Cartouche du document

Année : ING 1
Matière : Méthodologie d'analyse
Activité : Examen

Objectifs

Cet examen de méthodologie d'analyse porte sur

- L'analyse de l'environnement : cas d'utilisation et acteurs
- Les diagrammes de classes
- Les diagrammes de séquences
- Les diagrammes d'états (l'exercice sera simple)
- Les mappings UML vers Java
- Les notions d'interface et le pattern MVC

Tous les documents sont autorisés.

La durée de l'examen est de 1h30 heure.

Votre examen sera corrigé par le professeur qui vous a suivi en TD. Vous êtes donc priés d'inscrire votre groupe de TD sur votre copie.

Sommaire des exercices

Corps des exercices

1 - Etude de cas simple - 8 Points

Enoncé :

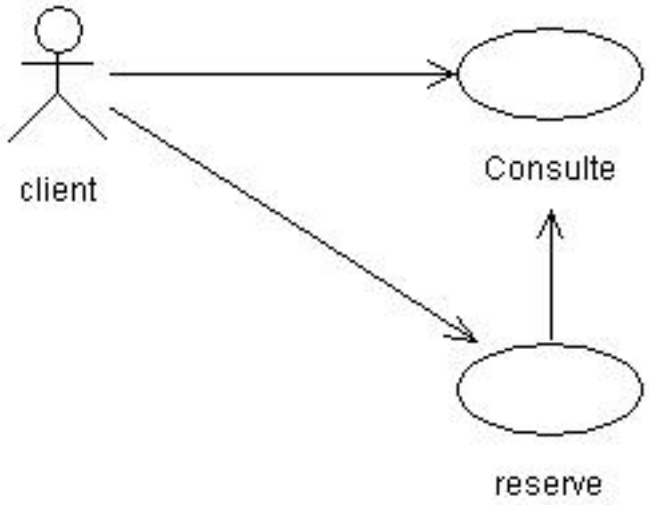
Nous nous intéressons au développement d'un Système automatique de réservation de billets de trains. L'utilisateur se présente devant une billetterie automatique. Il peut consulter l'ensemble de trains après avoir fixé le lieu de départ, le lieu d'arrivée ainsi qu'une heure approximative de départ. Le système fournit plusieurs choix possibles. Chaque choix correspond à une liste de trains possibles répondant à la requête de l'utilisateur (Le système élimine automatiquement chaque choix contenant un train complet par rapport aux places disponibles). L'utilisateur peut ensuite choisir une liste donnée et réserver sur l'ensemble de trains constituant cette liste.

Question 1)

Enoncé de la question

Donner le diagramme de cas d'utilisation.

Solution de la question



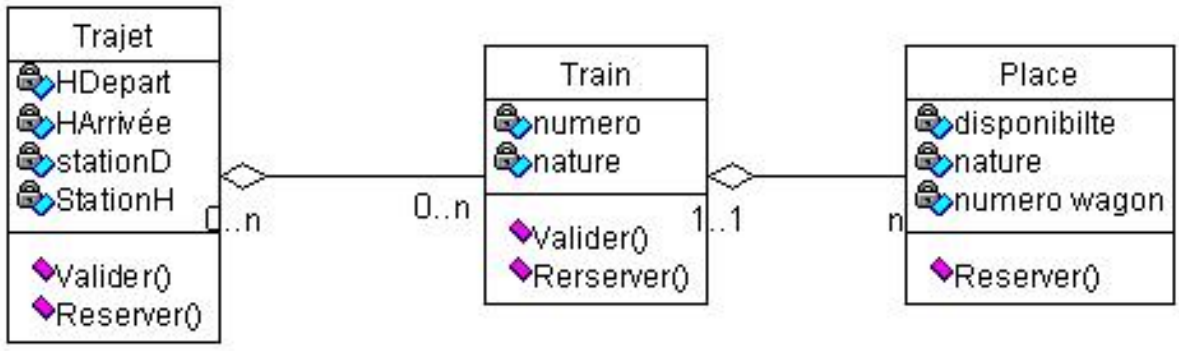
Question 2)

Enoncé de la question

Donner le diagramme de classes. Les attributs et les méthodes doivent être présentés.

Préciser sur ce diagramme les multiplicités et les contraintes.

