TP4

Thèmes

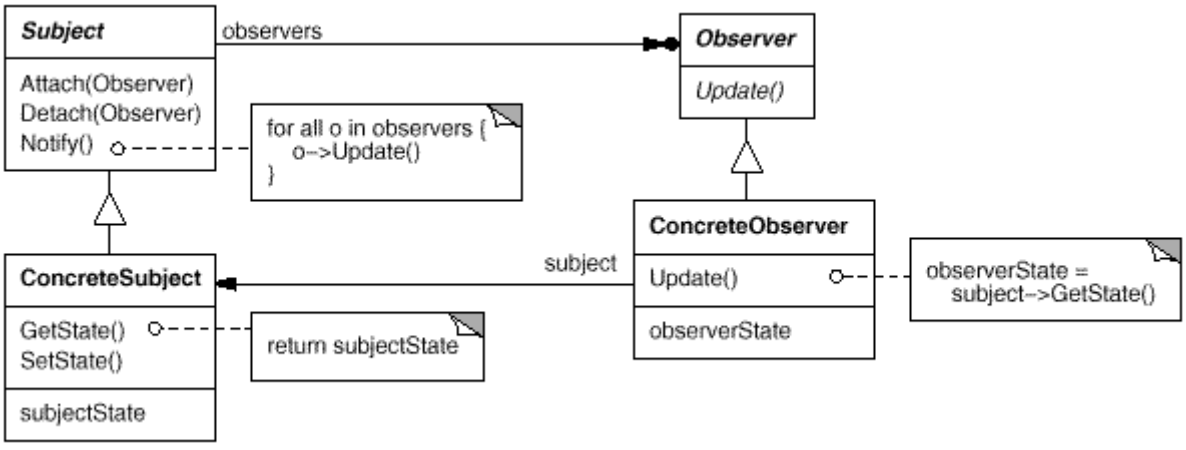
- Java_III
- Patterns Observer
- Evénements
- modèle MVC
  en mp3: MVC song de James. Dempsey, les paroles

lecture **préalable** du Pattern Observateur (note 10 et ce chapitre), du Modèle MVC et du modèle événementiel

Rappel : Avec Konqueror sous Linux, les applettes ne seront visibles qu'après avoir autorisé le langage Java (Outils/Configuration_HTML/Java et rechargement de la page).

(Test du pattern Observateur/Observé)

Soit le Pattern Observateur en notation UML selon **la** référence : *Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides Design Patterns Elements of Reusable Object-Oriented Software Addison-Wesley, 1995.* En Java, le paquetage `java.util` implemente ce Pattern et propose la classe `Observable` pour Subject et l'interface `Observer` (lire leur javadoc).



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**

1. L'observateur concret :la classe **ConcreteObserver** utilise une référence du sujet concret qu'il observe et réagit à chaque mise à jour

Pour cette question, nous souhaitons développer une classe de tests afin de "vérifier" le bon fonctionnement de ce Pattern,

Quelques exemples de "validation", d'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 ....

