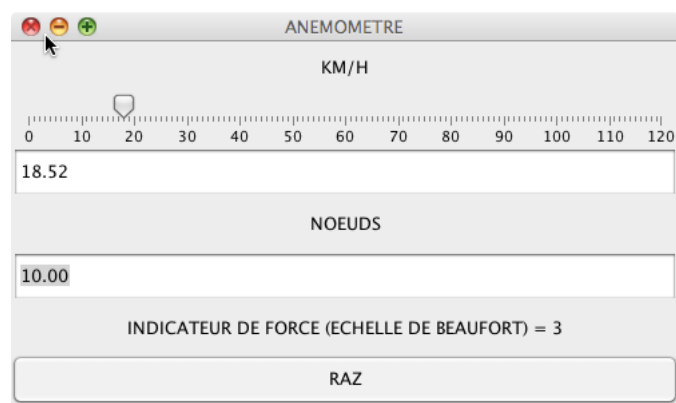


Interfaces graphiques

java.awt (Abstract Window Toolkit)
javax.swing

1

Exemple



2

noyau

Convertisseur

Attributes

protected double vent

Operations

public void setKMH(double v)
 public void setNoeuds(double n)
 public void raz()
 public double getKMH()
 public double getNoeuds()
 public int getForce()

Site Web de la Côte d'Opale

Que faire avec ce vent?

BULLETIN ET PREVISIONS METEO

5 - TENDANCE ULTERIEURE
 PREVISIONS POUR LE VENDREDI 8 MARS
 Vent : variable, force 1 à 3, venant Nord-est à Nord en fin de journée.
 Mer : belle.
 Pas de houle significative.
 Temps : couvert avec pluies faibles.
 Visibilité : 2 à 5 milles.

9 - OBSERVATIONS LE JEUDI 7 MARS A 05:00 UTC

DUNKERQUE : vent S 4 ND
 pression 998 hPa en baisse
 CAP GRIS-NEZ : vent NE 5 ND
 BOULOGNE : vent S 3 ND

KM/H

0 10 20 30 40 50 60 70 80 90 100 110 120

22.22


NOEUDES

12.00


INDICATEUR DE FORCE (ECHELLE DE BEAUFORT) = 4

RAZ


Activités



voile (force 2 minimum)



char à voile (force 2 à 4)



parapente (force 3 à 4)

Terminal — sh — 41x8

```
sh-3.2$ java AnemometreConsole
1: releve en km/h
2: releve en noeuds ?
2
releve?
10.00
km/h= 18.52 noeuds= 10.00 force= 3
sh-3.2$
```

ANEMOMETRE

KM/H

0 10 20 30 40 50 60 70 80 90 100 110 120

18.52

NOEUDES

10.00

INDICATEUR DE FORCE (ECHELLE DE BEAUFORT) = 3

RAZ

3

Démarche de conception

- I. Concevoir le noyau fonctionnel
 - appelé aussi « modèle » de l'application
 - indépendamment de toute interaction

- II. Concevoir l'interface graphique
 1. identifier les composants nécessaires
 2. les positionner (layout)
 3. programmer la dynamique
« programmation par évènements »
 4. activer l'application/l'applet

4

I. Noyau fonctionnel

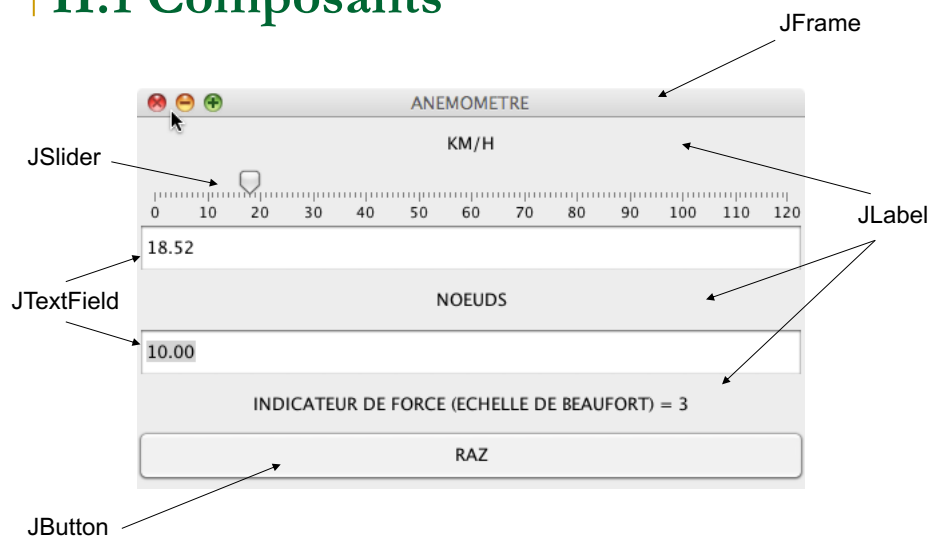
```
public class Convertisseur {
    static double NOEUD2KMH = 1.852;
    // 1 nd (mile marin/h) = 1.852 km/h
    protected double vent;
    // vitesse normalisee en km/h

    // set f(unite)
    public void setKMH(double v) {vent = v;}
    public void setNoeuds(double n) {vent = n * NOEUD2KMH;}
    public void raz() {vent = 0.0;}

    // get f(unite)
    public double getKMH() {return vent;}
    public double getNoeuds() {return vent / NOEUD2KMH;}
    public int getForce() {
        // echelle de Beaufort de 0 a 12
        // f(intervalle de vent)
    }
}
```