Solution de la question

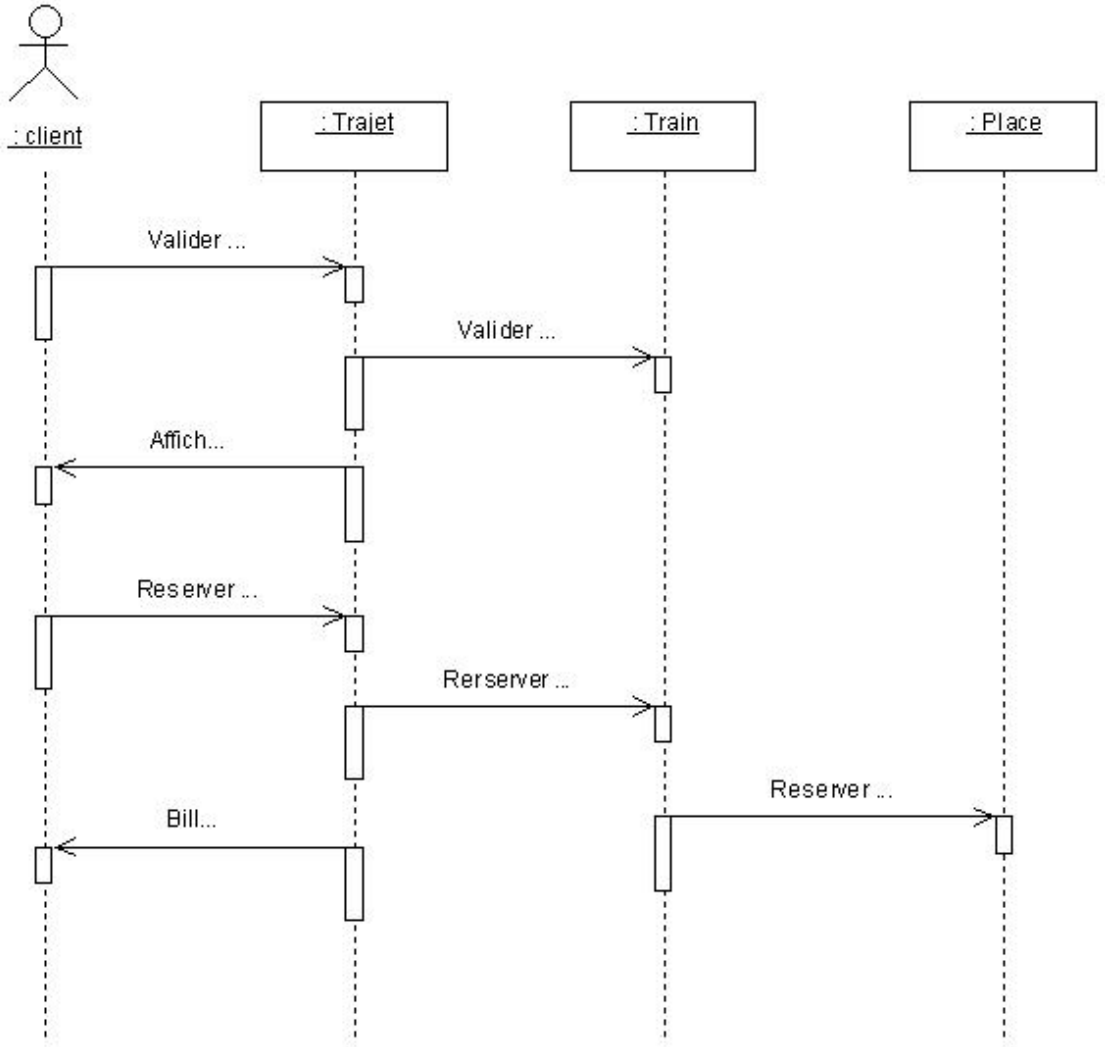


Question 3)

Enoncé de la question

Donner les diagrammes de séquences correspondant aux scénarios normaux pour chaque cas d'utilisation.

Solution de la question



2 - Etude d'une machine à laver - 5 Points

Enoncé :

L'objectif de cet exercice est de mettre en oeuvre un diagramme de transitions d'états pour l'objet *machine à laver*.