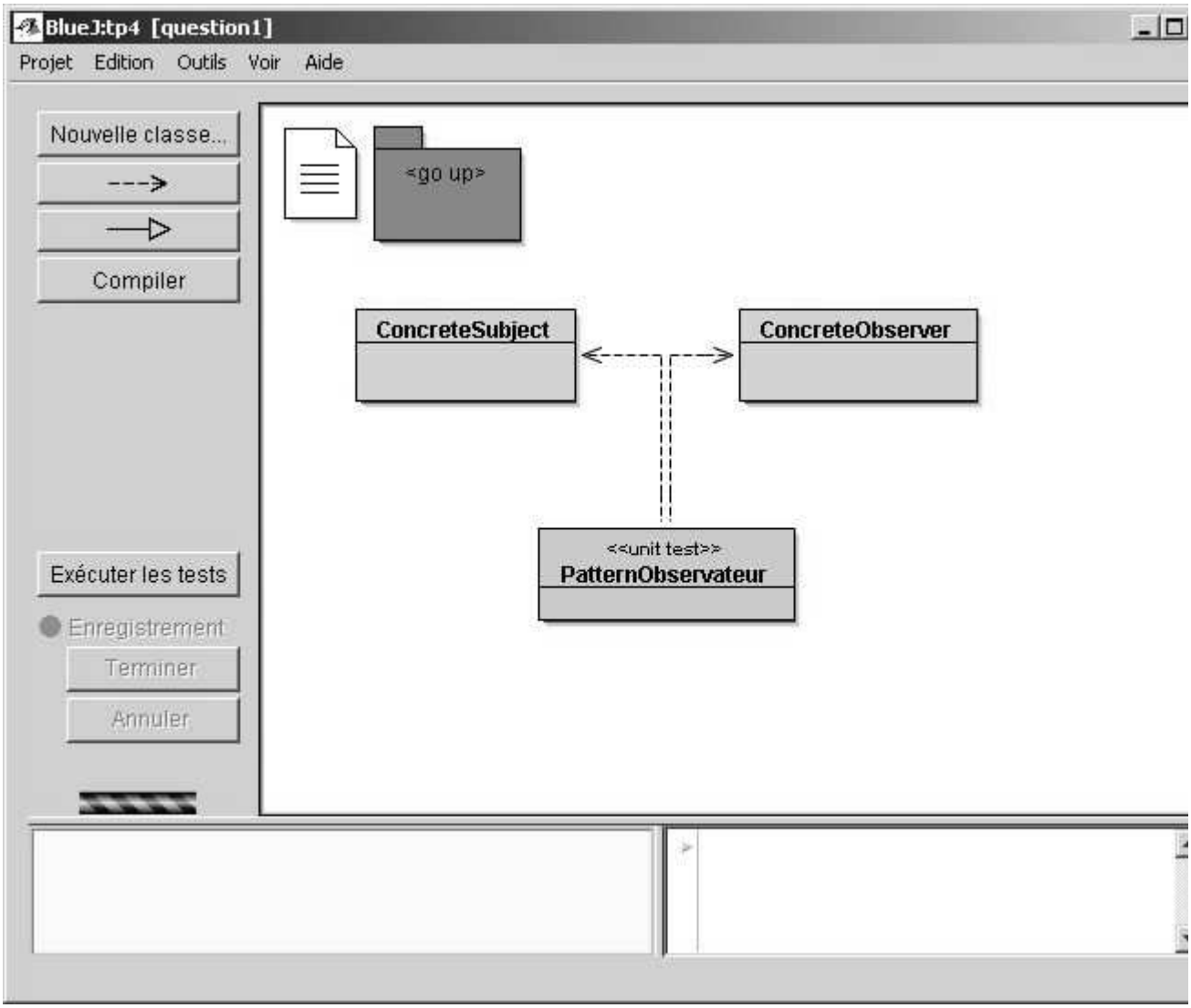
Un exemple de test avec BlueJ: **vérification qu'un observateur est bien notifié** avec le paramètre bien reçu

prémisses et classes retenues:
la classe **ConcreteSubject** gère une liste de noms, chaque modification de cette liste engendre une notification
la classe **ConcreteObserver** se contente, à chaque notification, d'afficher cette liste et de mémoriser l'origine des notifications et les paramètres transmis.
La mémorisation du notifiant et du paramètre transmis utilise deux piles (java.util.Stack<T>), **senders** et **arguments**, accessibles de l'"extérieur"

```
ConcreteSubject  list = new ConcreteSubject();       // création d'une
ConcreteObserver observer = new ConcreteObserver();  // création d'un o
list.addObserver(observer);                          // ajouter cet obs
list.insert("il fait beau, ce matin");               // modification de

// vérification :
assertFalse(observer.senders().empty());                          // e
assertEquals(list,observer.senders().pop());                      // e
assertEquals("il fait beau, ce matin",observer.arguments().pop()); // l
list.insert("super !!, je prends mon imperméable");
```

