NFP121 — Programmation Avancée
—
Interfaces graphiques avec Java/Swing

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

Motivations

Objectifs de ce cours :

- Savoir construire une interface graphique avec Java/Swing ;
- Voir un exemple d'application complexe (l'API Java/Swing) ;
- Voir un exemple réel de mise en pratique des concepts objets ;
- Comprendre la programmation événementielle et son implantation en Java.

Plan du cours :

- Principe d'une interface utilisateur
- Construction de la présentation (vue)
- La gestion des événements
- MVC passif et actif
- Conclusion

Partie 1 : Principe d'une interface utilisateur

Exercice fil rouge : réaliser un compteur

Exercice 1 On veut développer une application permettant d'incrémenter la valeur d'un compteur ou de le remettre à zéro.
Plusieurs interfaces homme/machine seront développées.

1.1 Décrire la logique de cette application.

1.2 *Interface textuelle.* Les touches +, `0` et `Q` permettent d'incrémenter le compteur, de le remettre à zéro ou de quitter.

1.3 *Ligne de commande.* Par exemple, si les arguments sont + 0 + +, le compteur prend successivement les valeurs 1, 0, 1 et 2.

1.4 *Menus textuels.* Un menu permet d'incrémenter le compteur, le remettre à zéro ou quitter l'application.

1.5 *Interface graphique.* La valeur du compteur est affichée et trois boutons permettent de l'incrémenter, le remettre à zéro et quitter l'application.

Analyse de l'exercice 1

Exercice 2 : Analyse de l'exercice précédent

Posons nous des questions sur la manière de résoudre l'exercice 1.

2.1 Qu'est-il possible de factoriser entre les quatre applications ?

2.2 Que faut-il changer dans les applications si le compteur doit pouvoir être arbitrairement grand ?

2.3 En déduire ce qu'il est conseillé de faire avant de développer les 4 IHM.

Partie 1 : Principe d'une interface utilisateur

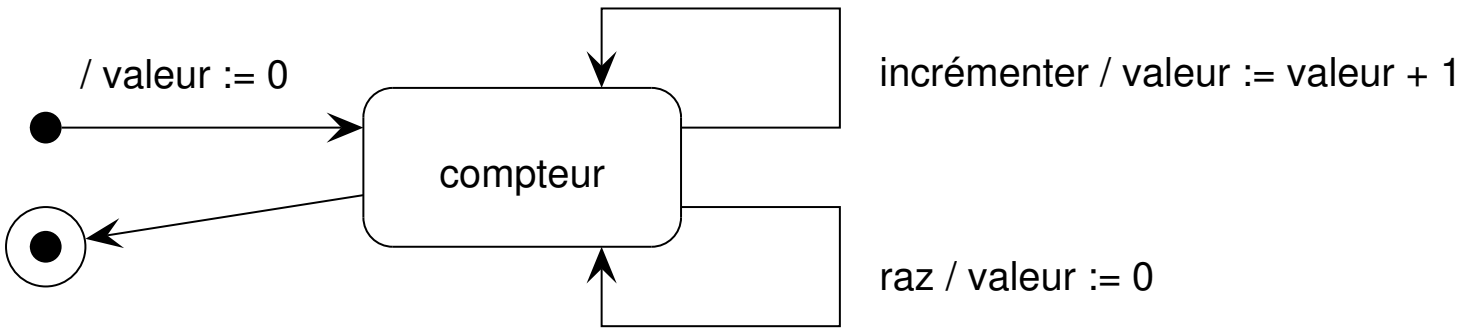
UML pour modéliser la dynamique de l'application

Principe : utiliser UML pour décrire la logique d'une application :

- les diagrammes de machine à états pour décrire, *de manière exhaustive*, la réaction de l'application aux événements extérieurs ;
- les diagrammes de séquence pour décrire des exemples d'utilisation
  - au niveau externe (seulement un objet système) ;
  - au niveau technique : en détaillant les objets du système ;
  - au niveau réalisation : avec les objets de la solution.

  Remarque : ces diagrammes sont utilisés au fur et à mesure de l'avancement du développement.

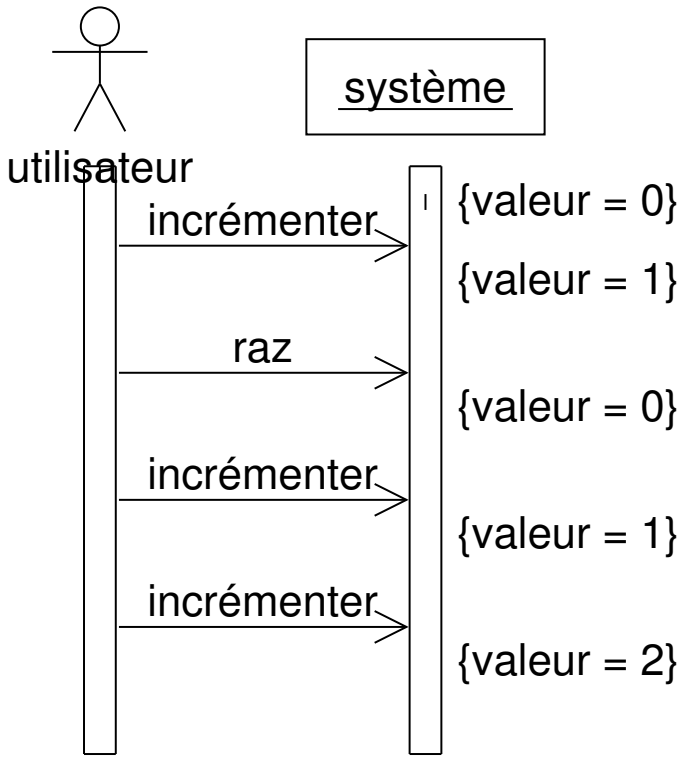
Diagramme de machine à états du compteur



- Les événements utilisateur (externes) sont :
  - incrémenter : augmente de 1 la valeur du compteur ;
  - raz : met à zéro la valeur du compteur.
- Lors du démarrage de l'application, la valeur du compteur est 0.

Diagrammes de séquence du compteur

Diagramme externe :



Diagrammes techniques et de réalisation :

- Sont faits plus tard (voir T. 21).

La classe Compteur

Du diagramme de machine à états, on peut en déduire le code de la classe Compteur.

```
1   public class Compteur {
2       private int valeur;     // valeur du compteur
3
4       /** Initialiser la valeur du compteur.
5        *@param v la valeur initiale du compteur */
6       public Compteur(int v)      { this.valeur = v; }
7
8       /** Augmenter d'une unité le compteur */
9       public void incrémenter()       { this.valeur++; }
10
11      /* Obtenir la valeur du compteur.
12       * @return la valeur du compteur.  */
13      public int getValeur()      { return this.valeur; }
14
15      /* Remettre à zéro le compteur */
16      public void raz()           { this.valeur = 0; }
17  }
```

Le *modèle*, ici Compteur.java, est défini une fois pour toute.
En général, le modèle ne se réduit pas à une seule classe !

Partie 1 : Principe d'une interface utilisateur

L'interface textuelle

```
1  public class CompteurTexte {
2      public static void main(String[] args) {
3          Compteur cptr = new Compteur(0);
4          String action;
5          do {
6              System.out.println("Compteur = " + cptr.getValeur());
7              action = Console.readLine("Action : ");
8              if (action.equals("+")) {
9                  cptr.incrémenter();
10             } else if (action.equals("0")) {
11                 cptr.raz();
12             } else if (! action.equals("Q")) {
13                 System.out.println("Action inconnue !");
14             }
15         } while (! action.equals("Q"));
16     }
17 }
```

Interface en ligne de commande

```
1  public class CompteurLigneCommande {
2      public static void main(String[] args) {
3          Compteur cptr = new Compteur(0);
4          for (int i = 0; i < args.length; i++) {
5              String action = args[i];
6              if (action.equals("+")) {
7                  cptr.incrémenter();
8              } else if (action.equals("0")) {
9                  cptr.raz();
10             }
11         }
12         System.out.println("Compteur = " + cptr.getValeur());
13     }
14 }
```

Compteur avec menu textuel

Pour utiliser les menus textuels, il faut :

- construire les commandes spécifiques du compteur
- construire le menu du compteur.

Remarque : On se limite à construire le menu et il se gère « tout seul » :
Présentation et gestion du menu sont programmées dans la classe Menu.

Compteur avec menu textuel

Pour utiliser les menus textuels, il faut :

- construire les commandes spécifiques du compteur

```
1  abstract public class CommandeCompteur implements Commande {
2      protected Compteur cptr;
3      public CommandeCompteur(Compteur c)     { this.cptr = c; }
4  }
```

```
1  public class CommandeIncrementer extends CommandeCompteur {
2      public CommandeIncrementer(Compteur c)  { super(c); }
3      public void executer()          { cptr.incrementer(); }
4      public boolean estExecutable()      { return true; }
5  }
```

- construire le menu du compteur.

```
1  public class CompteurMenu {
2      public static void main(String[] args) {
3          Compteur compteur = new Compteur(0);
4          Menu principal = new Menu("Menu", new CommandeAfficheurCptr(compteur));
5          principal.ajouter("Incrémenter", new CommandeIncrementer(compteur));
6          principal.ajouter("RAZ", new CommandeRAZ(compteur));
7          principal.gerer();
8      }
9  }
```

Remarque : On se limite à construire le menu et il se gère « tout seul » :
Présentation et gestion du menu sont programmées dans la classe Menu.

Compteur avec interface graphique

Pour développer une interface graphique pour le compteur, il faut :

- définir le **MODÈLE** de l'application (la classe Compteur) ;
- définir l'« ergonomie » de l'interface graphique (la **VUE**) ;
- programmer la réaction aux actions de l'utilisateur sur les éléments de l'interface graphique (le **CONTRÔLEUR**).

Compteur avec interface graphique

Pour développer une interface graphique pour le compteur, il faut :

- définir le **MODÈLE** de l'application (la classe Compteur) ;
- définir l'« ergonomie » de l'interface graphique (la **VUE**) ;
  L'interface graphique doit faire apparaître la valeur du compteur au centre d'une fenêtre et trois boutons en bas : INC pour incrémenter le compteur, RAZ pour le remettre à zéro et Quitter pour arrêter.

- programmer la réaction aux actions de l'utilisateur sur les éléments de l'interface graphique (le **CONTRÔLEUR**).

