IN201 : 11 – GUI 1

Solution

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Résumé

Le but de ce TP est de construire une application en utilisant le patron de conception MVC et d'utiliser quelques composants de Swing.

1 Contenu

Ce document est un corrigé succint du premier TP portant sur les interfaces graphiques. Tous les fichiers source sont disponibles sur le site `http://www.tofgarion.net/lectures/IN201`.

2 Présentation du problème

On souhaite réaliser une application qui stocke des chaines de caractères permettant de tester la faisabilité d'un *chat*. Cette application devra posséder deux vues graphiques qui permettront d'envoyer une chaîne de caractères et qui afficheront sur une zone de texte les messages reçus par l'application.

3 Définition du modèle

Le modèle sera donc une application `Chat` qui stockera un ensemble de chaînes de caractères sous la forme d'un objet de type `java.util.ArrayList`. Les services rendus par cette classe seront :
– ajouter une chaîne de caractères dans la liste ;
– récupérer le dernier élément de la liste.

4 Définition des vues

Les vues seront des fenêtres graphiques composées des éléments suivants :
– un bouton permettant d'envoyer le texte à l'application ;
– une zone de texte `JTextField` pour écrire le texte à envoyer ;
– une zone de texte pour écrire les messages provenant de l'application. Ce sera un objet de type `JTextArea`. `JTextArea` possède une méthode `append(String s)` permettant de rajouter une chaîne de caractères dans la zone.

5 Conception de l'application

Proposer un diagramme de classes simples modélisant l'application. On s'appliquera à bien choisir la façon dont la vue et le modèle vont communiquer.

Solution :
Vous trouverez une proposition de diagramme sur la figure 1. Il n'y avait pas de problème particulier pour les classes qui étaient simples. J'ai choisi d'utiliser le *pattern* Observateur pour la communication entre le modèle et les vues. Vous remarquerez que la classe `Chat` ne « connaît » pas les vues qui lui sont associées. Le *listener* associé au bouton est lié à une instance de `Chat`, mais il existe juste une relation de dépendance avec `VueChat`, car nous travaillerons avec l'instance de `JTextField` présente dans `VueChat` pour récupérer le texte à ajouter.
Si nous n'avions pas utilisé le *pattern* Observateur, il aurait fallu que l'observateur connaisse l'ensemble des vues à prévenir.

