

---

République Algérienne Démocratique et Populaire

الجمهورية الجزائرية الديمقراطية الشعبية

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

وزارة التعليم العالي و البحث العلمي

---



المدرسة الوطنية العليا للأعلام الآلي

Ecole nationale Supérieure d'Informatique

## **Projet PDC : Amélioration de la conception du projet de POO réalisé en 2CP en utilisant les patrons de conception.**

**Réalisé par :**

- TERRAS Juba
- AMOKRANE Ilhem.

**Groupe : 02**

**Spécialité :   Systèmes informatiques et logiciels (SIL).**

---

**Promotion : 2021-2022**

# Table des matières

I.	Introduction :	3
A.	Rappels sur les patrons de conceptions :	3
1.	Les principes du SOLID :	3
2.	Les patrons GRASP :	3
B.	Objectif du projet :	4
C.	Présentation de l'application à analyser :	4
II.	Analyse du système :	5
A.	Analyse du code source :	5
B.	Reconstitution du diagramme de classe :	5
C.	Identification des problèmes :	7
1.	Administrateur :	7
2.	Date :	7
3.	Bien :	7
4.	Habitable :	8
5.	Maison, Appartement, Terrain :	9
6.	Gestion :	10
III.	Solutions Proposées :	11
1.	Administrateur :	11
2.	Date :	11
3.	Bien :	11
4.	Maison , Appartement et Terrain :	12
5.	Gestion :	12
6.	Toutes les classes :	13
D.	Elaboration du nouveau diagramme de classe après amélioration :	14
IV.	Conclusion :	15

## I. Introduction :

### A. Rappels sur les patrons de conceptions :

Le terme « patron de conception » tire son origine des travaux de l'architecte américain Christopher Alexander et de sa compilation de modèles réutilisables. Son intention était d'impliquer les futurs utilisateurs des bâtiments dans le processus de conception.

Cette idée a ensuite été reprise par différents informaticiens. Le Gang of Four (bande des quatre), Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, ont aidé à populariser les patrons de conception grâce au livre **Design patterns**.

En génie logiciel, un patron de conception (Design pattern en anglais) est une solution générique d'implémentation répondant à un problème spécifique. Les patrons de conception décrivent des procédés de conception généraux et permettent de capitaliser l'expérience appliquée à la conception de logiciel. Ils ont une influence sur l'architecture logicielle d'un système informatique.

#### 1. Les principes du SOLID :

**Responsabilité Unique (SRP) :** Un objet fait une et une seule chose.

**Principe Ouvert/Fermé (OCP) :** Une évolution du projet minimise le nombre de modifications et exploite les capacités d'extensions.

**Substitution de Liskov (LSP) :** Les sous-types doivent être des substituts pour leur ancêtres.

**Ségrégation des interfaces (ISP) :** On préfère des interfaces spécialisées à des fourre-tout.

**Inversion de dépendances (DIP) :** Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstraction.

#### 2. Les patrons GRASP :

**Spécialiste de l'Information :** Donner la responsabilité à la classe qui connaît les informations permettant de faire cela.

**Créateur :** Affecter à la classe C la responsabilité de création des instances de C si par exemple : C est composée d'instances de C' ou C à des données permettant d'initialiser les instances de C'.

**Faible couplage :** Minimiser les dépendances entre les objets et réduire l'impact des changements.

**Contrôleur :** Inventer un objet qui va servir de zone tampon entre le système et l'application objet.

**Forte cohésion :** Attribuer les responsabilités de telle sorte que la cohésion soit forte.

**Polymorphisme :** Affecter la responsabilité aux types et en proposer plusieurs réalisations alternatives qui peuvent être interchangeables.

**Fabrication pure :** Affecter un ensemble de responsabilités fortement cohésives dans une classe créée artificiellement pour l'occasion.

**Indirection :** introduire un élément dédié à ce couplage pour laisser les éléments préexistants isolés.

**Protégé des Variations :** Trouver ce qui varie et l'encapsuler dans une interface stable.

### **B. Objectif du projet :**

L'objectif de ce projet est d'appliquer les principes avancés de conception orientée objet ainsi que les patrons de conception pour améliorer la conception faite lors du TP de POO en 2CP.

On sera amené à effectuer un travail de réingénierie en analysant un système existant afin d'identifier les problèmes de conception et de les résoudre en utilisant les patrons de conception.

