Grenoble INP – Ensimag
Deuxième année

# Analyse et Conception Objet de Logiciels

## Corrections d'exercices

## Table des matières

# Exercice 7. Figures géométriques

### 1. Diagramme de classes

cf. figure 1

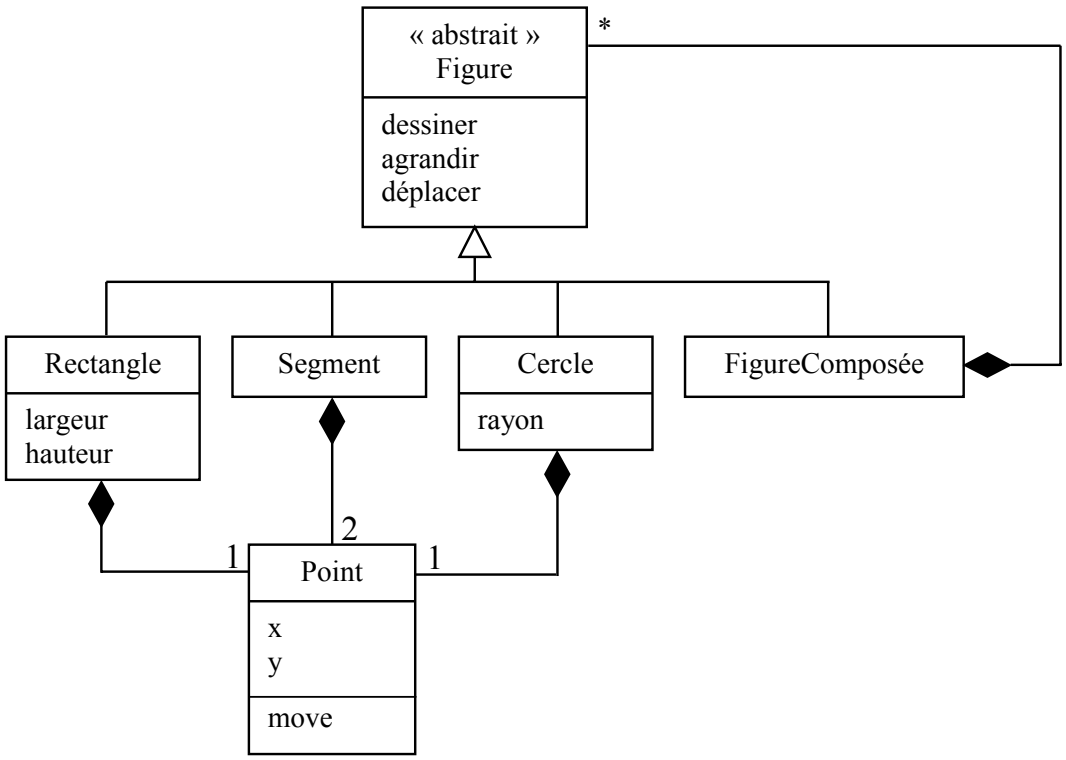



FIGURE 1 – Diagramme de classes

### 2. Diagramme d'objets

cf. figure 2

### 3. Diagramme de séquence

cf. figure 3

### 4. Diagramme de collaboration

cf. figure 4

### 5. Codage Java

Il serait souhaitable de déclarer les attributs comme *privés* et de définir des méthodes d'accès et de modifications pour ces attributs.

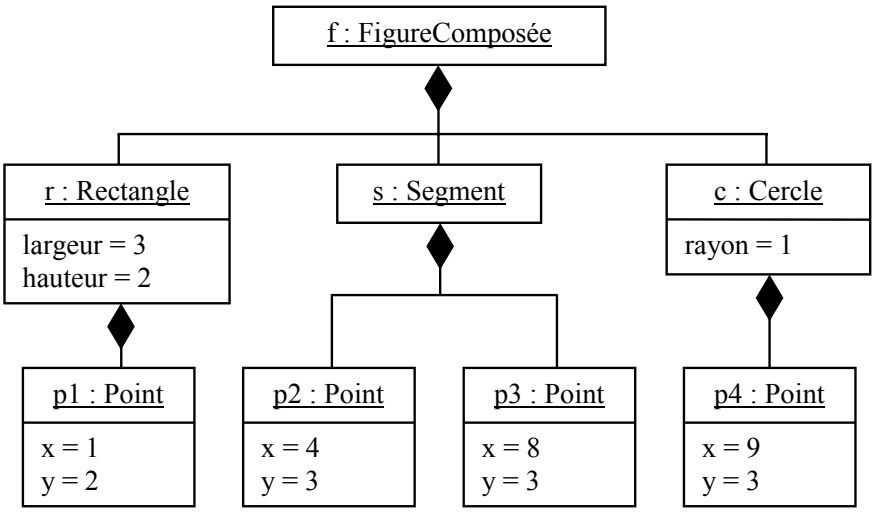



FIGURE 2 – Diagramme d'objets

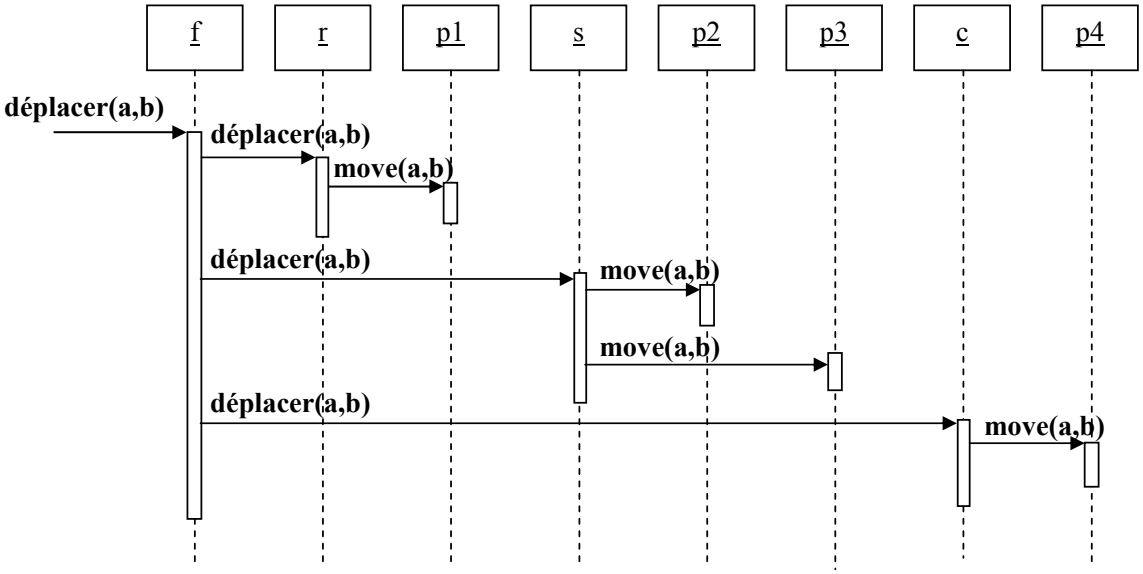



FIGURE 3 – Diagramme de séquence

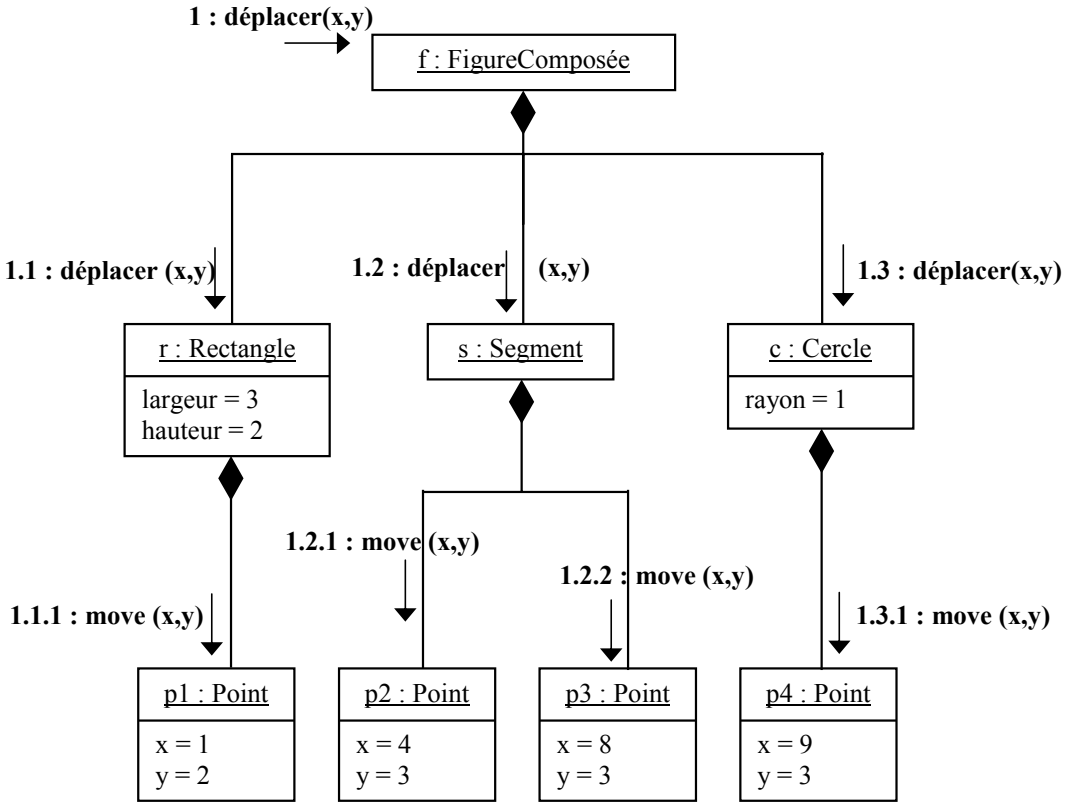



FIGURE 4 – Diagramme de collaboration

Seule la méthode `deplacer` est codée.

```
import java.util.Set ;
import java.util.HashSet ;
import java.util.Iterator ;

/** Classe des figures. */
abstract class Figure {

   /** Déplace cette figure de a suivant les abscisses et b suivant
       les ordonnées. */
   abstract void deplacer(float a, float b) ;
}

/** Classe des rectangles */
class Rectangle extends Figure {
   float largeur ;
   float hauteur ;
   Point pointHautGauche ;

   Rectangle(float largeur, float hauteur , Point pointHautGauche) {
      this.largeur = largeur ;
      this.hauteur = hauteur ;
      this.pointHautGauche = pointHautGauche ;
   }

   void deplacer(float a, float b) {
      pointHautGauche.move(a, b) ;
   }
}

/** Classe des segments. */
class Segment extends Figure {
   Point point1 ;
   Point point2 ;

   Segment(Point point1, Point point2) {
      this.point1 = point1 ;
      this.point2 = point2 ;
   }

   void deplacer(float a, float b) {
      point1.move(a, b) ;
      point2.move(a, b) ;
   }
}

/** Classe des cercles. */
class Cercle extends Figure {
   float rayon ;
   Point centre ;

   Cercle(float rayon, Point centre) {
      this.rayon = rayon ;
```

```
      this.centre = centre ;
   }

   void deplacer(float a, float b) {
      centre.move(a, b) ;
   }
}

/** Classe des figures composées. */
class FigureComposee extends Figure {
   Set<Figure> ensembleFigures ;
   // Ensemble des figures de la figure composée
   // Remarque : on utilise la généricité (Java 1.5)

   FigureComposee() {
      ensembleFigures = new HashSet<Figure>() ;
      // Constructeur générique (Java 1.5)
   }

   /** Ajoute la figure f à cette figure composée. */
   void ajouter(Figure f) {
      ensembleFigures.add(f) ;
   }

   void deplacer(float a, float b) {
      // Itération sur l'ensemble des figures
      // (Boucles for améliorée de Java 1.5)
      for (Figure f : ensembleFigures) {
         f.deplacer(a, b) ;
      }
   }
}
/** Classe des points. */
class Point {
   float x ;
   float y ;

   Point(float x, float y) {
      this.x = x ;
      this.y = y ;
   }

   /** Déplace cette figure de a suivant les abscisses et b suivant
       les ordonnées. */
   void move(float a, float b) {
      x = x + a ;
      y = y + b ;
   }
}

/**
 * Classe de test.
 * On teste la figure du diagramme d'objets.
```

```
 */
class TestFigures {
   public static void main(String[] args) {
      // Construction du rectangle
      Point p1 = new Point(1, 2) ;
      Rectangle r = new Rectangle(3, 2, p1) ;
      // Construction du segment
      Point p2 = new Point(4, 3) ;
      Point p3 = new Point(8, 3) ;
      Segment s = new Segment(p2, p3) ;
      // Construction du cercle
      Point p4 = new Point(9, 3) ;
      Cercle c = new Cercle(1, p4) ;
      // Construction de la figure composée
      FigureComposee f = new FigureComposee() ;
      f.ajouter(r) ;
      f.ajouter(s) ;
      f.ajouter(c) ;
      // Appel de la méthode deplacer
      f.deplacer(1, 2) ;
      // Vérification (oracle)
      if ((p1.x == 2) && (p1.y == 4) &&
          (p2.x == 5) && (p2.y == 5) &&
          (p3.x == 9) && (p3.y == 5) &&
          (p4.x == 10) && (p4.y == 5)) {
         System.out.println("Test␣ok") ;
      } else {
         System.out.println("Test␣échoué") ;
      }
   }
}
```

## Exercice 8. Classes et interfaces Java

### 1. Diagramme de classes

cf. figure 5.

Remarques :

- Un type est soit un type de base, soit une classe, soit une interface.
- Les paramètres d'une méthode sont ordonnés.
- Il ne faut pas mettre d'agrégation entre `Type` et `ÉlémentTypé` car un élément typé est « associé » à un type, mais ne « contient » pas de type. De plus, l'introduction d'une agrégation produirait un cycle dans le diagramme d'objets de la question 2.

Une variante avec l'utilisation d'un type énuméré (*enum*) est présenté figure 6

### 2. Diagramme d'objets

cf. figure 7.

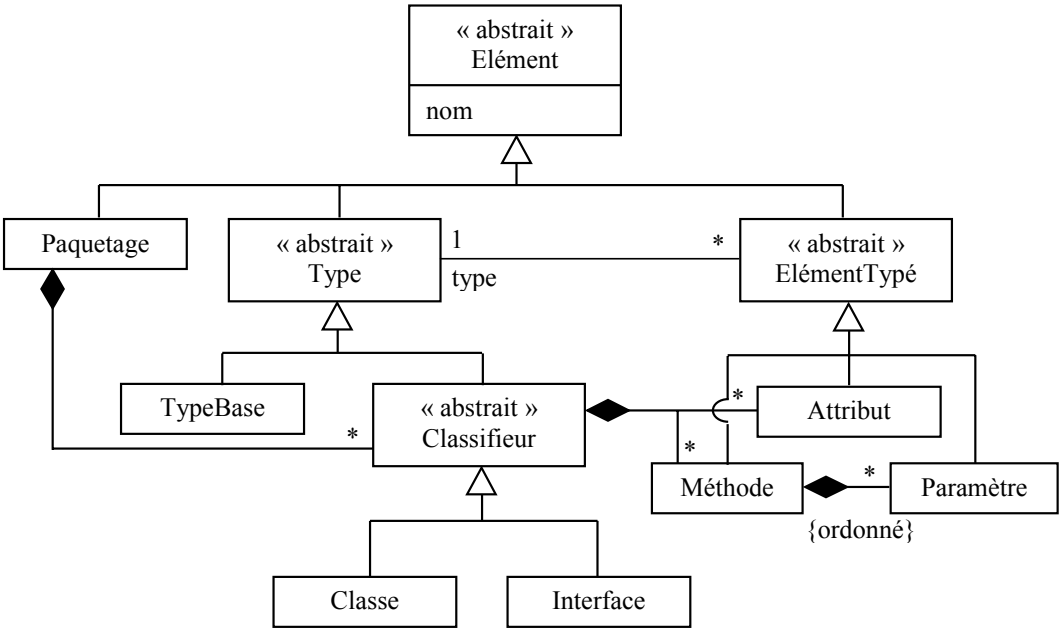



FIGURE 5 – diagramme de classes

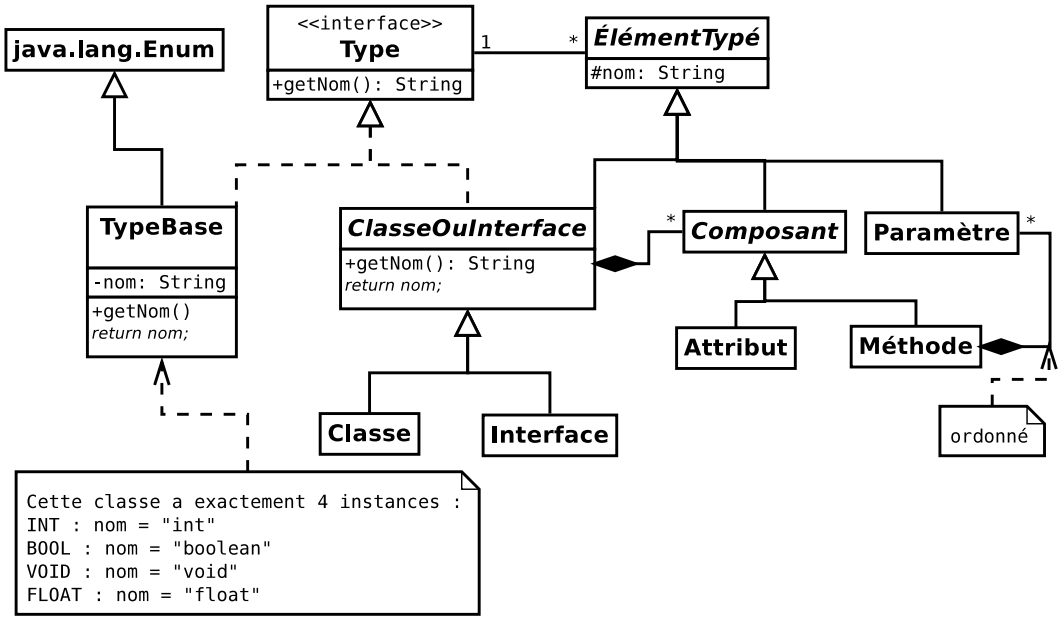



FIGURE 6 – Variante montrant l'utilisation d'un enum Java. Voir code section 5bis.

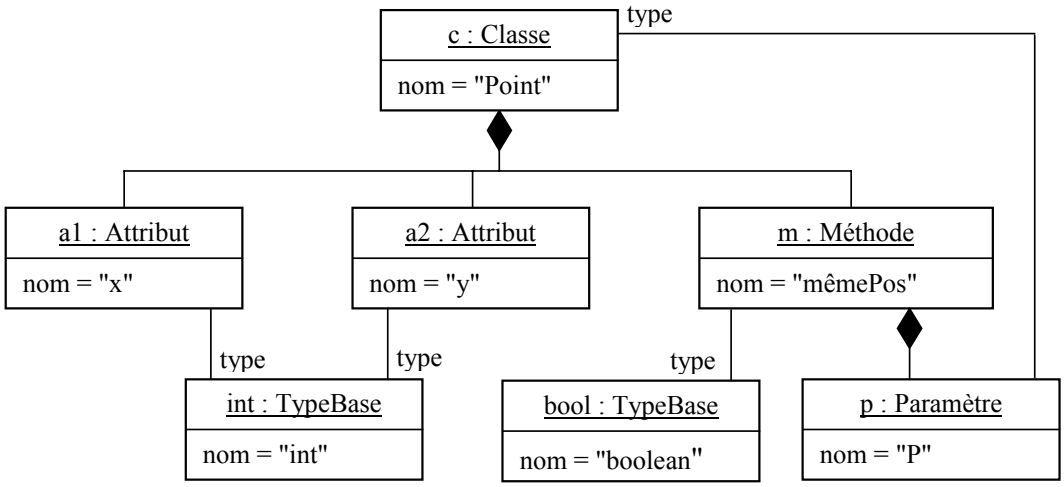



FIGURE 7 – Diagramme d'objets

### 3. Diagramme de séquences

cf. figure 8.

### 4. Diagramme de collaboration

cf. figure 9.