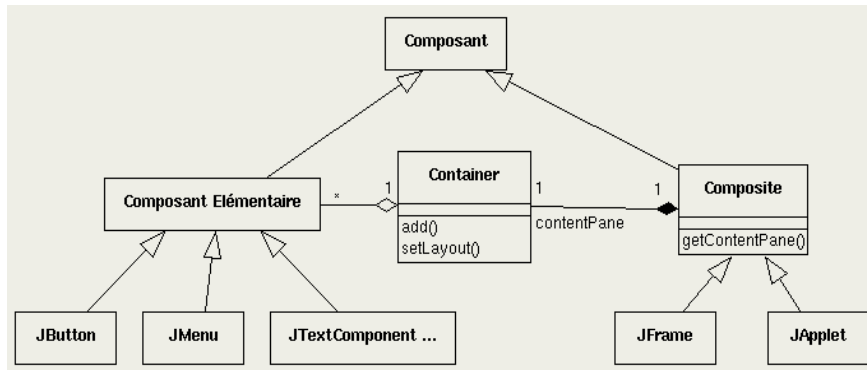
5

II.1 Composants



6

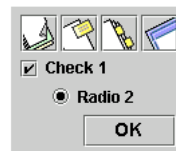
Composants



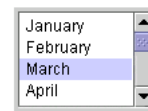
Application du patron de conception (Design Pattern) Composite

Composants élémentaires

```
java.awt.Component
javax.swing.JComponent
  javax.swing.AbstractButton
    javax.swing.JButton
    javax.swing.JCheckBox
    javax.swing.JRadioButton
  javax.swing.JList
  javax.swing.JMenuBar
  javax.swing.JLabel // etiquette
  javax.swing.JTextComponent // texte editable (ou non)
    javax.swing.JTextArea // sur plusieurs lignes
    javax.swing.JTextField // sur une ligne
  javax.swing.JSlider
  ...
```



[Buttons](#)



[List](#)

Composants composites

Contenant d'autres composants

- « fenêtres » principales

```
public interface javax.swing.RootPaneContainer {  
    public Container getContentPane();  
}
```

- application autonome

```
java.awt.Frame
```

```
javax.swing.JFrame
```

- applet

```
java.applet.Applet
```

```
javax.swing.JApplet
```

- et aussi :

- fenêtre de dialogue (JDialog)

- « sous-fenêtres » (JPanel), ...

II.2 Positionnement des composants

- Conteneur : ajout et positionnement des composants

```
class java.awt.Container {  
    // ajouter des composants :  
    void add(Component c);  
    // selon une strategie de positionnement:  
    void setLayout(LayoutManager lay);  
}
```

- **LayoutManager**'s

- par défaut : centre + les 4 orientations

```
java.awt.BorderLayout
```

- glissant de gauche à droite et de haut en bas

```
java.awt.FlowLayout
```

- grille LXC

```
java.awt.GridLayout
```

- grille LXC de cases de taille différente

```
javax.swing.BoxLayout
```

```
java.awt.GridBagLayout
```

- « pile »

```
java.awt.CardLayout
```

Exemple

```
public class AnemometreGraphique extends JFrame {

    // le systeme a interfacier
    Convertisseur convertisseur = new Convertisseur();

    // creation des composants
    private JLabel
        kmhLabel = new JLabel("KM/H", JLabel.CENTER),
        noeudsLabel = new JLabel("NOEUDS", JLabel.CENTER),
        forceLabel = new JLabel(
            "INDICATEUR DE FORCE (ECHELLE DE BEAUFORT) = 0", JLabel.CENTER);

    private JTextField
        kmh = new JTextField("0.0"),
        noeuds = new JTextField("0.0");

    private JButton raz = new JButton("RAZ");

    private JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 120, 0);

    //...
```

