IUT Informatique **2ème Année**
Université d'Orléans
G. Cleuziou

Module : Conception et Programmation Objet Avancées (CPOA)

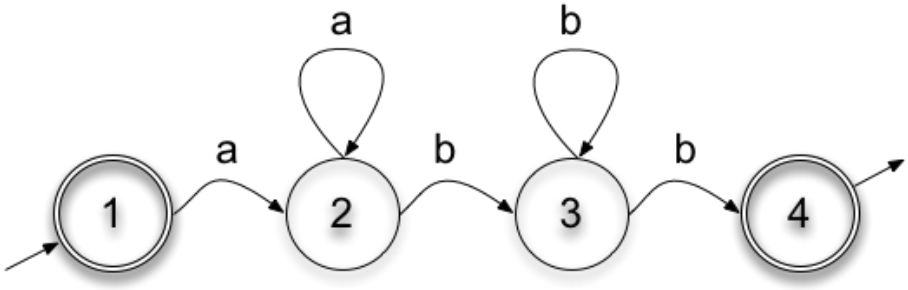
TD3 Conception Orientée Objet - Correction

PARTIE A. Conception d'un automate

Un automate fini (on dit parfois machine à états finie) est une machine abstraite utilisée en théorie de la calculabilité et dans l'étude des langages formels. C'est un outil fondamental en Informatique, où il intervient notamment en compilation des langages informatiques.

Un automate est constitué d'états et de transitions. Son comportement est dirigé par un mot fourni en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque symbole de l'entrée. L'automate est dit « fini » car il possède un nombre fini d'états distincts : il ne dispose donc que d'une mémoire bornée. L'automate est dit « déterministe » si, pour chacun de ses états, il y a au plus une transition possible pour un symbole donné et si, de plus, il a un seul état initial.

Dans l'exemple ci-dessous, l'automate est composé de quatre états numérotés de 1 à 4 ; l'état 1 est un état initial et l'état 4 est un état final. Les transitions contiennent les symboles « a » ou « b » de sorte que cet automate reconnaît certains mots et pas d'autres.



1. Cet automate est-il fini ? Est-il déterministe ?

Fini = OUI mais Déterministe = NON

2. Quels sont les mots reconnus par cet automate ?

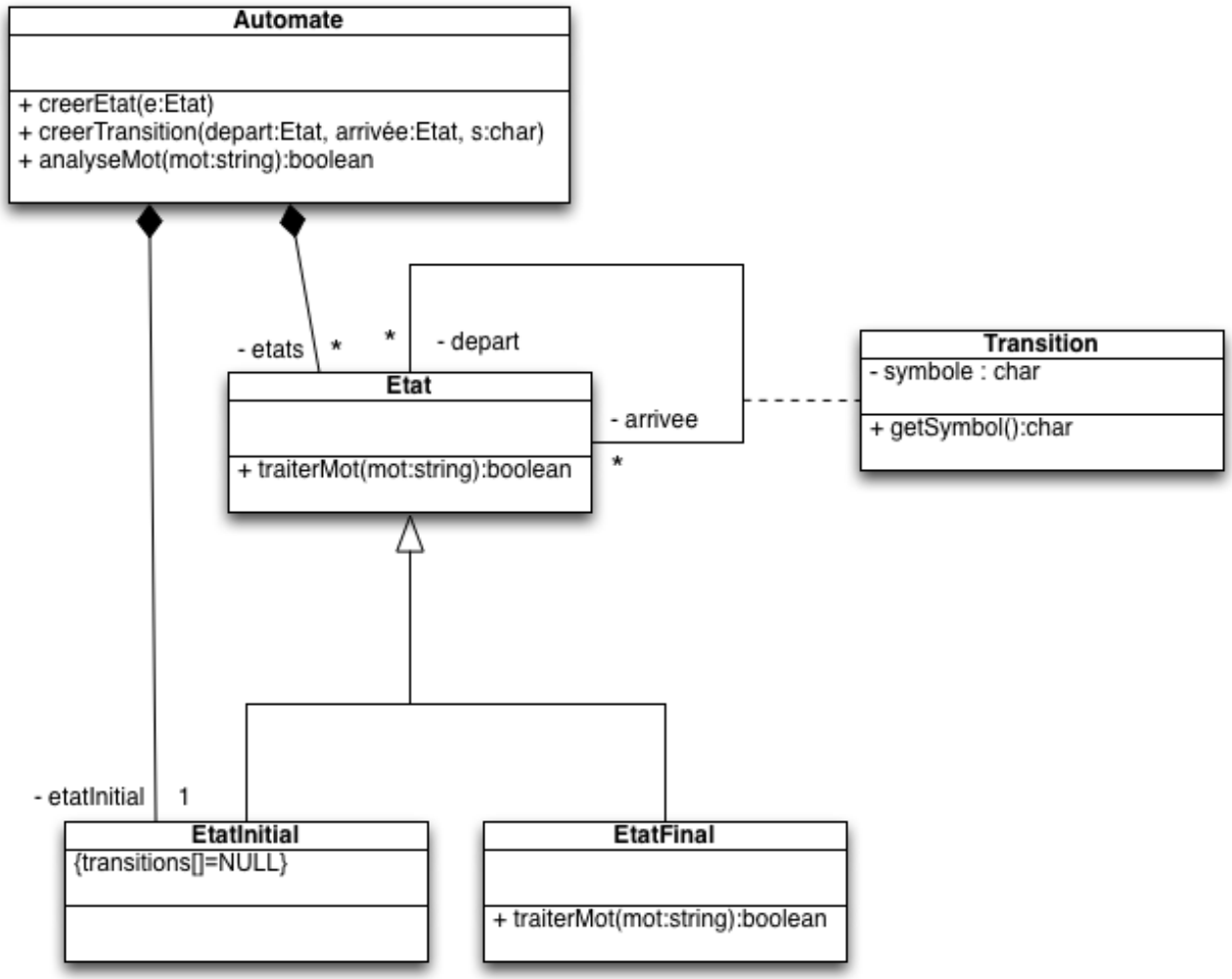
tous les mots sur l'alphabet {a,b}, commençant par des « a » (au moins un) et finissant par des « b » (au moins deux).