### 5. Codage Java

Codage des éléments en Java et de la méthode afficher. Version 1 : sans enum

Il serait souhaitable de déclarer les attributs comme *privés* et de définir des méthodes d'accès sur ces attributs.

```
import java.util.Set ;
import java.util.HashSet ;
import java.util.List ;
import java.util.Vector ;
import java.util.Iterator ;

/** Classe des éléments Java ayant un nom */
abstract class Element {
   String nom ; // Le nom de l'élément Java
   Element(String nom) {
      this.nom = nom ;
   }
   /** Affiche cet élément Java. */
   abstract void afficher() ;
}

/** Classe des paquetages Java */
```

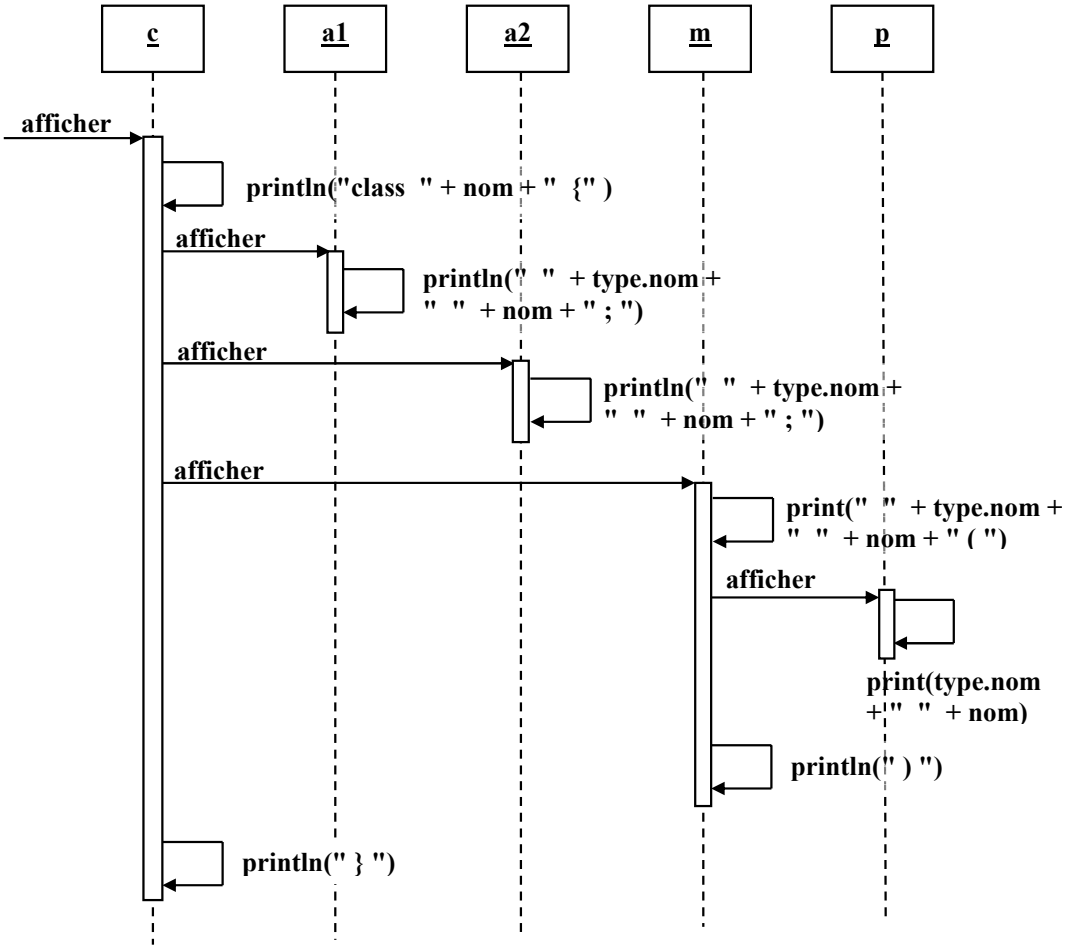



FIGURE 8 – Diagramme de séquence

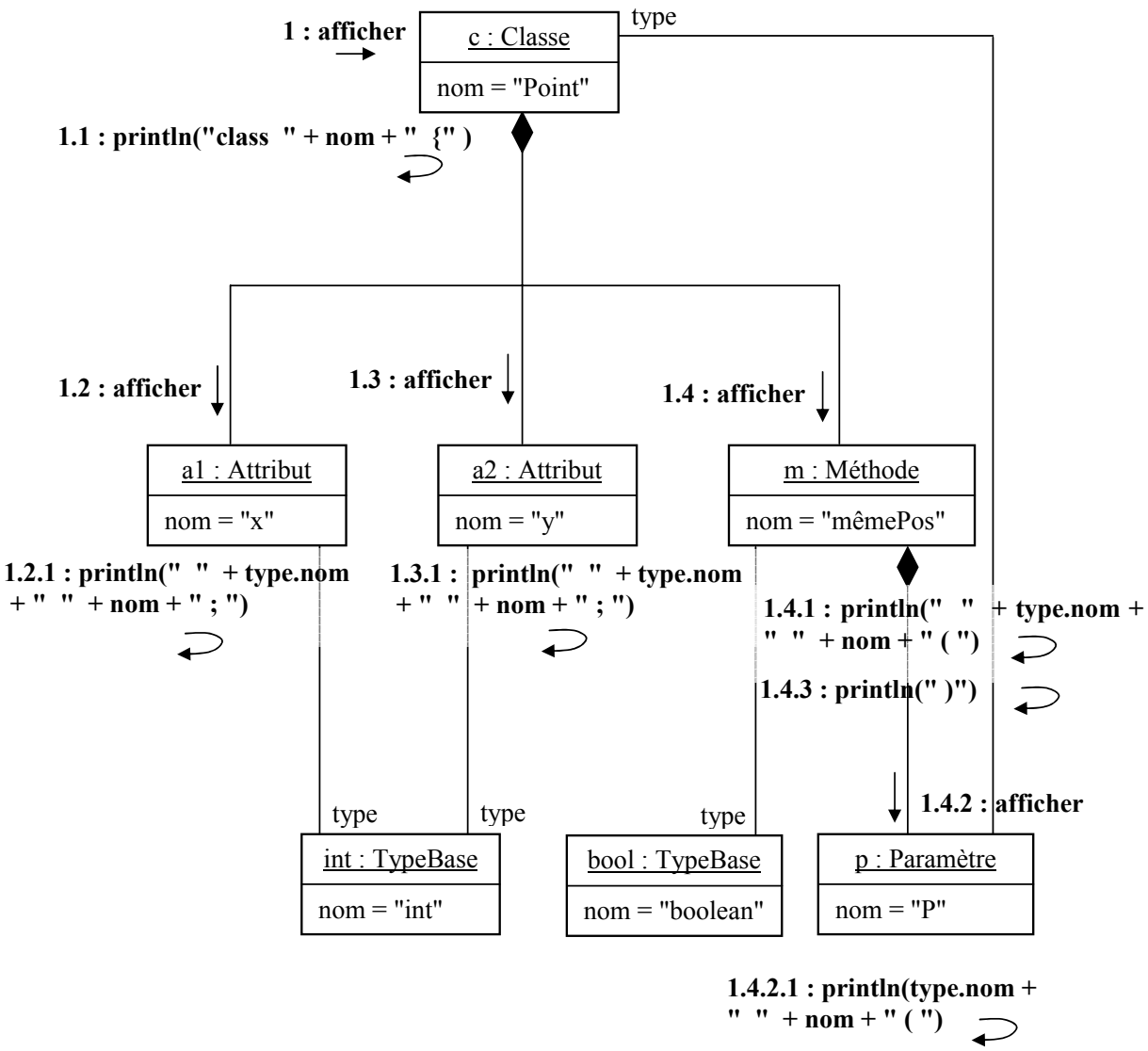



FIGURE 9 – Diagramme de collaborations

```
class Paquetage extends Element {
   // Ensemble des classes et interfaces de ce paquetage
   Set<Classifieur> listeClassifieurs ;
   Paquetage(String nom) {
      super(nom) ;
      listeClassifieurs = new HashSet<Classifieur>() ;
   }
   void afficher() {
      System.out.println("package " + nom) ;
      // Affichage de chaque classifieur de la liste
      for (Classifieur cl : listeClassifieurs) {
         cl.afficher() ;
      }
   }
}

/** Classe des types */
abstract class Type extends Element {
   Type(String nom) {
      super(nom) ;
   }
   void afficher() {
      System.out.print(nom) ;
   }
}

/** Classe des types de base */
class TypeBase extends Type {
   TypeBase(String nom) {
      super(nom) ;
   }
   // Les types de base prédéfinis
   static TypeBase typeInt = new TypeBase("int") ;
   static TypeBase typeFloat = new TypeBase("float") ;
   static TypeBase typeBoolean = new TypeBase("boolean") ;
}

/**
 * Classe des classifieurs.
 *  Un "classifieur" est soit une classe, soit une interface
 */
abstract class Classifieur extends Type {
   // Ensemble des attributs de ce classifieur
   Set<Attribut> ensembleAttributs ;
   // Ensemble des méthodes de ce classifieur
   Set<Methode> ensembleMethodes ;
   Classifieur(String nom) {
      super(nom) ;
      ensembleMethodes = new HashSet<Methode>() ;
      ensembleAttributs = new HashSet<Attribut>() ;
   }
   /** Ajoute une méthode à ce classifieur. */
   void ajouter(Methode m) {
```

```
      ensembleMethodes.add(m) ;
   }

   /** Ajoute un attribut à ce classifieur. */
   void ajouter(Attribut a) {
      ensembleAttributs.add(a) ;
   }

/** Affiche le contenu de ce classifieur. */
   void afficherContenu() {
      System.out.println("{") ;
      // Affichage de chaque attribut
      for (Attribut attribut : ensembleAttributs) {
         System.out.print("␣␣␣") ;
         attribut.afficher() ;
         System.out.println("␣;␣") ;
      }
      // Affichage de chaque méthode
      for (Methode methode : ensembleMethodes) {
         methode.afficher() ;
      }
      System.out.println("}") ;
   }
}

/** Classe des classes Java. */
class Classe extends Classifieur {
   Classe(String nom) {
      super(nom) ;
   }
   void afficher() {
      System.out.print("class␣" + nom + "␣") ;
      afficherContenu() ;
   }
}

/** Classe des interfaces Java. */
class Interface extends Classifieur {
   Interface(String nom) {
      super(nom) ;
   }
   void afficher() {
      System.out.print("interface␣" + nom + "␣") ;
      afficherContenu() ;
   }
}

/** Classe des éléments typés. */
abstract class ElementType extends Element {
   Type type ;
   ElementType(String nom, Type type) {
      super(nom) ;
      this.type = type ;
```

```
   }
   void afficher() {
      System.out.print(type.nom + "␣" + nom) ;
   }
}

/** Classe des attributs. */
class Attribut extends ElementType {
   Attribut(String nom, Type type) {
      super(nom, type) ;
   }
}

/** Classe des paramètres. */
class Parametre extends ElementType {
   Parametre(String nom, Type type) {
      super(nom, type) ;
   }
}

/** Classe des méthodes. */
class Methode extends ElementType {
   List<Parametre> listeParam ;
   Methode(String nom, Type typeRetour) {
      super(nom, typeRetour) ;
      listeParam = new Vector<Parametre>() ;
   }
   void afficher() {
      System.out.print("␣␣␣") ;
      super.afficher() ;
      System.out.print("(") ;
      // Itération sur les paramètres de la méthode
      Iterator<Parametre> it = listeParam.iterator() ;
      while (it.hasNext()) {
         Parametre param = it.next() ;
         param.afficher() ;
         if (it.hasNext()) {
            System.out.print(",␣") ;
            // La virgule ne doit être affichée que s'il reste
            // des paramètres
         }
      }
      System.out.println(")␣{␣}") ;
   }
}

/**
 * Une classe de test.
 * On affiche la classe Point correspondant au diagramme d'objets.
 */
class TestJava {
   public static void main(String[] args) {
      // Construction de la classe Point
```

```
        Classe point = new Classe("Point") ;
        // Construction des attributs x et y
        Attribut x = new Attribut("x", TypeBase.typeInt) ;
        Attribut y = new Attribut("y", TypeBase.typeInt) ;
        // Construction de la méthode mêmePos
        Methode memePos = new Methode("mêmePos", TypeBase.typeBoolean) ;
        Parametre p = new Parametre("P", point) ;
        memePos.listeParam.add(p) ;
        // Remplissage de la classe Point avec les attributs et la méthode
        point.ajouter(x) ;
        point.ajouter(y) ;
        point.ajouter(memePos) ;
        // Affichage de la classe
        point.afficher() ;
    }
}
```

### 5bis. Variante du code java, avec un enum

Ce code correspond au diagramme figure 6

```
/* Type.java */
public interface Type {
        public String getNom();
}

/* ÉlémentTypé.java */
import java.io.PrintStream;
abstract class ÉlémentTypé {

        protected final Type type;
        protected final String nom;

        ÉlémentTypé (Type t, String n) {
                type = t; nom = n;
        }

        public void afficher(PrintStream out) {
                out.print(type.getNom() + "␣" + nom);
        }
}

/* TypeBase.java */
public enum TypeBase implements Type {
        INT("int"),
        BOOL("boolean"),
        VOID("void"),
        FLOAT("float")
        ;

        private final String nom;
        private TypeBase(String n) {
                nom = n;
```

```
        }
        public String getNom() {return nom;}
}

/* Composant.java */
abstract class Composant extends ÉlémentTypé {

        protected Composant (Type type, String nom) {
                super(type, nom);
        }

}

/* ClasseOuInterface.java */
import java.io.PrintStream;
import java.util.HashSet;
import java.util.Set;
abstract class ClasseOuInterface implements Type {

        private final String nom;
        protected Set<Composant> composants;

        public String getNom() {return nom;}

        ClasseOuInterface(String nom) {
                this.nom = nom;
                composants = new HashSet<Composant>();
        }

        void ajouter(Composant c) {
                composants.add (c);
        }

        public void afficher(PrintStream out) {
                out.println(nom + "␣{");
                for (Composant c : composants) {
                        out.print("␣␣␣");
                        c.afficher(out);
                }
                out.println("}");
        }
}

/* Classe.java */
import java.io.PrintStream;
public class Classe extends ClasseOuInterface {

        public Classe(String nom) {
                super(nom);
        }

        public void afficher(PrintStream out) {
                out.print("class␣");
```

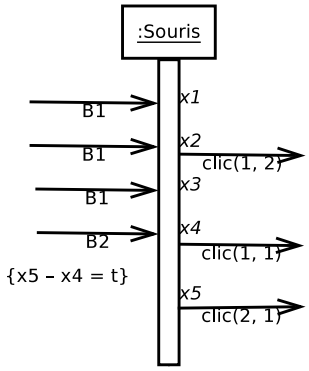
```
                super.afficher(out);
        }
}

/* Interface.java */
import java.io.PrintStream;
public class Interface extends ClasseOuInterface {

        public Interface(String nom) {
                super(nom);
        }

        public void afficher(PrintStream out) {
                out.print ("interface␣");
                super.afficher(out);
        }
}

/* Attribut.java */
import java.io.PrintStream;
public class Attribut extends Composant {

        public Attribut(Type type, String nom) {
                super(type, nom);
        }

        public void afficher(PrintStream out) {
                super.afficher(out);
                out.println(";");
        }

}

/* Paramètre.java */
public class Paramètre extends ÉlémentTypé {
        public Paramètre(Type t, String n) {
                super(t, n);
        }
}

/* Méthode.java */
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Méthode extends Composant {

        private List<Paramètre> params;

        public Méthode(Type type, String nom) {
                super(type, nom);
                params = new ArrayList<Paramètre>();
        }
```

```
        public void ajouterParamètre (Paramètre param) {
                params.add(param);
        }

        public void afficher (PrintStream out) {
                super.afficher(out);
                out.print("(");
                Iterator<Paramètre> i = params.iterator();
                while (i.hasNext()) {
                        i.next().afficher(out);
                        if (i.hasNext()) out.print(", ");
                }
                out.println(") { }");
        }

}
```

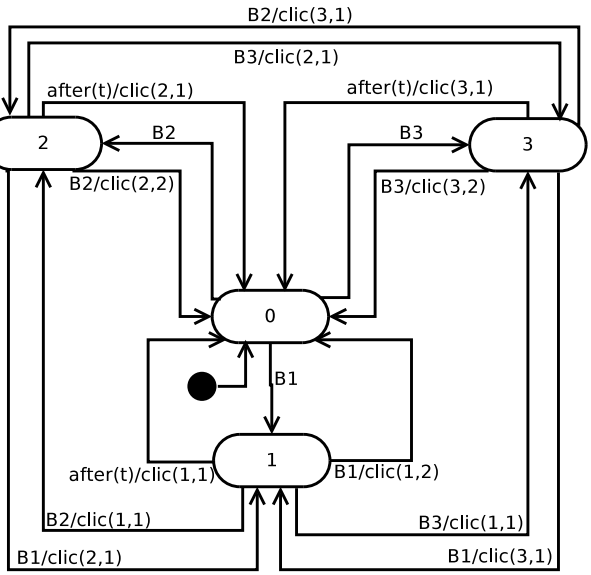
## Exercice 9. Clics et double-clics d'une souris à trois boutons

1. Un double-clic sur le bouton 1 est généré après le deuxième clic. Il n'y a pas de triple-clic donc le troisième clic est tout seul ; l'événement clic simple est généré dès l'appui sur un autre bouton. L'appui sur le bouton 2 génère également un clic simple une fois que le délai $t$ est écoulé (on sait alors qu'on n'a pas un double-clic).

   Diagramme de séquence :




2. Diagramme d'états-transition :




## Exercice 10. Comportement d'une platine cassettes

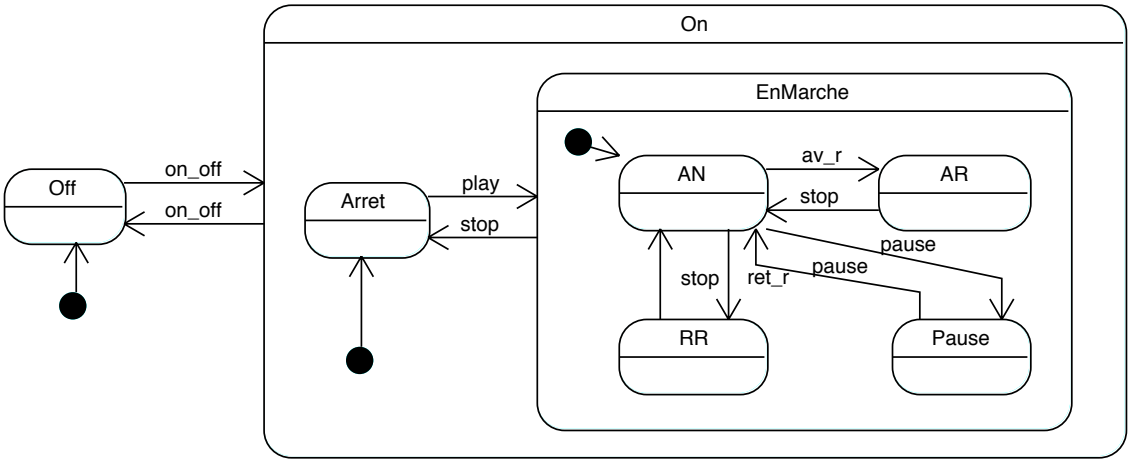



FIGURE 10 – Diagramme d'états-transitions de la platine cassettes

Voir figure 10.

## Exercice 11. Fenêtre d'impression

1. Evénements auxquels le système peut réagir :
   — Clic sur les boutons « tout imprimer » (événement *tout*), « imprimer les pages » (événement *pages*), « imprimer en recto-verso » (événement *rv*), « Ok » (événement *ok*), « Annuler » (événement *annuler*).
   — Clic sur « début » (événement *début*), « fin » (événement *fin*).

— Appui sur des chiffres lorsque « début » ou « fin » est séléctionné (événement *chiffre*).

2. Le diagramme d'états-transitions correspondant à la fenêtre d'impression est représenté Figure 2. On a deux sous-automates qui sont mis en parallèle, et l'état Pages qui est un état composite.

   On considère ici que la fenêtre se ferme lorsqu'on clique sur *ok* ou *annuler* : ces deux événement conduisent à l'état final.

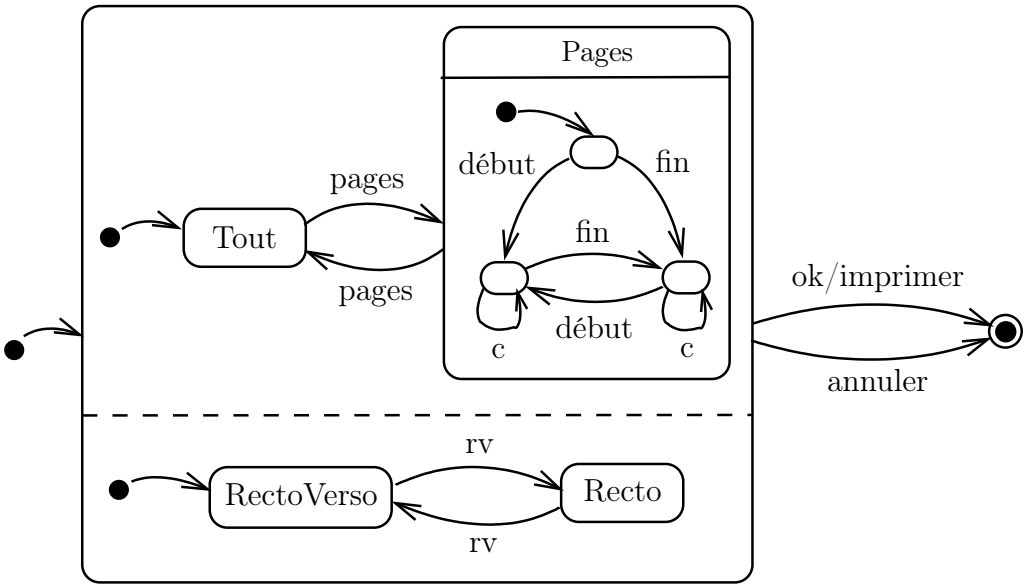



FIGURE 11 – Automate d'états-transitions correspondant à la fenêtre d'impression

3. Le diagramme d'états-transitions aplati correspondant à la fenêtre d'impression est représenté Figure 3. On peut remarquer qu'il manque beaucoup de flèches rouges sur ce diagramme : il faudrait en effet deux flèches rouges étiquetées respectivement par *ok/imprimer* et *annuler* qui partent de chaque état de l'automate et vont jusqu'à l'état final.

## Exercice 12. Bouton

### 1. Événements

Événements auxquels le bouton peut réagir :
- `setVisible(true)`, `setVisible(false)`
- `setEnabled(true)`, `setEnabled(false)`
- appui sur le bouton (`press`)
- relâchement du bouton (`release`)
- entrée de la souris sur le bouton (`enter`)
- sortie de la souris du bouton (`leave`)

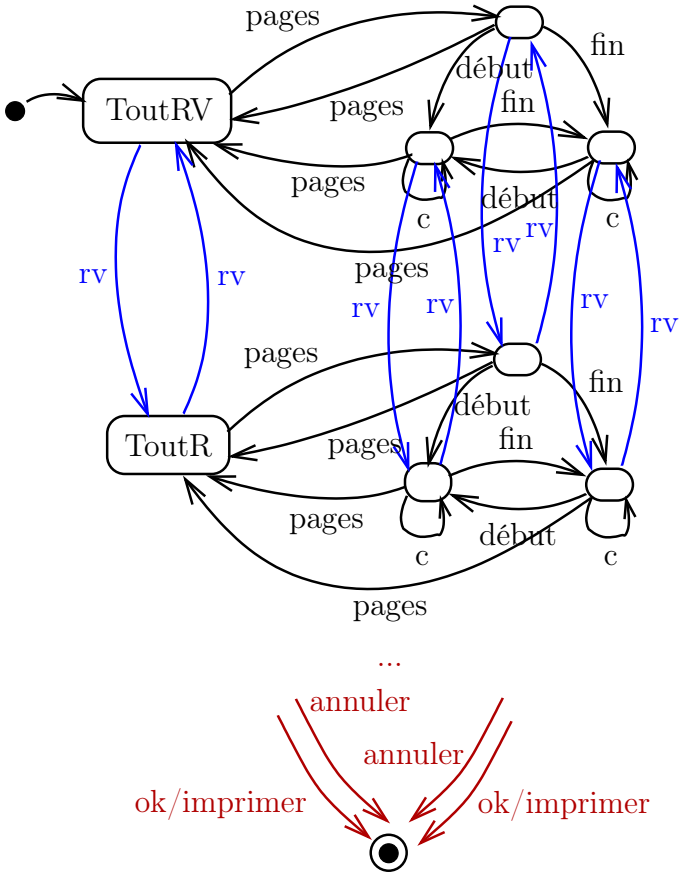



FIGURE 12 – Automate d'états-transitions aplati correspondant à la fenêtre d'impression

**2. Diagramme d'états-transitions**

On peut penser à coder le passage de visible à invisible et d'actif à inactif par deux automates en parallèle (cf. figure 13).