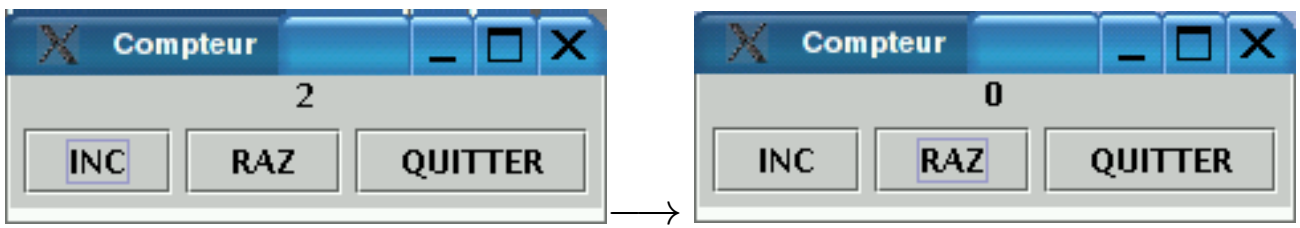
Compteur avec interface graphique

Pour développer une interface graphique pour le compteur, il faut :

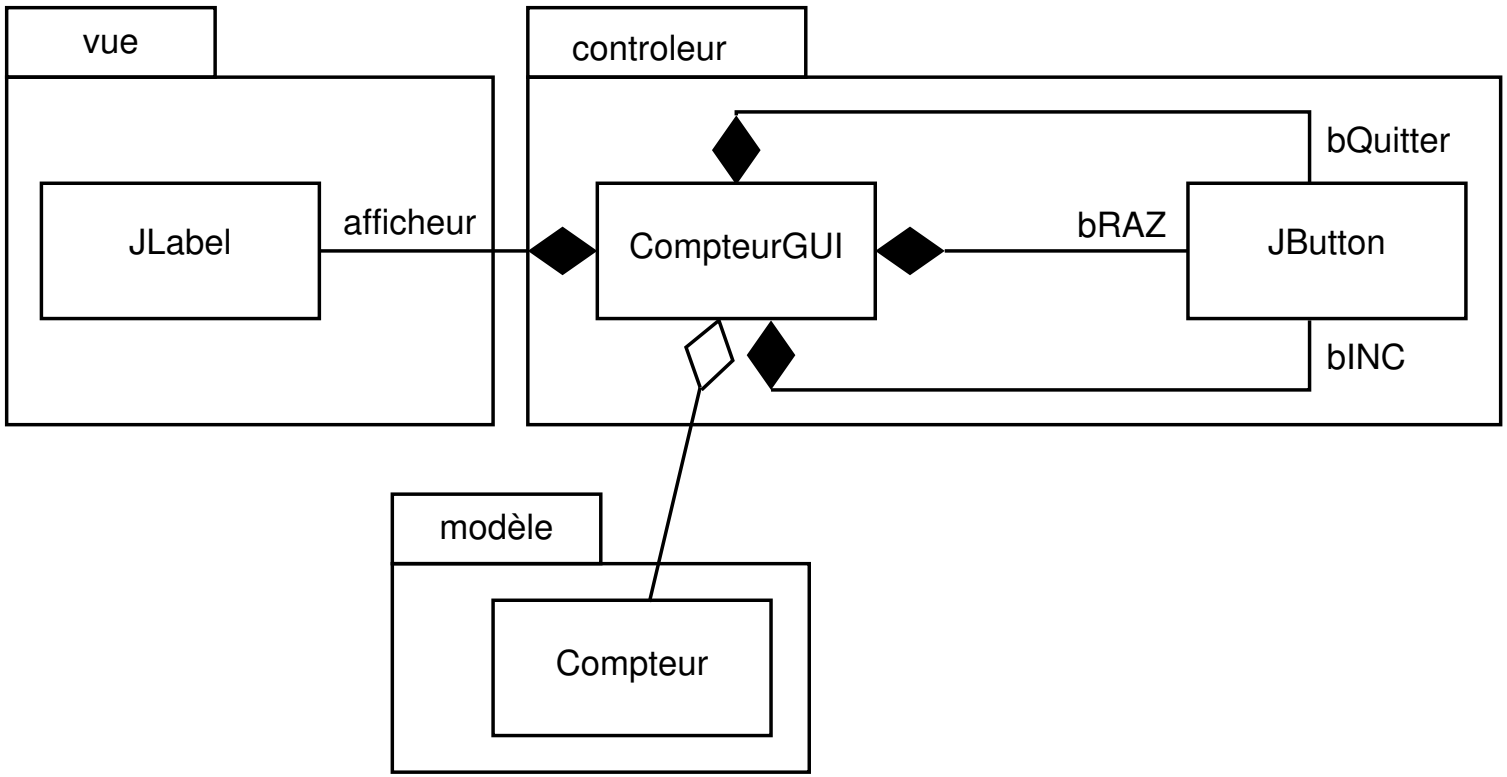
- définir le **MODÈLE** de l'application (la classe Compteur) ;
- définir l'« ergonomie » de l'interface graphique (la **VUE**) ;
  L'interface graphique doit faire apparaître la valeur du compteur au centre d'une fenêtre et trois boutons en bas : INC pour incrémenter le compteur, RAZ pour le remettre à zéro et Quitter pour arrêter.

- programmer la réaction aux actions de l'utilisateur sur les éléments de l'interface graphique (le **CONTRÔLEUR**).

  Exemple : l'utilisateur clique sur RAZ, le compteur doit prendre la valeur 0.



Construire la vue : diagramme de classes

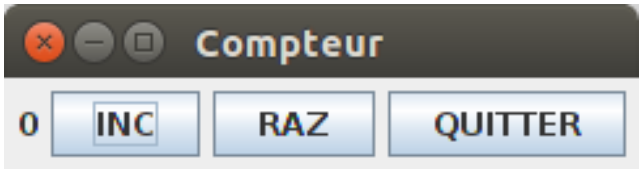


L'interface graphique est composée de :

- un label (JLabel) pour afficher la valeur du compteur ;
- trois boutons (JButton) pour INC, RAZ et Quitter ;

Le Compteur factorise la partie « métier » de l'application (le modèle).

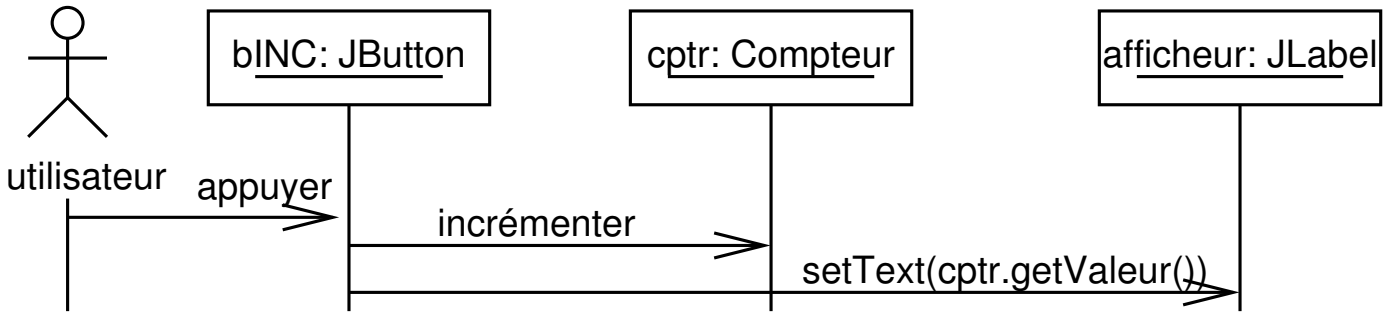
Construire la vue : code Java




```
1  import javax.swing.*;
2
3  class CompteurGUI {
4          public CompteurGUI(Compteur compteur) {
5                  JFrame fenetre = new JFrame("Compteur");
6
7                  java.awt.Container contenu = fenetre.getContentPane();
8                  contenu.setLayout(new java.awt.FlowLayout());
9
10                 JLabel afficheur = new JLabel("" + compteur.getValeur());
11                 contenu.add(afficheur);
12
13                 JButton bINC = new JButton("INC");
14                 contenu.add(bINC);
15                 JButton bRAZ = new JButton("RAZ");
16                 contenu.add(bRAZ);
17                 JButton bQuitter = new JButton("QUITTER");
18                 contenu.add(bQuitter);
19
20                 fenetre.pack();          // dimmensionner la fenêtre
21                 fenetre.setVisible(true);       // la rendre visible
22         }
23
24         public static void main(String[] args) {
25                 new CompteurGUI(new Compteur(0));
26 }       }
```

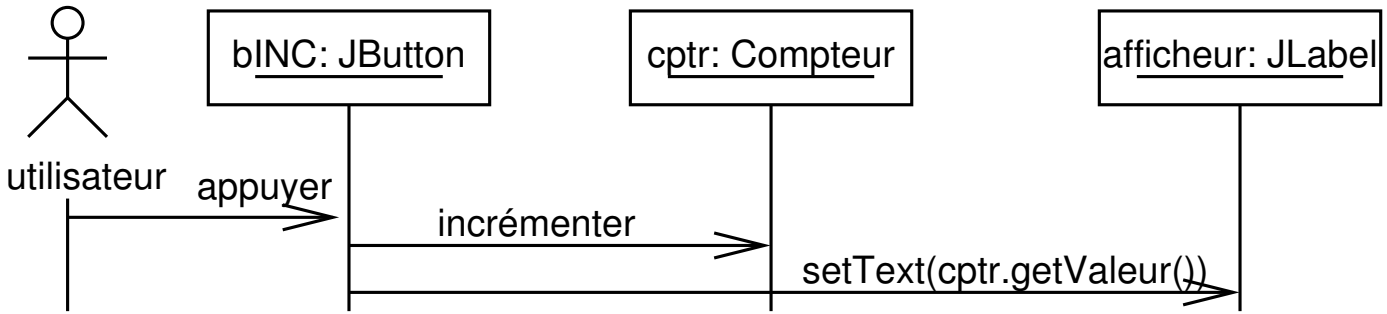
Scénarios pour comprendre le fonctionnement

Scénario 1 : la valeur du compteur est 3 et l'utilisateur appuie sur INC.

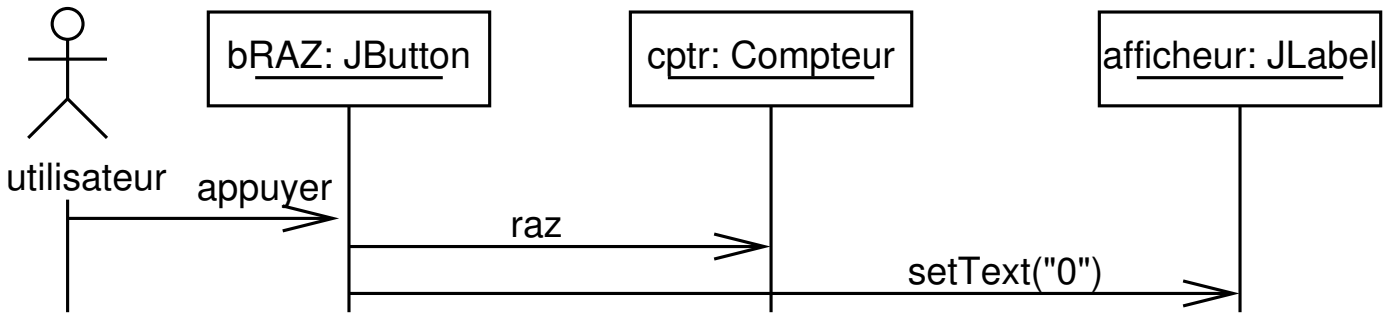


Scénarios pour comprendre le fonctionnement

Scénario 1 : la valeur du compteur est 3 et l'utilisateur appuie sur INC.

