Laboratoire «Design patterns»

MVC - MODÈLE OBSERVABLE-OBSERVÉ

OBJECTIFS **Application du modèle MVC, et en particulier du modèle «Observable-Observé».**

Un père Noël passe et repasse avec son chariot au dessus d'une campagne vallonnée pendant que la neige tombe à gros flocons.

Le père Noël est un objet actif dont les déplacement sont observés par le panneau graphique, chargé de le représenter.

Ce même panneau graphique observe chacun des flocons, qui constituent autant d'objets graphiques.

Contenu

1 Mise en place

Démarrer un projet «Together Control Center» basé sur les fichiers donnés par l'enseignant.

En tout et pour tout:

1. Noel.java *Programme à compléter*
2. PNoel.gif
3. Lune.gif
4. Montagne.gif

Si c'est la première fois avec Together..

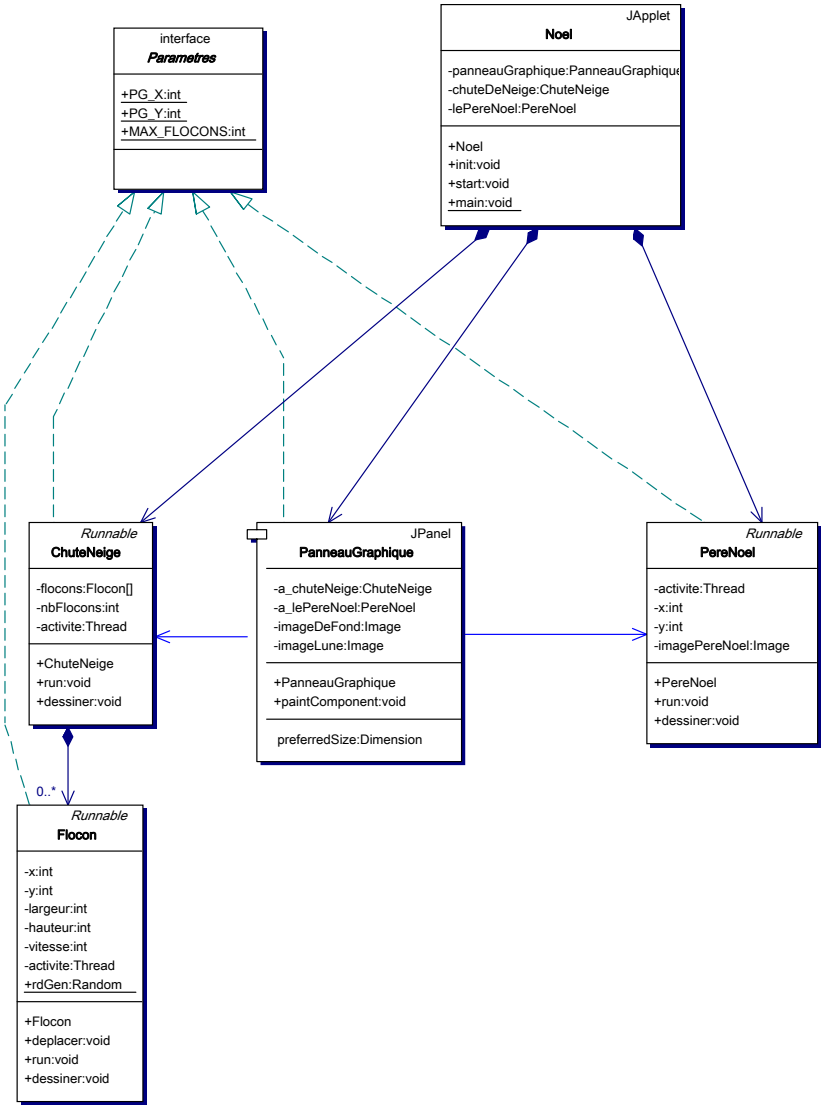
1. Créer un répertoire qui constituera votre «répertoire de travail»
2. Dans ce répertoire :
    1. Placer les 3 fichiers images
    2. Créer un répertoire «**`src`**», y placer le source `Noel.java`
3. Lancer Together..
    1. Créer un nouveau projet en sélectionnant le «répertoire de travail» mentionné plus haut.
    2. Répondre aux autres questions en conservant les choix proposés par défaut par Together.