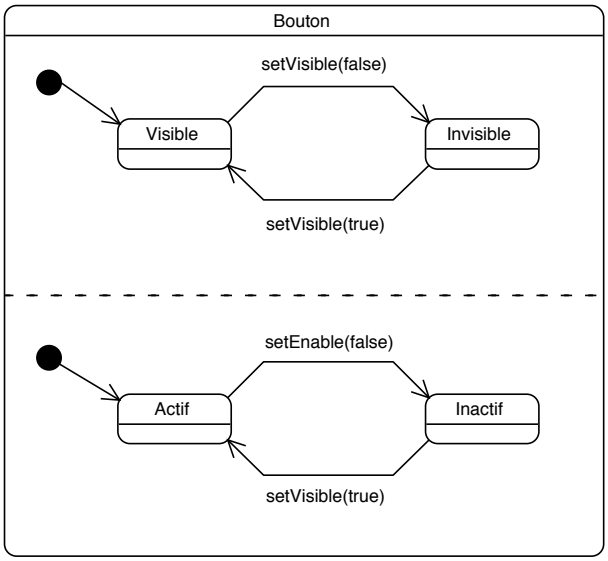



FIGURE 13 – Deux automates en parallèle

Cela pose un problème car on souhaiterait à la fois décomposer l'état `Visible` et l'état `Actif` en plusieurs sous-états. On peut aplatir cet automate (cf. Figure 14).

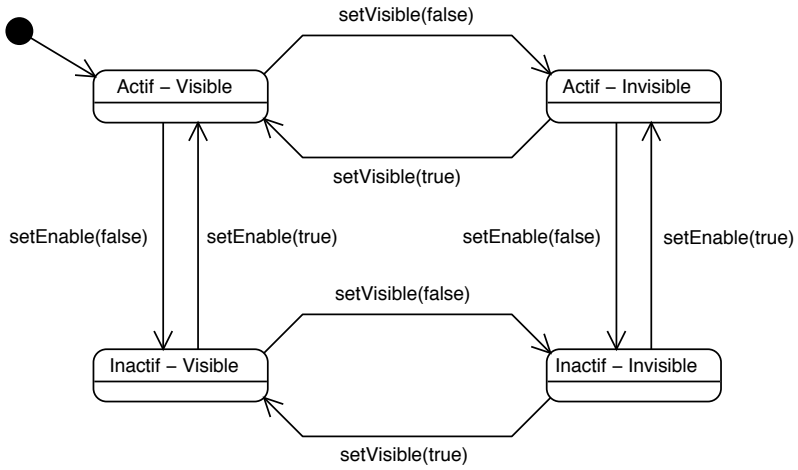



FIGURE 14 – Automate aplati

On peut maintenant décomposer l'état `Actif-Visible` en plusieurs sous-états (cf. Figure 15). Il se produit un clic lorsque partant de l'état `OutUp`, les événements `enter`, `press` et `release` sont reçus.

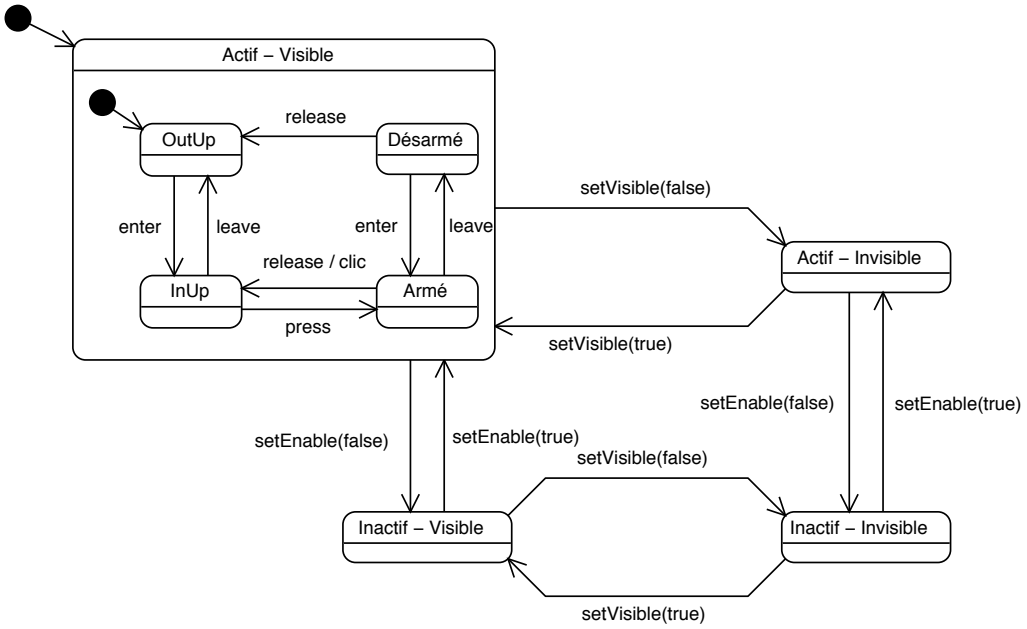



FIGURE 15 – Diagramme d'états-transitions d'un bouton

### 3. Effet d'une suite d'événements

On part de l'état `OutUp`.

- On appuie sur le bouton (événement `press`) : rien ne se passe, on reste dans le même état.
- On déplace la souris sur le bouton (événement `enter`) : on passe dans l'état `InUp`.
- On relâche le bouton (événement `release`) : rien ne se passe, on reste dans le même état.

Cette suite d'événements ne déclenche donc pas de clic.

## Exercice 16. Courriers électroniques : analyse et expression des besoins

### 1. Acteurs

L'utilisateur.

On peut à la rigueur ajouter des acteurs secondaires : serveur entrant, serveur sortant, imprimante.

### 2. Cas d'utilisation

Le diagramme de cas d'utilisation est représenté figure 16.

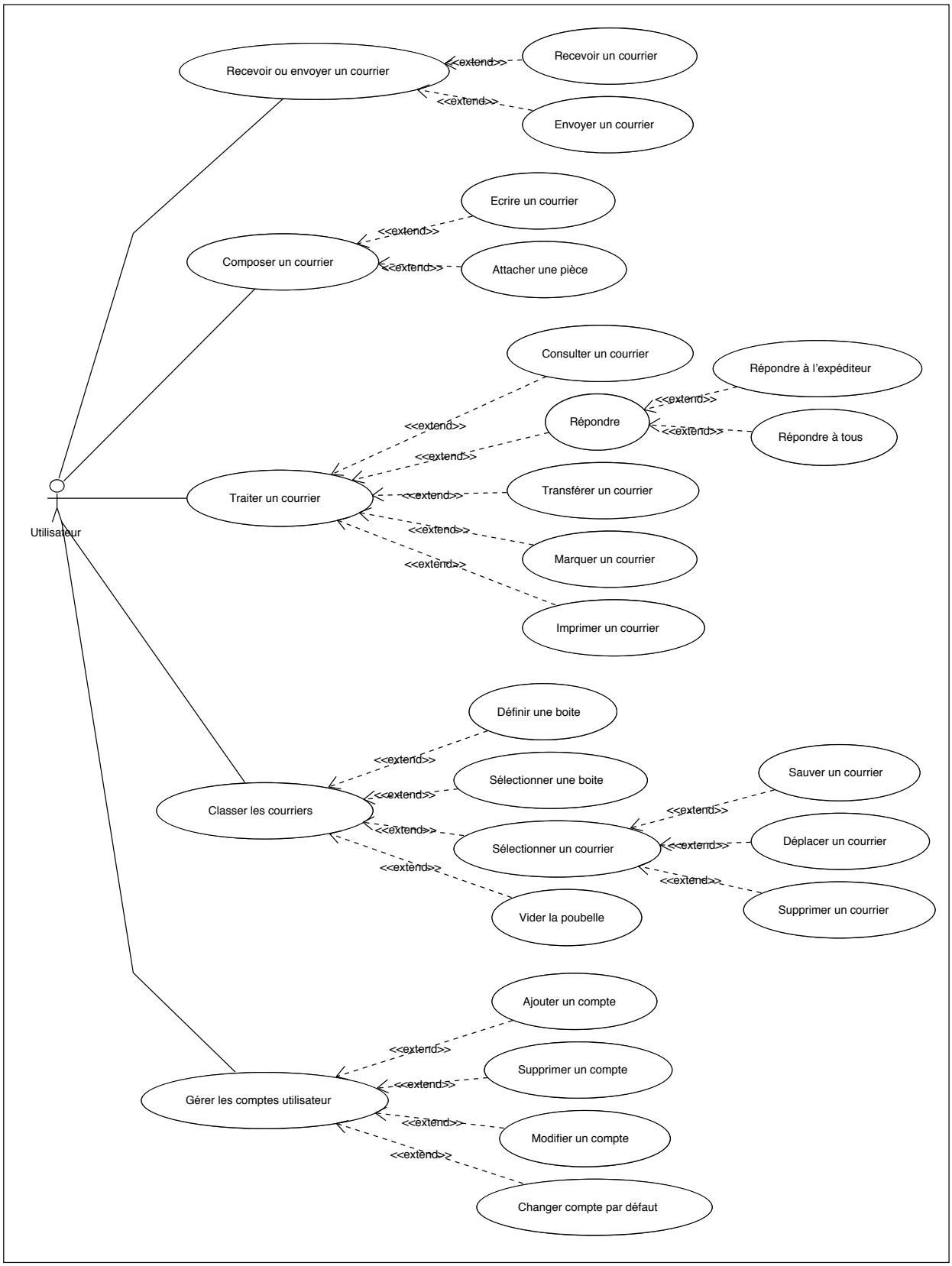



FIGURE 16 – Diagramme de cas d'utilisation

**3. Diagrammes de séquence système**

À compléter...

## Exercice 17. Courriers électroniques : diagramme de classes d'analyse

Le diagramme de classes est représenté figure 17.

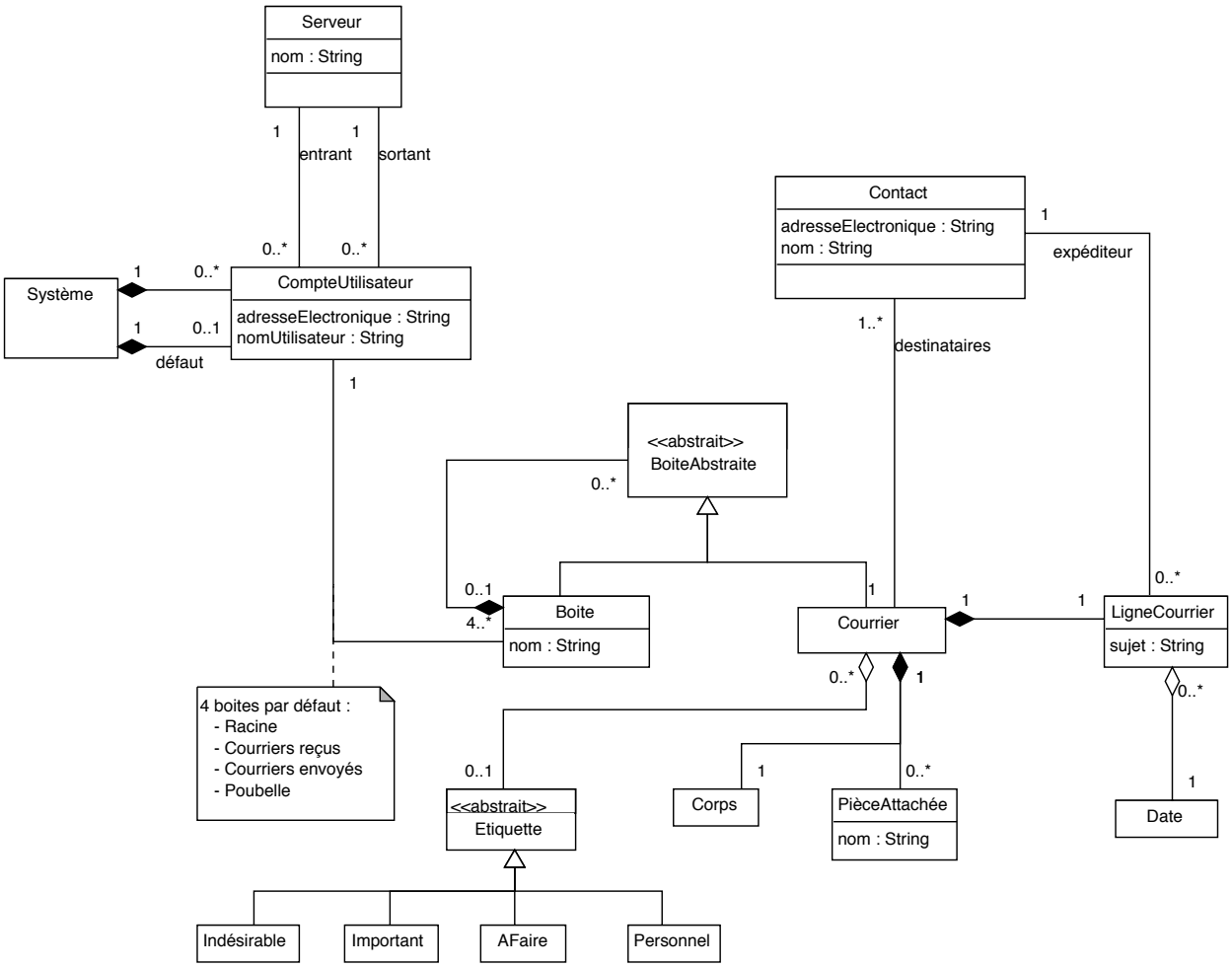



FIGURE 17 – Diagramme de classes

On suppose ici que les boîtes par défaut sont associées à un compte utilisateur (le cahier des charges n'est pas clair sur ce point).

## Exercice 18. Courriers électroniques : architecture

On propose d'utiliser une architecture en couches, avec les couches suivantes :

- *Couche Présentation.* La couche Présentation gère l'affichage de l'application. On y trouve en particulier les différentes fenêtres, avec leur composants (menus, boutons...).
- *Couche Application.* La couche Application traite les différentes fonctionnalités du système.
- *Couche Domaine.* On trouve dans la couche Domaine la plupart des classes de la modélisation objet du domaine, structurées en différents paquetages : BoitesEtCourriers, Comptes.
- *Couche Infrastructure* La couche Infrastructure contient les éléments de bas niveau : stockage des courriers, lien avec l'imprimante, liens avec les serveurs entrants et sortants. Pour le stockage des courriers, le cahier des charges ne présise pas comment les courriers doivent être stockés. On peut imaginer soit un stockage par fichiers et répertoires pour les boîtes, soit une base de données.

La figure 18 représente l'architecture du système.

## Exercice 20. Architecture MVC

### 1. Diagramme de classes du système

La figure 19 montre un diagramme de classes pour le système.

On a utilisé des flèches pour les associations, car on ne peut naviguer le long de ces associations que dans un seul sens. Par exemple, à partir de la classe `Anemometre`, on peut accéder au capteur (attribut `capteur`), mais pas l'inverse.

### 2. Décomposition MVC

La figure 20 montre un diagramme de classes qui décompose `Anemometre` en trois classes : `AnemometreModele`, `AnemometreVue` et `AnemometreControleur`.

Les flèches entre ces trois classes respectent ce qui est demandé dans l'énoncé : le contrôleur peut accéder à la vue et au modèle, mais pas l'inverse. Il n'y a pas de relation directe entre la vue et le modèle.

Dans la classe `AnemometreModele`, on met les données, en l'occurrence la vitesse du vent.

La classe `AnemometreVue` est une sous classe de `JFrame` (fenêtre graphique), et est reliée aux classes `JButton` (bouton de changement d'unité) et `JLabel` (affichage de la vitesse).

La classe `AnemometreControleur` comporte le *timer*, qui permet d'appeler la mise à jour de la vitesse toutes les secondes, et l'unité de vitesse.

Enfin la classe `AnemometreModele` accède à la classe `CapteurVent`, ce qui lui permet de mettre à jour son attribut `vitesse`.

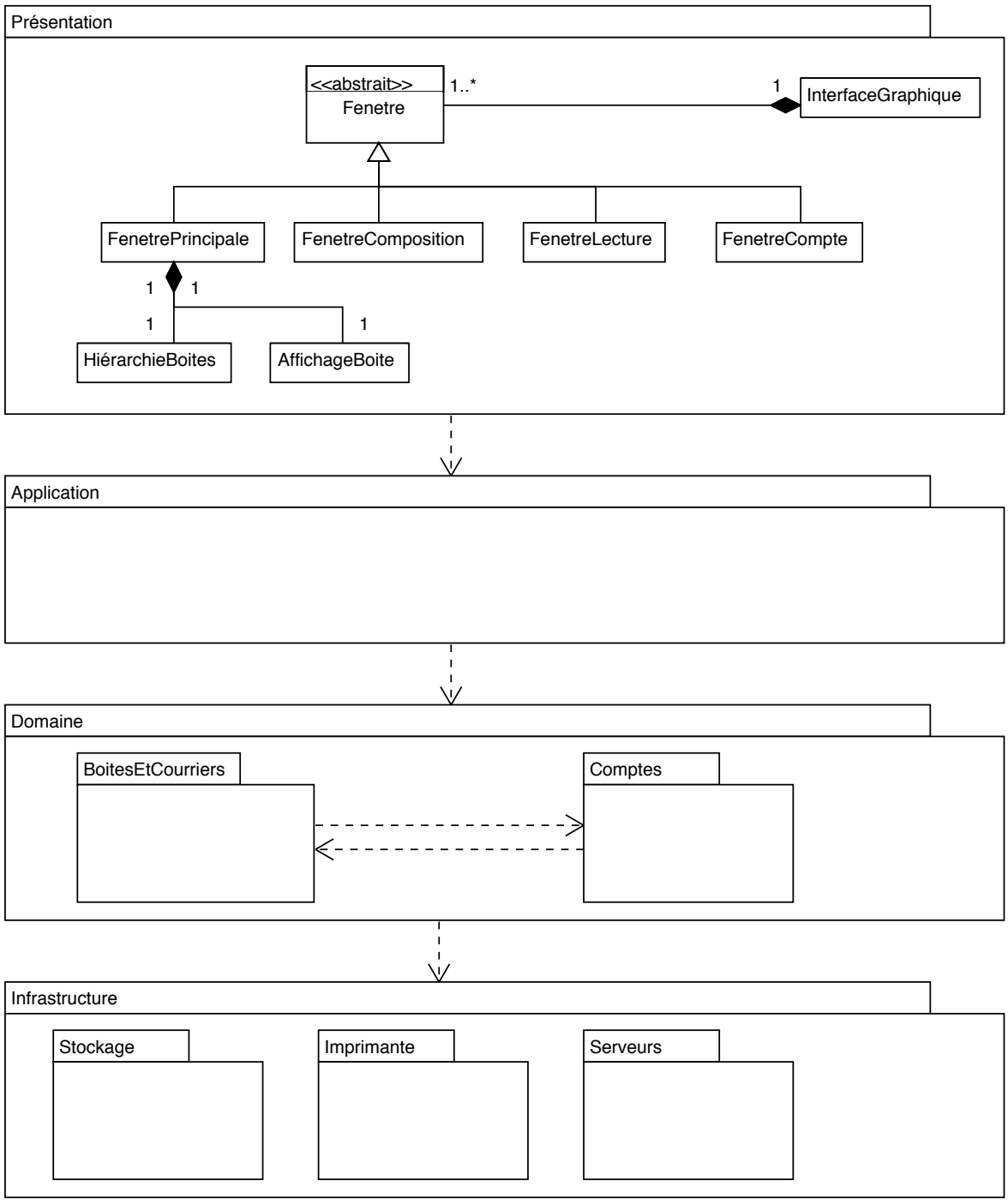



FIGURE 18 – Architecture du système

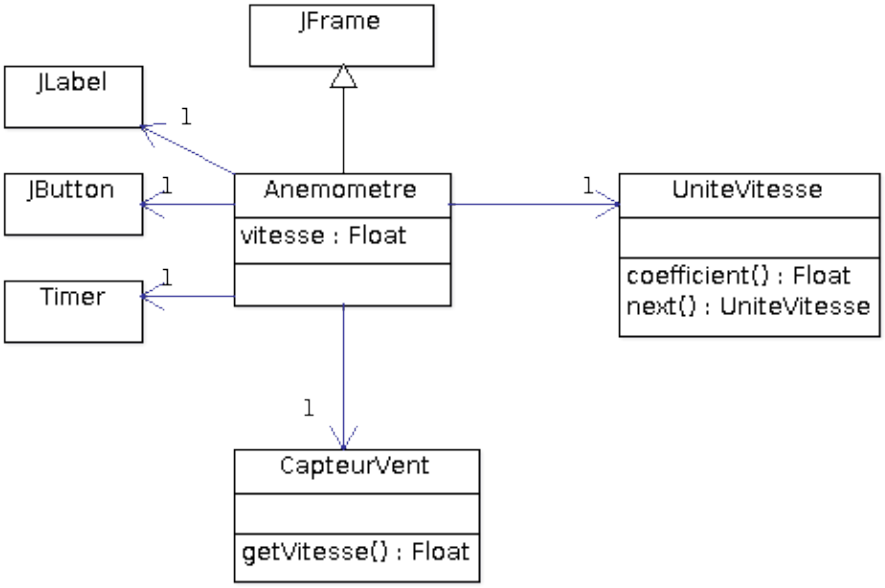



FIGURE 19 – Diagramme de classes du système

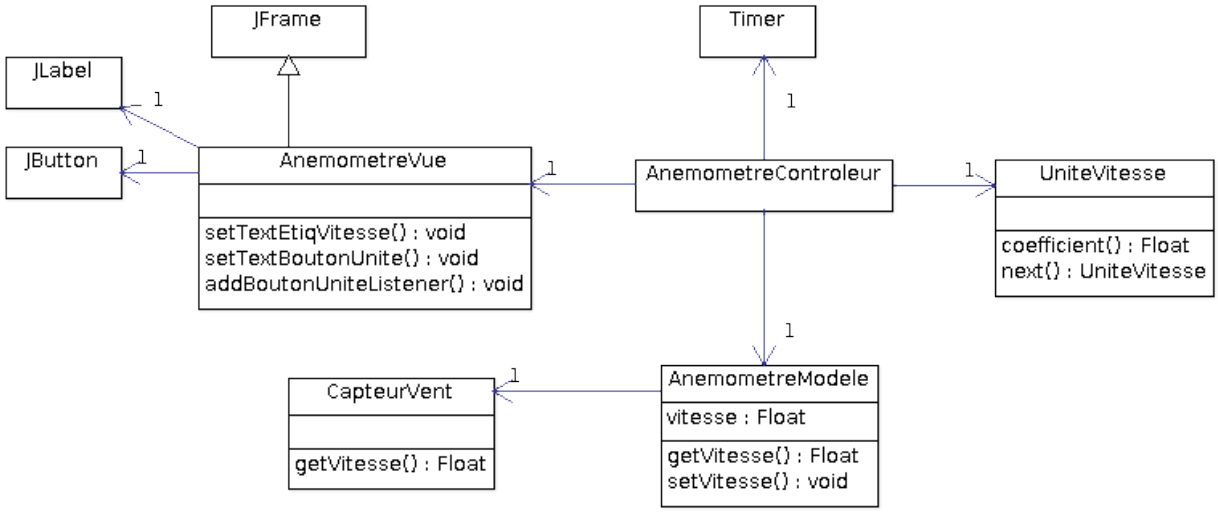



FIGURE 20 – Diagramme de classes avec décomposition MVC

**Fichier AnemometreModele.java**

```
/**
 * Classe modèle d'anémomètre.
 */

public class AnemometreModele {

   // Vitesse courante en km/h
   private float vitesse;

   // Le capteur de vent
   private CapteurVent leCapteur;

   /**
    * Constructeur.
    */
   public AnemometreModele() {
      leCapteur = new CapteurVent();
      setVitesse();
   }

   /**
    * La vitesse courante, en km/h.
    */
   public float getVitesse() {
      return vitesse;
   }

   /**
    * Met à jour la vitesse en utilisant le capteur.
    */
   public void setVitesse() {
       vitesse = leCapteur.getVitesse();
   }
}
```

