

Université Montpellier-II - Master Info Pro -

Parcours GL

Réutilisation et Composants

Partie III - Composants Assemblables - l'exemple
des **Java-Beans**

Notes de cours

Christophe Dony

1 Définitions

Les composants du type *JavaBeans* sont des éléments logiciels qui peuvent être paramétrés et assemblés (de façon visuelle et interactive) pour former des composites ou des applications.

Un composant java-beans sur étagère est soit un objet sérialisé pouvant être cloné soit une classe pouvant être instanciée.

Un composant peut être un objet simple, graphique ou non, ou plus complexe (accès à une base de données, calendrier, ...) voire représenter une application (feuille de calcul, visualiseurs d'équations, grapheurs, ...).

2 Modèle de composant et d'assemblage

Modèle de composant : les caractéristiques qui font d'un objet Java un composant JavaBeans sont :

- de posséder des propriétés nommées normées,
- de posséder des éditeurs graphiques pour le paramétrage interactif de ses propriétés,
- de posséder une version affichable si le composant est *visible*,
- de posséder des descripteurs.

Un composant “visible” est un composant qui a une représentation graphique dans l'interface utilisateur de l'application dans laquelle il est utilisé. Un composant “non visible” est un simple objet métier.

Modèle d'assemblage : l'assemblage des composants `java-beans` est basé sur :

- un modèle de communication de type **écouteur/écouté** également nommé **publication/souscription** décrit par le schéma de conception **Observer**)
- l'adaptation automatique de l'écouteur à l'écouté selon le protocole défini par le schéma de conception **Adapter**.

3 Le schéma comportemental : “Observateur”

3.1 But

Faire qu’un objet devienne un observateur d’un autre afin qu’à chaque fois que l’observé est modifié, l’observateur soit prévenu.

Exemple d’application :

- Abonnements, toutes formes de “publish/subscribe”,
- connexion non anticipée de composants
- IHM (MVC).

3.2 Principe général de la solution

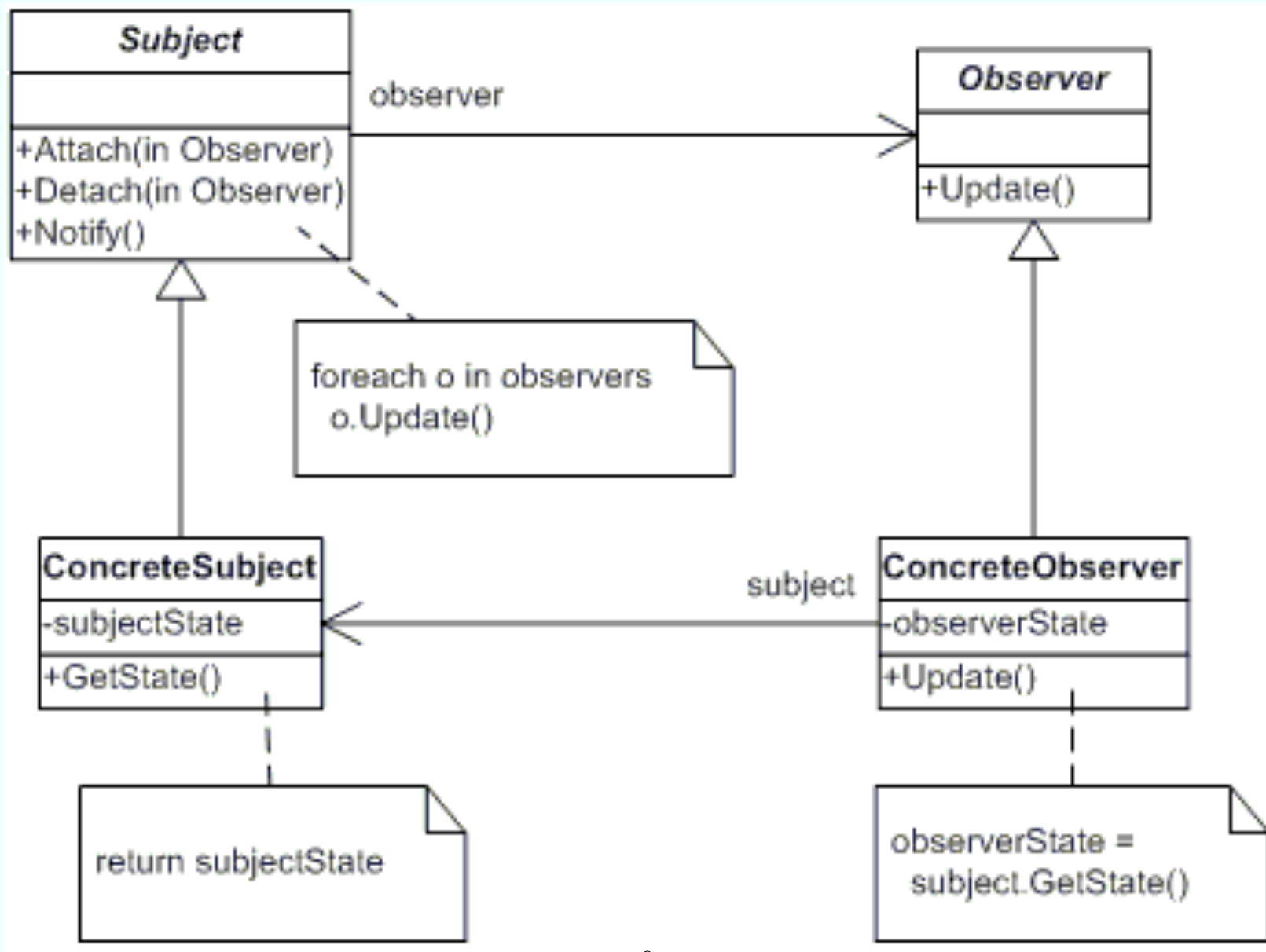


FIG. 1 – Observer : Collaboration “Observateurs-Observé”

3.3 Implantation

Gestion de liste d'écouteurs par l'observé.

Spécification par l'observé des points d'écoute.

Information des observateurs par envoi de message où d'évènement (parfois une émission d'évènement est implantée comme un envoi de message).

3.4 Une implantation exemple du framework MVC

MVC : Schéma de conception pour les application avec interface graphique.

Framework pour la réalisation d'applications graphiques permettant de :

- Découpler un objet de son interfaçage graphique (sa vue)
- Découpler le contrôle (souris, clavier) de l'affichage
- d'avoir plusieurs vues pour un même objet.

3.5 Architecture

Trois hiérarchies de classes : Les modèles, les vues et les contrôleurs.

Toute classe métier est sous-classe de la classe `Model`.

Chaque contrôleur connaît son modèle.

Chaque vue connaît son contrôleur et son modèle.

Le modèle ne connaît personne, il est observé par la vue et par le contrôleur.

