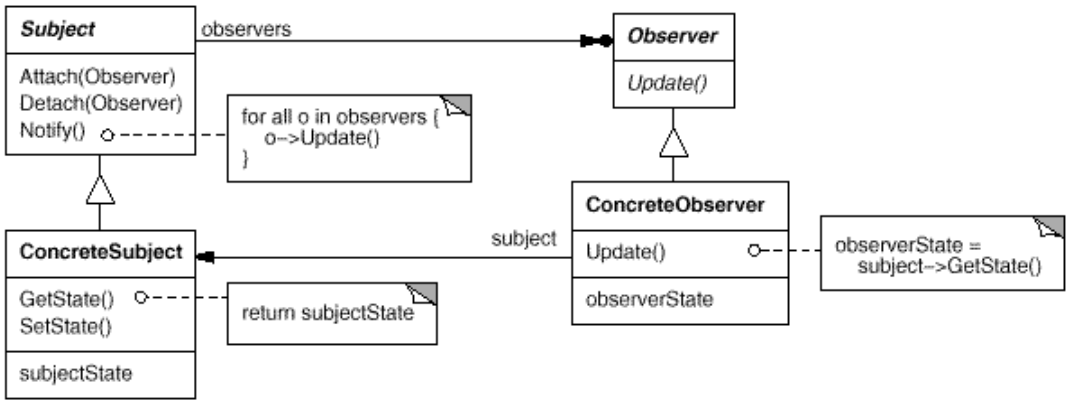
TP2

Thèmes	Lectures **préalables** :
• Pattern Observer • Evénements • modèle MVC	• Pattern Observateur (note 10 et ce chapitre), • Modèle MVC • modèle événementiel

- **Visualisez le sujet en ouvrant dans un navigateur le fichier `index.html` du répertoire qui a été créé à l'ouverture de `tp2.jar` par BlueJ; vous aurez ainsi accès aux applettes et pourrez expérimenter les comportements qui sont attendus.**
- **Soumettez chaque question à l'outil d'évaluation *jnews/junit3* pendant et après le tp, puis rendez le tp avant la date limite grâce à l'outil *jnews/depot*.**

Pattern Observateur/Observé

Soit le Pattern Observateur en notation UML selon LA référence en pattern : *"Design Patterns Elements of Reusable Object-Oriented Software" , Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (GOF) , ed. Addison-Wesley, 1995. (existe en français)*



En Java, le paquetage **`java.util`** implémente ce Pattern.

Il propose la classe `Observable` pour "Subject" du diagramme ci dessus et l'interface `Observer` (même nom dans le diagramme ci dessus) (lire leur javadoc dans la documentation JAVA).

Les participants :

- L'observé : la classe **Subject** ou **java.util.Observable**
- L'observateur : ici l'interface **Observer** ou **java.util.Observer**
- L'observé concret : la classe **ConcreteSubject** qui hérite de Observable
- L'observateur concret : la classe **ConcreteObserver** , qui implémente l'interface **Observer**, et qui utilise une référence du sujet **ConcreteSubject** qu'il observe et réagit à chaque mise à jour

---

Premier exemple d'implantation de ce Pattern en Java .

Classes retenues et proposées dans le paquetage `question1` :

La classe **ConcreteSubject** hérite de **java.util.Observable** (l'observé) et gère une liste de chaines (String), chaque modification de cette liste - introduction d'une nouvelle chaîne - (cf. méthode insert) engendre une notification aux observateurs en passant la nouvelle chaîne en paramètre..

La classe **ConcreteObserver (observateur)** à chaque notification, affiche cette nouvelle chaîne et mémorise l'origine des notifications (attribut "senders") et les paramètres transmis (attribut "parameters").

La mémorisation du notifiant et du paramètre transmis utilise deux piles (java.util.Stack<T>), **senders** et **arguments**, accessibles de l'"extérieur" par les méthodes "**public Stack<Observable> senders(){...}" et "public Stack<Observable> parameters(){...}"**

Pour cette première question, nous souhaitons développer une classe de tests afin de "vérifier" le fonctionnement de l'implantation de ce Pattern,

Quelques exemples de "validation" par assertions :

Vérifier que lors d'une notification, **TOUS** les observateurs ont bien été informés,
Vérifier que les arguments ont bien été **transmis**,
Vérifier que le **notifiant est le bon** ...etc ....