3. Proposez un diagramme de classe UML modélisant la structure d'un tel automate (à états initiaux/finaux uniques). Vous devrez y faire figurer toutes les classes, associations, attributs, multiplicités et opérations nécessaires à sa création et à son fonctionnement (analyse d'un mot qui doit être reconnu ou non). Pour vous aider, considérez les informations suivantes :

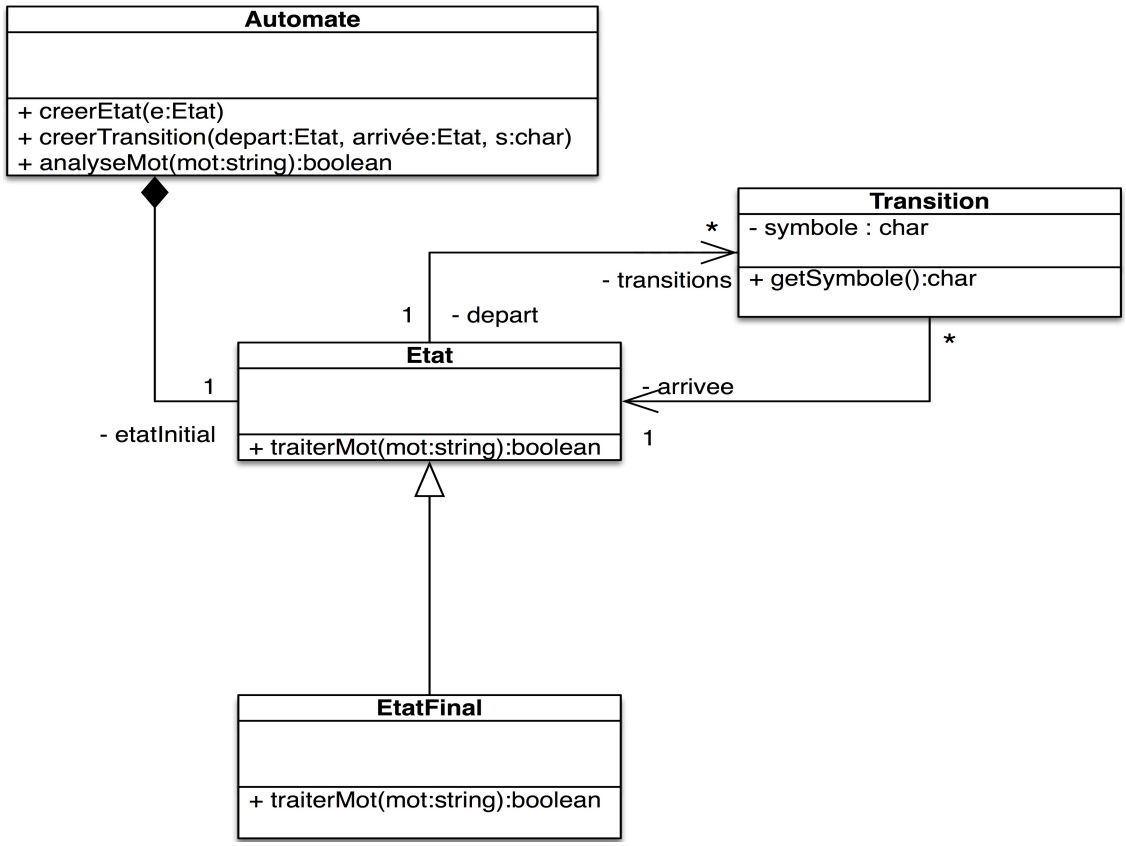
   i. un automate est composé d'états et de transitions
   ii. une transition relie un état de départ à un état d'arrivée et contient un symbole

iii. l'état final et l'état initial sont deux états particuliers

Je propose le diagramme suivant :