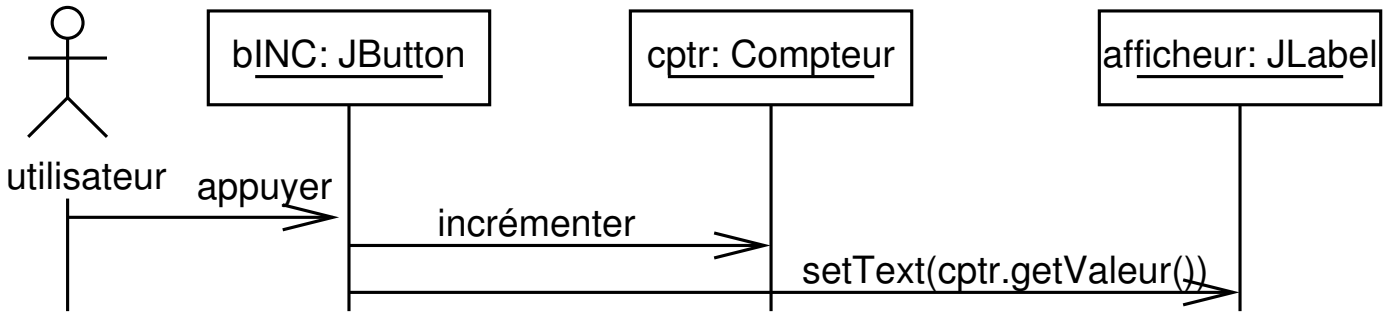


Scénario 2 : la valeur du compteur est 3 et l'utilisateur appuie sur RAZ.

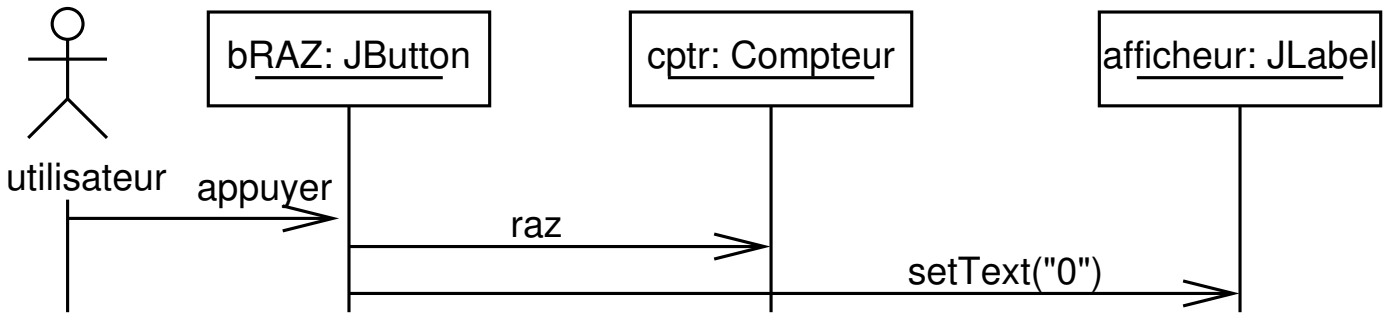


Scénarios pour comprendre le fonctionnement

Scénario 1 : la valeur du compteur est 3 et l'utilisateur appuie sur INC.



Scénario 2 : la valeur du compteur est 3 et l'utilisateur appuie sur RAZ.



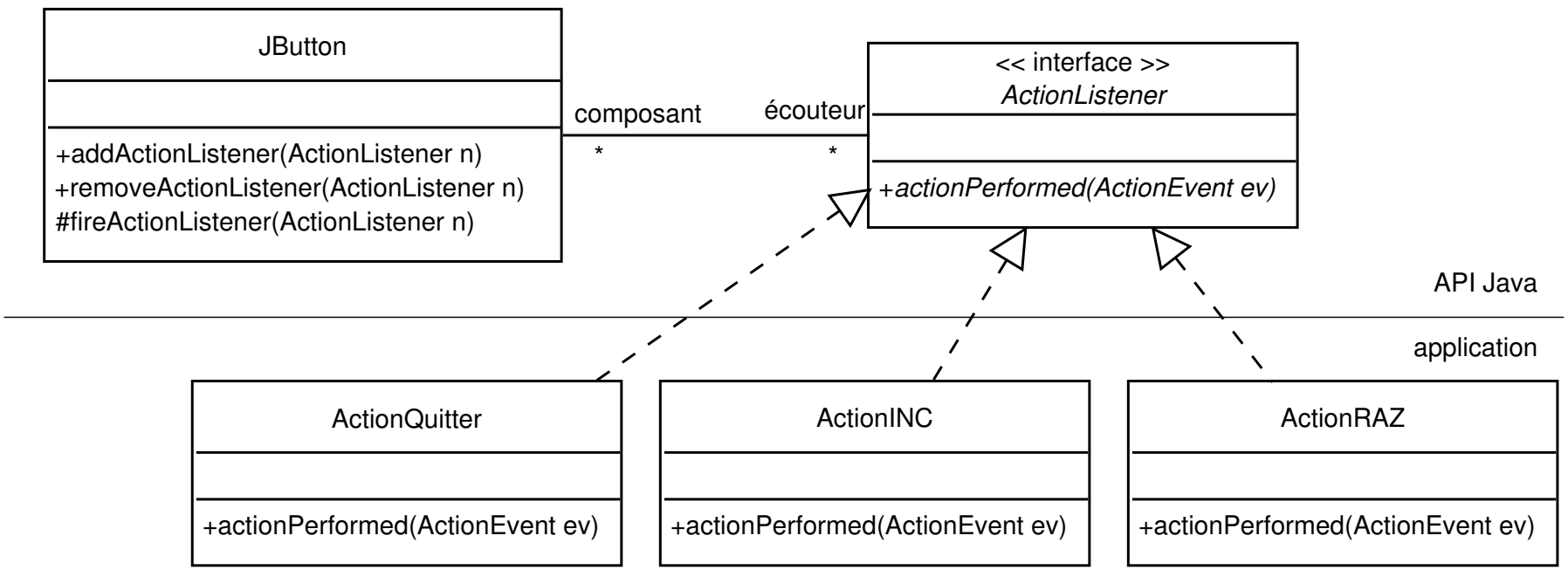
Questions : A-t-on une seule classe JButton ou deux ?
Comment associer des réactions différentes aux boutons bINC et bRAZ ?

Architecture pour programmer la réaction d'un bouton

Principe : Comme pour le menu, on ajoute une interface ActionListener qui abstrait la commande à exécuter lorsque le bouton est appuyé.

Il suffit alors de :

- réaliser cette interface pour définir les actions concrètes ;
- inscrire ces actions auprès des boutons concernés.



Programmation du bouton Quitter

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class CompteurGUIsimple {
6      public CompteurGUIsimple(final Compteur compteur) {
7          // Construction de la vue
8          ...
9
10         // Définition du contrôleur
11         bQuitter.addActionListener(new ActionQuitter());
12     }
13
14     public static void main(String[] args) {
15         new CompteurGUIsimple(new Compteur(0));
16     }
17 }
18
19 class ActionQuitter implements ActionListener {
20     public void actionPerformed(ActionEvent ev) {
21         System.exit(0);
22     }
23 }
```

Une approche similaire sera utilisée pour programmer la réaction des boutons INC et RAZ.

Partie 2 : Construction de la présentation (vue)

Paquetages pour les interfaces graphiques

Java fournit deux paquetages principaux :

- `java.awt` (*Abstract Window Toolkit*) :
  - Utilise les composants graphiques natifs ;
  - Peut avoir un comportement différent suivant la plate-forme ;
  - Limité aux caractéristiques communes à toutes les plates-formes cibles ;
  - Temps d'exécution assez rapide.
- `javax.swing`, initialement JFC (Java Foundation Classes)
  - Bibliothèque écrite en 100% *pure Java* ;
  - Bibliothèque très riche proposant des composants évolués (arbres, tables, etc.)
  - Construite au dessus de la partie portable de AWT ;
  - Application du MVC (*look & feel* modifiable) ;
  - Exécution assez lente.

Partie 2 : Construction de la présentation (vue)

Premier exemple

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  public class HelloGUI {
5
6      public static void main(String[] args) {
7          // Construire une fenêtre principale
8          JFrame fenetre = new JFrame("Je␣débute␣!");
9
10         // Construire un message texte
11         JLabel message = new JLabel("Bonjour␣!", SwingConstants.CENTER);
12
13         // Récupérer le conteneur de la fenêtre principale
14         Container contenu = fenetre.getContentPane();
15
16         // Ajouter le message dans le conteneur
17         contenu.add(message);
18
19         fenetre.pack();     // Calculer de la taille de la fenêtre
20         fenetre.setVisible(true);   // Rendre le fenêtre visible
21
22         // ...
23         // fenetre.dispose();   // Libérer les ressources utilisées
24     }
25
26 }
```

Commentaires sur ce premier exemple

- JFrame définit une fenêtre de premier niveau ;
- JLabel est un composant élémentaire ;
- getContentPane() permet de récupérer le conteneur de composants de la fenêtre pour y ajouter le message (JLabel) ;
- fenetre.pack() demande à la fenêtre de calculer sa dimension idéale en fonction des composants qu'elle contient ;
- fenetre.setVisible(true) rend la fenêtre visible.

Remarque : Au lieu de déclarer un attribut de type JFrame, on aurait pu hériter de la classe JFrame. Ceci est souvent utilisé... mais a souvent peu d'intérêt !

Composants de premier niveau

Définition : Ce sont les composants qui sont pris en compte par le gestionnaire de fenêtres de la plateforme.

- `JFrame` : fenêtre principale d'une application Swing composée :
  - d'un contenu (Container) accessible par `get/setContentPane`
  - d'une barre de menus (`setJMenuBar`)
- `JDialog` : fenêtre de dialogue (confirmation, choix, etc.) ;
- `JApplet` : une applet qui utilise les composants Swing.

Les composants de premier niveau utilisent des ressources de la plate-forme d'exécution qu'il faut libérer explicitement (avec `dispose`) quand la fenêtre n'est plus utilisée.