**Fichier AnemometreVue.java**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

/**
 * Vue d'anémomètre.
 */
public class AnemometreVue extends JFrame {

   // Eléments de l'interface
   // Affichage de la vitesse
   private JLabel etiqVitesse = new JLabel();
   // Bouton permettant de changer d'unité de vitesse
   private JButton boutonUnite = new JButton();
```

```
    /**
     * Contructeur.
     */
    public AnemometreVue() {
        // Mise en place de l'interface graphique
        JPanel panneau = new JPanel();
        setSize(500, 100);
        panneau.add(etiqVitesse);
        panneau.add(boutonUnite);
        this.add(panneau);
        setVisible(true);
    }

    /**
     * Modifie la vitesse affichée.
     */
    public void setTextEtiqVitesse(String s) {
        etiqVitesse.setText(s);
    }

    /**
     * Modifie le texte du bouton d'unité.
     */
    public void setTextBoutonUnite(String s) {
        boutonUnite.setText(s);
    }

    /**
     * Ajoute le listener a sur le bouton d'unité.
     */
    public void addBoutonUniteListener(ActionListener a) {
        boutonUnite.addActionListener(a);
    }
}
```

**Fichier AnemometreControleur.java**

```
import javax.swing.Timer;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Contrôleur d'anémomètre.
 */
public class AnemometreControleur {

    // Le modèle associé à ce contrôleur
    private AnemometreModele leModele;

    // La vue associée à ce contrôleur
    private AnemometreVue laVue;

    // L'unité de vitesse courante
```

```
private UniteVitesse unite = UniteVitesse.KMH;

// Un timer pour interroger le capteur de vitesse de vent à intervalles
// spécifiés
private Timer timer;

/**
 * Constructeur.
 */
public AnemometreControleur
      (AnemometreModele modele, AnemometreVue vue) {
   this.leModele = modele;
   this.laVue = vue;

   // Initialisation du texte du bouton d'unité
   laVue.setTextBoutonUnite(unite.toString());

   // Ce qui se passe lorsqu'on clique sur le bouton d'unités
   laVue.addBoutonUniteListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changerUnite();
      }
   });

   // Récupération de la vitesse toutes les secondes
   timer = new Timer(1000, new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         // On met à jour la vitesse sur l'interface
         miseAJourVitesse();
      }
   });
   timer.start();
}

/**
 * Changement d'unité pour l'affichage de la vitesse.
 */
private void changerUnite() {
   unite = unite.next();
   miseAJourUnite();
   miseAJourVitesse();
}

/**
 * Met à jour l'affichage de la vitesse avec la vitesse courante.
 */
private void miseAJourVitesse() {
   leModele.setVitesse();
   float v = leModele.getVitesse()/unite.coefficient();
   laVue.setTextEtiqVitesse(Float.toString(v));
}

/**
```

```
     * Met à jour l'affichage du bouton avec l'unité de vitesse courante.
     */
    private void miseAJourUnite() {
        laVue.setTextBoutonUnite(unite.toString());
    }
}
```

### 3. Deux fenêtres graphiques

```
/**
 * Classe de test d'anémomètre.
 */

public class AnemometreTest2 {
    public static void main(String[] args) {
        // Mise en place d'un anémomètre avec son modèle, sa vue
        // et son contrôleur
        AnemometreModele modele = new AnemometreModele();
        AnemometreVue vue1 = new AnemometreVue();
        AnemometreControleur ctrl1 =
            new AnemometreControleur(modele, vue1);

        // On peut facilement créer une deuxième vue sur le même modèle et
        // donc le même capteur
        AnemometreVue vue2 = new AnemometreVue();
        AnemometreControleur ctrl2 =
            new AnemometreControleur(modele, vue2);
    }
}
```

## Exercice 21. Patron Adaptateur

### 1. Diagramme de classes

On a au départ les trois classes `Four`, `Radiateur` et `Television` avec les méthodes spécifiques pour allumer et éteindre chaque appareil.

On définit une classe adaptateur pour chaque type d'appareil : `FourAdapt`, `RadiateurAdapt` et `TelevisionAdapt`. On a donc une classe `FourAdapt`, qui peut accéder à la classe `Four`, et de même pour `RadiateurAdapt` et `TelevisionAdapt`.

L'important ici est de réutiliser les classes `Four`, `Radiateur` et `Télévision` **sans les modifier**.

Chaque classe adaptateur implémente l'interface `Appareil` avec ses deux méthodes `on()` et `off()`.

On a enfin une classe `Multiprise`, qui est composée d'une liste d'appareils, et qui implémente également la classe `Appareil`. On remarque ici qu'en plus du patron Adaptateur, on utilise le patron Composite.

Le diagramme de classes obtenu se trouve figure 21.

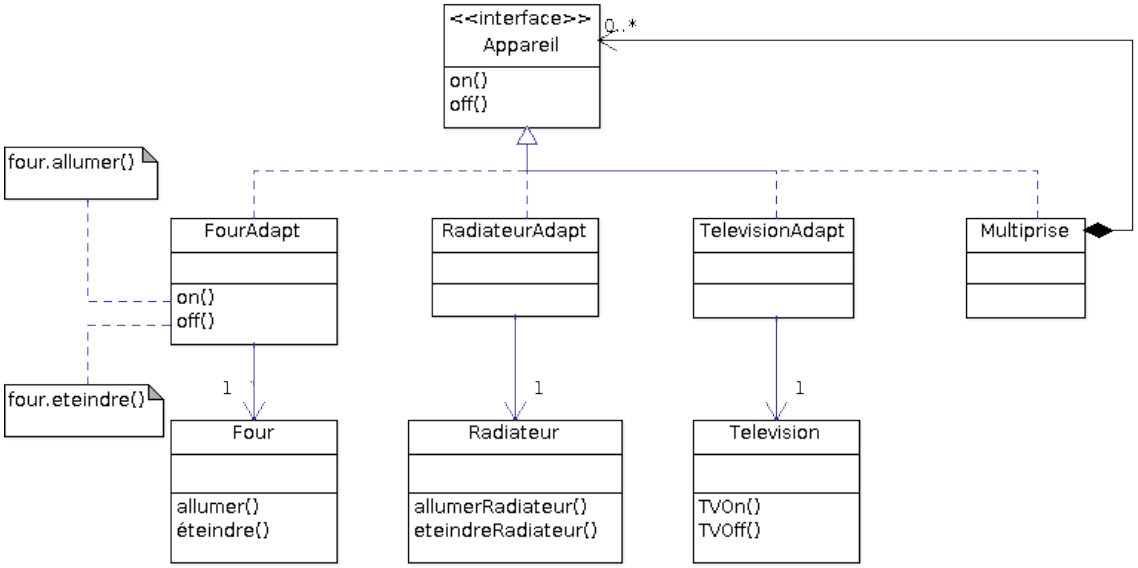



FIGURE 21 – Diagramme de classes pour les multiprises

### 2. Code Java

```
// Classe Four
public class Four {
    public void allumer() {
        System.out.println("Four allumé");
    }
    public void eteindre() {
        System.out.println("Four éteint");
    }
}

// Interface Appareil
interface Appareil {
    void on();
    void off();
}

// Classe d'adaptateur pour le four
public class FourAdapt implements Appareil {

    public Four four;

    FourAdapt(Four four) {
        this.four = four;
    }

    public void on() {
        four.allumer();
```

```
    }

    public void off() {
      four.eteindre();
    }
}

// Classe Multiprise
import java.util.List;
import java.util.ArrayList;

public class Multiprise implements Appareil {

    public List<Appareil> appareils;

    public Multiprise() {
        appareils = new ArrayList<Appareil>();
    }

    public void on() {
        for (Appareil appareil : appareils) {
            appareil.on();
        }
    }

    public void off() {
        for (Appareil appareil : appareils) {
            appareil.off();
        }
    }
}

// Classe de test
public class TestMultiprise {

    public static void main(String[] args) {
        Four four = new Four();
        Radiateur radiateur = new Radiateur();
        Television tv = new Television();
        FourAdapt fourA = new FourAdapt(four);
        RadiateurAdapt radiateurA = new RadiateurAdapt(radiateur);
        TelevisionAdapt tvA = new TelevisionAdapt(tv);
        Multiprise multiprise = new Multiprise();
        multiprise.appareils.add(fourA);
        multiprise.appareils.add(radiateurA);
        multiprise.appareils.add(tvA);
        multiprise.on();
        multiprise.off();
    }
}
```

## Exercice 21bis. Patron État : platine cassettes

Note : l'énoncé de cet exercice n'est pas dans le poly.

### Énoncé

On considère une platine cassettes dont le comportement est modélisé par le diagramme d'états-transitions figure 22.

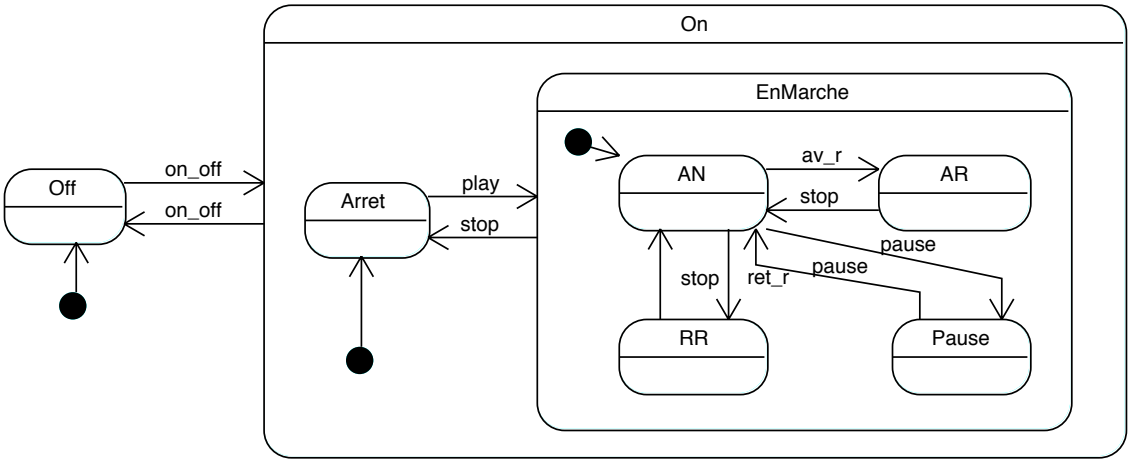



FIGURE 22 – Diagramme d'états-transitions de la platine cassettes

On définit en Java une classe `Etat` :

```
abstract class Etat {
   static Etat initial() { ... }
   Etat on_off() { ... }
   Etat play() { ... }
   Etat stop() { ... }
   Etat av_r() { ... }
   Etat ret_r() { ... }
   Etat pause() { ... }
}
```

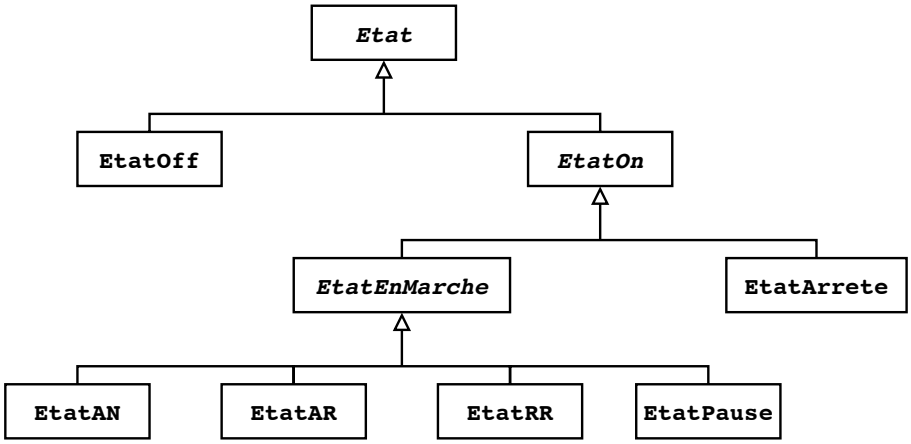
— La méthode `initial()` renvoie, pour un état élémentaire, une instance de cet état, et pour un état composite, une instance de l'état initial du sous-automate correspondant.
— À chaque événement qui peut être reçu par la platine est associée une méthode qui retourne l'état de la platine après les actions correspondantes.

### Questions

1. Définir une hiérarchie de classes pour les différents états de la platine.
2. Compléter le code de la classe `Etat`, et donner le code des différentes classes définies en 1.

### Correction

#### 1. Hiérarchie de classes pour les états




Il faut ici créer une hiérarchie de classes qui respecte la hiérarchie des états de l'automate.

Les classes qui correspondent à des états composites sont des classes abstraites : `Etat`, `EtatOn`, `EtatEnMarche`.

Par exemple, lorsque la platine est en marche, elle est soit dans l'état AN, soit dans l'état AR, soit dans l'état RR, soit dans l'état Pause.

#### 2. Programme Java

Les classes concrètes `EtatOff`, `EtatArrete`, `EtatAN`, `EtatAR`, `EtatRR` et `EtatPause` sont implémentées par des singletons. Dans chacune de ces classes, la méthode `initial()` renvoie l'instance unique de chaque classe.

On complète la classe `Etat` :

- l'état initial est l'état Off.
- pour chaque méthode, on retourne cet état. En effet, par défaut, on reste dans le même état. On spécifiera dans les sous-classes les changements d'états qui correspondent à des transitions de l'automate.

```
abstract class Etat {
   static Etat initial() {
      return EtatOff.initial() ;
   }

   Etat on_off() {
      return this ;
   }

   Etat play() {
      return this ;
   }
```

```
    Etat stop() {
        return this ;
    }

    Etat av_r() {
        return this ;
    }

    Etat ret_r() {
        return this ;
    }

    Etat pause() {
        return this ;
    }
}
```

Classe `EtatOff`, correspondant à l'état Off :

```
class EtatOff extends Etat {
    private static EtatOff state = new EtatOff() ;
    private EtatOff() { }

    static Etat initial() {
        return state ;
    }

    Etat on_off() {
        return EtatOn.initial() ;
    }
}
```

Classe `EtatArrete`, correspondant à l'état Arrêté :

```
class EtatArrete extends EtatOn {
    private static EtatArrete state = new EtatArrete() ;
    private EtatArrete() { }

    static Etat initial() {
        return state ;
    }

    Etat play() {
        System.out.println("play : Arrete -> EnMarche") ;
        return EtatEnMarche.initial() ;
    }
}
```

Classe `EtatEnMarche`, correspondant à l'état EnMarche :

```
abstract class EtatEnMarche extends EtatOn {
```

```
   static Etat initial() {
      return EtatAN.initial() ;
   }

   Etat stop() {
      System.out.println("stop : EnMarche -> Arrete") ;
      return EtatArrete.initial() ;
   }
}
```

Classe `EtatAR`, correspondant à l'état AR :

```
class EtatAR extends EtatEnMarche {
   private static EtatAR state = new EtatAR() ;
   private EtatAR() { }

   static Etat initial() {
      return state ;
   }

   Etat stop() {
      System.out.println("stop : AR -> AN") ;
      return EtatAN.initial() ;
   }
}
```

Classe `Platine` :

```
class Platine {
   Etat etat ;
   Platine() {
      etat = Etat.initial() ;
   }

   void on_off() {
      etat = etat.on_off() ;
   }

   void play() {
      etat = etat.play() ;
   }

   void stop() {
      etat = etat.stop() ;
   }

   void av_r() {
      etat = etat.av_r() ;
   }

   void ret_r() {
      etat = etat.ret_r() ;
   }
```

```
    void pause() {
        etat = etat.pause() ;
    }
}
```

Classe de test :

```
class Test {
    public static void main(String[] args) {
        Platine platine = new Platine() ;
        platine.on_off() ;
        platine.play() ;
        platine.av_r() ;
        platine.stop() ;
        platine.on_off() ;
        platine.play() ; // Ne fait rien
    }
}
```

*Remarques*

1. Soit un état $e_2$, correspondant à une classe `EtatE2`, contenu dans un état $e_1$, correspondant à une classe `EtatE1`. `EtatE2` est une sous-classe de `EtatE1`. Soit deux transitions étiquetées par une méthode `m` partant respectivement de $e_1$ et $e_2$. C'est la méthode de `EtatE2` qui sera appliquée, donc la transition correspondant à l'état $e_2$ (le plus imbriqué) qui sera effectuée.

2. Si aucune transition ne peut être effectuée, on hérite de la méthode de la classe `Etat` qui ne fait rien (et reste dans le même état).

3. Si on oublie le « `static` », on obtient un incident de segmentation (car le constructeur boucle).

```
class EtatOff extends Etat {
    Etat state = new EtatOff() ; // Oubli du static
    // Le constructeur boucle, puis incident de segmentation
    [...]
    }
}
```

## Exercice 22. Patron Observateur

### Diagramme de classes

cf. figure 23.

La classe `Sujet` et l'interface `Observateur` sont identiques à celles vues en cours.

La classe `Pourcentage` contient les données du système, à savoir les trois valeurs des pourcentages (attributs `a`, `b` et `c`).

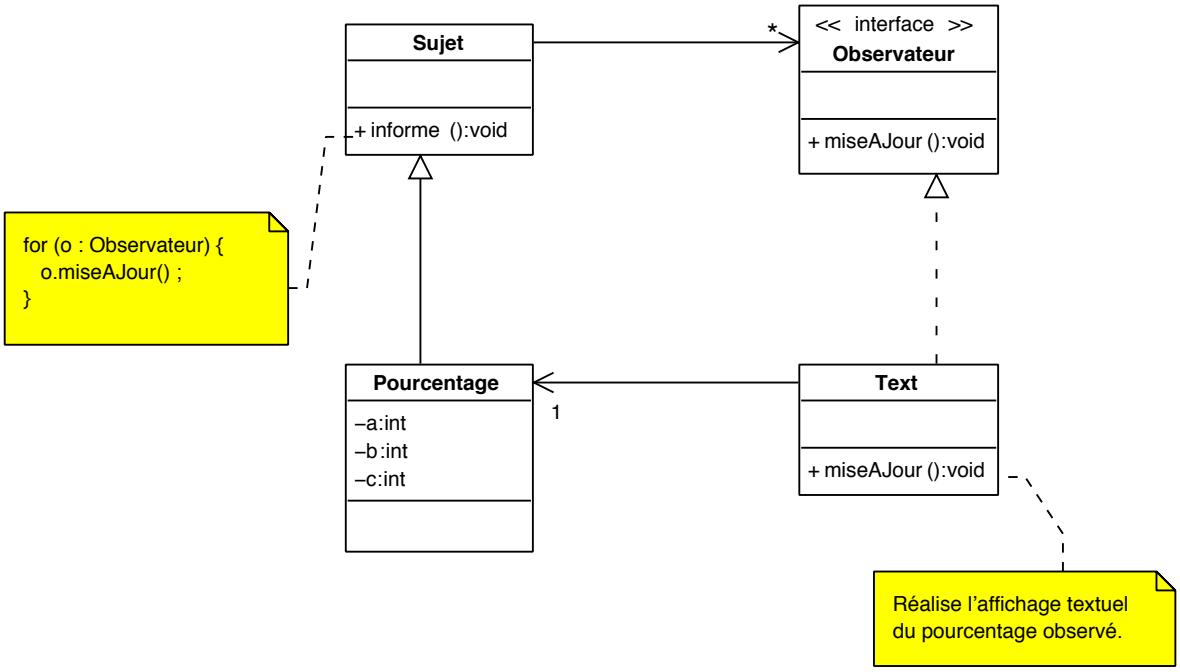



FIGURE 23 – Diagramme de classes

La classe `Text` permet l'affichage des trois valeurs $a$, $b$ et $c$. Elle implémente la classe `Observateur`, la méthode `miseAJour()` réalise cet affichage. La classe `Text` doit pour cela accéder à la classe `Pourcentage`, pour récupérer les trois valeurs $a$, $b$ et $c$.

Les attributs de la classe `Pourcentage` doivent être privés, de façon à ne pas pouvoir être modifiés sans contrôle. C'est la méthode `set` qui permet de modifier les attributs `a`, `b` et `c`.

**La méthode `set` de la classe `Pourcentage` appelle la méthode `informe()`, héritée de la classe `Sujet`, de façon à ce que tous les observateurs de la classe `Pourcentage` soient avertis de cette modification.**

### Programme Java

#### Classe `Sujet`

```
import java.util.HashSet ;
import java.util.Set ;
import java.util.Iterator ;

/**
 * Classe abstraite des sujets.
 */

abstract public class Sujet {
```

```
    // L'ensemble des observateurs
    private Set<Observateur> lesObservateurs ;

    /**
     * Constructeur.
     */
    public Sujet() {
        lesObservateurs = new HashSet<Observateur>() ;
    }

    /**
     * Attache l'observateur o à ce sujet.
     */
    public void attache(Observateur o) {
        lesObservateurs.add(o) ;
    }

    /**
     * Détache l'observateur o de ce sujet.
     */
    public void detache(Observateur o) {
        lesObservateurs.remove(o) ;
    }

    /**
     * Informe tous les observateurs de ce sujet que son état
     * a été modifié.
     * Méthode "protected" car elle ne doit, a priori, être appelée que
     * dans les classes qui héritent de Sujet.
     */
    protected void informe() {
        // Mise à jour de tous les observateurs
        for (Observateur o : lesObservateurs) {
            o.miseAJour() ;
        }
    }
}
```

**Interface** `Observateur`

```
/**
 * Interface des observateurs.
 */

public interface Observateur {

    /**
     * Mise à jour du Sujet auquel correspond cet observateur.
     */
    void miseAJour() ;
}
```

**Interface** `Pourcentage`

