Vendredi 18 Mars 2016 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.

Question de cours (3 points)

Q1. Dans un diagramme de classes, dans quels cas une classe A dépend d'une classe B ?
Correction :

- **Héritage : A hérite de B**
- **Association navigable : de A vers B**
- **Attribut typé : un attribut dans de type B**
- **Paramètre de l'opération : un paramètre d'une opération dans A est typé par B**

Notation :
25% par cas

Q2. Quelles sont les règles de cohérence entre un diagramme de séquence et un diagramme de classes de la même application ?
Correction :

1. **Les objets typés doivent être typés par des classes présentes dans le diagramme de classes. Une autre réponse correcte : toutes les classes utilisées comme types dans le diagrammes de séquence sont des classes qui existent dans le diagramme de classes.**
2. **Chaque message dans le diagramme de séquence doit correspondre à une invocation d'une opération. Cette opération doit être définie dans la class (présente dans le diagramme de classes) représentant le type de l'objet**

Notation :
50% par point.

1. **Pour le 1 ere point :**
   Il faut absolument parler d'un objet typé. Sinon seulement 25% (par exemple la réponse du genre : pour chaque objet dans le diagramme de séquence doit être typé par une classe dans le diagramme de classes => 25%-
2. **2 eme point**
   il faut indiquer clairement que l'opération doit être dans la classe représentant le type de l'objet qui reçoit le message. Sinon il faut seulement donner 20% sur les 50%.
   Par exemple la réponse du genre : chaque message doit correspondre à une opération ➔ 20%

Exercice (17 points)

Considérons le système **iDomotique** qui permet de commander les équipements domotiques dans un environnement résidentiel et selon les besoins de l'habitant. Le système comprend différents types d'équipements et de services assurés par ces équipements. En particulier, nous distinguons la Box qui gère un ensemble de capteurs et d'actionneurs. Les capteurs permettent de collecter les données de l'environnement dans l'habitat. Les deux capteurs les plus utilisés sont les capteurs de présence (pour détecter la présence d'intrus dans l'habitat) et le thermostat (pour détecter la température). Les actionneurs (appelés aussi contrôleurs), permettent de commander les équipements qui sont installés dans l'habitat. Les deux actionneurs considérés par **iDomotique** sont l'actionneur d'alarme (qui déclenche l'alarme) et l'actionneur de chauffage (qui contrôle le système de chauffage central).

Un étudiant de l'UPMC de L3 qui a suivi le cours 3I012 a réalisé un stage d'été dans une entreprise innovante travaillant sur **iDomotique.** Dans la phase de modélisation de ce système, il a proposé d'utiliser UML comme langage de modélisation et il a réalisé le diagramme de classes de la figure 1.

La classe *Box* modélise la box qui gère les différents *Service*, *Capteur* et *Actionneur*. Le système offre pour le moment deux types de services : *Surveillance* et *Chauffage*. Comme le montre le diagramme de classes de la figure 1, à travers l'association qui part de la classe *Service* vers *Actionneur*, chaque service est relié à un seul actionneur. Un service concret est lié aussi à un seul *Capteur*. Le service *Surveillance* est lié au capteur *CapteurPresence* quant au service *Chauffage*, il nécessite le capteur *Thermostat*.

Pour expliquer le fonctionnement du système **iDomotique,** l'étudiant a rédigé un document décrivant en détail les fonctionnalités de ce système. Nous présentons dans ce qui suit un résumé de la description de des étapes concernant l'initialisation et l'ajout de services :

Initialisation et Activation de la Box. Il s'agit de la première étape qui se déroule selon le scénario suivant :

- l'utilisateur se connecte à la box (l'opération *seConnecter()* dans *Box*)
- L'utilisateur initialise la box (en invoquant l'opération *initialiser()* de *Box*). Cette initialisation est réalisée par la box en créant les quatre objets instances des quatre types de capteurs et d'actionneurs présents (*DeclencheurAlarme*, *ControleurChauffage*, *Thormostat*, *CapteurDePresence*).
- L'utilisateur active la box (l'opération *activer()* de la *Box*). Cette activation consiste à mettre à jour la valeur de la propriété *etatBox* (elle passe à *True* à l'aide de l'opération *setEtat()* de *Box*)

Ajout et activation de services. Une fois le système initialisé, l'utilisateur peut ajouter les différents services selon ses souhaits. Le scénario se déroule comme suit :