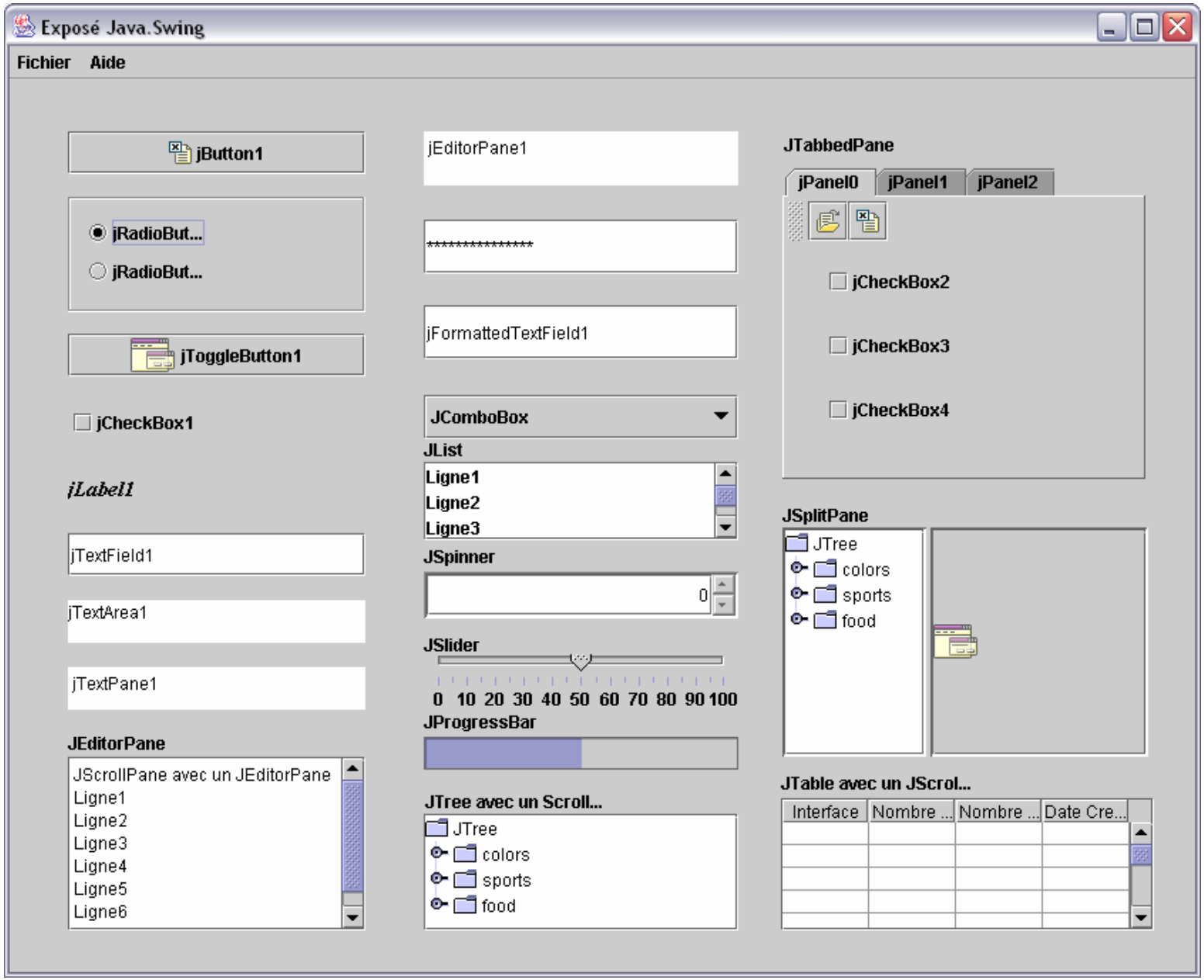
Composants élémentaires

Définition : Ce sont les composants (Component) de base pour construire une interface (*widgets* sous Unix et *contrôles* sous Windows).

- JLabel : un élément qui peut contenir du texte et/ou une image ;
- JButton : comme un JLabel mais a vocation à être appuyé ;
- JTextField : zone de texte éditable ;
- JProgressBar : barre de progression ;
- JSlider : choix d'une valeur numérique par curseur ;
- JColorChooser : choix d'une couleur dans une palette ;
- JTextComponent : manipulation de texte (éditeur) ;
- JTable : afficher une table à deux dimensions de cellules ;
- JTree : affichage de données sous forme d'arbre ;
- JMenus : menus
- ...

Visualisation des composants élémentaires et containers

http://jfod.cnam.fr/NFP121/supports/NFP121_cours_4_2_swing_MVC.pdf



Conteneurs de composants

Principe : Un conteneur (Container ou JContainer) permet d'organiser et de gérer plusieurs composants.

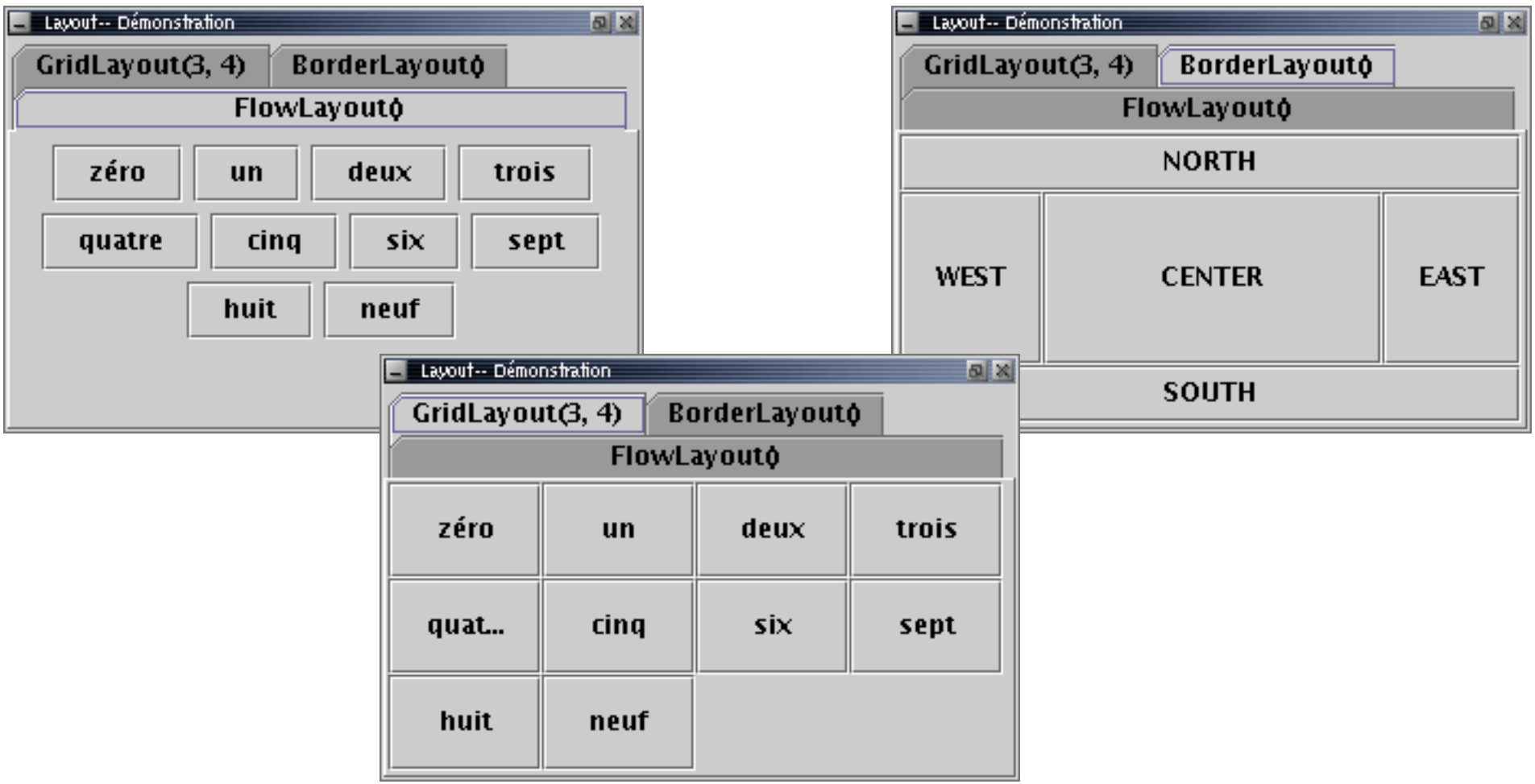
- Les conteneurs généraux :
  - JPanel : le plus flexible
  - JSplitPane : deux composants
  - JScrollPane : avec ascenseur
  - JTabbedPane : intercalaires avec onglets
  - JToolBar : ligne ou colonne de composants
- Les conteneurs spécialisés :
  - JInternalFrame : même propriétés qu'une JFrame mais... interne
  - JLayeredPane : recouvrement possible (profondeur)

Remarque : Un Container étant un Component, on peut imbriquer les conteneurs.

Partie 2 : Construction de la présentation (vue)

Gestionnaires de placement

Principe : Associé à un Container, un gestionnaire de placement (LayoutManager) est chargé de positionnement ses composants suivant une politique prédéfinie.



Principaux gestionnaires de placement

Suivant le gestionnaire de placement, les composants sont placés :

FlowLayout	les uns à la suite des autres
BorderLayout	au centre ou à un des 4 points cardinaux ;
BoxLayout	sur une seule ligne ou une seule colonne ;
GridLayout	dans un tableau à deux dimensions (toutes les cases ont même taille) ;
GridBagLayout	dans une grille mais les composants peuvent être de tailles différentes en occupant plusieurs lignes et/ou colonnes ;
CardLayout	seulement un seul composant est affiché ;

Exemple du compteur

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  public class CompteurGUIsimple {
5      public CompteurGUIsimple(Compteur compteur) {
6          JFrame fenetre = new JFrame("Compteur");
7          Container contenu = fenetre.getContentPane();
8          contenu.setLayout(new BorderLayout());
9
10         JLabel afficheur = new JLabel();          // l'afficheur
11         afficheur.setHorizontalAlignment(JLabel.CENTER);
12         afficheur.setText("" + compteur.getValeur());
13         contenu.add(afficheur, BorderLayout.CENTER);
14
15         // Définir les boutons au SUD
16         JPanel boutons = new JPanel(new FlowLayout());
17         contenu.add(boutons, BorderLayout.SOUTH);
18         JButton bINC = new JButton("INC");        // bouton Incrémenter
19         boutons.add(bINC);
20         JButton bRAZ = new JButton("RAZ");        // bouton RAZ
21         boutons.add(bRAZ);
22         JButton bQuitter = new JButton("QUITTER");  // bouton Quitter
23         boutons.add(bQuitter);
24
25         fenetre.pack();          // dimmensionner la fenêtre
26         fenetre.setVisible(true);   // la rendre visible
27     }
28 }
```

Code pour l'exemple du gestionnaire de placement (1/2)

```
1       private final static String[] chiffres = { "zéro", "un", "deux", "trois",
2                       "quatre", "cinq", "six", "sept", "huit", "neuf" };
3
4       /** Construire un JPanel avec les boutons des chiffres et le layout. */
5       private static JPanel creerPanel(LayoutManager layout) {
6               JPanel result = new JPanel(layout);
7               for (int i = 0; i < chiffres.length; i++) {
8                       result.add(new JButton(chiffres[i]));
9               }
10              return result;
11      }
12
13      /** Construire un JPanel avec un BorderLayout. */
14      private static JPanel creerPanelBorderLayout() {
15              JPanel result = new JPanel(new BorderLayout());
16              result.add(new JButton("CENTER"), BorderLayout.CENTER);
17              result.add(new JButton("EAST"), BorderLayout.EAST);
18              result.add(new JButton("WEST"), BorderLayout.WEST);
19              result.add(new JButton("SOUTH"), BorderLayout.SOUTH);
20              result.add(new JButton("NORTH"), BorderLayout.NORTH);
21              return result;
22      }
```

Code pour l'exemple du gestionnaire de placement (2/2)