```
/**
 * Classe Pourcentage.
 * Cette classe fait partie du "modèle".
 *
 * L'état de cette classe est constitué des attributs a, b et c.
 *
 * On déclare les attributs a, b et c comme privés, afin d'interdire
 * leur modification autrement qu'en utilisant la méthode set.
 * Cela permet de forcer l'appel à informe à chaque modification
 * de l'état.
 */

public class Pourcentage extends Sujet {

   private int a ;
   private int b ;
   private int c ;

   /**
    * Constructeur de pourcentage.
    * Précondition : a >= 0, b >= 0, a + b <= 100.
    */
   public Pourcentage(int a, int b) {
      this.a = a ;
      this.b = b ;
      this.c = 100 - a - b ;
   }

   /**
    * Changement des valeurs des pourcentages.
    * Précondition : a >= 0, b >= 0, a + b <= 100.
    */
   public void set(int a, int b) {
      this.a = a ;
      this.b = b ;
      this.c = 100 - a - b ;
      // Cet objet informe ses observateurs de son changement d'état.
      informe() ;
   }

   /**
    * Retourne la valeur de a.
    */
   public int getA() {
      return a ;
   }

   /**
    * Retourne la valeur de b.
    */
   public int getB() {
      return b ;
   }
```

```
   /**
    * Retourne la valeur de c.
    */
   public int getC() {
      return c ;
   }
}
```

**Classe** `Text`

```
/**
 * Classe permettant l'affichage textuel d'un pourcentage.
 * Cette classe fait partie de la "vue".
 */

class Text implements Observateur {

   // Le pourcentage auquel est associé cet affichage textuel.
   Pourcentage p ;

   /**
    * Constructeur d'affichage textuel.
    */
   Text(Pourcentage p) {
      this.p = p ;
      p.attache(this) ;
      miseAJour() ;
   }

   /**
    * Mise à jour de cet affichage textuel.
    */
   public void miseAJour() {
      System.out.println(
            "␣␣␣a␣=␣" + p.getA() +
         "%\n␣␣␣b␣=␣" + p.getB() +
         "%\n␣␣␣c␣=␣" + p.getC() + "%") ;
   }
}
```

**Classe** `Test`

```
/**
 * Classe de test.
 */

class Test {
   public static void main(String[] args) {
      System.out.println("--␣Creation␣du␣pourcentage") ;
      Pourcentage p = new Pourcentage(20, 30) ;
      System.out.println("--␣Creation␣de␣l'affichage␣textuel") ;
      Text t = new Text(p) ;
```

```
        System.out.println("-- Modification de l'etat du pourcentage") ;
        p.set(0, 0) ;
        System.out.println("-- Creation d'un deuxieme affichage textuel") ;
        Text t2 = new Text(p) ;
        System.out.println("-- Modification de l'etat du pourcentage") ;
        p.set(10, 20) ;
    }
}
```

## Exercice 23. Patron Interprète

### 1. Diagramme de classes

On a une classe abstraite pour chaque non terminal de la grammaire, donc les classes `Exp`, `ExpBin` et `ExpUn`. Dans la grammaire, Exp dérive vers ExpBin et ExpUn, donc les classes `ExpBin` et `ExpUn` sont des sous-classes de `Exp`.

Pour les règles Exp → num et Exp → var, on introduit les classes concrètes `Num` et `Var`, qui sont des sous-classes de `Exp`. La classe `Num` contient un attribut `val` qui permet de stocker la valeur numérique correspondant à Num. La classe `Var` contient un attribut `nom`, qui permet de stocker le nom de la variable.

Pour les règles ExpBin → Exp + Exp, ExpBin → Exp - Exp, ExpBin → Exp * Exp, on introduit les trois classes `ExpBinPlus`, `ExpBinMoins` et `ExpBinMult`. Ces trois classes doivent contenir deux attributs `gauche` et `droit` de type `Exp`. Ces deux attributs sont factorisés dans la super classe `ExpBin`.

Un environnement permet d'associer à une variable sa valeur. On définit pour cela une classe `Env`, qui contient un attribut de type `Map` et deux méthodes `associe` et `valeur` qui permettent respectivement d'associer une valeur à une variable et de récupérer une valeur associée à une variable.

La figure 24 montre le diagramme de classe.

### 2. Diagramme d'objets

La figure 25 montre le diagramme d'objets.

### 3. Programme Java

On code l'environnement à l'aide l'interface `Map` et de la classe `Hashtable`.

```
import java.util.Map ;
import java.util.Hashtable ;

/**
 * Classe des exceptions levées lorsqu'on essaie de lire une valeur
 * indéfinie dans un environnement.
 */
```

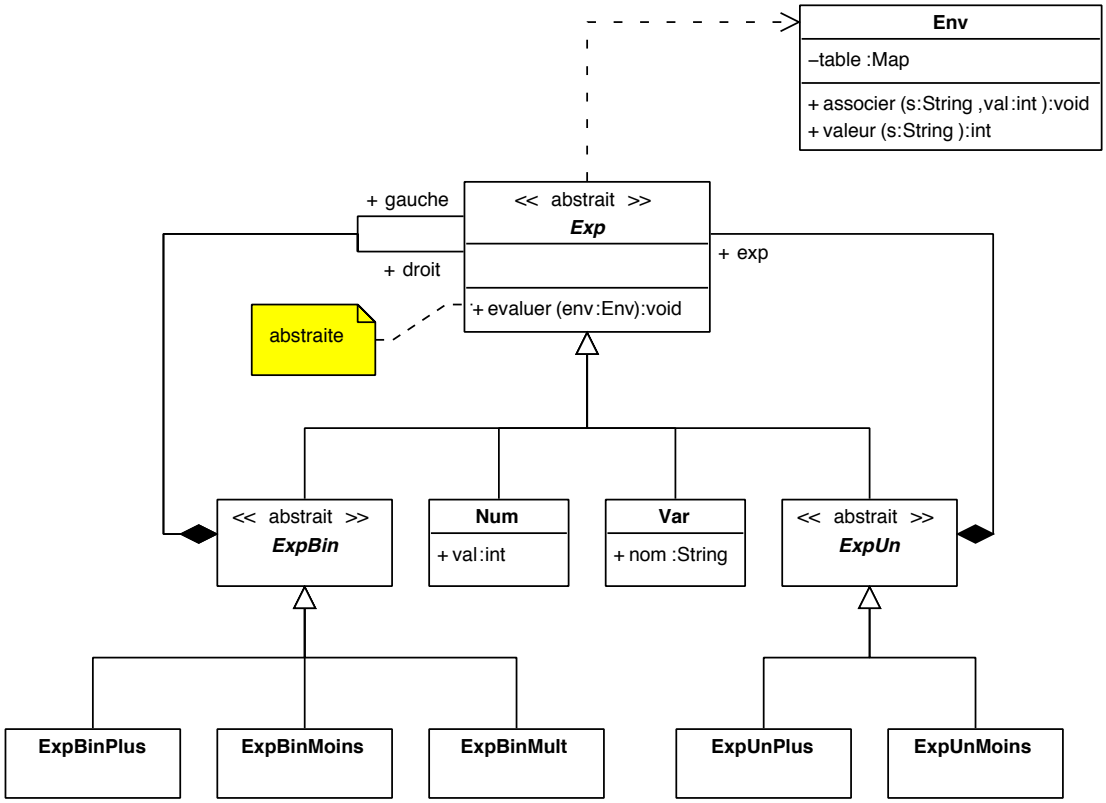



FIGURE 24 – Diagramme de classes

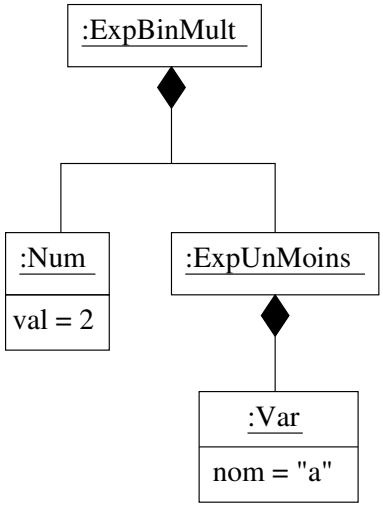



FIGURE 25 – Diagramme d'objets

```
class ValeurIndefinie extends RuntimeException { }

/**
 * Classe des environnements.
 * Un environnement associe à des chaînes de caractères des entiers.
 */

class Env {

   // Un environnement est codé à l'aide d'une fonction ("map"),
   // qui associe des entiers à des chaînes de caractères.
   private Map<String, Integer> table ;

   /**
    * Constructeur.
    */
   public Env() {
      table = new Hashtable<String, Integer>() ;
   }

   /**
    * Associe la valeur i à la chaine s.
    */
   public void associe(String s, int i) {
      table.put(s, i) ;
      // Autoboxing (conversion int -> Integer implicite)
   }

   /**
    * Récupère la valeur associée à la chaine s.
    * Lève l'exception ValeurIndefinie si s n'est associée à aucune
    * valeur dans cet environnement.
    */
   public int valeur(String s) {
      if (table.containsKey(s)) {
         return table.get(s) ;
            // Autoboxing (conversion Integer -> int implicite)
      } else {
         throw new ValeurIndefinie() ;
      }
   }
}

/**
 * Expressions.
 */

abstract class Exp {

   /**
    * Evaluation de cette expression dans l'environnement env.
    * Lève l'exception ValeurIndefinie si une variable apparaissant dans
    * cette expression n'a pas de valeur dans env.
```

```
     */
    abstract public int eval(Env env) ;
}

/**
 * Expressions binaires.
 */

abstract class ExpBin extends Exp {
    // Sous-expression gauche
    Exp gauche ;

    // Sous-expression droite
    Exp droit ;

    /**
     * Constructeur.
     */
    public ExpBin(Exp g, Exp d) {
        gauche = g ;
        droit = d ;
    }
}

/**
 * Noeud plus binaire.
 */

class ExpBinPlus extends ExpBin {

    /**
     * Constructeur.
     */
    public ExpBinPlus(Exp g, Exp d) {
        super(g, d) ;
    }

    /**
     * Evaluation de cette expression dans l'environnement env.
     */
    public int eval(Env env) {
        return gauche.eval(env) + droit.eval(env) ;
    }
}

/**
 * Noeud moins binaire.
 */

class ExpBinMoins extends ExpBin {

    /**
     * Constructeur.
```

```
     */
    public ExpBinMoins(Exp g, Exp d) {
        super(g, d) ;
    }

    /**
     * Evaluation de cette expression dans l'environnement env.
     */
    public int eval(Env env) {
        return gauche.eval(env) - droit.eval(env) ;
    }
}

/**
 * Noeud mult binaire.
 */

class ExpBinMult extends ExpBin {

    /**
     * Constructeur.
     */
    public ExpBinMult(Exp g, Exp d) {
        super(g, d) ;
    }

    /**
     * Evaluation de cette expression dans l'environnement env.
     */
    public int eval(Env env) {
        return gauche.eval(env) * droit.eval(env) ;
    }
}

/**
 * Expressions unaires.
 */

abstract class ExpUn extends Exp {

    // La sous-expression de cette expression unaire
    Exp exp ;

    /**
     * Constructeur.
     */
    public ExpUn(Exp e) {
        exp = e ;
    }
}

/**
 * Noeud plus unaire.
```

```
 */

class ExpUnPlus extends ExpUn {

   /**
    * Constructeur.
    */
   public ExpUnPlus(Exp e) {
      super(e) ;
   }

   /**
    * Evaluation de cette expression dans l'environnement env.
    */
   public int eval(Env env) {
      return exp.eval(env) ;
   }
}

/**
 * Noeud moins unaire.
 */

class ExpUnMoins extends ExpUn {

   /**
    * Constructeur.
    */
   public ExpUnMoins(Exp e) {
      super(e) ;
   }

   /**
    * Evaluation de cette expression dans l'environnement env.
    */
   public int eval(Env env) {
      return -exp.eval(env) ;
   }
}

/**
 * Expressions constantes entières.
 */

class Num extends Exp {

   // Valeur de cette constante entière.
   int val ;

   /**
    * Constructeur.
    */
   public Num(int val) {
```

```
      this.val = val ;
   }

   /**
    * Evaluation de cette expression dans l'environnement env.
    */
   public int eval(Env env) {
      return val ;
   }
}

/**
 * Expressions formées d'une variable.
 */

class Var extends Exp {

   // Nom de cette variable
   String nom ;

   /**
    * Constructeur.
    */
   public Var(String nom) {
      this.nom = nom ;
   }

   /**
    * Evaluation de cette expression dans l'environnement env.
    */
   public int eval(Env env) {
      return env.valeur(nom) ;
   }
}

/**
 * Classe permettant de tester les expressions.
 */

class TestExp {
   static void ok(boolean b) {
      if (b) {
         System.out.print("ok␣") ;
      } else {
         System.out.print("erreur␣") ;
      }
   }

   public static void main(String[] args) {
      Env env = new Env() ;
      env.associe("a", 1) ;
      env.associe("b", 2) ;
      Exp exp = new ExpBinMult(new Num(2), new ExpUnMoins(new Var("a"))) ;
```

```
        ok(exp.eval(env) == -2) ;
        System.out.println("") ;
    }
}
```

## Exercice 24. Patron Visiteur

Dans l'exercice précédent (patron Interprète), le code de la méthode `eval` de la classe `Exp` est disséminé dans les sous-classes de `Exp`.

L'objectif du patron Visiteur est de coder cette méthode `eval` en regroupant tous ces morceaux de code dans une seule classe. Ceci se fait en trois étapes :

1. On définit une interface qu'on appelle `Visiteur` et qui contient une méthode `void visite(X e)` pour chaque sous-classe concrète `X` de la classe `Exp`.
2. On ajoute dans la classe `Exp` et dans chaque sous-classe concrète de `Exp` une méthode `void applique(Visiteur v)`. Cette méthode est abstraite dans la classe `Exp` et est implémentée dans les sous-classes avec le code suivant :

   ```
   void applique(Visiteur v) {
       v.visite(this);
   }
   ```

   On peut remarquer que ce code ne peut pas être factorisé au niveau de la classe `Exp` car la méthode `visite` appelée est différente dans chaque sous-classe (`this` est de type différent pour chaque sous-classe).
3. On code la méthode `eval` en implémentant l'interface `Visiteur` avec la classe `Evaluation`. Cette classe contient deux attributs `env` et `resultat` qui contiennent respectivement l'environnement dans lequel l'expression doit être évaluée et le résultat de l'évaluation. La méthode `eval` permet de factoriser l'application du visiteur et la récupération du résultat de l'évaluation.

### Diagramme de classes

La figure 26 montre le diagramme de classes.

### Programme Java

```
/**
 * Evaluation d'expressions arithmétiques.
 * Implémentation à l'aide du patron de conception "Visiteur".
 *
 * Remarques sur cette solution :
 *    - Cette solution utilise la généricité et l'"autoboxing" de Java 1.5.
 *    - La classe "Evaluation" contient une méthode "eval", qui permet
 *      de factoriser l'application du visiteur et la récupération du
 *      résultat.
 */
```

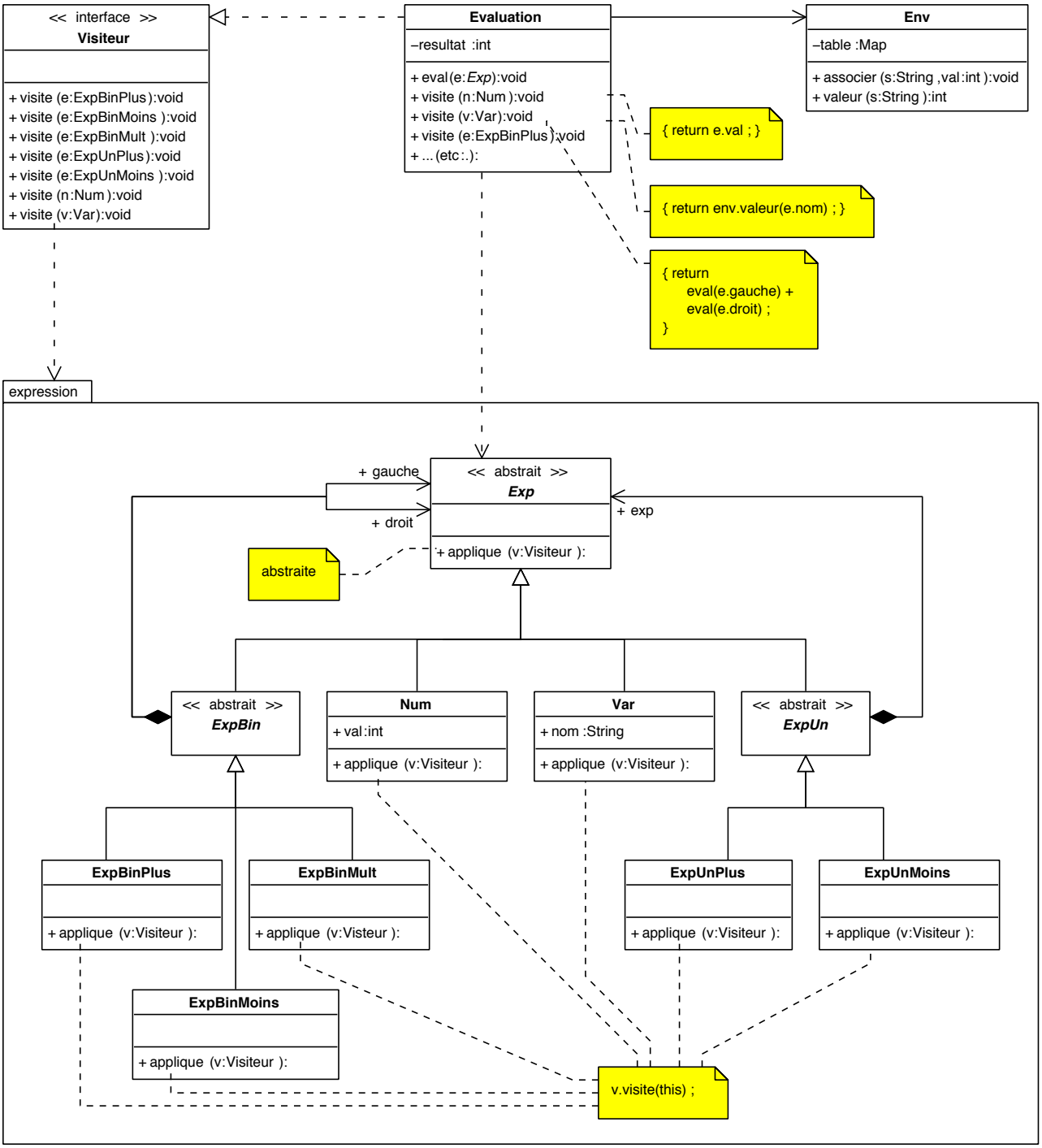



FIGURE 26 – Diagramme de classes

```
/**
 * Classe des exceptions levées lorsqu'on essaie de lire une valeur
 * indéfinie dans un environnement.
 */

class ValeurIndefinie extends RuntimeException { }

/**
 * Classe des environnements.
 */

import java.util.Map;
import java.util.Hashtable;

class Env {

   // Un environnement est codé à l'aide d'une fonction ("map"), qui
   // permet d'associer des entiers à des chaînes de caractères.
   private Map<String, Integer> table ;

   /**
    * Constructeur.
    */
   public Env() {
      table = new Hashtable<String, Integer>() ;
   }

   /**
    * Associe la valeur i à la chaine s.
    */
   public void associe(String s, int i) {
      table.put(s, i) ;
   }

   /**
    * Récupère la valeur associée à la chaine s.
    * Lève l'exception ValeurIndefinie si s n'est associée à aucune
    * valeur dans cet environnement.
    */
   public int valeur(String s) {
      if (table.containsKey(s)) {
         return table.get(s) ;
      } else {
         throw new ValeurIndefinie() ;
      }
   }
}

/**
 * Expressions arithmétiques.
 */
```

```
abstract class Exp {
   abstract public void applique(Visiteur v) ;
}

/**
 * Expressions binaires.
 */

abstract class ExpBin extends Exp {
   // Sous-expression gauche
   Exp gauche ;

   // Sous-expression droite
   Exp droit ;

   /**
    * Constructeur.
    */
   public ExpBin(Exp g, Exp d) {
      gauche = g ;
      droit = d ;
   }
}

/**
 * Noeud plus binaire.
 */

class ExpBinPlus extends ExpBin {

   /**
    * Constructeur.
    */
   public ExpBinPlus(Exp g, Exp d) {
      super(g, d) ;
   }

   public void applique(Visiteur v) {
      v.visite(this) ;
   }
}

/**
 * Noeud moins binaire.
 */
class ExpBinMoins extends ExpBin {

   /**
    * Constructeur.
    */
   public ExpBinMoins(Exp g, Exp d) {
      super(g, d) ;
   }
```

```
   public void applique(Visiteur v) {
      v.visite(this) ;
   }
}

/**
 * Noeud mult binaire.
 */

class ExpBinMult extends ExpBin {

   /**
    * Constructeur.
    */
   public ExpBinMult(Exp g, Exp d) {
      super(g, d) ;
   }

   public void applique(Visiteur v) {
      v.visite(this) ;
   }
}

/**
 * Expressions unaires.
 */

abstract class ExpUn extends Exp {

   // La sous-expression de cette expression unaire
   Exp exp ;

   /**
    * Constructeur.
    */
   public ExpUn(Exp e) {
      exp = e ;
   }
}

/**
 * Noeud plus unaire.
 */

class ExpUnPlus extends ExpUn {

   /**
    * Constructeur.
    */
   public ExpUnPlus(Exp e) {
      super(e) ;
   }
```

```
   public void applique(Visiteur v) {
      v.visite(this) ;
   }
}

/**
 * Noeud moins unaire.
 */

class ExpUnMoins extends ExpUn {

   /**
    * Constructeur.
    */
   public ExpUnMoins(Exp e) {
      super(e) ;
   }

   public void applique(Visiteur v) {
      v.visite(this) ;
   }
}

/**
 * Expressions constantes entières.
 */

class Num extends Exp {

   // Valeur de cette constante entière.
   int val ;

   /**
    * Constructeur.
    */
   public Num(int val) {
      this.val = val ;
   }

   public void applique(Visiteur v) {
      v.visite(this) ;
   }
}

/**
 * Expressions formées d'une variable.
 */

class Var extends Exp {

   // Nom de cette variable
   String nom ;
```

```
    /**
     * Constructeur.
     */
    public Var(String nom) {
        this.nom = nom ;
    }

    public void applique(Visiteur v) {
        v.visite(this) ;
    }
}


/*****************************************************************************
 * L'interface Visiteur
 *****************************************************************************/

interface Visiteur {
    void visite(ExpBinPlus e) ;
    void visite(ExpBinMoins e) ;
    void visite(ExpBinMult e) ;
    void visite(ExpUnPlus e) ;
    void visite(ExpUnMoins e) ;
    void visite(Num e) ;
    void visite(Var e) ;
}

