Vendredi 16 Mars 2018 UPMC SORBONNE UNIVERSITÉS 3I012 – Partiel –Durée 1h45

Aucun document autorisé sauf les deux feuilles précisant les règles de génération de code et reverse engineering. Toute réponse doit impérativement être justifiée !!! Barème donné à titre indicatif et susceptible d'être modifié.

Questions de cours (3 points)

C1. Si le corps d'une opération UML d'une classe A contient un appel à une opération d'une classe B, peut-on dire qu'il y a une dépendance UML de A vers B ?

Correction :
Non, ca ne fait pas partie des 4 causes de dépendances UML possibles vues en cours.
Notation :
100% si tout
0% sinon

C2. Quelle différence y-a-t-il entre une agrégation et une composition en UML en ce qui concerne les objets qui les constituent ?

Correction :

Agrégation : les objets agrégés peuvent appartenir à d'autres agrégations. Si je détruis l'agrégation, les objets agrégés ne sont pas détruits (cycle de vie non liés)
Composition : les objets composants ne peuvent appartenir qu'à une composition. Si je détruis la composition, les objets composants sont détruits (cycle de vie liés)

Notation
100% si tout
50% explication correcte agrégation – uniquement 25% si l'idée est là mais que ce n'est pas complet
50% explication correcte composition – uniquement 25% si l'idée est là mais que ce n'est pas complet
C3. Quel patron de conception faut-il utiliser pour casser les cycles de dépendances entre packages ?

Correction :

Aucun ! Pas le problème adressé par un patron en particulier, effet de bord.

Notation :
Binaire 100% si on dit bien qu'on n'utilise aucun patron, 0% sinon

Exercice1 (4 points)

La Figure 1 montre une partie d'un diagramme de classes UML pour une application simple de gestion de commandes par un gérant d'une entreprise de distribution. La Figure 2 montre un exemple d'un diagramme de séquence de la **même** application.

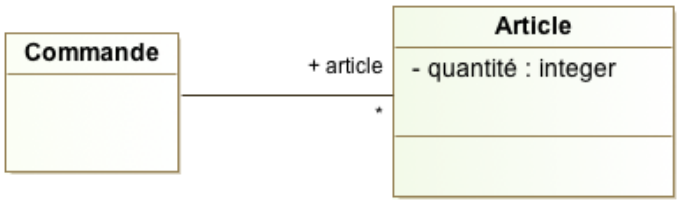


Figure 1. Diagramme de classes d'une application de gestion de commandes.

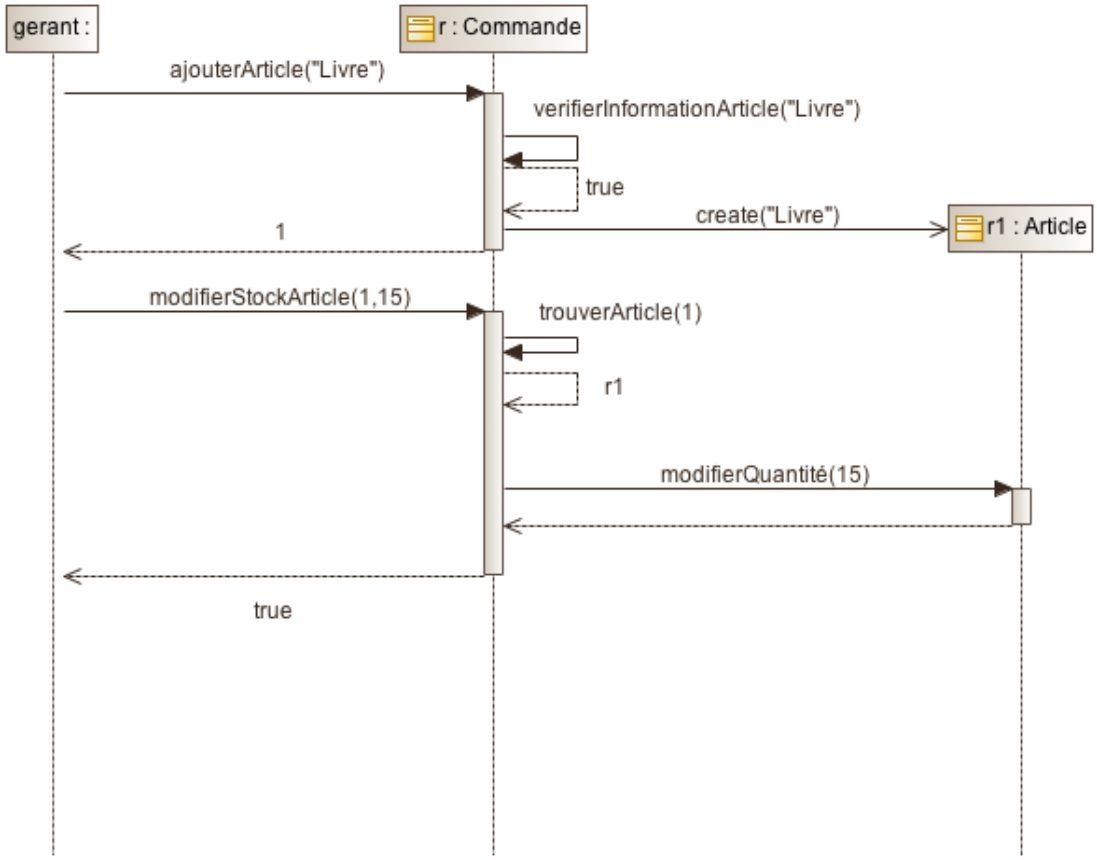
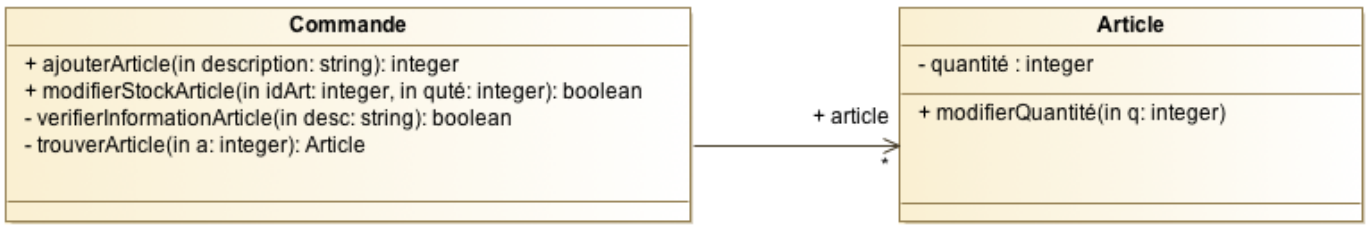