Par défaut la machine est en mode arrêt. Dés le déclenchement, la machine se met en mode pré-lavage. Au bout de deux minutes elle passe en mode lavage.

Quand la machine est en phase de lavage ou de pré-lavage, le client peut appuyer sur le bouton d'arrêt d'urgence. Dans ce cas la machine se met en attente. Le client a alors deux minutes pour reprendre le lavage ou le pré-lavage : la machine se met automatiquement en phase de lavage ou de

pré-lavage suivant la phase au cours de laquelle elle a été interrompue. Si le client n'intervient pas au bout des deux minutes, la machine s'arrête.

La machine passe en phase de séchage au bout de quatre minutes après le lavage. Là encore, le client peut interrompre le séchage mais dans ce cas la machine se mettra automatiquement en arrêt. Au bout de deux minutes de séchage la machine s'arrête quoiqu'il arrive.

Question 1)

Enoncé de la question

Inspirer vous du texte pour lister l'ensemble d'opérations (ou de méthodes) que doit avoir la classe *machine à laver* . Dites pour chaque opération (ou méthode) s'il s'agit d'une action ou d'une activité.

Solution de la question

- Attente();
- Prelaver();
- Laver();
- Sechage();

Question 2)

Enoncé de la question

Lister les événements externes et temporels.

Solution de la question

Les évenements externes

- arrêt d'urgence et reprise par le client
- interruption du séchage par le client

Les évenements temporels

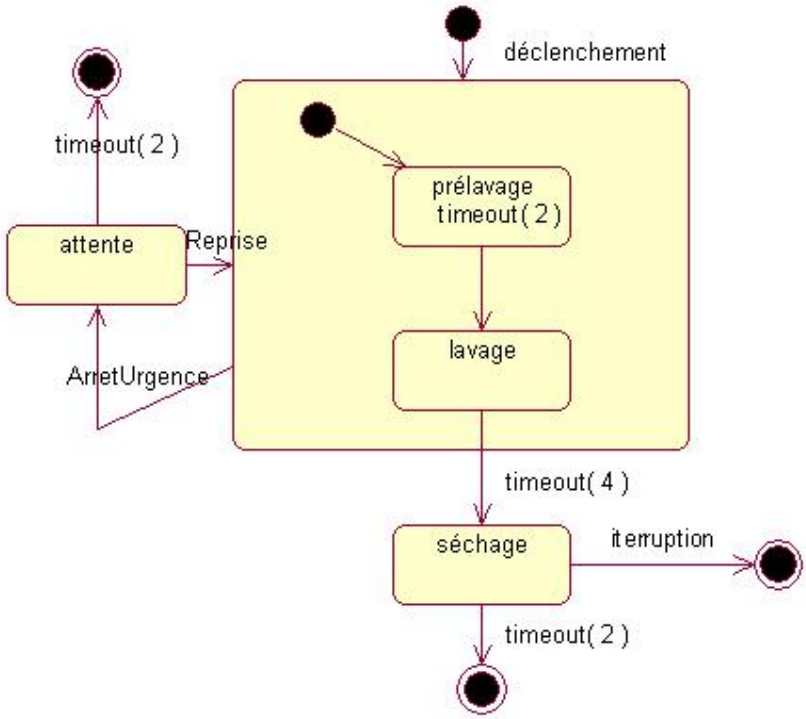
- tout changement d'état de la machine dû au TIMER

Question 3)

Enoncé de la question

Proposer un diagramme de transitions d'états pour l'objet *machine à laver*.

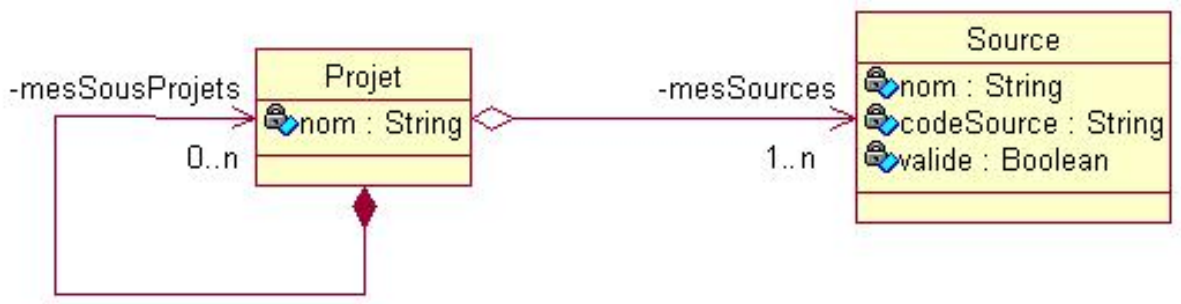
Solution de la question



3 - Mapping UML vers Java - 4 Points

Enoncé :