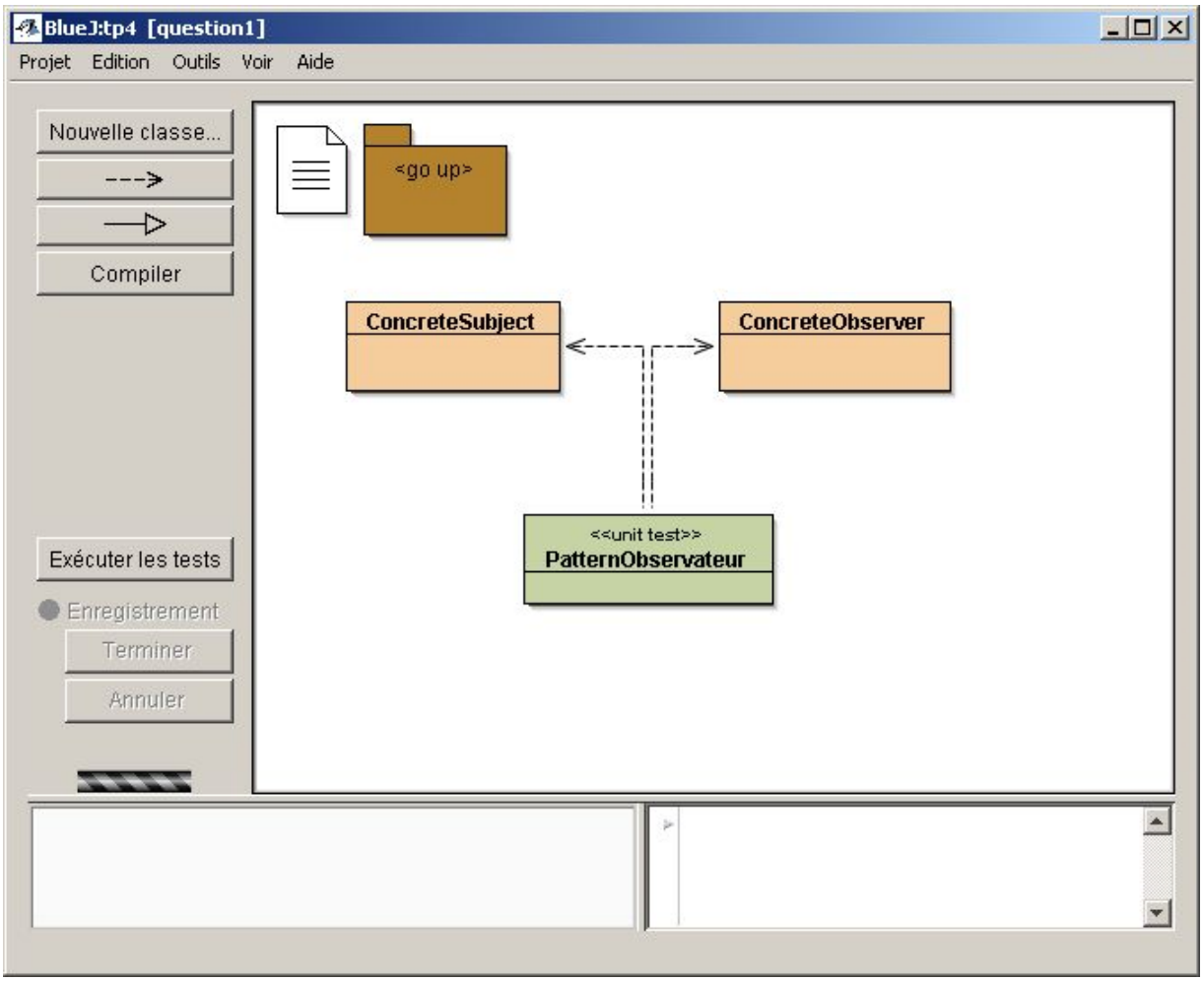
Extrait du code de vérification : classe **"class PatternObservateurTest "**

