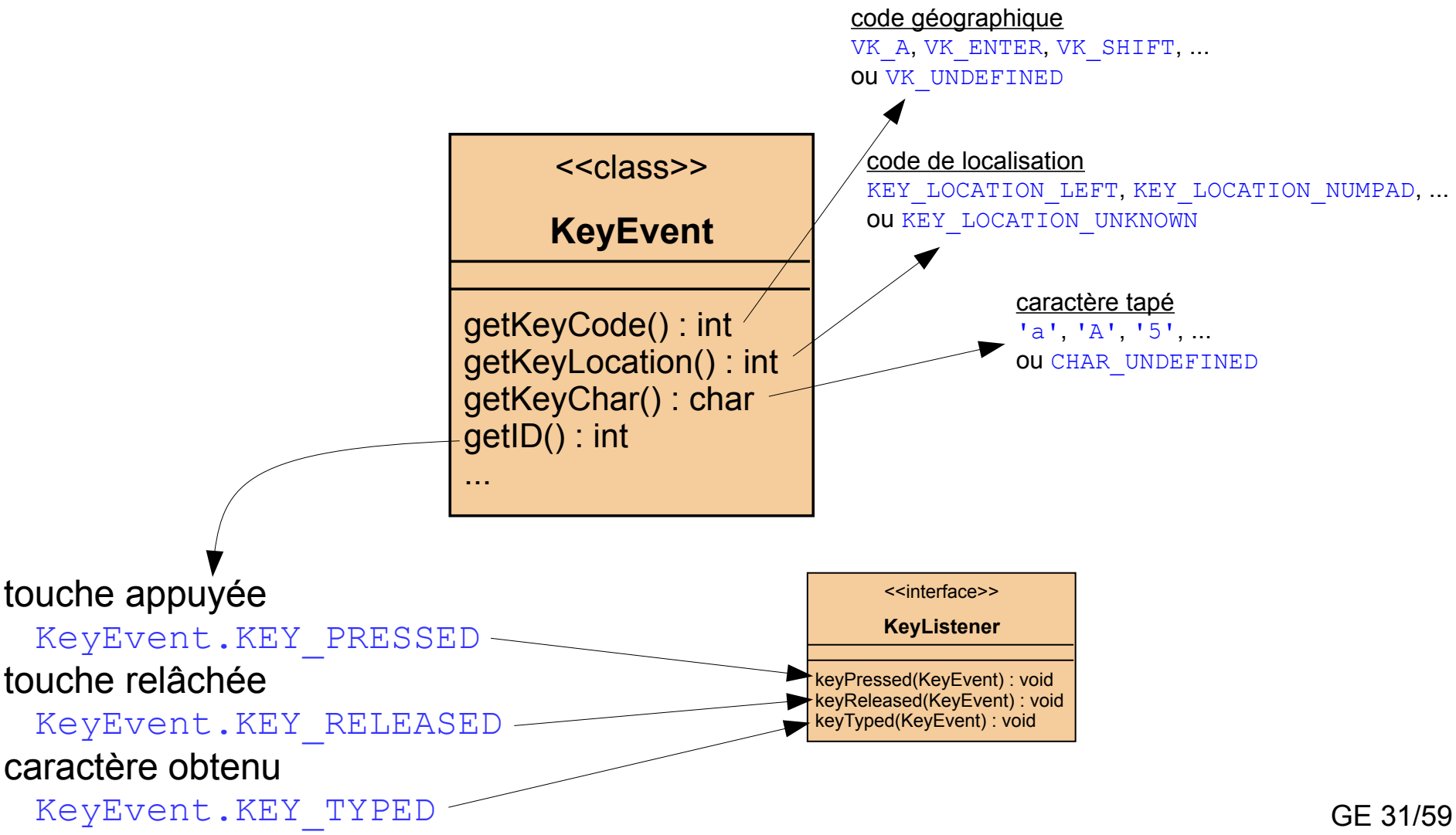
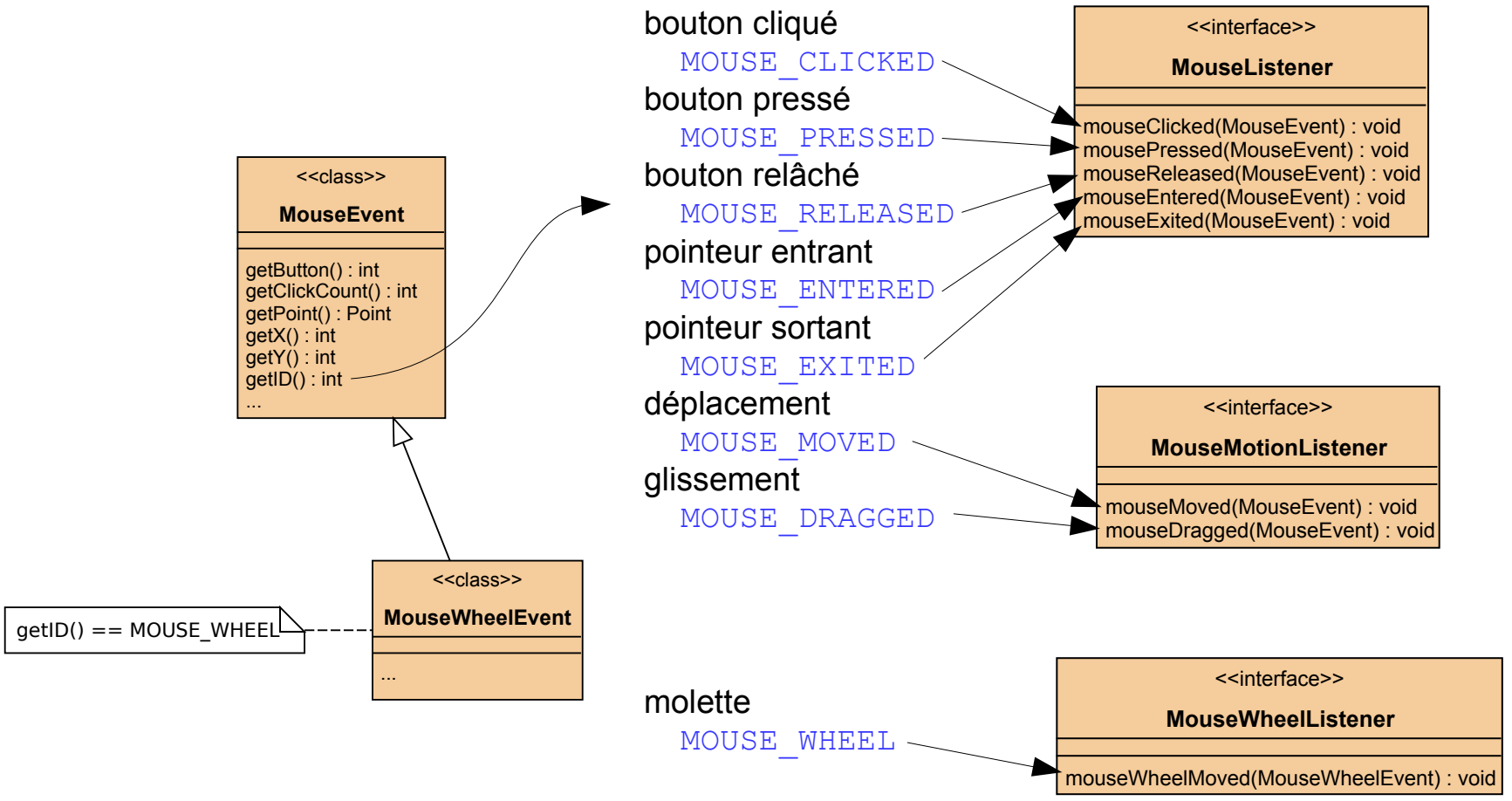
Détection des événements clavier