3.6 Modèle d'interaction

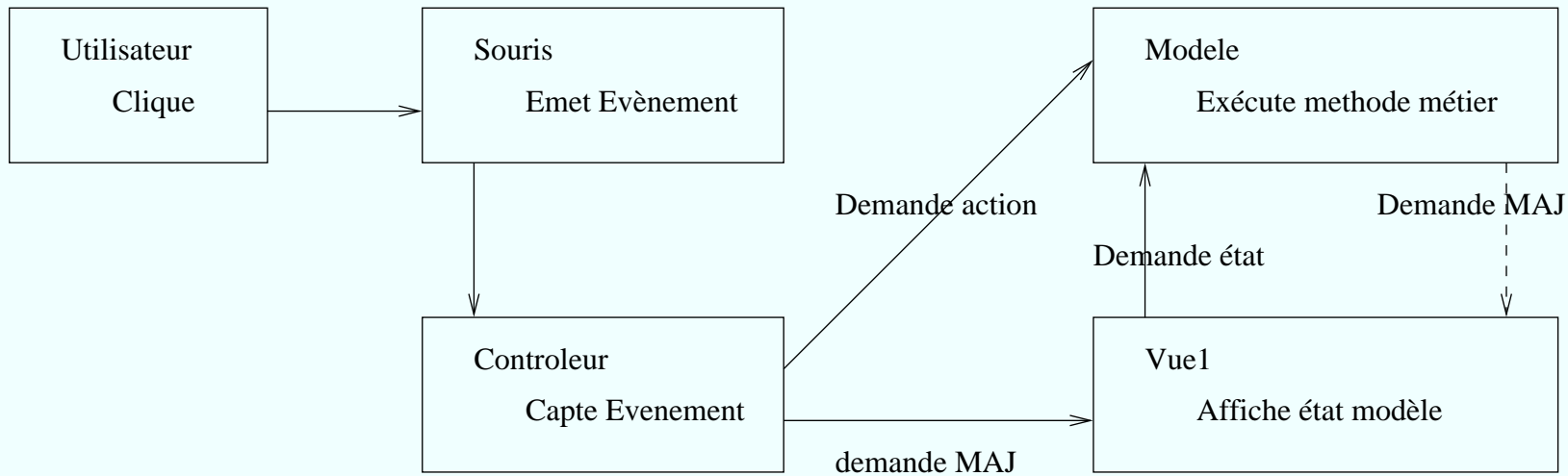


FIG. 2 – Diagramme d'interaction entre modèles, vues et contrôleurs MVC

3.7 Implémentation (squelette) inspirée de l'implantation originelle (Smalltalk)

3.7.1 Gérer les dépendances “observateur-observé”

Classe du framework

```
public class MV-Association{
    //utilise un dictionnaire pour stocker les couples model-controller
    static Dictionary MVDictionary = new Hashtable();

    //permet d'associer un modèle à une vue
    public static void add(Model m, View v) { ... }

    // rend la collection de vues associées à un modèle
    public static Collection<View> getViews(Model m) { ... }
```

3.7.2 Les modèles

Classe du framework

```
public class Model{  
  
    public void changed(Object how){  
        Iterator i = MV-Association.getViews(this).iterator();  
        while (i.hasNext())  
            (i.next()).update(how);}  
}
```

Classe de l'application

```
public class Compteur extends Model{
    protected int valeur = 0;

    protected changerValeur(i){
        valeur = valeur + i;
        this.changed("valeur");}

    public int getValeur(){return valeur;}
    public void incrémenter(){this.changerValeur(1);}
    public void décrémenter(){this.changerValeur(-1);}
```

3.7.3 Les vues

Classe du framework

```
public class View{
    Controller cont;
    Model model;

    public View(Model m, Controller c){
        model = m;
        cont = c;
        MVAssociation.add(this, m);

    public abstract void update (Object how);
    public void open(){...}
    public void redisplay(){...}

}
```

Classe de l'application

```
public class CompteurView extends View{  
    ...  
    private JLabel l;  
    ...  
  
    public void update(Object how){  
        l = new JLabel(String.valueOf(m.getValeur()), JLabel.CENTER);  
        this.redisplay();  
    }  
}
```

3.7.4 Les contrôleurs

Classe du framework

```
public abstract class controller implements ActionListener{
    Model m;

    public Controller(Model m){
        model = m;
    }
}
```

Classe de l'application

```
public class CompteurController extends controller

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand() == "incr") {m.incrémenter();}
        if (e.getActionCommand() == "decr") {m.decrémenter();}}
}
```

3.7.5 Lancement de l'application

```
Compteur m = new Compteur();  
CompteurView v = new CompteurView(m, new CompteurController(m));  
v.open();
```

4 Outils d'assemblage

Un certain nombre d'environnements (Visual Age, Delphi, Visual Basic, Eclipse, NetBeans) intègrent aujourd'hui des fenêtres spécialisées pour la programmation visuelle à base de composants similaires aux JavaBeans.

BDK : *bean development kit*, API de base de développement et d'assemblage de composants *Beans*

beanbox, netbeans : environnements d'assemblage

5 Exemple de création d'une application avec la beanbox

Un exemple avec deux écoutés, un écouteur et deux adapteurs.

5.1 La génération automatique d'adapteurs

La figure 3 montre l'application de *Adapter* à ce problème.

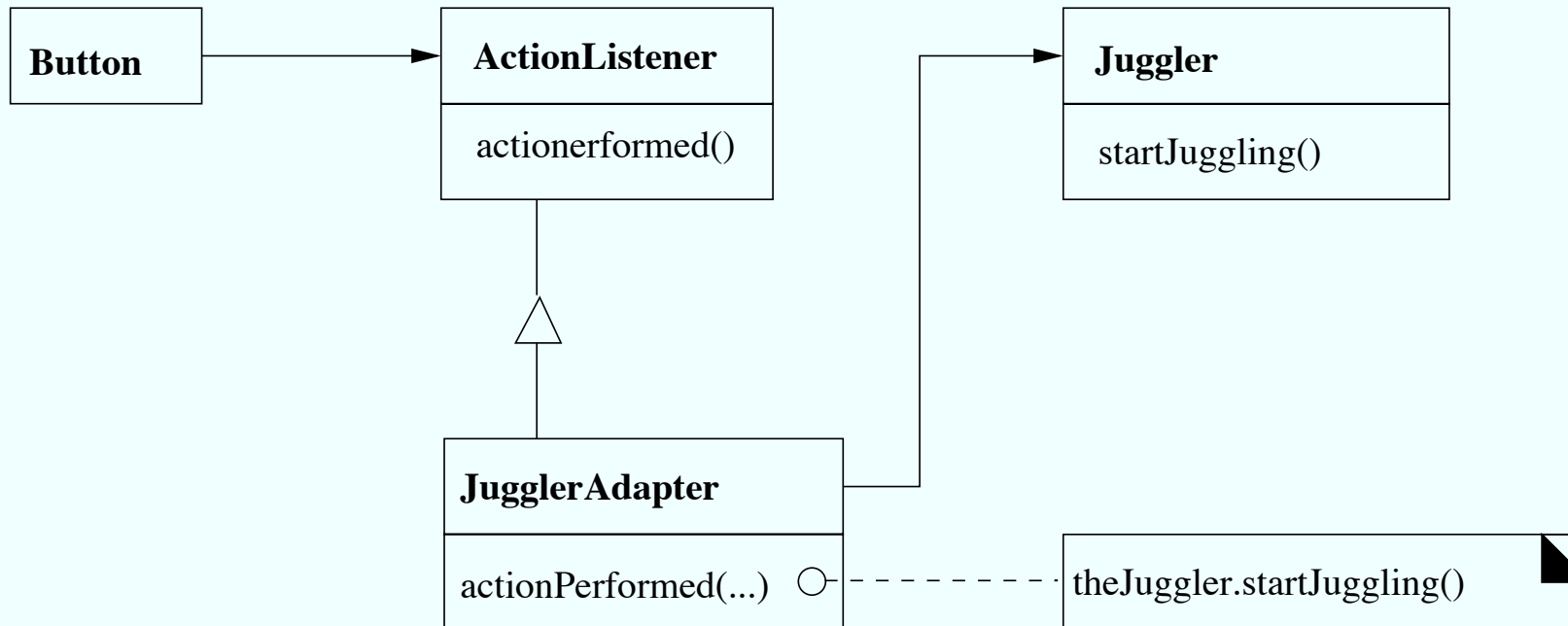


FIG. 3 – Adaptation par composition

Le texte suivant montre le code automatiquement généré^a pour implanter ce schéma de conception.

```
more ___Hookup_16add0c757.java
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.juggler.Juggler;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_16add0c757 implements java.awt.event.ActionList
ener, java.io.Serializable {
    private sunw.demo.juggler.Juggler target;
    public void setTarget(sunw.demo.juggler.Juggler t) {
        target = t;    }
    public void actionPerformed(java.awt.event.ActionEvent arg0) {
        target.startJuggling(arg0);    }
}
```

^aVoir répertoire \$BINDIR/beanbox/tmp/sunw/beanbox

6 Le cycle de vie d'un composant par l'exemple

Réalisation et mise sur étagère d'un premier composant (`SimplestBean`) visuel^a.

