Chapitre 5

Swing et la gestion des événements

L'API SWING permet de réaliser des interfaces graphiques plus sophistiquées que celles que permettait AWT, son ancêtre. Outre les nombreuses classes fournissant des composants graphiques, il faut aussi connaître le mécanisme complexe qui permet de gérer les événement qui surviennent dans cet environnement.

5.1 L'interface graphique de Java

Java 1.0 était déjà pourvu d'une interface graphique, nommée AWT. Elle a rapidement montré ses limites. On la trouve toujours dans l'API JAVA parce que certaines de ses classes sont utilisées par la nouvelle interface graphique, nommée SWING. Concurremment, IBM a développé une autre interface, nommée SWT, qu'on trouve notamment utilisée par le produit Eclipse. SWT présente un certain nombre d'avantages sur Swing, notamment son aspect esthétique. Je n'aborderai que Swing parce que c'est l'interface standard, toujours présente sur une machine Java, et parce qu'elle est fortement intégrée dans NetBeans, qui dispose de puissants outils pour la gérer. Swing permet de réaliser deux types d'interfaces : une interface indépendante et une interface intégrée dans un navigateur. Pour l'essentiel, les deux technologies font appel aux mêmes composants, seules diffèrent les initialisations. Les applets ne seront pas présentées dans cette initiation.

Selon qu'on programme avec un éditeur classique ou avec l'aide de NetBeans, on aura deux techniques différentes pour réaliser une fenêtre :

- dans un programme réalisé sans assistance, la fenêtre principale sera une instance de JFrame qu'il faudra créer (généralement dans la méthode main()). On ajoutera à la main les composants avant d'afficher la fenêtre (à l'aide de l'appel de méthode `setVisible(true)` [1]).
- si on utilise l'assistance de NetBeans, la préparation d'une fenêtre passe par la création de deux fichiers :
  - un fichier d'extension `.form` contient, sous format XML, tous les paramètres choisis par l'utilisateur : composants, tailles, positions, événements gérés...
  - un fichier de classe normal contient les instructions Java pour réaliser des instances de fenêtres conformes à ce qu'a prévu le programmeur.

  Les deux fichiers sont modifiés par le biais d'une interface graphique accessible par le bouton *Design*. Tous les paramètres définis par l'interface graphique sont placés dans des zones protégées du fichier de classe, zone que NetBeans ne permet pas de modifier. La classe ainsi définie contient uniquement des variables d'instance, par défaut privées, correspondant aux composants placés dans la fenêtre, des méthodes privées permettant

[1] La méthode `show` est dépréciée.

leur initialisation et des *call-backs*, méthodes réduites à un en-tête avec des paramètres qui seront appelées par les différents événements pris en compte par l'interface. Le travail du programmeur consiste uniquement à programmer le contenu de ces callbacks[2].

5.1.1 Un exemple de base (réalisé sans NetBeans)

Un première fenêtre va reprendre les éléments minimum :

- la fenêtre avec son titre (placé comme paramètre du constructeur)
- un *layout* de base, qui se contente de placer les composants à la queue leu leu, pour autant que la largeur de la fenêtre le permette, et qu'on instancie avant de l'associer à l'aide de la méthode `setLayout()`
- quatre composants, deux étiquettes, une zone de texte et une zone de mot de passe, qu'on place directement sur la fenêtre à l'aide de la méthode `add()`. Le fait de ne pas leur faire correspondre des variables d'instance risque de rendre leur gestion assez complexe.
- la méthode `pack()` donne à la fenêtre des dimensions suffisante pour afficher ses composants.
- il ne faut pas oublier de préciser ce qui arrivera en cas de fermeture de la fenêtre (ici fermer l'application, faute de quoi on garderait un programme sans interface en toile de fond).

```
private static void myDisplay() {
      JFrame fenetre=new JFrame("Ma␣première␣fenêtre");
      fenetre.setLayout(new FlowLayout(LEFT,5,5));
      fenetre.add(new JLabel("Pseudonyme:␣"));
      fenetre.add(new JTextField(10));
      fenetre.add(new JLabel("Mot␣de␣passe:␣"));
      fenetre.add(new JPasswordField(10));

      fenetre.pack();
      fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      fenetre.setVisible(true);
 }
```

5.1.2 Utilisation de NetBeans

La place manque pour décrire le fonctionnement de l'interface graphique de NetBeans, qui se révèle assez intuitive. L'exemple suivant tente de réaliser la même fenêtre que dans la sous-section précédente. L'illustration de la figure 5.1 montre les trois zones principales :

- l'inspecteur permet de naviguer dans les composants
- la fenêtre de création montre l'aspect de la fenêtre

[2]En principe, le contenu des callbacks est préservé en cas de modification de l'interface, sauf si l'événement qui est censé l'appeler est supprimé. Si le code est spécialement complexe, il peut être prudent de le protéger en faisant une copie, ou mieux en le plaçant dans une autre méthode que le callback se contentera d'appeler. L'existence d'une autre méthode est aussi un moyen simple d'assigner un même traitement à plusieurs événements.