Figure 2. Un diagramme de séquence de la même application de gestion de commandes.

Question 1 : Comment est modélisé le gérant de l'entreprise dans le diagramme de séquence de la Figure 2 ?

Correction :
Le gérant est modélisé par un objet non typé puisque c'est une entité externe à l'application donc ne peut pas le typer par une classe de l'application.
Notation:
50% objet non typé
50% justification correcte

Question 2 : Donnez le nouveau diagramme de classes de l'application en apportant au diagramme de classes de la figure 1 les modifications nécessaires pour qu'il soit **cohérent** avec le diagramme de séquence de la Figure 2.
Correction :



Notation :
20% par opération

- **5% l'opération existe et est placée dans la bonne classe**
- **10% TOUS les paramètres de l'opération sont présents et bien typés 0% sinon**
- **5% la visibilité (public ou private) est correcte**

-10% global si l'association n'a pas été rendue navigable de Commande vers Article ou si elle est navigable de Article vers Commande.

Exercice 2 (6 points)

Considérons l'application *eRobots* dont le diagramme de classes est présenté dans la Figure 3. Nous souhaitons appliquer le design pattern *Observer* (voir annexe) pour la gestion de deux robots, appelés *Robot1* et *Robot2* dans le diagramme de classe de la Figure 3. Les deux robots sont liés à une caméra de surveillance (la classe **CameraSurveillance**). A chaque fois qu'une image suspecte est détectée (l'opération *imageSuspecteDetectee()*), les deux robots doivent être notifiés, récupérer les informations détaillées de l'image détectée (l'opération *getInfoDetaillées()*) et réagir de deux manières différentes : 1) Robot1 doit émettre un son (l'opération *emettreSon()*) 2) Robot2 doit appeler le propriétaire sur son téléphone (l'opération *appelerProprietaire()*).