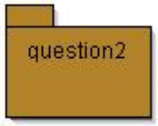
```
ConcreteSubject  list = new ConcreteSubject();       // création d'un "observé" composé d'une
ConcreteObserver observer = new ConcreteObserver();  // création d'un observateur
list.addObserver(observer);                          // ajouter cet observateur à la liste
list.insert("il fait beau, ce matin");               // modification de cette liste, l'observ

// vérification :
assertFalse(observer.senders().empty());                          // elle ne doit pas être v
assertEquals(list,observer.senders().pop());                      // est-ce le bon émetteur
assertEquals("il fait beau, ce matin",observer.arguments().pop()); // le paramètre reçu est-i
list.insert("super !!, je prends mon imperméable");
```

Un exemple de test avec BlueJ: **vérification qu'un observateur est bien notifié** avec le paramètre bien reçu.

Complétez les 3 méthodes de test de la classe "PatternObservateurTest"





Introduction aux évènements de l'AWT

(paquetage java.awt.event, évènements engendrés par une instance de la classe javax.swing.JButton)

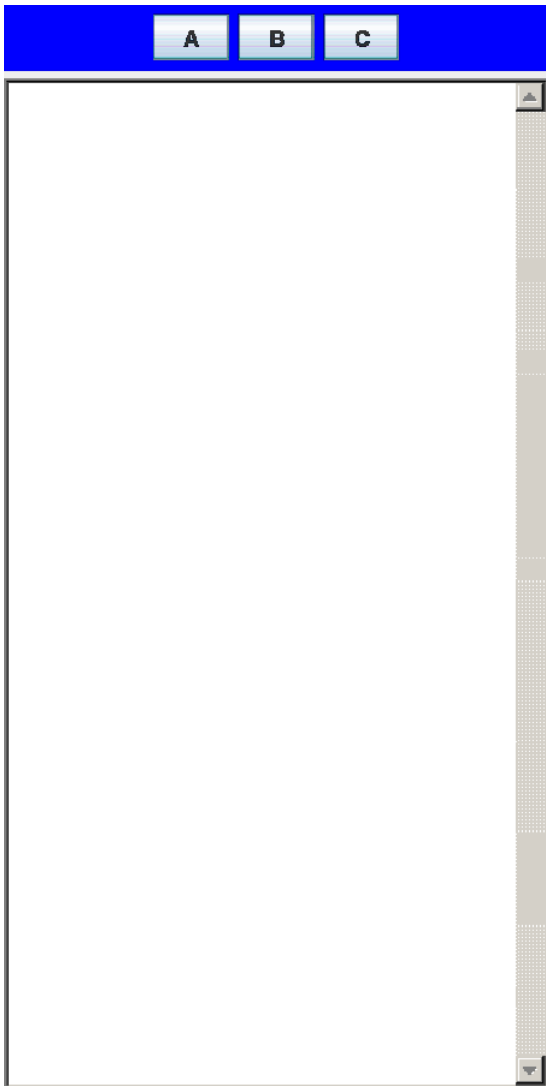
En java, les api AWT ou SWING utilisent le pattern Observateur pour la gestion des évènements, seuls les noms des méthodes diffèrent. Les notifications sont ici engendrées par un changement d'état de l'interface graphique : un clic sur un bouton, un déplacement de souris, etc...

Exemple : la classe ***Observable*** "est remplacée par" la classe ***javax.swing.JButton***

- la méthode ***addObserver(Observer o)*** "correspond à" ***addActionListener(ActionListener l)***
- la méthode ***notifyObservers(Object arg)*** "est remplacée par" ***actionPerformed(ActionEvent ae)***

