

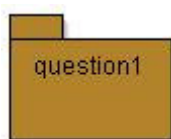
TP4

Thèmes

- Patterns [Observer](#)
- [Événements](#)
- [modèle MVC](#)

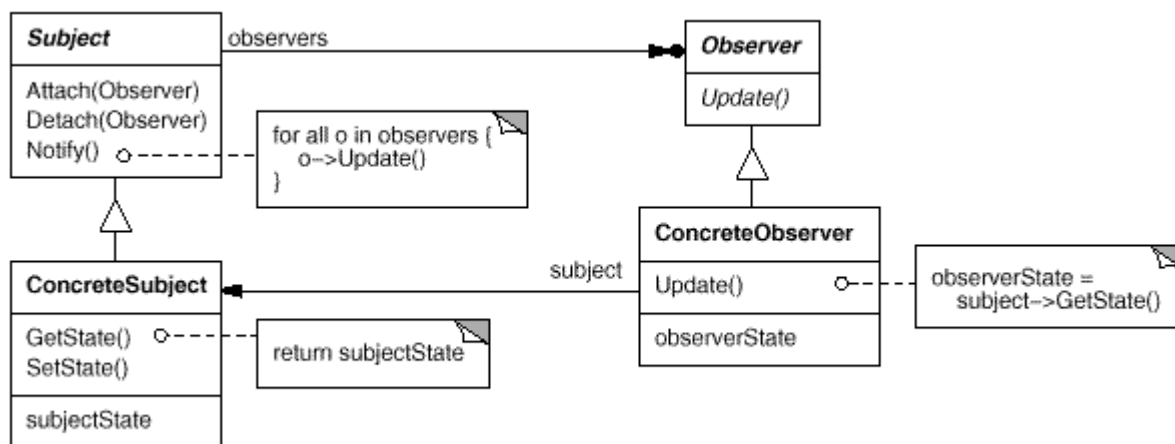
lecture préalable du [Pattern Observateur \(note 10 et ce chapitre\)](#), du Modèle [MVC](#) et du modèle [événementiel](#)

- Visualisez le sujet en ouvrant `index.html` du répertoire qui a été créé à l'ouverture de `tp4.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 `junit3`.



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` implémente 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**
- 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"

