Université Montpellier-II
UFR des Sciences - Département Informatique
Master Informatique - Génie Logiciel

Réutilisation et Composants.

Des objets aux composants : couplage faible et non anticipation, l'exemple des JavaBeans

Notes de cours

Christophe Dony

1 Définitions

Les composants du type *JavaBeans* sont des éléments logiciels disponibles “sur étagère” qui peuvent être paramétrés et assemblés (de façon interactive) pour former des composites (une application est un composite ultime) eux même éligibles au status de composant.

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

Un *java-beans* peut être un objet minimal ou un composite de tai(calendrier, ...) voire une application (feuille de calcul, visualiseurs d’équations, grapheurs, ...).

2 Modèle de composant et d'assemblage

Voir : `http://docs.oracle.com/javase/tutorial/javabeans/`.

Modèle : extension du modèle d'objet

les caractéristiques additionnelles qui transforment un objet `Java` en composant `JavaBeans` sont :

- de posséder des "propriétés" (simples, ou liées ou à véto)
- d'être capable d'introspection, (capacité à s'auto-décrire pour renseigner des éditeurs de propriétés),
- de pouvoir être assemblé à d'autres composant selon un modèle de couplage faible,
- de posséder une version affichable si le composant est *visible* ; il est alors utilisable dans une interface graphique d'application.

Modèle d'assemblage :

- assemblage en **couplage faible** de type **écouté/écouteur** (également nommé **publication/souscription** ou **Observé/Observateur**) décrit par le schéma de conception "**Observateur**".

- assemblage **non anticipé** côté selon le schéma de conception "**Adapteur**".

3 Utilisations

- Tout composant *Swing* est un *Java-Beans*

- Programmation Visuelle

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

- **beanbox**, **netbeans** : environnements d'assemblage

- base de nombreux framework de développement d'applications avec GUI, dont :

OpenJDK JavaFX - successeur de SWING - `https://wiki.openjdk.java.net/display/OpenJFX/Main`)

> *For many years, the Java programming language has used the JavaBeans component architecture to represent the property of an object. This model consists of both an API and a design pattern ; it is widely understood by Java application developers and development tools alike. This release introduces property support into JavaFX, support that is based on the proven JavaBeans model, but expanded and improved.*
>
> Oracle documentation : Using JavaFX Properties and Binding

4 Historique : exemple de création d'une application avec la beanbox

Un exemple avec l'environnement historique "beanbox" incluant deux écoutés (boutons), un écouteur (appli graphique "jongleur") : figures 1 à 7