L'objectif de cet exercice est de mettre en oeuvre le mapping UML vers Java du diagramme suivant :



Ce diagramme modélise les projets informatiques développés dans une SSII.

Question 1)

Enoncé de la question

Faire le mapping en Java de ce diagramme .

Solution de la question

```
/**
*  Les concepts à prendre en compte pour le mapping :
* 1) encapsulation (private)
* 2) navigabilité (un projet connait ses sources et ses sous- projets, pas dans l'autre sens)
* 3) cardinalité (il faut différencier entre 2 cas : 1..n pour mesSources et 0..n pour
mesSousProjets)
* 4) agrégation (mesSources)
* 5) composition (mesSousProjets)
*
* Le code Java complet :
* Classes + attributs      : 1) + 2) + 3)
                    ====> 2 pts
* Constructeurs : 3) différences de cardinalité
            ====> 1 pt
* Méthodes : 4) + 5) différences entre agrégation et composition ====> 1 pt
*/ */

import java.util.*;

/**
La classe qui permet de gérer des projets
*/ */
public class Projet {
   private String nom;
   private Vector < Source > mesSources;
   private Vector < Projet > mesSousProjets;

  /**
  * @param n : le nom du projet
  * @param s : une référence sur le premier source du projet
  * pour assurer qu'un projet a au moins 1 source
  */ */
   Projet(String n, Source s) {
      nom = n;
      mesSources = new Vector < Source >();
      mesSousProjets = new Vector < Projet >();
      mesSources.add(s);
   }

   // méthodes get éventuellement
   // méthodes set éventuellement

   /**
    * Un projet connait ses sources, la relation est l'agrégation
    * @param s : une référence sur le source à ajouter
```

```
     */ */
    void ajouterSource(Source s) {
        mesSources.add(s);
    }

    /** Un projet connait ses sous projets, la relation est une composition
     * on crée le sous projet dans la classe Projet pour assurer cette composition
     * conséquences : quand on veut ajouter des sources et des sous-projets à
     * un sous-projet, il faut tourjours passer par le projet père
      * @param n : le nom du sous projet
     * @param s : une référence sur le premier source du sous projet
     */ */
    void ajouterSousProjet(String n, Source s) {
        Projet sp = new Projet(n,s);
        mesSousProjets.add(sp);
    }
}

/**
La classe qui permet de gérer des sources
*/ */
public class Source {
    private String nom;
    private String codeSource;
    private Boolean valide;

    /** Un projet connait ses sous projets, la relation est une composition
     * on crée le sous projet dans la classe Projet pour assurer cette composition
     * conséquences : quand on veut ajouter des sources et des sous-projets à
     * un sous-projet, il faut tourjours passer par le projet père
      * @param n : le nom du source
      * @param c : le contenu du source
     * @param v : un booléen pour préciser si le source est valide
     */ */
    Source(String n, String c, Boolean v) {
        nom = n;
        codeSource = c;
        valide = v;
    }

    // méthodes get éventuellement
    // méthodes set éventuellement
}
```

Question 2)