- Exemples :

Touche frappée	Type evt	Key Code	Key Location	Key Char
A	KEY_PRESSED	VK_A	KEY_LOCATION_STANDARD	a
	KEY_TYPED	VK_UNDEFINED	KEY_LOCATION_UNKNOWN	a
	KEY_RELEASED	VK_A	KEY_LOCATION_STANDARD	a
Ctrl(g)	KEY_PRESSED	VK_CONTROL	KEY_LOCATION_LEFT	CHAR_UNDEFINED
	KEY_RELEASED	VK_CONTROL	KEY_LOCATION_LEFT	CHAR_UNDEFINED
Ctrl(d)	KEY_PRESSED	VK_CONTROL	KEY_LOCATION_RIGHT	CHAR_UNDEFINED
	KEY_RELEASED	VK_CONTROL	KEY_LOCATION_RIGHT	CHAR_UNDEFINED
BackSpace	KEY_PRESSED	VK_BACK_SPACE	KEY_LOCATION_STANDARD	\b
	KEY_TYPED	VK_UNDEFINED	KEY_LOCATION_UNKNOWN	\b
	KEY_RELEASED	VK_BACK_SPACE	KEY_LOCATION_STANDARD	\b
flèche haut	KEY_PRESSED	VK_UP	KEY_LOCATION_STANDARD	CHAR_UNDEFINED
	KEY_RELEASED	VK_UP	KEY_LOCATION_STANDARD	CHAR_UNDEFINED
Shift+A	KEY_PRESSED	VK_SHIFT	KEY_LOCATION_LEFT	CHAR_UNDEFINED
	KEY_PRESSED	VK_A	KEY_LOCATION_STANDARD	A
	KEY_TYPED	VK_UNDEFINED	KEY_LOCATION_UNKNOWN	A
	KEY_RELEASED	VK_A	KEY_LOCATION_STANDARD	A
	KEY_RELEASED	VK_SHIFT	KEY_LOCATION_LEFT	CHAR_UNDEFINED

Détection des événements souris



java.awt.event
InputEvent

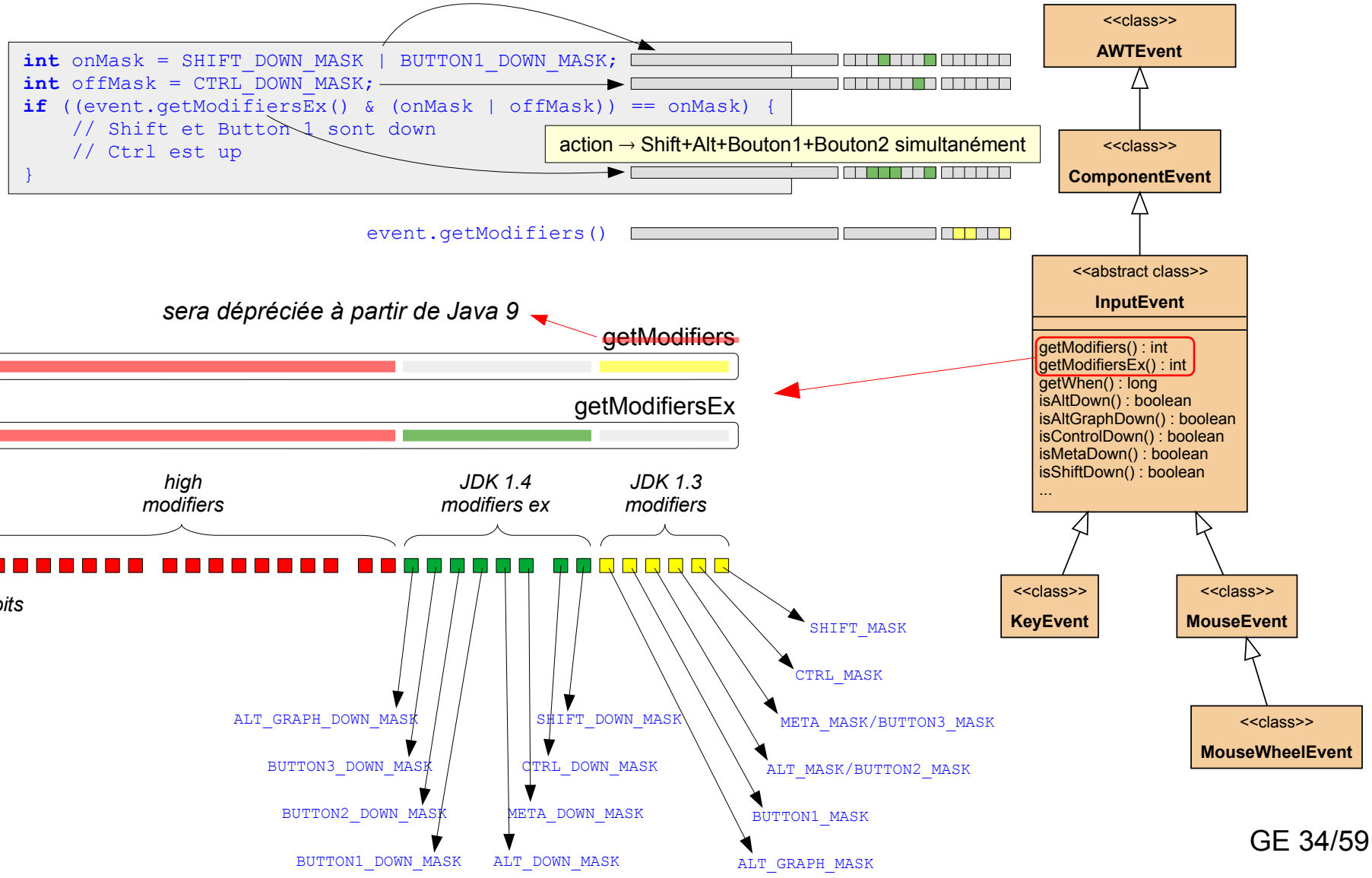


Tableau récapitulatif

This table lists Swing components with their specialized listeners

Component	Action Listener	Caret Listener	Change Listener	Document Listener, Undoable Edit Listener	Item Listener	List Selection Listener	Window Listener	Other Types of Listeners
button	✔		✔		✔			
check box	✔		✔		✔			
color chooser			✔					
combo box	✔				✔			
dialog							✔	
editor pane		✔		✔				hyperlink
file chooser	✔							
formatted text field	✔	✔		✔				
frame							✔	
internal frame								internal frame
list						✔		list data
menu								menu
menu item	✔		✔		✔			menu key menu drag mouse
option pane								
password field	✔	✔		✔				
popup menu								popup menu
progress bar			✔					
radio button	✔		✔		✔			
slider			✔					
spinner			✔					
tabbed pane			✔					
table						✔		table model table column model cell editor
text area		✔		✔				
text field	✔	✔		✔				
text pane		✔		✔				hyperlink
toggle button	✔		✔		✔			
tree								tree expansion tree will expand tree model tree selection
viewport (used by scrollpane)			✔					