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



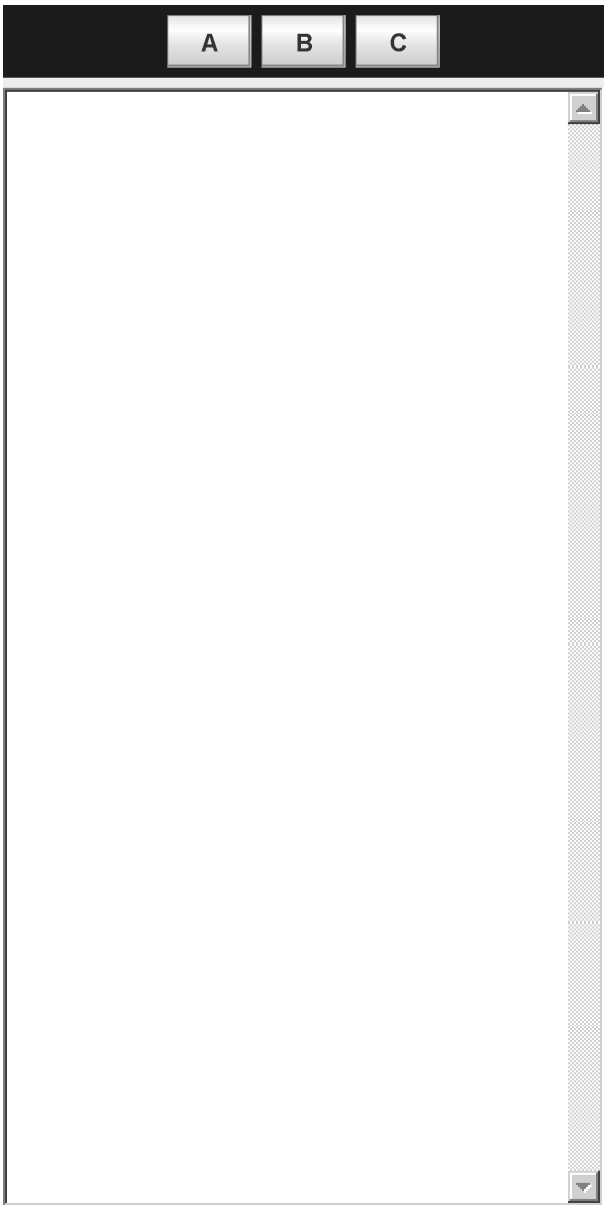
Introduction aux événements de l'AWT (paquetage java.awt.event, événements engendrés par une instance de la classe java.awt.JButton)

En java, la gestion des évènements utilise le pattern Observateur, 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...

la méthode ***addObserver*** "est remplacée par" ***java.awt.event.addActionListener***
la méthode ***notifyObservers()*** "est remplacée par" ***actionPerformed (ActionEvent ae)***

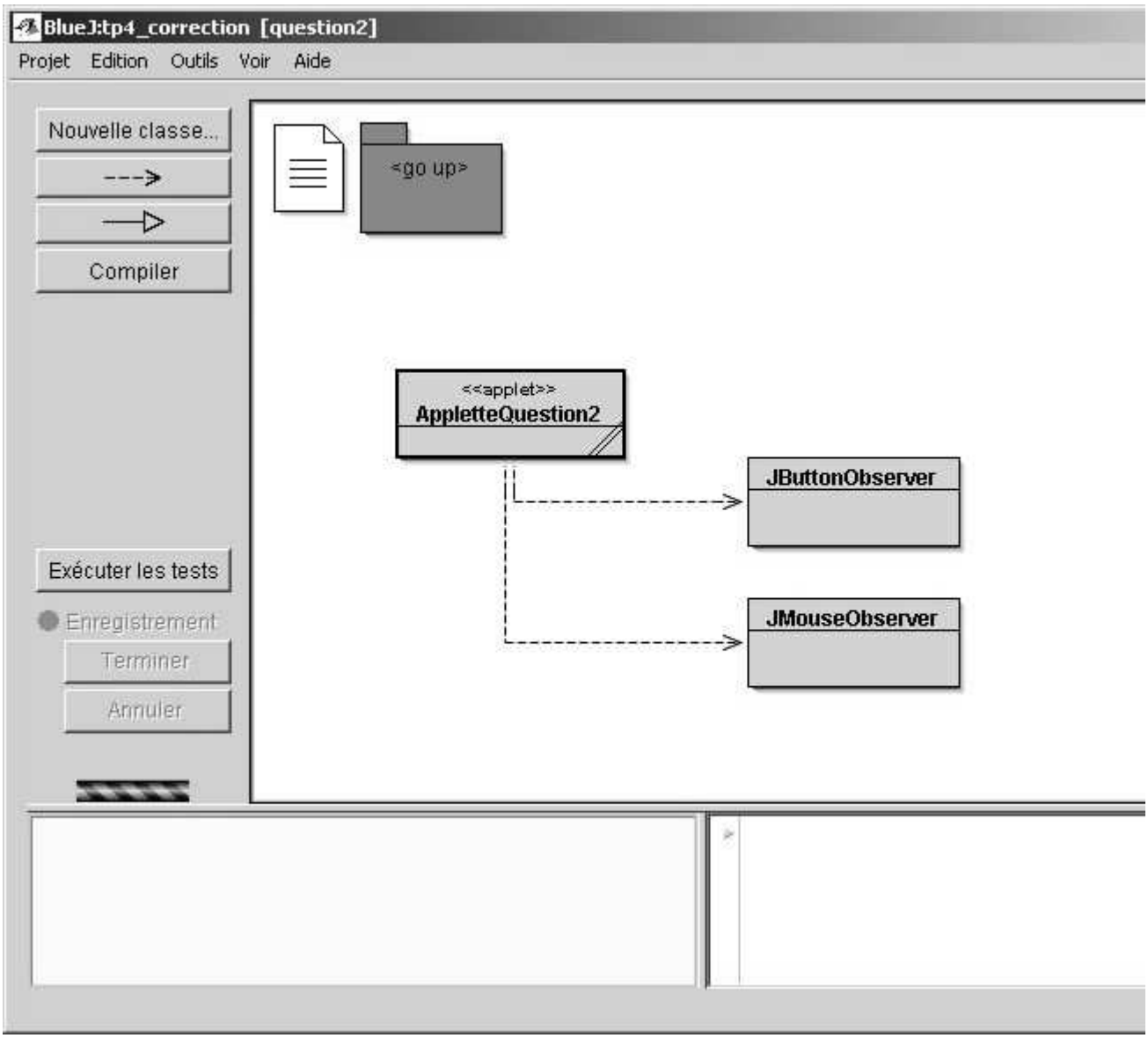
l'interface ***Observer*** "est remplacée par" l'interface ***java.awt.event.ActionListener***
la classe ***Observable*** "est remplacée par" la classe ***java.awt.JButton***