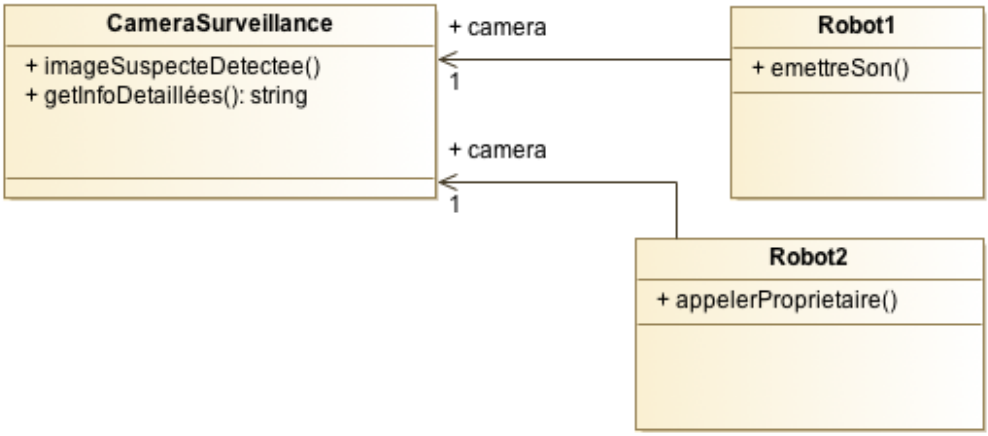
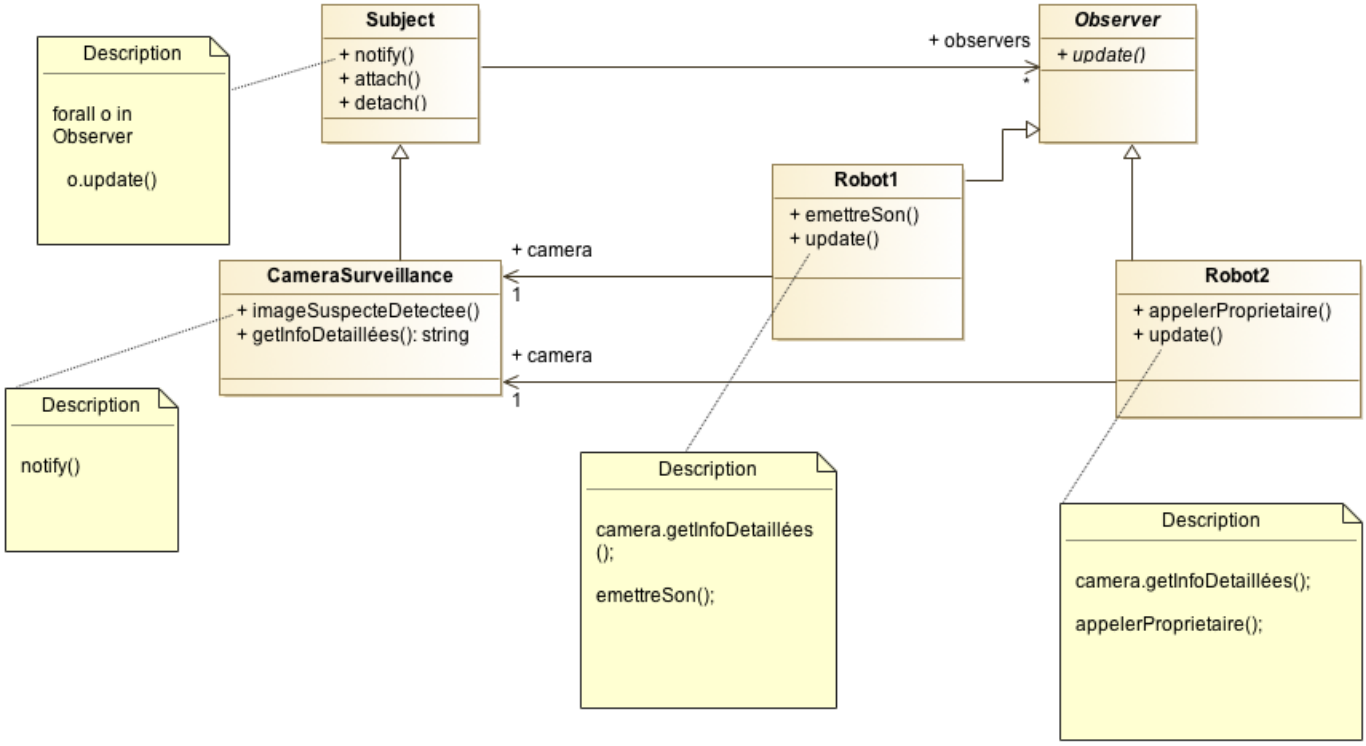


Figure 3. Diagramme de classes de l'application *eRobots*.

Question 1 : Appliquez le design pattern *Observer* présenté en annexe pour gérer la notification des robots et leurs réactions.

CORRECTION



NOTATION :

15% Observer
- **5% classe**
- **10% update()**

40% Subject
- **5% classe**
- **10% association navigable vers Observer**
- **5% notify()**
- **10% note code notify()**
- **5% attach()**
- **5% detach()**

