Examen

Nom : **Prénom :**

- Il est conseillé de lire complètement le sujet avant de commencer à y répondre !
- Penser à mettre son nom sur le sujet et à le rendre avec la copie puisque certaines réponses peuvent être données directement sur le sujet.
- Barème indicatif :

Exercice	1	2	3	4	5
Points	5	3	2	5	5

Exercice 1 : Clavier virtuel

Les organismes bancaires qui offrent à leurs clients un accès à leurs comptes par Internet ont tendance à proposer un clavier virtuel pour saisir le code en lieu et place du clavier traditionnel. Ceci est fait essentiellement pour lutter contre les outils qui espionnent le clavier. La Caisse d'Épargne, par exemple, propose le clavier virtuel proposé à la figure 1. L'utilisateur clique sur les chiffres pour composer son code (une étoile est ajoutée dans la zone code confidentiel pour chaque chiffre), puis il clique sur le bouton « VALIDER ». En cliquant sur « CORRIGER », il remet à zéro le code confidentiel.

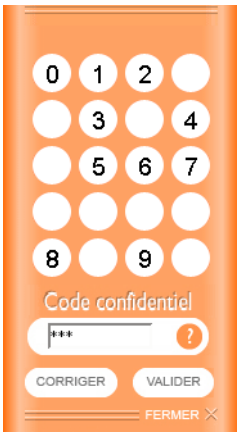

FIGURE 1 – Clavier virtuel

1.1 *Aspects graphique de cette application.* On s'intéresse dans un premier temps à la manière de représenter graphiquement en Java/Swing cette interface. Aucune ligne de code n'est demandée. On ne traitera ni la partie qui permet de fermer la fenêtre (et donc le clavier), ni le bouton « ? » qui donne accès à une page d'aide.

1.1.1 Indiquer les composants graphiques élémentaires à utiliser.

1.1.2 Indiquer comment faire pour agencer ces composants graphiques élémentaires.

1.2 *Action sur le bouton CORRIGER.* Lorsque l'utilisateur clique « CORRIGER », le texte du code confidentiel est remis à vide. Indiquer comment faire pour programmer ce comportement en Java/Swing. On indiquera le principe et on pourra s'appuyer sur du pseudo-code Java.

1 Implantation du patron de conception Observateur

L'objectif de cet exercice est de voir le patron de conception Observateur [1] et le principe de l'implantation qui en est faite dans les API Java [2].

La figure 2 décrit le diagramme de classe du patron Observateur. Un sujet (`Sujet`) peut être observé par plusieurs observateurs (`Observateur`). Les opérations `inscrire` et `annuler` permettent respectivement d'ajouter un nouvel observateur à un sujet ou de supprimer un observateur inscrit. Lorsque le sujet change, il avertit chaque observateur du changement en exécutant sa méthode `mettreAJour`. Cette méthode prend deux paramètres, le premier correspondant au sujet qui a changé (et qui a donc appelé cette méthode), le second à une information transmise par le sujet à l'observateur.

Dans l'exemple, quand la méthode `setÉtat` de `SujetConcret` est appelée, les observateurs sont avertis et leur méthode `mettreAJour` est exécutée. L'observateur peut alors utiliser l'information transmise (`param`) ou interroger directement le sujet (`source`) pour réaliser son traitement.

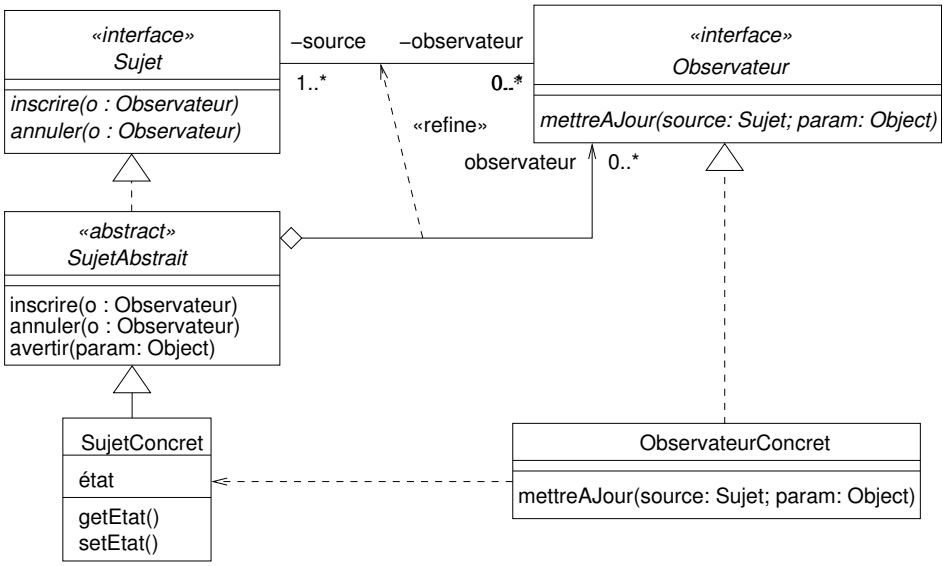

FIGURE 2 – Le patron de conception Observateur selon GoF

Exercice 2 : Compréhension du diagramme de classe

Dans un premier temps, regardons les choix faits sur le diagramme de classe proposé (figure 2).