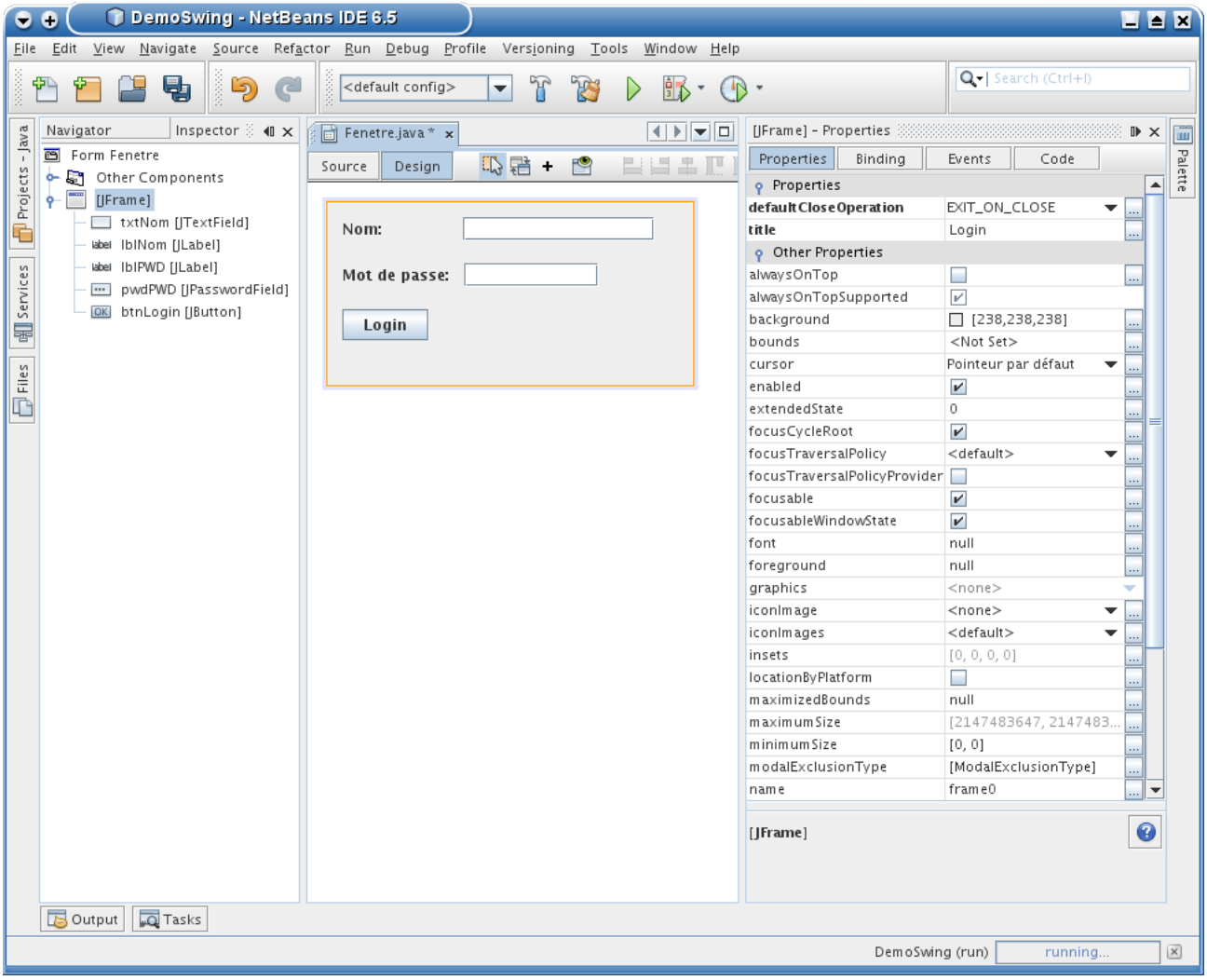


FIG. 5.1 – L'interface Matisse

– la zone de propriétés permet d'initialiser le composant avec les valeurs correctes des différents paramètres et à assigner les événements. Les valeurs modifiées se distinguent des valeurs par défaut par leur mise en caractères gras.

Voici le code généré par NetBeans. On notera la présence d'une méthode `main()`, destinée à faciliter les tests. On y voit qu'il faut instancier un objet de la classe `Fenetre()`. J'ai pris soin de renommer les composants, ce qu'il est conseillé de faire dès que possible.

```
package demoswing;

/**
 *
 * @author jacques
 */
public class Fenetre extends javax.swing.JFrame {

    /** Creates new form Fenetre */
    public Fenetre() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
```

```
     * WARNING: Do NOT modify this code. The content of this
        method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code
       ">
    private void initComponents() {

        txtNom = new javax.swing.JTextField();
        lblNom = new javax.swing.JLabel();
        lblPWD = new javax.swing.JLabel();
        pwdPWD = new javax.swing.JPasswordField();
        btnLogin = new javax.swing.JButton();

        setDefaultCloseOperation(javax.swing.WindowConstants.
           EXIT_ON_CLOSE);
        setTitle("Login");

        lblNom.setText("Nom:");

        lblPWD.setText("Mot␣de␣passe:");

        btnLogin.setText("Login");

                /* Pour simplier j'ai enlevé le code concernant
                   la gestion
                   du layout, particulièrement obscur */


        pack();
    }// </editor-fold>

    /**
    * @param args the command line arguments
    */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new Fenetre().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton btnLogin;
    private javax.swing.JLabel lblNom;
    private javax.swing.JLabel lblPWD;
    private javax.swing.JPasswordField pwdPWD;
    private javax.swing.JTextField txtNom;
    // End of variables declaration

}
```

5.1.3 Conseils d'utilisation

Il me semble futile de s'obliger à programmer la création d'une interface un peu complexe. L'outil Matisse inclus dans NetBeans permet de concevoir et de modifier facilement l'interface

d'une application. On peut réaliser des fenêtres (JFrame) ou des panneaux (JPannel) à intégrer dans d'autres fenêtres (par exemple dans des onglets ou des panneaux superposés). Si on veut modifier le comportement de la classe ainsi générée, on peut utiliser deux techniques :

- la première consiste à placer son code dans le fichier de la classe sous la forme de méthodes ou d'instructions dans un callback. Le danger est qu'une modification de l'interface ne vienne perturber ou effacer ce code.
- la seconde consiste à mettre un minimum de code dans la classe « graphique » et à produire une classe dérivée pour placer son code. Le code contenu dans la classe-mère se limitera à faire ce qui ne peut être fait dans la classe fille par suite de problèmes de visibilité.

5.2 Composants de base

Composants de saisie		Multiligne	Multistyle	Composants inclus
Classe	Utilisation			
`JTextField`	Champ de saisie simple	Non	Non	Non
`JPasswordField`	Champ pour les mots de passe	Non	Non	Non
`JFormatedTextField`	Lecture formatée des nombres et des dates	Non	Non	Non
`JTextArea`	Champ de saisie sur plusieurs lignes	Oui	Non	Non
`JEditorPane`	Texte formaté en RTF ou HTML	Oui	Oui	Non
`JTextPane`	Feuilles de styles, mise en forme et inclusion d'autres composants	Oui	Oui	Oui

Composants de présentation	
Classe	Utilisation
`JLabel`	Étiquette
`JToolTip`	Info bulle
`JSeparator`	Séparateur
`JProgressBar`	Barre de progression

Composants arbres et tableaux	
Classe	Utilisation
`JTree`	Arbre hiérarchique
`JTable`	Tableau à données éditables ou non

Composants de choix	
Classe	Utilisation
`JButton`	Bouton
`JCheckBox`	Case à cocher
`JToggleButton`	Bouton à bascule
`JRadioButton`	Bouton radio
`JList`	Liste de valeurs
`JComboBox`	Liste déroulante
`JSpinner`	Sélection de nombre
`JSlidder`	Choix par glissière
`JScrollBar`	Ascenseur

Composants de menu	
Classe	Utilisation
`JMenuBar`	Barre de menu (détachable)
`JMenu`	Menu
`JMenuItem`	Élément de menu
`JCheckBoxMenuItem`	Élément de menu coché
`JRadioButtonMenuItem`	Élément de menu « radio »
`JPopupMenu`	Menu contextuel

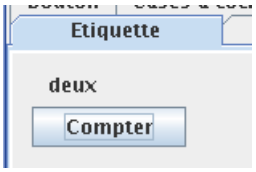
Conteneurs fenêtres	
Classe	Utilisation
`JFrame`	Fenêtre de base d'une application indépendante
`JApplet`	Fenêtre de base d'une application intégrée à un navigateur
`JWindow`	Fenêtre sans décoration
`JDialog`	Boîte de dialogue modale ou non modale
`JInternalFrame`	Fenêtre interne d'un document MDI

Conteneurs intermédiaires	
Classe	Utilisation
`JToolBar`	Barre d'outils détachable
`JTabbedPane`	Panneau à onglets
`JSplitPane`	Panneau partagé
`JScrollPane`	Panneau à ascenseur
`JLayeredPane`	Panneau à plusieurs couches
`JDesktopPane`	Panneau des fenêtres MDI
`JRootPane`	Panneau principal d'une fenêtre
`JPanel`	Panneau pour le rangement d'autres composants

Boîtes de dialogues standards	
Classe	Utilisation
`JColorChooser`	Dialogue de sélection d'une couleur
`JFileChooser`	Dialogue de sélection d'un fichier
`JOptionPane`	Boîte de dialogue (message, saisie, confirmation)

La suite de la section abordera les composants les plus utilisés.