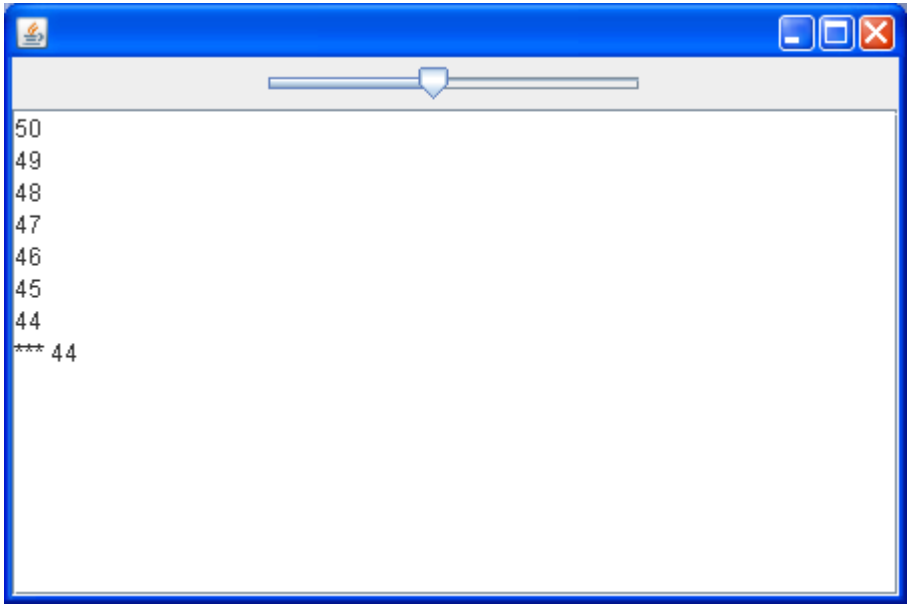
source
https://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html#many

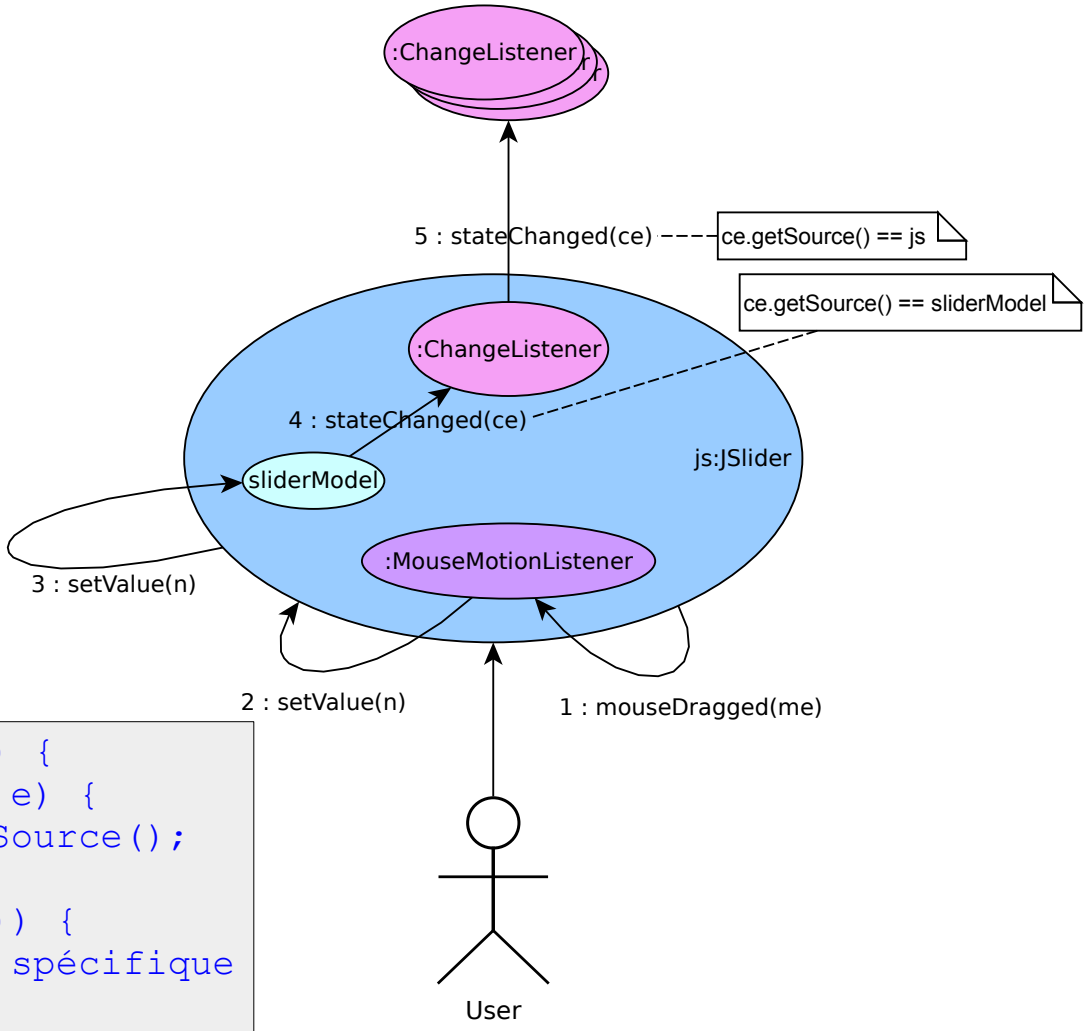
javax.swing.event
ChangeEvent

- `ChangeEvent` : classe des *événements légers*
  - ne connaît que la source de l'événement
- Contexte :
  - notifications très fréquentes
    - ou bien détails inutiles
  - source auto-suffisante
- Utilisation :
  - un seul événement
  - stocké dans la source
  - réémis à chaque fois

économie de temps et d'espace

Exemple

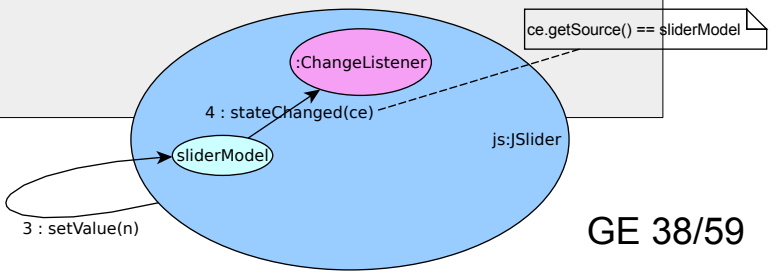





```
js.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        JSlider source = (JSlider) e.getSource();
        int val = source.getValue();
        if (!source.getValueIsAdjusting()) {
            // curseur relâché -> action spécifique
            output.append("*** ");
        }
        output.append(val + "\n");
    }
});
```