A chaque clic, un ou plusieurs observateurs sont réveillés

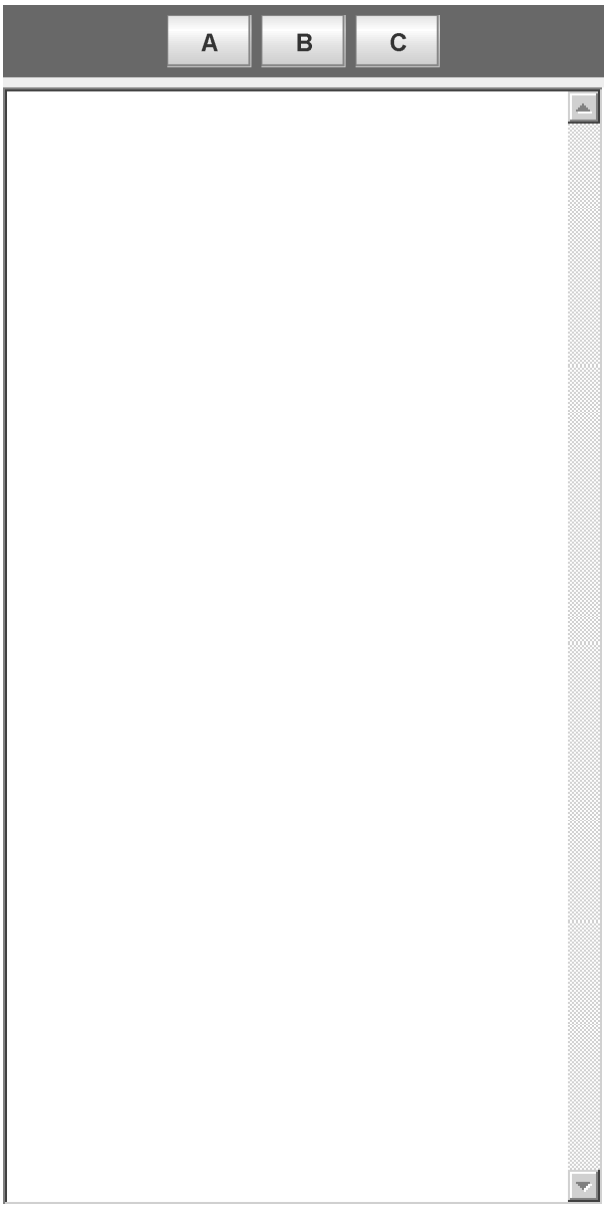


Le bouton A a 3 observateurs***(jbo1,jbo2 et jbo3***)
le bouton B a 2 observateurs ***(jbo1 et jbo2)***
le bouton C a 1 observateur ***(jbo1)***

