

Examen de rattrapage

Vendredi 22 juin 2007

Durée : 2h

Tous documents autorisés !

Toute réponse doit impérativement être justifiée !!!

Questions de Cours (4 points)

C1. En quoi un cas de test abstrait est-il différent d'un cas de test concret ? (1 point)

Correction : Test abstrait n'est pas exécutable, l'autre oui.

100% si tout

0% sinon

C2. Peut-on dire qu'UML est une méthode de développement de logiciel ? Justifiez votre réponse. (1 point)

Correction : Non UML n'est pas une méthode. UML n'est qu'un langage de modélisation proposant un ensemble de notations (diagrammes) pour la modélisation de logiciels. Il ne propose pas des étapes comme toute méthodes de développement de logiciels.

100% si non avec justification

0% sinon

C3. Présentez brièvement les dépendances entre classes visibles dans le diagramme de classes UML de la figure 1. (1 point)

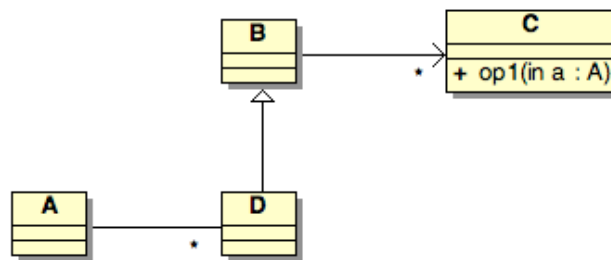


Figure 1 – Un diagramme de classes UML

Correction :

D dépend de B - héritage

B dépend de C - association navigable

C dépend de A - paramètre d'opération typé par A

100% si tout

-33% par dépendance non justifiée (cause non établie) ou oubliée

C4. Considérons les deux classes A et B définies dans la figure 2. Tout objet instance de A est un agrégat d'objets instances de B. Que signifie la destruction d'un objet instance de A pour ses objets agrégés instances de B ? (1 point)



Figure 2 - Agrégation de classes

Correction : les cycles de vie ne sont pas liés, donc la destruction de l'objet de type A n'entraîne pas celle des objets de type B.

100% si tout

0% sinon

Exercice 1 (6 points_[CB1])

<pre> import java.util.Iterator ; import java.util.ArrayList ; package UEGenerique ; public class UE { private Iterator iter ; private ArrayList<Bien> liste ; public void chercher(int id){ liste.get(id) ; } public void ajouter (Etudiant e) { liste.add(e) ; } public void supprimer (String id){ liste.remove(id) ; } </pre>	<pre> package Utilisateurs; public class Etudiant{ public int rangEtudiantUE(UE monUE) { //calcul du rang de l'étudiant //dans l'UE monUE } } </pre>
---	--

1.1. Donnez le diagramme de classes UML obtenu en appliquant strictement les règles de reverse engineering vues en cours sur le code ci-dessus. (3 points)

Correction :

70% package UEGenerique

5% package

10% association vers ArrayList avec 5% rôle liste + 5% cardinalité

10% association vers Iterator avec 5% rôle iter + 5% cardinalité

10% dépendance vers Utilisateurs

35% classe UE

5% classe

10% par opération (5% paramètre)

30% package Utilisateurs

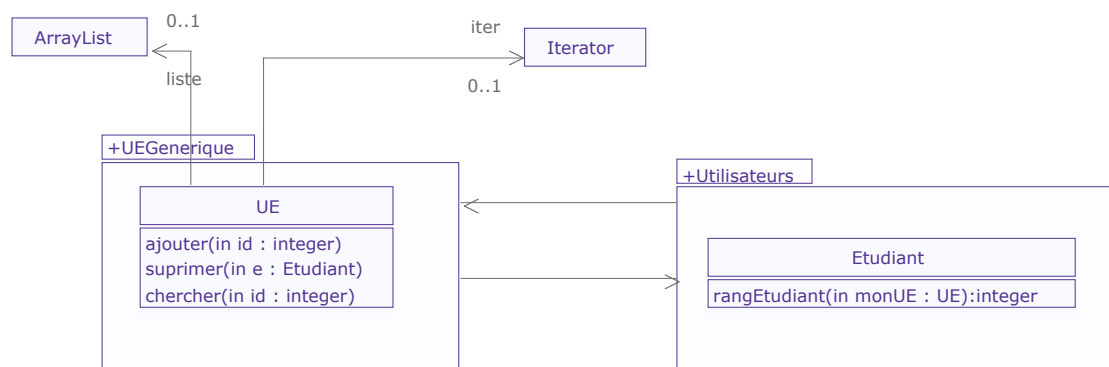
5% package

15% classe étudiant

5% classe

10% pour l'opération (5% paramètre)