11

Exemple

```
// suite de la classe AnemometreGraphique ...
// dans le constructeur

public AnemometreGraphique() {

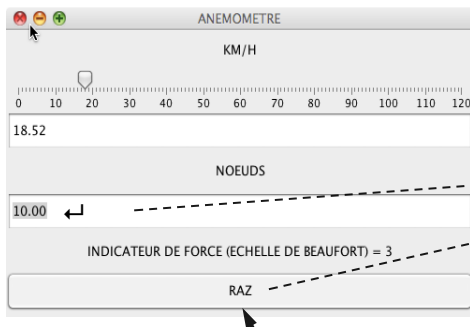
    // choix du LayoutManager
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(7, 1));

    // Ajout des composants
    add(this.kmhLabel); // = this.getContentPane().add(...);
    add(this.slider);
    add(this.kmh);
    add(this.noeudsLabel);
    add(this.noeuds);
    add(this.forceLabel);
    add(this.raz);

    // a suivre...
}
}
```

12

II.3 Dynamique de l'interface graphique



Convertisseur	
<i>Attributs</i>	
protected double	vent
<i>Operations</i>	
public void	setKMH(double v)
public void	setNoeuds(double n)
public void	raz()
public double	getKMH()
public double	getNoeuds()
public int	getForce()

13

II.3 Dynamique de l'interface graphique

TextField
noeuds

10.00 ←

actionPerformed(ActionEvent x)

?

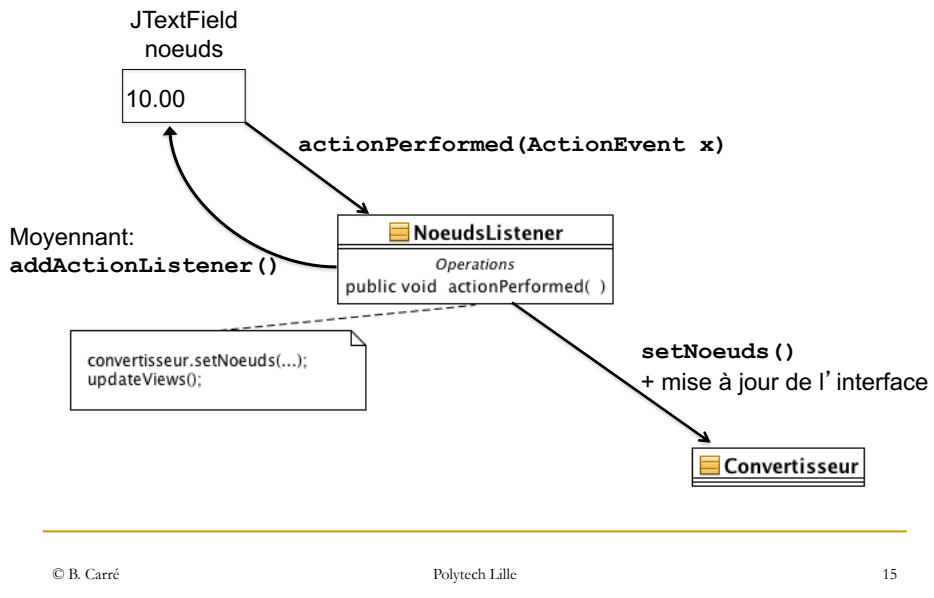
setNoeuds ()