Enoncé de la question

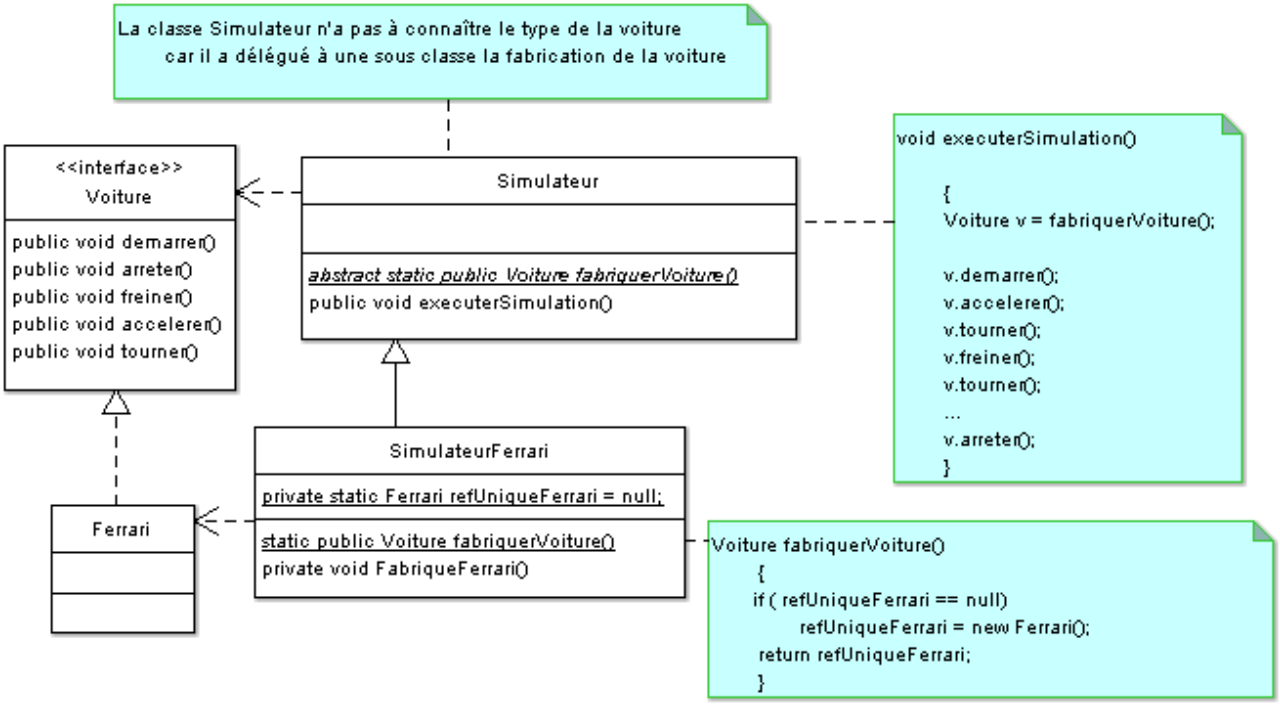
Justfier le contenu de chaque classe.

Solution de la question

4 - Interface et MVC - 4 Points

Enoncé :

Dans cet exercice, on reprend la modélisation d'un logiciel de jeu qui simule le déplacement d'une ferrari.



Question 1)

Enoncé de la question

On suppose que vous avez réellement à développer tout le simulateur avec une interface graphique. On demande d'implémenter le code en utilisant le pattern MVC (Modèle Vue Contrôleur). Expliquer pourquoi les différentes classes et interfaces (Simulateur, SimulateurFerrari, Voiture, Ferrari) sont dans l'élément Modèle du MVC.

Solution de la question

La classe Ferrari permet de traiter les messages (interface Voiture) envoyés à un objet Ferrari. La classe Simulateur permet de traiter les messages envoyés à un objet Simulateur. Un objet Simulateur va envoyer des messages à des objets qui ont interface Voiture.

Ces deux classes ainsi que l'interface Voiture décrivent donc les données et la logique de fonctionnement de ces données. En ce sens, cet ensemble constitue donc le modéle dans une logique MVC. Le futur contrôleur est un objet qui enverra des messages au simulateur en fonction des actions des utilisateurs (acteurs). De même à tout moment :

- le simulateur doit connaître la position des différentes voitures.
- chaque voiture doit connaître sa forme, sa vitesse, son accélération, ...

mais aucun de ces objets ne doit gérer son affichage. C'est un objet Vue qui sait le faire. Cet objet Vue recevra lui même des messages du contrôleur pour afficher l'espace du simulateur et les différentes voitures dans cet espace.

Question 2)

Enoncé de la question

On nous demande de créer une nouvelle version de ce jeu qui doit maintenant simuler le déplacement d'une toyota plexus.

Que peut-on reprendre de l'existant et pourquoi ?