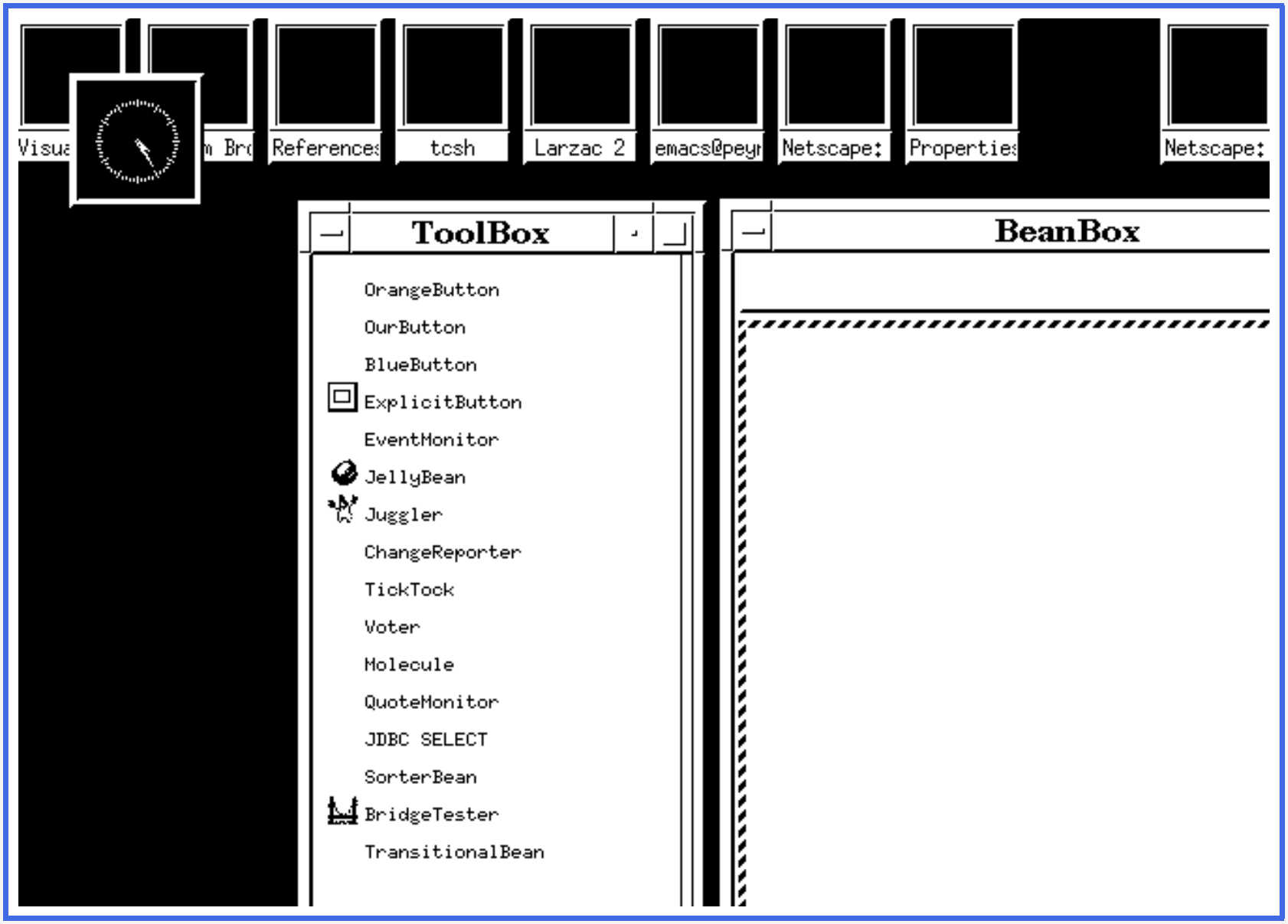

Figure (1) – *Nouveau projet et Palette*

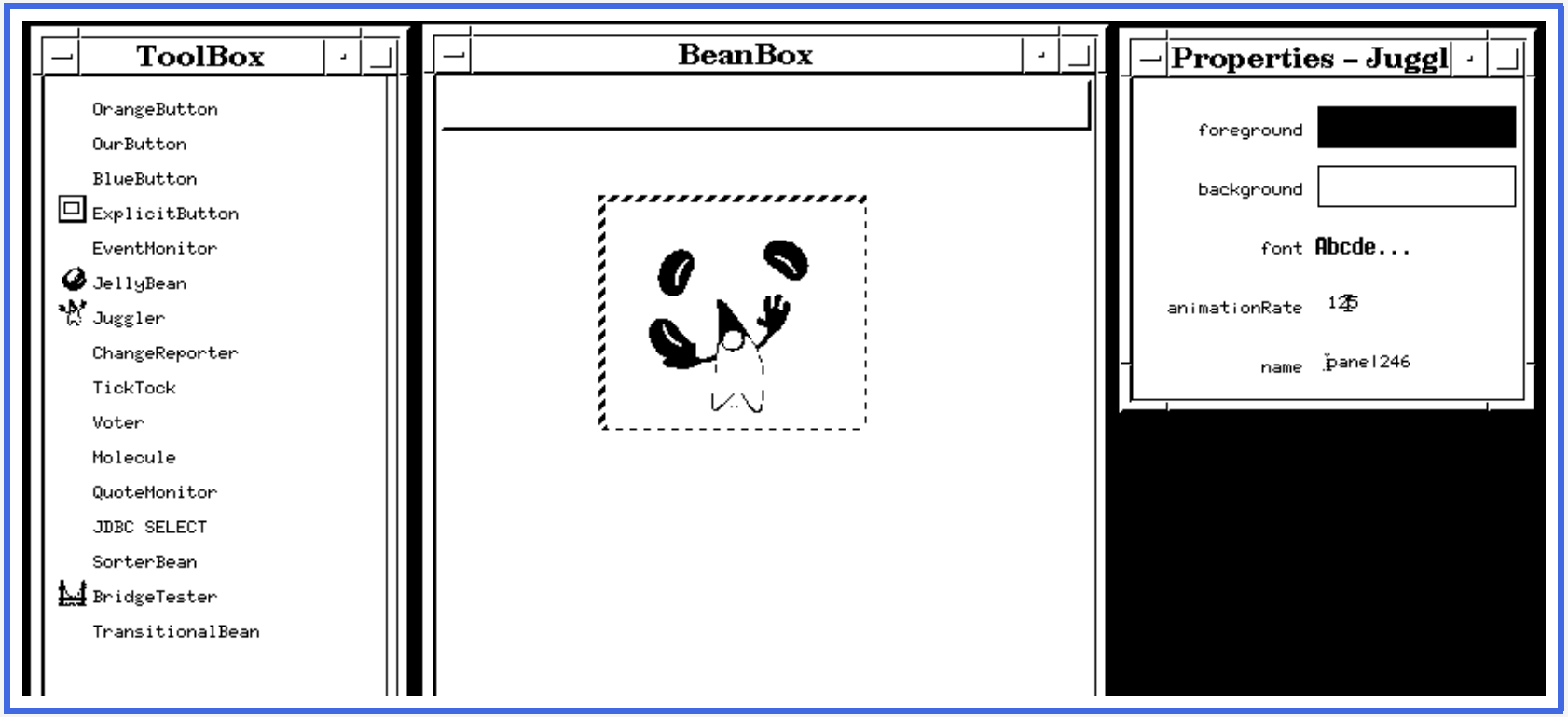