- L'utilisateur demande à la box d'ajouter un service (en invoquant l'opération *creerService()* de *Box* en passant comme argument le nom de service souhaité. e.g. ''Surveillance'' pour le service Surveillance). Le scénario se déroule comme suit :
  - La *Box* Invoque une opération de *FabriqueService* qui correspond à la création du service demandé. Par exemple, pour le service Surveillance, l'opération à invoquer est *creerServiceSurveillance*() de *FabriqueService*. Comme le montre le diagramme de classes de la figure 1, les opérations de création de service dans *FabriqueService* prennent deux arguments de types *Capteur* et *Actionneur*. Il s'agit des capteur et actionneur nécessaires pour le service demandé. Pour créer le service Surveillance par exemple il faut invoquer l'opération *creerServiceSurveillance*() avec deux arguments de type respectivement *CapteurDePresence* et *DeclencheurAlarme*.
  - Pour chaque invocation d'une opération de *FabriqueService*, le processus d'ajout et d'activation de service se termine selon le scénario suivant :
    - La fabrique de service crée un objet instance de la classe correspondant au service choisi (créer une instance de la classe *Surveillance* par exemple pour l'opération *creerServiceSurveillance()*). Elle ajoute à cet objet les capteurs et les actionneurs nécessaires (les opérations *ajouterCapteur()* et *ajouterActionneur()* héritées de *Service*)
    - La fabrique de service active le service (l'opération *activerService()* héritée de *Service*). Cette invocation de l'opération *activerService()* engendre : 1) la modification de l'état de service (la valeur de *etatService* qui passe à *True*). 2) l'activation des capteurs et actionneurs nécessaires (les opérations *initialiserCapteur()* et *initialiserActionneur()* héritées de *Capteur* et *Actionneur*)
  - L'objet instance de la box ajoute le service créé et retourné par la fabrique en invoquant l'opération *ajouterService()* de *Box*.

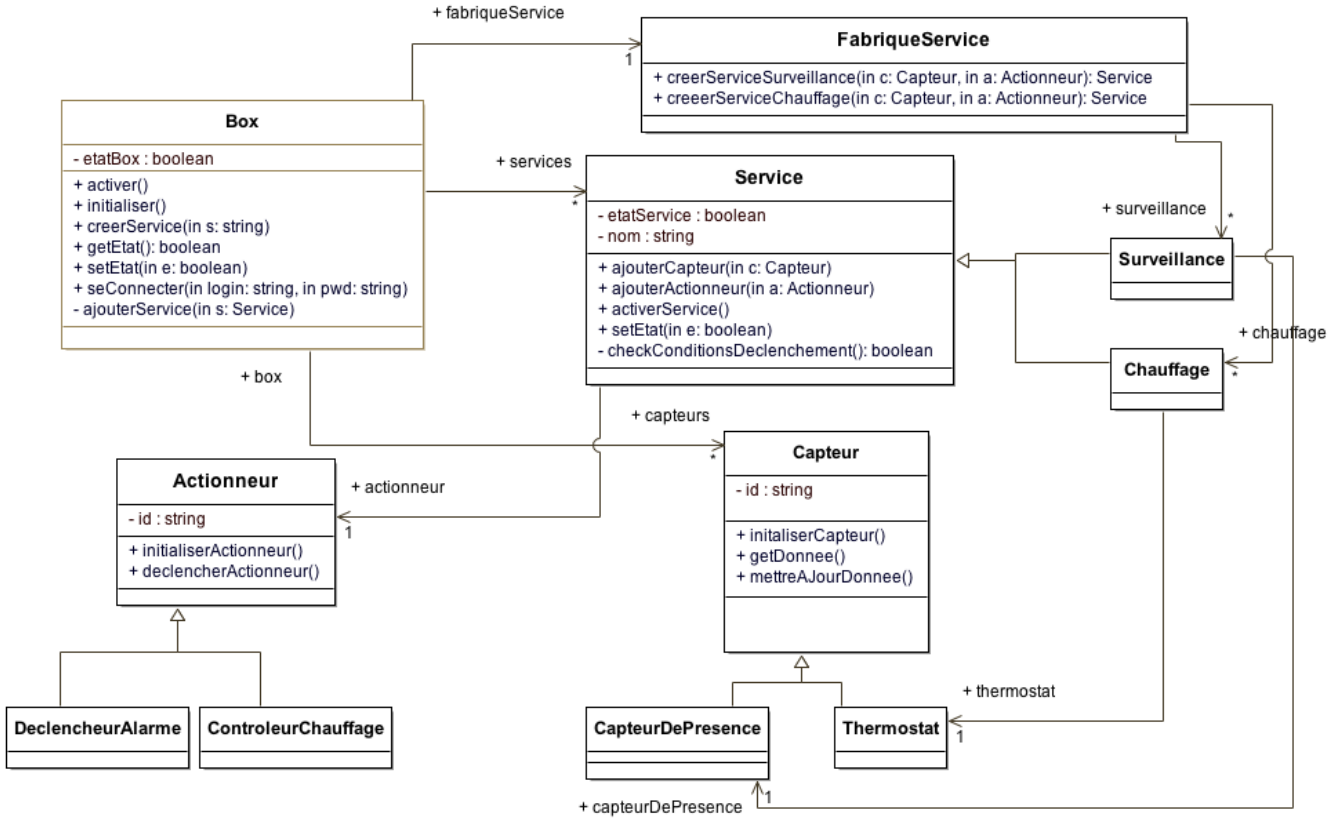
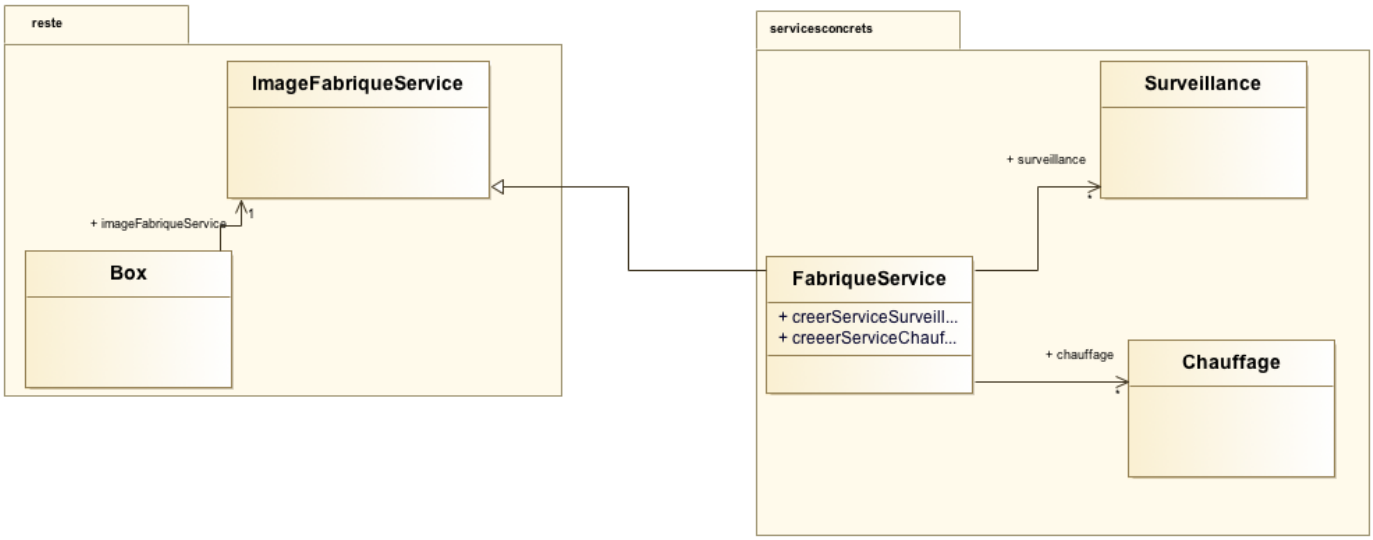