```
1       public static void main(String[] args) {
2               JFrame fenetre = new JFrame("Layout - Démonstration");
3               Container contenu = fenetre.getContentPane();
4
5               JTabbedPane onglets = new JTabbedPane();
6               onglets.addTab("FlowLayout()", creerPanel(new FlowLayout()));
7               onglets.addTab("GridLayout(3, 4)", creerPanel(new GridLayout(3, 4)));
8               onglets.addTab("BorderLayout()", creerPanelBorderLayout());
9
10              contenu.add(onglets);
11
12              fenetre.setSize(new Dimension(300, 200));
13              fenetre.setVisible(true);
14
15              // Définition du contrôleur
16              fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17      }
```

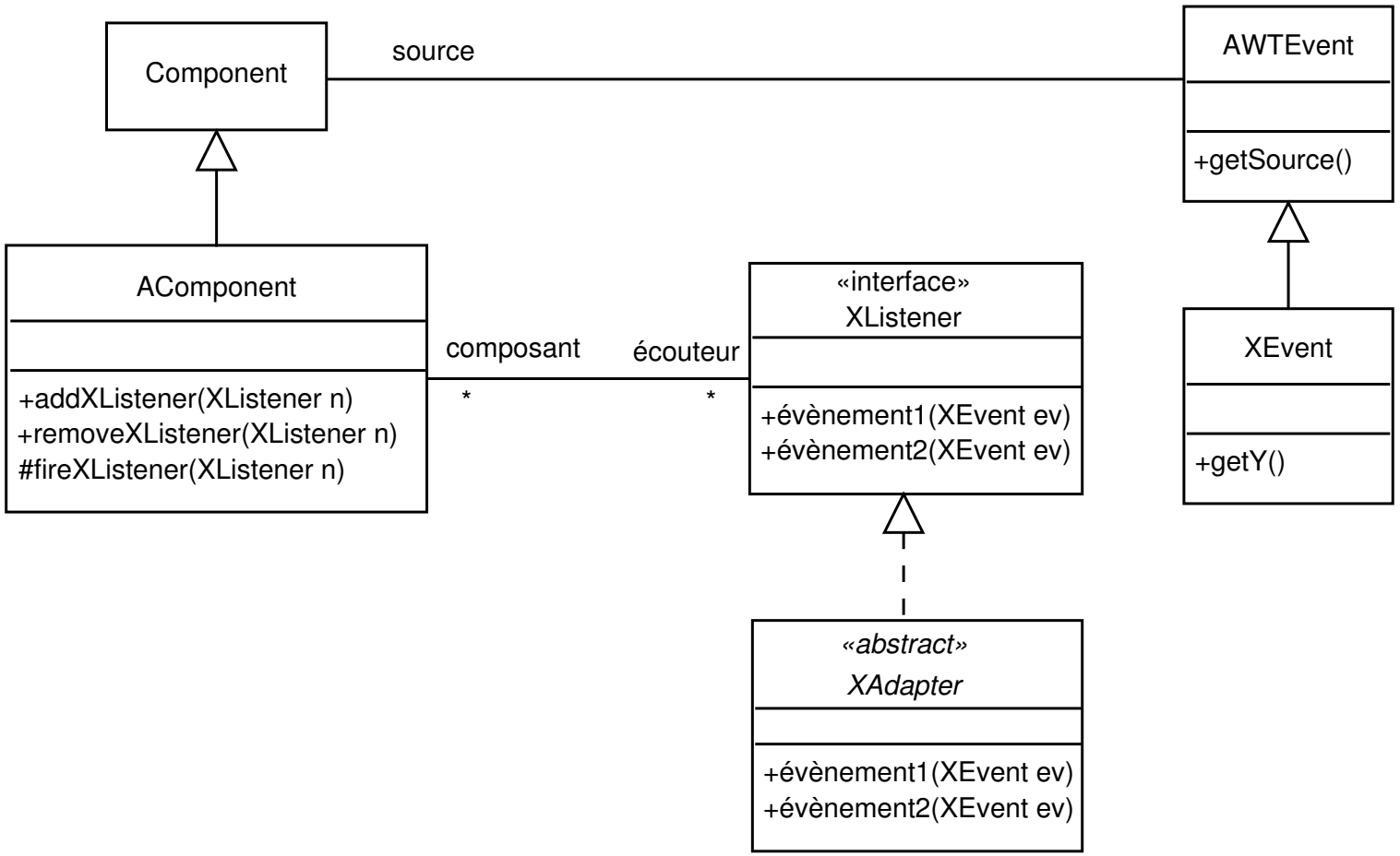
Partie 3 : La gestion des événements

Le modèle événementiel

Principe : Toute partie de l'application peut réagir aux événements produits par une autre partie de l'application.
Vocabulaire : Un *événement* est une information produite par un composant appelé *émetteur* et qui pourra déclencher des réactions sur d'autres éléments appelés *récepteurs*.
Exemples : Appui sur un bouton, déplacement d'une souris, appui sur une touche, expiration d'une temporisation, etc.
En Java :

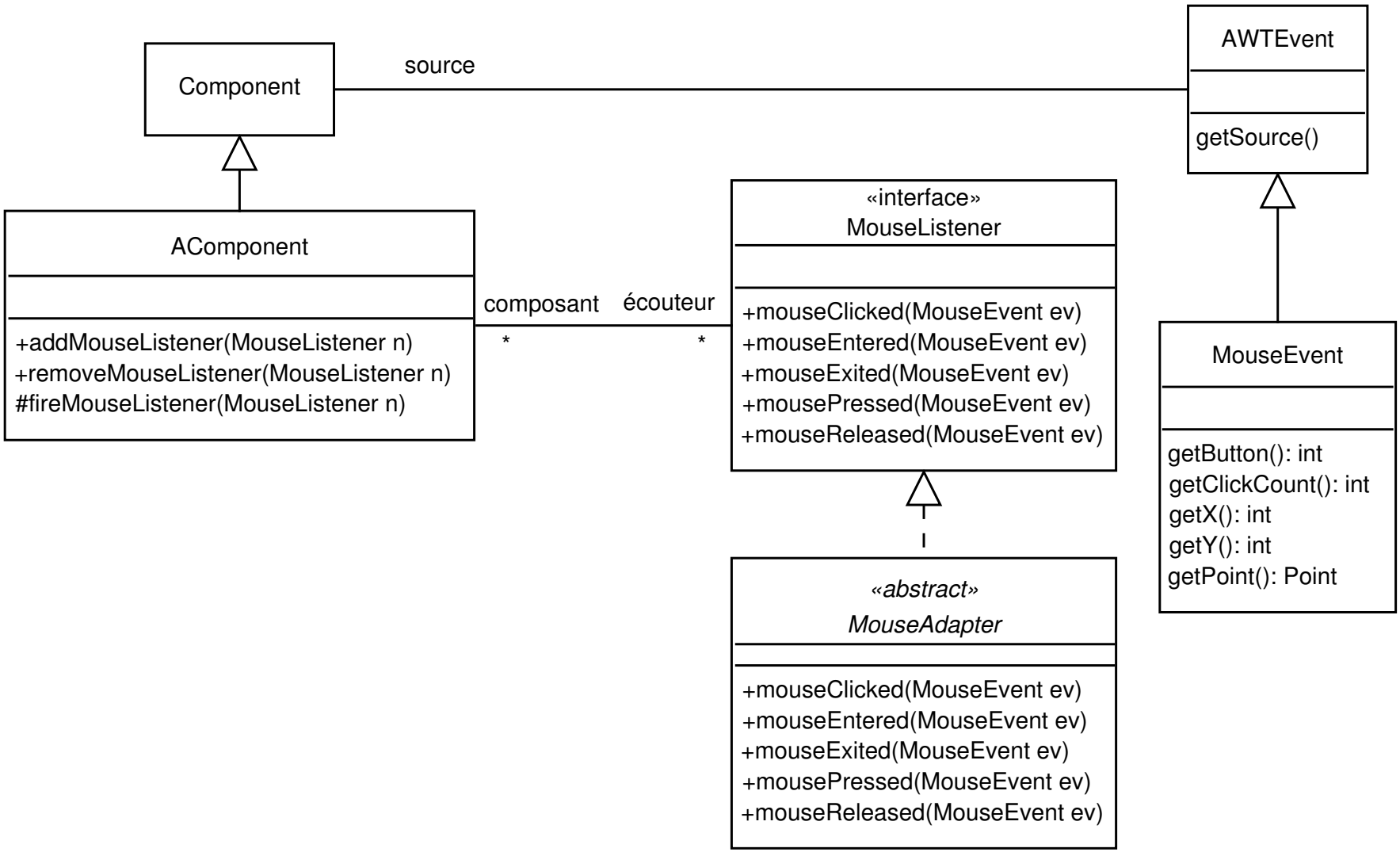
- La programmation événementielle n'existe pas !
- Elle est simulée par le patron de conception Observateur (Listener) :
  - le récepteur doit définir un gestionnaire d'événements (XListener) ;
  - le récepteur doit l'inscrire (addXListener) auprès d'un émetteur ;
  - le composant avertit le gestionnaire d'événements quand un événement se produit (fireXListener) ;
  - le récepteur peut désinscrire son gestionnaire d'événements ;
  - un gestionnaire d'événements est spécifique à type d'événement X.

Architecture des Listeners pour un type d'événement



Remarque : La classe XAdapter est une réalisation de l'interface XListener avec des méthodes dont le code est vide (ne fait rien).

Exemple : MouseListener



Partie 3 : La gestion des événements

Exercice fil rouge

Exercice 3 Écrire une application Java/Swing qui dit bonjour à l'utilisateur.

3.1 L'application possède un bouton « Coucou » qui écrit « Bonjour ! » dans le terminal si on appuie dessus.

3.2 Ajouter une zone de saisie pour permettre à l'utilisateur de donner son nom. L'appui sur le bouton « Coucou » affichera alors « Bonjour » suivi du nom de l'utilisateur (s'il a été renseigné).

3.3 Ajouter un bouton « Quitter » pour permettre à l'utilisateur de quitter l'application.

Exercice fil rouge

Exercice 3 Écrire une application Java/Swing qui dit bonjour à l'utilisateur.

3.1 L'application possède un bouton « Coucou » qui écrit « Bonjour ! » dans le terminal si on appuie dessus.

3.2 Ajouter une zone de saisie pour permettre à l'utilisateur de donner son nom. L'appui sur le bouton « Coucou » affichera alors « Bonjour » suivi du nom de l'utilisateur (s'il a été renseigné).

3.3 Ajouter un bouton « Quitter » pour permettre à l'utilisateur de quitter l'application.

La classe Vue/Contrôleur réalise ActionListener

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  class CoucouGUI implements ActionListener {
6
7      public CoucouGUI() {
8          JFrame fenetre = new JFrame("Je␣débute␣!");
9          JButton coucou = new JButton("Coucou");
10         fenetre.getContentPane().add(coucou);
11         coucou.addActionListener(this);
12         fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         fenetre.pack();     // Calculer de la taille de la fenêtre
14         fenetre.setVisible(true);   // Rendre la fenêtre visible
15     }
16
17     public void actionPerformed(ActionEvent coucou) {
18         System.out.println("Bonjour␣!");
19     }
20
21     public static void main(String[] args) {
22         new CoucouGUI();
23     }
24 }
```

- Cette relation de réalisation est-elle logique ?