5.2.1 `JLabel`



Description

Étiquette servant à afficher des informations non modifiables par l'utilisateur. Elle n'est modifiable que par programmation. Outre son utilisation en conjonction avec des zones de texte, elle peut servir pour afficher des informations (par exemple dans un panneau sud, une ligne de statut).

Initialisation

Utiliser la méthode `void setText(String)` ou un constructeur.

Lecture

Utiliser la méthode `String getText()`.

Événements

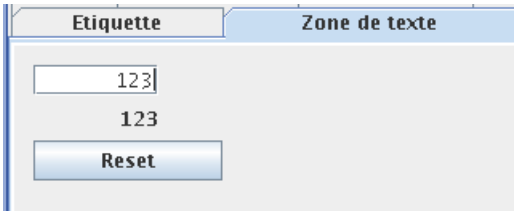
Une étiquette est peu susceptible de déclencher un événement. Les événements prévus pour elle sont très peu employés.

Exemple

Le code suivant, associé à un bouton, remplace le texte de l'étiquette `lblNombre` qui contient un nombre écrit en lettre par le nombre suivant (en faisant un cycle de cinq nombres).

```
private void btnCompterActionPerformed(java.awt.event.ActionEvent evt) {
    String[] nombres = {"un","deux","trois","quatre","cinq"};
    String sContenu=lblNombre.getText();
    int iPos=0;
    while(!nombres[iPos]equals(Contenu))
        iPos++;
    iPos=++iPos%5;
    lblNombre.setText(nombres[iPos]);
}
```

5.2.2 JTextField



Description

Cette zone de texte permet l'entrée d'un texte court pour encoder des données simples.

Initialisation

Ici encore, c'est la méthode `void setText(String)`. On peut aussi passer la valeur au constructeur, mais il n'est pas accessible dans l'environnement NetBeans.

Lecture

La méthode `String getText()` est commode pour récupérer du texte.

Événement

De nombreux événements sont liés à une zone de texte. Il peut être utile d'utiliser l'événement *KeyReleased* pour faire une mise à jour d'un autre champ en fonction de ce qui est en cours de frappe. Curieusement, l'événement *KeyTyped* vient toujours une touche en retard.

Exemple

Deux extraits de code montrent comment réinitialiser un champ et comment réagir à une modification.

```
private void txtNombreKeyReleased(java.awt.event.KeyEvent evt) {
    lblNombre2.setText(txtNombre.getText());
}
```

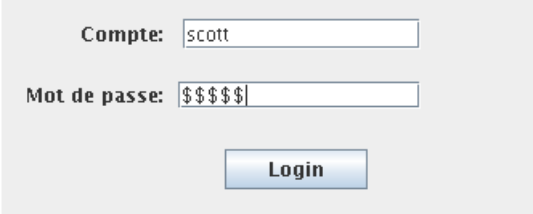
```
private void btnResetActionPerformed(java.awt.event.ActionEvent evt) {
    txtNombre.setText("0");
    lblNombre2.setText(txtNombre.getText());
}
```

Remarque

La zone de texte est un des nombreux composants permettant d'afficher du texte. Il fait partie d'une hiérarchie, qui sera évoquée plus loin. Il possède aussi un contenu gérable à travers la classe Document. On peut par ce biais :

- avoir un contrôle parfait sur le nombre de caractères contenus dans la zone de texte (et par exemple, en refuser)
- gérer plus finement l'interaction du composant avec le contexte global.

5.2.3 `JPasswordField`



Description

Très utile pour saisir un mot de passe.

Initialisation

Possible, mais pas conseillée. On peut modifier le caractère affiché par la méthode `setEchoChar(char)`, dans l'exemple, le caractère `'$'`.

Lecture

Bien que ce composant descende de JTextField, la méthode `getText()` est dépréciée. On recommande l'usage de `char[] getPassword()` dont le test est nettement plus compliqué. Nous verrons dans l'exemple deux méthodes pour opérer la comparaison des mots de passe et d'une valeur interne.

Événement

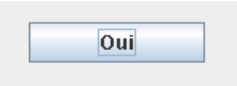
Il n'est pas conseillé de faire autre chose que lire un mot de passe à l'aide de ce composant.

Exemple

```
private void btnLoginActionPerformed(java.awt.event.ActionEvent evt) {
        String message="Mot de passe incorrect";
      // Le mot de passe est éclaté dans un tableau de char
        char[] tab = {'1','2','3','4'};
        if(Arrays.equals(pwdLogin.getPassword(),tab))
```

```
            message="Mot␣de␣passe␣correct";
        // transformation du tableau de char en String
          if(String.valueOf(pwdLogin.getPassword()).equals("1234"))
              message="Mot␣de␣passe␣correct␣-␣2";
          JOptionPane.showMessageDialog(this,message,"Connexion",
             JOptionPane.WARNING_MESSAGE);
    }
```

5.2.4 JButton



Description

Le bouton est l'un des composants les plus fréquents dans une interface. Il sert souvent à lancer des tâches ou fermer des fenêtres.

Initialisation

Souvent initialisé par son créateur, le bouton peut se voir modifier son texte par la méthode `setText()`.

Lecture

Peu fréquente, puisque seul le programme peut modifier le texte figurant sur le bouton. La méthode `getText()` fonctionne parfaitement.

Événement

Il dispose de toute une panoplie pour réagir au passage de la souris. Son événement préféré est sans nul doute *ActionPerformed* qui correspond à l'enfoncement et au relâchement du bouton.

Exemple

On voit ici une simple bascule entre deux textes (Oui et Non) affichés sur le bouton.

```
private void btnYesNoActionPerformed(java.awt.event.ActionEvent evt) {
  btnYesNo.setText(btnYesNo.getText().equals("Oui")?"Non":"Oui");
}
```

5.2.5 JCheckBox

Description

Il s'agit de la traditionnelle case à cocher, qui peut donc prendre deux valeurs. Rappelons qu'une telle case est toujours indépendante. On pourra également utiliser des instances de `JToggleButton` et `JCheckBoxMenu`, qui fonctionnent de la même façon.

Initialisation

Elle se fait au moyen de la méthode `void setSelected(boolean b)`. Certaines surcharges du constructeur permettent aussi de choisir l'état initial du composant.