### **C. Présentation de l'application à analyser :**

L'application à analyser est développée pour une agence immobilière afin de lui permettre de proposer des biens immobiliers, que ce soit des appartements, des maisons où même des terrains à la vente, à la location ou à l'échange.

Le système offre deux types d'accès différents :

**Accès administrateur :** où un agent administrateur est responsable de publier les biens proposés par les propriétaires, de modifier ces biens, de les supprimer ou les archiver, comme il peut afficher pour un propriétaire donné les biens qu'il possède.

**Accès invité :** ou un invité peut consulter tous les biens disponibles en les triant par ordre décroissant de la date d'ajout. De plus, il permet aux utilisateurs de visualiser tous les détails d'un bien et de filtrer selon un ou plusieurs critères.

## II. Analyse du système :

### A. Analyse du code source :

Le code source se compose exactement de 49 classes :

- 17 classes sont utilisés pour le noyau fonctionnel de l'application.
- 16 classes sont utilisés sous forme d'interfaces graphiques.
- 16 classes sont utilisées sous forme de contrôleurs pour maintenir un lien fonctionnel entre le noyau et l'IHM.

Le code source est repartie selon 4 packages :

- Package **Noyau** : renfermant les classes du noyau.
- Package **UserInterface** : renfermant les interfaces graphiques de l'application.
- Package **Contrôleurs** : renfermant les classes contrôleurs.
- Package **Illustrations** : renfermant toutes les ressources extérieures dont a besoin l'application pour fonctionner correctement tel que : des images, des icones etc...

Le code a été écrit à la base afin de respecter le principe d'encapsulation mais les classes se retrouvent submergées de guetteurs et setters qui ne sont pas utilisés.

Certaines classes sont plus couplées que d'autre et plus particulièrement la classes « Bien » qui possède un nombre énorme de dépendances.

Nous avons aussi remarqué de la redondance de code ce qui viole complètement le principe de DRY.

L'avant dernier point, nous avons remarqué de la faible cohésion et plus précisément au niveau de classe gestion qui se voit assigner un nombre énorme de responsabilités.

Enfin les possibilités d'extension du système restent modestes et limitées.

### B. Reconstitution du diagramme de classe :

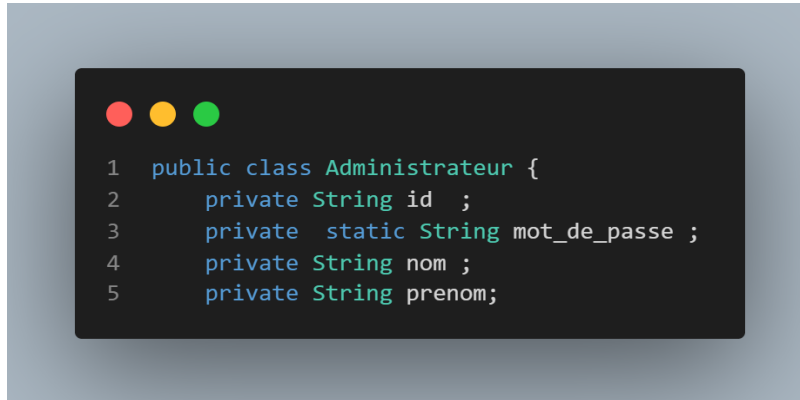
Après avoir bien analyser le code source nous avons sortie le diagramme de classe suivant, en respectant les normes et les notations du format UML :



## C. Identification des problèmes :

Dans cette partie nous allons analyser chaque classe afin de repérer les problèmes liés à la conception.

### 1. Administrateur :

A screenshot of a code editor window with a dark background and light-colored text. The code defines a Java class named 'Administrateur'. It has five lines of code: a public class declaration, and four private attributes: 'id' (String), 'mot\_de\_passe' (static String), 'nom' (String), and 'prenom' (String). The code is numbered 1 through 5 on the left side of the editor.

```
1 public class Administrateur {  
2     private String id ;  
3     private static String mot_de_passe ;  
4     private String nom ;  
5     private String prenom;
```

- Pour cette classe le seul problème repéré concerne les deux attributs « nom » et « prenom » que nous jugeons inutiles (du code en plus).