10% dépendance vers UEGenerique



1.2. Expliquez pourquoi le package UEGenerique n'est pas réutilisable sans le package

Utilisateurs. Justifiez votre réponse. (1 point)

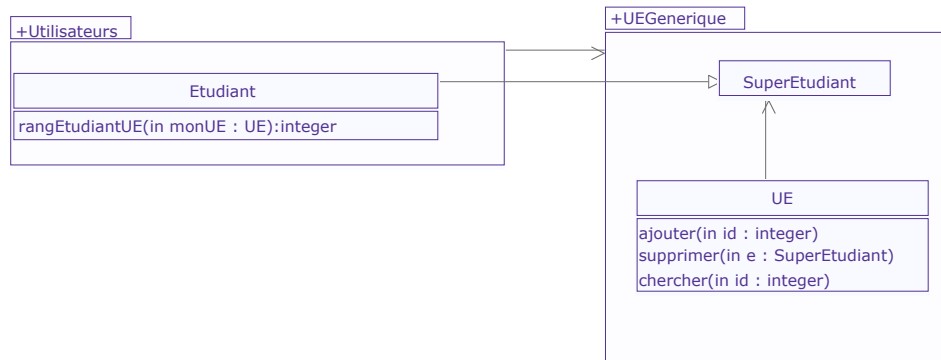
Correction :

50% UEGénérique dépend d'Utilisateurs

50% justification : UEGenerique dépend d'Utilisateurs à cause du paramètre e dans supprimer()

1.3. Proposez une solution pour que le package UEGenerique soit réutilisable sans le package Utilisateurs. (2 points)

Correction :

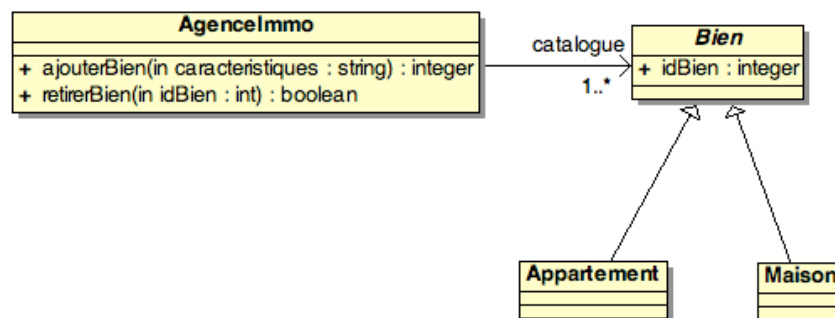


100% si tout

0% sinon

Exercice 2 (10 points)

Nous considérons dans cet exercice une application *AgenceImmobiliere* qui permet à un agent immobilier de gérer un catalogue de biens immobiliers (appartements, maisons). Un agent peut ajouter et enlever des biens au catalogue qu'il gère. Le diagramme de classes UML de l'application *AgenceImmobiliere* est fourni en figure 3.

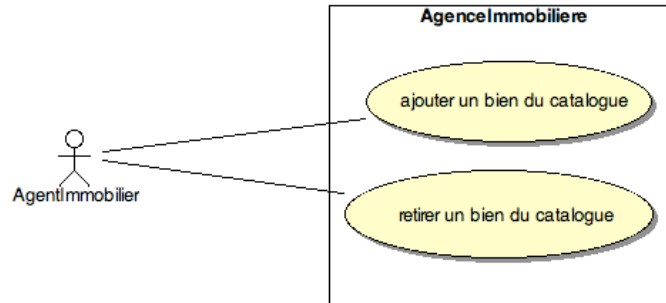
**Figure 3 - Diagramme de classes de l'application *AgenceImmobiliere***

La classe abstraite *Bien* représente un bien enregistré dans le catalogue, ce bien étant soit une instance de la classe *Appartement*, soit une instance de la classe *Maison*. Un bien se voit attribuer un identifiant entier unique lors de sa création (propriété *idBien* de la classe *Bien*).