Définition d'un ActionListener spécifique

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  class CoucouGUI {
6
7      public CoucouGUI() {
8          JFrame fenetre = new JFrame("Je␣débute␣!");
9          JButton coucou = new JButton("Coucou");
10         fenetre.getContentPane().add(coucou);
11         coucou.addActionListener(new ActionCoucou());
12         fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         fenetre.pack();     // Calculer de la taille de la fenêtre
14         fenetre.setVisible(true);   // Rendre la fenêtre visible
15     }
16
17     public static void main(String[] args) {
18         new CoucouGUI();
19     }
20 }
21
22 class ActionCoucou implements ActionListener {
23     public void actionPerformed(ActionEvent coucou) {
24         System.out.println("Bonjour␣!");
25     }
26 }
```

Exercice fil rouge

Exercice 3 Écrire une application Java/Swing qui dit bonjour à l'utilisateur.

3.1 L'application possède un bouton « Coucou » qui écrit « Bonjour ! » dans le terminal si on appuie dessus.

3.2 Ajouter une zone de saisie pour permettre à l'utilisateur de donner son nom. L'appui sur le bouton « Coucou » affichera alors « Bonjour » suivi du nom de l'utilisateur (s'il a été renseigné).

3.3 Ajouter un bouton « Quitter » pour permettre à l'utilisateur de quitter l'application.

La Vue réalise ActionListener

```
1  class CoucouGUI implements ActionListener {
2      private JTextField nom; // le nom devient un attribut *privé* !
3      public CoucouGUI() {
4          JFrame fenetre = new JFrame("Je␣débute␣!");
5          Container contenu = fenetre.getContentPane();
6          contenu.setLayout(new FlowLayout());
7          JLabel texteNom = new JLabel("Nom␣:");  contenu.add(texteNom);
8          nom = new JTextField(20);           contenu.add(nom);
9          JButton coucou = new JButton("Coucou"); contenu.add(coucou);
10         coucou.addActionListener(this);
11         ...
12     }
13
14     public void actionPerformed(ActionEvent coucou) {
15         System.out.print("Bonjour");
16         if (nom.getText() != null && nom.getText().length() > 0)  {
17             System.out.print("␣" + nom.getText());
18         }
19         System.out.println("␣!");
20     }
21 }
```

- pour accéder à la zone de saisie (JTextField), ce doit être un attribut
- il peut être privé !

Listener comme classe externe

```
1  class CoucouGUI {
2      JTextField nom; // le nom devient un attribut mais PAS *privé* !
3      public CoucouGUI() {
4          JFrame fenetre = new JFrame("Je␣débute␣!");
5          Container contenu = fenetre.getContentPane();
6          contenu.setLayout(new FlowLayout());
7          JLabel texteNom = new JLabel("Nom␣:");  contenu.add(texteNom);
8          nom = new JTextField(20);            contenu.add(nom);
9          JButton coucou = new JButton("Coucou"); contenu.add(coucou);
10         coucou.addActionListener(new ActionCoucou(this));
11         ...
12     }
13 }
14
15 class ActionCoucou implements ActionListener {
16     private CoucouGUI vue;
17     public ActionCoucou(CoucouGUI c)    { vue = c; }
18     public void actionPerformed(ActionEvent coucou) {
19         System.out.print("Bonjour");
20         if (vue.nom.getText() != null && vue.nom.getText().length() > 0)  {
21             System.out.print("␣" + vue.nom.getText());
22         }
23         System.out.println("␣!");
24 }   }
```

- l'attribut ne peut pas être privé car il est utilisé par la classe externe ActionCoucou
- la transmission de la vue (ou du JTextField) est lourde

Listener comme classe interne

```
1  class CoucouGUI {
2      private JTextField nom; // le nom devient un attribut *PRIVÉ* !
3      public CoucouGUI() {
4          JFrame fenetre = new JFrame("Je␣débute␣!");
5          Container contenu = fenetre.getContentPane();
6          contenu.setLayout(new FlowLayout());
7          JLabel texteNom = new JLabel("Nom␣:");  contenu.add(texteNom);
8          nom = new JTextField(20);              contenu.add(nom);
9          JButton coucou = new JButton("Coucou"); contenu.add(coucou);
10         coucou.addActionListener(new ActionCoucou(this));
11         ...
12     }
13     static class ActionCoucou implements ActionListener {
14         private CoucouGUI vue;
15         public ActionCoucou(CoucouGUI c)    { vue = c; }
16             public void actionPerformed(ActionEvent coucou) {
17                 System.out.print("Bonjour");
18                 if (vue.nom.getText() != null && vue.nom.getText().length() > 0)  {
19                     System.out.print("␣" + vue.nom.getText());
20                 }
21                 System.out.println("␣!");
22             }
23     }
24 }
```

- l'attribut peut être déclaré privé (le listener est une classe interne) !
- il faut quand même transmettre la vue ! $\Longrightarrow$ supprimer le **static**

Listener comme classe interne membre

```
1   class CoucouGUI {
2       private JTextField nom; // le nom devient un attribut *PRIVÉ* !
3       public CoucouGUI() {
4           JFrame fenetre = new JFrame("Je␣débute␣!");
5           Container contenu = fenetre.getContentPane();
6           contenu.setLayout(new FlowLayout());
7           JLabel texteNom = new JLabel("Nom␣:");  contenu.add(texteNom);
8           nom = new JTextField(20);           contenu.add(nom);
9           JButton coucou = new JButton("Coucou"); contenu.add(coucou);
10          coucou.addActionListener(new ActionCoucou());
11          ...
12      }
13
14      class ActionCoucou implements ActionListener {
15          public void actionPerformed(ActionEvent coucou) {
16              System.out.print("Bonjour");
17              if (nom.getText() != null && nom.getText().length() > 0)  {
18                  System.out.print("␣" + nom.getText());
19              }
20              System.out.println("␣!");
21  }   }   }
```

- l'attribut est privé !
- on ne transmet pas la vue : l'objet ActionCoucou a accès à l'instance de CoucouGUI qui l'a créé

Listener comme classe anonyme

```
1  class CoucouGUI {
2      public CoucouGUI() {
3          JFrame fenetre = new JFrame("Je␣débute␣!");
4          Container contenu = fenetre.getContentPane();
5          contenu.setLayout(new FlowLayout());
6          JLabel texteNom = new JLabel("Nom␣:");          contenu.add(texteNom);
7          final JTextField nom = new JTextField(20);      contenu.add(nom);
8          JButton coucou = new JButton("Coucou");         contenu.add(coucou);
9          coucou.addActionListener(new ActionListener() {
10             public void actionPerformed(ActionEvent coucou) {
11                 System.out.print("Bonjour");
12                 if (nom.getText() != null && nom.getText().length() > 0)  {
13                     System.out.print("␣" + nom.getText());
14                 }
15                 System.out.println("␣!");
16             }
17         });
18         ...
19     }
20 }
```

- le JTextField peut rester une variable locale du constructeur
- il doit cependant être déclaré **final**
- Pas forcément très lisible !

Exercice fil rouge

Exercice 3 Écrire une application Java/Swing qui dit bonjour à l'utilisateur.

3.1 L'application possède un bouton « Coucou » qui écrit « Bonjour ! » dans le terminal si on appuie dessus.

3.2 Ajouter une zone de saisie pour permettre à l'utilisateur de donner son nom. L'appui sur le bouton « Coucou » affichera alors « Bonjour » suivi du nom de l'utilisateur (s'il a été renseigné).

3.3 Ajouter un bouton « Quitter » pour permettre à l'utilisateur de quitter l'application.

Autant de Listener que de réactions

```
1  class CoucouGUI {
2      public CoucouGUI() {
3          JFrame fenetre = new JFrame("Avec deux ActionListeners");
4          fenetre.getContentPane().setLayout(new FlowLayout());
5          JButton coucou = new JButton("Coucou");
6          fenetre.getContentPane().add(coucou);
7          coucou.addActionListener(new ActionCoucou());
8          JButton quitter = new JButton("Quitter");
9          fenetre.getContentPane().add(quitter);
10         quitter.addActionListener(new ActionQuitter());
11         ...
12     }
13 }
14
15 class ActionCoucou implements ActionListener {
16     public void actionPerformed(ActionEvent coucou) {
17         System.out.println("Bonjour !");
18 }   }
19
20 class ActionQuitter implements ActionListener {
21     public void actionPerformed(ActionEvent ev) {
22         System.exit(1);
23 }   }
```

- Chaque Listener ne fait qu'une chose !
- Chaque Listener est inscrit auprès du bon bouton (pas de conditionnelle)
- Mais beaucoup de Listener

Un seul Listener pour toutes les réactions

```
1  class CoucouGUI implements ActionListener {
2      private JButton coucou = new JButton("Coucou");
3      private JButton quitter = new JButton("Quitter");
4      public CoucouGUI() {
5          JFrame fenetre = new JFrame("Réalise␣ListenerAction");
6          fenetre.getContentPane().setLayout(new FlowLayout());
7          fenetre.getContentPane().add(coucou);
8          coucou.addActionListener(this);
9          fenetre.getContentPane().add(quitter);
10         quitter.addActionListener(this);
11         ...
12     }
13     public void actionPerformed(ActionEvent ev) {
14         if (ev.getSource() == coucou) {
15             System.out.println("Bonjour␣!");
16         } else if (ev.getSource() == quitter) {
17             System.exit(1);
18         }
19     }
20 }
```

- un seul Listener (ce pourrait être une classe spécifique)
- et donc des tests pour retrouver le bouton qui a été cliqué
- donc couplage fort : que faire si des entrées de menu déclenchent les mêmes actions ?

Un seul Listener mais plus indépendant de la source