Extrait de la classe `DefaultBoundedRangeModel`

```
public class DefaultBoundedRangeModel implements BoundedRangeModel, ... {
    protected transient ChangeEvent changeEvent = null;
    protected EventListenerList listenerList = new EventListenerList();
    ...
    public void setValue(int n) {
        ...
        value = <acceptable value for n>;
        fireStateChanged();
    }
    protected void fireStateChanged() {
        Object[] listeners = listenerList.getListenerList();
        for (int i = listeners.length - 2; i >= 0; i -= 2) {
            if (listeners[i] == ChangeListener.class) {
                if (changeEvent == null) {
                    changeEvent = new ChangeEvent(this);
                }
                ((ChangeListener) listeners[i+1]).stateChanged(changeEvent);
            }
        }
    }
}
```

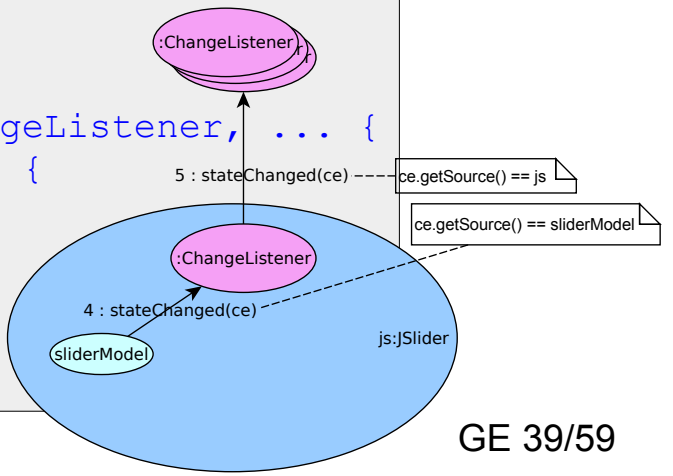


Extrait de la classe `JSlider`

```
public class JSlider extends JComponent implements ... {
    protected BoundedRangeModel sliderModel;
    protected transient ChangeEvent changeEvent = null;
    public JSlider(...) {
        ...
        sliderModel.addChangeListener(new ModelListener());
    }
    protected void fireStateChanged() {
        Object[] listeners = listenerList.getListenerList();
        for (int i = listeners.length - 2; i >= 0; i -= 2) {
            if (listeners[i] == ChangeListener.class) {
                if (changeEvent == null) {
                    changeEvent = new ChangeEvent(this);
                }
                ((ChangeListener) listeners[i+1])
                        .stateChanged(changeEvent);
            }
        }
    }
    private class ModelListener implements ChangeListener, ... {
        public void stateChanged(ChangeEvent e) {
            fireStateChanged();
        }
    }
    ...
}
```

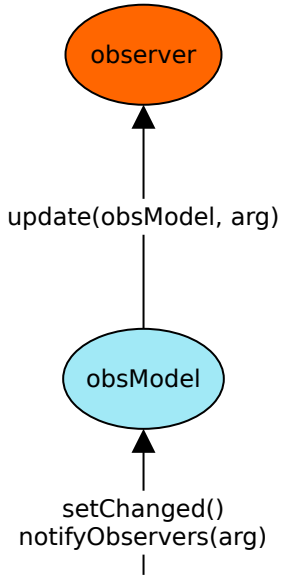
récupère les `Listener`s associés au `JSlider` **pas à son modèle** !