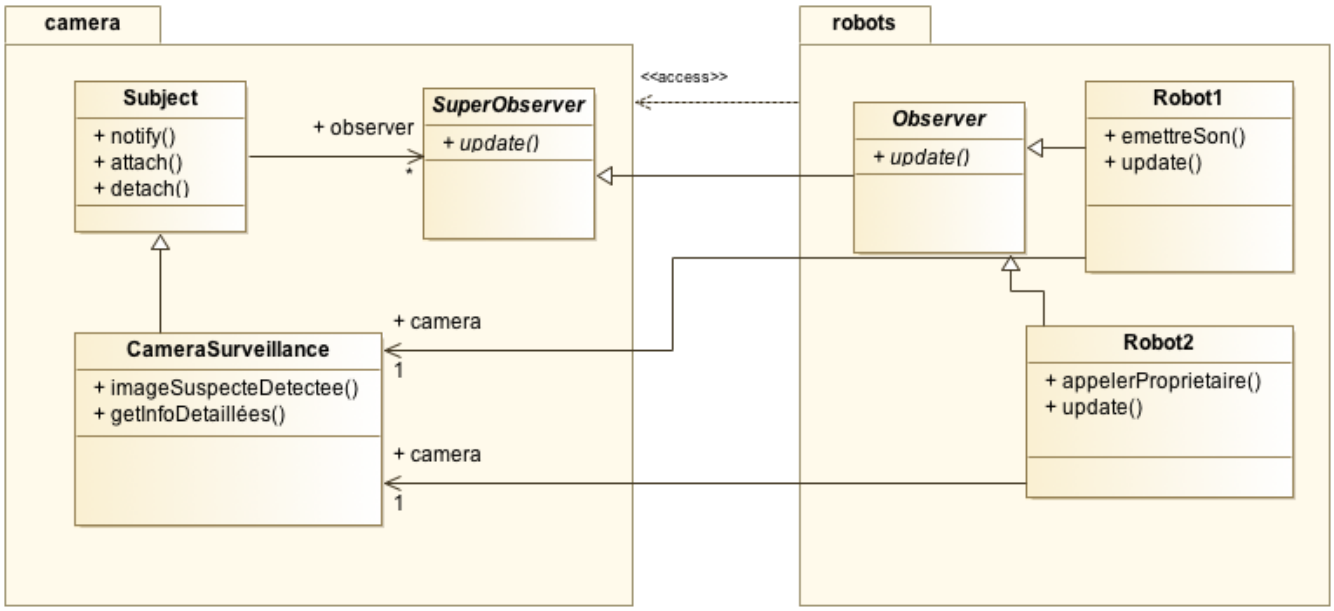
5% CaméraSurveillance hérite de Subject
10% note code imageSuspecteDetectee()
15% Robot1
- **5% héritage**
- **10% note code update()**

15% Robot2
- **5% héritage**
- **10% note code update()**

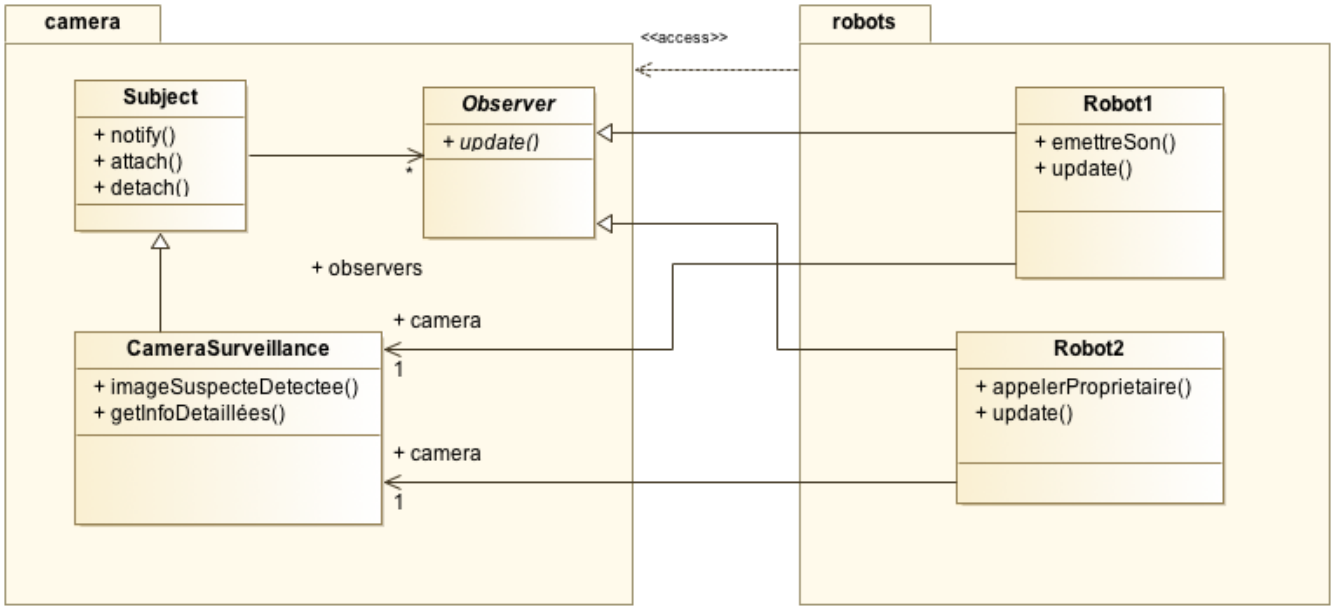
Question 2 : En se basant sur votre nouveau diagramme de classes obtenu après application du design pattern Observer, on souhaite mettre **toutes les classes** dans deux packages différents *camera* et *robots*. Proposez un découpage qui respecte les règles vues en cours et qui respecte les contraintes suivantes :

1. Les deux classes *Robot1* et *Robot2* doivent être dans le package *robots*.
2. La classe *CameraSurveillance* doit être dans le package *camera*.
3. Vous placerez chaque classe restante dans le package *camera* ou *robots* de manière à ce que le package *camera* soit réutilisable sans le package *robots*.