/*****************************************************************************
 * Evaluation
 *****************************************************************************/

class Evaluation implements Visiteur {

    // L'environnement dans lequel l'expression doit être évaluée.
    Env env ;

    // Le résultat de l'évaluation
    private int resultat ;

    /**
     * Constructeur.
     */
    public Evaluation(Env e) {
        env = e ;
    }

    /**
     * Retourne l'évaluation de e dans l'environnement env.
     */
    int eval(Exp e) {
        e.applique(this) ;
        return resultat ;
```

```
    }

    /**
     * Evaluation d'un noeud plus binaire e.
     */
    public void visite(ExpBinPlus e) {
        resultat = eval(e.gauche) + eval(e.droit) ;
    }

    /**
     * Evaluation d'un noeud moins binaire e.
     */
    public void visite(ExpBinMoins e) {
        resultat = eval(e.gauche) - eval(e.droit) ;
    }

    /**
     * Evaluation d'un noeud mult e.
     */
    public void visite(ExpBinMult e) {
        resultat = eval(e.gauche) * eval(e.droit) ;
    }

    /**
     * Evaluation d'un noeud plus unaire e.
     */
    public void visite(ExpUnPlus e) {
        resultat = eval(e.exp) ;
    }

    /**
     * Evaluation d'un noeud moins unaire e.
     */
    public void visite(ExpUnMoins e) {
        resultat = -eval(e.exp) ;
    }

    /**
     * Evaluation d'un entier.
     */
    public void visite(Num e) {
        resultat = e.val ;
    }

    /**
     * Evaluation d'une variable.
     */
    public void visite(Var e) {
        resultat = env.valeur(e.nom) ;
    }
}
```

```
/**
 * Classe permettant de tester les expressions.
 */

class TestExp {
   static void ok(boolean b) {
      if (b) {
         System.out.print("ok␣") ;
      } else {
         System.out.print("erreur␣") ;
      }
   }

   public static void main(String[] args) {
      Env env = new Env() ;
      env.associe("a", 1) ;
      env.associe("b", 2) ;
      Evaluation v = new Evaluation(env) ;
      Exp exp = new ExpBinMult(new Num(2), new ExpUnMoins(new Var("a"))) ;
      ok(v.eval(exp) == -2) ;
      Exp exp2 = new ExpBinPlus(new Var("a"), new Num(1)) ;
      ok(v.eval(exp2) == 2) ;
      Exp exp3 = new ExpBinMoins(new Var("a"), new Num(1)) ;
      ok(v.eval(exp3) == 0) ;
      Exp exp4 = new ExpBinMult(new Var("b"), new Var("a")) ;
      ok(v.eval(exp4) == 2) ;
      Exp exp5 = new ExpUnPlus(new Num(-1)) ;
      ok(v.eval(exp5) == -1) ;
      Exp exp6 = new ExpUnMoins(new Num(-1)) ;
      ok(v.eval(exp6) == 1) ;
      System.out.println() ;
   }
}
```

**Programme Java avec la variante générique du patron Visiteur**

```
interface Visiteur<T> {
   T visite(ExpBinPlus e) ;
   T visite(ExpBinMoins e) ;
   T visite(ExpBinMult e) ;
   T visite(ExpUnPlus e) ;
   T visite(ExpUnMoins e) ;
   T visite(Num e) ;
   T visite(Var e) ;
}

abstract class Exp {
   abstract public <T> T applique(Visiteur<T> v) ;
}

class ExpBinPlus extends ExpBin {
   public ExpBinPlus(Exp g, Exp d) {
      super(g, d) ;
```

```
    }
    public <T> T applique(Visiteur<T> v) {
        return v.visite(this) ;
    }
}

// ... Idem pour les autres classes qui dérivent de Exp

class Evaluation implements Visiteur<Integer> {
    private final Env env;
    public Evaluation(Env e) {env = e;}
    public Integer visite(ExpBinPlus e) {
        return e.gauche.applique(this) + e.droit.applique(this);
    }
    public Integer visite(ExpBinMoins e) {
        return e.gauche.applique(this) - e.droit.applique(this);
    }
    public Integer visite(ExpBinMult e) {
        return e.gauche.applique(this) * e.droit.applique(this);
    }
    public Integer visite(ExpUnPlus e) {
        return e.exp.applique(this);
    }
    public Integer visite(ExpUnMoins e) {
        return -e.exp.applique(this);
    }
    public Integer visite(Num n) {
        return n.val;
    }
    public Integer visite(Var v) {
        return env.valeur(v.nom);
    }
}

// On implémente un autre visiteur permettant d'afficher une expression

class Afficheur implements Visiteur<String> {
    public String visite(ExpBinPlus e) {
        return "(" + e.gauche.applique(this) + "␣+␣"
          + e.droit.applique(this) + ")";
    }
    public String visite(ExpBinMoins e) {
        return "(" + e.gauche.applique(this) + "␣-␣"
          + e.droit.applique(this) + ")";
    }
    public String visite(ExpBinMult e) {
        return e.gauche.applique(this) + "␣*␣" + e.droit.applique(this) ;
    }
    public String visite(ExpUnPlus e) {
        return "+" + e.exp.applique(this);
    }
    public String visite(ExpUnMoins e) {
        return "-" + e.exp.applique(this);
```

```
    }
    public String visite(Num n) {
        return String.valueOf(n.val);
    }
    public String visite(Var v) {
        return v.nom;
    }
}

class Test {
     public static void main(String[] args) {
        Exp e = new ExpBinMult(new Num(2), new ExpUnMoins(new Var("a")));
        Env env = new Env();
        env.associe("a", -36);
        Evaluation eval = new Evaluation(env);
        Afficheur aff = new Afficheur();
        System.out.println(e.applique(aff) + "␣=␣" + e.applique(eval));
    }
}
```

## Exercice 25. Patron Décorateur

### 1. Diagramme de classes

La classe `Composant` du patron Décorateur correspond à une classe abstraite `Pizza`. Les composants concrets correspondent aux pizzas à pâte fine et pizzas à pâte épaisse (classes `PizzaPateFine` et `PizzaPateEpaisse`).

On a une classe `DecorateurPizza` pour décorer les pizzas. Les décorateurs concrets sont les différents ingrédients d'une pizza : `Jambon`, `Oeuf`, `Gruyere`, `Mozzarella` ... etc.

Il faut stocker à la fois le prix des pâtes à pizza et le prix de chaque ingrédient. On fait le choix de représenter ces prix dans un unique attribut `prix`, factorisé au niveau de la classe `Pizza`. Cet attribut est protégé.

On pourrait mettre une méthode calculerPrix() dans chaque sous-classe de `DecorateurPizza`. On fait le choix de factoriser ces méthodes dans `DecorateurPizza`.

cf. Figure 27.

### 2. Diagramme d'objets

cf. Figure 28.

### 3. Code Java

```
abstract class Pizza {
    protected float prix;
    float calculerPrix() {
        return prix;
```

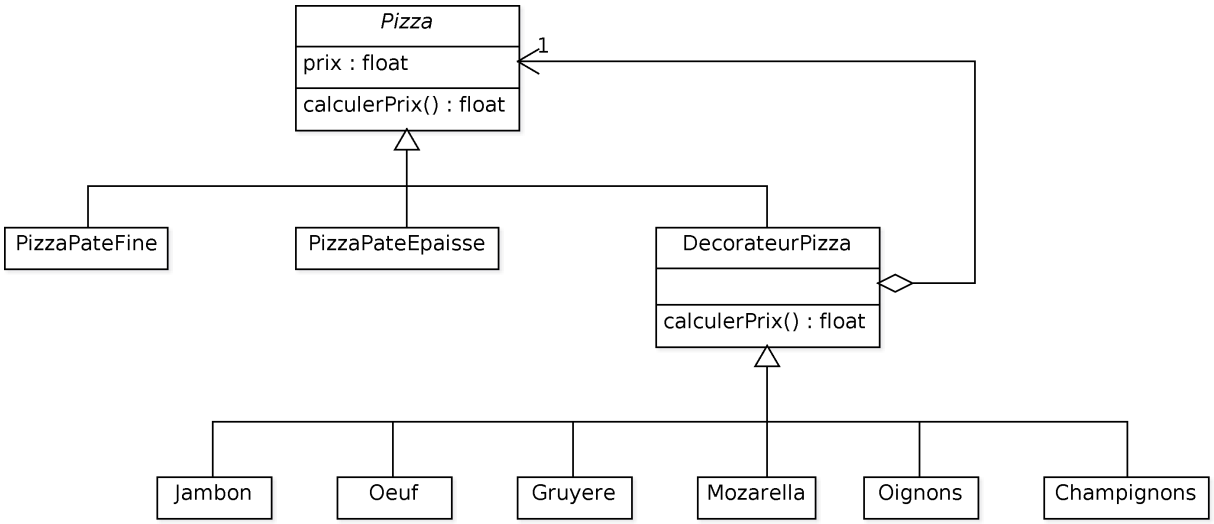



FIGURE 27 – Diagramme de classes

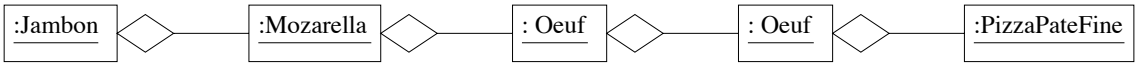



FIGURE 28 – Diagramme d'objets

```
    }
}

class PizzaPateFine extends Pizza {
    PizzaPateFine() {
        prix = 3.0f;
    }
}

abstract class DecorateurPizza extends Pizza {
    Pizza pizza;
    DecorateurPizza(Pizza pizza) {
        this.pizza = pizza;
    }
    float calculerPrix() {
        return pizza.calculerPrix() + prix;
    }
}

class Jambon extends DecorateurPizza {
    Jambon(Pizza pizza) {
        super(pizza);
        prix = 1.5f;
    }
}
```

```
class Oeuf extends DecorateurPizza {
   Oeuf(Pizza pizza) {
      super(pizza);
      prix = 1.2f;
   }
}

class Test {
   public static void main(String[] args) {
      Pizza p = new Oeuf(new Oeuf(new Jambon(new PizzaPateFine())));
      System.out.println("Prix␣:" + p.calculerPrix());
   }
}
```

## Exercice 26. Exercice sur les circuits

### Question 1. Diagramme de classes

On utilise le patron Interprète pour la représentation des circuits (un circuit a la même structure qu'une expression booléenne, la porte Non est une expression unaire, et les deux portes Et et Ou sont des expressions binaires).

Le diagramme de classes correspondant est représenté figure 29.

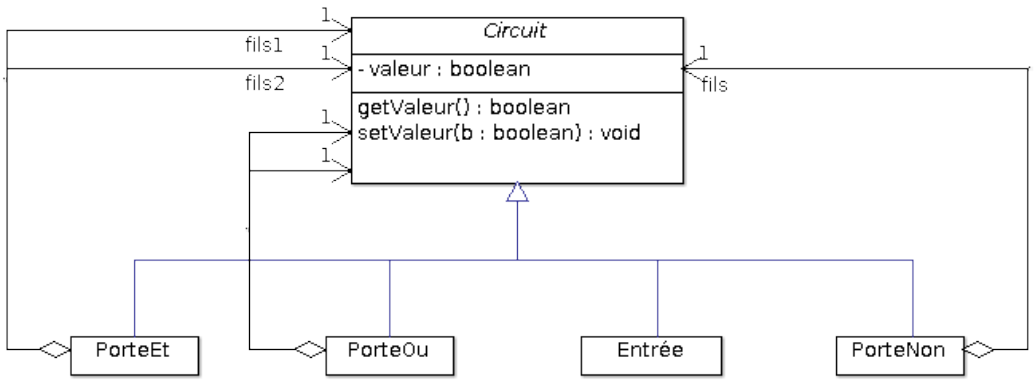



FIGURE 29 – Diagramme de classes pour le patron Interprète

On utilise le patron Observateur pour la mise à jour des sorties. Dans la solution présentée ici, la classe `Circuit` est à la fois Sujet et Observateur.

Lorsqu'une valeur d'entrée est modifiée, les valeurs de sorties des portes utilisant cette entrée doivent être mises à jour. Ces sorties, qui peuvent être des fils (entrées) d'autres portes doivent à leur tour provoquer la mise à jour des sorties de ces autres portes. Sur l'exemple de l'énoncé, la modification de l'entrée $e_2$ doit modifier la valeur des sorties des portes Non et Et, qui vont modifier la sortie de la porte Ou.

Un circuit va donc être à la fois Sujet (son état, correspondant à l'attribut valeur peut être modifié), et cette modification va entraîner la mise à jour de toutes les Portes ayant sa sortie comme entrée. Toutes les portes sont donc des Observateurs.

Dans la solution présentée figure 30, on fait le choix que tous les circuits implémentent l'interface `Observateur` mais ce n'est en fait pas nécessaire pour la classe `Entrée`.

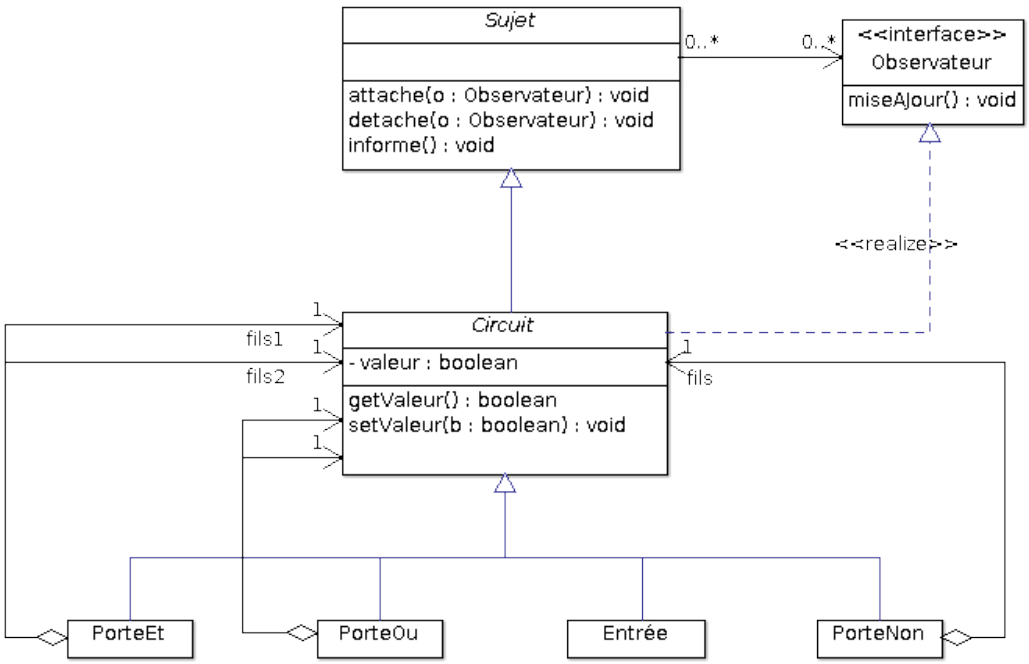



FIGURE 30 – Diagramme de classes complet

### Question 2. Code Java

```
import java.util.Set ;
import java.util.HashSet ;

// L'interface Observateur du patron Observateur
interface Observateur {
   void miseAJour() ;
}

// La classe Sujet du patron Observateur
abstract class Sujet {
   private Set<Observateur> lesObservateurs ;
   Sujet() {
      lesObservateurs = new HashSet<Observateur>() ;
   }
   void attache(Observateur o) {
      lesObservateurs.add(o) ;
```

```
      o.miseAJour() ;
   }
   void detache(Observateur o) {
      lesObservateurs.remove(o) ;
   }
   void informe() {
      for (Observateur o : lesObservateurs) {
         o.miseAJour() ;
      }
   }
}

// Les circuits sont à la fois Sujet et Observateur
abstract class Circuit extends Sujet implements Observateur {
   // Attribut privé, afin que les sous-classes soient obligées de
   // passer par la méthode setValeur, qui appellera informe().
   private boolean valeur ;
   boolean getValeur() {
      return valeur ;
   }
   void setValeur(boolean valeur) {
      this.valeur = valeur ;
      // L'état est modifié, il faut donc informer tous les observateurs
      informe() ;
   }
   public abstract void miseAJour() ;
}

class Entree extends Circuit {
   Entree(boolean valeur) {
      setValeur(valeur) ;
   }
   public void miseAJour() { } // Rien à faire
}

class PorteNon extends Circuit {
   Circuit fils ;
   PorteNon(Circuit fils) {
      this.fils = fils ;
      fils.attache(this) ;
   }
   public void miseAJour() {
      setValeur(!fils.getValeur()) ;
   }
}

class PorteEt extends Circuit {
   Circuit fils1 ;
   Circuit fils2 ;
   PorteEt(Circuit fils1, Circuit fils2) {
      this.fils1 = fils1 ;
      this.fils2 = fils2 ;
      fils1.attache(this) ;
```

```
      fils2.attache(this) ;
   }
   public void miseAJour() {
      setValeur(fils1.getValeur() && fils2.getValeur()) ;
   }
}

class PorteOu extends Circuit {
   Circuit fils1 ;
   Circuit fils2 ;
   PorteOu(Circuit fils1, Circuit fils2) {
      this.fils1 = fils1 ;
      this.fils2 = fils2 ;
      fils1.attache(this) ;
      fils2.attache(this) ;
   }
   public void miseAJour() {
      setValeur(fils1.getValeur() || fils2.getValeur()) ;
   }
}

// Un programme de test
class Test {
   public static void main(String[] args) {
      Entree e1 = new Entree(false) ;
      Entree e2 = new Entree(true) ;
      Circuit p1 = new PorteNon(e1) ;
      Circuit p2 = new PorteEt(p1, e2) ;
      System.out.println("e1.getValeur()␣=␣" + e1.getValeur()) ;
      System.out.println("e2.getValeur()␣=␣" + e2.getValeur()) ;
      System.out.println("p1.getValeur()␣=␣" + p1.getValeur()) ;
      System.out.println("p2.getValeur()␣=␣" + p2.getValeur()) ;
      e1.setValeur(true) ;
      System.out.println("\ne1␣=␣true") ;
      System.out.println("e1.getValeur()␣=␣" + e1.getValeur()) ;
      System.out.println("e2.getValeur()␣=␣" + e2.getValeur()) ;
      System.out.println("p1.getValeur()␣=␣" + p1.getValeur()) ;
      System.out.println("p2.getValeur()␣=␣" + p2.getValeur()) ;
   }
}
```

## Exercice 27. Conception de la minuterie

### 1. Utilisation du patron Observateur

On décide de gérer les mises à jour de l'interface graphique à l'aide du patron Observateur. Ce patron définit une classe Sujet dont héritent tous les objets qui peuvent être observés et une interface Observateur qui contient une méthode miseAJour. Tout Observateur, c'est-à-dire tout objet qui peut observer un Sujet, implémente l'interface Observateur. Chaque fois qu'un Sujet change d'état, il appelle la méthode informe. Cette méthode appelle la méthode miseAJour de tous les Observateurs associés à ce Sujet.

Une Etiquette est un observateur de Compteur, qui sert à afficher la valeur du Compteur. La classe Etiquette dérive de la classe Java-Swing JLabel.

Un IndicateurAlarme est un observateur d'Alarme, qui sert à afficher le niveau de l'alarme. La classe IndicateurAlarme dérive de la classe Java-Swing JProgressBar.

Le diagramme de classe correspondant est représenté figure 31.

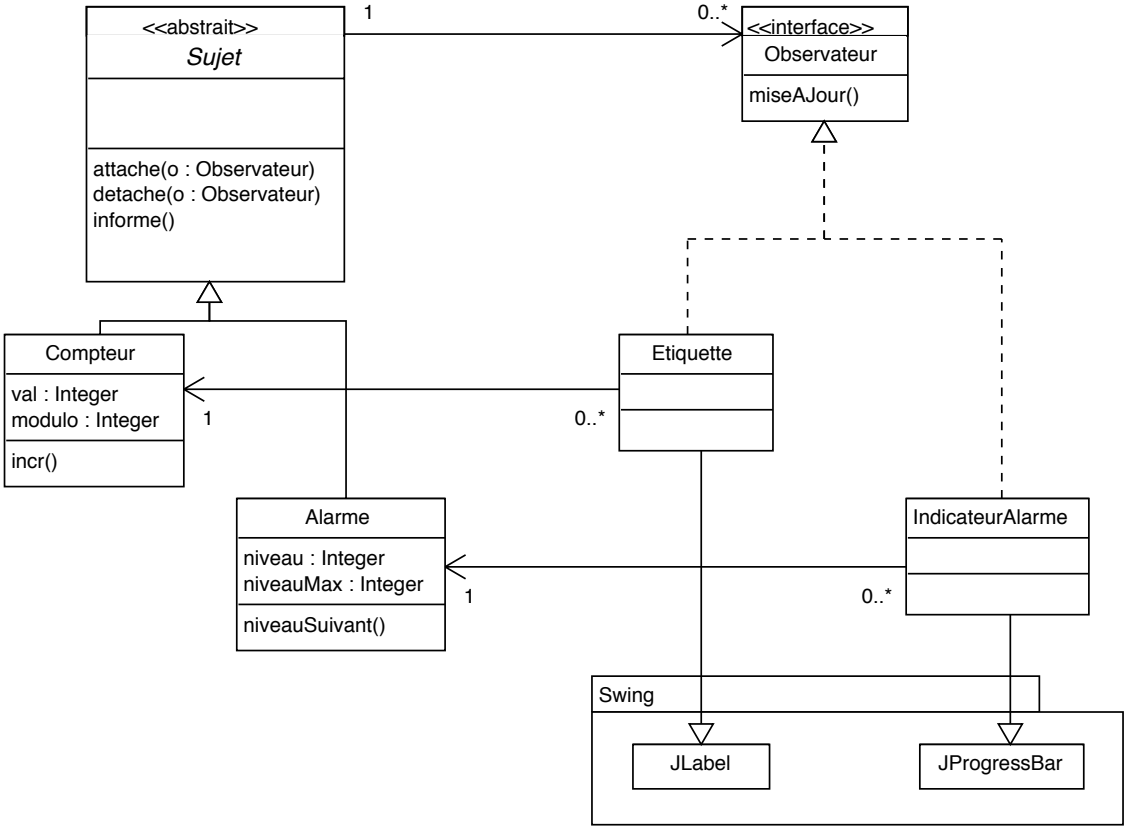



FIGURE 31 – Diagramme de classes pour le patron Observateur

La méthode incr de la classe Compteur modifie la valeur du compteur. Elle doit donc terminer par un appel à la méthode informe, qui réalisera la mise à jour de tous les observateurs de Compteur. De même pour la méthode niveauSuivant de la classe Alarme.

### 2. Utilisation du patron Etat

Les différents états du contrôleur sont définis à l'aide d'une hiérarchie d'états qui reflète la hiérarchier état composite / sous-état de l'automate.

Les états composites sont représentés par des classes abstraites, dont dérivent les classes qui

coreespondent aux sous-états.

Les états élémentaires sont représentés par des classes concrètes. Chaque classe concrète définit une instance unique d'état : attribut statique `etat`.

Les différents états de l'automate sont donc `Normal.etat`, `EditHeures.etat`, `EditMinutes.etat`, `Marche.etat`, `AlarmeActive.etat`.

Le diagramme de classes correspondant est représenté figure 32.

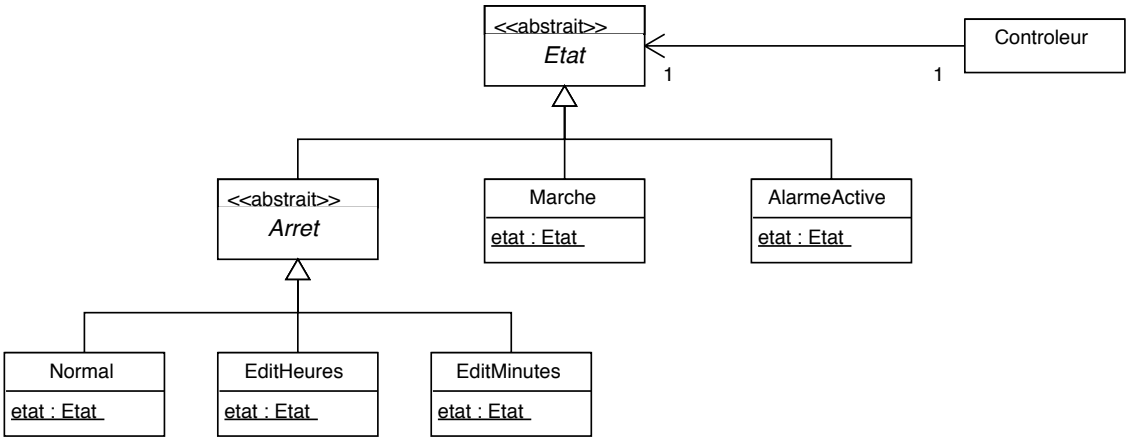



FIGURE 32 – Diagramme de classes pour le patron Etat

### 3. Utilisation du patron Visiteur

Les actions à effectuer en cas de clic sur les boutons Mode, Incr et Start/Stop dépendent de l'état courant du contrôleur, défini par une valeur de type Etat. On utilise donc le patron Visiteur pour définir ces actions sur la hiérarchie d'états.

On définit une interface Visiteur avec une opération

```
void visite(EtatXXX etat);
```

pour chaque classe concrète `EtatXXX` qui dérive de `Etat`.