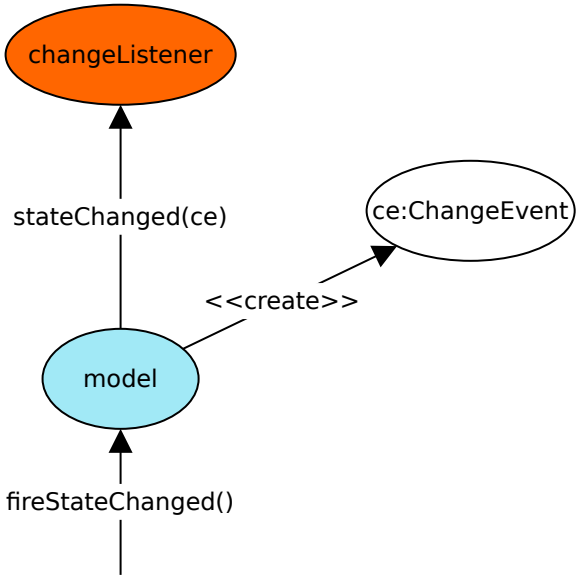
attribut hérité de `JComponent`

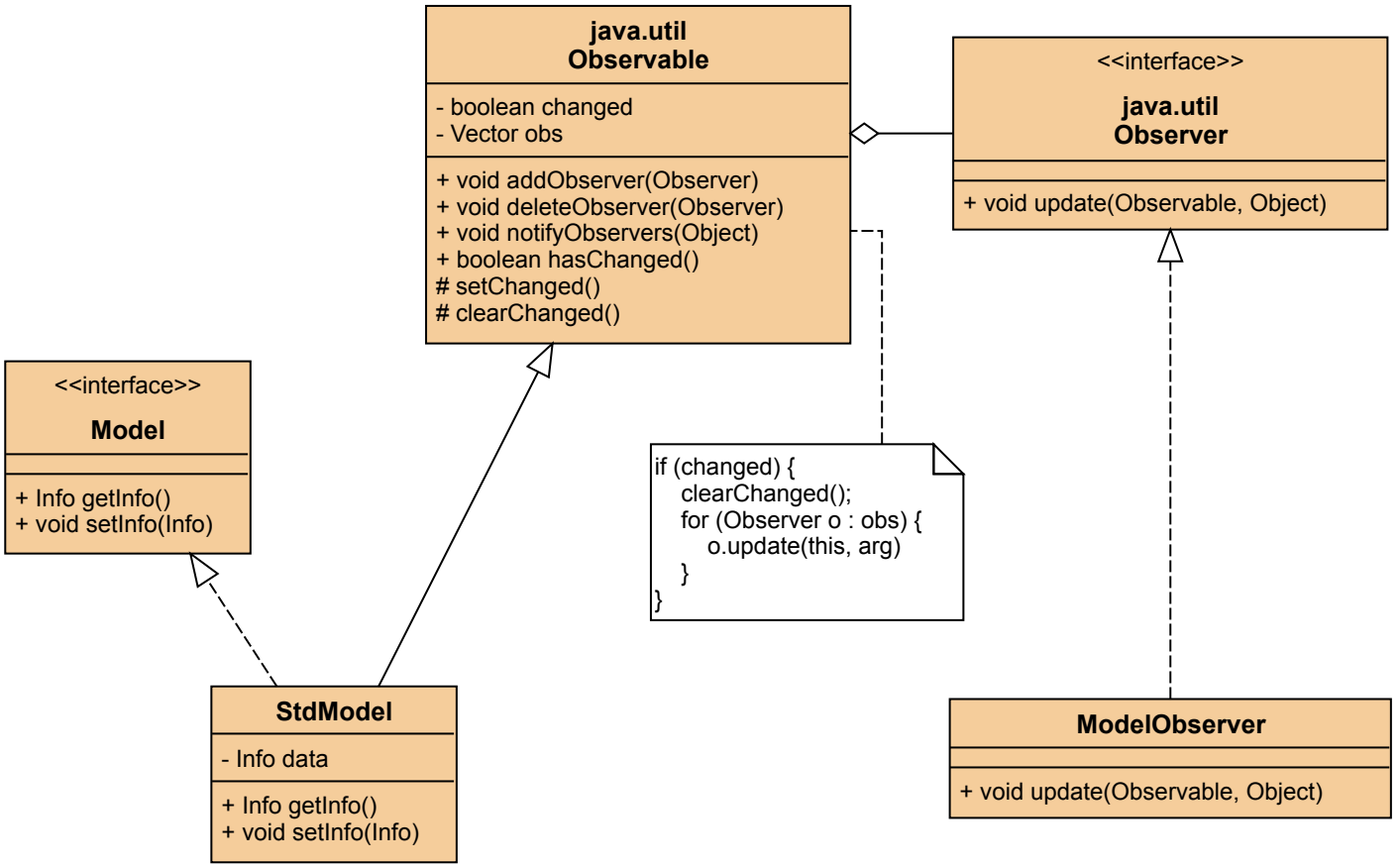


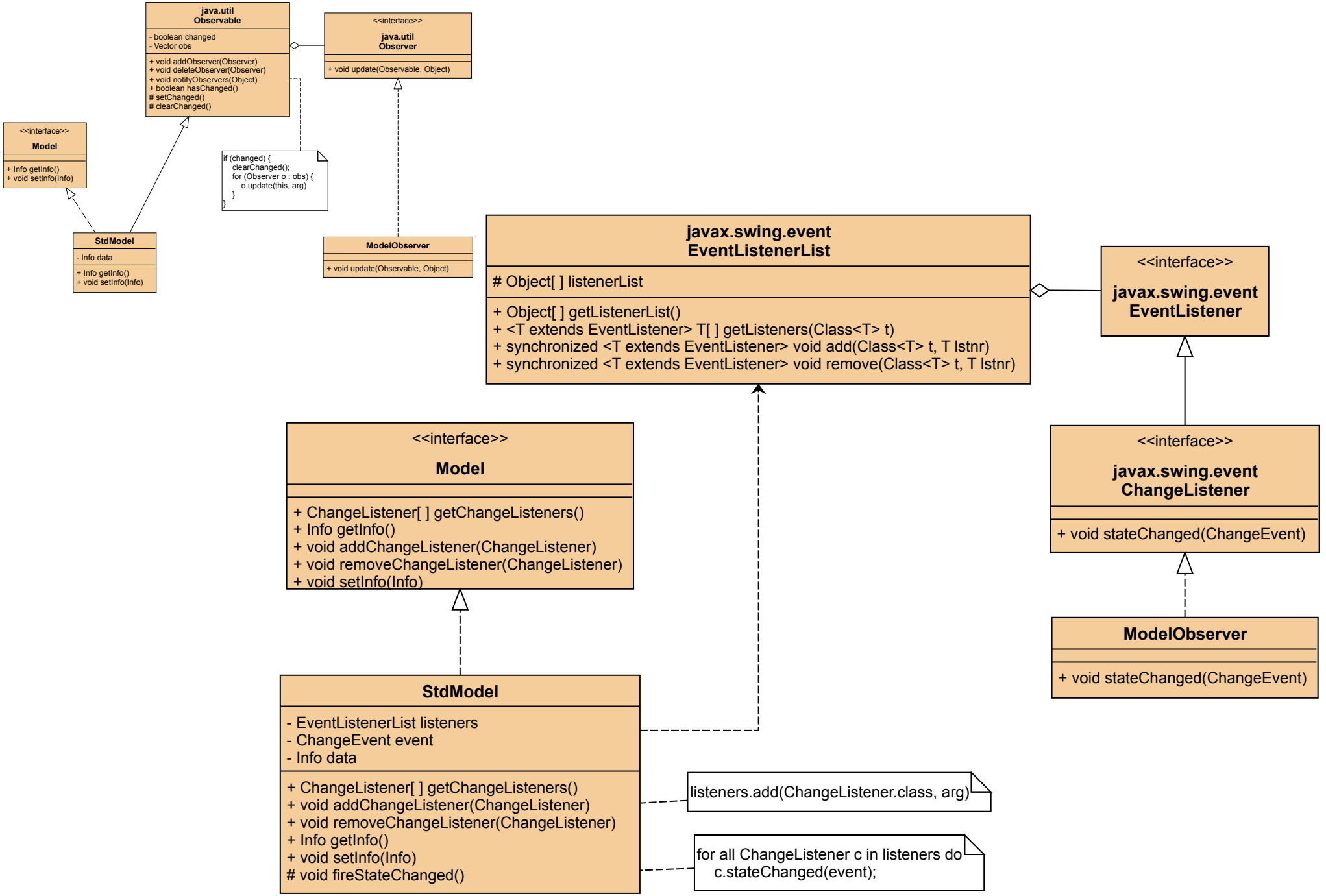
Comparaison avec le pattern Observer

- On peut utiliser un `ChangeListener` au lieu du pattern `Observer`/`Observable` (Java 1.0)






java.util
Observable
- boolean changed
- Vector obs
+ void addObserver(Observer)
+ void deleteObserver(Observer)
+ void notifyObservers(Object)
+ boolean hasChanged()
# setChanged()
# clearChanged()
<<interface>>
java.util
Observer
+ void update(Observable, Object)
<<interface>>
Model
+ Info getInfo()
+ void setInfo(Info)
if (changed) {
clearChanged();
for (Observer o : obs) {
o.update(this, arg)
}
}
StdModel
- Info data
+ Info getInfo()
+ void setInfo(Info)
ModelObserver
+ void update(Observable, Object)


java.util
Observable
- boolean changed
- Vector obs
+ void addObserver(Observer)
+ void deleteObserver(Observer)
+ void notifyObservers(Object)
+ boolean hasChanged()
# setChanged()
# clearChanged()
<<interface>>
java.util
Observer
+ void update(Observable, Object)
<<interface>>
Model
+ Info getInfo()
+ void setInfo(Info)
if (changed) {
clearChanged();
for (Observer o : obs) {
o.update(this, arg)
}
}
StdModel
- Info data
+ Info getInfo()
+ void setInfo(Info)
ModelObserver
+ void update(Observable, Object)
javax.swing.event
EventListenerList
Object[ ] listenerList
+ Object[ ] getListenerList()
+ <T extends EventListener> T[ ] getListeners(Class<T> t)
+ synchronized <T extends EventListener> void add(Class<T> t, T lstnr)
+ synchronized <T extends EventListener> void remove(Class<T> t, T lstnr)
<<interface>>
javax.swing.event
EventListener
<<interface>>
Model
+ ChangeListener[ ] getChangeListeners()
+ Info getInfo()
+ void addChangeListener(ChangeListener)
+ void removeChangeListener(ChangeListener)
+ void setInfo(Info)
<<interface>>
javax.swing.event
ChangeListener
+ void stateChanged(ChangeEvent)
ModelObserver
+ void stateChanged(ChangeEvent)
StdModel
- EventListenerList listeners
- ChangeEvent event
- Info data
+ ChangeListener[ ] getChangeListeners()
+ void addChangeListener(ChangeListener)
+ void removeChangeListener(ChangeListener)
+ Info getInfo()
+ void setInfo(Info)
# void fireStateChanged()
listeners.add(ChangeListener.class, arg)
for all ChangeListener c in listeners do
c.stateChanged(event);