^aOn verra plus loin des composants non visuels (*invisible beans*).

6.1 Fabrication

Un composant *JavaBeans* visuel est décrit par une classe héritant d'une classe décrivant des objets graphique (`Canvas` par exemple).

```
import java.awt;

public class SimplestBean extends Canvas {
    public SimplestBean(){
        this.setBackground(Color.red);
    }

    public Dimension getMinimumSize(){
        return new Dimension(50,50);
    } }
}
```

6.2 Archivage

Ceci est évidemment automatisable (makefile). Effectuons ici les opérations à la main.

1. Finalisation du code. Dans notre exemple, compiler le source.
2. Description du composant.

Création d'un fichier appelé *manifeste* décrivant le composant.

```
cat > SimplestBean.mf
Name: SimplestBean.class
Java-Bean: True
```

3. **Archivage.**

Les paquets sont usuellement nommés “archives” : avec les beans, un fichier *.jar*.

```
jar -cfm simplestBean.jar simplestBean.mf SimplestBean.class
```

4. **Rangement.**

L'emplacement d'archivage est propre à chaque outil.

```
cp simplestBean.jar /home/dony/obj/java/beans/jars
```

6.3 Utilisation : sélection, paramétrage, assemblage et déploiement du résultat

1. Sélection d'un composant.

Avec le couple *BDK-Beanbox*, tout composant correctement archivé apparaît sur étagère (*Toolbox*), peut être sélectionné puis déposé sur le plan d'assemblage.

Le composant sur étagère est soit copié (objet sérialisé) soit instancié (classe). Les valeurs des propriétés de l'instance sont fixées par les règles usuelles (valeurs par défaut, constructeurs).

2. Paramétrage.

Les *propriétés* du composant résultant sont éditables (cf. fig. 4).



FIG. 4 – Intégration d'un nouveau composant

3. Déploiement.

Le déploiement est le placement du composant dans une application exécutable. Le déploiement est automatisé par la beanbox.

7 Aparté : Serialisation en Java

La sérialisation est utile quand l'initialisation d'un composant est longue ou complexe ou lorsque l'on souhaite créer plusieurs composants à partir de la même classe.

Le service de sérialisation de Java permet de sauvegarder des objets (tous les attributs non “static” et non “transcient”) dans des fichiers, sans SGBD, et de les recréer ultérieurement à partir de cette sauvegarde. Il utilise des flots de type `ObjectStream`.

Le service de serialisation est défini par l'interface *serialisable*.

Exemple

Création, initialisation serialisation puis archivage de deux composants instances de la classe Button.

```
import java.applet.*;
import java.awt.*;
import java.io.*;

public class SerialButton{

    static void serialize(Object o, String filename){
        try{
            FileOutputStream f = new FileOutputStream(filename);
            ObjectOutputStream s = new ObjectOutputStream(f);
            s.writeObject(o);
            s.flush(); }
        catch (Exception e) { System.out.println(e); }
    }
}
```

instantiation et serialisation :

```
public static void main(String[] args){
    // create instances of Button
    Button b1 = new Button();
    b1.setFont(new Font("System", Font.BOLD, 36));
    b1.setLabel("Serial Button 1");

    Button b2 = new Button();
    b2.setFont(new Font("System", Font.BOLD, 14));
    b2.setLabel("Serial Button 2");

    serialize(b1,"SerialButton1.ser");
    serialize(b2,"SerialButton2.ser"); } }
```

Déclaration des composants - fichier “manifest”

Name: SerialButton1.ser

Java-Bean: True

Name: SerialButton2.ser

Java-Bean: True

Archivage des deux instances pour mise sur étagère dans la **beanbox**.

```
jar -cfm SerialButtons.jar
```

```
    SerialButton.mf
```

```
    SerialButton1.ser SerialButton2.ser
```

```
cp SerialButtons.jar /home/dony/obj/java/beans/jars
```



FIG. 5 – Composants sérialisés sur étagère

8 Les propriétés

Propriété : caractéristique nommée et paramétrable d'un composant.

Avec les **beans**, une propriété est implantée par un attribut.

8.1 Les propriétés simples

Propriété simple : propriété mono-valuée.

Conventions de nommage

Afin d'être identifiable automatiquement par les environnements d'assemblage, les noms des accesseurs des propriétés sont contraints par les règles suivantes :

lecture	<PType> get<PName>()
écriture	void set<PName>(<PType> value)
test (si booléenne)	boolean is<PName>

8.2 Les Propriétés indexées

Propriété indexée : propriété multi-valuée avec valeurs accessibles via un index (collection indexée).

Conventions de nommage

lecture	<code><PType>[] get<PName>()</code>
écriture	<code>void set<PName>(<PType>[] value)</code>
lecture d'un élément	<code><PType> get<PName>(int index)</code>
écriture d'un élément	<code>void set<PName>(int index, <PType>value)</code>

9 Exemple de réalisation d'un composant possédant une propriété

Cette section montre la création d'un compteur graphique doté d'une propriété nommée *Compteur* implémentée avec une variable d'instance nommée *cpt*.

9.1 Définition de GraphicCptBean

```
import java.awt.*;
import java.io.Serializable;

public class GraphicCptBean extends Panel implements Serializable{
    private int cpt;
    private TextField textfield;

    public GraphicCptBean() { //Constructor
        cpt = 0;
        setLayout(new FlowLayout(FlowLayout.CENTER, 30, 10));
        textfield = new TextField(this.newLabel());
    }
}
```

```
        add(textfield); }

public int getCompteur(){ return cpt; }

public void setCompteur(int newValue) {
    cpt = newValue;
    textfield.setText(this.newLabel());
    this.repaint();}

public void incr(){
    setCompteur(getCompteur() + 1);}

public void decr(){
    setCompteur(getCompteur() - 1);}

private String newLabel(){
    return(String.valueOf(getCompteur()));} }
```