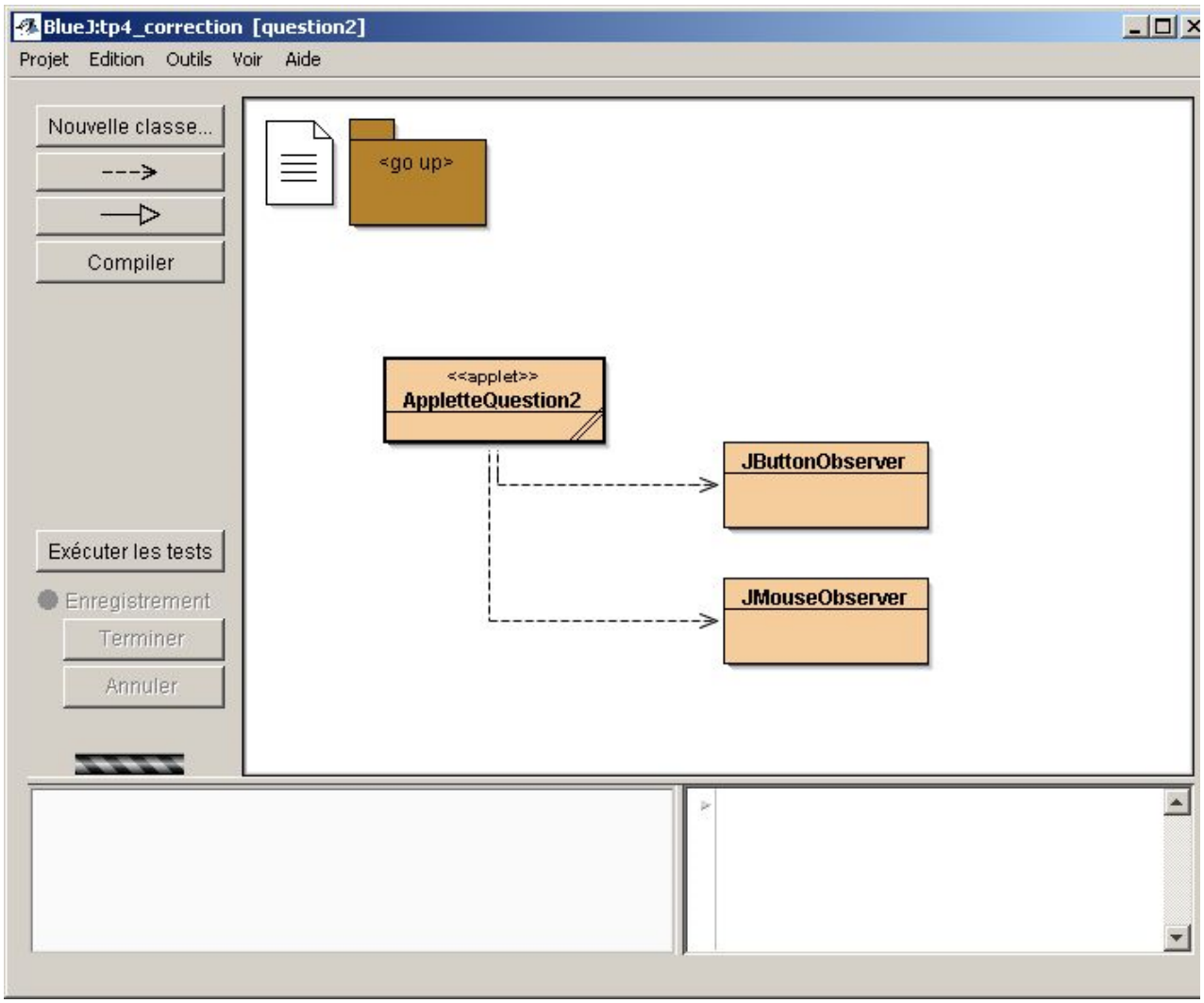
l'interface ***Observer*** "est remplacée par" l'interface ***java.awt.event.ActionListener***

Question 2.1) A chaque clic, un ou plusieurs observateurs sont réveillés : essayez !