 **Si tout s'est bien déroulé, on doit voir apparaître le diagramme de classes opéré par «reverse engineering»**

4. Compiler et exécuter le programme
   Pour compiler et exécuter le programme, il suffit de cliquer sur le triangle vert situé dans la barre d'outils de Together.

   A l'occasion de la première exécution, Together demande à l'utilisateur d'indiquer le nom de la classe principale («Select class with main..»). Dans notre cas, il suffit alors de sélectionner la seule classe qui dispose d'une fonction «`main`»: la classe «`Noel`».

2 Diagramme de classes du programme



☞ Essayez d'obtenir ce diagramme avec l'outil **Together** !

Complétez le diagramme généré automatiquement par Together en ajoutant les symboles de composition (losanges) et la cardinalité `0..*` entre «`ChuteNeige`» et «`Flocon`».

- Observez l'effet de ces modifications sur le code du programme
- Tâtez le terrain en essayant d'ajouter des attributs ou des méthodes bidon à partir du diagramme de classes ou à partir du code lui-même. Observez l'effet de

ces adjonctions sur l'artefact correspondant (diagramme ou code).

Vous aurez alors constaté l'impact de la modification du diagramme de classes sur le code.

Soyez donc attentif, à l'avenir, à opérer de temps à autres des sauvegardes de votre code..

3 Compréhension du programme

Lire le code du programme et tâcher de comprendre le rôle des différents objets mis en oeuvre..

En gros:

- **`Noel`** (classe principale)
  Contient les 3 principaux objets: le «Père Noël», la «Chute de neige» et le «Panneau graphique». Le rôle de la classe «`Noel`» consiste principalement à créer ces 3 objets.
- **`PereNoel`**
  Représente le «Père Noel», un objet actif («Runnable»), qui passe son temps à traverser le panneau graphique
- **`ChuteNeige`**
  Contient un tableau de 300 flocons. La chute de neige est un objet actif («Runnable») dont l'activité se limite à la création «en douceur» des 300 flocons, à raison d'un nouveau flocon toutes les 100 msec.
- **`Flocon`**
  Représente un flocon, un objet actif («Runnable»), qui prend naissance en haut du panneau graphique à une position x aléatoire, puis descend jusqu'au sol pour «renaître à nouveau» en haut du panneau graphique et ainsi de suite.

  Ce programme, très gourmand en ressource machine, ne comporte pas moins de 301 threads tournant simultanément !!! Ce n'est pas forcément un exemple à suivre ..
- **`PanneauGraphique`**
  Un `JPanel` dont la procédure de dessin («`paintComponent`») est dédiée principalement à l'affichage des images de fond, du père Noël et des flocons.

4 P1: Afficher les déplacements du père Noël

Mettre en place le modèle Observable-Observé entre le père Noël (sujet observé) et le panneau graphique (l'observateur).

Pour ce faire, utiliser la classe `Observable` et l'interface `Observer` mis à disposition par l'environnement Java.

A chaque fois que l'observateur (le panneau graphique) recevra un message de mise à jour du sujet observé, le message **`repaint()`** sera envoyé de manière à invoquer l'exécution de la procédure **`paintComponent`**.

5 P2: Afficher le déplacement des flocons

Vous l'aurez constaté, la réalisation du point précédent a pour effet d'afficher le déplacement des flocons !! mais c'est trop facile...
Vous allez donc désactiver provisoirement la mécanique «Observable/Observé» du point précédent en plaçant l'instruction «notifyObservers» dans un commentaire.

Puis, vous allez mettre en place le modèle Observable-Observé entre chacun des flocons (sujets observés) et le panneau graphique (l'observateur).

Mêmes remarques que pour le point précédent.

6 P3: Mettre en place une couche de neige

La neige s'accumule et forme une couche de neige ...