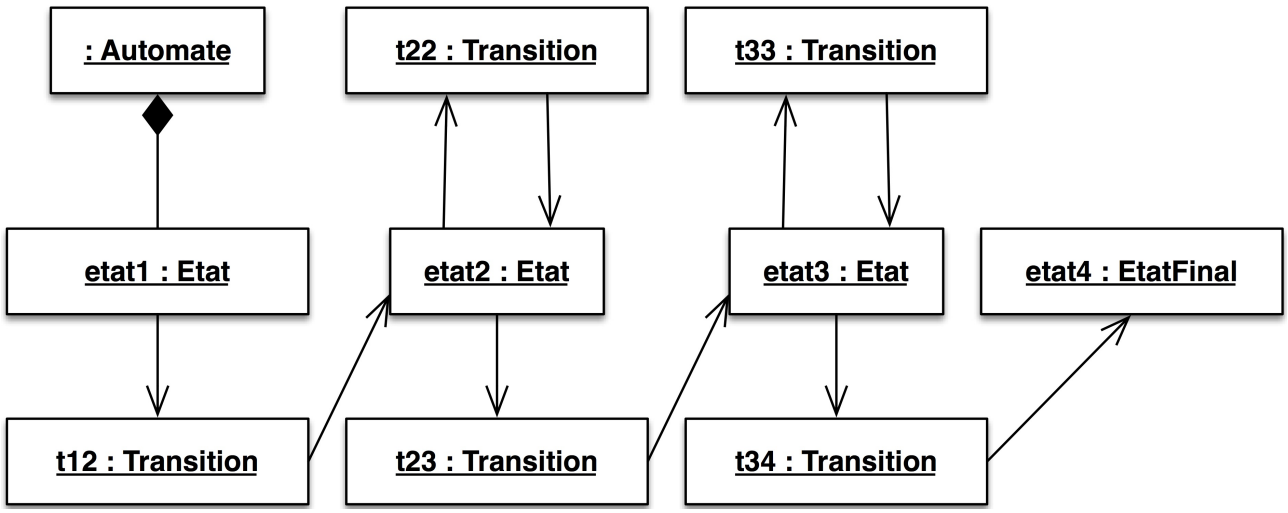


Qui se transforme en décomposant la « classe association » et se précise en réflechissant aux navigabilités indispensables :



Leur faire écrire au minimum un code Java de la structure (classes, attributs) et un pseudo-code des opérations (disponible sous célène au besoin).

Leur faire illustrer l'exemple par un diagramme d'objets :