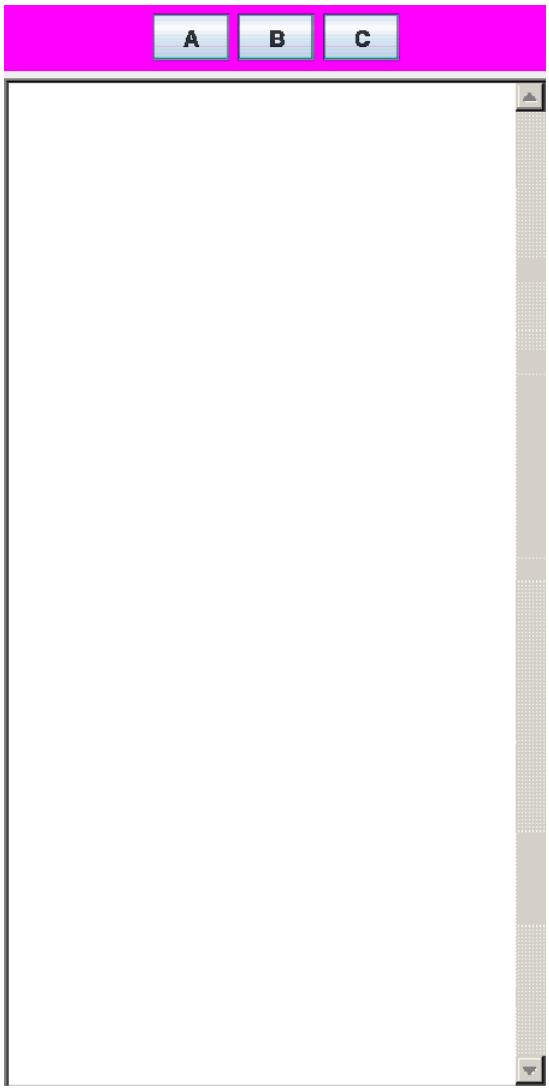
Le bouton A a 3 observateurs ***(jbo1,jbo2 et jbo3)***.
Le bouton B a 2 observateurs ***(jbo1 et jbo2).***
Le bouton C a 1 observateur ***(jbo1).***

Compléter les classes JButtonObserver puis AppletteQuestion2, afin d'obtenir le même comportement et les mêmes traces.
JButtonObserver doit être un ActionListener (voir la javadoc de cette classe).



---

question2-2) : Compléter la classe JMouseObserver :



Le bouton A a 1 observateur de souris ***(jmo1)***
Le bouton B a 1 observateur de souris ***(jmo2).***
Le bouton C a 1 observateur de souris ***(jmo3).***

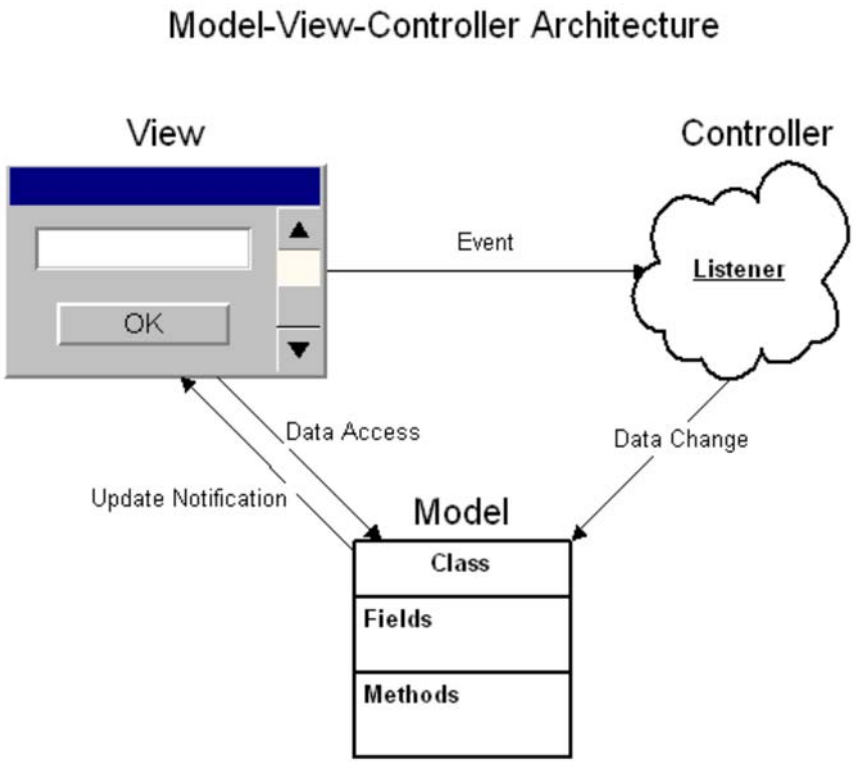
Cette fois :