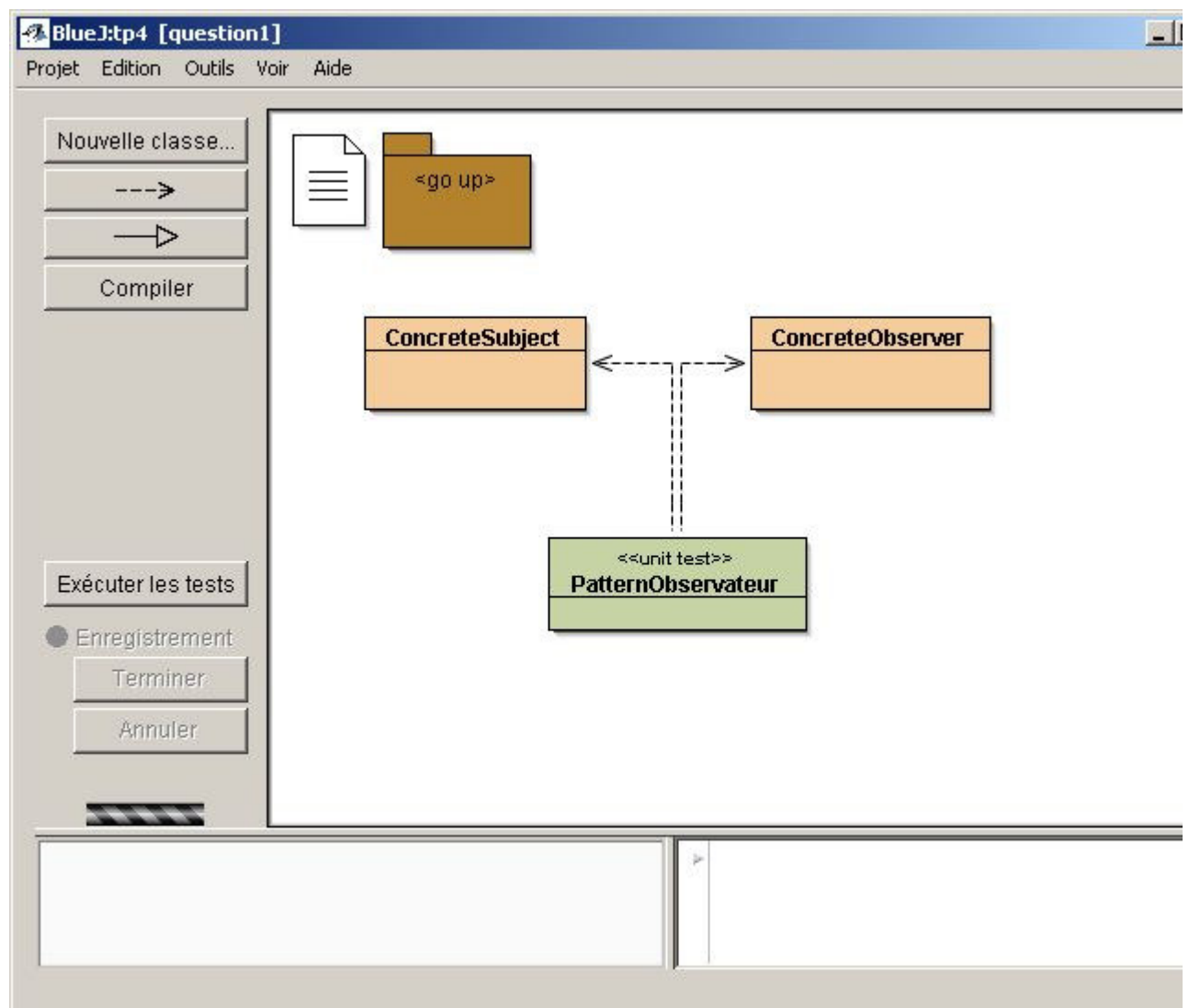
senders : mémorise les émetteurs des notifications