Lecture

La méthode `boolean isSelected()` renvoie l'état de la case.

Événement

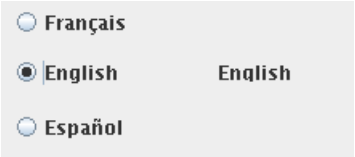
Il arrive fréquemment qu'une partie de l'interface soit modifiée par le changement d'état d'une case à cocher. Deux événements permettent de rendre compte de cela : *ActionPerformed* et *ItemStatChanged*, qui n'ont pas exactement la même signification, puisque l'état peut changer par suite d'un appel de méthode.

Exemple

Dans l'exemple suivant, on voit que l'état de deux cases (Bête et Méchant) commande l'affichage d'une description.

```
private void miseAJourDescription() {
        StringBuffer description=new StringBuffer("");
        if(chkBete.isSelected())
            description.append("Bête");
        if(chkMechant.isSelected())
            if(description.length()>0)
                description.append("␣et␣méchant");
            else
                description.append("Méchant");
        lblDescription.setText(description.toString());
}
```

5.2.6 `JRadioButton` et `ButtonGroup`



Description

Les boutons « radio » vont par groupes : seul un bouton du groupe peut être sélectionné. Leur manipulation est un peu plus délicate.

Placement sur le formulaire

Logiquement, les boutons radio devraient figurer dans un même `JPanel`, pour des raisons de lisibilité. Mais cette proximité ne suffit pas à les faire membre du même groupe. Il faut placer un objet de type `ButtonGroup` dans la feuille et associer les différents membres du groupe à ce `ButtonGroup`[3]. À partir de ce moment, les boutons seront synchronisés.

Initialisation

Par défaut, un groupe de boutons radio se présentent sans bouton préalablement sélectionné. On peut néanmoins choisir l'un des boutons à l'aide de sa méthode `setSelected(boolean)`. En cas d'appel de cette méthode, tout bouton du groupe préalablement sélectionné sera désélectionné. Il n'existe pas de moyen simple de les désélectionner tous, si ce n'est en leur adressant un message à chacun.

Lecture

La lecture individuelle d'un bouton se fait à l'aide de la méthode `boolean isSelected()`. Pour savoir quel bouton est sélectionné dans un groupe, c'est beaucoup plus complexe. Heureusement, dans la plupart des cas, le problème n'est pas tellement d'avoir accès au bouton, mais de connaître l'information que l'utilisateur a voulu transmettre. Je propose ici trois méthodes[4] :

Solution 1 : Parcourir les boutons pour déterminer quel bouton est sélectionné (voir l'exemple 1).

Solution 2 : Gérer soi-même la valeur renvoyée : cela suppose de définir une variable qui reflètera toute modification de l'état des boutons du groupe (par du code appelé par ces événements).

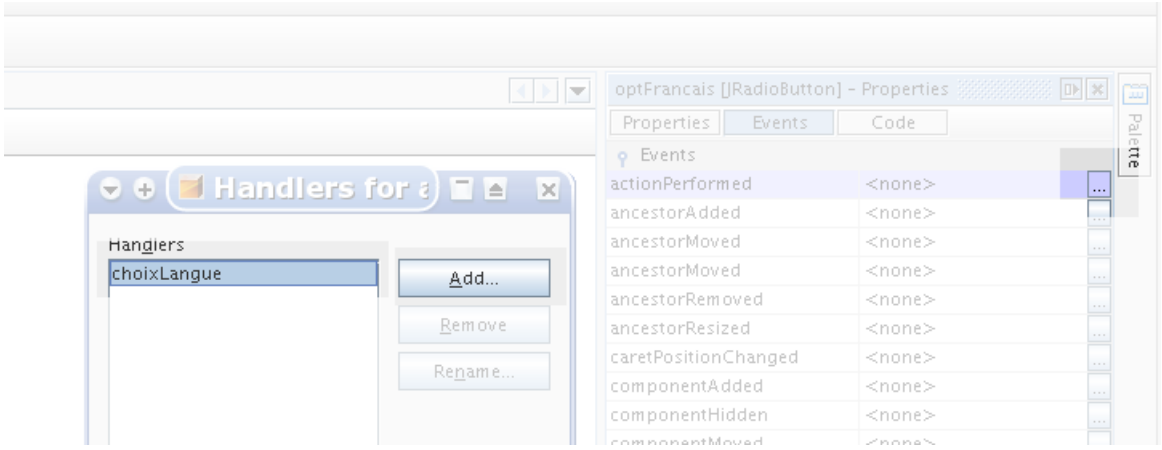
Solution 3 : Interroger directement le modèle du bouton. Cette méthode semble plus directe, mais elle présente deux inconvénients. Si aucune sélection n'est opérée, aucun modèle n'est renvoyé (valeur *null*), il n'est donc pas question d'appeler ses méthodes sans précaution. D'autre part, un modèle ne donnant pas accès aux attributs du bouton, il faut initialiser explicitement la chaîne renvoyée par `getActionCommand()`, par défaut nulle (voir exemple 2). Le comportement de cette méthode est assez irritant. Appliquée à un descendant de `JAbstractButton`, elle renvoie l'étiquette du bouton quand aucune chaîne spécifique n'a été attribuée pour le nom de la commande. Par contre, au niveau du modèle, une valeur nulle n'est pas remplacée (et ne pourrait pas l'être puisque l'étiquette n'est pas accessible). Pour couronner le tout, NetBeans recopie l'étiquette dans la liste des propriétés, mais ne fait appel à `setActionCommand()` que quand la propriété a été modifiée par le programmeur.

[3] Cela peut se faire dans NetBeans en sélectionnant les différents boutons puis en modifiant la valeur de la propriété. Il faut noter que l'objet ButtonGroup n'apparaît pas dans la hiérarchie des composants visuels, mais dans le groupe *Other Components*.

[4] J'ai trouvé sur Internet une classe `JButtonGroup` qui se dit plus performante que `ButtonGroup`. Outre le fait qu'elle s'intègre mal à NetBeans, je ne suis pas parvenu à la faire fonctionner. La principale différence de cette classe par rapport à la classe standard est qu'elle mémorise les boutons et non leur modèle.

Événement

La gestion des événements n'est pas trop simple avec l'interface de NetBeans. Si on veut éviter de devoir définir un callback par bouton, on peut définir un callback commun en tapant soi-même son nom dans la fenêtre accessible uniquement par le bouton marqué de trois points dans l'onglet events de la fenêtre des propriétés. Il n'est pas possible de faire la même chose par le menu contextuel.



L'événement `actionPerformed` est provoqué à chaque modification du groupe. Comme tous les gestionnaires d'événements, il transmet un Event dont on peut invoquer la méthode `getActionCommand()` pour obtenir le contenu de l'étiquette du bouton.