9.2 Utilisation

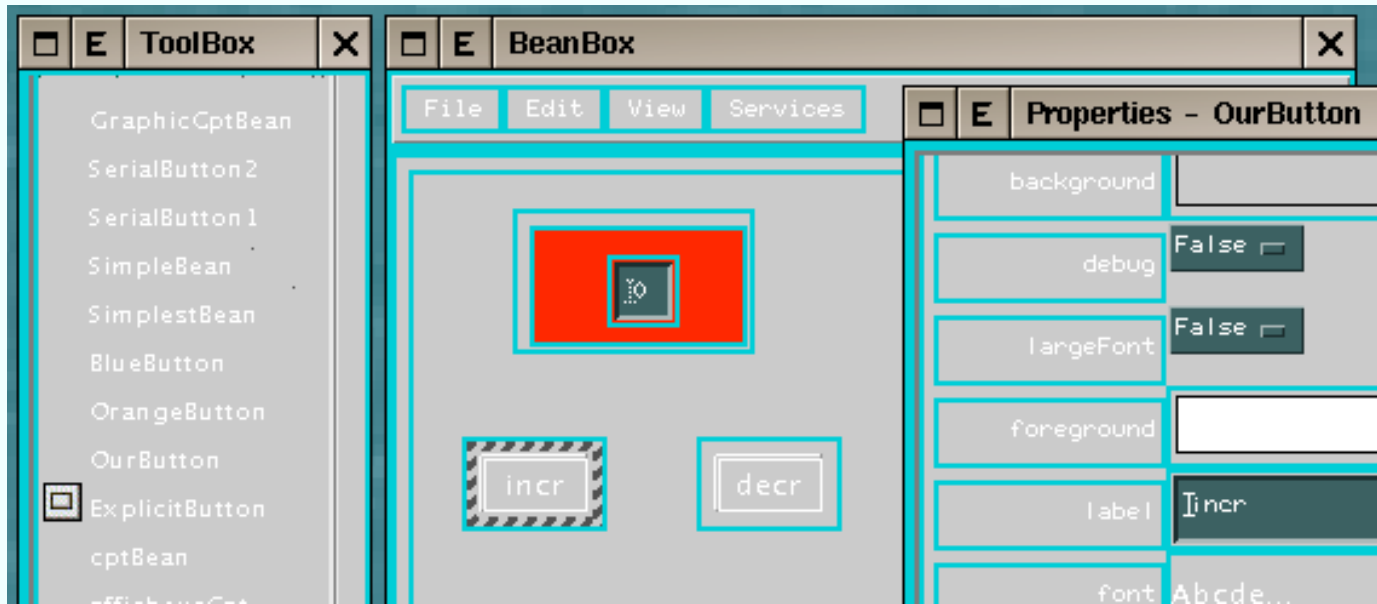


FIG. 6 – Edition d'un bouton

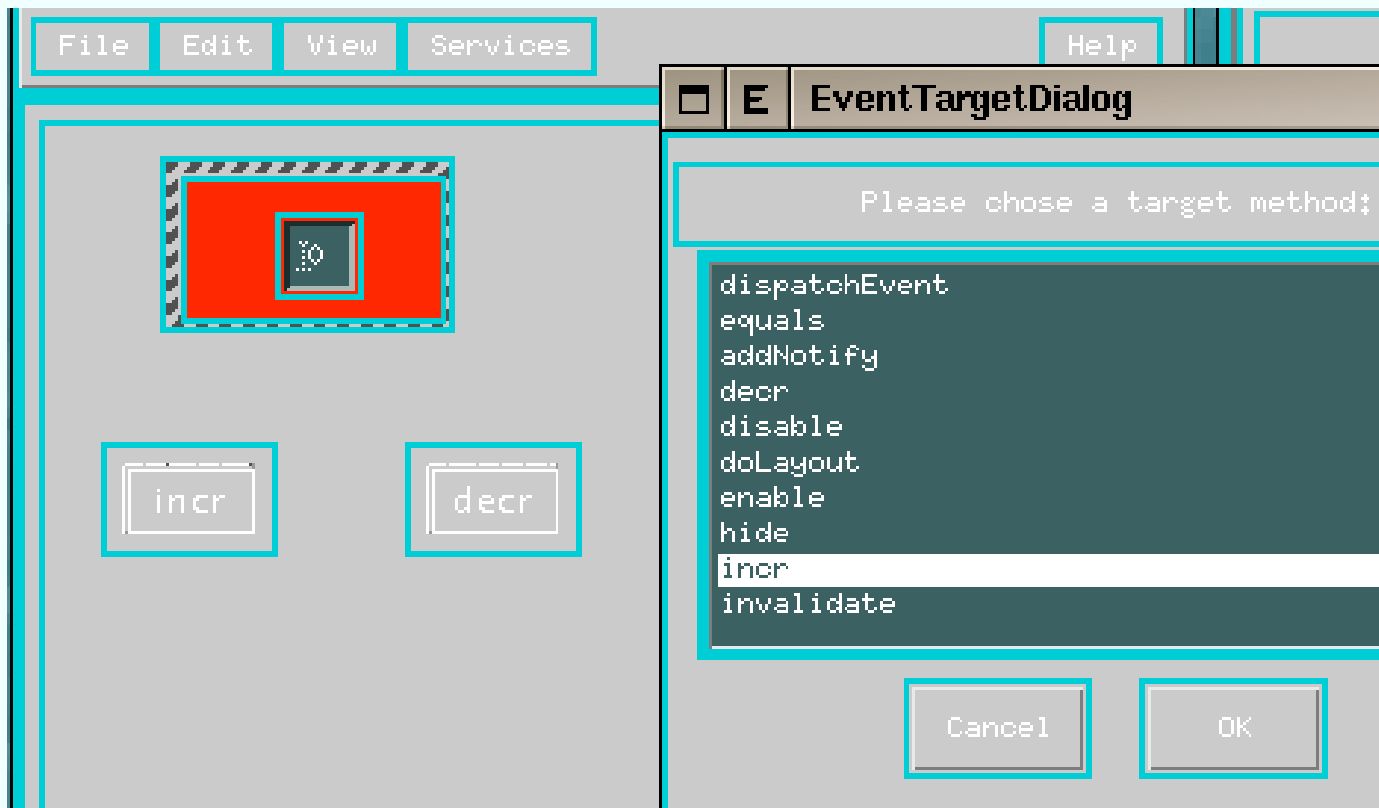


FIG. 7 – Liaison d'un bouton au compteur

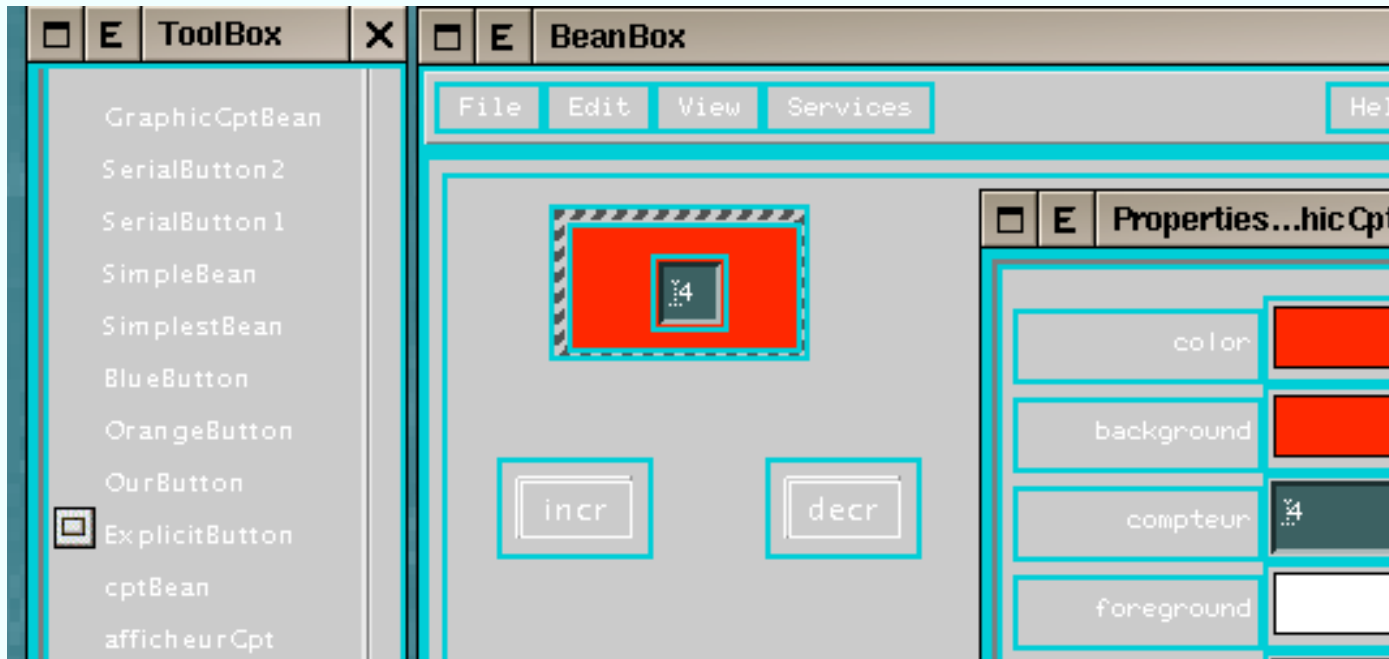


FIG. 8 – Utilisation du compteur graphique

10 Écoutés versus Ecouteurs

Les JavaBeans sont des composants que l'on assemble via un protocole de type “écouteur-écouté” ou “publish-subscribe”.

