TP4 – Modèle événementiel Java

Programmation par composants

OBJECTIF : Ce TP est consacré à quelques exercices pratiques sur le patron de conception **Observer** ainsi qu'au **Modèle événementiel de Java**. Il permet surtout de mettre en évidence les trois notions : événement, émetteur, auditeur sur lesquelles ce modèle est fondé.

Cet énoncé, ainsi que toutes les ressources à télécharger mentionnées ci-dessous, sont disponibles à l'adresse http://www-info.iut2.upmf-grenoble.fr/intranet/enseignements/composants/2010-20011/index_files/Page526.htm.

Exercice 1 – Patron « Observer »

« Observer » est un patron utilisé dans la conception de logiciels. Il permet de séparer la logique des données de leur présentation effective. Ainsi, deux éléments peuvent essentiellement être identifiés : *l'observable* (`java.util.Observable` en Java) et *l'observateur* (`java.util.Observer` en Java).

L'objectif de ce premier exercice est d'implémenter une application de météo suivant le patron « Observer ». Cette application est constituée d'une classe `MeteoModel` qui représente le modèle de données, et d'une autre classe `MyGUIView` qui représente l'observateur ou la *vue* du modèle.

Cette application est entièrement implémentée. Vous pouvez la récupérer à partir du site web.

- Créer un nouveau projet. Créer un package `tp3.exo1` et copier les classes récupérées dans le répertoire créé.
- Tester l'application puis analyser minutieusement le code source. Répondre en particulier aux points suivants :
  - Quelle est la super-classe du modèle ?
  - Quelles sont les données représentées dans le modèle ? Qu'associe-t-on à chaque donnée ? Pouvez-vous expliquer le code de la méthode `setTempVille1(…)`.
  - Quelle interface doit implémenter la vue ? Quelle méthode en particulier ? C'est quoi le rôle de cette méthode ?
  - Quelle est la réaction lorsque les « Scrollbar » changent de valeur ?
- Modifier le modèle en ajoutant une nouvelle donnée pour représenter la température d'une troisième ville. Dé-commenter les lignes correspondant à la 3ème ville dans la classe `MyGUIView`.
- La classe `MyView` (que vous avez récupérée) représente une autre vue qui peut être associée au modèle de données. Cette classe contient tous les éléments graphiques nécessaires, cependant, elle n'est pas totalement implémentée. Compléter cette classe.
  - Modifier le constructeur de cette classe en vous inspirant de la classe `MyGUIView`.
  - Quelle est l'interface que doit implémenter cette classe ?
  - Implémenter la méthode `update()` appelée automatiquement par le modèle pour notifier d'éventuels changements :

```
public void update(Observable m, Object obj)
{
	jScrollBar1.setValue(model.getTempVille1());
	...
}
```

- Le clic sur le bouton « **+** » augmente d'un degré la température de la ville sélectionnée. Le bouton « **-** » fait l'opération inverse. Compléter ces deux méthodes en vous inspirant du code suivant :

```
private void boutonPlusClique()
{
	if (jRadioButton1.isSelected())
	{
		model.setTempVille1(model.getTempVille1() + 1);
	}
	else
		// à compléter
}
```

```
private void boutonMoinsClique()
{
	if (jRadioButton1.isSelected())
	{
		model.setTempVille1(model.getTempVille1() - 1);
	}
	else
		// à compléter
}
```

- Modifier la classe **`Main`** pour tester la nouvelle vue. Remarquer que l'action sur l'une des deux vues, déclenche une réponse sur l'autre.

Exercice 2 – SWING, une application du patron « Observer »

Les éléments graphiques Swing sont un exemple d'application du patron « Observer ». D'une façon générale, un composant swing est caractérisé par une interface graphique et par un modèle de données.

Prenons l'exemple du composant **`JTree`** (arbre). Ce composant possède un modèle de type **`TreeModel`**. Des auditeurs peuvent souscrire aux événements envoyés par ce modèle, ils doivent pour cela implémenter l'interface **`TreeModelListener`** qui regroupe les méthodes appelées par le modèle pour notifier d'éventuels changements à ses auditeurs. Implémenter l'interface **`TreeModelListener`** oblige l'auditeur à implémenter toutes les méthodes définies dans cette interface même s'il n'est intéressé que par une seule méthode. Pour résoudre ce problème, le concept d'*adaptateur* a été introduit. Au lieu d'implémenter l'interface, l'auditeur peut simplement étendre l'adaptateur et