Exemples

Exemple 1 : interroger le bouton sélectionné dans un groupe Il n'est pas nécessaire de placer les boutons dans un tableau puisque l'objet `ButtonGroup` peut en renvoyer l'énumération (méthode `getElements()`).

```
for(Enumeration<AbstractButton> langues =grpLangue.getElements();
          langues.hasMoreElements();){
      AbstractButton ab = langues.nextElement();
      if(ab.isSelected())
         System.out.println(ab.getText());
}
```

Exemple 2 : faire appel au modèle du bouton sélectionné dans un groupe Il convient d'abord de recopier les étiquettes pour que l'identification du bouton soit possible (on peut évidemment préférer initialiser le nom de la commande avec une valeur différente). Le mieux est de placer la boucle dans le constructeur de la classe, juste après l'appel de `initComponents()` :

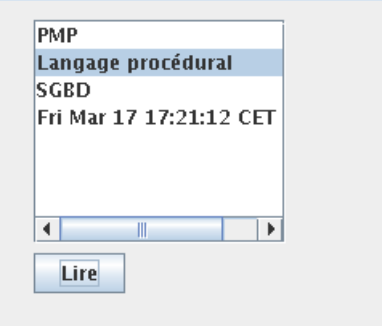
```
initComponents();
for(Enumeration<AbstractButton> enAB =grpLangue.getElements();
          enAB.hasMoreElements();){
    AbstractButton ab=enAB.nextElement();
    ab.setActionCommand(ab.getText());
}
```

L'identification de la valeur choisie est maintenant plus simple.

```
ButtonModel bm = grpLangue.getSelection();
if(bm!=null)System.out.println("M2 "+bm.getActionCommand());
```

A noter que, si on initialise l'un des boutons avant de lancer le programme, comme il n'est pas possible de désélectionner tous les boutons, on peut se passer de vérifier que la variable `bm` n'est pas nulle.

5.2.7 `JList`



Description

Les listes permettent d'afficher plusieurs valeurs et d'un sélectionner une ou plusieurs (selon le paramétrage original). Comme le contenu de la liste peut dépasser l'espace prévu à l'écran, il est conseillé de placer les listes dans un `JScrollPane`. La version 5.0 de NetBeans le fait systématiquement (les ascenseurs n'apparaissent que si c'est nécessaire).

Initialisation

L'initialisation porte sur le comportement de la sélection (`SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` ou `MULTIPLE_INTERVAL_SELECTION`) et sur le contenu de la liste . La liste ne se réduit pas nécessairement à une simple énumération de texte (ce que propose de générer NetBeans en créant un modèle à l'aide d'une classe anonyme). La liste, par l'entremise de son modèle peut contenir n'importe quel objet. Il faut simplement veiller à ce qu'il s'affiche de manière lisible (emploi de sa méthode `toString()`). Voici un exemple d'initialisation pour une liste à choix multiple contenant des objets quelconques :

```
lstFormations.setSelectionMode(
     ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
DefaultListModel model= new DefaultListModel();
lstFormations.setModel(model);
model.removeAllElements();
model.addElement("PMP");
model.addElement("Langage␣procédural");
model.addElement("SGBD");
model.addElement(new Date());
```

Lecture

Selon le mode de sélection choisi, on utilisera les méthodes `getSelectionIndex()` ou `getSelectionIndices()` pour récupérer le ou les indices des objets à récupérer depuis le modèle (voir exemple avec plusieurs indices potentiels). On notera que `getModel()` renvoie une valeur de type `ListModel` qu'on peut éventuellement caster sur le type de modèle choisi à l'initialisation si c'est nécessaire.

Événements

Différents événements sont écoutés par les listes, essentiellement liés à la gestion de la souris.

Exemple

```
private void btnLectureListeActionPerformed(java.awt.event.ActionEvent evt) {
    int[] choisis =lstFormations.getSelectedIndices();
    ListModel modele = lstFormations.getModel();
    System.out.println("=========================================");
    for(int i : choisis)
       System.out.println(modele.getElementAt(i));
}
```

5.3 Mécanisme des événements

NetBeans offre une méthode très confortable pour gérer les événements de l'interface graphique : il produit automatiquement le code d'écoute et procure un callback qu'il suffit de modifier pour qu'un événement donné produise son effet. Comme il fait appel à une classe anonyme, le code est à la limite de la lisibilité. Dans cette section, nous allons gérer manuellement les événements d'un bouton de manière à mieux en comprendre le mécanisme.

L'exemple sera simple : le survol d'un bouton par la souris va faire changer la couleur du fond du bouton.

5.3.1 Utilisation de classes publiques ou privées

Un événement n'est rien d'autre qu'une modification du contexte d'une application qui doit entraîner des répercutions sur d'autres parties de l'application : l'appui sur un bouton doit ouvrir une fenêtre ou incrémenter la valeur d'une variable. L'interface graphique possède des dizaines d'événements internes qui font que certaines manipulations du clavier ou de la souris entraînent des modifications de l'aspect de l'interface. En général, le programmeur ne doit pas s'en soucier. Si le bouton semble s'enfoncer quand on pousse dessus, c'est normal. Par contre, on voudrait que l'enfoncement s'accompagne d'une action dans l'application, alors que les concepteurs du bouton n'avaient aucune connaissance de ce que contient l'application. La gestion des évènements permet de mettre en relation deux ou plusieurs classes qui ne se connaissent pas mutuellement.

L'ajout d'une gestion d'événements à un composant susceptible de les gérer se déroule en trois phases : définir une classe *listener* liée au type d'événement qu'on veut prendre en compte, prévoir les traitements spécifiques (à placer dans les méthodes de la classe *listener*