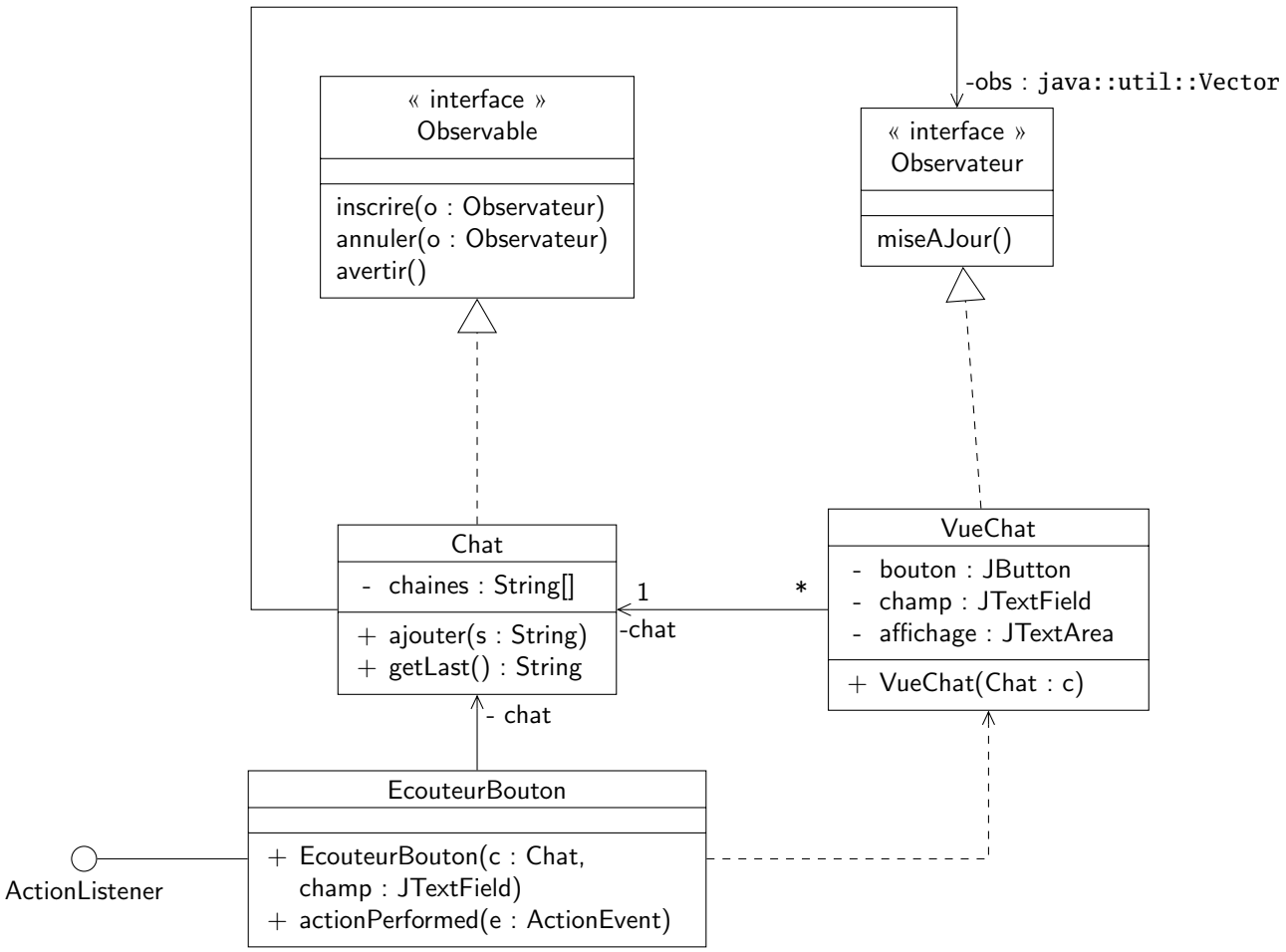

FIGURE 1 – Diagramme de classes de l'application

6 Implantation

Il faut maintenant implanter la classe applicative, les vues et les contrôleurs associés. Vous pourrez réutiliser les classes et interfaces développées dans le TP 10 sur le *pattern* Observateur. Vous pourrez également utiliser les classes et interfaces fournies par le JDK implantant ce *pattern*, i.e. `java.util.Observable` et `java.util.Observer`. Cette classe et cette interface fonctionnent à peu près comme celles que nous avons développées, sauf que l'on peut passer des paramètres aux méthodes correspondant à `miseAJour`, `avertir` etc. De plus, la classe `java.util.Observable` utilise un booléen pour représenter que l'état de l'objet a réellement changé (pour éviter de prévenir les observateurs si ce n'est pas nécessaire). Référez-vous à la javadoc de ces classes pour plus de détails ou au corrigé du TP10.

1. écrire la classe `Chat` et la tester rapidement.

   Solution :

   Pas de problème particulier, j'ai écrit une classe de test JUnit pour vérifier que les fonctionnalités de la classe étaient correctes. Il fallait faire attention en manipulant les instances de `ArrayList` de bien utiliser les bonnes méthodes. J'ai choisi ici de spécialiser la classe `java.util.Observable` pour le mécanisme d'observateur.

2. écrire la classe `VueChat` et vérifier qu'elle s'affiche correctement. On pourra utiliser d'autres *layout managers* que celui vu en cours si nécessaire.

Solution :

Là encore, pas de problème particulier. Il suffisait d'appliquer ce qui avait été vu en cours. `VueChat` est une sous classe de `JFrame`, donc on pouvait utiliser les méthodes de `JFrame` comme `pack` directement sur l'instance courant de `VueChat`. J'ai choisi de coder comme attribut les éléments de la vue. On aurait également pu les déclarer comme variables locales du constructeur, mais ce n'est pas recommandé.

La classe réalise l'interface `java.util.Observer`.

La méthode `update` permettait de récupérer la dernière chaîne entrée sur l'instance de `Chat` associée.

Enfin, il fallait inscrire la vue comme observateur de l'instance de `Chat` associée.

3. tester l'application.

Solution :

Je vous propose sur le site une classe applicative `TestChat` créant une instance de `Chat` et deux vues sur cette instance. On vérifie bien que les deux vues sont mises à jour en même temps. J'ai utilisé la méthode `setLocation` sur les vues pour pouvoir les positionner différemment. Il se peut que cela ne fonctionne pas très bien suivant votre environnement.

Aspects (très) avancés : on peut se poser la question de l'automatisation des tests pour les interfaces graphiques. En effet, il peut être extrêmement fastidieux de tester « à la main » une interface graphique. Il existe plusieurs *frameworks* permettant d'écrire des tests unitaires pour les interfaces graphiques utilisant Swing. Nous en présentons deux ici, Google Fest [1]et SwingUnit [2]. Ils sont tous les deux disponibles gratuitement sous des licenses libres.

Ces deux *frameworks* utilisent en fait la classe `java.awt.Robot`, qui est une classe de base permettant de simuler des interactions avec une interface graphique. Elle est assez difficile à utiliser, car il faut connaître précisément la position des composants par exemple. Les deux *frameworks* présentés permettent de simplifier l'utilisation de cette classe (elle est même transparente). Ce sont tous les deux des extensions de JUnit (on écrit donc des classes de test JUnit).

Google Fest est le plus simple à utiliser. Le principe est simple : lorsque l'on veut tester un composant graphique, on crée un *fixture* qui correspond au composant à tester. On dispose ensuite de méthodes permettant de récupérer un sous-composant, de cliquer sur un composant, de tester l'apparition d'une fenêtre ou d'un texte etc. Par exemple, si l'on veut tester le comportement d'une instance de `JFrame`, on utilise un objet de type `FrameFixture` qui possède une méthode `button` permettant de récupérer un *fixture* pour un bouton particulier de l'instance de `JFrame`, ce *fixture* possédant lui-même une méthode `click` pour cliquer sur le bouton. Pour pouvoir référencer les différents composants des vues, on utilise le nom des composants que l'on peut positionner dans le code Java avec la méthode `setName`[1].

La documentation de Fest est très claire et vous pourrez la consulter. Vous trouverez sur le listing 1les tests JUnit correspondant aux tests fonctionnels (de base et non exhaustifs !) de `VueChat`. On remarquera que pendant l'exécution des tests, les fenêtres sont réellement créées et la souris se déplace pour cliquer sur les boutons etc. J'utilise la méthode `setAlwaysOnTop` pour être sûr que c'esy la bonne vue qui est en premier plan, car lors du lancement les deux vues sont superposées.

Listing 1– Tests JUnit Google Fest de VueChat

```
package fr.supaero.gui;

import javax.swing.JFrame;
import org.fest.swing.fixture.FrameFixture;
import org.junit.*;

import static org.junit.Assert.*;

/**
 *  Unit Test for class VueChat. It uses the Fest framework.
 *
 *
 * Created: Sun Feb 22 22:10:29 2009
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class VueChatTestFest {

    private FrameFixture app1;
    private FrameFixture app2;
```

```
    private VueChat v1;
    private VueChat v2;

    @Before public void setUp() {
        Chat chat = new Chat();

        v1 = new VueChat(chat, "Vue1");
        app1 = new FrameFixture(v1);
        app1.show();

        v2 = new VueChat(chat, "Vue2");
        app2 = new FrameFixture(app1.robot, v2);
        app2.show();
    }

    @After public void tearDown() {
        app1.cleanUp();
        app2.cleanUp();
    }

    /**
     * <code>testVueUn</code> permet de verifier que l'entree d'un
     *  mot sur la premiere vue affiche bien le texte dans les deux
     *  vues.
     */
     @Test public void testVueUn() {
         v1.setAlwaysOnTop(true);
         app1.textBox("champ").enterText("Coucou");
         app1.button("bouton").click();
         app1.textBox("aff").requireText("Coucou\n");
         app2.textBox("aff").requireText("Coucou\n");
     }

    /**
     * <code>testVueDeux</code> permet de verifier que l'entree d'un
     *  mot sur la premiere vue affiche bien le texte dans les deux
     *  vues.
     */
    @Test public void testVueDeux() {
         v2.setAlwaysOnTop(true);
         app2.textBox("champ").enterText("Blibli");
         app2.button("bouton").click();
         app2.textBox("aff").requireText("Blibli\n");
         app1.textBox("aff").requireText("Blibli\n");
     }

    /**
     * <code>testDeuxVues</code> permet de verifier que l'entree d'un
     *  mot sur la premiere vue puis sur la deuxieme affiche bien les
     *  deux textes dans les deux vues.
     */
     @Test public void testDeuxVues() {
         v1.setAlwaysOnTop(true);
         app1.textBox("champ").enterText("Coucou");
         app1.button("bouton").click();
         v1.setAlwaysOnTop(false);
         v2.setAlwaysOnTop(true);
         app2.textBox("champ").enterText("Blibli");
```

```
        app2.button("bouton").click();
        app1.textBox("aff").requireText("Coucou\nBlibli\n");
        app2.textBox("aff").requireText("Coucou\nBlibli\n");
    }
}// ChatTest
```

SwingUnit permet d'écrire des scénarios simulant des actions sur une interface graphique en utilisant la classe `java.awt.Robot`. Pour cela, on écrit les scénarios dans des fichiers au format XML.

Par exemple, le listing 2présente plusieurs scénarios d'interaction avec deux vues sur un chat. Là encore, pour pouvoir référencer les différents composants des vues, on utilise le nom des composants que l'on peut positionner dans le code Java avec la méthode `setName`.

Listing 2– Fichier XML décrivant des scénarios d'interaction avec les vues

```
<?xml version="1.0" encoding="utf-8" ?>
<document>
  <scenario name="AJOUT_VUE1">
    <Click name="champ" componentClass="JTextField"
           windowName="Vue1" windowClass="JFrame"/>
    <TypeText text="Coucou"/>
    <Click name="bouton" componentClass="JButton"
           windowName="Vue1" windowClass="JFrame"/>
    <VerifyText text="Coucou&#xa;" componentClass="JTextArea"
                name="aff" windowName="Vue1" windowClass="JFrame"/>
    <VerifyText text="Coucou&#xa;" componentClass="JTextArea"
                name="aff" windowName="Vue2" windowClass="JFrame"/>
  </scenario>

  <scenario name="AJOUT_VUE2">
    <Click name="champ" componentClass="JTextField"
           windowName="Vue2" windowClass="JFrame"/>
    <TypeText text="Blibli"/>
    <Click name="bouton" componentClass="JButton"
           windowName="Vue2" windowClass="JFrame"/>
    <VerifyText text="Blibli&#xa;" componentClass="JTextArea"
                name="aff" windowName="Vue1" windowClass="JFrame"/>
    <VerifyText text="Blibli&#xa;" componentClass="JTextArea"
                name="aff" windowName="Vue2" windowClass="JFrame"/>
  </scenario>

  <scenario name="AJOUT_VUES">
    <Click name="champ" componentClass="JTextField"
           windowName="Vue1" windowClass="JFrame"/>
    <TypeText text="Blibli"/>
    <Click name="bouton" componentClass="JButton"
           windowName="Vue1" windowClass="JFrame"/>
    <Click name="champ" componentClass="JTextField"
           windowName="Vue2" windowClass="JFrame"/>
    <TypeText text="Coucou"/>
    <Click name="bouton" componentClass="JButton"
           windowName="Vue2" windowClass="JFrame"/>
    <VerifyText text="Blibli&#xa;Coucou&#xa;" componentClass="JTextArea"
                name="aff" windowName="Vue1" windowClass="JFrame"/>
    <VerifyText text="Blibli&#xa;Coucou&#xa;" componentClass="JTextArea"
                name="aff" windowName="Vue2" windowClass="JFrame"/>
  </scenario>
</document>
```

Prenons par exemple le premier scénario décrit dans le listing 2. Voici les actions effectuées par ce scénario :

1. on sélectionne l'instance de `JTextField` appelée `bouton` de la fenêtre `Vue1` ;
2. on introduit la chaîne de caractères `"Bonjour"` dans ce champ ;
3. on vérifie que le texte contenu dans l'instance appelée `aff` de `JTextArea` de `Vue1` est bien `"Bonjour"` (`&#xa` permet de représenter un retour à la ligne en XML) ;
4. on vérifie que le texte contenu dans l'instance appelée `aff` de `JTextArea` de `Vue2` est bien `"Bonjour"`.

Si jamais le texte n'est pas le même, le test JUnit échouera.
La classe de test JUnit correspondante est présentée sur le listing 3. La méthode `setUp` permet de créer deux vues sur un même chat en utilisant des *threads* (non abordés dans ce cours).

Listing 3– Test JUnit utilisant les scénarios développés précédemment

```
package fr.supaero.gui;

import org.junit.*;
import static org.junit.Assert.*;
import java.awt.Robot;
import javax.swing.SwingUtilities;
import swingunit.extensions.ExtendedRobotEventFactory;
import swingunit.framework.Finder;
import swingunit.framework.EventPlayer;
import swingunit.framework.ExecuteException;
import swingunit.framework.FinderMethodSet;
import swingunit.framework.Scenario;

/**
 *  Unit Test for class VueChat. It uses the TestChat application and
 *  the SwingUnit framework.
 *
 *
 * Created: Sun Jan 29 22:10:29 2006
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class VueChatTestSwingUnit {

    private VueChat app1;
    private VueChat app2;
    private Scenario scenario;
    private Robot robot;

    @Before public void setUp() throws Exception {
        // On demarre l'application
        Runnable r = new Runnable() {
                public void run() {
                    Chat chat = new Chat();
                    app1 = new VueChat(chat, "Vue1");
                    app1.setLocation(10, 10);
                    app2 = new VueChat(chat, "Vue2");
                    app2.setLocation(500, 10);
                }
            };
        SwingUtilities.invokeAndWait(r);

        robot = new Robot();

        String filePath = VueChatTestSwingUnit.class.getResource("VueChatTest.xml").getFile();
```

```
        scenario = new Scenario(new ExtendedRobotEventFactory(),
                                new FinderMethodSet());

        scenario.read(filePath);
    }

    @After public void tearDown() throws Exception {
        // Terminate application.
        Runnable r = new Runnable() {
                public void run() {
                    app1.dispose();
                    app1 = null;
                    app2.dispose();
                    app2 = null;
                }
            };
        SwingUtilities.invokeAndWait(r);
        scenario = null;
        robot = null;
    }

    /**
     * <code>testVueUn</code> permet de verifier que l'entree d'un
     *  mot sur la premiere vue affiche bien le texte dans les deux
     *  vues.
     *
     * @exception ExecuteException if an error occurs
     */
     @Test public void testVueUn() throws ExecuteException {
        EventPlayer player = new EventPlayer(scenario);
        player.run(robot, "AJOUT_VUE1");
    }

    /**
     * <code>testVueDeux</code> permet de verifier que l'entree d'un
     *  mot sur la premiere vue affiche bien le texte dans les deux
     *  vues.
     *
     * @exception ExecuteException if an error occurs
     */
     @Test public void testVueDeux() throws ExecuteException {
        EventPlayer player = new EventPlayer(scenario);
        player.run(robot, "AJOUT_VUE2");
    }

    /**
     * <code>testDeuxVues</code> permet de verifier que l'entree d'un
     *  mot sur la premiere vue puis sur la deuxieme affiche bien les
     *  deux textes dans les deux vues.
     *
     * @exception ExecuteException if an error occurs
     */
     @Test public void testDeuxVues() throws ExecuteException {
        EventPlayer player = new EventPlayer(scenario);
        player.run(robot, "AJOUT_VUES");
    }
}// ChatTest
```

Il faut que le fichier XML contenant les scénarios soit au même endroit que le *bytecode* de `VueChatTest`. Lors du lancement du test, le « robot » exécute les scénarios d'interaction avec l'interface et vérifie les propriétés demandées.

On pourra enfin remarquer que SwingUnit propose également une interface graphique permettant d'enregistrer les interactions effectuées par un utilisateur sur l'interface graphique à tester et de construire un scénario de façon plus conviviale.

Cependant, d'après mon expérience personnelle, Fest est plus stable surtout s'il est utilisé avec un système de *build* automatique comme Ant par exemple.

Références

[1] *Google Fest - Fixtures for Easy Software Testing*. URL : http://code.google.com/p/fest/.

[2] *SwingUnit*. URL : https://swingunit.dev.java.net/.