+ mise à jour de l' interface

Convertisseur

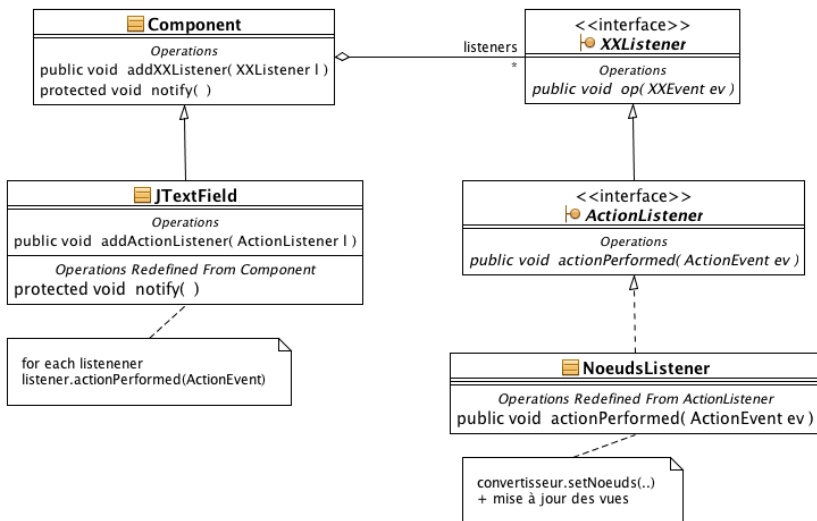
14

II.3 Dynamique de l'interface graphique



15

Design Pattern Observer



16

Evènements

- Représentés par des objets évènements

```
abstract class java.awt.AWTEvent
class java.awt.event.ActionEvent
class java.awt.event.AdjustmentEvent
class java.awt.event.ComponentEvent
class java.awt.event.ContainerEvent
class java.awt.event.FocusEvent
class java.awt.event.InputEvent
class java.awt.event.KeyEvent
class java.awt.event.MouseEvent
class java.awt.event.WindowEvent
class java.awt.event.ItemEvent
class java.awt.event.TextEvent
```

- ... Générés par des composants « sources d'évènements »

Sources d'évènements

Composant	Evènements générés
JApplet	ContainerEvent, KeyEvent, MouseEvent, ...
JFrame	WindowEvent , ContainerEvent, KeyEvent, MouseEvent, ...
JButton	ActionEvent , KeyEvent, MouseEvent, ...
JTextField	ActionEvent , TextEvent, KeyEvent, MouseEvent, ...
JMenu, JMenuItem, ...	ActionEvent , ...
JSlider, ...	ChangeEvent , ...

Ecouteurs d'évènements

Les composants sources d'évènement informent des objets «écouteurs» de l'application

- au travers de protocoles spécifiés par des interfaces de `Listener`
- à un type d'évènement `XXEvent` correspond une interface `XXListener`
- que les écouteurs doivent implémenter pour les besoins de l'application (traitement de l'évènement)
- => programmer en conséquence les classes d'écouteurs de l'application implémentant les interfaces concernées
- *Remarque:*
Certains listeners fournissent des implantations par défaut au travers d'une classe `Adapter` (extensible pour redéfinition).

Interfaces d'écouteurs

<i>Interface de Listener</i>	<i>méthodes</i>	<i>classe Adapter</i>
ActionListener	actionPerformed(ActionEvent)	
WindowListener	windowActivated(WindowEvent) windowClosing(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)	WindowAdapter
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	MouseAdapter
ChangeListener	stateChanged(ChangeEvent)	
...		

Branchement des écouteurs

- “abonner” les objets listeners (soit *l*) à l’écoute des composants sources (soit *c*) par envois de messages de la forme :
`c.addXXListener(l)`
 - avec: *XX* = nom de l’événement (*XXEvent*)
 - qui détermine aussi le nom de l’interface du listener correspondant : *XXListener*
- ces messages sont définis pour chaque type de composants susceptibles de générer l’évènement correspondant.
Par exemple:
`addActionListener(ActionListener l)`
est définie dans : `JButton`, `Jtextfield`
- « désabonner »:
`c.removeXXListener(XXListener)`

Exemple: programmer la dynamique

```
public class AnemometreGraphique extends JFrame {
    // 1. programmer les (inners) classes des ecouteurs

    class NoeudsListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            double vitesse = Double.parseDouble(noeuds.getText());
            convertisseur.setNoeuds(vitesse);
            updateFromDigits();
        }
    }

    class KmhSliderListener implements ChangeListener {
        public void stateChanged(ChangeEvent event) {
            int vitesse = slider.getValue();
            convertisseur.setKMH(vitesse);
            updateFromSlider();
        }
    }

    class RAZListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            slider.setValue(0); // => change...
        }
    }
}
```

Exemple (suite)

```
class KmhListener implements ActionListener {
    public void actionPerformed(ActionEvent arg0) {
        double vitesse = Double.parseDouble(kmh.getText());
        convertisseur.setKMH(vitesse);
        updateFromDigits();
    }
}

// utilitaires
void updateFromSlider() {
    this.slider.setValue((int) convertisseur.getKMH());
    this.kmh.setText(String.valueOf(
        new DecimalFormat("0.00").format(
            convertisseur.getKMH())));
    this.noeuds.setText(String.valueOf(
        new DecimalFormat("0.00").format(
            convertisseur.getNoeuds())));
    this.forceLabel.setText(
        "INDICATEUR DE FORCE (ECHELLE DE BEAUFORT) = "
        + convertisseur.getForce());
}
```