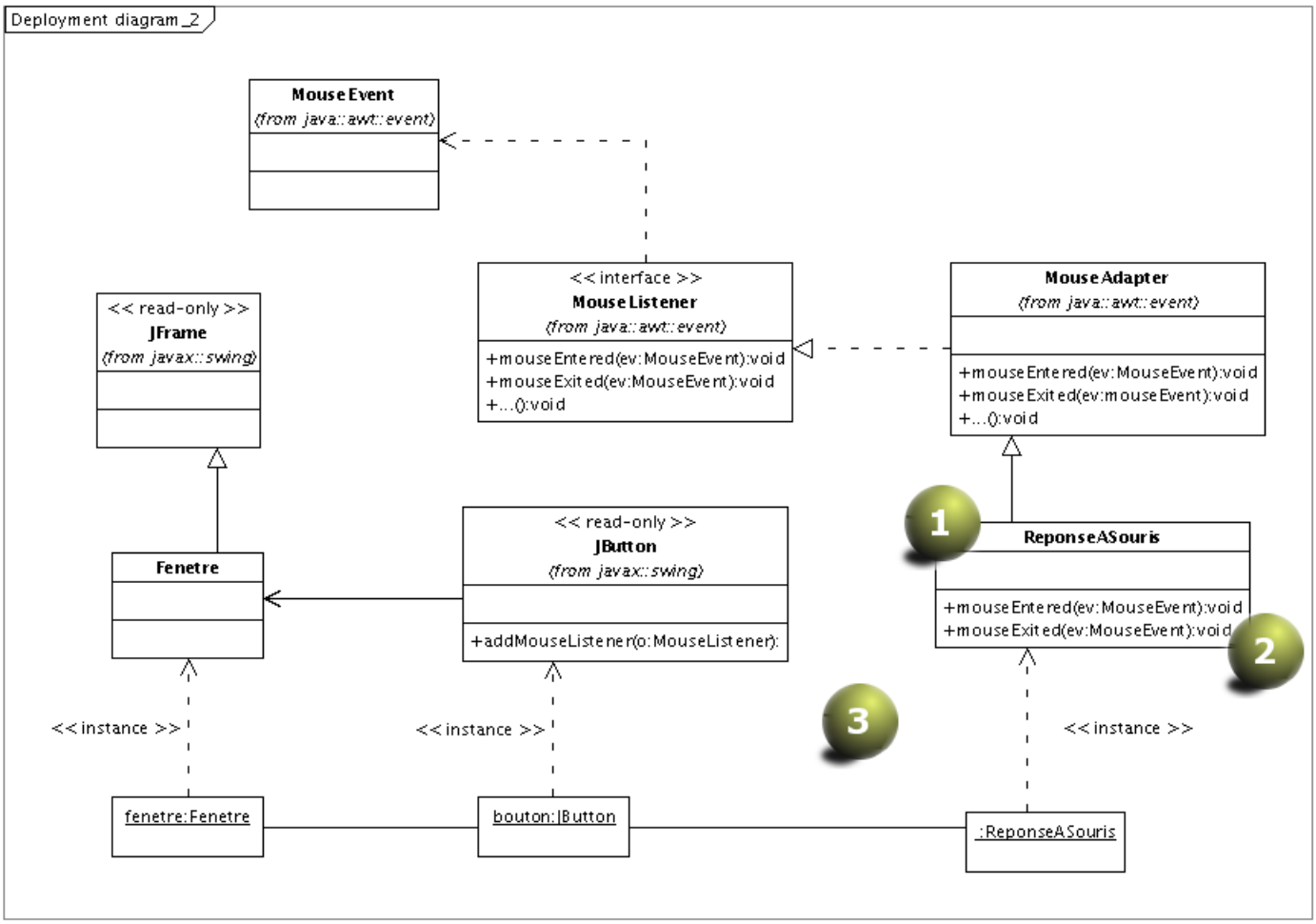


FIG. 5.2 – Exemple de gestion d'événement

ou indirectement dans des callback) et enfin établir une liaison entre l'objet susceptible de provoquer l'événement et la classe qui va le gérer concrètement.

Étape n°1 : définition d'une classe *listener*

Heureusement Java, a prédéfini une série d'événements qui peuvent se produire dans une interface graphique. Pour créer une classe *listener*, il suffit alors de dériver la nouvelle classe d'une classe existante et de surdéfinir les méthodes qui correspondent aux événements précis que l'on va gérer. Il est aussi possible de réaliser une interface, mais cela nous oblige à définir du code pour les méthodes qui ne nous intéressent pas. Les classes prédéfinies possèdent des méthodes qui ne font rien et ne doivent pas nécessairement être redéfinies.

Pour ce qui concerne la gestion de la souris, on peut par exemple réaliser l'interface `MouseListener` qui comporte cinq méthodes à définir : `mouseClicked`, `mousePressed`, `mouseReleased`, `mouseEntered` et `mouseExited`. Si on ne veut prendre en compte que les deux dernières méthodes, ce qui est notre cas, il sera plus simple de dériver notre classe de `MouseAdapter`.

```
class ReponseASouris extends MouseAdapter{
// code à implémenter
}
```

Étape n°2 : spécification des traitements à réaliser

En fonction de la classe définie lors de l'étape précédente, on va définir les méthodes spécifiques permettant les traitements. Comme toujours en informatique, on va essayer de favoriser la réutilisation du code. Deux méthodes sont possibles :

– on peut définir des traitements qui ne mentionnent pas réellement les objets concernés. Par exemple, les deux méthodes suivantes vont mettre en rouge la surface de n'importe quel bouton qui déclencherait l'événement. On y parvient par l'appel de la méthode `getSource()` sur l'objet passé en argument, qui n'est autre que l'événement déclencheur[5].

```
public void mouseEntered(java.awt.event.MouseEvent e) {
     JButton b = (JButton) e.getSource();
     oldColor = b.getBackground();
     b.setBackground(new Color(255,128,128));
}
public void mouseExited(java.awt.event.MouseEvent e) {
     JButton b = (JButton)e.getSource();
     b.setBackground(oldColor);
}
```

– l'utilisation de callback permet de partager facilement des traitements entre plusieurs événements. Dans ce cas, les deux méthodes prévues vont simplement appeler deux méthodes de la classe correspondant à l'interface graphique. On pourra placer le code effectif dans les callbacks.

```
public void mouseEntered(java.awt.event.MouseEvent e) {
  modifierContexte(java.awt.event.MouseEvent e));
}
public void mouseExited(java.awt.event.MouseEvent e) {
  rétablirContexte(java.awt.event.MouseEvent e));
}
```

Il faudra au moins définir ces deux nouvelles méthodes. On pourra ensuite leur affecter du code, soit en l'introduisant dans la classe qui la définit (c'est ce que NetBeans propose de faire dans la fenêtre en cours de définition) ou dans une classe qui en dérive, par redéfinition.

Étape n°3 : ajout d'un écouteur d'événements à l'objet concerné

Il reste à informer l'objet concerné qu'il est écouté pour les événements prévus. Un simple appel de méthode réalise cette tâche :

```
btnMessage.addMouseListener(new ReponseASouris());
```

5.3.2 Utilisation de classes anonymes

Un peu de réflexion nous permet de constater que dans la plupart des cas, la classe *listener* n'a qu'une seule instance, directement passée en paramètre à `addXXXListener()`. C'est

[5] La variable d'instance `oldColor` doit néanmoins exister dans la classe. On pourrait s'en passer à condition de postuler que le bouton a la couleur par défaut.

un cas d'école pour utiliser une classe anonyme (ce que fait d'ailleurs NetBeans). On notera que si plusieurs objets partagent les mêmes réactions face à un événement, il reste possible d'utiliser des classes anonymes à condition que leurs méthodes fassent appel à des callbacks. Il n'est pas sûr cependant que l'optimisation soit au rendez-vous.