Compléter les classes AppletteQuestion2 et JButtonObserver afin d'obtenir le même comportement et les mêmes traces



question2-2) : Complétez la classe JMouseObserver



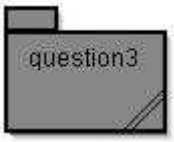
Le bouton A a 1 observateur de souris***(jmo1)***
Le bouton B a 1 observateur de souris***(jmo2)***
Le bouton C a 1 observateur de souris***(jmo3)***

cette fois

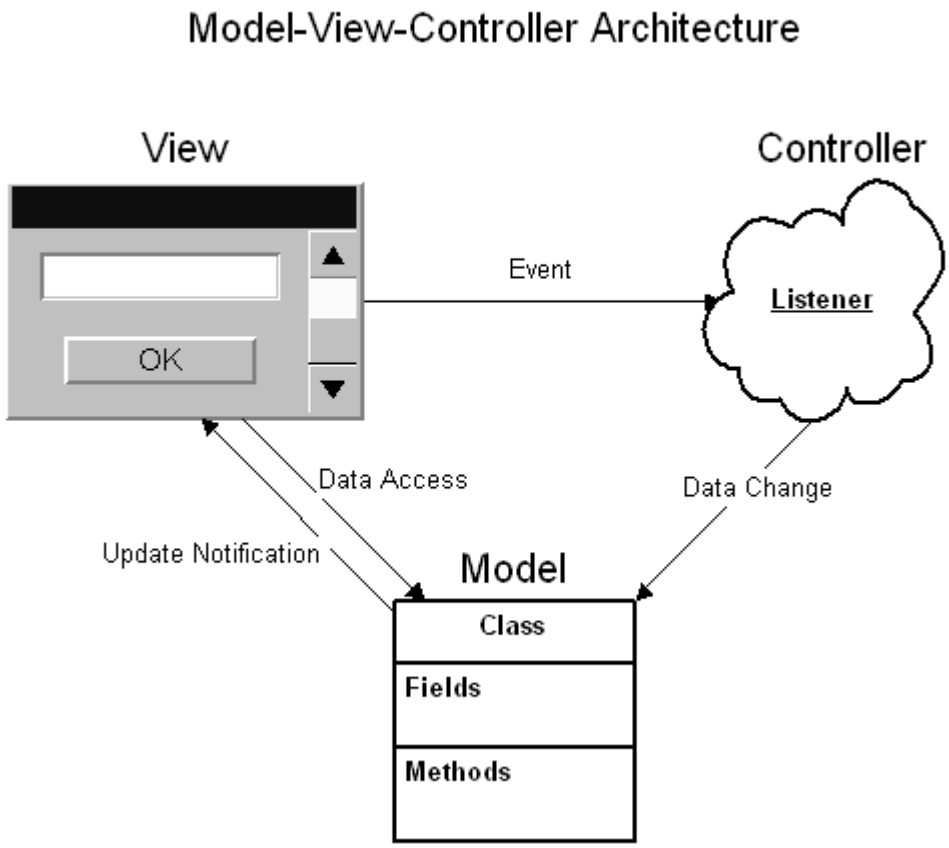
la méthode ***addObserver*** est remplacée par ***java.awt.event.addMouseListener***
la méthode ***notifyObservers()*** est remplacée par ***mouseXxxxxed(MouseEvent ae)***
l'interface ***Observer*** est remplacée par l'interface ***java.awt.event.MouseListener***

Xxxxxed implémentant une des méthodes de votre choix : mouseClicked ou mouseEntered ou mouseExited ou mousePressed ou mouseReleased

---



Le modèle MVC



Selon le "pattern MVC" (Modèle Vue Contrôleur)