```
interface Visiteur {
   // Action à effectuer par ce visiteur lorsque le contrôleur est dans
   // l'état Normal.
   void visite(Normal etat);
   // Action à effectuer par ce visiteur lorsque le contrôleur est dans
   // l'état EditHeures.
   void visite(EditHeures etat);
   // Action à effectuer par ce visiteur lorsque le contrôleur est dans
   // l'état EditMinutes.
   void visite(EditMinutes etat);
```

```
    // Action à effectuer par ce visiteur lorsque le contrôleur est dans
    // l'état Marche.
    void visite(Marche etat);
    // Action à effectuer par ce visiteur lorsque le contrôleur est dans
    // l'état AlarmeActive.
    void visite(AlarmeActive etat);
}
```

On écrit une méthode abstraite

```
abstract void applique(Visiteur v);
```

dans la classe `Etat`. Dans chaque classe concrète de la hiérarchie d'états, on définit une méthode concrète

```
void applique(Visiteur v) {
    v.visite(this);
}
```

On définit les classes `AppuiMode`, `AppuiIncr` et `AppuiStartStop`, qui implémentent l'interface `Visiteur`, et définissent respectivement les actions à effectuer en réponse à un clic sur les boutons Mode, Incr et Start/Stop.

Chaque classe implémente les différentes méthodes `visite`. Par exemple, la méthode

```
void visite(Marche etat) { ... }
```

de la classe AppuiMode définit l'effet d'un clic sur le bouton Mode lorsque le contrôleur est dans l'état Marche.

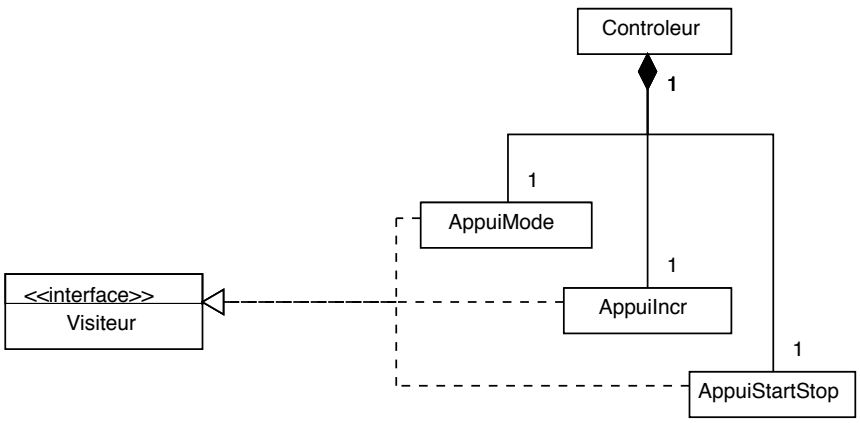



FIGURE 33 – Diagramme de classes pour le patron Visiteur

# Exercice 28. Courriers électroniques : conception

## 1. Patron pour la hiérarchie de boîtes

Le patron utilisé pour la hiérarchie de boîtes est le patron Composite. La classe BoiteAbstraite correspond au Composant, la classe Boite correspond au composé, et la classe Courrier à un composant simple (cf. figure 34).

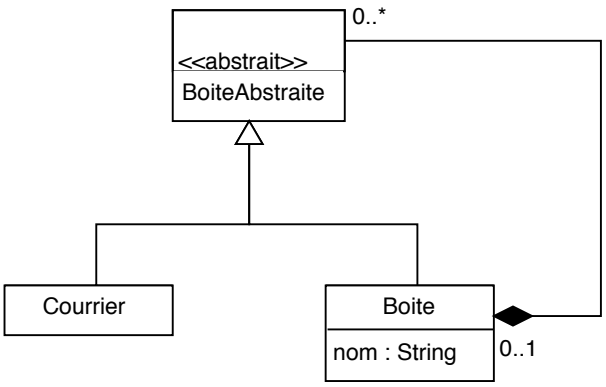



FIGURE 34 – Application du patron Composite

## 2. Patron pour l'affichage des boîtes

On peut utiliser le patron Observateur. Le sujet concret est la classe Boite, l'oservateur concret est la classe AffichageBoite, de la couche Présentation (cf. figure 35).

## 3. Patron Visiteur

- On ajoute une méthode `void applique (Visiteur v)` dans chaque classe de la hiérarchie BoiteAbstraite.
- On définit une interface Visiteur, avec une méthode `void visite(Element e)` pour chaque classe concrète Element héritée de BoiteAbstraite.
- Le codage de la recherche se fait en implémentant l'interface Visiteur.

La figure 36 montre le diagramme de classes correspondant.

### Code Java

```
// Fichier BoiteAbstraite.java
/**
 * Classe des boîtes abstraites. Les courriers et les boites
 * sont des boîtes abstraites.
 */
abstract class BoiteAbstraite {
   abstract void applique(Visiteur v);
}
```

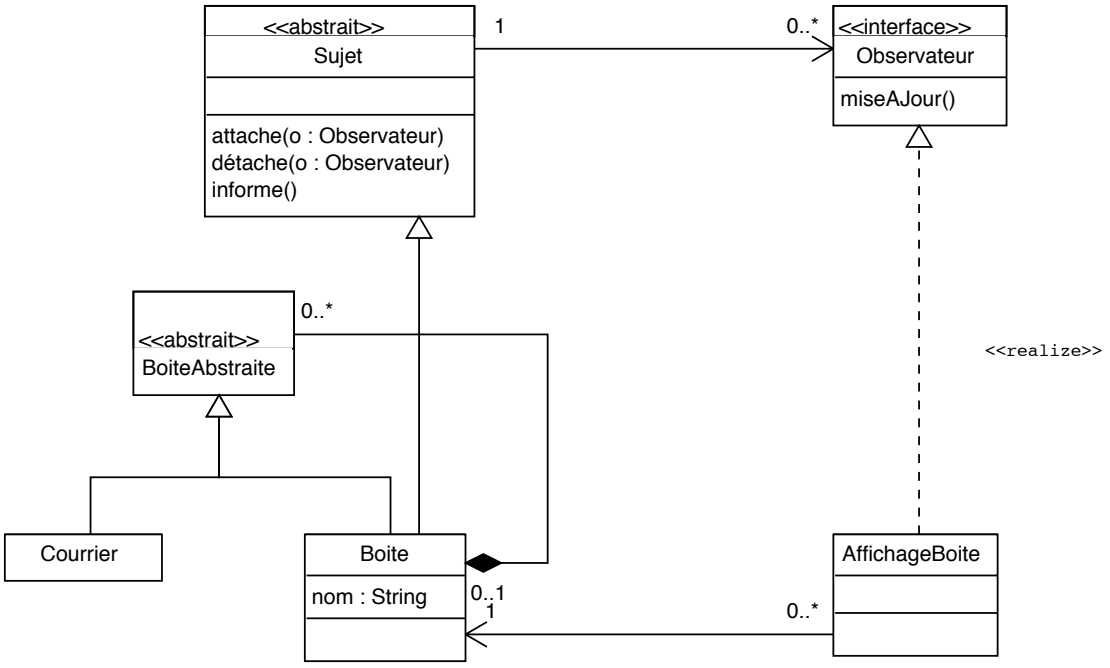



FIGURE 35 – Application du patron Observateur

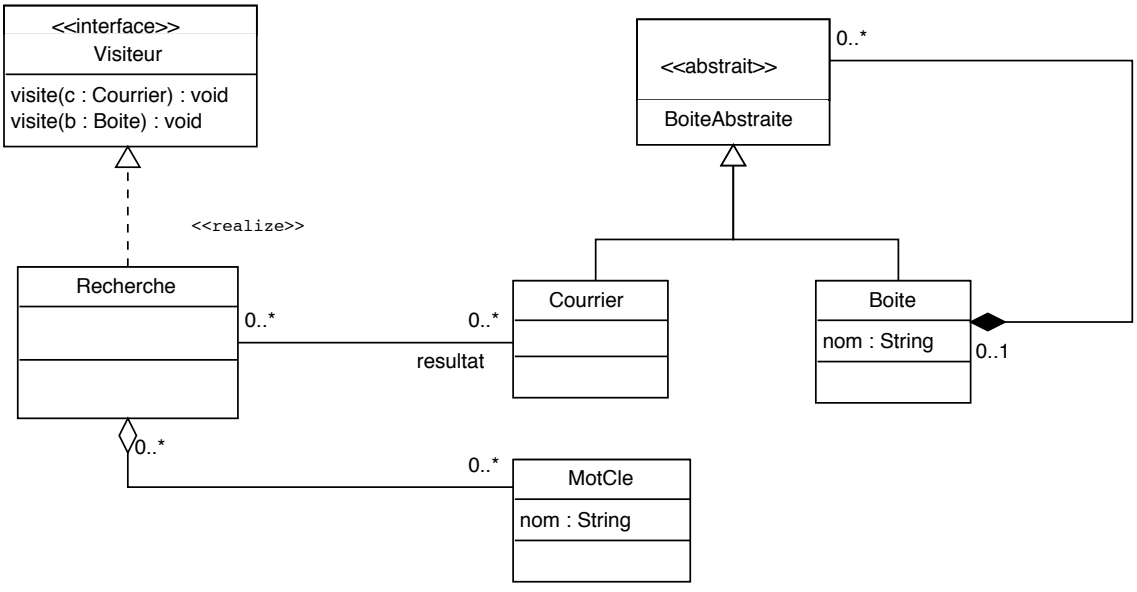



FIGURE 36 – Application du patron Visiteur

```
// Fichier Courrier.java
/**
 * Classe des courriers.
 */
class Courrier extends BoiteAbstraite {
   String corps ; // Corps du courrier
   Courrier(String corps) {
      this.corps = corps ;
   }
   void applique(Visiteur v) {
      v.visite(this) ;
   }
}

// Fichier Boite.java
import java.util.Set ;
import java.util.HashSet ;
/**
 * Classe des boîtes, qui peuvent contenir d'autres boîtes,
 * ou des courriers.
 */
class Boite extends BoiteAbstraite {
   Set<BoiteAbstraite> contenu ; // Contenu de la boîte
   String nom ; // Nom de la boîte
   Boite(String nom) {
      this.nom = nom ;
      contenu = new HashSet<BoiteAbstraite>() ;
   }

   void applique(Visiteur v) {
      v.visite(this);
   }
}

// Fichier Visiteur.java
/**
 * Interface Visiteur du patron Visiteur.
 */
interface Visiteur {
   void visite(Courrier c) ;
   void visite(Boite b) ;
}

// Classe Recherche.java
import java.util.Set ;
import java.util.HashSet ;
/**
 * Classe qui implémente la recherche de courriers qui contiennent
 * un ensemble de mots clés.
 * Le résultat se trouve dans this.resultat.
 */
class Recherche implements Visiteur {
```

```
   Set<Courrier> resultat ;
   Set<String> lesMotsCles ;

   Recherche(Set<String> lesMotsCles) {
      resultat = new HashSet<Courrier>() ;
      this.lesMotsCles = lesMotsCles ;
   }

   static boolean contient
         (Courrier courrier, Set<String> listeMotsCles) {
      boolean ok = true ;
      for (String motCle : listeMotsCles) {
         if (!courrier.corps.contains(motCle)) {
            ok = false ;
         }
      }
      return ok ;
   }

   public void visite(Courrier courrier) {
      if (contient(courrier, lesMotsCles)) {
         resultat.add(courrier);
      }
   }

