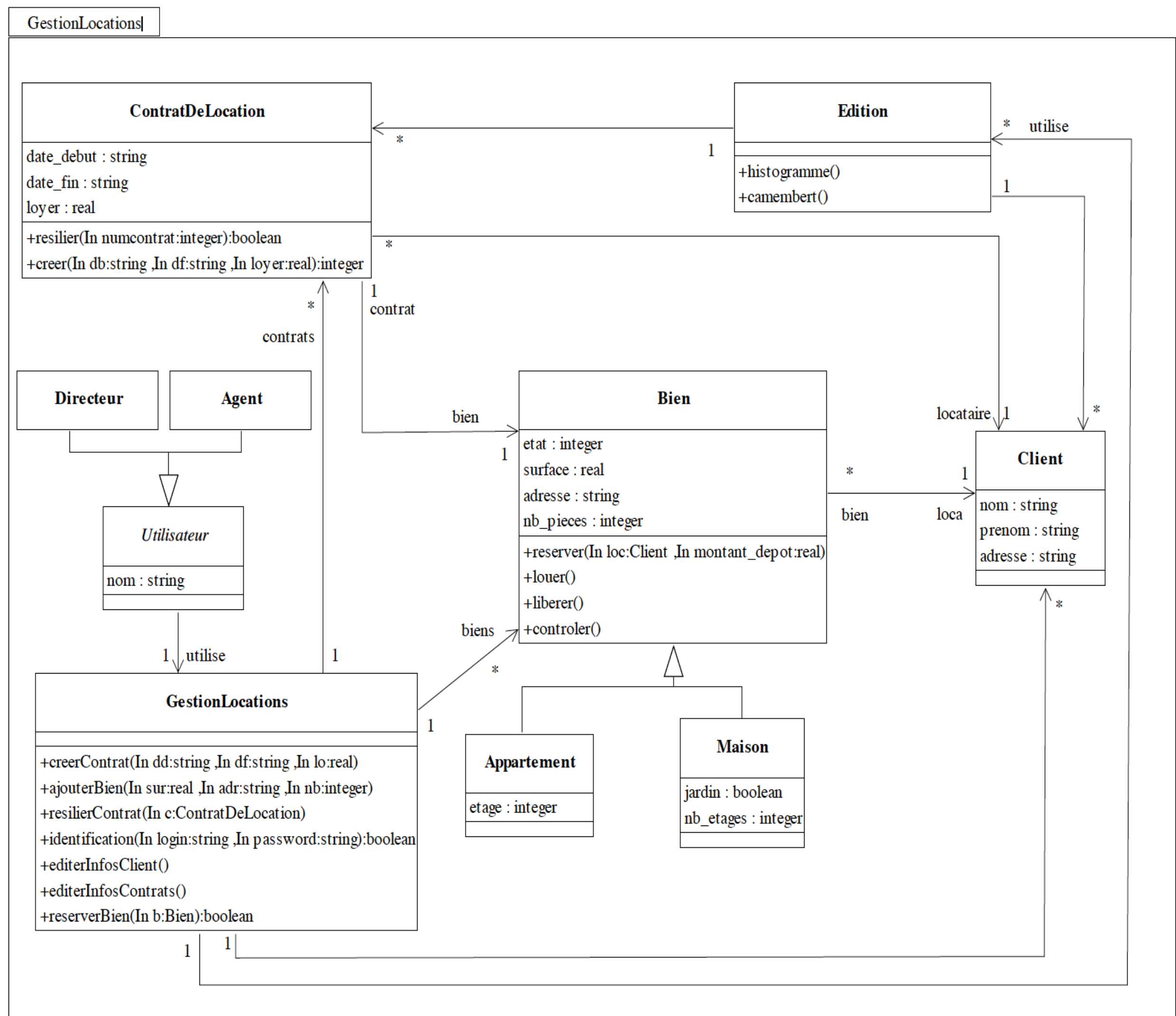
Dans cette agence, on considère qu'un bien passe par plusieurs états : libre, loué, réservé, qui conditionnent ce qu'on peut faire du Bien. Ainsi seul un bien 'libre' peut être réservé par un client ; un contrat de location est créé lorsqu'un bien 'réservé' devient 'loué'. Un bien 'réservé' ou 'loué' 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- Compléter le Diagramme des Cas d'utilisation de l'application, proposé ci-dessous pour l'agence.