L'environnement JavaBeans génère automatiquement des adapteurs donc n'importe quel composant peut être un écouteur (le jongleur dans l'exemple no 1).

Les écoutés sont tous les objets produisant des résultats utiles à d'autres. Ils sont assemblables s'ils émettent des évènements pour publier les résultats qu'ils produisent.

Exemple : composant graphique tel un bouton Swing ou compasant métier fournissant des résultats comme un compteur.

11 Réalisation de Composants de type “écoutés”

11.1 Rappel, le protocole “écouteur/écouté” en Java

Soit un exemple Java d’un objet température émettant un événement *tempChangedEvent* à chaque changement de température (“Developing JAVA Beans, Robert Englander, O’Reilly”). Le protocole écouteur/écouté peut se résumer de la façon suivante.

1. Dans la classe de l’écouteur,
 - (a) s’enregister auprès de l’objet émetteur des évènements que l’on souhaite écouter :

```
emetteur.addTempChangeListener(this);
```

- (b) Déclarer que l’on est écouteur d’un certain type d’évènements en implantant la bonne interface :

```
public class MyClass implements TempChangeListener {
```

- (c) implanter la méthode à invoquer lorsque l’évènement est signalé.

```
public void tempChanged(TempChangedEvent e) {  
    ... }  
}
```

2. Dans la classe de l'écouté

- (a) être doté d'une collection permettant de stocker les références sur les écouteurs,
- (b) définir les méthodes permettant aux écouteurs de s'enregistrer et de se désabonner,
- (c) prévenir les écouteurs en émettant, via un envoi de message (`tempChanged`), l'évènement idoine.

Une solution moins lourde à base de simples envois de messages et sans "évènements" est mise en oeuvre dans l'environnement Smalltalk.

11.2 Propriétés liées

Définition

Propriété liée (*bound property*) : propriété d'un composant écouté dont le changement de valeur doit être propagé aux écouteurs.

Dans l'exemple *temperature-thermomètre* la propriété *currentTemperature* du composant *temperature* sera utilement définie comme une propriété liée.

Gestion des propriétés liées

L'API `javaBeans` (`java.beans.*`) propose des données et des protocoles prédéfinis pour gérer les propriétés liées.

- La classe `PropertyChangeEvent`

```
PropertyChangeEvent(  
    Object source,  
    String propertyName,  
    Object oldValue,  
    Object newValue)
```

- l'interface `PropertyChangeListener` définissant la méthode `propertyChanged(...)` utilisable pour alerter les écouteurs.
- La classe (`PropertyChangeSupport`) définissant une propriété liée capable d'avoir des écouteurs (méthode `addPropertyChangeListener`).

“This is a utility class that can be used by beans that support bound properties. You can use an instance of this class as a member field of your bean and

delegate various work to it. This class is serializable. When it is serialized it will save (and restore) any listeners that are themselves serializable”.

Cette classe peut s'utiliser de deux manière :

- par héritage^a de *PropertyChangeSupport* si la classe du composant à créer n'a pas d'autre surclasse.
- par composition et redirection.

12 Un Exemple plus complet avec NetBeans ...

Le composant précédant *GraphicCptBean* ne propose pas un découplage métier/interface.

Une solution plus générique à base de deux composants :

- Un visible (Afficheur), écouteur de tout évènement via adaptation

^aLa solution avec héritage ne fonctionne pas. Le constructeur de super-classe doit être invoqué avant que l'on puisse faire une référence à *this*. Il est donc impossible de passer *this* en paramètre au constructeur de la super-classe comme le réclame cette solution.

- Un invisible (Compteur), possédant une propriété liée (`value`) et émettant des évènements `PropertyChangeEvent`.

Un beans compteur - partie 1.

```
import java.io.Serializable;
import java.beans.*;

public class CounterBean implements Serializable {

    public static final String PROP2 = "value";

    private Integer value = new Integer(0);
    private PropertyChangeSupport propertySupport;

    public CounterBean() {
        propertySupport = new PropertyChangeSupport(this);
    }

    public Integer getValue() { return value;}

    public void setValue(Integer v) {
        Integer oldValue = value;
```

```
this.value = v;  
propertySupport.firePropertyChange(PROP2, oldValue, value);  
}
```

Un beans compteur - partie 2

```
public void incr(){
    this.setValue(this.getValue()+1);    }

public void decr(){
    this.setValue(this.getValue()-1);    }

public void addPropertyChangeListener(PropertyChangeListener listener) {
    propertySupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener) {
    propertySupport.removePropertyChangeListener(listener);
}
}
```

Un beans afficheur d'entiers.

```
public class AfficheurBean extends Panel{

    /** Creates a new instance of affichInt */
    public AfficheurBean() {
        super();
        setLayout(new FlowLayout(FlowLayout.CENTER, 30, 10));
        setSize(60,40);
        setBackground(color);
        textfield = new TextField("0");
        add(textfield);
    }

    private TextField textfield;

    private Color color = Color.red;

    public void affiche(String s){
        textfield.setText(s); this.repaint();}
}
```

```

    public void affiche(Integer i){
        this.affiche(String.valueOf(i));}
}

```

Classe CounterApp, Adapteurs générés :

```

class CounterApp {
    private void initComponents() {

        counterBean1 = new counterApp.CounterBean();
        afficheurBean1 = new counterApp.AfficheurBean();
        jButton1 = new javax.swing.JButton();
        jButton2 = new javax.swing.JButton();

        counterBean1.addPropertyChangeListener(new java.beans.PropertyChangeListener() {
            public void propertyChange(java.beans.PropertyChangeEvent evt) {
                counterBean1PropertyChange(evt);            }            });

        jButton1.setText("incr");
        jButton1.addActionListener(new java.awt.event.ActionListener() {

```

```

        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jButton1ActionPerformed1(evt);          }      });
    }

    ...

    ...
private void jButton1ActionPerformed1(java.awt.event.ActionEvent evt) {
    counterBean1.incr();    }

private void jButton2ActionPerformed1(java.awt.event.ActionEvent evt) {
    counterBean1.decr();    }

private void counterBean1PropertyChange(java.beans.PropertyChangeEvent evt) {
    afficheurBean1.setText(counterBean1.getValue());    }