Figure 1: Diagramme de classes de iDomotique

E1 (2 points). Nous souhaitons organiser les classes de **iDomotique** en créant deux packages : *servicesconcrets* et *reste*.
La package *servicesconcrets* contient les classes liées à la gestion des services concrets : *FabriqueService*, *Surveillance* et *Chauffage.*
Le package *reste* contient le reste de classes.

- Identifiez et justifiez le problème de conception engendré par ce découpage.
- Proposez une solution pour ce résoudre ce problème de manière à rendre le package *reste* soit **réutilisable** sans le package *servicesconcrets (***Votre solution doit assurer donc que le package** *reste* **ne dépendra pas de** *servicesconcrets*).

Correction :
`

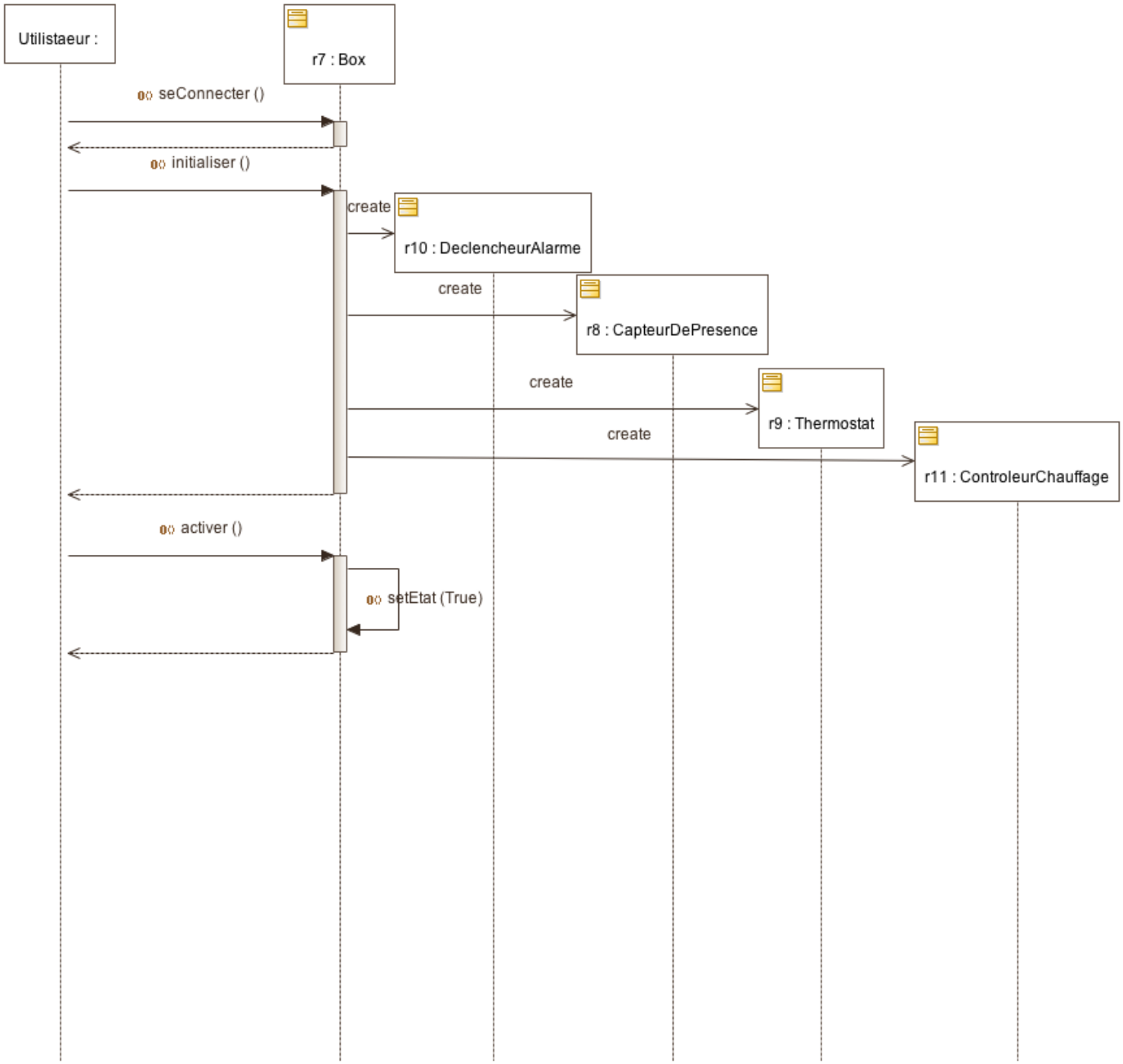


Une super classe ImageFabrique (ou SuperFabrique) dans package reste avec une association qui part de Box vers ImageFabrique. C'est la seule solution correcte.

Notation :
30% Détecter le problème de cycle.
70% solution pour casser le cycle (attention la seule solution est celle de casser la dépendance de reste vers serviceconcrets. La solution inverse est fausse car la question précise clairement il faut que le package reste soit réutilisable !!)

Pour les questions E2, E3 et E4 outre les opérations présentées dans le diagramme de la figure 1, les seules autres opérations pouvant (et devant) être utilisées sont les constructeurs

E2 (2 points). Donnez le diagramme de séquence correspondant au scénario montrant un utilisateur qui se connecte à la box et l'initialise en suivant les détails de l'étape **Initialisation et Activation de la Box** présentés ci-dessus.
Correction :



Notation :