### 2. Date :

- Cette classe n'a pas lieu d'être, car premièrement elle rajoute un couplage inutile avec la classe Bien, de plus il existe un type prédéfini dans Java qui permet de manipuler des objets de type Date.

### 3. Bien :

#### i. Les principes de base :

**Abstraction :** On remarque que la classe « Bien » a été déclarée comme étant une classe concrète au lieu de classe abstraite. Ce qui va à l'encontre du principe de réutilisation et d'extension.

**Héritage :** D'après la conception réalisée, on remarque qu'elle prend en considération la possibilité d'ajout de biens habitables et cela grâce à une relation de généralisation entre la classe « Bien » et « Habitable », mais cette dernière ne prend pas en considération la possibilité d'ajout de bien non habitables dans le système.

#### ii. Les principes du SOLID :

**DRY:** Le fait de ne pas avoir déclaré la classe « Bien » comme une classe abstraite a forcé les concepteurs à donner un corps aux méthodes : `calculer_prix_vente()`, `calculer_prix_echange()`, `calculer_prix_location()` et `modifier()` et le dupliquer dans les sous-classes de la classe « Bien ».

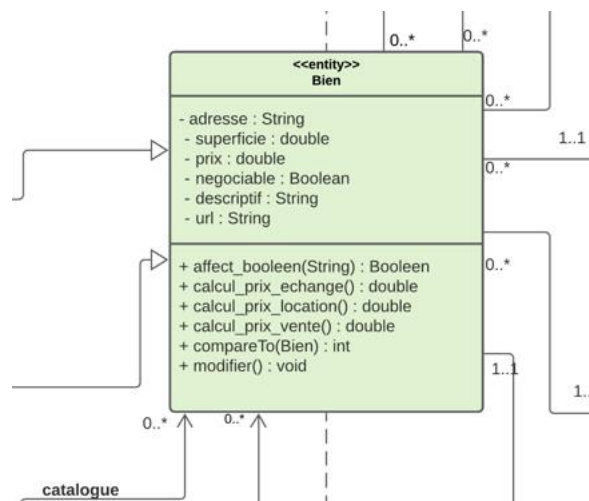
```

1  /******
2  /*Les methodes de calcul de prix*/
3  public double calcul_prix_vente() { return 0; }
4  public double calcul_prix_location() { return 0; }
5  public double calcul_prix_echange() { return 0; }
6  /******
7  /*surcharge de la methode de modification*/
8  public void modifier() {}
9  public int getnbpieces(){return 0;}
10 /******

```

### iii. Les patrons GRASP :

**Faible couplage :** On remarque que cette classe est couplée à un nombre important de classes, ce qui va à l'encontre du patron faible couplage. Par conséquent on aura une difficulté de maintenance du code et une possibilité d'extension modérée dans le futur.



## 4. Habitable :

### i. Les principes de base :

**Abstraction :** On voit d'après la conception, que les concepteurs n'ont pas réfléchi à un éventuel ajout de nouveaux comportements pour les biens habitables car « Habitable » est déclarée comme étant une classe concrète et le moindre changement de besoins entraînera une modification du code.

### ii. Les principes SOLID :



**DRY** : on remarque la duplication inutile du code de la méthode `modifier()` déjà présente dans la classe mère « Bien » et les sous classe « Appartement » et « Maison » qui ne sera jamais utilisée au sein de la classe « Habitable », car on aura jamais à instancier un objet de type « Habitable ».

### iii. Les patrons GRASP :

**Forte cohésion** : La présence de la méthode `afficher_bien()` ne fait que diminuer la cohésion de la classe puisque d'une part on instancie jamais la classe `Habitable`, et d'autre part, cette méthode existe déjà dans les deux sous-classes « Maison » et « Appartement ». Donc c'est un ajout inutile.