```

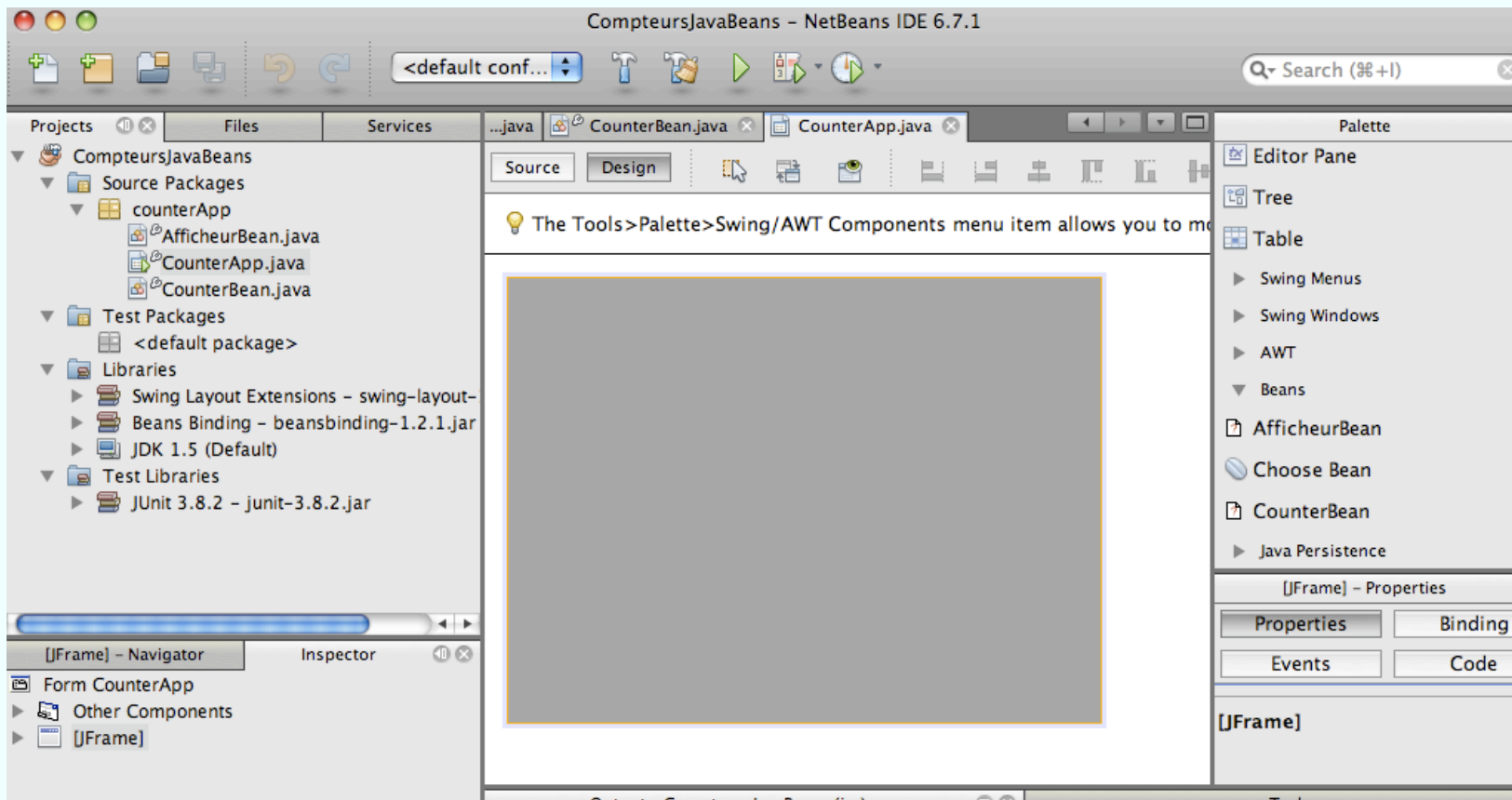


FIG. 9 – Nouveau projet et Palette

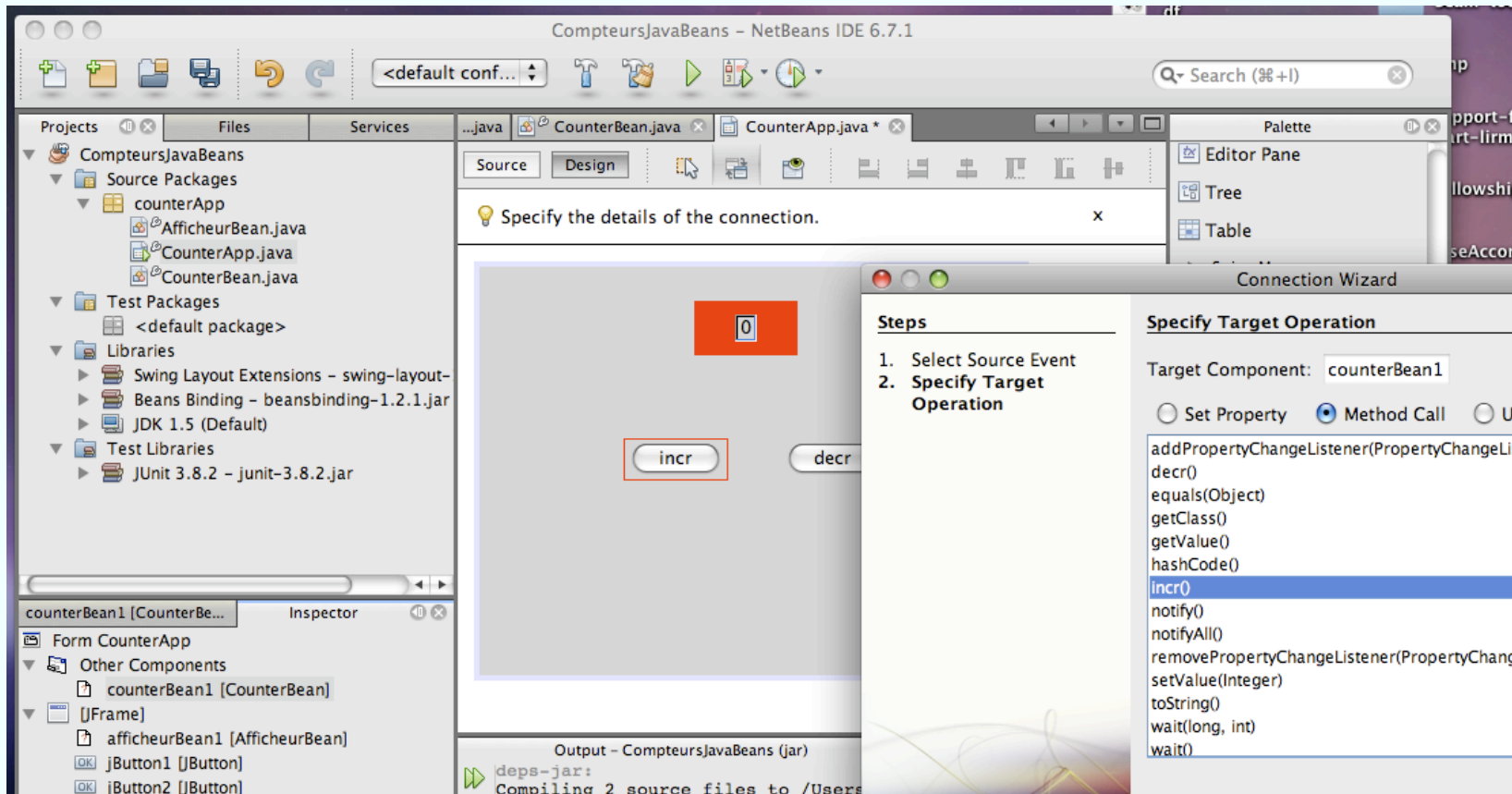


FIG. 10 – Connexion du bouton au compteur

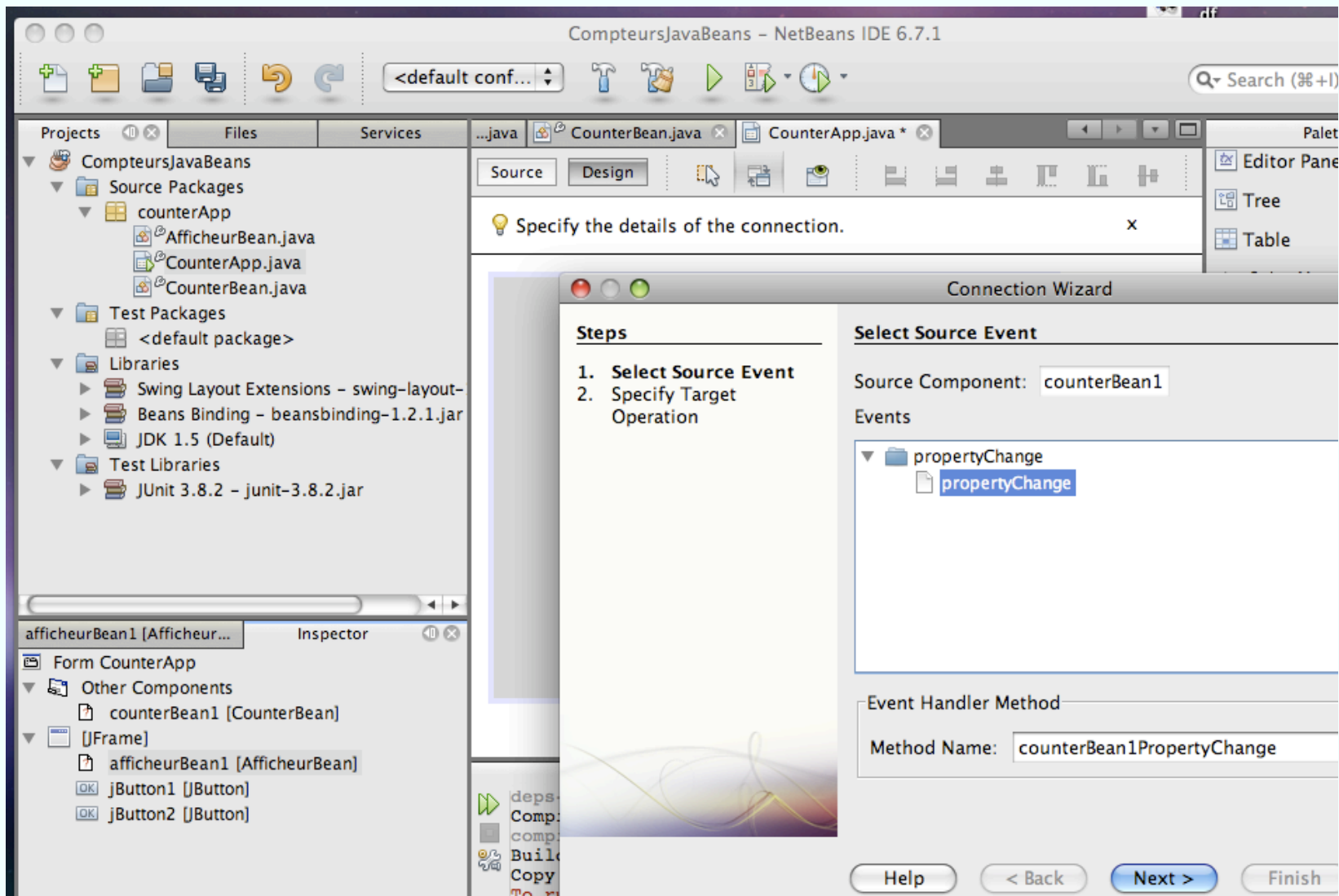


FIG. 11 – Connexion du compteur à l’afficheur - étape 1

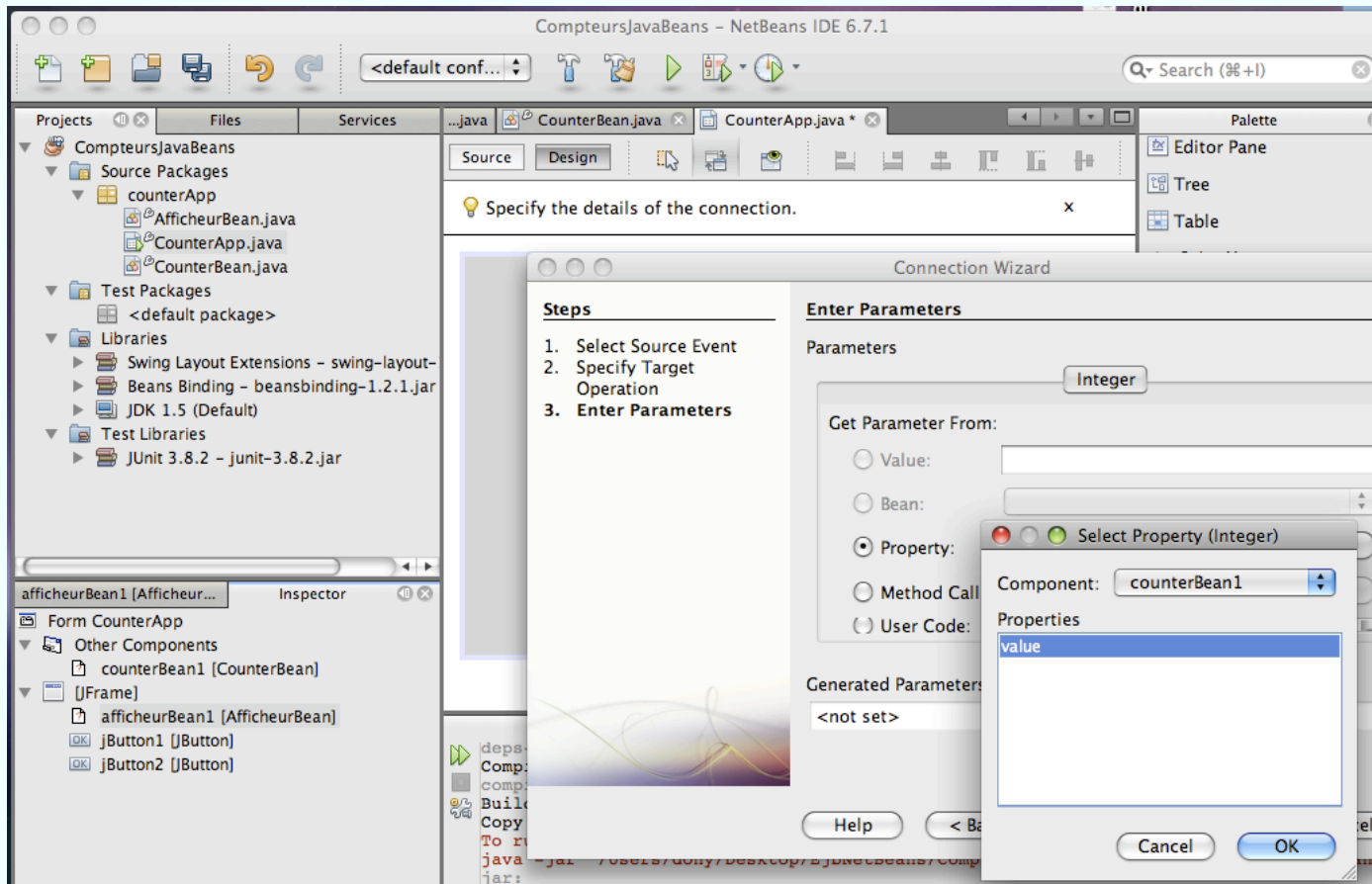


FIG. 12 – Connexion du compteur à l’afficheur - étape 2

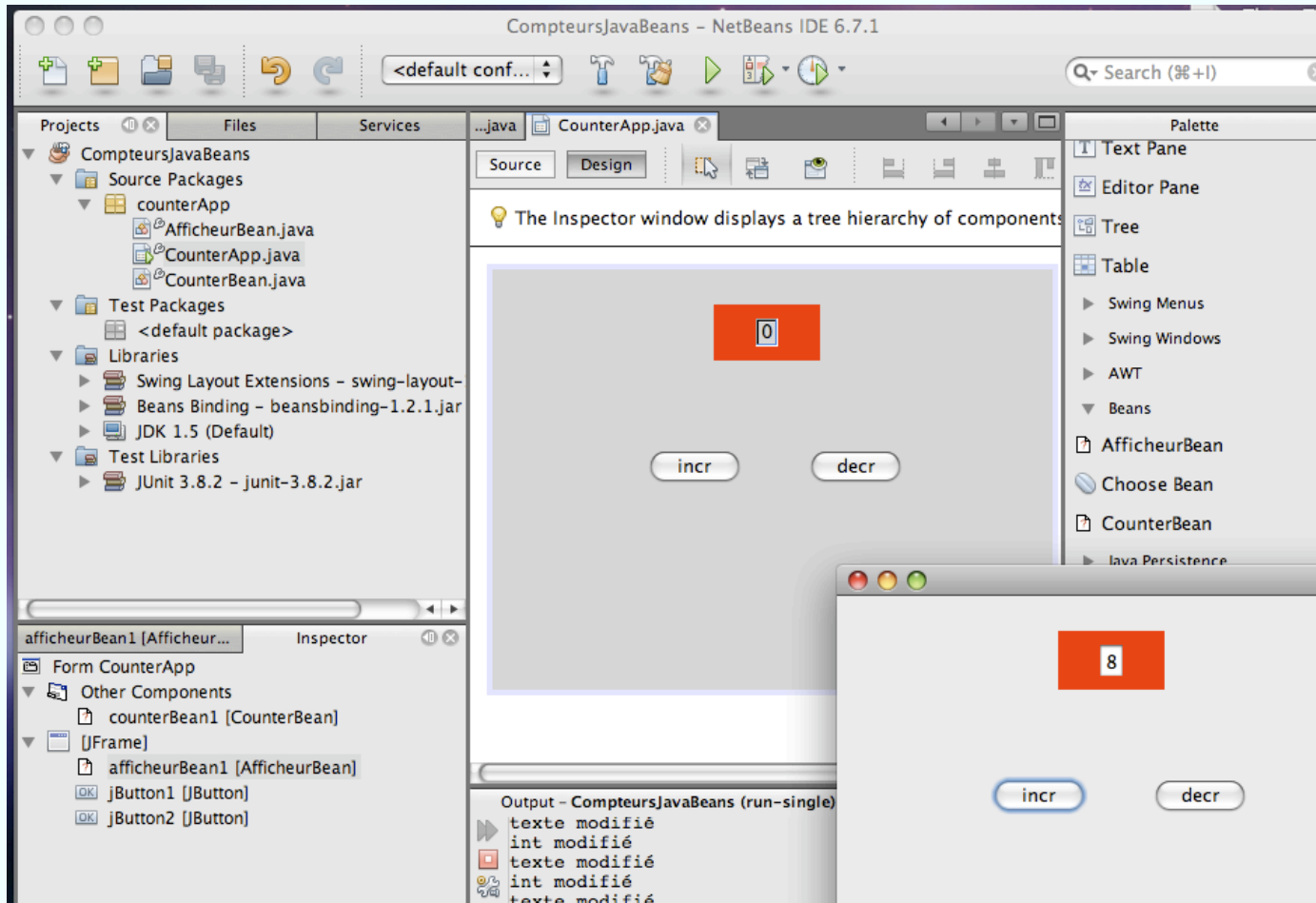


FIG. 13 – Execution

13 Autres supports à l'assemblage

“écoutateur/écouté”

13.1 Propriétés contraintes

Propriété contrainte : Propriété dont le propriétaire ainsi que les objets dûment enregistrés pour le faire, peuvent contrôler la modification à certaines valeur (droit de véto).

Les objets ayant le droit de véto sont les écoutateurs de l'évènement *VetoableEvent* (sic).

L'exemple suivant, tiré de la distribution du BDK1.1, montre ce qu'il est nécessaire d'écrire pour réaliser une propriété liée et une propriété contrainte.

Si un objet veut mettre un véto à la modification d'une propriété, il signale l'exception *PropertyVetoException* lorsqu'il reçoit la notification d'une demande de changement réalisée par l'émission de l'évènement *VetoableChange*.

Exemple. Un composant qui demande aux autres l'autorisation de modification d'un prix.

```
public class JellyBean extends Canvas {

    private VetoableChangeSupport vetos = new VetoableChangeSupport(this);
    private int ourPrice = 2;

    public synchronized int getPrice() {
        return ourPrice;    }

    public void setPrice(int newPrice) throws PropertyVetoException {
        int oldPrice = ourPrice;
        vetos.fireVetoableChange("price",
                                new Integer(oldPrice),
                                new Integer(newPrice));
        ourPrice = newPrice;    }

    public void addVetoableChangeListener(VetoableChangeListener l) {
        vetos.addVetoableChangeListener(l);    }