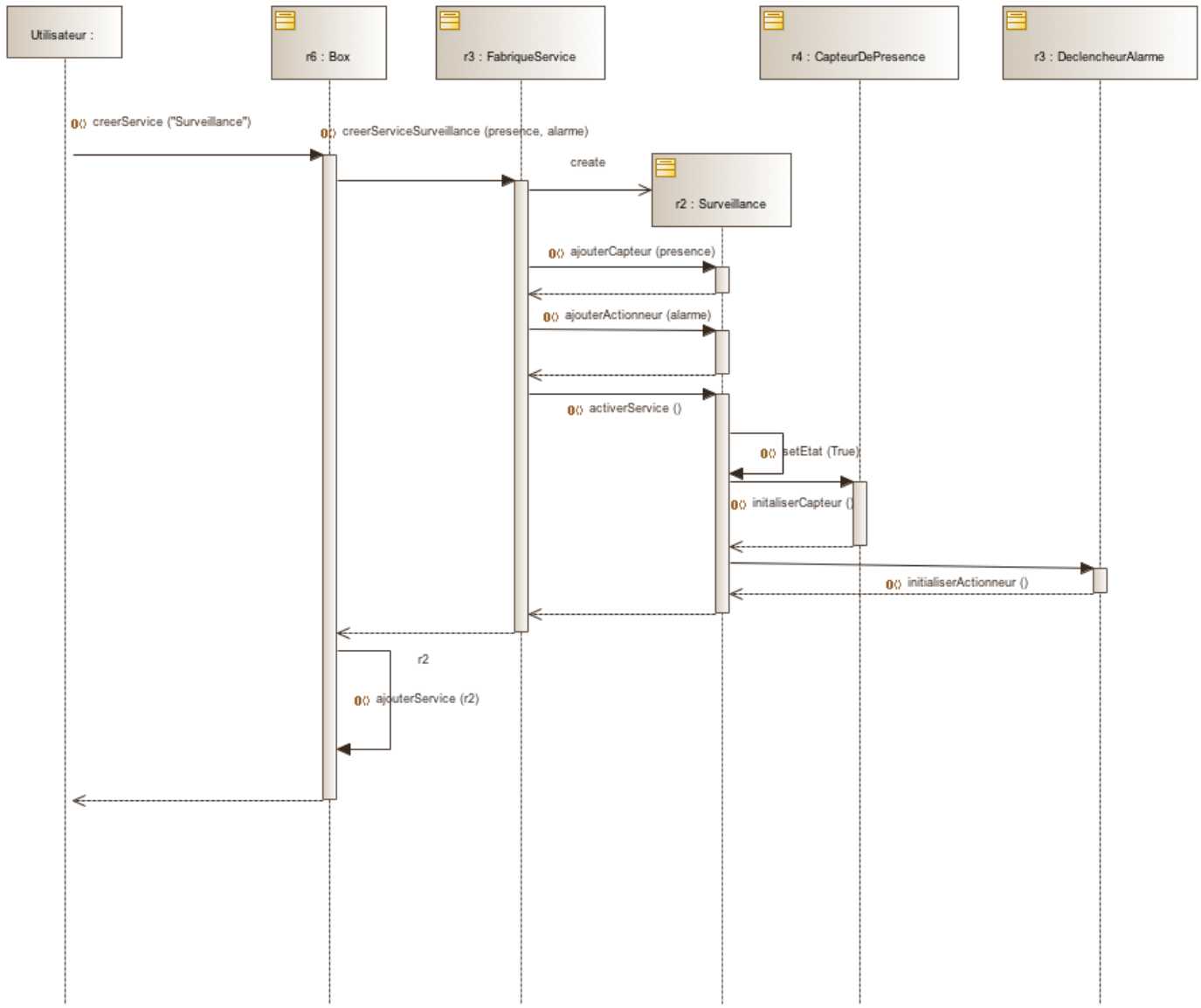
- **15% pour les objets dans le diagramme de séquence. Utilisateur est un objet non typé (-5% sinon).**
- **20% pour la connexion**
- **40% pour l'étape d'initialisation dont :**
  - **10% message initialiser**
  - **5% pour chaque création (au total 20%)**
  - **10% pour montrer que la création des objets est réalisée dans le contexte de initialiser(). On voit clairement que le message de retour de initialiser vient après les messages de création.**
- **25% activiation dont :**
  - **10% message activer()**
  - **10% message setEtat(True) (-5% qi on montre pas la valeur de paramètre à True)**
  - **5% pour le fait de voir clairement que le message de retour de activer vient après les message setEtat().**
  -

REMARQUES SUPPLEMENTAIRES :

Un message est correcte si on a le bon sender et le bon receiver avec els bons paramètres. Sinon 0%

E3 (3 points). Donnez le diagramme de séquence correspondant au scénario de l'étape **Ajout et activation de services** présentée ci-dessus. Vous considérez l'exemple d'un utilisateur qui veut créer et ajouter un service de surveillance.

Correction :



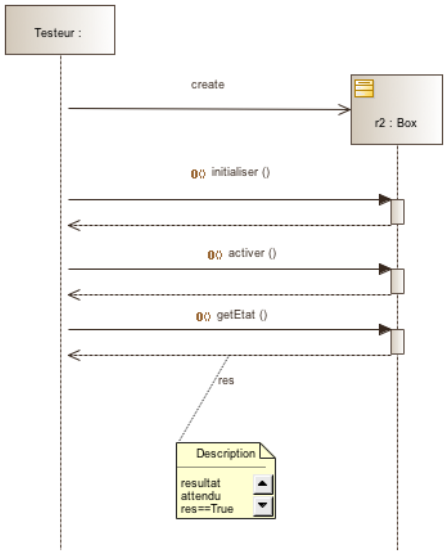
Notation :

- **10% pour les objets corrects**
- **Création de service :**
  - **10% creerService() avec le bon paramètre 0% si paramètre faux !)**
  - **10% creerServiceSurveillance() (-0% si pas de parametre)**
  - **10% message de création de surveillance (0% si on crée un Service)**
  - **5% ajouterCapteur()**
  - **5% ajouterActionneur()**
  - **35% Activation dont :**
    - **10% pour activer()**
    - **10% setEtat(True) (0% si pas de valeur du paramètre)**
    - **5%% initialiserCaptteur()**
    - **5%%initialiserActionneur()**

        - **5% les messages setEtat(), initialiserCapteur(), initialiserActionneur() sont réalisés dans le contexte de activer(). On voit clairement que le message de retour de activer vient après les autres messages.**
    - **15%ajouter Service(r2) (0% si pas de paramètre)**
    - **10% On voit clairement que le message de retour de creerService() vient après les autres messages.**
    -
-

E4 (3 points). Définissez une faute pouvant se produire lors de l' activation de la box. Donnez le diagramme de séquence de test permettant de révéler **cette faute**.