Indications:¨

- La couche de neige (une nouvelle classe..) peut être mise en oeuvre au moyen d'un tableau dont chaque case contient la hauteur de la couche en fonction de la coordonnée x du pixel.
- Chaque flocon peut être observé par la couche de neige (en plus du panneau graphique..), qui elle-même sera observée par le panneau graphique.

7 P4: Diagramme de classes

Le programme est terminé..

Tâchez d'opérer un «print» du diagramme de classes que vous avez obtenu avec Together.

8 Code du programme à compléter

```
/*
Fichier:  Noel.java
Rem:
        Exemple d'utilisation du modèle Observer/Observable
        Exécuter le programme en application autonome
        Ou exécuter le programme en mode Applet en ouvrant une fenêÍtre de 315*315

       Le répertoire contenant les classes à exécuter doit contenir les 3 fichiers image:
       "Lune.gif", "Montagne.gif" et "PNoel.gif"


Date :        Eric Lefrançois, Mars 2005
*/

import java.util.*;    // Random, Observer, Observable
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.ImageObserver;
import java.applet.*;

// ----------------------------------------------------------------
interface Parametres {
         // Constantes globales:  taille des composants

         int PG_X = 315;     // Taille en X du panneau graphique
         int PG_Y = 315;     // Taille en Y du panneau graphique

         // Flocons
         int MAX_FLOCONS = 300;     // Nombre max. de flocons

}
```

```
// ----------------------------------------------------------------------
class ChuteNeige implements Runnable, Parametres {

        private Flocon[] flocons  = new Flocon [MAX_FLOCONS];
        private int nbFlocons;    // Nombre de flocons créés
        private Thread activite;

        public ChuteNeige () {
                activite = new Thread (this);
                activite.start();
        }


        public void run () {
        // Activité limitée à la création des flocons
                while (nbFlocons < MAX_FLOCONS) {
                     try {Thread.sleep(100);}
                     catch (InterruptedException e) {}
                     // Créer un nouveau flocon
                     flocons[nbFlocons]=new Flocon();

                     nbFlocons++;
                }
        }

        public void dessiner(Graphics g) {
                for (int i = 0; i < nbFlocons; i++) {
                        flocons[i].dessiner(g);
                }
        }
}
```

```
// ----------------------------------------------------------------------
class Flocon  implements Runnable, Parametres{

        private int x, y;            // Coordonnées courantes du flocon
        private int largeur;         // largeur du flocon (en pixels)
        private int hauteur;         // hauteur du flocon (en pixels)
        private int vitesse;         // DeltaT entre deux dÈplacements en Y

        private Thread activite;

        // Un générateur de nombres aléatoires
        public static Random rdGen = new Random();

        public Flocon () {
                x = rdGen.nextInt(PG_X);
                y = 0;

                // Génération nb aléat. entre 1 et 4
                largeur = 1+rdGen.nextInt(3);
                hauteur = largeur + rdGen.nextInt(2);

                // Vitesse directement fonction de la taille
                vitesse = hauteur;
                activite = new Thread(this);
                activite.start();
        }

        public synchronized void deplacer () {
        // Synchronisation nécessaire:  les coordonnées du flocon sont une
        // ressource critique (dessin et mise à jour simultanés)
                boolean aDroite = rdGen.nextInt(2)==1;
                x = aDroite ? x+1 : x-1;
                y += vitesse ;
                if (y+hauteur> PG_Y) {
                        x = rdGen.nextInt(PG_X);
                        y = 0;
                }
        }

        public void run () {
           while (true) {
                try {Thread.sleep(50*(7-vitesse));}
                catch(InterruptedException e) {}
                deplacer();
           }
        }

        public synchronized void dessiner (Graphics g) {
        // Synchronisation nécessaire:  les coordonnées du flocon sont une
        // ressource critique  (dessin et mise à jour simultanés)
                g.setColor(Color.white);
                g.fillRect(x, y, largeur, hauteur);
                g.setColor(Color.lightGray);
                g.drawRect(x, y, largeur, hauteur);
        }

}
```

```
// ---------------------------------------------------------------------
class PereNoel implements Runnable, Parametres {

        private Thread activite;
        private int x;