La classe *AgenceImmo* est responsable des opérations permettant d'ajouter et d'enlever des biens au catalogue auquel elle est liée. L'opération *ajouterBien* prend en paramètre d'entrée une unique chaîne de caractères contenant l'ensemble des caractéristiques du bien à ajouter au catalogue – la première étant le type de bien : appartement ou maison, puis adresse, puis des informations diverses - les informations étant séparées par des points-virgules ; elle retourne l'identifiant du bien créé dans le catalogue. L'opération *retirerBien* prend en paramètre l'identifiant d'un bien et retourne un booléen dont la valeur est TRUE si le bien a été correctement retiré du catalogue, FALSE sinon.

2.1 Donnez le diagramme de cas d'utilisation de l'application *AgenceImmobiliere*. (1 point)

Correction :



40% par cas d'utilisation

20% acteur

-25% par erreur : use case / acteur qui n'a pas lieu d'être

2.2 Donnez un diagramme de séquence montrant l'ajout d'un appartement dans le catalogue. Vous choisirez les caractéristiques de l'appartement. (2 points)


Correction :

20% acteur agent immobilier : -10% si instance non typée

20% pour une instance de la classe AgenceImmo

60% ajouterBien

20% appel correct ajouterBien : 10% appel + 10% chaîne de caractères bien formée, i.e. avec Appartement en 1ere position.

20% création d'une instance de la classe Appartement

20% retour de l'opération = identifiant entier

2.3 Donnez le code de la classe *AgenceImmo* obtenue en effectuant l'opération de génération de code UML vers Java vue en cours. (2 points)

Correction :

```

import java.util.ArrayList ;
public class AgenceImmo {
    ArrayList catalogue ;
    int ajouterBien(String caracteristiques) {}
    boolean retirerBien(int idBien) {}
}
    
```

40% catalogue : 20% ArrayList + 20% import

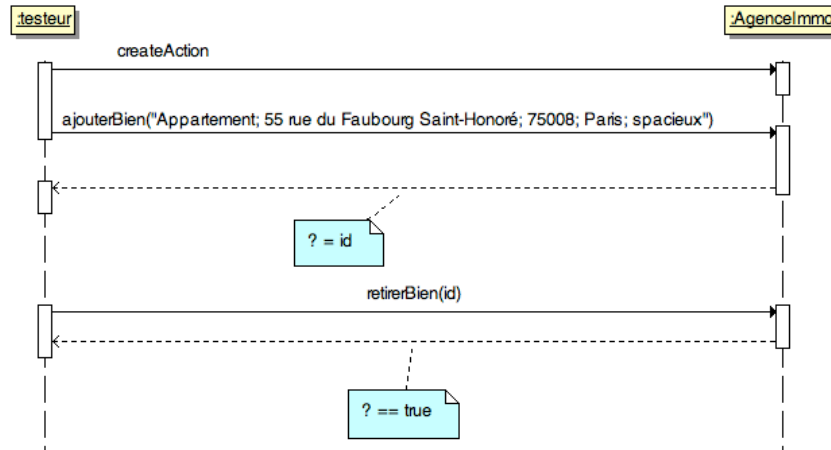
30% ajouterBien : 10% ajouterBien existe, 10% param, 10% résultat

30% retirerBien : 10% retirerBien existe, 10% param, 10% résultat

2.4 Donnez un diagramme de séquence de test permettant de tester la suppression d'un bien

présent dans le catalogue. (2 points)

Solution : il faut créer ce qu'on va supprimer ...



Correction :

20% testeur

20% création agenceImmo

30% ajouterBien

20% appel dont 10% paramètre

10% stockage id

30% retirerBien

20% appel dont 10% paramètre

10% vérification == true

On souhaite pouvoir effectuer facilement de nouvelles opérations sur l'ensemble des biens du catalogue d'une agence. Considérons par exemple l'opération d'évaluation de la valeur marchande d'un bien. Chaque bien (*Appartement*, *Maison*) est responsable d'une opération permettant cette évaluation. Ainsi un objet de type *Appartement* possède une opération *evalAppartement()*, tandis qu'un objet de type *Maison* possède une opération *evalMaison()*. La figure 4 présente le nouveau diagramme de classe de l'application *AgenceImmobilie*.