Solution de la question

On peut reprendre de l'existant :

- L'interface Voiture car pour être utilisé par le simulateur un objet ToyotaPlexus doit implémenter l'interface Voiture
- La classe Simulateur car les objets de cette classe envoient des messages aux objets Voiture qu'à travers l'interface Voiture

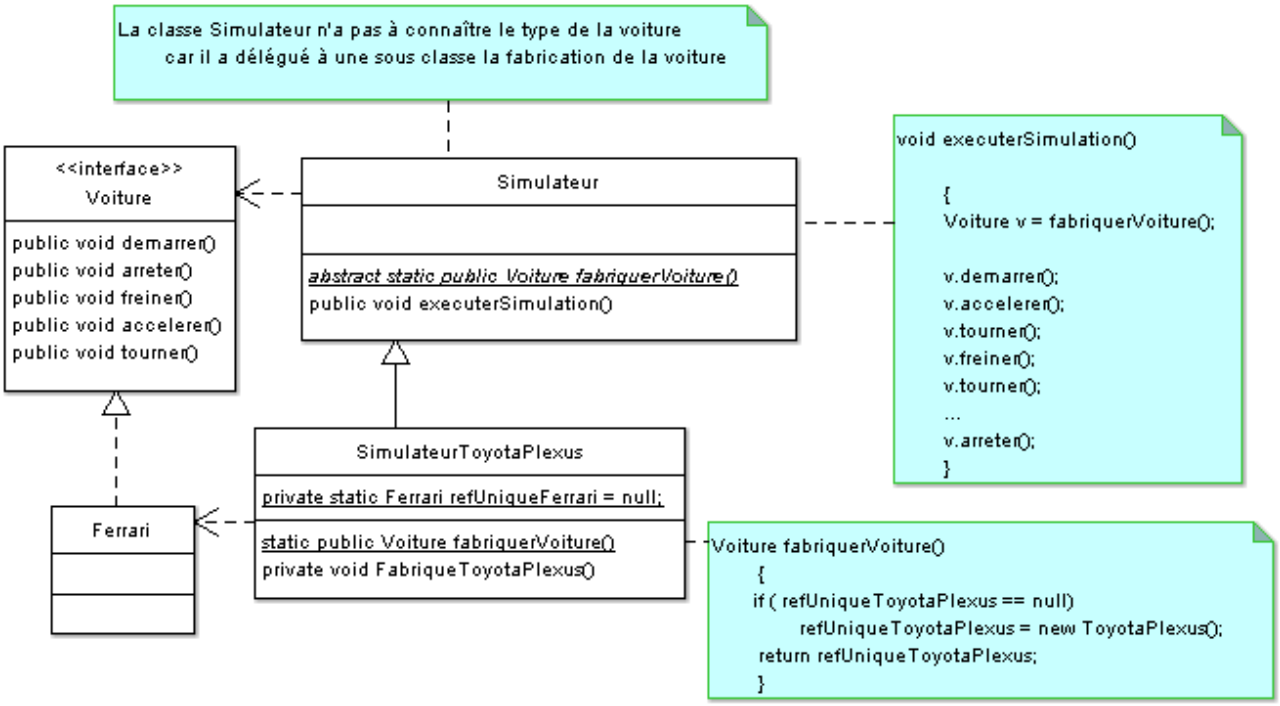
Question 3)

Enoncé de la question

Modéliser les nouvelles classes.

Solution de la question

Les nouvelles classes sont SimulateurToyotaPlexus et ToyotaPlexus.


La classe Simulateur n'a pas à connaître le type de la voiture
car il a délégué à une sous classe la fabrication de la voiture
<<interface>>
Voiture
public void demarrer()
public void arreter()
public void freiner()
public void accelerer()
public void tourner()
Simulateur
abstract static public Voiture fabriquerVoiture()
public void executerSimulation()
void executerSimulation()
{
Voiture v = fabriquerVoiture();
v.demarrer();
v.accelerer();
v.tourner();
v.freiner();
v.tourner();
...
v.arreter();
}
SimulateurToyotaPlexus
private static Ferrari refUniqueFerrari = null;
static public Voiture fabriquerVoiture()
private void FabriqueToyotaPlexus()
Ferrari
Voiture fabriquerVoiture()
{
if (refUniqueToyotaPlexus == null)
refUniqueToyotaPlexus = new ToyotaPlexus();
return refUniqueToyotaPlexus;
}