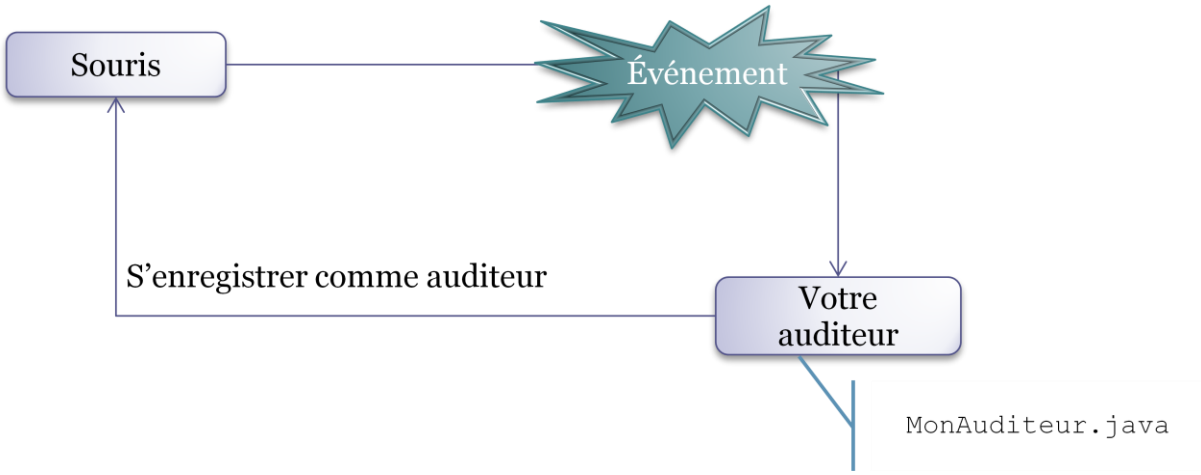
redéfinir les méthodes qui l'intéressent. Dans le cas de l'interface **`TreeModelListener`**, aucun adaptateur n'est fourni par défaut.

L'application que vous pouvez télécharger du site web montre un exemple d'utilisation du composant **`JTree`**.

- Récupérer et tester cette application.
- Déterminer la partie du code qui fait le lien entre l'arbre et son modèle de données (concentrez-vous sur la classe **`SwingObserver`**).
- Modifier la classe **`SwingObserver`** pour qu'elle puisse écouter les événements envoyés par le modèle lorsque l'un des nœuds change de nom (dans ce cas, la méthode **`treeNodesChanged()`** est invoquée). Comme réponse, il faut afficher un message qui indique le chemin du nœud qui a changé de nom (utiliser par exemple la méthode **`getTreePath()`**).
  - En implémentant l'interface **`TreeModelListener`**, combien de méthodes faut-il obligatoirement implémenter ?
  - Utiliser l'adaptateur **`MyTreeModelListenerAdapter`** présent dans l'archive récupérée et redéfinir la méthode **`treeNodesChanged()`**.
- *N'oubliez pas d'ajouter l'instance du **`SwingObserver`** dans la liste des auditeurs du modèle.*

Exercice 3 – Événements graphiques

On va s'intéresser aux événements déclenchés par la souris. Dans ce cas, l'émetteur est alors la souris, les événements sont des objets spécifiques envoyés dans certaines conditions par la souris, et c'est à vous d'écrire les auditeurs qui doivent réagir en fonction des événements émis. Ceci est illustré par la figure ci-dessous.



IMPORTANT

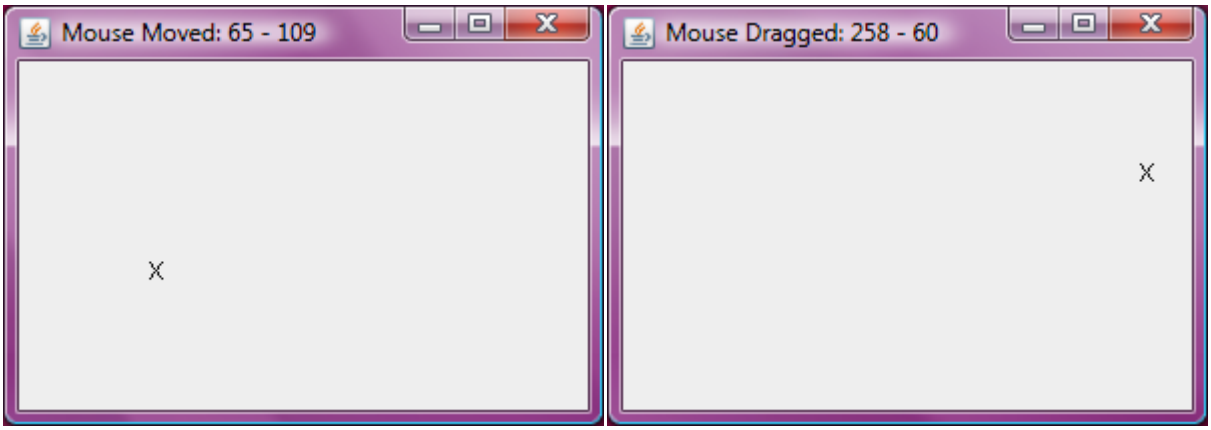
Pour travailler avec les événements, la règle générale est la suivante :

« Pour un événement de type **`X`** (**`Action`**, **`Mouse`**, **`MouseMotion`**…), l'objet émis porte en général le nom **`XEvent`** (**`ActionEvent`**, **`MouseEvent`**, **`MouseEvent`** (et non **`MouseMotionEvent`**), …) et l'auditeur implémente l'interface **`XListener`** (**`ActionListener`**, **`MouseListener`**, **`MouseMotionListener`**, …) »

La classe **`MonAuditeur.java`**, que vous pouvez directement télécharger du site web, représente une simple fenêtre que vous allez utiliser comme support de tests.