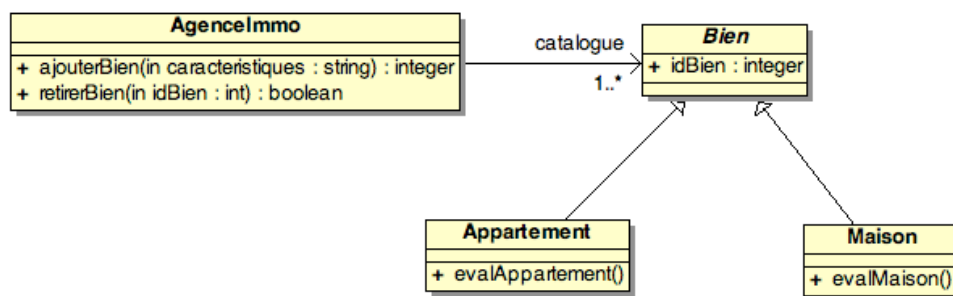
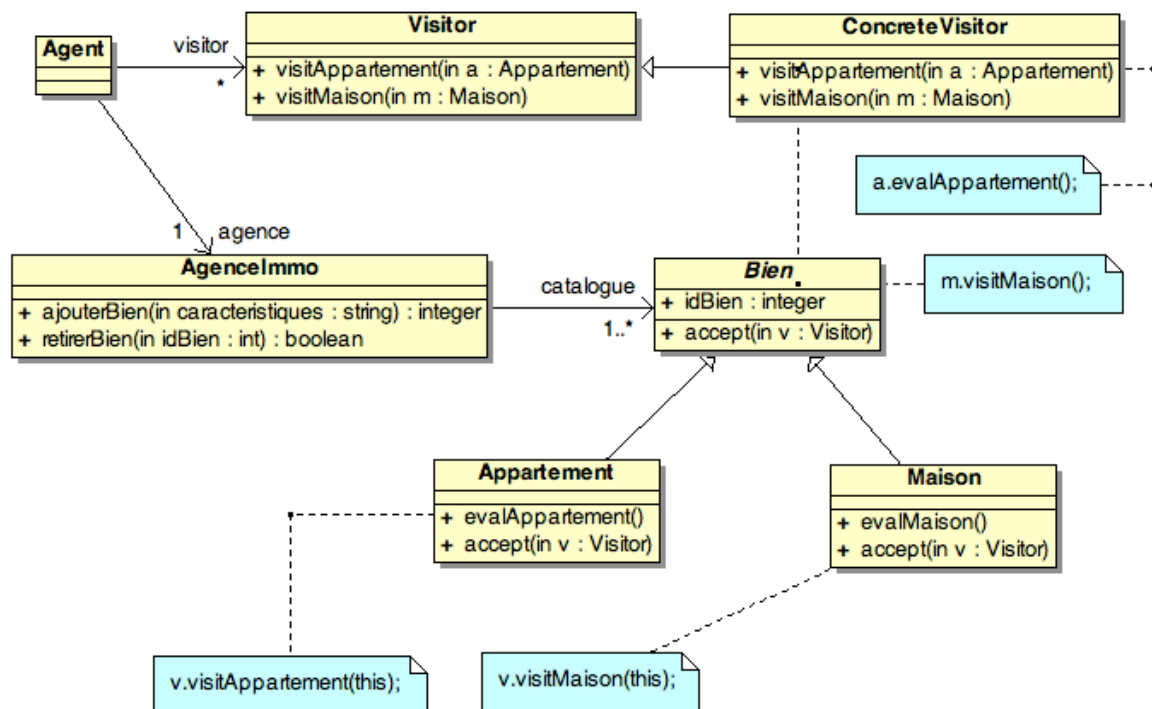


Figure 4 – Nouveau diagramme de classes de l'application *AgenceImmobilie*

2.5 Présentez dans un diagramme de classes UML une solution utilisant le design pattern Visitor (fourni en annexe page 4) permettant d'évaluer l'ensemble des biens du catalogue. Vous explicitez les ajouts dans le code des opérations utilisées par le pattern (3 points)

Solution :



L'agent n'a même pas à apparaître, il est juste là pour rappeler qu'il tient le rôle du client dans le pattern.