Correction
Une faute : un utilisateur active la box mais à la fin de la procédure l'état de la box est toujours incativée
DS de Test :



Notation
100% si tout

- **40% pour la définition d'une faute possible**
- **60% pour le test (le test doit révéler la faute décrite dans la première partie, 0% sinon).**
    - **une autre instance que est présente ➔ 0% sur 60%**
    - **une ou plusieurs opérations présentes ne sont pas à l'initiative du testeur. Exemple si setEtat() est apparue sur m'objet Box est apparue ➔ 0% sur 60%**
    - **-10% sur le diagramme de séquence de test qui ne montre pas la création de l'instance Box.**
    -

REMARQUES SUPPLEMENTAIRES :

- **On accepte aussi une solution qui ne montre que les messages activer() et getEtat() sans le message d'initialisation.**

E5 (2 points). Considérons l'opération *desactiverService*() qui permet de désactiver un service actif. Pour chacune des signatures suivantes, dites quelle(s) classe(s) pourrai(en)t définir cette opération :

1. *desactiverService (in service : Service)*
2. desactiverService(In nomService : string)

Correction :

1. *desactiverService (in service : Service*) dans toutes les classes car on a la référence du service comme argument.
2. desactiverService(In nomService : string) seulement dans la classe Box. En effet, à partir du nom la seule classes qui peut identifier le service concerner c'est la classe Box.

Notation :