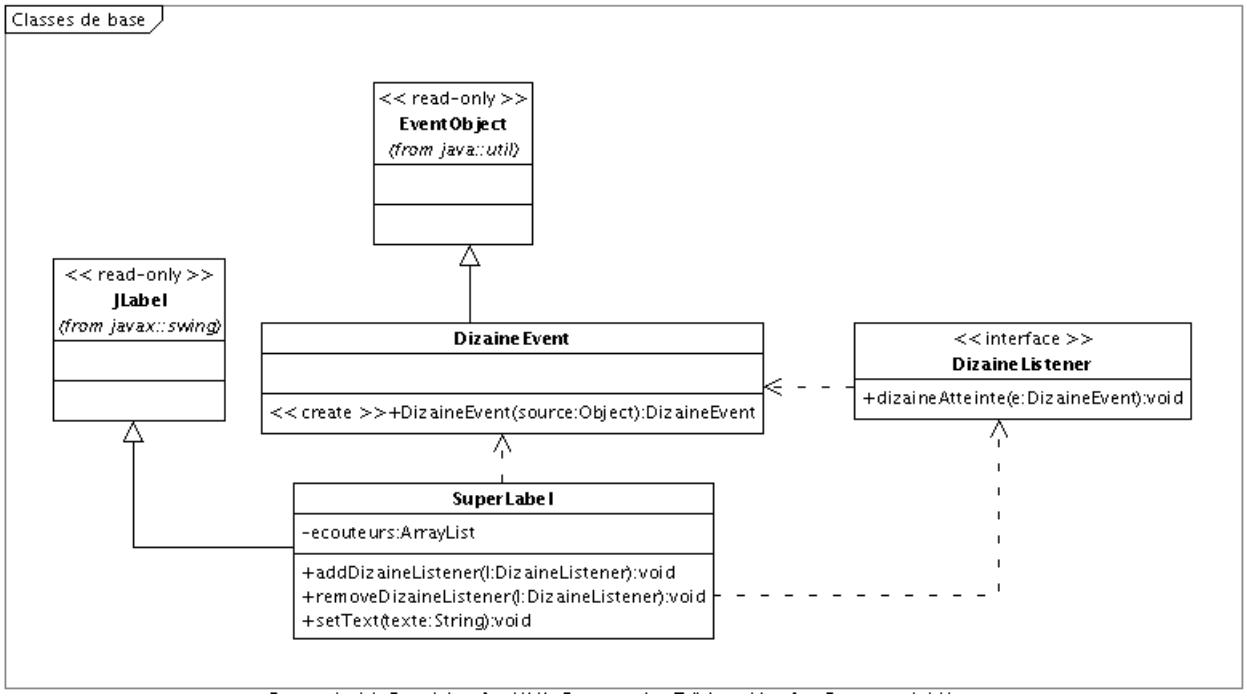
Voici comment on gérerait nos fonds colorés avec une classe anonyme :

```
btnMessage.addMouseListener(new MouseAdapter(){
      public void mouseExited(java.awt.event.MouseEvent e) {
      // code à prévoir
      }
      public void mouseEntered(java.awt.event.MouseEvent e) {
      // code à prévoir
      }
});
```

5.4 Créer ses propres événements

Un programmeur peut parfois trouver avantage à ajouter ses évènements à un programme. On pourrait citer l'exemple d'un afficheur numérique qui permettrait d'afficher des secondes, des minutes et des heures et générerait des événements lorsqu'il dépasse sa capacité maximale par le haut ou par le bas. Nous prendrons un exemple plus simple : un label qui émet un événement chaque fois que son contenu devient un nombre multiple de 10.

Java ne propose pas de mécanisme prédéfini pour gérer les événements, à part une classe `EventObject` de base. Le mécanisme d'une gestion d'événement est donc basé sur une convention, que le programmeur est invité à imiter.



La conception d'un nouveau type d'événement consiste à faire dériver une nouvelle classe de `EventObject` : c'est la classe `DizaineEvent` du schéma. Il suffira d'écrire un constructeur qui pourra passer un paramètre objet au constructeur de la classe mère, afin que la méthode

`getSource()` puisse renvoyer une référence à l'objet qui a créé l'événement. Il faut ensuite définir une interface d'écoute (`DizaineListerner`) en relation avec ce nouvel événement. Cette interface va définir autant de méthodes qu'il y a de variantes à l'événement. Dans notre cas, seule la méthode `dizaineAtteinte()` est nécessaire. Dans le cas de l'afficheur numérique, il aurait fallu distinguer des dépassements de deux types (*underflow* et *overflow*). On peut aussi définir une classe abstraite qui prévoit des méthodes non abstraites, mais ne faisant rien, si on ne veut pas forcer la redéfinition des méthodes qui seront sans objet dans un contexte donné. Dans ce cas, la classe, qui suit en fait le modèle *adaptateur*, implémentera l'interface (voire même la remplacera). Ce n'est pas la solution de notre exemple, où il n'y a de toute façon qu'une seule méthode.

Il reste à implémenter le mécanisme des écouteurs au sein des classes qui vont gérer les événements. Cette implémentation se fait à l'aide d'une variable d'instance privée servant à mémoriser les objets à l'écoute de l'événement et deux accesseurs spéciaux qui permettent d'ajouter ou de retirer un *listener*[6]. Le code peut prévoir l'impossibilité d'ajouter deux fois le même *écouteur* à la liste. La manière de mémoriser ces écouteurs est laissée à l'appréciation du programmeur. J'ai choisi un *ArrayList*, que le formalisme de Poseidon ne m'a pas permis de rendre générique : en fait c'est un `ArrayList<DizaineListener>`. Le schéma vu plus haut montre les relations entre les classes : généralisation entre l'événement et la classe `EventObject`, et trois dépendances entre les deux classes et l'interface, ces dépendances s'expliquant par le type des paramètres. Voici le code complet des deux classes et de l'interface.

```
public class DizaineEvent extends EventObject {

    public DizaineEvent(Object source) {
        super(source);
    }

}
public interface DizaineListener {

    void dizaineAtteinte(DizaineEvent e);
}

public class SuperLabel extends JLabel {

    private ArrayList<DizaineListener> ecouteurs =
```

[6]Java définit un mécanisme plus complexe en gérant tous les écouteurs au niveau de la classe de base de son interface graphique (`Component`), cela lui permet de gérer tous les événements de manière centralisée. Le programmeur aura sans doute du mal à intégrer ses événements dans ce contexte, s'il ne possède pas une vue bien claire du fonctionnement interne de Swing. L'un des avantages de cette technique est de favoriser le passage des événements à un conteneur de plus haut niveau si c'est nécessaire. On voit le soin mis dans la programmation des méthodes d'ajout :

```
public synchronized void addMouseListener(MouseListener l) {
       if (l == null) {
           return;
       }
       mouseListener = AWTEventMulticaster.add(mouseListener,l);
       newEventsOnly = true;
       // if this is a lightweight component, enable mouse events
       // in the native container.
       if (peer instanceof LightweightPeer) {
           parent.proxyEnableEvents(AWTEvent.MOUSE_EVENT_MASK);
       }
   }
```

```
            new ArrayList<DizaineListener>();
        public SuperLabel() {
            super();
        }

        public void addDizaineListener(DizaineListener l){
            if(! ecouteurs.contains(l))
                ecouteurs.add(l);
        }

        public void removeDizaineListener(DizaineListener l){
            ecouteurs.remove(l);
        }

        public void setText(String texte){
            super.setText(texte);
            try{
                int valeur = Integer.valueOf(getText());
                if(valeur%10==0){
                    DizaineEvent e = new DizaineEvent(this);
                    for(DizaineListener l:ecouteurs){
                        l.dizaineAtteinte(e);
                    }
                }
            } catch(Exception e) {
            }
        }
    }
```

Dans la définition de `SuperLabel`, la méthode `setText()` permet de voir comment l'événement se produit : quand la nouvelle valeur est un multiple de 10, on crée un événement en passant l'instance responsable de cette création au constructeur de l'événement[7]. La boucle qui suit va parcourir tous les objets qui écoutent l'événement et active le code de leur méthode `dizaineAtteinte()`.