arguments : mémorise les arguments transmis lors d'une notification

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

// "vérification" :
assertFalse(observer.senders().empty());              // la pile senders ne doit pas être
assertEquals(list,observer.senders().pop());          // est-ce le bon émetteur ?
assertEquals("il fait beau, ce matin",observer.arguments().pop()); // le paramètre reçu est-il correct
```

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



question2

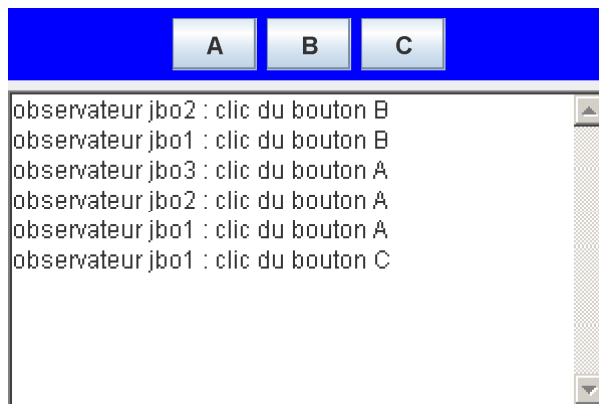
Introduction aux événements de l'AWT

(paquetage `java.awt.event`, événements engendrés par une instance de la classe `javax.swing.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