- Créer un nouveau package **`tp4.exo3`**, puis télécharger la classe **`MonAuditeur.java`** dans le répertoire **`tp4/exo3`** créé dans votre projet.
- Tester la classe téléchargée.
- Modifier cette classe pour la transformer en un auditeur des événements des mouvements de la souris (événements de type **`MouseEvent`**).
  - Quelle interface doit implémenter la nouvelle classe ? (voir la règle ci-dessus).
  - Implémenter les méthodes nécessaires :
    i. Lorsque la souris est déplacée, on modifie le titre de la fenêtre pour afficher sa nouvelle position (figure ci-dessous). L'image (au milieu de la fenêtre) doit aussi suivre le pointeur de la souris.
    ii. Lorsque la souris est « cliquée-déplacée » *(dragged)*, le titre doit également changer (figure ci-dessous).

Note **:** ne pas oublier d'inscrire votre classe comme auditeur.



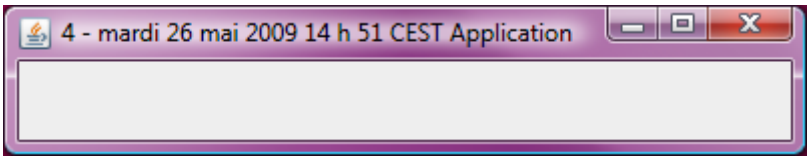
- Passer à l'exercice 4. Ne faites cette partie que si vous avez suffisamment de temps
  - En général, au lieu de faire en sorte qu'une classe particulière implémente elle-même une interface d'écoute, on utilise plutôt des classes dédiées auxquelles on délègue le traitement des événements. Ces classes peuvent être explicites comme elles peuvent être anonymes.
    i. Modifier la classe **`MonAuditeur.java`** en utilisant une classe explicite pour traiter les événements de la souris.

ii. Modifier la classe **`MonAuditeur.java`** en utilisant une classe anonyme pour traiter les événements de la souris.

Exercice 4 – Événements en général

Dans l'exercice précédent, on s'est intéressé aux événements graphiques. L'objectif de cet exercice est de traiter les événements d'une manière générale. On souhaite par exemple réaliser un fournisseur qui diffuse des mises-à-jour vers des clients intéressés. Une mise-à-jour est emballée sous forme d'un « objet événement ».

- Implémenter d'abord la classe **`MAJEvent`** qui représente une mise-à-jour. Cette classe doit avoir comme attributs le nom du logiciel (**`String`**), sa version (**`int`**), sa taille (**`int`**) et son URL (**`String`**). Ces attributs doivent être initialisés dans le constructeur de la classe. Chacun d'eux doit fournir une méthode qui permet de le consulter.
  - *Attention* : tout événement est avant tout un **`EventObject`** ! (**`java.util.EventObject`**).
- Définir l'interface **`MAJListener`** qui doit être implémentée par les auditeurs potentiels. Cette interface doit comporter par exemple une seule méthode **`evenementRecu()`** : `public void evenementRecu(MAJEvent evt);`
- Développer la classe du client.
  - Pour gagner du temps, télécharger cette classe, partiellement développée à partir du site web.
  - Compléter cette classe. A la réception d'une requête de mise à jour, le client doit, par exemple, afficher le nombre total de requêtes reçues (il faut alors un compteur), le nom du dernier logiciel reçu et la date de réception (calculée avec l'instruction approximative : `DateFormat.format(new Date())`[1] par exemple). Au lieu d'envoyer ces messages vers la console, il est préférable de les concaténer et de les associer au titre du client (figure ci-dessous).



- Développer la classe du fournisseur. De même, cette classe, partiellement développée, est disponible. Contentez-vous de la modifier.
  - Le fournisseur doit maintenir un tableau d'abonnés (utiliser **`ArrayList`** pour simplifier).

[1] Voir la javadoc pour plus de précisions sur l'utilisation de la classe `DateFormat`.

- Il doit déclarer deux méthodes pour gérer ses abonnés. Comment ces deux méthodes doivent-elles s'appeler ? Implémentez-les.
- Implémenter la méthode de notification : **`private void notifier()`**. Cette méthode doit être appelée lorsque le bouton du serveur est cliqué. Son rôle est de construire un objet événement et de le notifier à tous les abonnés.

- Ecrire un programme de test. Vous pouvez utiliser par exemple :

```
public class Main
{
	public static void main(String[] args)
	{
		Fournisseur f = new Fournisseur();
		Client c1 = new Client("Client 1");
		f.addMAJListener(c1);
		Client c2 = new Client("Client 2");
		f.addMAJListener(c2);
		Client c3 = new Client("Client 3");
		f.addMAJListener(c3);
	}
}
```

Extension :

On souhaite étendre cette application pour que chaque client, après avoir reçu un événement de mise-à-jour, informe le fournisseur qu'il l'a bien reçu. Ceci permet au fournisseur de connaître, par exemple, les clients pour qui il faut prévoir un nouvel envoi. Proposer et implémenter une ou des solutions adéquates.