Le code ci-dessous montre l'installation de deux listeners de l'événement `DizaineEvent`. Le premier est une instance d'une classe anonyme dont la méthode `dizaineAtteinte()` se contentera d'afficher un message sur la console. Il se met en place dans la méthode `initSuperLabel()` qui gère l'insertion de l'étiquette sur la fenêtre

```
    private void initSuperLabel() {
        lblNombre2 = new SuperLabel();
        lblNombre2.setHorizontalAlignment(javax.swing.SwingConstants.RIGHT);
        lblNombre2.setText("1");
        add(lblNombre2, new org.netbeans.lib.awtextra.AbsoluteConstraints(50,
            30, 110, 20));
        lblNombre2.addDizaineListener(new DizaineListener(){
            public void dizaineAtteinte(DizaineEvent e) {
                System.out.println("Nouvelle␣dizaine␣dans␣"+e.getSource());
            }
```

[7] Il faudrait idéalement mémoriser la valeur antérieure de l'objet SuperLabel, de manière à ce que l'événement ne se produise pas au cas où on affecte une deuxième fois la même valeur, mais cela rendrait le code plus lourd dans un exemple.

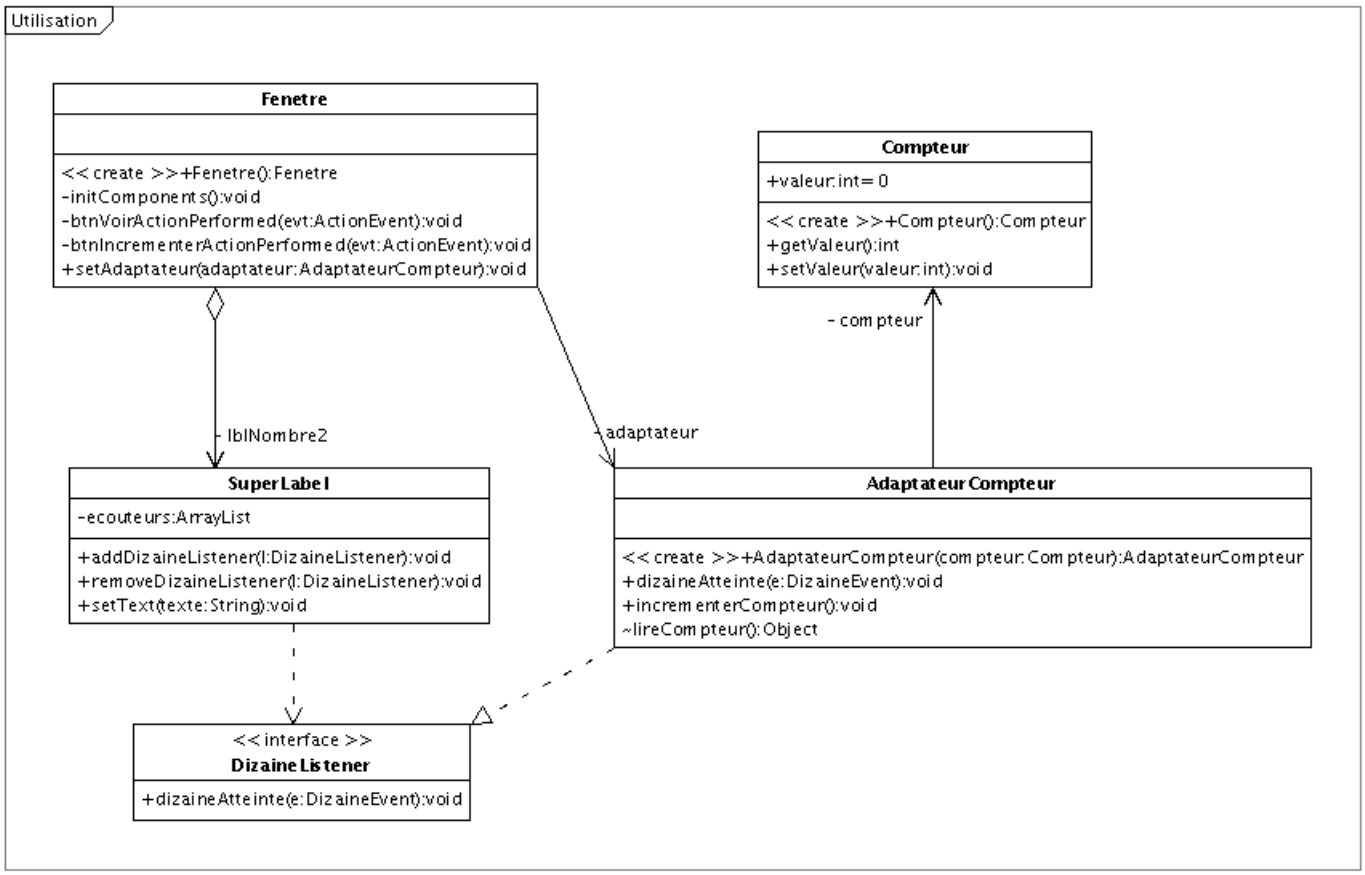


FIG. 5.3 – Utilisation de `DizaineEvent`

```
        });
    }
```

Le second listener est une instance de `AdaptateurCompteur`, un adaptateur qui va relayer l'événement vers un compteur externe qu'il est seul à connaître. Les détails de l'implémentation sont laissés dans le code source complet, disponible sur Internet. La variable d'instance `adaptateur` est initialisée par un accesseur. L'inscription de l'adaptateur parmi les écouteurs de l'événement est placée dans cet accesseur[8]. Comme le montre le schéma, les `AdaptateurCompteurs` implémentent évidemment l'interface `DizaineListener`.

```
public void setAdaptateur(AdaptateurCompteur adaptateur) {
    this.adaptateur = adaptateur;
    if(adaptateur!=null)
        lblNombre2.addDizaineListener(adaptateur);
}
```

[8]Comme rien n'empêche d'utiliser plusieurs fois l'accesseur, il faudrait idéalement penser à retirer de la liste des écouteurs l'adaptateur installé lors du précédent appel.