## 5. Maison, Appartement, Terrain :

### i. Les patrons GRAPS :

**Variation protégées (ne pas parler aux inconnus)** : Ce principe n'est pas respecté car les classes « Maison », « Appartement », « Terrain » accèdent directement à la classe « Wilaya » via l'appel suivant `getWilaya().getprixmettre carre()`. Or ces dernières ne possèdent pas de lien direct avec la classe « Wilaya ». Pareil pour l'appel `getwilayaechange().getnumwilaya()`.

```

1 public double calcul_prix_location() {
2     double nvprix;
3     if(getsuperficie()<60)
4     {
5         if(getwilaya().getprixmettre carre() < 50000)
6         {
7             nvprix=(getprix() *1)/ 100 + getprix();
8         }
9         else
10        {
11            nvprix=(getprix() *1.5)/100 +getprix();
12        }
13    }

```

```

1 public double calcul_prix_echange() {
2     double prix_echange;
3     prix_echange=calcul_prix_vente();
4     if(getwilayaechange().getnumwilaya()!=getwilaya().getnumwilaya()) {
5         prix_echange=prix_echange+(getprix()*0.25)/100;
6     }
7     return prix_echange;
8 }

```

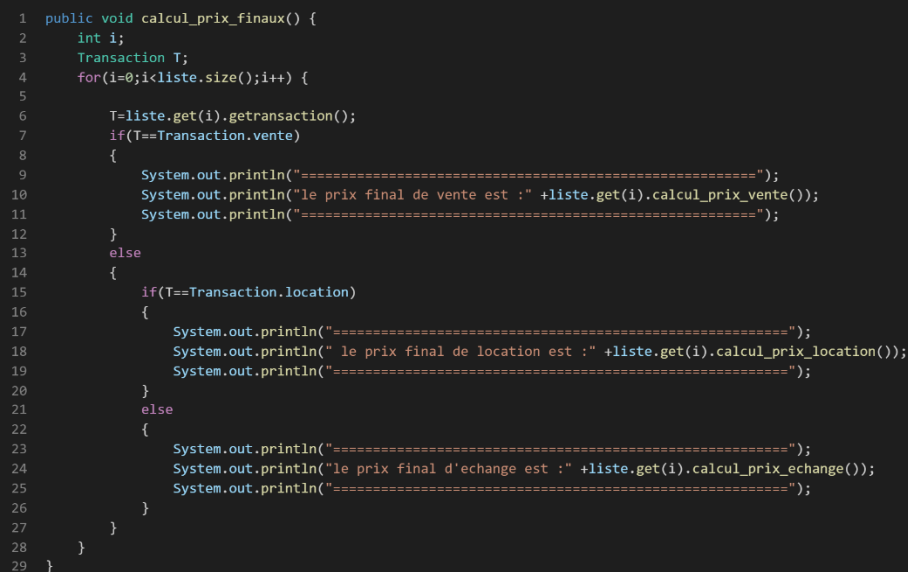
## 6. Gestion :

### i. Les principes SOLID :

**SRP:** on remarque que la classe gestion est submergée par de nombreuses responsabilités notamment ce qui concerne : la gestion des biens, la gestion de l'archivage et la gestion de la boîte aux lettres. Ce qui contredit le principe de la responsabilité unique.

**DIP:** on remarque que la classe « Gestion » manipulent des collections d'objets de type « Bien » qui implémente l'interface comparable, or d'après DIP on doit dépendre d'abstractions et non pas de classes concrètes.

**OCP:** La classe « Gestion » ne respecte pas du tout le principe OCP, et cela se voit notamment dans la méthode calculer\_prix\_finaux() ou on retrouve un if else , pour calculer le prix du bien selon la transaction. Le fait de rajouter un nouveau type de transaction a l'avenir entrainer la modification du code.



```

1 public void calculer_prix_finaux() {
2     int i;
3     Transaction T;
4     for(i=0;i<liste.size();i++) {
5
6         T=liste.get(i).gettransaction();
7         if(T==Transaction.vente)
8         {
9             System.out.println("=====");
10            System.out.println("le prix final de vente est :"+liste.get(i).calculer_prix_vente());
11            System.out.println("=====");
12        }
13        else
14        {
15            if(T==Transaction.location)
16            {
17                System.out.println("=====");
18                System.out.println(" le prix final de location est :"+liste.get(i).calculer_prix_location());
19                System.out.println("=====");
20            }
21            else
22            {
23                System.out.println("=====");
24                System.out.println("le prix final d'echange est :"+liste.get(i).calculer_prix_echange());
25                System.out.println("=====");
26            }
27        }
28    }
29 }

```

### ii. Les Patrons GRASP:

**Forte Cohésion:** Comme la classe « Gestion » ne respecte pas le principe SRP, ceci provoque une faible cohésion entre les méthodes de la classe.