- la méthode ***addObserver*** est remplacée par ***java.awt.event.addMouseListener***,
- la méthode ***notifyObservers()*** est remplacée par ***mouseXXXXX(MouseEvent ae)***,
- l'interface ***Observer*** est remplacée par l'interface ***java.awt.event.MouseListener***.

Compléter la méthode mouseEntered afin d'obtenir le même comportement et les mêmes traces.

Pour pouvoir tester votre applette, il faut ajouter le parametre NAME=mouse VALUE=oui .

Le modèle MVC



Selon le "pattern MVC" (Modèle-Vue-Contrôleur)

- Le Modèle contient la logique et l'état de l'application, il prévient ses observateurs lors d'un changement d'état.
- La Vue représente l'interface utilisateur.
- Le Contrôleur assure la synchronisation entre la vue et le modèle.

Question 3.1) Développez une application de type calculette à pile, selon le paradigme MVC.

L'évaluation d'une expression arithmétique peut être réalisée par l'usage d'une pile d'entiers.

Par exemple l'expression **3 + 2** engendre la séquence :

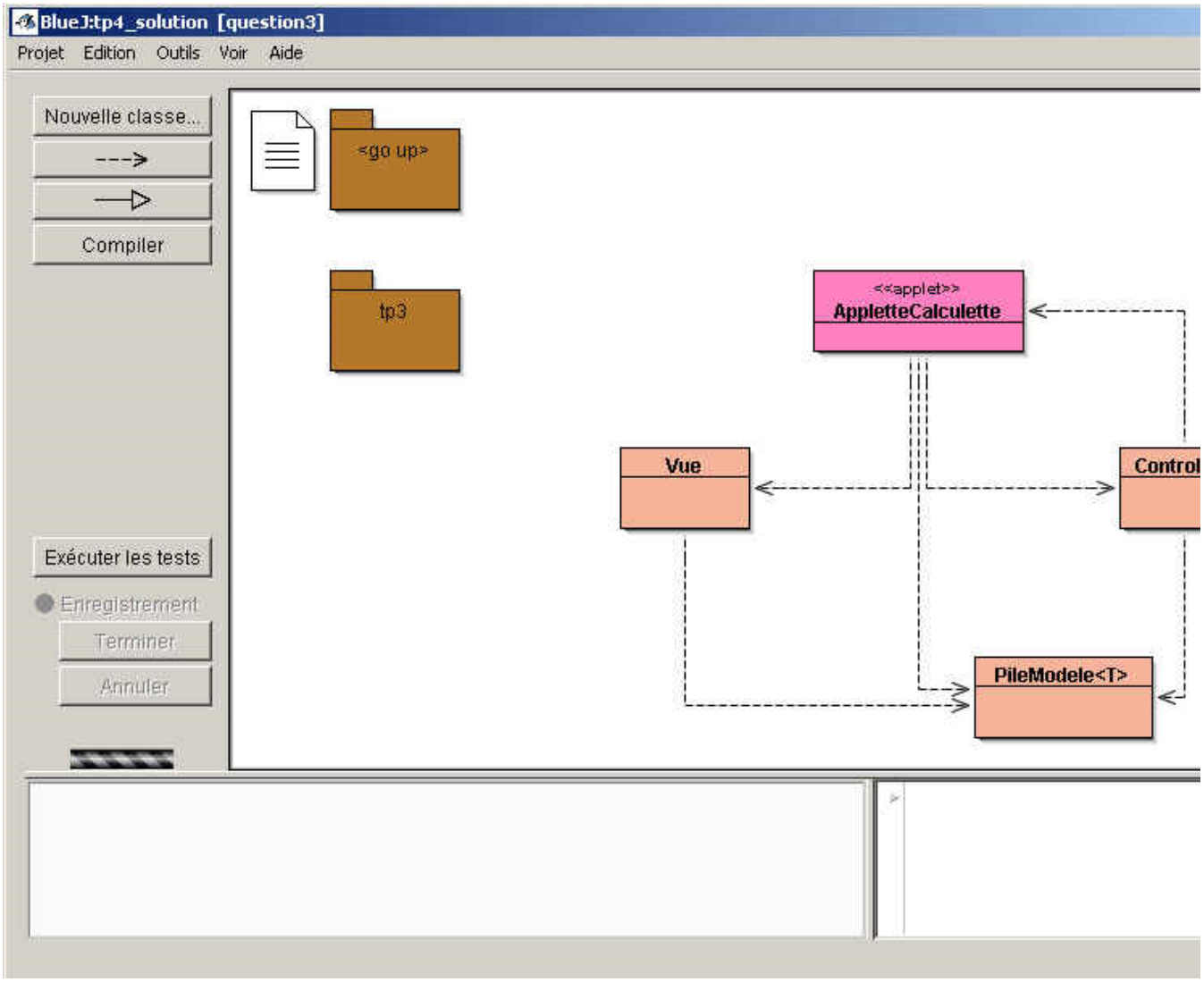
empiler(3);
empiler(2);
empiler(depiler()+depiler()) ;