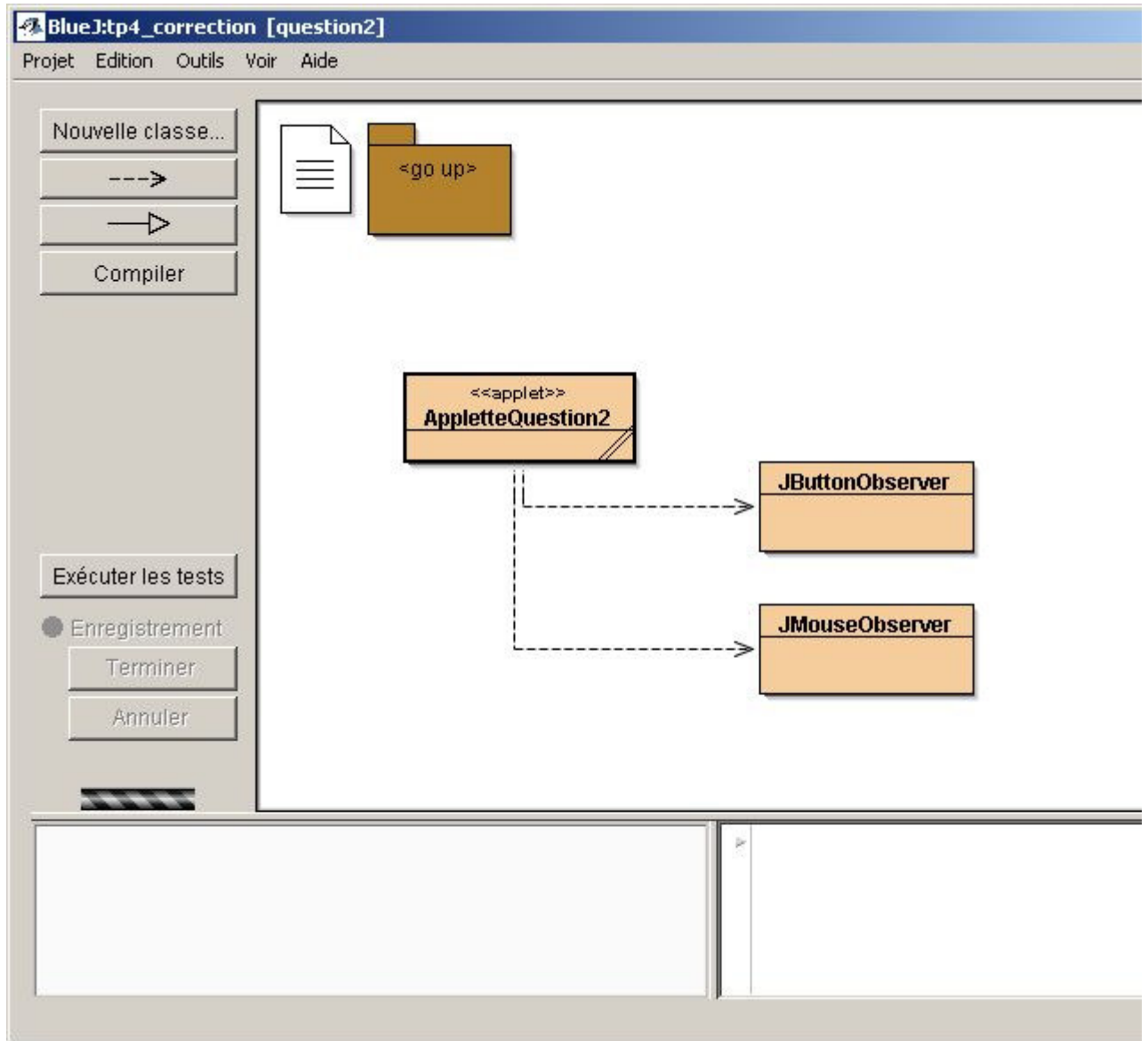


- le bouton A a 3 observateurs (*jbo1, jbo2 et jbo3*)
- le bouton B a 2 observateurs (*jbo1 et jbo2*)
- le bouton C a 1 observateur (*jbo1*)

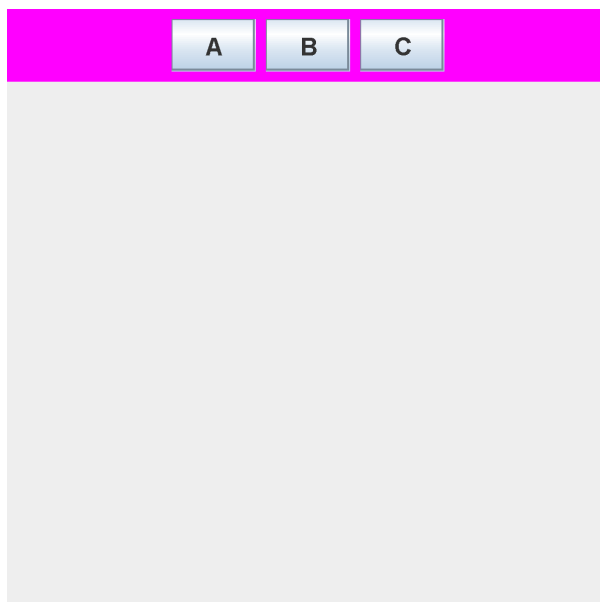
question2

.1) Complétez les classes `JButtonObserver` et `AppletteQuestion2` afin d'obtenir le même comportement et les

mêmes traces



.2) Complétez la classe JMouseObserver pour obtenir le comportement ci-dessous



- 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

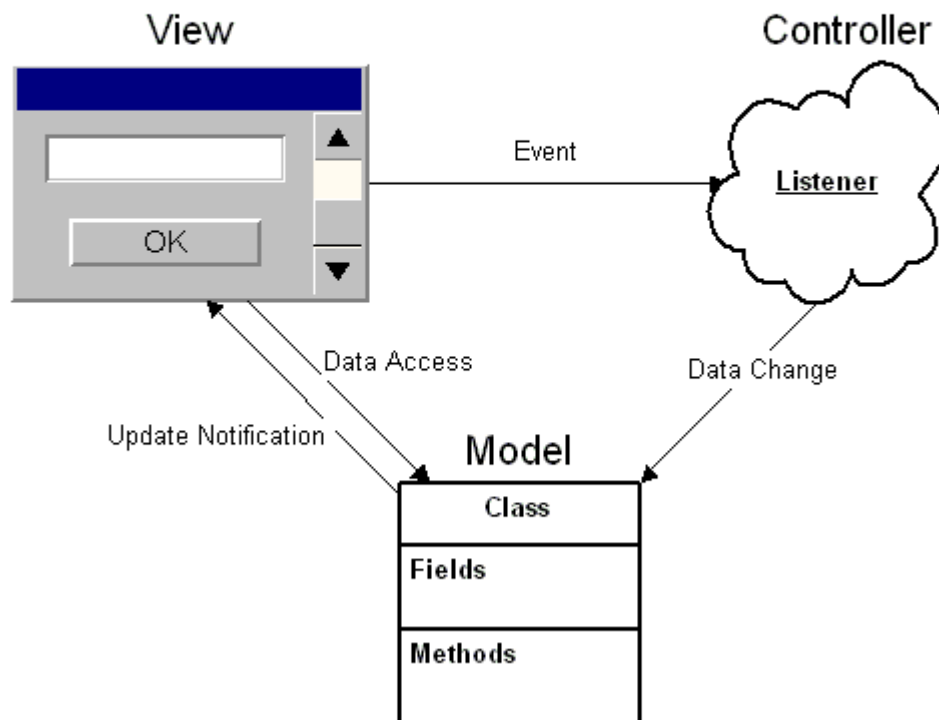
la méthode *addObserver* est remplacée par *java.awt.event.addMouseListener*
 la méthode *notifyObservers()* est remplacée par *mouseXXXXX(MouseEvent ae)*