**Pure Fabrication:** les concepteurs de cette classe n'ont pas respecté ce principe, et cela en refusant de déléguer certaines responsabilités à de nouvelles classes fabriquées artificiellement.

### III. Solutions Proposées :

#### 1. Administrateur :

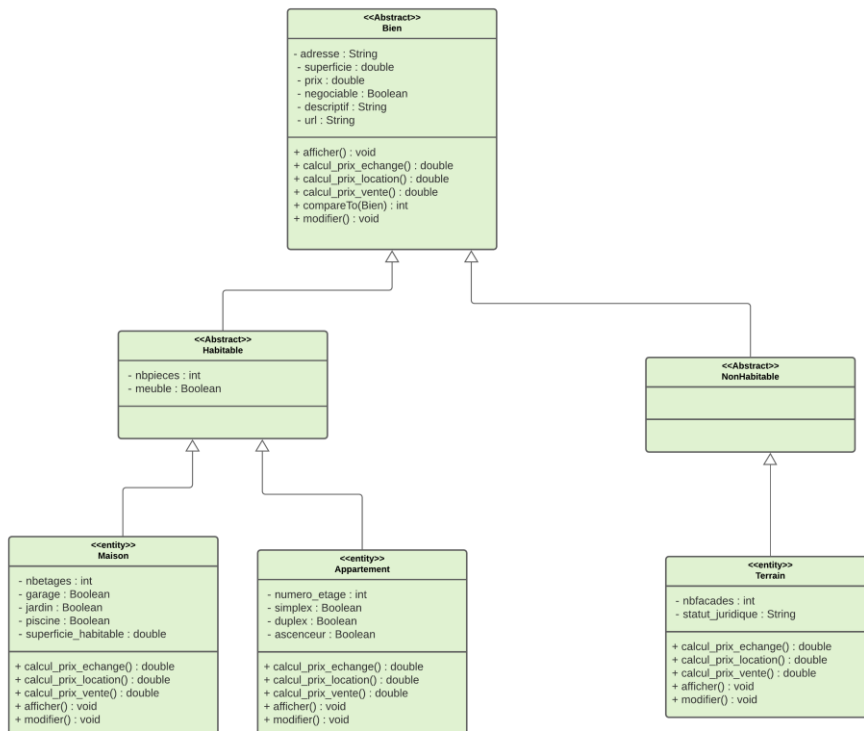
Pour éviter l'utilisation du code inutile, il est préférable de supprimer les attributs « nom » et « prénom ».

#### 2. Date :

Vu la disponibilité d'un type prédéfinie dans java pour la manipulation des objets de type DATE, la classe Date n'a plus de raison d'exister dans le système.

#### 3. Bien :

- Pour remédier au problème d'abstraction, on transforme les classes « Bien » et « Habitable » en classes abstraites, et on déclare les quatre méthodes suivantes comme méthodes abstraites : `calculer_prix_vente()`, `calculer_prix_location()`, `calculer_prix_echange()`, `modifier()`.
- Concernant l'héritage, la solution la plus adéquate est d'ajouter une nouvelle classe abstraite « NonHabitable » qui va dériver de la classe « Bien » pour anticiper l'ajout de nouveaux types et comportements de biens non habitables ainsi la classe « terrain » va dériver de la classe « NonHabitable ».