de même que l'expression **3 + 2 * 5** correspond à la séquence**: empiler(3);empiler(2);empiler(5); empiler(depiler()*depiler(); empiler(depiler()+depiler())**

Attention ! La pile ne doit pas être modifiée en cas de division par zéro.

L'architecture logicielle induite par l'usage du paradigme MVC nous donne :

- Le Modèle est une pile (classe **PileModele<T>**).
  Le Modèle lors d'un changement d'état prévient ses observateurs.
- La Vue correspond à l'affichage de l'état de la pile (classe **Vue**).
  La vue s'inscrit auprès du Modèle lors de l'appel du constructeur d'une Vue; à chaque notification, la vue s'enquiert de l'état du modèle et l'affiche.
- Le Contrôleur gère les évènements issus des boutons +, -, *, /,[] (classe **Controleur**).
  Le contrôleur gère localement les écouteurs (Listener) des boutons de l'IHM; notons que la gestion des boutons utilise une architecture MVC.
- L'applette crée et assemble le modèle, la vue et le contrôleur (classe **AppletteCalculette**).



Une des implémentations des piles issue du tp1 est ici installée dans le package tp3 (!?). Proposer l'implémentation des classes PileModele<T> et Contrôleur.

Selon "MVC", la classe **PileModele<T>** hérite de la classe Observable et implémente PileI<T>; à chaque changement d'état ou modification de la pile, les observateurs inscrits seront notifiés.

La pile2 du tp1, sans modification, est utilisée; seules certaines méthodes seront redéfinies, enrichies, décorées ...

La classe **Controleur** implémente les actions (évènements engendrés par l'utilisateur); à chaque opération souhaitée, le contrôleur altère les données du modèle de la pile; celle-ci, à chaque occurrence d'un changement d'état, prévient ses observateurs; la vue est en un.

Une AppletteVue au comportement souhaité

Soumettez cette question à JNEWS avant de poursuivre.

Question 3.2) à ne pas soumettre à JNEWS

Modification de l'application respectant le principe "MVC".

Ajouter cette nouvelle Vue au modèle, et vérifiez que seule la classe Applette est concernée par cet ajout.

```
public class Vue2 extends JPanel implements Observer {
  private JSlider jauge;
  private PileModele<Integer> pile;

  public Vue2(PileModele<Integer> pile) {
    super();
    this.pile = pile;
    this.jauge =  new JSlider(JSlider.HORIZONTAL, 0, pile.capacite(), 0);
    this.jauge.setValue(0);
    setLayout(new FlowLayout(FlowLayout.CENTER));
    this.jauge.setEnabled(false);
    add(this.jauge);
    setBackground(Color.magenta);
    pile.addObserver(this);
  }

  public void update(Observable obs, Object arg) {
    jauge.setValue(pile.taille());
  }

}
```

Essayez l'applette seulement en local.