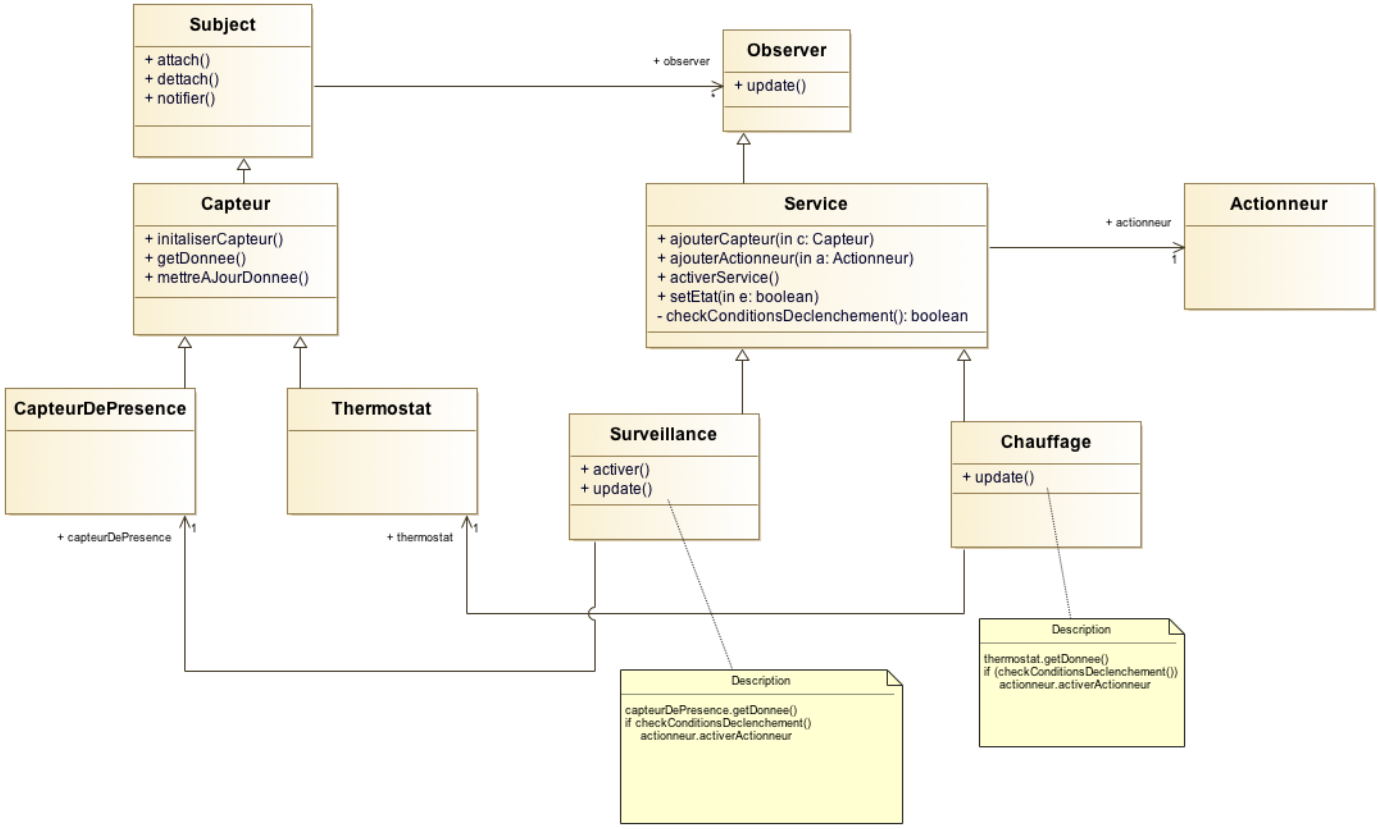
50% par réponse correcte.
0% sinon

E6 (3 points). En plus des deux étapes **Initialisation** et **Ajout et activation de services**, **iDomotique** doit gérer l'étape de **Contrôle de Services** qui permet de déclencher les actionneurs quand les capteurs détectent des données particulières de l'environnement. L'étudiant de L3 demande votre aide pour mettre à jour le diagramme de classes de la figure 1 pour réaliser cette fonctionnalité. Le principe de contrôle de service est illustré par le texte suivant extrait de la documentation fournie par l'étudiant :

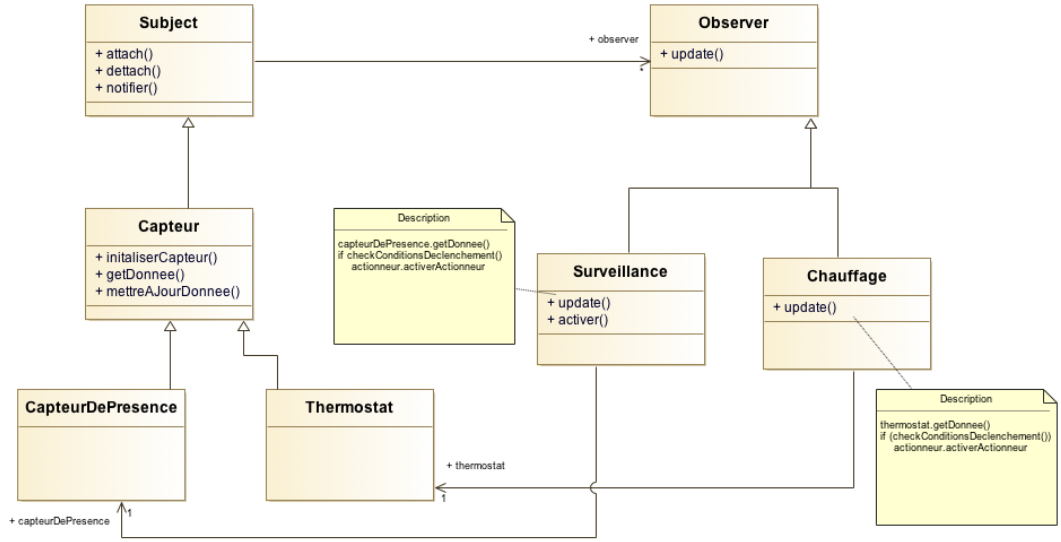
« ….à chaque fois que l'opération *mettreAJourDonnee()* de la classe *Capteur* est appelée, le capteur informe le service concerné (i.e., le service lié à ce capteur). Pour chaque notification, le service récupère les nouvelles données du capteur (à l'aide de l'opération *getDonnee()* de *Capteur*) . Il vérifie par la suite les conditions de déclenchement (opération *verifierConditionsDeclenchement()* héritée de *Service*) et en fonction de cette vérification, il active l'actionneur lié à ce service (en invoquant l'opération *declencherActionneur()*) héritée de *Actionneur* … »

Mettez en place une solution pour réaliser la fonctionnalité **Contrôle de Services** en se basant sur le Design Pattern Observer (vue en cours sont la description est rappelée en Annexe) et mettez à jour le diagramme de classes de la figure 1 (vous ne présenterez que la partie du diagramme qui concerne le design pattern Observer).