    public void removeVetoableChangeListener(VetoableChangeListener l) {
```

```
vetos.removeVetoableChangeListener(1);    }    }
```

14 Edition des propriétés des composants

14.1 Introspection, le package Reflect

Les environnements de développement de *JavaBeans* doivent être capables, à partir soit d'un composant sérialisé soit d'une classe compilée de retrouver les propriétés des composants.

Ils doivent pour cela utiliser le package *Reflect*.

Exemple d'utilisation de ce package.

```
import java.lang.Reflect;

public class TestReflect {

    public static void main (String[] args) throws NoSuchMethodException{
        GraphicCptBean g = new GraphicCptBean();

        Class gClass = g.getClass();
```

```

Class gSuperclass = gClass.getSuperclass();

Method gMeths[] = gClass.getDeclaredMethods();

Method getCompteur = gClass.getDeclaredMethod("getCompteur", null);

try {System.out.println(getCompteur.invoke(g, null));}
catch (Exception e) {}
}
}

```

Trace de l'exemple.

```

class GraphicCptBean
class java.awt.Panel
[Ljava.lang.reflect.Method;@5a2edaa1
0

```

15 La documentation des composants : la classe BeanInfo

Par défaut, lorsqu'un composant est déposé dans la fenêtre d'assemblage (*beanbox*), toutes ses propriétés structurelles définies avec les accesseurs *get* et *set*, tous les événements qu'il émet et toutes ses méthodes (y compris celles héritées) sont répertoriées par analyse du code (package *Reflect*).

Si l'on souhaite que seules quelques unes de ces propriétés soient recensées, l'API rend possible de définir pour tout composant une classe le décrivant.

Le nom de cette classe doit être celui du composant suivi de *BeanInfo*. Elle doit implanter l'interface *BeanInfo* :

```
public BeanDescriptor getBeanDescriptor();
public PropertyDescriptor[] getPropertyDescriptors()
public EventSetDescriptor[] getEventSetDescriptors()