        private int y;

      // Coordonnées courantes du flocon
        private  Image imagePereNoel;

        public PereNoel () {
           imagePereNoel =
                Toolkit.getDefaultToolkit().getImage ("PNoel.gif");
           activite = new Thread (this);
           activite.start();
        }

        public void run () {
           x = 0;
           y = PG_Y/4;
           while (true) {
                try {Thread.sleep(50);}
                catch (InterruptedException e) {}
                x += 2;
                if (x > PG_X+100) x = -200;
           }
        }

        public void dessiner(Graphics g, ImageObserver imObs) {
            g.drawImage (imagePereNoel, x, y, imObs);
        }

}
//---------------------------------------------------------------------
class PanneauGraphique extends JPanel implements Parametres {
// Vue passive, pour affichage uniquement


      private ChuteNeige a_chuteNeige;          // Chute de neige

      private PereNoel a_lePereNoel;           // Père Noêl

      private  Image imageDeFond, imageLune;    // Images de fond



// Constructeur
        public PanneauGraphique (ChuteNeige chNg, PereNoel lePereNoel){
           a_chuteNeige = chNg;
           a_lePereNoel = lePereNoel;


           setBackground(new Color (0, 0, 65));
           // Chargement de l'image Montagne.gif, située dans le répertoire
           // des classes.  Image de taille 315*44 pixels
           imageDeFond =
                Toolkit.getDefaultToolkit().getImage ("Montagne.gif");

            // Chargement de l'image Lune
           imageLune =
```

```
                Toolkit.getDefaultToolkit().getImage ("Lune.gif");


            }

            public void paintComponent (Graphics g) {
                    super.paintComponent(g);
                    g.drawImage (imageDeFond, 0, PG_Y-44, this);
                    g.drawImage (imageLune, PG_X-100, PG_Y/4, this);
                    a_chuteNeige.dessiner(g);
                    a_lePereNoel.dessiner(g, this);

                    /*Note .....
                Le "this", 4ème paramètre de drawImage représente "l'image observer"".
                Cet objet contrôle le chargement de l'image en mémoire (chargée
                habituellement depuis un fichier).  Il est responsable de dessiner
                cette image de manière asynchrone au reste du programme, au fur et à
                mesure que l'image se charge.
                Ainsi, le programmeur peut donner l'ordre de charger une image ("getImage"),
                puis il peut la dessiner aussitôt (drawImage), sans attendre qu'elle
                soit chargée.  La procédure drawImage retourne aussitôt.
                L'image observer est implémenté par la classe Component (dont hérite
                la classe Canvas).
                Le cas échéant, il est possible de redéfinir cet objet, ce qui permettrait
                de contrôler le chargement de l'image, d'attendre qu'elle soit entièrement
                chargée avant de l'afficher, etc...
                */
          }

        public Dimension getPreferredSize() {
        // Retourne la taille souhaitÈe pour le composant (remplace le "getSize")
             return new Dimension (PG_X, PG_Y);
        }
    }
    //-------------------------------------------------------------
    public class Noel extends JApplet {
    // Programme "Principal" -->  mise en place du programme
    // Création des "modèles", des "vues"
    // Associations diverses

          private PanneauGraphique panneauGraphique; // Vue

          private ChuteNeige chuteDeNeige = new ChuteNeige();

          private PereNoel lePereNoel = new PereNoel();

          public Noel () {}

          public void init () {
             getContentPane().setLayout (new BorderLayout ());

             // Création des vues, eventuellement associées aux modèles
             panneauGraphique =
                     new PanneauGraphique (chuteDeNeige, lePereNoel);

             // Positionnement des composants
             getContentPane().add (panneauGraphique, "Center");
                // Ajout du panneau graphique
          }

          public void start() {}
```

```
    public static void main (String[] arg) {
    // Point d'entrée du programme
          JFrame f = new JFrame ();
          f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          f.setTitle("Chablon");
          Noel a = new Noel();
          f.getContentPane().add (a, "Center");
          a.init();
          f.pack();
          f.setResizable(false);
          f.setVisible(true);
          a.start();
   }
}
//-----------------------------------------------------------------
```