```
1   class CoucouGUI implements ActionListener {
2       public CoucouGUI() {
3           JFrame fenetre = new JFrame("Avec␣ActionCommand");
4           fenetre.getContentPane().setLayout(new FlowLayout());
5           JButton coucou = new JButton("Coucou");
6           coucou.setActionCommand("COUCOU");
7           fenetre.getContentPane().add(coucou);
8           coucou.addActionListener(new ActionCoucou());
9           JButton quitter = new JButton("Quitter");
10          quitter.setActionCommand("QUITTER");
11          fenetre.getContentPane().add(quitter);
12          quitter.addActionListener(new ActionQuitter());
13          ...
14      }
15
16      public void actionPerformed(ActionEvent ev) {
17          AbstractButton bouton = (AbstractButton) ev.getSource();
18          if (bouton.getActionCommand().equals("COUCOU")) {
19              System.out.println("Bonjour␣!");
20          } else if (bouton.getActionCommand().equals("QUITTER")) {
21              System.exit(1);
22  }   }   }
```

- Principe : associer une information (String) a un bouton (AbstractButton)
- $\Longrightarrow$ plus abstrait et donc indépendant des composants de la vue
- L'AbstractButton pourrait être une entrée de menu

Qui réalise le XListener ?

La classe « principale » (vue) réalise (spécialise) les XListener (contrôleur)

- − la relation de sous-typage n'est pas réellement logique
- + accès aux informations de la vue par le contrôleur
- − impossible d'écrire plusieurs écouteurs différents et du même type

Définition d'une classe spécifique pour chaque écouteur

- + Bonne séparation des éléments... mais grand nombre de classes ;
- Comment l'écouteur a accès aux informations de la vue et du modèle ?
  - − rendre accessible les éléments de la vue !
    $\Longrightarrow$ Violation (partielle) du principe d'encapsulation !
  - + utiliser les classes internes (préserve l'encapsulation).
  - + définir des opérations de haut niveau sur le modèle ;

Stratégies pour définir les gestionnaires d'événements

Problème : Un gestionnaire d'événements doit savoir quel traitement réaliser.
Exemple : Suivant le bouton appuyé, il faut dire « Bonjour » ou arrêter l'application.

Solution 1 : Un seul gestionnaire d'événements inscrit auprès de tous les composants :

- dans le gestionnaire, utiliser ev.getSource() pour savoir quel traitement faire
  - **Problème :** Forte dépendance entre Vue et Contrôleur.
- utiliser actionCommand (sur les boutons) pour diminuer le couplage.
  - **Avantage :** Indépendance / vue (boutons ou entrées de menus)
  - **Attention :** Série de tests peu logique dans une approche objet.

Solution 2 : Définir des gestionnaires d'événements spécifiques :

- chaque gestionnaire réalise une seule action.
- **Remarque :** C'est le principe utilisé pour les menus textuels.

Faire communiquer la vue/le modèle et l'écouteur

Problème : Un gestionnaire d'événements peut nécessiter plus d'information que la source pour s'exécuter.
Exemple : Le nom de l'utilisateur pour le saluer.

- Le gestionnaire d'événements est défini dans l'espace de nom des classes possédant les informations (vue, modèle) :
  - la classe réalise le XListener ;
  - le gestionnaire d'événement est une classe interne (locale) ;
  - le gestionnaire d'événement est une classe anonyme ;

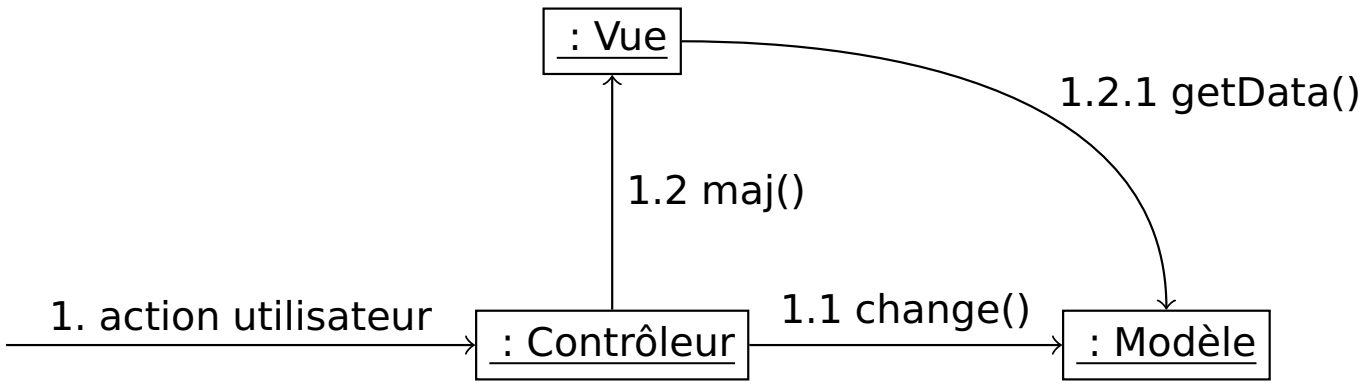
  \+ accès aux informations de la vue/contrôleur
  – risque d'avoir une application monolithique (couplage fort)
- Transmission des informations au gestionnaire d'événements lors de sa création (ou après sa création).
- Spécialisation du composant pour y attacher les informations supplémentaires. L'écouteur devra alors transtyper la source pour accéder à ces informations.
- Certainement d'autres possibilités...

Exemple : Voir le jeu du Morpion.

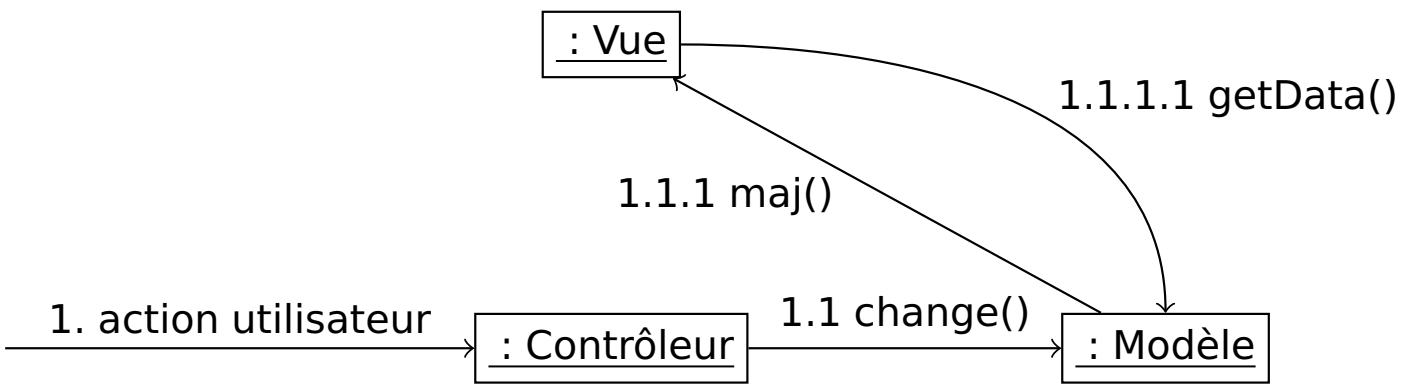
Partie 4 : Retour sur le MVC

Modèle passif vs modèle actif

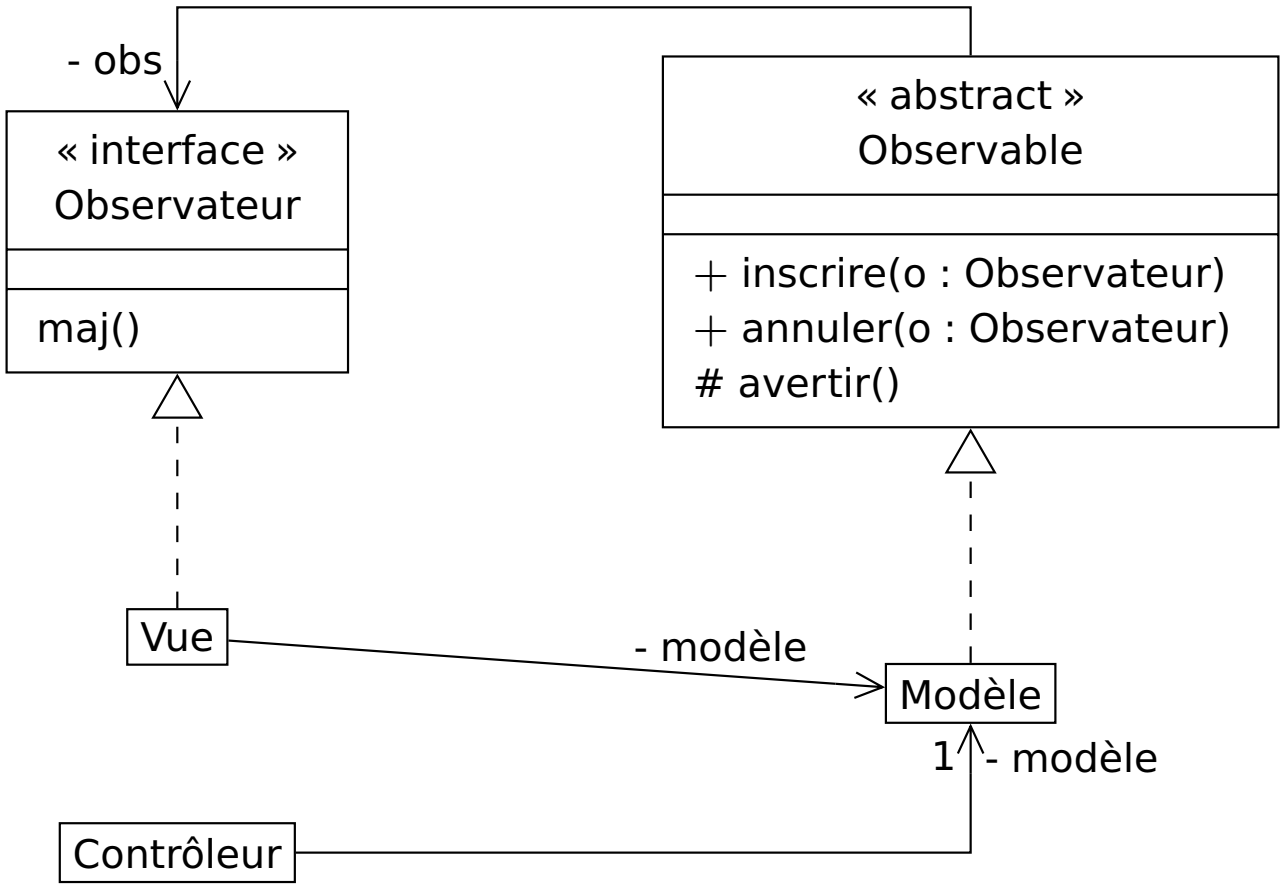
Modèle passif : le contrôleur connaît modèle et vues



Modèle actif : le contrôleur ne connaît que le modèle



Le modèle doit être indépendant des vues : patron Observateur !



Partie 4 : Retour sur le MVC

L'application souhaitée

- Apparence et comportement souhaités :
  - 0 : remet le compteur à 0
  - ++ : incrémente la valeur du compteur
  - set : change la valeur pour celle de la zone de saisie

1. Quel modèle ?
2. Quelle vue ?
3. Quel contrôleur ?
4. Et l'application complète ?

Partie 4 : Retour sur le MVC

Le modèle passif

```java
public class ModeleCompteur {
    private int valeur;     // valeur du compteur

    /** Initialiser la valeur du compteur.
     * @param v la valeur initiale du compteur
     */
    public ModeleCompteur(int v)    {
        this.valeur = v;
    }