2- L'équipe d'analyse propose une première version du DCL (ci-dessous). Corrigez ce qui est incorrect.



3 – Votre équipe considère finalement que la classe *Edition* devrait être réutilisable pour d'autres applications de votre ESN.
→ Modifiez le diagramme pour que la classe *Edition* soit **dans un package séparé** du reste de l'application, et montrez dans quel sens se fait la dépendance.

4 – Elaborez un **diagramme de séquence objet** correspondant au comportement nominal de l'opération *reserverBien()* de la classe GestionLocations. Faire de même pour un scénario d'exception.

5 – 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 ».**

5.1 – Créer une **classe abstraite** *EtatBien* représentant les différents états du bien, qui possède le comportement (abstrait) d'un Bien : réserver, louer, libérer. Définir qu'un Bien possède un état de type *EtatBien*.

5.2 – Créer **3 classes concrètes** correspondant aux différents états du Bien, avec les méthodes correctement placées (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.*

6 – 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.

7 – 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 : Classe statique vs. Singleton

Certains développeurs font des classes statiques au lieu de Singleton. L'objectif de n'avoir qu'une instance (la classe elle-même) est atteint, le compilateur interdisant d'instancier une classe statique.

Certains pensent donc que Singleton et classe `static` sont effectivement la même chose.

2-1. Quelle est pourtant la **grosse différence** ?

2-2. Quel **autre design pattern** connaissez-vous qui contrôle l'accès à un objet (pas d'accès direct) ?

Exercice 3 – 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();

    }
}
```

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

2- Procédez à une opération de *Reverse Engineering* avec PowerAMC, étudier le DCL obtenu et identifier le Design Pattern utilisé avec la ClasseC.

EXERCICE 4 – Gestion de Franchises

Imaginons la mini-spécification suivante. Nous gérons des marques et leurs franchises associées (ex. : une marque sera décrite par son nom et l'ensemble de ses franchises. Une marque doit être validée avant de posséder des franchises (par ex. ici, avoir un nom non nul). On peut ajouter des franchises ou en supprimer dans une marque donnée. Pour supprimer une marque, nous devons nous assurer qu'il n'existe pas de franchise associée.

Expliquer rapidement comment représenter cette spécification avec le bon pattern.

EXERCICE 5 – Quel pattern ?

Dans cette application, on effectue des recherches sur des mots-clefs. On a donc une interface Search :

```
public interface Search {

    String searchFor(String... keywords) throws Exception;
}
```

On définit une première classe d'implémentation de **`Search`** :

```
public class OneSearch implements Search {

    @Override
    public String searchFor(String... keywords) {
        return "Result: " + String.join(" ", Arrays.asList(keywords));
    }
}
```

Puis une deuxième classe d'implémentation de **`Search`** :

```
public class ControlledSearch implements Search {

    private static Set<String> bannedSearches = new
HashSet<>(Arrays.asList("China", "Russia", "Iran", "Pakistan"));

    private OneSearch searchService;

    public ControlledSearch() {
        this.searchService = new OneSearch();
    }

    @Override
    public String searchFor(String... keywords) throws Exception {
        for (String keyword : keywords) {
           if (bannedSearches.contains(keyword.trim().toLowerCase())) {
               throw new Exception(keyword.toUpperCase() + " is a
restricted keyword!");
            }
        }
        return this.searchService.searchFor(keywords);
    }
}
```

Notre application n'utilise que **`ControlledSearch`**. Quel pattern avons-nous utilisé dans cette deuxième classe ?