2.1 Sur le diagramme de classe, la relation entre les interfaces `Sujet` et `Observateur` n'est que conceptuelle car un sujet n'a pas à révéler ses observateurs. Cette relation ne se traduira pas par du code dans l'interface `Sujet` (ni dans l'interface `Observateur`). Cependant, un sujet doit gérer une collection d'observateurs qui pourront être avertis en cas de changement du sujet. Ainsi, la relation sera en fait définie entre `SujetAbstrait` et `Observateur`.

1. Le patron Observateur est décrit dans Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (trad. Jean-Marie Lasvergères), Design Patterns - Catalogue de modèles de conceptions réutilisables, 1999.
2. Il s'agit de la classe `java.util.Observable` et l'interface `java.util.Oberserver`.

Expliquer pourquoi cette relation ne peut pas se traduire par du code dans l'interface `Sujet`.

2.2 La classe `SujetAbstrait` dont toutes les méthodes sont définies est abstraite. Pourquoi ?

2.3 Indiquer s'il est possible ou non de définir un constructeur sur une classe abstraite.

2.4 Indiquer, en les justifiant, les droits d'accès à mettre sur les différents attributs et opérations présents sur le diagramme de classe de la figure 2.

2.5 Expliquer l'intérêt de définir l'interface `Sujet` en plus de la classe `SujetAbstrait`.

2.6 Expliquer pourquoi le type du paramètre `param` des méthodes `avertir` et `mettreAJour` est du type `Object`.

Exercice 3 : Les interfaces `Observateur` et `Sujet`

Intéressons nous aux deux interfaces `Observateur` et `Sujet`.

3.1 Indiquer pourquoi, pour une interface, il est essentiel de donner les commentaires de documentation de ses différentes méthodes.

3.2 Écrire les deux interfaces `Observateur` et `Sujet`. Même s'ils sont importants, on ne donnera pas les commentaires de documentation.

Exercice 4 : La classe abstraite `SujetAbstrait`

La classe `SujetAbstrait` doit stocker les observateurs inscrits. On impose deux contraintes :

- le type de l'attribut contenant les observateurs doit être `java.util.Collection`.
- même si la méthode `mettreAJour` d'un observateur lève une exception, les autres observateurs inscrits doivent être avertis.

Écrire la classe `SujetAbstrait` en respectant les contraintes ci-dessus.

Exercice 5 : Exemple d'utilisation

Dans cet exercice, nous souhaitons faire de la classe `ListeTab` (listing 1) un sujet observable. Il est important de lire toutes les questions de cet exercice avant de commencer à y répondre.

5.1 Modifier la classe ListeTab (listing 1) pour qu'elle devienne un sujet. On pourra répondre en annotant le listing 1.

5.2 Écrire l'observateur `Trace` qui affiche le dernier élément ajouté ou retiré de la liste observée.

5.3 Écrire un observateur `Moyenne` qui affiche la moyenne des valeurs de la liste observée à chaque changement de cette liste. Dans le cas où la liste est vide, une exception appelée `MoyenneIndéfinieException` sera levée. Cette exception est bien entendu à définir.

5.4 Écrire un programme qui consiste à :

- créer une liste l,
- inscrire un observateur Trace auprès de cette liste l,
- inscrire un observateur Moyenne auprès de la liste l,
- ajouter l'élément 10 dans la liste l en position 0,
- supprimer l'observateur Trace inscrit auprès de la liste l,
- ajouter l'élément 20 dans la liste l en position 1,
- inscrire un observateur Trace auprès de cette liste l,
- supprimer la valeur à la position 0 de la liste l.
- supprimer la valeur à la position 0 de la liste l.

5.5 Donner le résultat de l'exécution du programme précédent.

Listing 1 – La classe ListeTab

```
/** Liste simple utilisant un tableau pour stocker les éléments.
 */
public class ListeTab {
   private double[] elements; // les éléments de la liste
   private int nb;      // la taille de la liste

   /** Construire une liste vide.
    * @param capacite capacité initiale de la liste
    */
   public ListeTab(int capacite) {
      this.elements = new double[capacite];
      this.nb = 0;      // la liste est initialement vide
   }
   /** Obtenir la taille de la liste.
    * @return taille de la liste */
   public int taille() {
      return this.nb;
   }

   /** Obtenir un élément de la liste.
    * @param indice position de l'élément */
   public double item(int index) {
      return this.elements[index];
   }

   /** Supprimer un élément de la liste.
    * @param indice indice de l'élément à supprimer */
   public void supprimer(int index) {
      System.arraycopy(this.elements, index+1,
            this.elements, index, this.nb-index-1);
      this.nb--;
   }

   /** Ajouter un élément dans la liste.
    * @param indice indice où doit se trouver le nouvel élément
    * @param x élément à insérer */
   public void ajouter(int index, double x) {
      if (this.nb >= this.elements.length) { // tableau trop petit !
         // agrandir le tableau : pourrait être plus efficace !
         double[] nouveau = new double[this.nb+3]; // 3 arbitraire
         System.arraycopy(this.elements, 0, nouveau, 0, this.nb);
         this.elements = nouveau;
      }
      // décaler les éléments à partir de index
      System.arraycopy(this.elements, index,
            this.elements, index+1, this.nb-index);
      // ranger le nouvel élément
      this.elements[index] = x;
      this.nb++;
   }
}
```