public MethodDescriptor[] getMethodDescriptors()
```

Exemple : la classe *cptBeanBeanDescriptor* décrit le composant *cptBean*.

La description générale du composant.

```
import java.beans.*;

class cptBeanBeanDescriptor extends SimpleBeanInfo {

    public BeanDescriptor getBeanDescriptor() {
        BeanDescriptor bd = new BeanDescriptor(cptBean.class);
        bd.setDisplayName("Composant compteur");
        bd.setShortDescription("Peut être incrémenté et décrémenté");
        return bd;}
}
```

La description de ses propriétés, ici il n'y en a qu'une de nom *cpt*.

```
public PropertyDescriptor[] getPropertyDescriptors() {  
    try {PropertyDescriptor[] pds = {  
        new PropertyDescriptor("cpt", cptBean.class)};  
        return pds; }  
    catch(IntrospectionException e) {e.printStackTrace();}  
}
```

La description des évènements qu'il émet. Il n'y en a qu'un : *propertyChange*.

Pour chaque descripteur d'évènement, il faut indiquer : la classe qui l'émet, le nom de l'évènement, l'interface définissant les abonnés, le nom de la méthode que doivent implanter les abonnés.

```
public EventSetDescriptor[] getEventSetDescriptors() {
    try {EventSetDescriptor[] eds = {
        new EventSetDescriptor
            (cptBean.class,
            "propertyChange",
            PropertyChangeListener.class,
            "propertyChange") };
        return eds;}
    catch(IntrospectionException e) {e.printStackTrace();}
}
```

La description des méthodes qui pourront être sélectionnées dans l'environnement d'assemblage. Dans l'exemple, il y en a deux : *incr* et *decr*.

```
public MethodDescriptor[] getMethodDescriptors() {
    try {MethodDescriptor[] mds = {
        new MethodDescriptor(cptBean.class.getMethod("incr", null)),
        new MethodDescriptor(cptBean.class.getMethod("decr", null))};
        return mds; }
    catch(NoSuchMethodException e) {e.printStackTrace();}
}
```

QT4 Creator