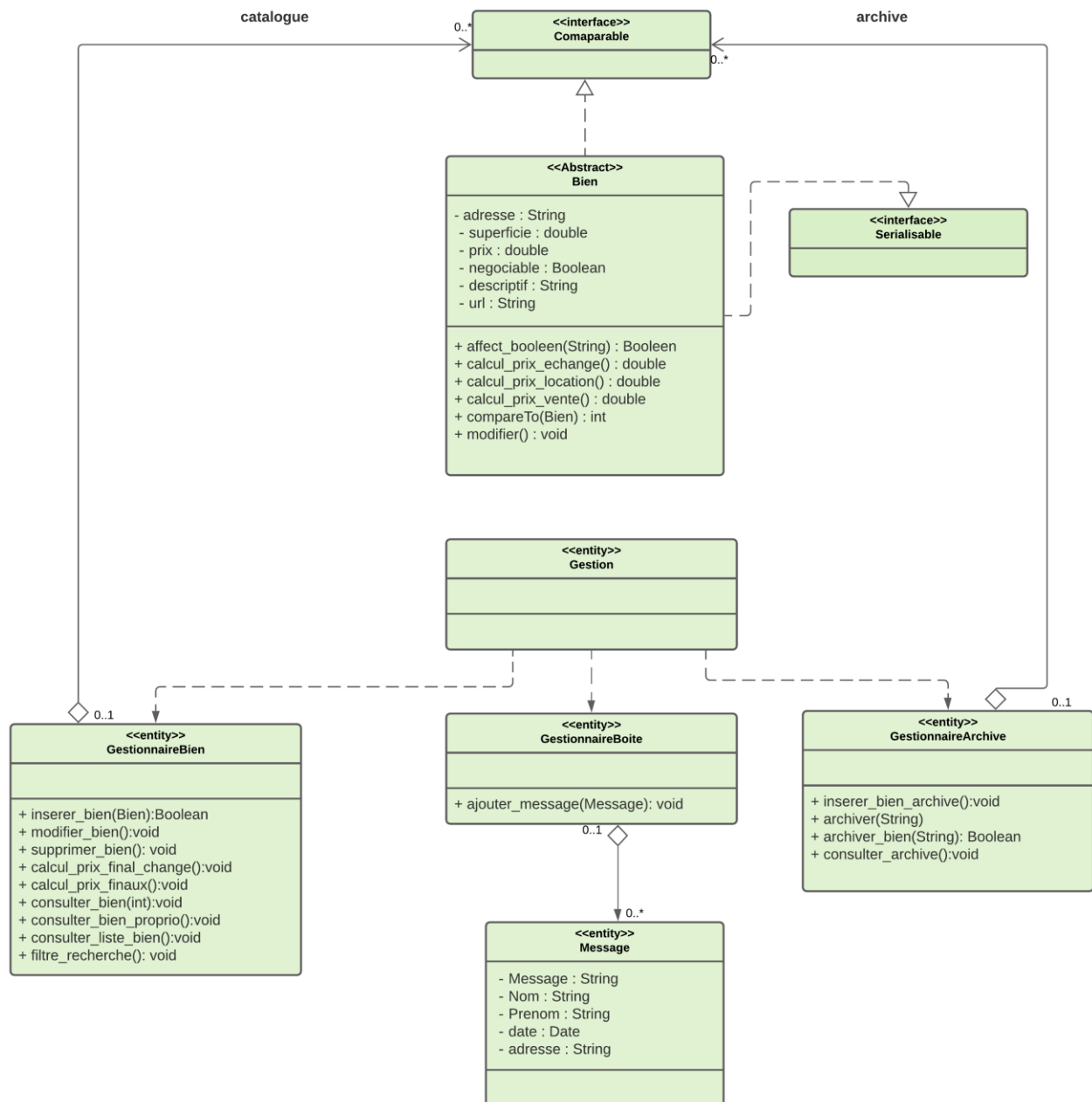


#### **4. Maison, Appartement et Terrain :**

- Pour résoudre le problème des variations protégées (Ne pas Parler aux Inconnus), il suffit d'imposer aux classes filles qui sont « Maison », « Appartement » et « Terrain » d'utiliser les méthodes fournies par leur classe mère « Bien ».

#### **5. Gestion :**

- Commençons par résoudre le problème de responsabilité unique (SRP) , nous allons appliqué le patron de pure fabrication , pour créer trois nouvelles classes artificielles :  
« GestionnaireBien », « GestionnaireArchive » et « GestionnaireBoite » qui auront chacune les responsabilités adéquates, ce qui va engendrer une forte cohésion au sein de ces nouvelles classes et dans la classe «Gestion » d'où l'élimination du problème faible cohésion.
- A présent, pour éliminer la violation du principe DIP, la classe « Gestion » doit manipuler directement des objets de types « Comparable » au lieu des objets de type concret « Bien »