Correction :

20% Classe Visitor :

10% `visitAppartement(in a : Appartement)` dont 5% paramètre

10% `visitMaison(in m : Maison)` dont 5% paramètre

40% Classe ConcreteVisitor :

10% héritage

15% sur `visitAppartement()` : 5% délégation + 10% note de code

15% sur `visitMaison()` : 5% délégation + 10% note de code

10% Classe Bien : opération `accept(in v : Visitor)` : 5% paramètre

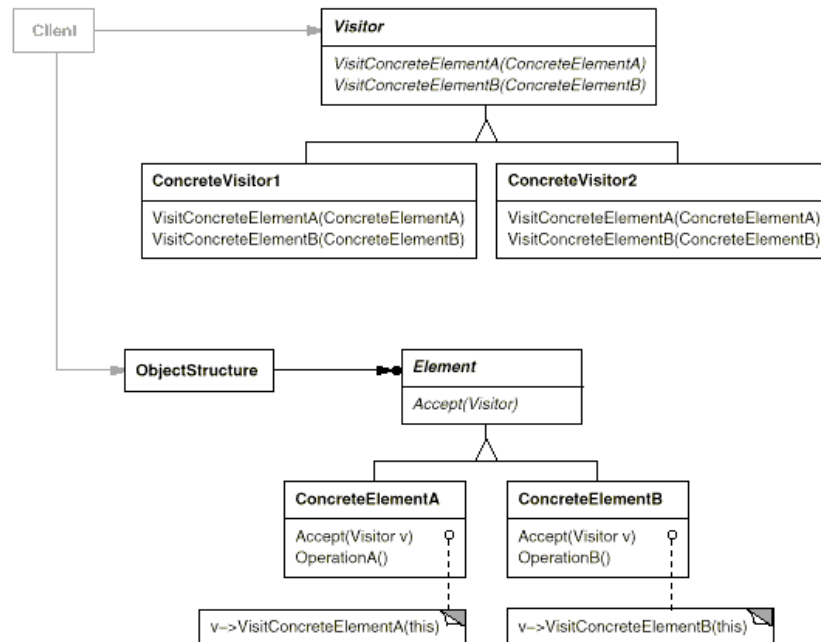
15% Classe Appartement : sur `accept()` : 5% délégation + 10% note de code

15% Classe Maison : sur `accept()` : 5% délégation + 10% note de code

Annexe : Pattern Visitor

Intention

Représenter une opération qui doit être effectuée sur les éléments d'une structure d'objets. Le Visiteur permet de définir une nouvelle opération sans changer les classes des éléments sur lesquelles elles agissent.

Structure**Participants**

Visitor déclare une opération *VisitConcreteElement* pour chaque classe d'élément concret dans la structure d'objet (**Object Structure**). Le nom de l'opération et sa signature identifient la classe qui envoie la requête *Visit* au visiteur. Ceci permet au visiteur de déterminer la classe concrète de l'élément visité. Le visiteur peut alors accéder à l'élément directement au travers de son interface.

ConcreteVisitor implémente chaque opération déclarée par **Visitor**. Chaque opération implante un fragment de l'algorithme défini pour la classe correspondante dans la structure d'objet.

Element définit une opération *Accept* qui prend un visiteur (**Visitor**) en argument.

ConcreteElement implémente une opération *Accept* qui prend un visiteur (**Visitor**) en argument

ObjectStructure peut énumérer ses éléments

Collaborations

Un client qui utilise le pattern Visitor doit créer un objet **ConcreteVisitor** et parcourir la structure d'objet, en visitant chaque élément de la structure avec le visiteur. Quand un élément est visité, il appelle l'opération du visiteur (Classe **Visitor**) qui correspond à sa classe, en se passant lui-même en argument de l'opération (*this*) pour que le visiteur puisse accéder à son état si besoin est.

Le diagramme de séquence suivant illustre les collaborations entre une structure d'objet (**ObjectStructure**), un visiteur (**Visitor**) et deux éléments (**ConcreteElement**).