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

question3-1) 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

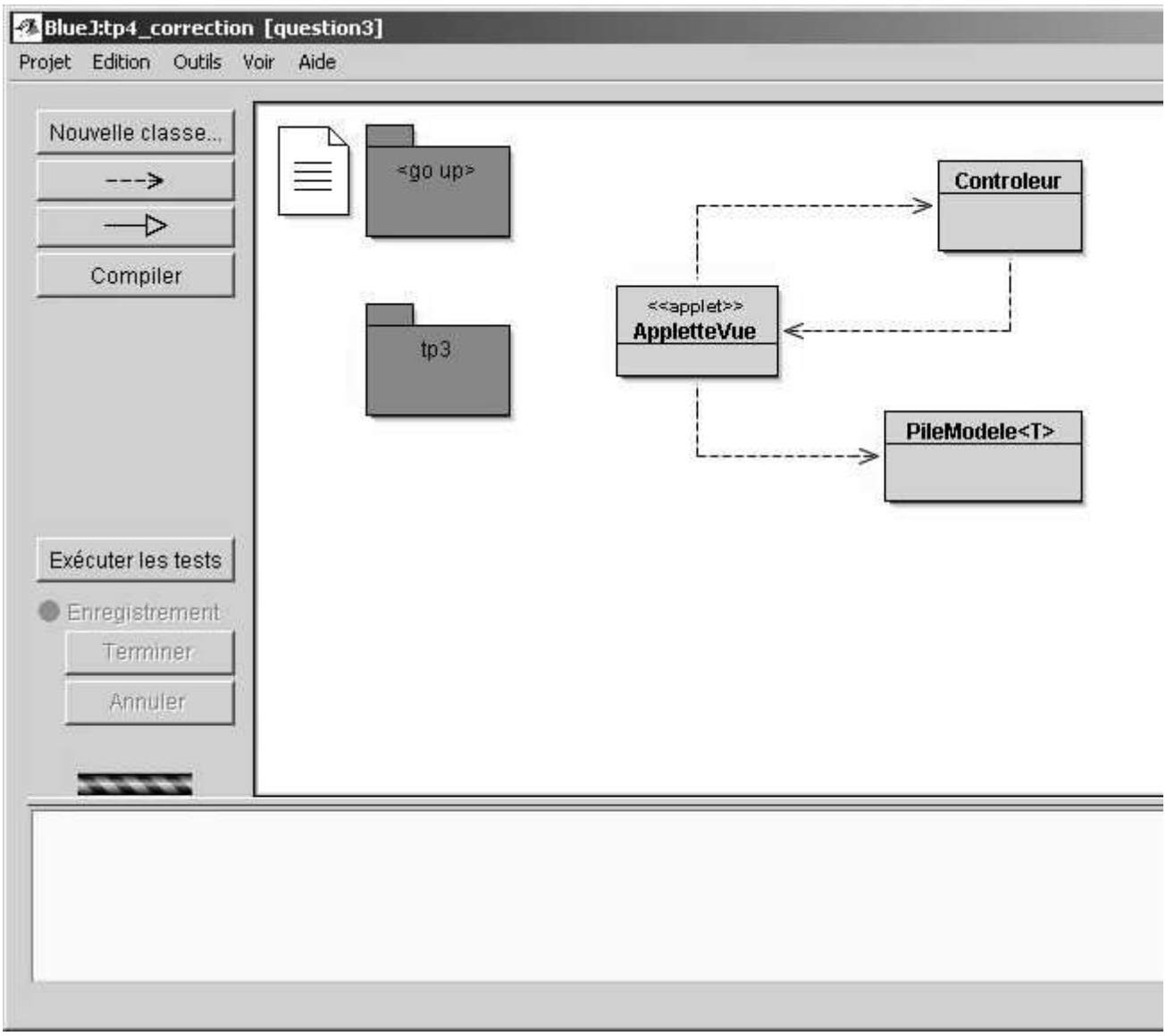
Par exemple l'expression **3 + 2** engendre la séquence :

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());**

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

- Le Modèle est une pile (classe **PileModele<T>**).
- La Vue correspond à l'applette (classe **AppletteVue**).
- Le Contrôleur gère les évènements issus des boutons +, -, *, /,[] (classe **Controleur**)



Une des implémentations des piles issue du tp3, est installée dans le package tp3, proposer l'implémentation des classes PileModele<T> et Controleur

selon "MVC" la classe **PileModele<T>** hérite de la classe Observable et implémente PileI<T>, à chaque changement d'état, modification de la pile les observateurs inscrits seront notifiés. La pile du tp3, sans la modifier, 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 l'occurrence d'un changement d'état prévient ses observateurs, l'applette est en un.

Une AppletteVue au comportement souhaité

AIDE : 2 façons de faire possibles :

1. L'applette est l'observateur de tous les boutons.
   Inconvénient : un grand if else if else ... pour traiter tous les cas, et un gros fichier .java contenant tout le code nécessaire à toutes les actions.
2. Un observateur différent est attribué à chaque bouton.
   Inconvénient : il faut créer une classe par bouton.

N'oubliez pas de faire un submit avant de quitter la salle !