Optimisations de `fireStateChanged`

bcp plus rapide d'utiliser `Object[] getListenerList()`
que `T[] getListeners(Class<T>)`

le tableau retourné est garanti non **`null`**

```
protected void fireStateChanged() {
    Object[] listeners = listenerList.getListenerList();
    for (int i = listeners.length - 2; i >= 0; i -= 2) {
        if (listeners[i] == ChangeListener.class) {
            if (changeEvent == null) {
                changeEvent = new ChangeEvent(this);
            }
            ((ChangeListener) listeners[i + 1]).stateChanged(changeEvent);
        }
    }
}
```

plus rapide de tester `i >= 0`
que `i < listeners.length - 1`

plus frugal de ne créer l'événement
que s'il y a des `ChangeListener`s

Exemple : Rainbow

```java
import java.awt.Color;

import javax.swing.event.ChangeListener;

/**
 * Ce modèle permet d'obtenir des couleurs.
 * On peut faire évoluer le modèle en lui commandant de changer de couleur,
 *  et consulter son état à tout moment.
 * @inv <pre>
 *      getColor() != null </pre>
 */
public interface RainbowModel {

    // REQUETES

    /**
     * La couleur courante du modèle.
     */
    Color getColor();

    ChangeListener[] getChangeListeners();

    // COMMANDES

    void addChangeListener(ChangeListener lst);

    /**
     * Modification de l'état interne.
     * Un appel à cette méthode oblige le modèle à changer de couleur.
     * @post <pre>
     *      !getColor().equals(old getColor()) </pre>
     */
    void changeColor();

    void removeChangeListener(ChangeListener lst);
}
```

```java
public class StdRainbowModel extends Observable implements RainbowModel {

    // ATTRIBUTS

    private static final Color[] COLORS = {
        Color.BLACK, Color.BLUE, Color.CYAN, Color.DARK_GRAY, Color.GRAY,
        Color.GREEN, Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE,
        Color.PINK, Color.RED, Color.WHITE, Color.YELLOW
    };
    private int currentColorIndex;

    // CONSTRUCTEURS

    /**
     * Un modèle standard, dont la couleur courante est la première couleur
     *  du tableau.
     */
    public StdRainbowModel() {
        currentColorIndex = 0;
    }

    // REQUETES

    @Override
    public Color getColor() {
        return COLORS[currentColorIndex];
    }

    // COMMANDES

    @Override
    public void changeColor() {
        currentColorIndex = (currentColorIndex + 1) % COLORS.length;
        // le modèle notifie le changement d'état
        setChanged();
        notifyObservers();
    }
}
```

```java
public class StdRainbowModel implements RainbowModel {
    // ATTRIBUTS
    ...
    private final EventListenerList listeners;
    private final ChangeEvent event;

    // CONSTRUCTEURS
    /**
     * Un modèle standard, ...
     */
    public StdRainbowModel() {
        currentColorIndex = 0;
        listeners = new EventListenerList();
        event = new ChangeEvent(this);
    }

    // REQUETES
    ...
    @Override
    public ChangeListener[] getChangeListeners() {
        return listeners.getListeners(ChangeListener.class);
    }

    // COMMANDES
    @Override
    public void changeColor() {
        currentColorIndex = (currentColorIndex + 1) % COLORS.length;
        // le modèle notifie le changement d'état
        fireStateChanged();
    }

    @Override
    public void addChangeListener(ChangeListener lst) {
        listeners.add(ChangeListener.class, lst);
    }

    @Override
    public void removeChangeListener(ChangeListener lst) {
        listeners.remove(ChangeListener.class, lst);
    }

    // OUTILS
    protected void fireStateChanged() {
        Object[] lst = listeners.getListenerList();
        for (int i = lst.length - 2; i >= 0; i -= 2) {
            if (lst[i] == ChangeListener.class) {
                ((ChangeListener) lst[i + 1]).stateChanged(event);
            }
        }
    }
}
```

```java
public class Rainbow {

    // ATTRIBUTS

    ...

    // CONSTRUCTEURS

    public Rainbow() {
        // MODELE
        ...
        // VUE
        ...
        placeComponents();
        // CONTROLEUR
        connectControllers();
    }

    // COMMANDES

    /**
     * Rend l'application visible au centre de l'écran.
     */
    public void display() {
        ...
    }

    // OUTILS

    ...

    private void createController() {
        ((Observable) model).addObserver(new Observer() {
            @Override
            public void update(Observable o, Object arg) {
                refresh();
            }
        });
        ...
    }

    // POINT D'ENTREE

    public static void main(String[] args) {
        ...
    }
}
```