    /** Augmenter d'une unité le compteur */
    public void incrementer() {
        this.valeur++;
    }

    /** Obtenir la valeur du compteur.
     * @return la valeur du compteur.
     */
    public int getValeur() {
        return this.valeur;
    }

    /** Changer la valeur du compteur.
     * @param nv nouvelle valeur
     * @return la valeur du compteur.
     */
    public void setValeur(int nv) {
        this.valeur = nv;
    }

    /** Remettre à zéro le compteur */
    public void raz() {
        this.valeur = 0;
    }
}
```

La vue (modèle passif)

```
import javax.swing.*;

public class VueCompteur extends JLabel {
    private ModeleCompteur modele;

    public VueCompteur(ModeleCompteur modele) {
        this.modele = modele;
        this.setFont(this.getFont().deriveFont(48f));
        this.setHorizontalAlignment(SwingConstants.CENTER);
        mettreAJour();
    }

    public void mettreAJour() {
        this.setText(modele.getValeur() + "");
    }
}
```

Le contrôleur (modèle passif)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ControleurCompteur extends JPanel {

 public ControleurCompteur(final ModeleCompteur modele,
   final VueCompteur vue)
 {
  super(new FlowLayout());
  // Définition de la vue du contrôleur
  final JTextField zoneSaisie = new JTextField(6);
  final JButton bSET = new JButton("set");
  final JButton bRAZ = new JButton("0");
  final JButton bINC = new JButton("++");
  this.add(zoneSaisie);
  this.add(bSET);
  this.add(bRAZ);
  this.add(bINC);

  // Définition des contrôleurs du contrôleur
  bRAZ.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    modele.raz();   // appeler opération du modèle
    vue.mettreAJour();
   }
  });

  bINC.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    modele.incrementer();
    vue.mettreAJour();
   }
  });

  bSET.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    try {
     String newValue = zoneSaisie.getText();
     int intValue = Integer.parseInt(newValue);
     modele.setValeur(intValue);
     vue.mettreAJour();
    } catch (NumberFormatException exception) {
     zoneSaisie.setText("erreur");
    }
   }
  });
 }
}
```

L'application complète (modèle passif)

```
import javax.swing.*;
import java.awt.*;

public class IHMCompteur extends JFrame {

    public IHMCompteur() {
        ModeleCompteur modele = new ModeleCompteur(0);
        VueCompteur vue = new VueCompteur(modele);
        ControleurCompteur controleur = new ControleurCompteur(modele, vue);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(vue, BorderLayout.CENTER);
        this.getContentPane().add(controleur, BorderLayout.SOUTH);
        this.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new IHMCompteur();
    }
}
```

Partie 4 : Retour sur le MVC

Le modèle actif

```
public class ModeleCompteur
    extends java.util.Observable
{
  private int valeur;   // valeur du compteur

  /** Initialiser la valeur du compteur.
   * @param v la valeur initiale du compteur
   */
  public ModeleCompteur(int v)  {
    this.valeur = v;
  }

  /** Augmenter d'une unité le compteur */
  public void incrementer() {
    this.valeur++;
    this.avertir();
  }

  /** Obtenir la valeur du compteur.
   * @return la valeur du compteur.
   */
  public int getValeur() {
    return this.valeur;
  }

  /** Changer la valeur du compteur.
   * @param nv nouvelle valeur
   * @return la valeur du compteur.
   */
  public void setValeur(int nv) {
    this.valeur = nv;
    this.avertir();
  }

  /** Remettre à zéro le compteur */
  public void raz() {
    this.valeur = 0;
    this.avertir();
  }

  /** Avertir tous les observateurs inscrits
  private void avertir() {
    this.setChanged();
    this.notifyObservers();
  }
}
```

La vue (modèle actif)

```
import javax.swing.*;

public class VueCompteur extends JLabel {
    private ModeleCompteur modele;

    public VueCompteur(ModeleCompteur modele) {
        this.modele = modele;
        this.setFont(this.getFont().deriveFont(48f));
        this.setHorizontalAlignment(SwingConstants.CENTER);
        mettreAJour();
        modele.addObserver(new java.util.Observer() {
            public void update(java.util.Observable o, Object arg) {
                mettreAJour();
            }
        });
    }

    private void mettreAJour() {
        this.setText(modele.getValeur() + "");
    }
}
```

Le contrôleur (modèle actif)

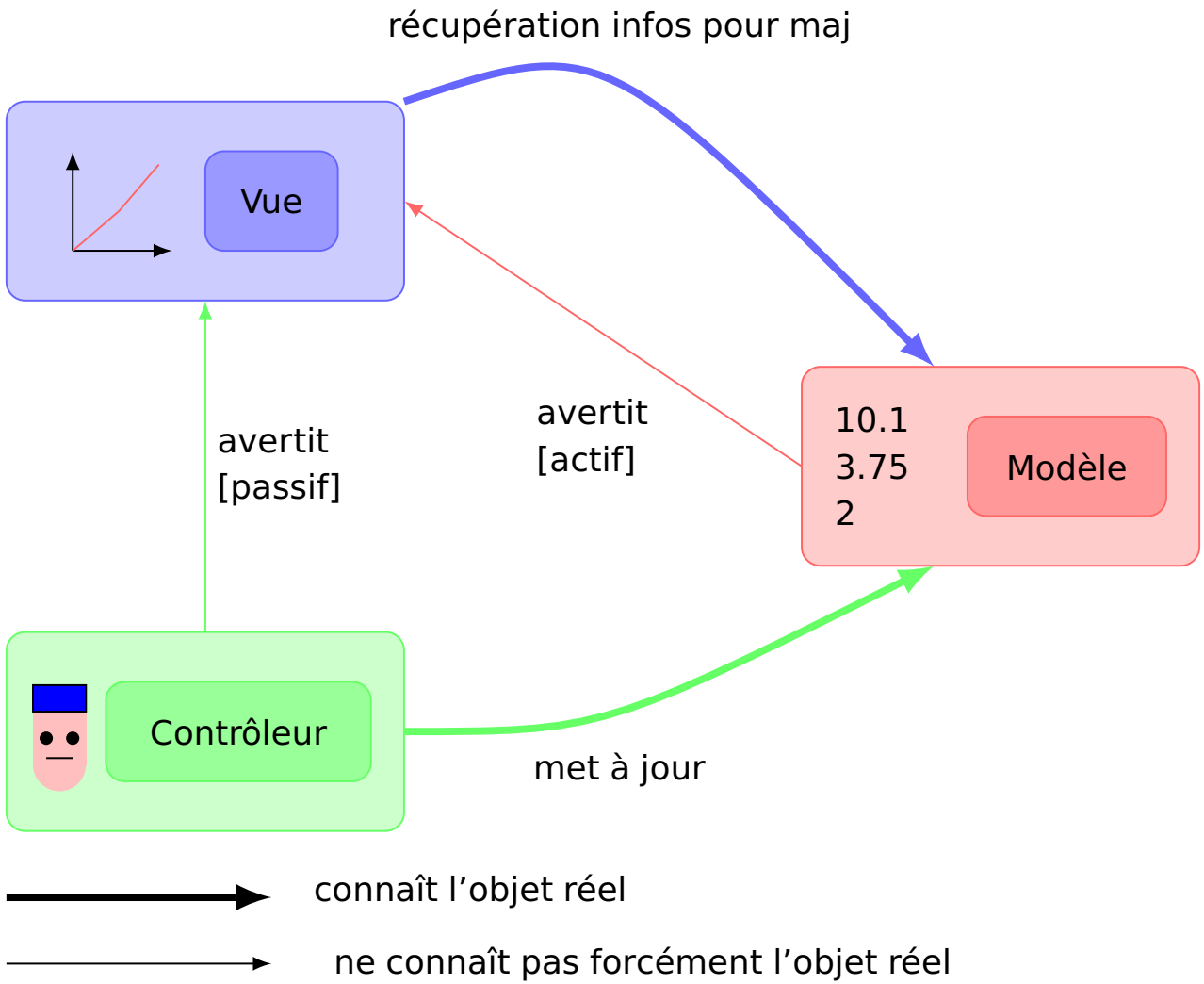
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ControleurCompteur extends JPanel {
 public ControleurCompteur(final ModeleCompteur modele) {
  super(new FlowLayout());
  // Définition de la vue du contrôleur
  final JTextField zoneSaisie = new JTextField(6);
  final JButton bSET = new JButton("set");
  final JButton bRAZ = new JButton("0");
  final JButton bINC = new JButton("++");
  this.add(zoneSaisie);
  this.add(bSET);
  this.add(bRAZ);
  this.add(bINC);

  // Définition des contrôleurs du contrôleur
  bRAZ.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    modele.raz();   // appeler opération du modèle
   }
  });

  bINC.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    modele.incrementer();
  } });

  bSET.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    try {
     String newValue = zoneSaisie.getText();
     int intValue = Integer.parseInt(newValue);
     modele.setValeur(intValue);
    } catch (NumberFormatException exception) {
     zoneSaisie.setText("erreur");
    }
   }
  });
 }
}
```

L'application complète (modèle actif)

```
import javax.swing.*;
import java.awt.*;

public class IHMCompteur extends JFrame {

    public IHMCompteur() {
        ModeleCompteur modele = new ModeleCompteur(0);
        VueCompteur vue = new VueCompteur(modele);
        ControleurCompteur controleur = new ControleurCompteur(modele);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(vue, BorderLayout.CENTER);
        this.getContentPane().add(controleur, BorderLayout.SOUTH);
        this.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new IHMCompteur();
    }
}
```

Partie 4 : Retour sur le MVC

MVC : résumé


récupération infos pour maj
Vue
avertit
[passif]
avertit
[actif]
10.1
3.75
2
Modèle
Contrôleur
met à jour
connaît l'objet réel
ne connaît pas forcément l'objet réel

Partie 5 : Conclusion

Attention au thread de répartition des événements !

- Java est un langage concurrent (plusieurs threads d'exécution).
- Un thread particulier EventQueue s'occupe de la répartition des événements reçus par les composants graphiques vers les écouteurs associés.
- Pour éviter les conflits (et les erreurs difficiles à reproduire et localiser), les éléments Swing doivent toujours être manipulés depuis le EventQueue :

```
1  EventQueue.invokeLater(new Runnable() {
2      public void run() {
3          instructions
4      }
5  });
```

- Exemple :

```
1  public static void main() {
2      EventQueue.invokeLater(new Runnable() {
3          public void run() {
4              new CoucouGUI();
5          }
6      });
7  }
```

Partie 5 : Conclusion

Ce qu'il faut retenir

Pour définir une application adaptable et réutilisable, il faut :

- définir la logique de l'application (diagrammes de machine à états) ;
- bien séparer le modèle, la vue et le contrôleur (patron MVC) ;
- construire le modèle ;
- construire la présentation ;
- programmer les réactions, donc constuire le contrôleur ;

Partie 5 : Conclusion

Le jeu du Morpion

Exercice 4 : Jeu du Morpion
Programmer un jeu du Morpion qui offre une interface graphique en utilisant Java/Swing.