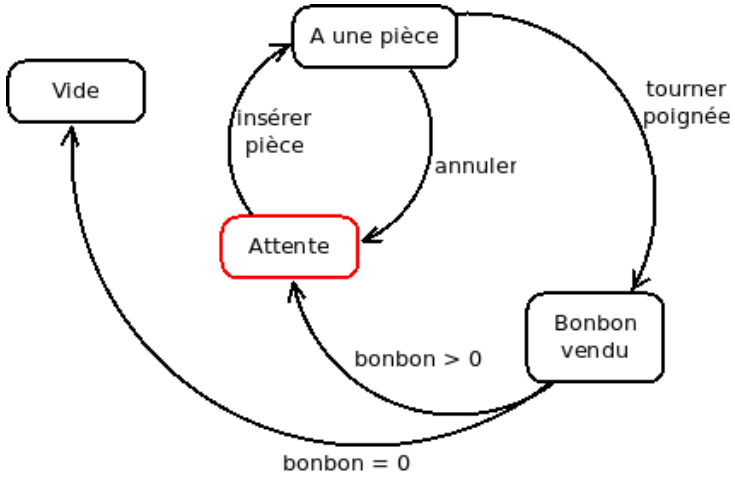
TD n°3 : Pattern State

Exercice 1 (Le distributeur de bonbon)

Le but est ici de programmer un distributeur de bonbon à l'ancienne (avec une poignée à tourner pour servir le bonbon), avec des fonctionnalités basiques, puis d'y ajouter une nouvelle fonctionnalité.

Voici le diagramme d'état modélisant le comportement du distributeur.



1. Écrire l'interface Etat qui contiendra la déclaration des méthodes associées à chaque action.
2. Écrire une implémentation de l'interface Etat pour chaque état possible.
3. Écrire la classe Distributeur ayant comme attributs une variable de type Etat pour chaque état possible et le nombre de bonbon en stock, un constructeur initialisant chaque état et le nombre de bonbon, et les méthodes correspondant aux transitions que peuvent effectuer un utilisateur du distributeur.
4. On souhaite modifier le comportement du distributeur ainsi : lorsque quelqu'un tourne la poignée, il aura une chance sur 10 d'avoir deux bonbons au lieu d'un, si les stocks le permettent. Modifier le diagramme d'état ainsi que votre programme en vous appuyant sur l'aide suivante :

```
// Bibliotheque a importer pour les diverses fonctionnalites
// concernant les generateurs pseudo-aleatoires.
import java.util.*;

// Initialiser un generateur pseudo-aleatoire
Random generateur = new Random( System.currentTimeMillis() );

// Tirer (pseudo) aleatoirement un nombre entre 0 et n-1 inclus.
int nombre = generateur.nextInt(n);
```

Exercice 2 (Le ventilateur)

Le code suivant décrit le changement d'état, et donc d'action, d'un ventilateur en fonction des pressions successives sur un bouton de commande.

```
class Ventilateur
{
  private final int ETEINT_ = 0;
  private final int FAIBLE_ = 1;
  private final int MOYEN_  = 2;
  private final int FORT_   = 3;

  private int etatCourant;

  public Ventilateur()
  {
    etatCourant = ETEINT_;
  }

  public void bouton()
  {
    if (etatCourant == ETEINT_)
      {
        etatCourant = FAIBLE_;
        System.out.println("Faible vitesse.");
      }
    else if (etatCourant == FAIBLE_)
      {
        etatCourant = MOYEN_;
        System.out.println("Vitesse moyenne.");
      }
    else if (etatCourant == MOYEN_)
      {
        etatCourant = FORT_;
        System.out.println("Grande vitesse.");
      }
    else
      {
        etatCourant = ETEINT_;
        System.out.println("Ventileur eteint.");
      }
  }
}
```

Le code actuel n'est pas facilement modifiable, essentiellement dû à la présence de la suite de conditionnelles (notez qu'un `switch` ne serait pas mieux).

Changer la structure de ce programme à l'aide du pattern State.