Correction :
Solution 1 :



Solution 2 :



Notation :

- **5% Subject (on accepte aussi Observable) avec les 3 méthodes (attach, dettach, notifier) + association * vers Observer**
- **5% Observer avec la méthode update()**
- **20% SubjetConcret : la classe Capteur qui hérite de Subject**
- **20% Service hérite de Observer (ou 2 eme solution : Chauffage et Surveillance héritent de Observer)**
- **50% notes de code:**
  - **25% pour la note de code de update() dans Surveillance**
  - **25% pour la note de code de update() pour Chauffage**

REMARQUES SUPPLEMENTAIRES :

- **le fait qu'il n y a pas de lien entre Service et Capteur, la définition de update doit être obligatoirement dans les opérations update() des sous services concrets (Surveillance et Chauffage).**
- **Si update est fournie dans Service avec une note de code contenant checkConditionsDeclenchement() et l'invocation de l'actionneur mais sans getDonnee() de Capteur ➔ 10% sur la note de code sur un total de 50%.**
- **Si update est fournie dans Service avec une note de code + un nouveau lien est ajouté entre Service et Capteur :**
  - **20% sur 50% car redondance avec les liens entre les services concrets et les capteurs concrets déjà présents dans le diagramme de classes.**
  - **Si on parle de cette redondance et on propose une solution sans redondance ➔ 50% sur 50%**

E7 (2 points). Pour aider l'étudiant stagiaire à comprendre l'application du design pattern Observer, nous souhaitons accompagner la réponse de la question **E6** par un diagramme de séquence présentant le fonctionnement de **Contrôle de Services**. Le diagramme de séquence de la figure 2 montre le premier message concernant l'invocation de *mettreAJourDonnee()* de *CapteurDePresence* par un objet non typé que nous appelons *PersonneDetectee* (en réalité cette entité est gérée par les signaux de Capteur mais pour simplifier, nous considérons cet objet non typé). Complétez ce diagramme de séquence de la figure 2 pour montrer le fonctionnement de **Contrôle de Services** depuis la notification et jusqu'au déclenchement de l'alarme. Il est demandé donc de montrer les interactions engendrées par l'invocation de l'opération *mettreAJourDonnee()* suivant les principes de design pattern Observer.

Figure 1: Diagramme de séquence à finaliser pour Contrôle de Services de iDomotique

Correction :

Notation :

- **10% pour les objets dont :**
  - **5% pour l'objet de type Surveillance (0% si l'objet est de type Service)**
  - **5% pour l'objet de type DeclencheurAlarme**
- **20% pour notifier**
- **20% pour update() vers le service concret de type Surveillance**
- **20% getDonnee() (0% si le message part d'un objet typé Service)**
- **15% checkConditionsDeclenchement()**
- **15% declencherActionneur()**

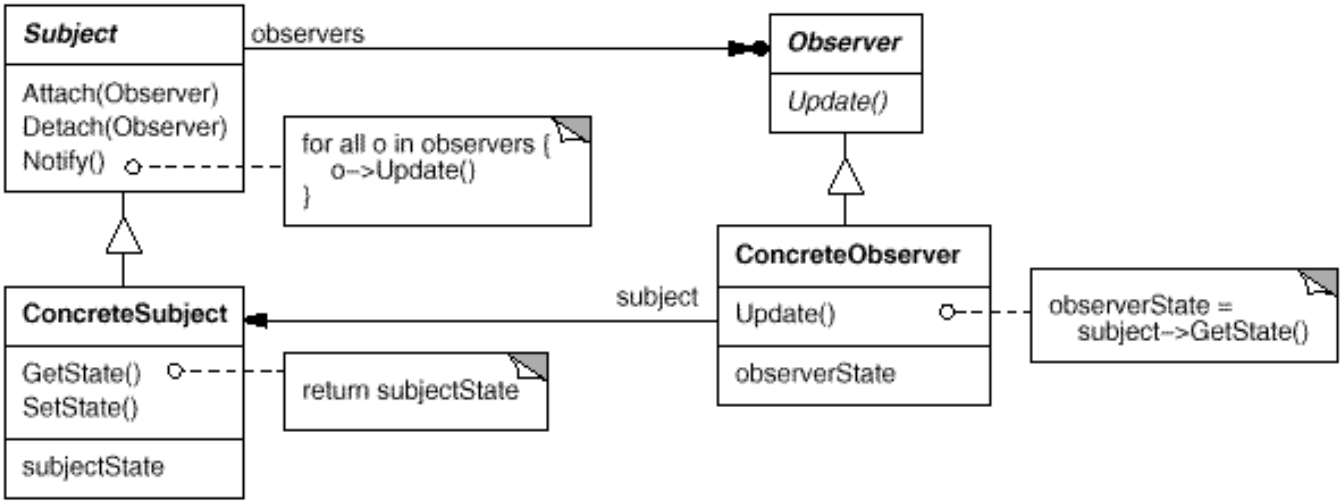
REMARQUES SUPPLEMENTAIRES :

Il faut que le diagramme de séquence doit cohérent avec la réponse à la E6.

ANNEXE : Design Pattern « Observer»

Objectif et Motivation : Permet de définir un lien de dépendance entre un objet appelé « Subject» et plusieurs objets « Observer». A chaque fois que l'état de « Subject » est modifié, tous les objets « Observer » sont notifiés.

Solution :


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
ConcreteSubject
subject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState