IN201 : 7 – GUI (chat)

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

Isae
Institut Supérieur de l'Aéronautique et de l'Espace
SUPAERO

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 TP 7 portant sur les interfaces graphiques. Tous les fichiers source sont disponibles sur le site http://www.tofgarion.net/lectures/IN201.

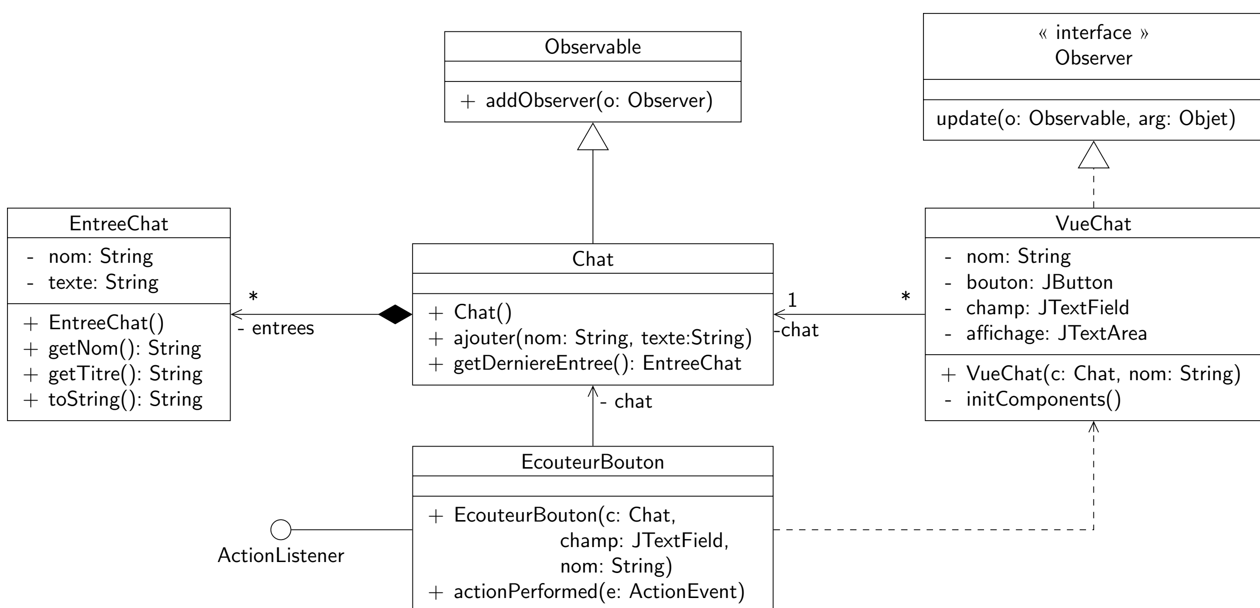
2 Présentation du problème

On souhaite réaliser une interface graphique pour une application de *chat*[1]. On rappelle qu'un *chat* permet à des utilisateurs de s'envoyer des messages de façon informelle. L'application à développer 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 ainsi que les noms d'utilisateurs correspondant à chaque message.

3 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 bien évidemment utilisé le *pattern* Observateur pour la communication entre le modèle et les vues en utilisant l'interface `Observer` fournie par l'API. 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 explicitement l'ensemble des vues à prévenir.



1. Causette en francais, clavardage en québécois...

Figure 1 – Diagramme de classes de l'application

4 Implantation

Il faut maintenant implanter les vues et les contrôleurs associés. La classe `Chat` vous est fournie dans l'archive disponible sous le répertoire `lib` de votre dépôt. Vous devez utiliser les classes et interfaces fournies par le JDK implantant le patron de conception Observateur, 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. Référez-vous à la javadoc de ces classes pour plus de détails.

1. un premier observateur pour `Chat` vous est fourni dans l'archive. Il s'agit de la classe `ChatFileLogger` qui écrit les conversations du Chat dans un fichier texte (cf. javadoc). Écrire un programme utilisant cette classe et vérifier que les entrées ajoutées dans le *chat* sont bien écrites dans le fichier choisi.

   Solution :
   Pas de problème particulier, le listing de la classe est présenté sur le listing 1. Il fallait faire attention aux différents paquetages utilisés.

   Listing 1– Programme utilisant l'observateur écrivant dans un fichier

```java
package fr.isae.chat.log;

import fr.isae.chat.model.Chat;

/**
 * <code>TestChatFileLogger</code> permet de tester le logger vers
 * un fichier.
 *
 * @author <a href="mailto:garion@isae.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TestChatFileLogger {

    /**
     * Programme testant l'observateur enregistrant les entrees
     * du chat dans un fichier.
     *
     * @param args non utilise ici
     */
    public static void main(String[] args) {
        Chat c = new Chat();
        ChatFileLogger logger = new ChatFileLogger("log.txt", c);
        c.ajouter("Christophe", "Coucou !");
        c.ajouter("Christophe", "Ca va ?");
        c.ajouter("John Doe", "Ben oui.");
    }
}
```

2. écrire la classe `VueChat` (on pourra utiliser d'autres *layout managers* que celui vu en cours si nécessaire). Pour cela :
   (a) quels sont les paramètres du constructeur de `VueChat` ? Implanter le constructeur en conséquence, en prévoyant que la classe est une sous classe de `JFrame` (titre de la fenêtre à mettre en place etc).

Solution :

La solution était normalement déjà trouvée lors de la conception. Lorsque l'on construit une vue, on a besoin :

- d'une instance de `Chat` qui représente la conversation à laquelle on participe ;
- d'un nom d'utilisateur (plusieurs utilisateurs participent à une conversation, donc on aura plusieurs vues sur le même *chat*).

(b) la méthode `initComponents` créant les composants et les plaçant via un `FlowLayout` vous est donnée. Modifier le constructeur de `VueChat` afin de l'appeler. Modifier le programme créé à la question 1 en créant deux vues sur le *chat* avant d'ajouter des entrées sur le *chat*. Que se passe-t-il ?

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 d'implanter les éléments de la vue comme des attributs de la classe. On aurait également pu les déclarer comme variables locales du constructeur (ou de `initComponents`, cf. suite), mais ce n'est pas recommandé. J'ai utilisé une méthode privée `initComponents` pour initialiser et placer les composants de la vue comme vu en cours.

Vous remarquerez que j'ai utilisé deux *layout managers*, `BorderLayout` et `FlowLayout` pour construire la vue. J'ai utilisé une instance de `JPanel` intermédiaire pour pouvoir placer les composants.

Lorsque l'on exécute le programme, les deux vues se créent, mais par contre on ne voit pas apparaître les entrées du *chat*.

(c) faire en sorte que `VueChat` soit un observateur de `Chat`. Que se passe-t-il lorsque l'on exécute le programme précédent ?

Solution :

La classe réalise l'interface `java.util.Observer`. La méthode `update` permettait de récupérer la dernière entrée entrée sur l'instance de `Chat` associée. Il ne fallait pas oublier d'inscrire la vue comme observateur de l'instance de `Chat` associée. Il ne fallait pas oublier d'inscrire la vue comme observateur du *chat*.

Lorsque l'on exécute le programme, normalement les entrées ajoutées dans le *chat* doivent apparaître dans les deux vues.

3. implanter le contrôleur permettant de lier la vue et le modèle. Créer un programme de test créant un *chat* et deux vues sur le *chat* et vérifier que tout fonctionne.

Solution :

Le contrôleur n'était pas très compliqué à écrire, vous le trouverez dans la classe `EcouteurBouton`. Il aurait été beaucoup plus judicieux d'implanter `EcouteurBouton` comme une une classe *interne* à `VueChat`, ce qui aurait permis d'avoir accès directement aux attributs de `VueChat` nous intéressant (cf. transparents du cours).

Un diagramme de séquence (simplifié, en particulier les interactions entre l'utilisateur et l'IHM sont représentés par des appels de « méthode », la méthode `actionPerformed` n'est pas appelée par le programme de test etc.) vous est présenté sur la figure 2.

CC BY-NC-SA

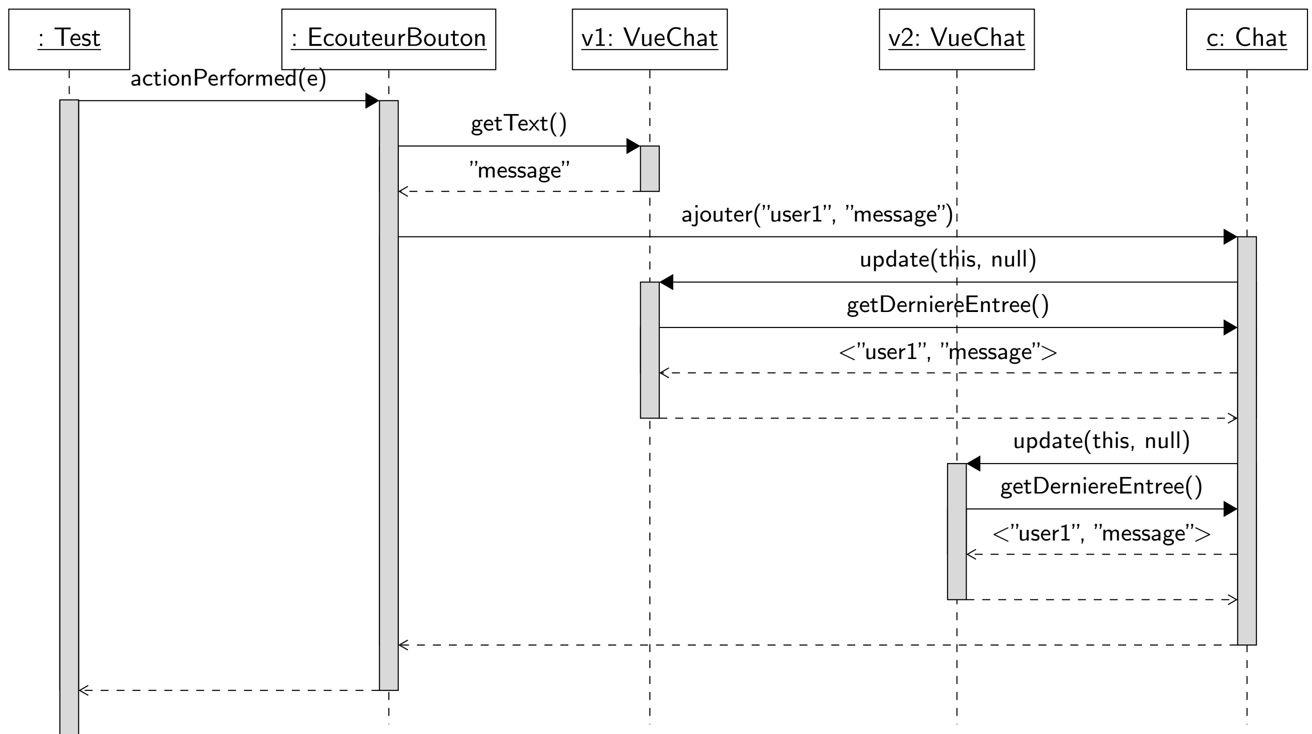


FIGURE 2 – Un diagramme de séquence présentant un envoi de message au *chat* via la vue 1

Vous remarquerez que j'ai ajouté le même *listener* sur l'instance de JTextField afin que le message soit envoyé lorsque l'utilisateur appuie sur Return .

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.

4. utiliser un autre *layout manager* pour placer le bouton et la zone de texte en dessus de la fenêtre de *chat*. On pourra consulter [2] pour trouver le *layout manager* à choisir.

Solution :

La solution la plus simple ici était d'utiliser une instance de BorderLayout. Ce *layout manager* possède 5 emplacements : NORTH, SOUTH, WEST, EAST et CENTER. Rien de bien difficile pour l'utiliser.

5. (facultatif) ajouter un autre bouton et une zone de texte pour pouvoir changer le nom de l'utilisateur. On pourra utiliser une classe interne ou une classe anonyme (cf. slides et votre vacataire).

Solution :

Il fallait dans un premier temps ajouter deux nouveaux attributs, un pour le bouton et un pour la zone de texte. J'ai décidé de changer les noms des attributs existants (en ajoutant « Texte ») pour être plus clair.

J'ai choisi de mettre les boutons et les zones de texte dans des instances de JPanel en suivant un FlowLayout. J'ai ensuite mis ces deux panels dans un autre panel équipé d'une instance de BoxLayout qui me permettait de les agencer verticalement.

Pour placer un *listener* sur boutonNom, j'ai choisi d'utiliser une classe anonyme (nous verrons dans la suite que cela pose pas mal de problèmes) :

```
104         this.boutonNom.addActionListener(new ActionListener() {
105                 public void actionPerformed(ActionEvent e) {
106                     nom = champNom.getText();
107                     setTitle("Vue " + nom);
108 
109                     for (ActionListener a: boutonTexte.getActionListeners()) {
110                         boutonTexte.removeActionListener(a);
111                         champTexte.removeActionListener(a);
112                     }
```

CC BY-NC-SA

```
113
114                     boutonTexte.addActionListener(
115                         new EcouteurBouton(chat,
116                                             champTexte,
117                                             nom));
118                     champTexte.addActionListener(
119                         new EcouteurBouton(chat,
120                                             champTexte,
121                                             nom));
122                 }
123             });
124     }
```

Une classe anonyme ne porte pas de nom et on construit une instance de cette classe directement. L'intérêt est de pouvoir accéder directement aux attributs de la classe englobante et aux variables locales de la méthodes dans laquelle on crée l'instance (sous certaines conditions). L'accès aux attributs peut également se faire via une classe interne (cf. transparents du cours). Ici, le choix d'une classe anonyme n'était pas forcément judicieux, car le *listener* pouvait également servir pour la zone de texte associée, comme pour le bouton « Envoi ».

Dans le code de la méthode `actionPerformed` du *listener*, je change le nom de l'utilisateur en fonction du contenu de la zone de texte (remarquez que je ne peux pas utiliser **this**.nom !), je change le nom de la fenêtre et ensuite je change le *listener* sur le bouton d'envoi et la zone de texte associée, car l'instance de *listener* qui y est associée utilise l'ancien nom ! Il faut également enlever l'ancienne instance, ce qui est assez lourd. . . Moralité : une classe interne pour le *listener* d'envoi aurait été intéressante, car on aurait eu accès directement à l'attribut `nom` de la vue. On n'aurait pas eu à changer le *listener*.

6. (facultatif) utiliser l'extension WindowBuilder d'Eclipse (cf. page Logiciels du site web pour l'installer sur votre machine personnelle) pour construire une nouvelle vue pour votre application. Le manuel utilisateur de WindowBuiler est disponible sur [1] et un petit tutoriel vous est proposé sur le site du cours.
7. (facultatif) tester l'application avec un *framework* de test unitaire comme Google Fest [3].

Solution :

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 [3]et SwingUnit [4]. 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`[2].

La documentation de Fest est très claire et vous pourrez la consulter. Vous trouverez sur le listing 2 les 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 2– Tests JUnit Google Fest de `VueChat`

```
package fr.isae.chat.gui;

import fr.isae.chat.model.Chat;
import javax.swing.JFrame;
import org.fest.swing.fixture.FrameFixture;
```

```java
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@isae.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", "User1");
        app1 = new FrameFixture(v1);
        app1.show();

        v2 = new VueChat(chat, "Vue2", "User2");
        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("<User1>: Coucou\n");
         app2.textBox("aff").requireText("<User1>: 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("<User2>: Blibli\n");
        app1.textBox("aff").requireText("<User2>: 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("<User1>: Coucou\n<User2>: Blibli\n");
        app2.textBox("aff").requireText("<User1>: Coucou\n<User2>: Blibli\n");
    }
}
```

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 3 présente 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 3– Fichier XML décrivant des scénarios d'interaction avec les vues

```xml
<?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 3. 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"` (`
` 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 4. La méthode `setUp` permet de créer deux vues sur un même chat en utilisant des *threads* (non abordés dans ce cours).

Listing 4– Test JUnit utilisant les scénarios développés précédemment

```java
package fr.isae.chat.gui;

import fr.isae.chat.model.Chat;
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@isae.fr">Christophe Garion</a>
 * @version 1.0
 */
public class VueChatTestSwingUnit {

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

(cc) BY-NC-SA

```
    @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", "Christophe");
                    app1.setLocation(10, 10);
                    app2 = new VueChat(chat, "Vue2", "Toto");
                    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, "AJOUTVUE1");
    }

    /**
     * <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
     */
```

```java
        @Test public void testVueDeux() throws ExecuteException {
            EventPlayer player = new EventPlayer(scenario);
            player.run(robot, "AJOUTVUE2");
        }

        /**
         * <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, "AJOUTVUES");
        }
    }
```

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. *WindowBuilder User Guide*. 2013. URL : https://developers.google.com/java-dev-tools/wbpro/.

[2] Oracle. *Creating a GUI With JFC/Swing*. 2013. URL : http://docs.oracle.com/javase/tutorial/uiswing/.

[3] The Fest Developers. *Google Fest – Fixtures for Easy Software Testing*. 2013. URL : http://code.google.com/p/fest/.

[4] The Swingunit team. *Swingunit*. 2013. URL : http://java.net/projects/swingunit.

License CC BY-NC-SA 3.0

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmite) and to Remix (adapt) this work under the following conditions:

Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial – You may not use this work for commercial purposes.

Share Alike – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See http://creativecommons.org/licenses/by-nc-sa/3.0/ for more details.