Exercice 3 (Billets d'avion)

Vous programmez une application de réservation de billets d'avion. Nous allons considérer qu'un billet réservé peut être modifié ou annulé. Lorsque le billet est confirmé et payé, il n'est plus possible de l'annuler, mais la date reste modifiable. Après le départ du vol concerné, le billet ne peut être modifié ni annulé.

Cette description est volontairement ambigüe et/ou redondante.

1. Modélisez ce programme en UML à l'aide du pattern State.

2. Écrire le programme. Les actions consisteront à afficher à l'écran indiquant si l'opération est possible ou non.

Exercice 4 (Le Monopoly)

Le programme suivant représente une version simplifiée du jeu Monopoly ™.

```
class Joueur
{
  private String nom_;
  private int    argent_;

  public Joueur( String n, int m )
  {
    nom_    = n;
    argent_ = m;
  }

  public String getNom()
  {
    return nom_;
  }

  public int getArgent()
  {
    return argent_;
  }

  public void debit( int m )
  {
    argent_ -= m;
  }

  public void credit( int m )
  {
    argent_ += m;
  }
}

class Propriete
{
  private String nom_;
  private int    prix_;
  private int    droits_;
  private Joueur proprio_;

  public Propriete( String n )
  {
    nom_    = n;
    prix_   = 100;
    droits_  = 10;
  }

  public String getNom()
  {
    return nom_;
  }

  public int getPrix()
  {
    return prix_;
  }
```

```
  public int getDroitsDePassage()
  {
    return droits_;
  }

  public Joueur getProprio()
  {
    return proprio_;
  }

  public void setProprio( Joueur j )
  {
    proprio_ = j;
  }

  void visitePar( Joueur j )
  {
    System.out.print( j.getNom() + " arrive sur " + nom_ );
    if (getProprio() == null)
      {
        System.out.print( " - pas encore vendu\n" + j.getNom() );
        if (j.getArgent() < getPrix())
          {
            System.out.println( " ne peut pas acheter ce terrain." );
          }
        else
          {
            j.debit( getPrix() );
            setProprio( j );
            System.out.println( " achete " + nom_ );
          }
      }
    else
      {
        System.out.println( " - appartient a " + getProprio().getNom() );
        if (j != getProprio())
          {
            j.debit( getDroitsDePassage() );
            getProprio().credit( getDroitsDePassage() );
            System.out.println( getProprio().getNom() + " a maintenant "
                                + getProprio().getArgent() + " euros." );
          }
      }
    System.out.println( j.getNom() + " a " + j.getArgent()
                        + " euros." );
  }
}

public class Monopoly
{
  public static void main( String [] args )
  {
    Joueur j1 = new Joueur( "Tom", 50 );
    Joueur j2 = new Joueur( "   Leo", 500 );
    Propriete prop = new Propriete( "Cours 50 Otages" );
    prop.visitePar( j1 );
    prop.visitePar( j2 );
    prop.visitePar( j1 );
    prop.visitePar( j2 );
    prop.visitePar( j1 );
  }
}
```

```
// Tom arrive sur Cours 50 Otages - pas encore vendu
// Tom ne peut pas acheter ce terrain.
// Tom a 50 euros.
//    Leo arrive sur Cours 50 Otages - pas encore vendu
//    Leo achete Cours 50 Otages
//    Leo a 400 euros.
// Tom arrive sur Cours 50 Otages - appartient a    Leo
//    Leo a maintenant 410 euros.
// Tom a 40 euros.
//    Leo arrive sur Cours 50 Otages - appartient a    Leo
//    Leo a 410 euros.
// Tom arrive sur Cours 50 Otages - appartient a    Leo
//    Leo a maintenant 420 euros.
// Tom a 30 euros.
```

Cependant on s'aperçoit qu'un terrain possède deux états : libre ou possédé (et on pourrait imaginer par la suite des extensions du Monopoly avec d'autres états, par exemple un terrain possédé pour un joueur A mais loué par un joueur B). Modifier le programme avec un pattern State.