 l'interface *Observer* est remplacée par l'interface *java.awt.event.MouseListener*

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



Le modèle MVC

Model-View-Controller Architecture



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.

Pour cette question, 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

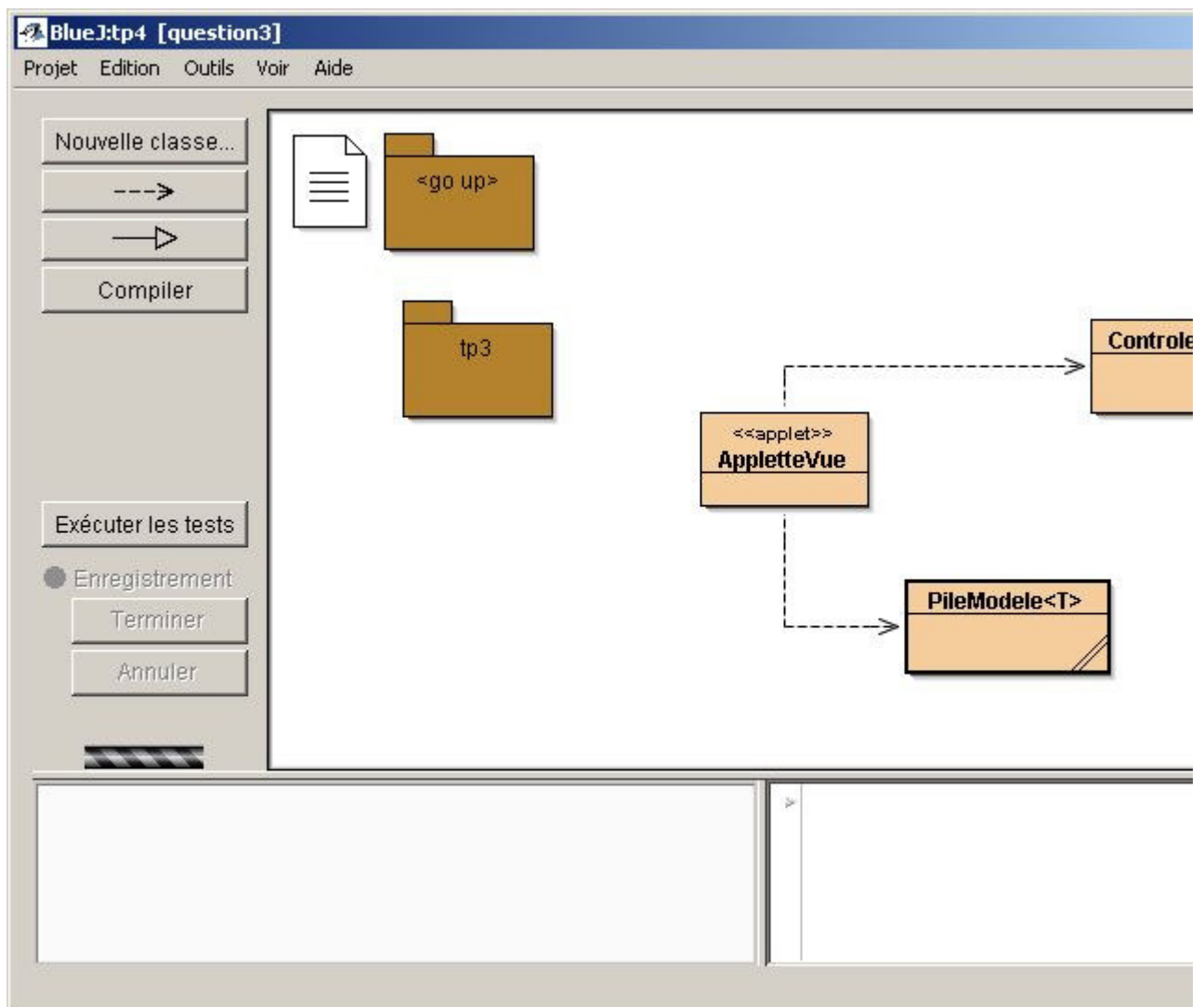
Par exemple l'expression $3 + 2$ engendre la séquence :

```
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.

Complétez l'implémentation des 3 classes.

selon "MVC" la classe **PileModele<T>** hérite de la classe Observable et implémente Pile<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