Exemple (suite)

```
void updateFromDigits() {
    double vitesse = convertisseur.getKMH();
    this.slider.setValue((int) vitesse); // mais approxime ...
    convertisseur.setKMH(vitesse); // retablissement ...
    this.kmh.setText(String.valueOf(
        new DecimalFormat("0.00").format(convertisseur.getKMH())));
    this.noeuds.setText(String.valueOf(
        new
        DecimalFormat("0.00").format(convertisseur.getNoeuds())));
    this.forceLabel.setText(
        "INDICATEUR DE FORCE (ECHELLE DE BEAUFORT) = "
        + convertisseur.getForce());
}

// 2. instancier et brancher les ecouteurs sur les composants
// dans le constructeur
public AnemometreGraphique () { // suite..
    this.noeuds.addActionListener(new NoeudsListener());
    this.slider.addChangeListener(new KmhSliderListener());
    this.kmh.addActionListener(new KmhListener());
    this.raz.addActionListener(new RAZListener());
}
}
```

Classes internes, anonymes

- Classes internes
 - les listeners étant propres à (la classe de) l'interface graphique, on les programme généralement par des "inner classes"
 - d'où génération de classes de la forme (ne pas supprimer!):
`AnemometreGraphique$NoeudsListener.class`
- et anonymes
 - Identification superflue si elles ne servent qu'une fois, d'où:
`noeuds.addActionListener(new ActionListener() {`
`public void actionPerformed(ActionEvent e) {`
`double vitesse = Double.parseDouble(noeuds.getText());`
`convertisseur.setNoeuds(vitesse);`
`updateFromDigits();`
`}});`
 - les classes générées sont alors « numérotées » (ne pas supprimer!):
`AnemometreGraphique$1.class, ...`

Et même par Lambdas expressions ...

- Interfaces de `Listener` avec 1 seule méthode = *interfaces fonctionnelles*
+ implémentation sans état (sans variables d'instances)
=> formulation possible par *lambda expression*

```
noeuds.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        double vitesse = Double.parseDouble(noeuds.getText());
        convertisseur.setNoeuds(vitesse);
        updateFromDigits();
    }
});
```

=>

```
noeuds.addActionListener( // par Lambda expression
    e -> { // meme code ...
        double vitesse = Double.parseDouble(noeuds.getText());
        convertisseur.setNoeuds(vitesse);
        updateFromDigits();
    }
});
```

Activation de l'application

- instancier l'application en précisant les paramètres de forme
- activer l'interface graphique : `setVisible(true)`
- Exemple:

```
public class Main{
    public static void main(String argv[]) {
        AnemometreGraphique fenetre
            = new AnemometreGraphique();
        fenetre.setSize(500, 300);
        fenetre.setTitle("ANEMOMETRE");
        fenetre.setVisible(true);
    }
}
```

Applet

- Structure
 - sous-classe `java.applet.Applet` ou `javax.swing.JApplet`
 - instancier et positionner les composants dans la méthode `init()` du protocole des applets (cf. activation)
- Evénements
 - idem sauf les `WindowEvent` gérés par le browser
- Activation
 - via une page HTML : tags `<APPLET>` `</APPLET>`
 - protocole standard du browser :
`init()` `->` `start()` `<->` `stop()` `->` `destroy()`
- Pour des raisons de sécurité, les applets ont moins de droits qu'une application autonome:
 - pas d'accès au système client où elles s'exécutent (fichiers, programmes, bibliothèques dynamiques),
 - communication restreinte au serveur d'où elles proviennent.

De l'applet à l'application...

```
import javax.swing.*;
import java.awt.event.*;
public class Fenetre extends JFrame { // « conteneur » de l'applet
    AnemometreApplet applet = new AnemometreApplet(); // creer l'applet
    Fenetre() {
        this.getContentPane().add(applet); // l'ajouter a l'application
        applet.init(); // et la lancer « a la main »
        applet.start();
        this.addWindowListener(new FenetreListener());
    }

    class FenetreListener extends WindowAdapter {
        public void windowClosing(WindowEvent e) {System.exit(0);}
    }
    public static void main(String argv[]) { // pour tester...
        Fenetre fenetre= new Fenetre();
        fenetre.setSize(500, 300);
        fenetre.setTitle("ANEMOMETRE");
        fenetre.setVisible(true);
    }
}
```

En guise de conclusion

- Bien séparer
 - l'application elle-même: « modèle » ou “noyau fonctionnel”
 - des modes d'interaction: interfaces, « vue(s) »
- Permet de greffer plusieurs interfaces, ou en changer :
 - frames, applets, console (mode ligne)
 - catégories d'utilisateurs = interfaces différentes
- C'est un critère de réutilisation
 - modèles applicatifs réutilisables indépendamment de leurs modes d'interaction
 - en particulier dans une perspective client-serveur: application sur une machine, interactions sur d'autres postes ou dispositifs (mobiles,...).