Figure (2) – *Dépose d'un composant, écouteur, graphique*

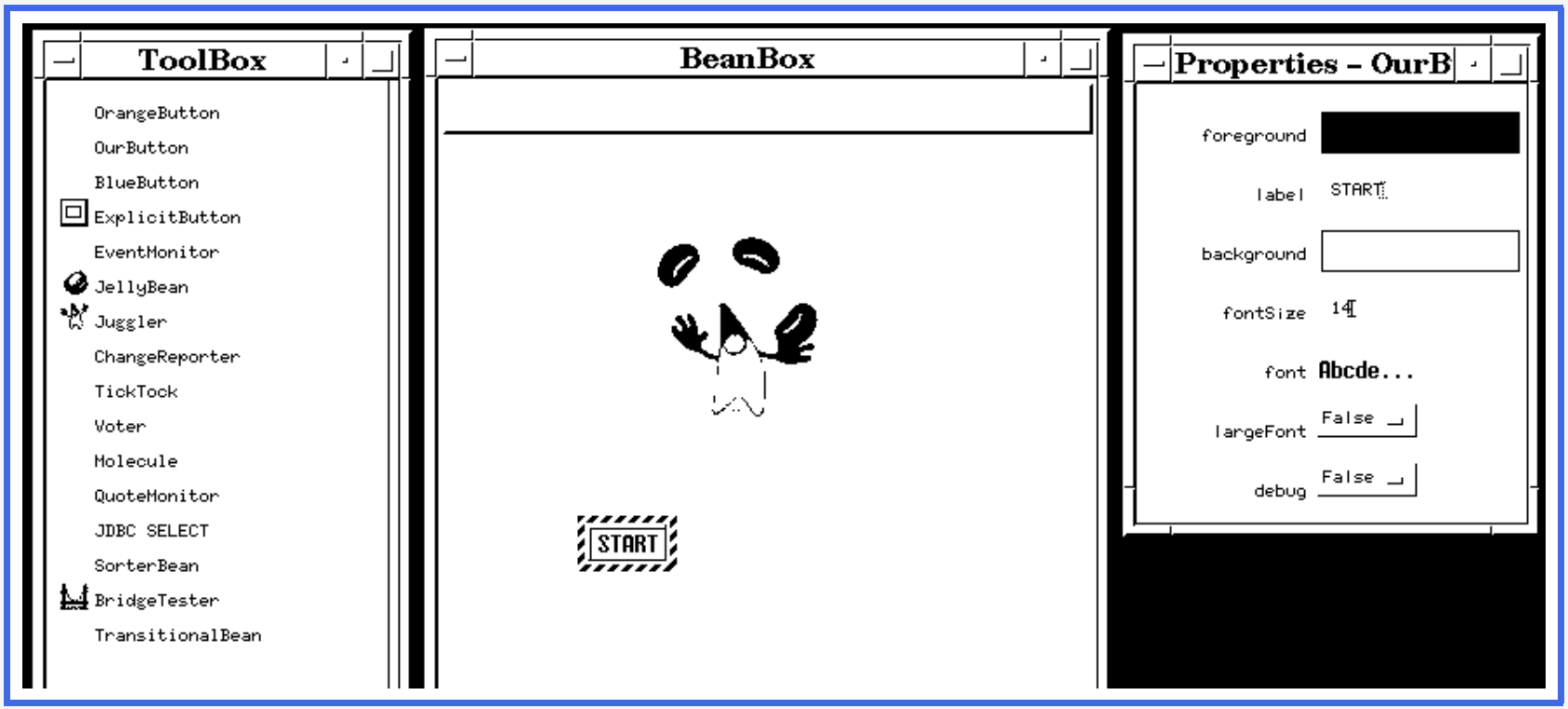

Figure (3) – *Dépose