CORRECTION
2 solutions :



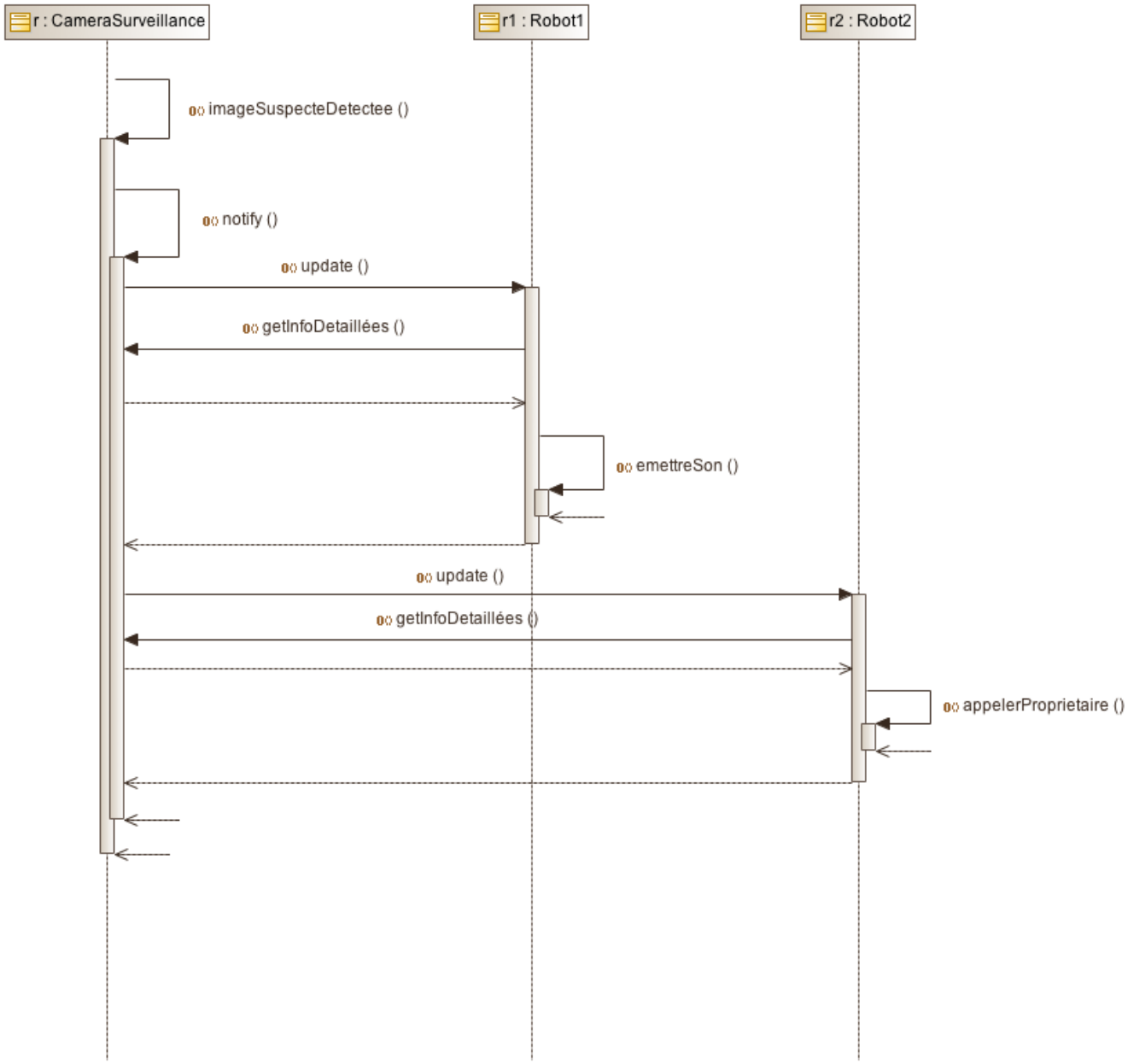
Ou



NOTATION :
Binaire 100% ou 0% pour l'une des 2 solutions possibles

Question 3 : Montrez à l'aide d'un diagramme de séquence l'utilisation de ce design pattern depuis l'appel de l'opération *imageSuspecteDetectee()* (cette opération est invoquée par la caméra de surveillance sur elle-même) jusqu'au traitement effectué par les robots.