   public void visite(Boite boite) {
      for (BoiteAbstraite ba : boite.contenu) {
         ba.applique(this) ;
      }
   }
}
```

## Exercice 29. Outil de gestion de conférences

### 1. Analyse

a) Les acteurs : cf. figure 37

b) Cas d'utilisation : cf. figures 38 et 39.

On précise le cahier des charges en indiquant qu'un rapporteur peut télécharger les articles qu'il évalue, et que tous les membre du comité peuvent télécharger tous les articles.

c) Scénarios d'utilisation du système

La figure 40 montre un diagramme de séquence qui représente la première soumission d'un article, suivie de sa correction par un co-auteur.

La figure 41 montre un diagramme de séquence qui représente la nomination d'un rapporteur délégué par un membre du comité, ainsi que la soumission de la note et du rapport par ce rapporteur délégué.

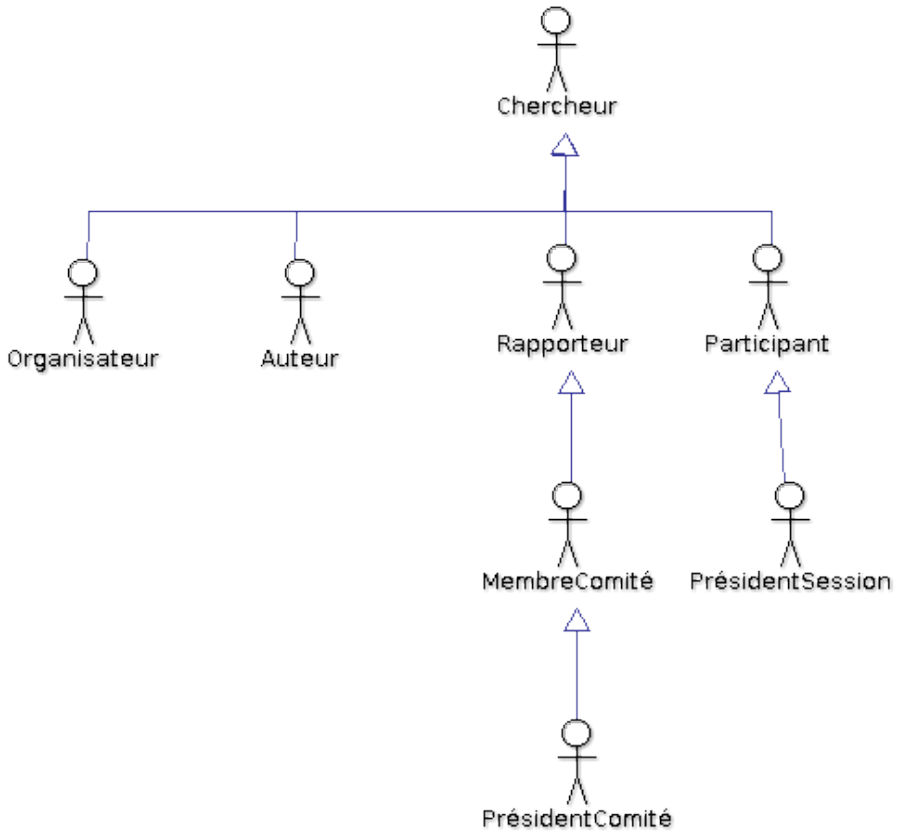



FIGURE 37 – Acteurs

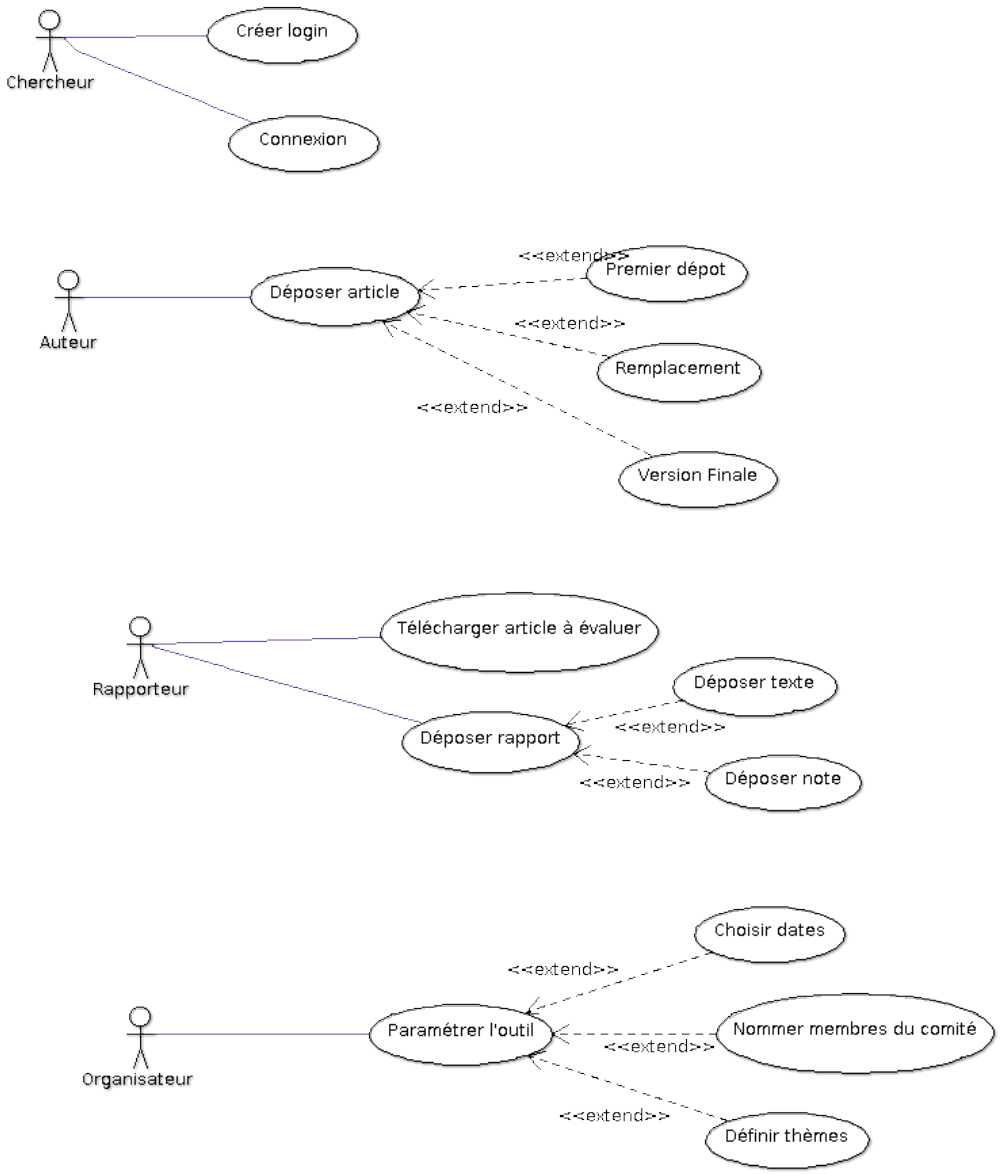



FIGURE 38 – Cas d'utilisation (1)

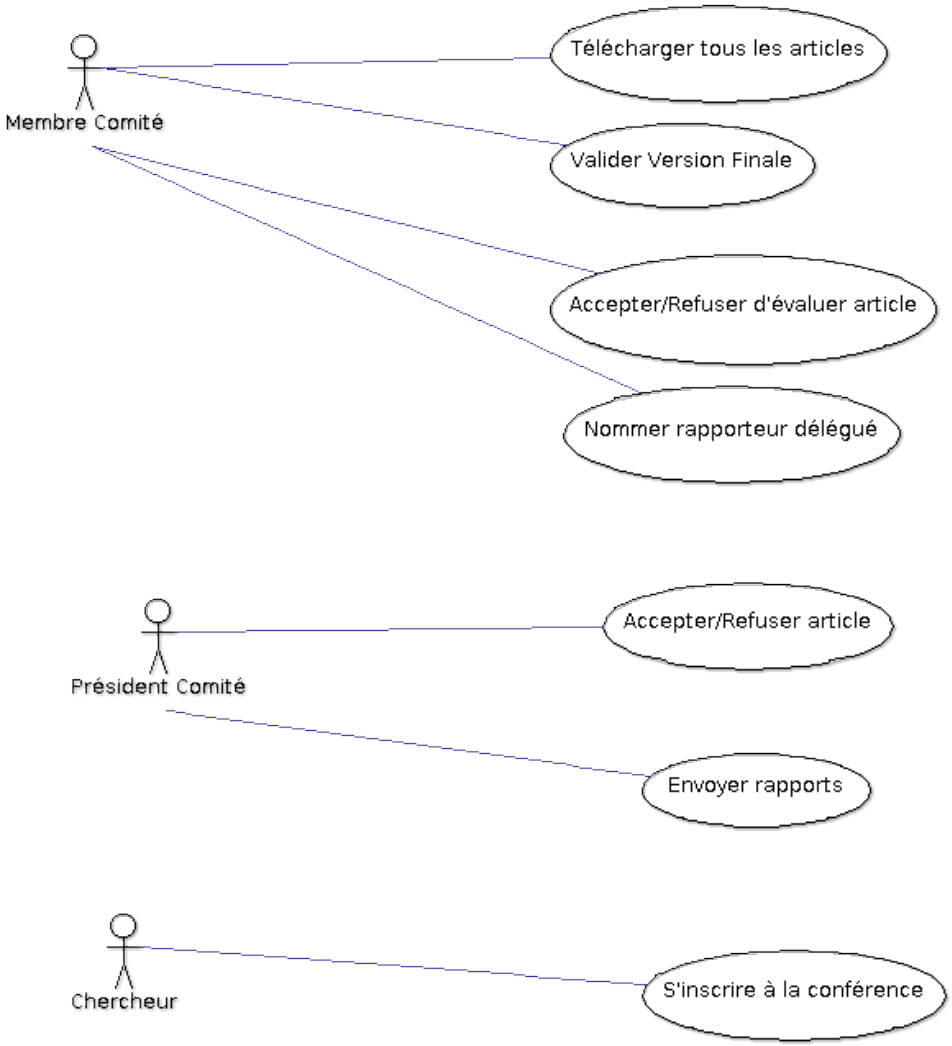



FIGURE 39 – Cas d'utilisation (2)

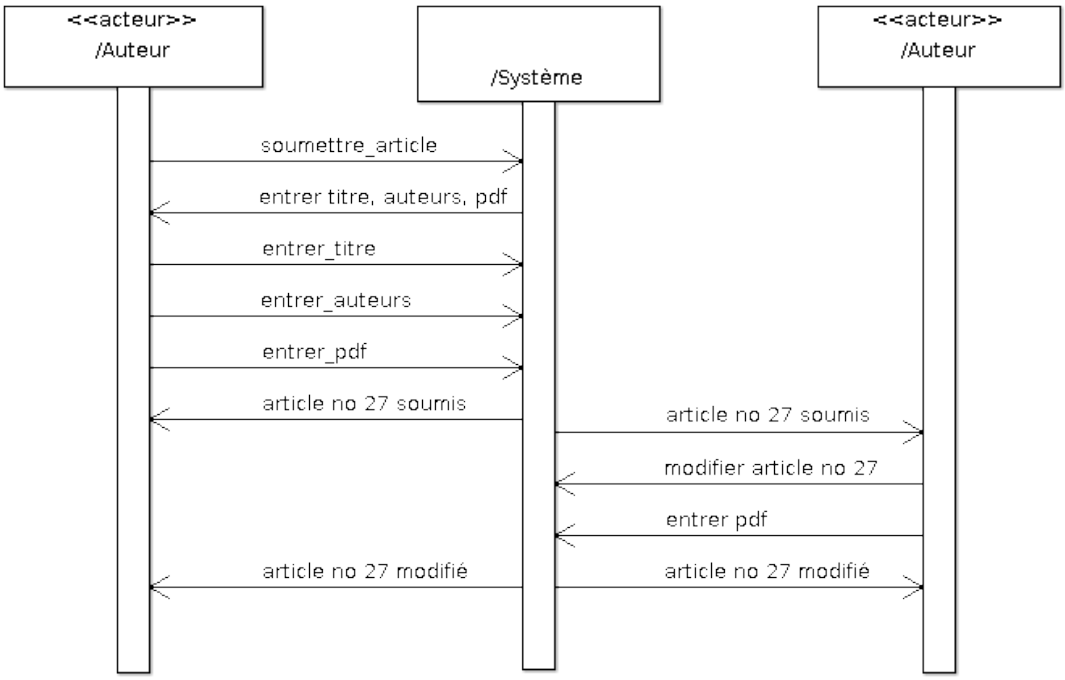



FIGURE 40 – Diagramme de séquence : soumission et correction d'un article

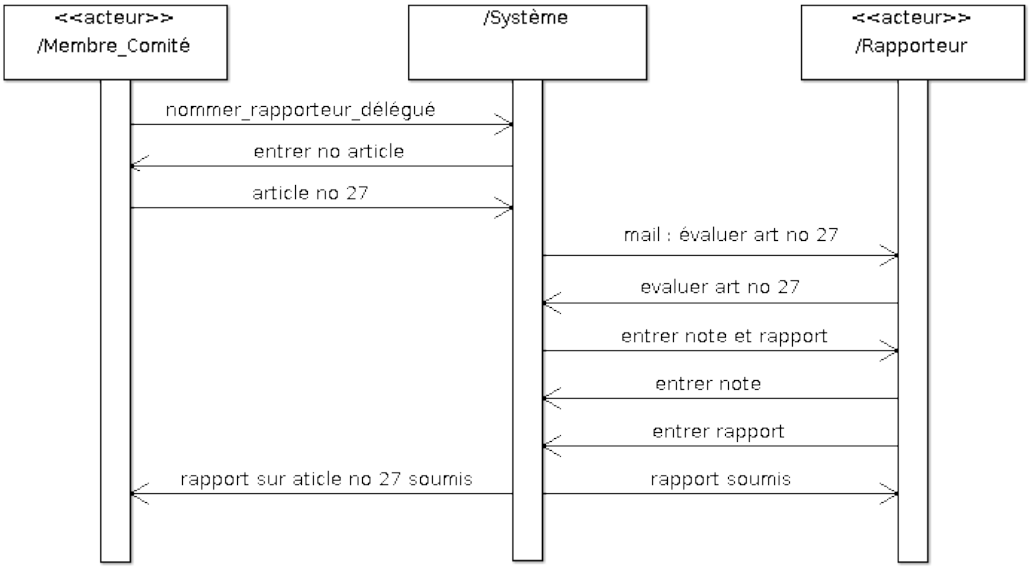



FIGURE 41 – Diagramme de séquence : nomination d'un rapporteur délégué et soumission d'une note et d'un rapport

La figure 42 montre un diagramme de séquence qui représente le signal d'envoi de tous les résultats par le président du comité, suivi de l'envoi des résultats aux auteurs de l'article numéro 27.

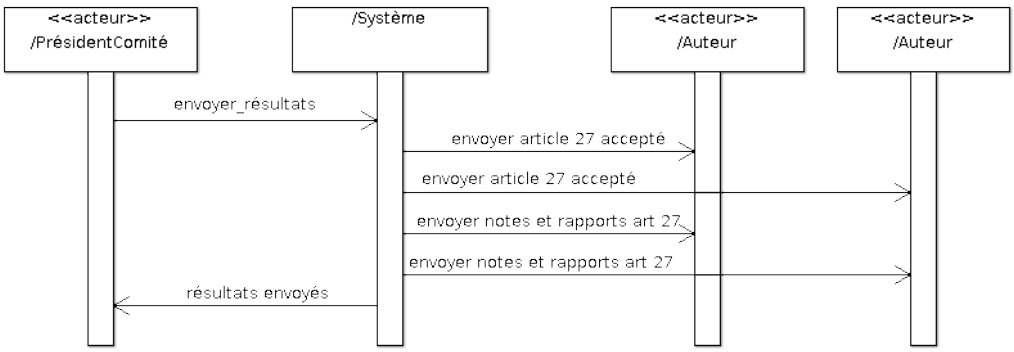



FIGURE 42 – Diagramme de séquence : signal d'envoi de tous les résultats aux auteurs par le président du comité

**2. Diagramme de classes d'analyse**

*Solution 1*

cf. diagramme de classes Figure 43

Chaque article est évalué par entre 1 et 3 membres du comité. A chaque évaluation correspond une instance de la classe Rapport, qui est donc une classe-association entre les classes Article et Chercheur.

De plus, on introduit une relation entre Rapport et Chercheur : à chaque rapport est associé 0 ou 1 chercheur, lorque le lien existe entre un rapport et un chercheur, cela signifie que ce chercheur a été nommé comme rapporteur délégué par le membre du comité chargé d'évaluer l'article.

*Solution 2 : avec rôles*

cf. diagramme de classes Figure 44.

Dans ce diagramme de classes, on introduit une classe `Rôle`, car un chercheur peut avoir plusieurs rôles pour une conférence donnée (il peut par exemple être à la fois membre du comité et auteur).

D'autre part, un chercheur peut avoir des rôles différents suivant les conférences. Pour modéliser cela, on introduit une classe `Couple-Chercheur-Conférence`, qui représente tous les

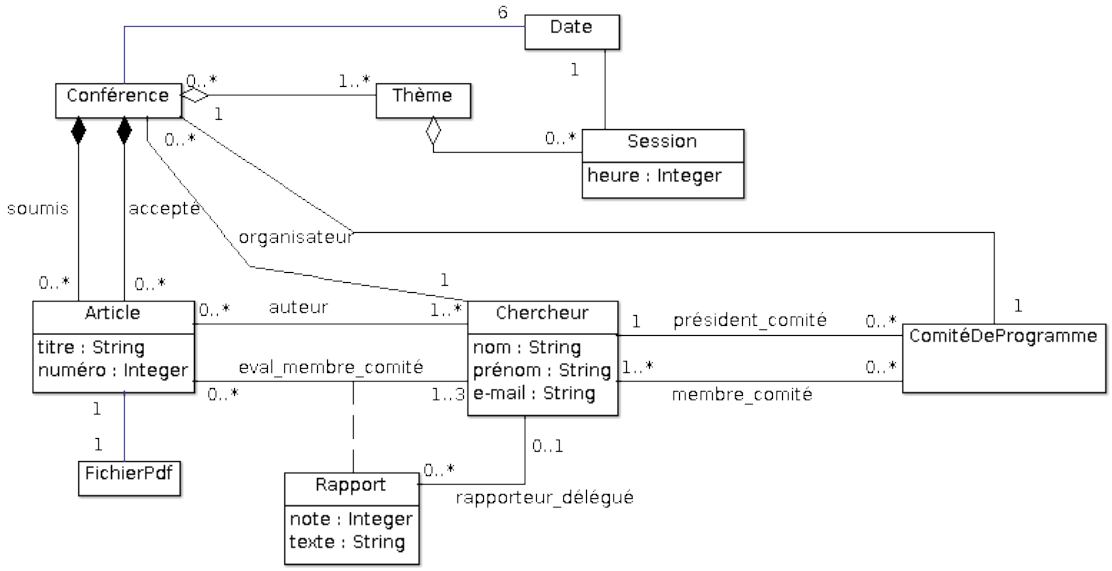



FIGURE 43 – Diagramme de classes (solution 1)

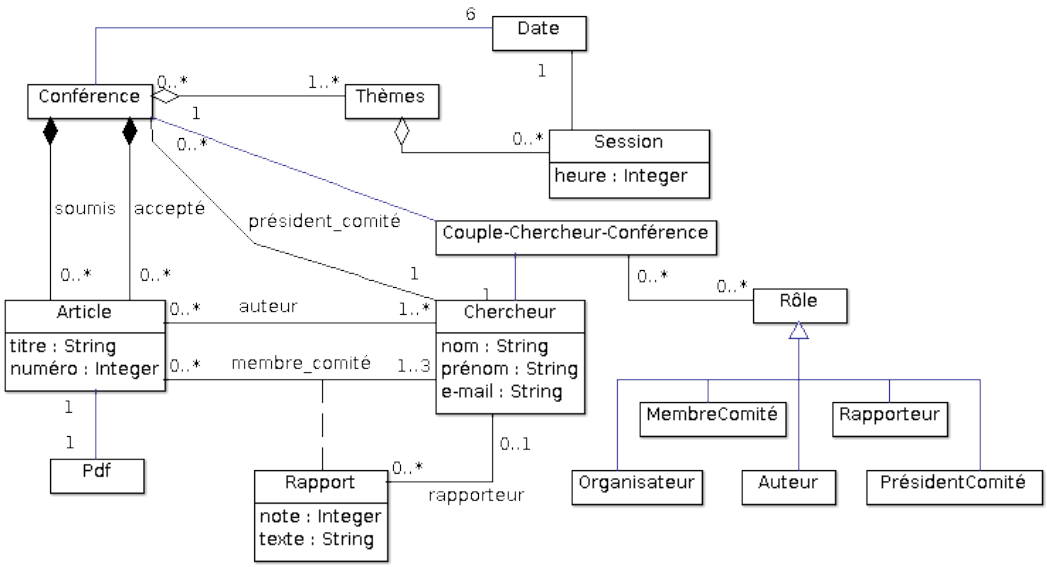



FIGURE 44 – Diagramme de classes : solution 2, avec rôles explicites

couples (chercheur, conférence). À chaque couple (chercheur, conférence), on associe un ensemble de rôles.

Les différents rôles (Auteur, Rapporteur, Organisateur...) seront ensuite implémentés par des singletons. On aura donc une instance unique de chacune des sous-classes de Rôle.

### 3. Architecture

On peut par exemple choisir une architecture en couches (cf. Figure 45).

#### Contenu des différentes couches

*Présentation*

La couche présentation comporte une vue contenant un menu général qui donne accès à toutes les fonctionalités que peut réaliser une personne en fonction de ses rôles. On a ensuite une vue par cas d'utilisation général, par exemple :

- Soumettre un article
- Soumettre un rapport
- Paramétrer l'outil
- Nommer un rapporteur délégué
- Accepter/Refuser un article
- ...

*Application*

La couche application implémente les diverses fonctionalités de l'outil. On retrouve donc dans cette couche un paquetage par cas d'utilisation général.

*Modèle*

La couche Modèle reprend les principales classes du diagramme de classes d'analyse. Ici on fait le choix de regrouper ces classes en trois paquetages : GestionConf, GestionComptes, GestionArticles.

*Infrastructure*

La couche infrastructure comporte des classes qui permettent de faire le lien avec la BD.

On peut également choisir une architecture de type MVC (cf. figure 46). On a choisi ici une architecture où les contrôleurs sont centraux, et la vue complètement indépendante du modèle (architecture MVC de type "framework").

#### Contenu des différents paquetages

*Modèle*

Comme précédemment, la couche Modèle reprend les principales classes du diagramme de classes d'analyse. On a en plus un paquetage LienAvecBD qui réalise la liaison lien avec la

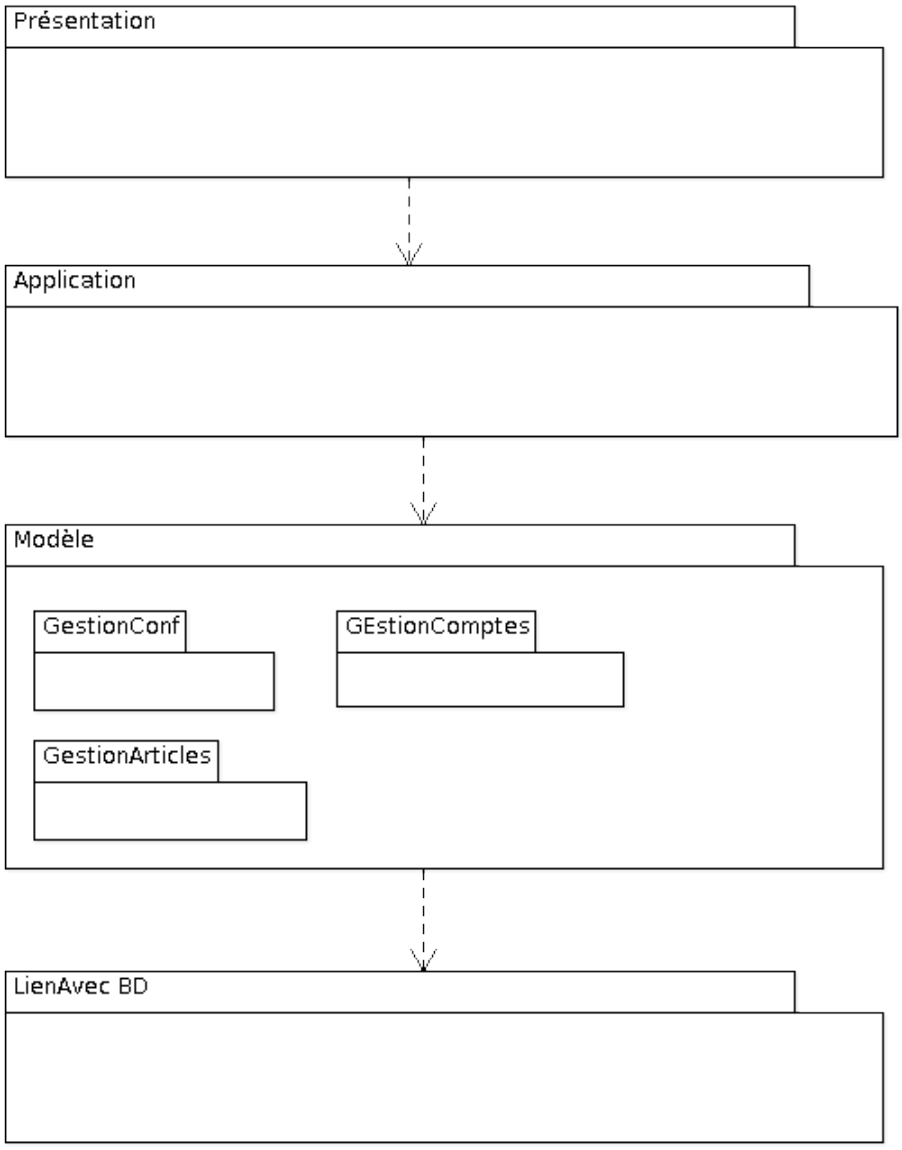



FIGURE 45 – Architecture en couches

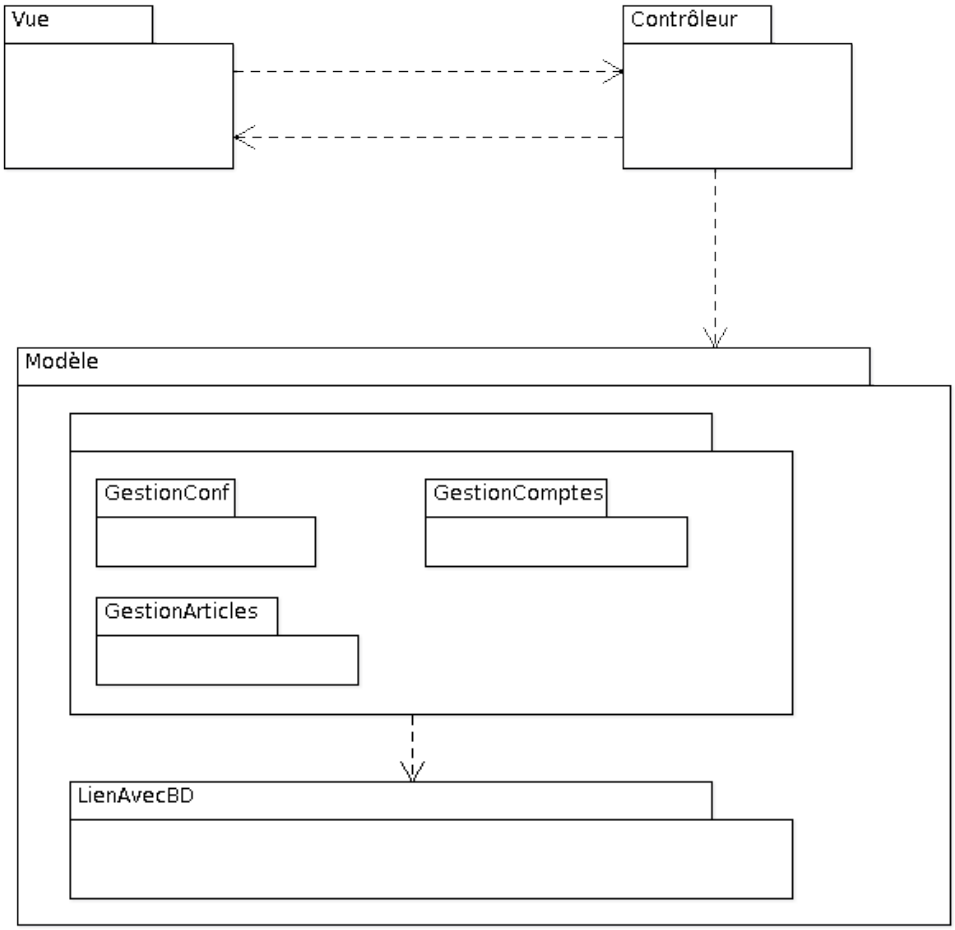



FIGURE 46 – Architecture MVC

BD.

*Vue*

Toujours comme précédemment, on choisit de faire une vue contenant un menu général qui donne accès à toutes les fonctionalités que peut réaliser une personne en fonction de ses rôles. On a ensuite une vue par cas d'utilisation général.

*Contrôleur*

Étant donné la taille de l'application, on choisir d'avoir plusieurs contrôleurs. Plusieurs choix sont possibles pour ces contrôleurs : on peut par exemple

— avoir un contrôleur par rôle,
— ou avoir un contrôleur par vue.

### 4. Diagramme d'états-transitions pour un article

cf. Figure 47.

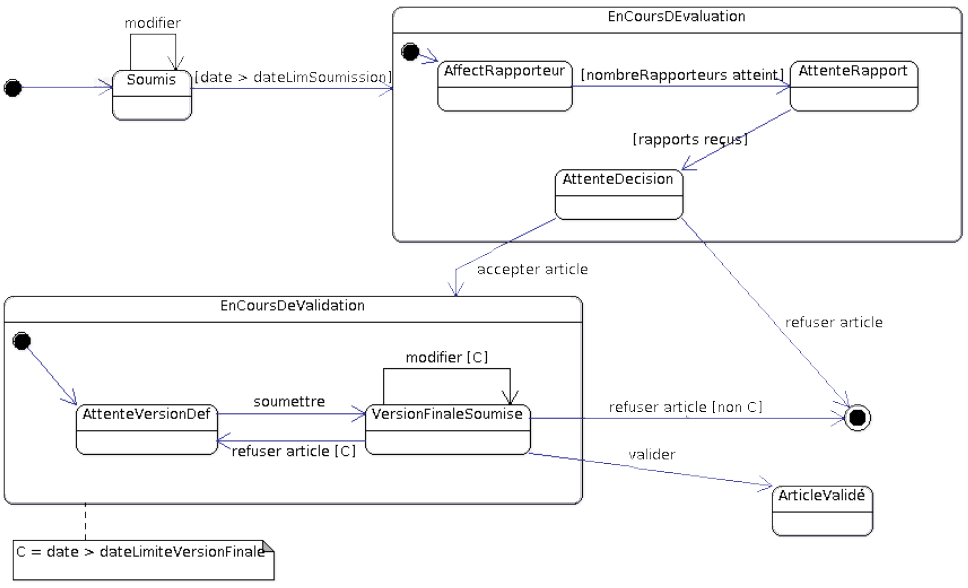



FIGURE 47 – Diagramme d'états-transitions pour un article