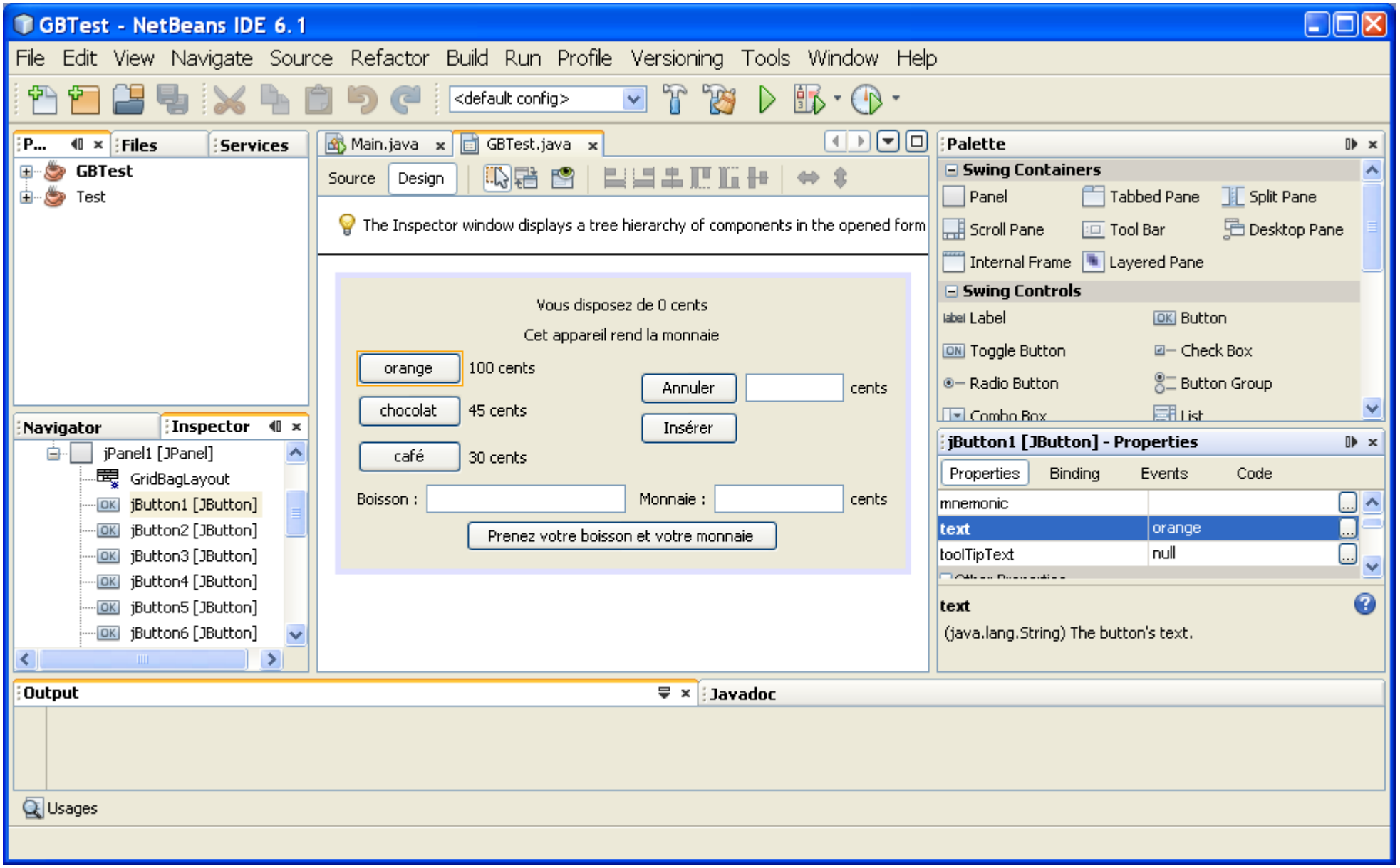
```java
public class Rainbow {

    // ATTRIBUTS

    ...

    // CONSTRUCTEURS

    public Rainbow() {
        // MODELE
        ...
        // VUE
        ...
        placeComponents();
        // CONTROLEUR
        connectControllers();
    }

    // COMMANDES

    /**
     * Rend l'application visible au centre de l'écran.
     */
    public void display() {
        ...
    }

    // OUTILS

    ...

    private void createController() {
        model.addChangeListener(new ChangeListener() {
            @Override
            public void stateChanged(ChangeEvent e) {
                refresh();
            }
        });
        ...
    }

    // POINT D'ENTREE

    public static void main(String[] args) {
        ...
    }
}
```

Rapide introduction à JavaBean

- *JavaBean*
  - spécification
  - composants Java réutilisables (graphiques ou pas)
  - pouvant être intégrés « à chaud » dans un logiciel
- Points clés :
  - introspection dynamique du code
  - configuration de l'apparence et du comportement
  - communication par événements
  - propriétés accessibles (lecture/écriture) au niveau du code
  - persistance

Propriétés des beans

- Un bean définit des *propriétés* qui caractérisent son état interne et son apparence
- Une propriété est définie par :
  - un nom (`name`)
  - un éventuel accesseur en lecture (`getName`/`isName`)
  - un éventuel accesseur en écriture (`setName`)
- Tous les composants graphiques AWT/Swing sont des beans


GBTest - NetBeans IDE 6.1
File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help
<default config>
Files
Services
GBTest
Test
Main.java
GBTest.java
Source
Design
The Inspector window displays a tree hierarchy of components in the opened form
Vous disposez de 0 cents
Cet appareil rend la monnaie
orange
100 cents
chocolat
45 cents
café
30 cents
Annuler
Insérer
cents
Boisson :
Monnaie :
cents
Prenez votre boisson et votre monnaie
Palette
Swing Containers
Panel
Tabbed Pane
Split Pane
Scroll Pane
Tool Bar
Desktop Pane
Internal Frame
Layered Pane
Swing Controls
Label
Button
Toggle Button
Check Box
Radio Button
Button Group
Navigator
Inspector
jPanel1 [JPanel]
GridBagLayout
jButton1 [JButton]
jButton2 [JButton]
jButton3 [JButton]
jButton4 [JButton]
jButton5 [JButton]
jButton6 [JButton]
jButton1 [JButton] - Properties
Properties
Binding
Events
Code
mnemonic
text
orange
toolTipText
null
text
(java.lang.String) The button's text.
Output
Javadoc
Usages

Propriété simple

- *Propriété simple* (*simple property*) : constituée d'une seule valeur, indépendante des autres propriétés
- Exemple : propriété de nom `"background"`
  - Attribut mémorisant la valeur de la propriété
    - `Color background`
  - Accesseur en lecture
    - `public Color getBackground() { ... }`
  - Accesseur en écriture
    - `public void setBackground(Color c) { ... }`

Exemple : la propriété `background` de `Component`

```
public Color getBackground() {
    Color background = this.background;
    if (background != null) {
        return background;
    }
    Container parent = this.parent;
    return (parent != null) ? parent.getBackground() : null;
}
public void setBackground(Color c) {
    Color oldColor = background;
    ComponentPeer peer = this.peer;
    background = c;
    if (peer != null) {
        c = getBackground();
        if (c != null) {
            peer.setBackground(c);
        }
    }
    // This is a bound property, so report the change to
    // any registered listeners.  (Cheap if there are none.)
    firePropertyChange("background", oldColor, c);
}
```

Propriété indexée

- *Propriété indexée* (*indexed property*) : constituée d'une séquence de valeurs de type `X`
- Accesseur en lecture
  - `X[] getName()`
- Accesseur en écriture
  - `void setName(X[] values)`
- Accès aux éléments
  - `X getName(int i)`
  - `void setName(int i, X value)`

Propriété liée

- *Propriété liée* (*bound property*) : tout changement de valeur est notifié (par événements) aux écouteurs préalablement enregistrés
- Accesseur en écriture (`setName`) :
  - génère obligatoirement des événements de type `PropertyChangeEvent`
  - notifie les `PropertyChangeListener` enregistrés
    - optimisation : seulement si la propriété change de valeur

Propriété contrainte

- *Propriété contrainte* (*constrained property*) : propriété liée dont le changement de valeur est soumis à l'approbation d'autres composants
- Accesseur en écriture (`setName`) :
  - notifie des `VetoableChangeListener` avant les `PropertyChangeListener`
  - autorise le changement de valeur seulement s'il n'y a pas eu de veto

PCS = PropertyChangeSupport
PCE = PropertyChangeEvent
PCL = PropertyChangeListener