CORRECTION



Notation :
5% opération imageSuspecteDetectee()
5% opération notify() dans imageSuspecteDetectee() 0% sinon
45% par update() (pas d'ordre pour les robots) dans notify()

- **5% opération update()**
- **20% getInfoDétaillées()**
- **20%emettreSon() sur r1:Robot1 ou appelerPropriétaire() sur r2:Robot2**

Exercice 3 (7 points)

Nous considérons une application web eAdP6, un projet d'étudiants de Paris 6 qui vise à permettre la gestion des informations relatives aux aéroports de Paris (AdP : CDG ou Orly). Plus précisément, on souhaite offrir des fonctionnalités à trois types d'utilisateurs : un personnel des AdP, un internaute quelconque, un passager enregistré sur un vol. Tous peuvent consulter la liste des vols au départ d'un aéroport des AdP. Un personnel des AdP est le seul à pouvoir enregistrer un nouveau vol pour qu'il apparaisse sur eAdP6 et changer le statut de n'importe quel vol pour lui donner une valeur parmi 3 statuts possibles : « à l'heure », « embarquement », « a décollé ». L'affichage de ce statut peut être demandé par un internaute ou un passager enregistré. Un internaute peut s'enregistrer pour un vol précis, auquel cas il devient pour eAdP6 passager de ce vol. Seul un passager enregistré peut visualiser la liste des places

disponibles pour le vol, et changer de place en fonction des places encore disponibles dans l'avion. Ce qui signifie que pour changer de place il faut toujours auparavant afficher la liste des places disponibles pour le vol. Le diagramme de classes de l'application est présenté en figure 1.

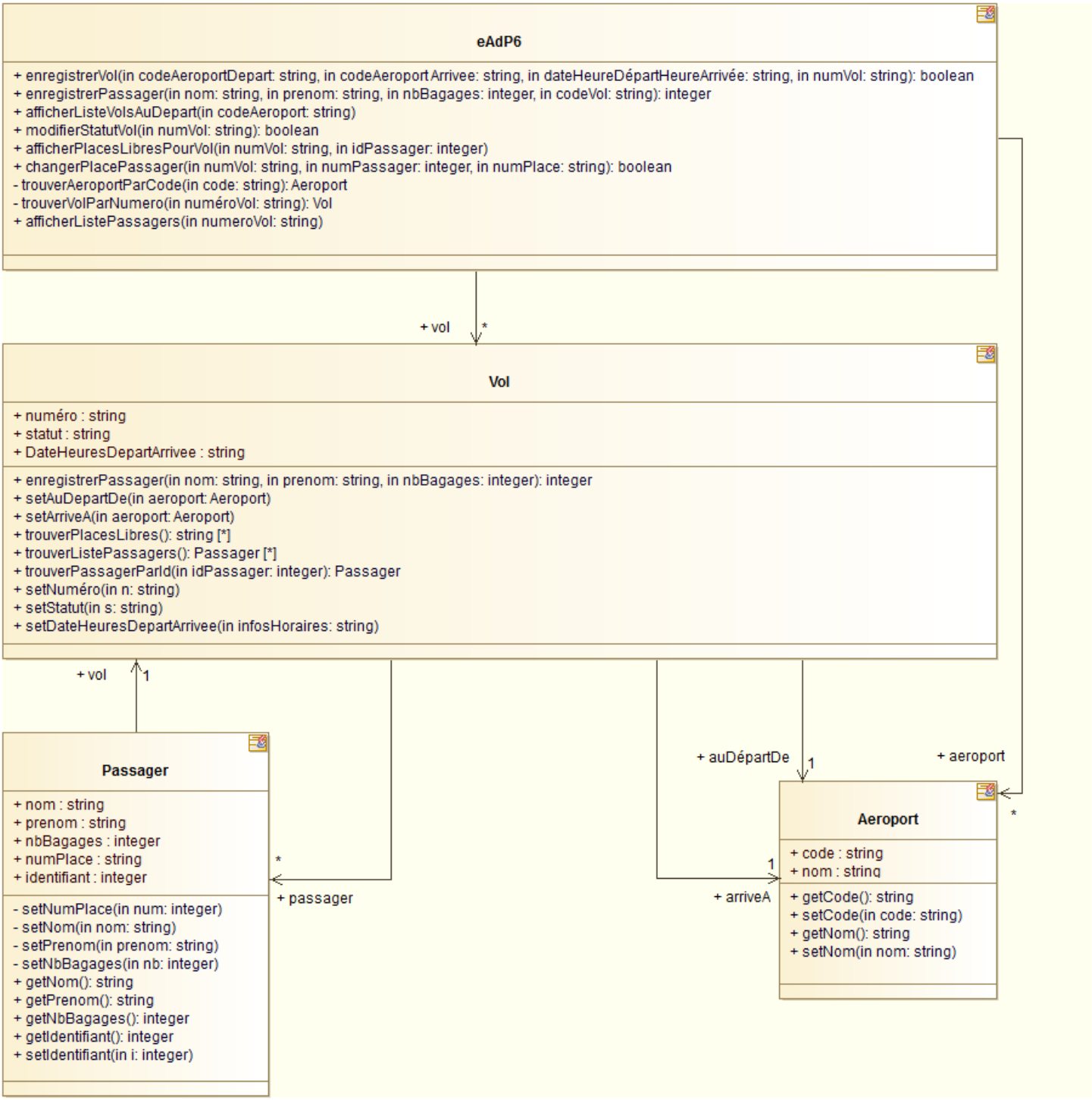
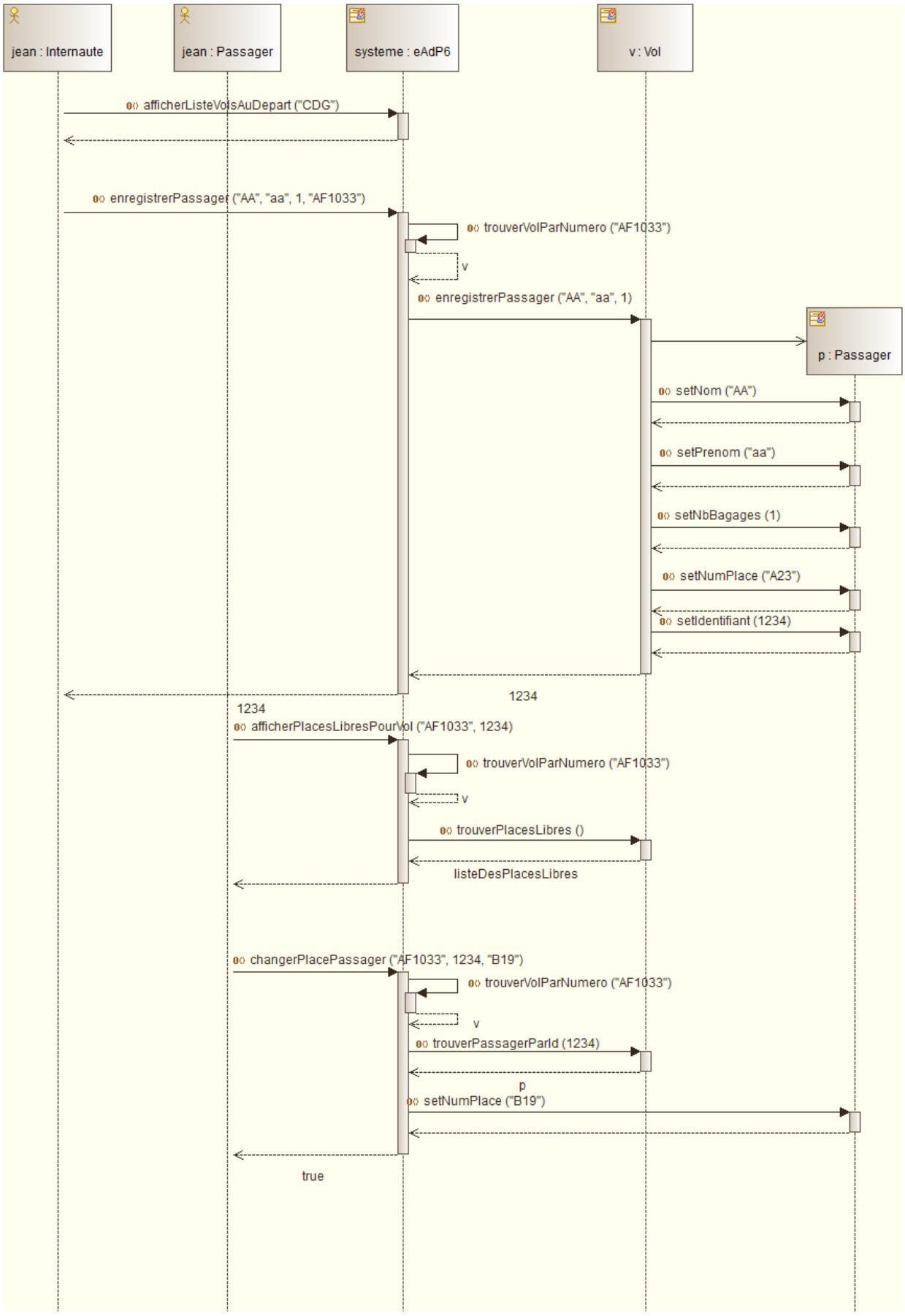