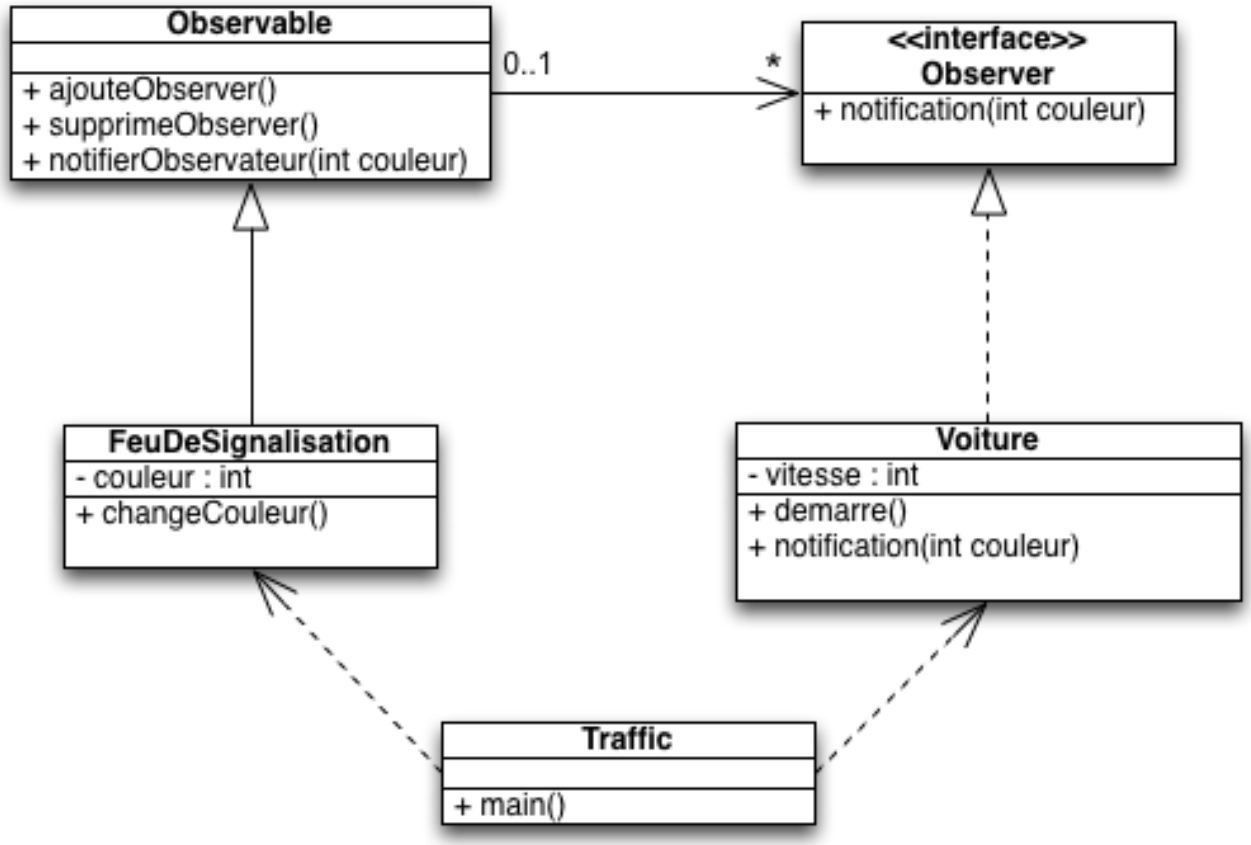


PARTIE B. Schéma de conception

On souhaite modéliser l'interaction entre un feu de signalisation et des voitures, dans une version où le feu n'a nul besoin de connaître les voitures qui lui font face et qui démarrerons dès son passage au vert. Le feu se contente d'être observé par des voitures qui se rajoutent à lui, non plus directement à partir de sa classe, mais par l'entremise d'une tierce classe (ici la classe Traffic qui

fait le raccord entre le feu et les voitures). Lorsque le feu change de couleur, il ne le signale pas aux voitures, mais celles-ci (qui lui furent associées dans la classe Traffic) en sont malgré tout avisées et agiront en conséquence. De surcroît, elle peut transmettre certaines informations concernant la nature de l'événement (par exemple, ici, la couleur du feu). Les voitures se rajoutent au feu au fur et à mesure de leur arrivée, ici gérée par la classe Traffic.

L'idée est clairement ici d'appliquer le pattern *observer* avec une classe *Observable* à laquelle s'abonne des *observer(s)* qui pourraient être de n'importe quel type (ici uniquement des voitures).



Vu qu'ils auront également à le faire en TP (sur un autre exemple), si il reste du temps on peut leur faire écrire le code des différentes classes et opérations

```
class Observable{
   private List<Observer> m_observateurs;

   public Observable()
   {
      m_observateurs = new LinkedList<Observer>();
   }

   public void notifierObservateurs(int couleur)
   {
      Iterator<Observer> it = m_observateurs.iterator();
      // Notifier tous les observers
      while(it.hasNext()){
         Observer obs = it.next();
         obs.notification(n);
      }
   }

   void ajouterObservateur(Observer observateur)
   {
      // On ajoute un abonné à la liste en le plaçant en premier (implémenté en pull).
      // On pourrait placer cet observateur en dernier (implémenté en push, plus commun).
      m_observateurs.add(observateur);
   }

   void supprimerObservateur(Observer observateur)
   {
      // Enlever un abonné a la liste
      m_observateurs.remove(observateur);
   }
}
```

```
class FeuDeSignalisation extends Observable {

private int couleur;

//constructeur
public FeuDeSignalisation(int couleur){
this.couleur = couleur;}

public void changeCouleur () {
          couleur++;
          if (couleur == 4) couleur = 1;
          if (couleur == 1) notifierObservateurs(couleur);
          else{
                    if (couleur == 3) notifierObservateurs(couleur);
                    }
          }
}
```

```
interface Observer{
   public void notification(int couleur);
}
```

```
class Voiture implements Observer {

private int vitesse;

//constructeur
```

```
public Voiture(int vitesse){this.vitesse = vitesse;}

public void demarre() {vitesse = 50;System.out.println("je demarre");}

public void notification(int couleur) {
System.out.println(couleur);
if (couleur==1) {demarre();}
}
}
```

```
public class Traffic {
public static void main(String[] args){
        FeuDeSignalisation unFeu = new FeuDeSignalisation(1);
        ArrayList<Voiture> l = new ArrayList<Voiture>();
        for (int i=0; i < 3; i++){
                l.add(new Voiture(0));
                unFeu.ajouterObservateur(l.get(i));
                }
        for (int i=0; i<4; i++){
                unFeu.changeCouleur();
                }
        }
}
```