Propriété liée

- Propriété liée = les changements de valeur sont observables à l'aide d'un *PCL*

```
public class MonBean {
    public static final String PROP_P = "p";
    private X p;
    private PropertyChangeSupport pcs;

    public MonBean() {
        p = <init value>;
        pcs = new PropertyChangeSupport(this);
    }

    public PCL[] getPCLs([String n]) { ... return pcs.getPropertyChangeListeners(n); }
    public void addPCL([String n, ]PCL pcl) { ... pcs.addPropertyChangeListener([n, ]pcl); }
    public void removePCL([String n, ]PCL pcl) { ... pcs.removePropertyChangeListener([n, ]pcl); }

    public X getP() {
        return p;
    }
    public void setP(X newValue) {
        X oldValue = p;
        p = newValue;
        pcs.firePropertyChange(
                PROP_P,
                oldValue,
                newValue);
    }
}
```

(principe du) code de *PCS*.firePropertyChange(p, ov, nv) :

```
if (ov != null && nv != null && ov.equals(nv)) {
    return;
}
PCE e = new PCE(source, p, ov, nv);
for (PCL pcl : <tousLesPCL>) {
    pcl.propertyChange(e);
}
```

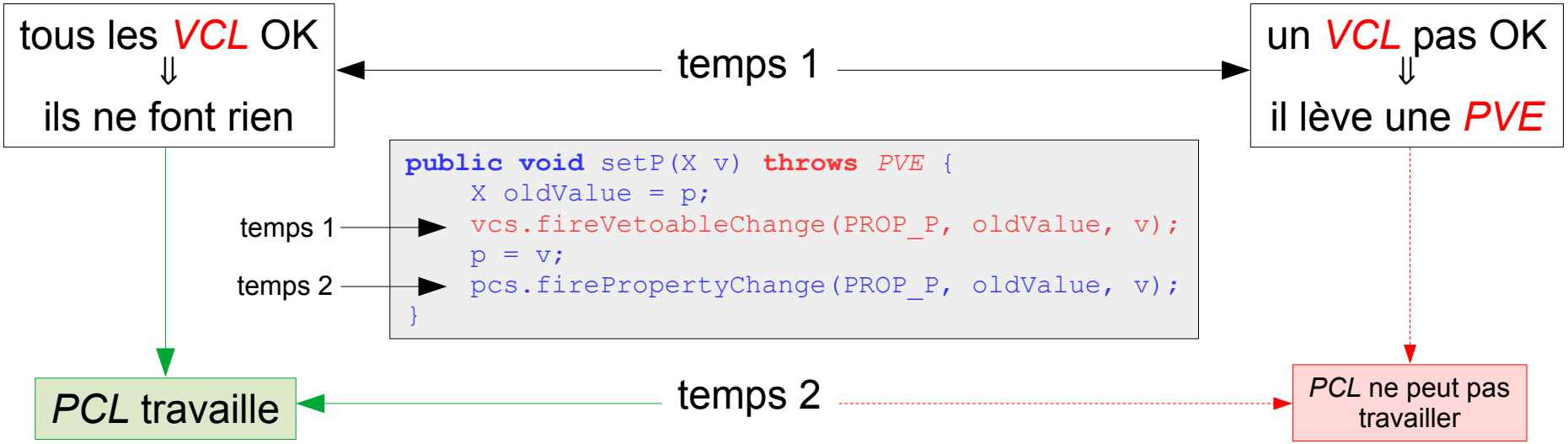
utilisation du bean :

```
monBean.addPCL([PROP_P,]
    new PCL() {
        public void propertyChange(PCE e) {
            <tenir compte du changement
             de valeur de la propriété>
        }
    }
);
...
monBean.setP(v);
```

Propriété contrainte

VCS = VetoableChangeSupport
VCL = VetoableChangeListener
PVE = PropertyVetoException

- Propriété contrainte =
  - propriété liée → observable avec des *PCL*
  - modifications annulables → à l'aide de *VCL*
- Principe de fonctionnement en 2 temps :



Fonctionnement des *VCS*

(principe du) code de *VCS*.fireVetoableChange(p, ov, nv) :

```
if (ov != null && nv != null && ov.equals(nv)) {
    return;
}
PCE e = new PCE(source, p, ov, nv);
for (VCL vcl : <tousLesVCL>) {
    try{
        vcl.vetoableChange(e);
    } catch (PVE exc1) {
        PCE e2 = new PCE(source, p, nv, ov);
        for (VCL vcl2 : <tousLesVCL>) {
            try {
                vcl2.vetoableChange(e2);
            } catch (PVE exc2) {
                // cette fois on ignore les PVE
            }
        }
        throw exc1;
    }
}
```

rollback

seule sémantique possible :
« OUI le changement est acceptable
ou NON le changement n'est pas acceptable »

Attention au *rollback* !
avant → `vetoableChange(e)`
pendant → `vetoableChange(e2)`

Beug si *rollback* !
tous les *VCL* subissent `vetoableChange(e2)` !

... sera fixé dans le SDK 7b38 :
seuls les *VCL* s'étant **déjà exprimés** subissent le *rollback*

Règle méthodologique (Java 6) pour coder `vetoableChange`

```
SI changement non acceptable ALORS
    lever une PVE (et c'est tout !)
SINON
    ne rien faire
FINSI
```

Du bon usage des *PCL* et *VCL*

```
monBean.addVCL(PROP_P,
    new VCL() {
        public void vetoableChange(PCE e) throws PVE {
            X v = (X) e.getNewValue();
            if (!accept(v)) {
                throw new PVE(<message>, e);
            }
        }
        private boolean accept(X v) {
            // retourne true ssi v est acceptable
        }
    }
);
...
monBean.addPCL(PROP_P,
    new PCL() {
        public void propertyChange(PCE e) {
            <traitement adapté à 'v acceptée'>
        }
    }
);
...
try {
    monBean.setP(v);
} catch (PVE e) {
    <traitement adapté à 'v refusée'>
}
```

Rôle du *VCL* :

- si changement refusé : lever *PVE*
- si changement accepté : NE RIEN FAIRE
- si *rollback* : ne rien faire ou lever *PVE*

mais elle sera ignorée !

Rôle du *PCL* :

- adapter l'environnement au changement de valeur

Rôle de la capture :

- adapter l'environnement au refus de valeur

```
public class MonBean {
    public static final String PROP_P = "p";
    private X p;
    private final PCS pcs;
    private final VCS vcs;
    public void setP(X value) throws PVE {
        X oldValue = p;
        vcs.fireVetoableChange(PROP_P, oldValue, value);
        p = value;
        pcs.firePropertyChange(PROP_P, oldValue, value);
    }
    ...
}
```

Problème : notifications identiques

- Exemple :
  - propriété liée `step` (R/W, `Direction`)
  - comment notifier « deux pas vers le nord » ?
    - `walker.setStep(NORTH);`
      `walker.setStep(NORTH);`

ajouter un *PCL*

problème...

- Solution :
  - notifications forcées (désactivation de l'optimisation)
    - `pcs.firePropertyChange(PROP_STEP, null, v);`
  - *idem* pour les notifications de veto
    - `vcs.fireVetoableChange(PROP_STEP, null, v);`