Figure 1. Diagramme de classes conceptuel de eAdP6

1.1. Donnez le diagramme de séquence présentant le scénario suivant :
Un internaute affiche la liste des vols au départ de CDG. Il choisit un vol pour Athènes et il s'enregistre pour ce vol. Une fois enregistré, il change son numéro de place sur le vol.

Correction :



Notation :
100% si tout
10% afficherListeVolsAuDepart()
40% phase enregistrerPassager()

- **5% opération enregistrerPassager()**

- **5% trouverVolParNuméro()**
- **30% enregistrerPassager()**
    - **10% opération enregistrerPassager()**
    - **10% création objet Passager**
    - **10% setters appelés sur l'objet Passager par l'objet Vol ou en interne par l'objet passager sur lui-même, on accepte aussi si tout est passé dans le create au moment de la création de l'objet Passager : create(« AA », « aa »,1, »A23 »,1234)**

20%phase afficherPlacesLibresPourVol()

- **5%opération afficherPlacesLibresPourVol()**
- **5%trouverVolParNuméro()**
- **10% trouverPlacesLibres()**

30% phase changerPlacesPassager()

- **5% opération changerPlacesPassager()**
- **5% trouverVolParNumero()**
- **10% trouverPassagerParId()**
- **10% setNumPlace()appelé sur l'objet Passager par l'objet eADP6 ou en interne par l'objet passager sur lui-même**

0% sinon

1.2. Donnez le code Java obtenu lorsqu'on applique l'opération de génération de code vue en cours à la classe **Vol**.

Correction :

```
import java.util.ArrayList;

public class Vol {
   public ArrayList passager = new ArrayList<Passager>();
   public Aeroport auDépartDe;
   public Aeroport arriveA;
   public String numéro;
   public String statut;
   public String dateHeuresDepartArrivee;

   public int enregistrerPassager(String nom, String prenom, int nbBagages) {}
   public void setAuDepartDe(Aeroport aeroport) {}
   public void setArriveA(Aeroport aeroport) {}
   public String[] trouverPlacesLibres() {}
   public Passager[] trouverListePassagers() {}
   public Passager trouverPassagerParId(Passager idPassager) {}
   public void setNuméro(String n){}
   public void setStatut(String s){}
   public void setDateHeuresDepartArrivee(String d){}
}
```

Notation :
100% si tout
15% gestion ArrayList : attribut passager (avec ou sans import on donne les 15%)
5% par autre attribut
60% pour les 9 opérations => -10% par erreur jusqu'à 0% sur les 60%
0% sinon

1.3. On souhaite ajouter l'opération **informationsPassager()** qui permet d'afficher l'ensemble des informations sur un passager (voyageur enregistré pour un vol). Pour **chacune** des signatures suivantes, indiquer dans quelle(s) classe(s) il faudrait placer l'opération **informationsPassager().**

1) informationsPassager(in idPassager : Integer)
2) informationsPassager(in idVol : String, in idPassager : Integer)
3) informationsPassager()
4) informationsPassager(in appli : eAdP6)

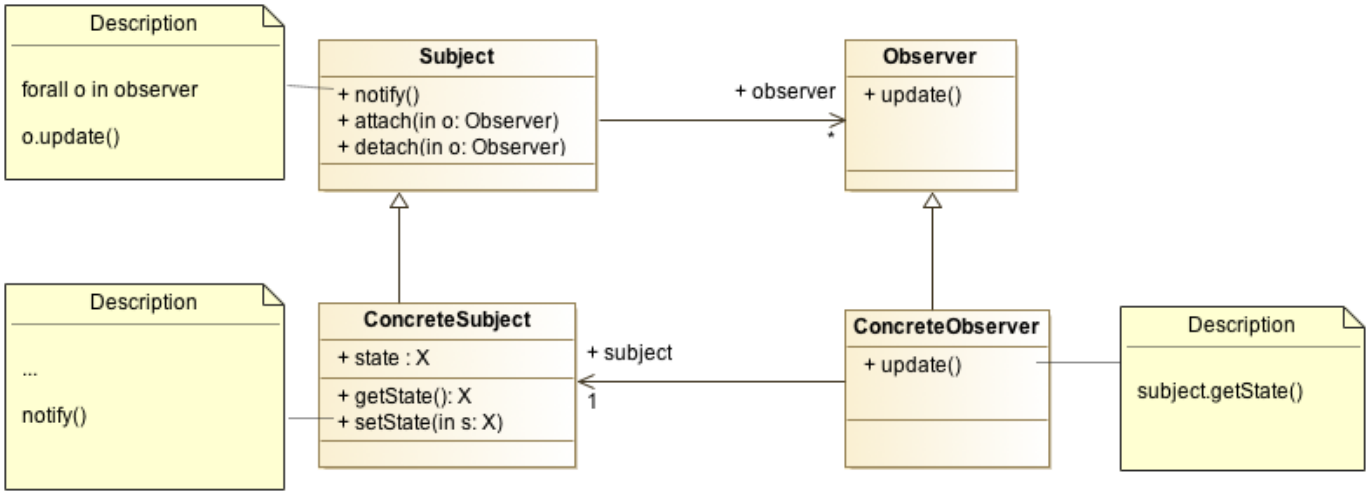
Correction :

1) Vol
2) eAdP6
3) Passager
4) aucune, car paramètre inutile. **Aussi accepté**: uniquement Passager MAIS paramètre inutile

Notation :
100% si tout
25% par signature
0% sinon

ANNEXE : Pattern Observer

Intention : le sujet possède un état interne. Les observateurs doivent se synchroniser avec les changements de l'état du sujet.



Rôles :

Subject : abstraction du sujet, définit les opérations permettant au sujet de notifier les observateurs et à des objets de s'ajouter ou de s'enlever à l'ensemble des observateurs.

ConcreteSubject : le sujet concret, possède l'état interne.

Observer : l'abstraction de l'observateur, définit la méthode de mise à jour permettant de récupérer la valeur de l'état du sujet.

ConcreteObserver : l'observateur concret, doit se synchroniser avec les changements de l'état du sujet.