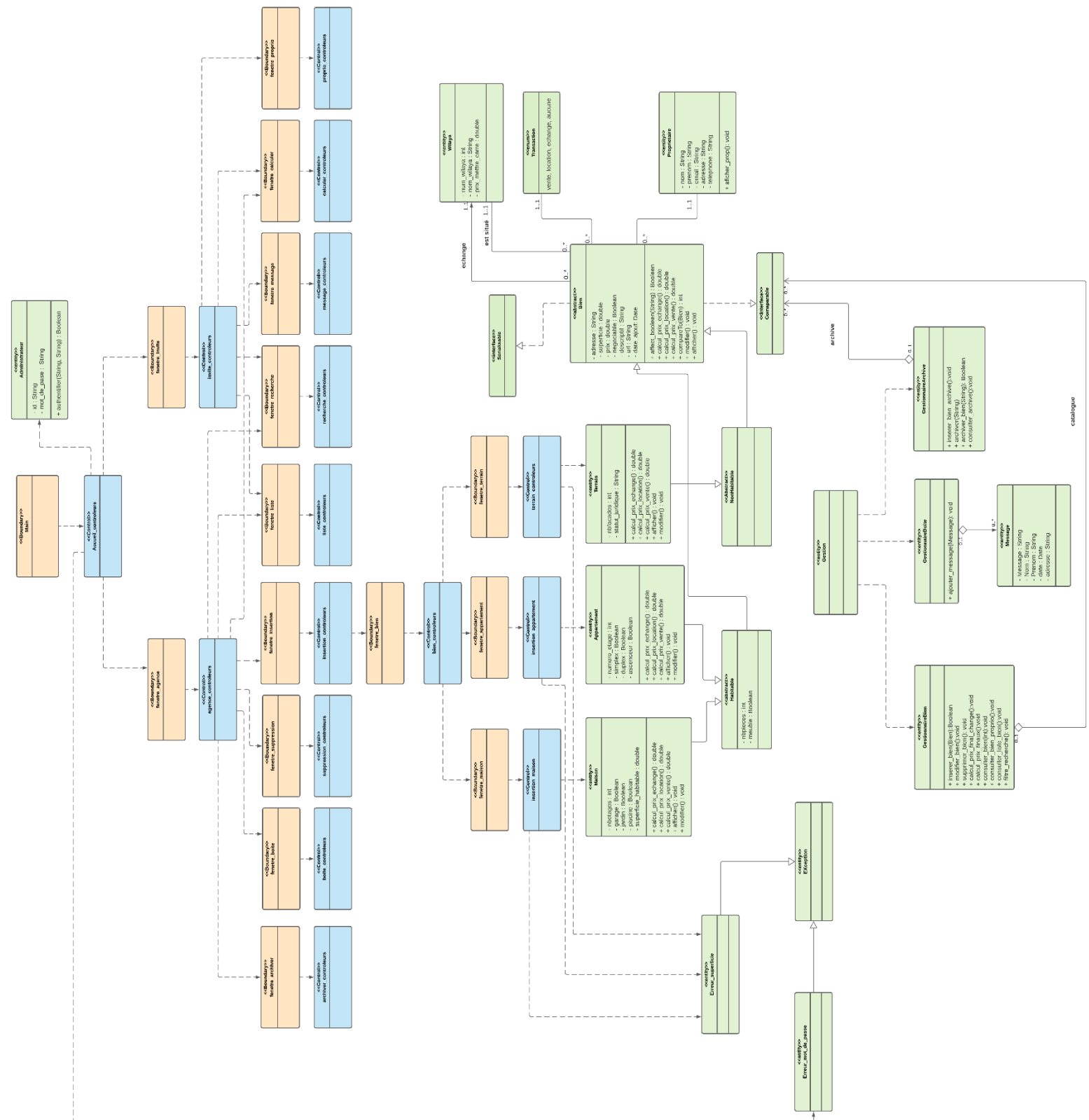


## 6. Toutes les classes :

On peut ajouter de nouvelles améliorations qui s'appliquent à n'importe quelle classe :

- Eliminer les guetteurs et setters inutilisés.
- Utiliser la notation Upper Camel Case pour les noms de classes.
- Utiliser la notation lower Camel Case pour les noms des méthodes ainsi que les attributs.

#### D.Elaboration du nouveau diagramme de classe après amélioration :



#### **IV. Conclusion :**

Après avoir bien analyser le code source, on a constaté que les concepteurs se sont focalisés sur la solution elle-même sans prendre en considération la qualité du logiciel fournie.

En se référant à ce que nous avons appris en cours concernant les patrons de conception, on a pu proposer de nouvelles solutions qui rendent l'application plus facile à maintenir avec une possibilité d'extension non négligeable à l'avenir, tout en veillant a la possibilité de réutilisation du code.