d'un composant, écouté, bouton*

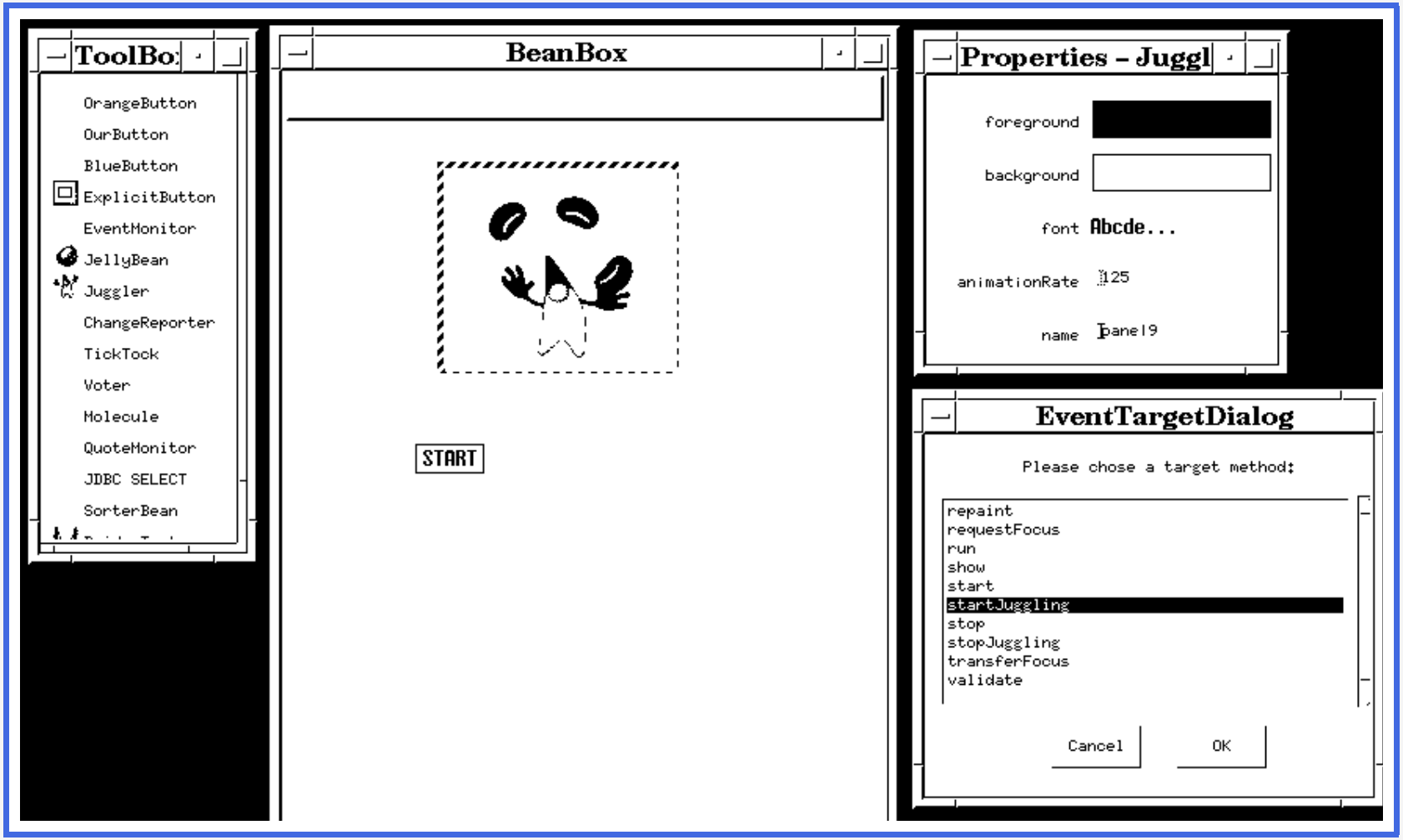

Figure (4) – *Connexion des deux précédents*

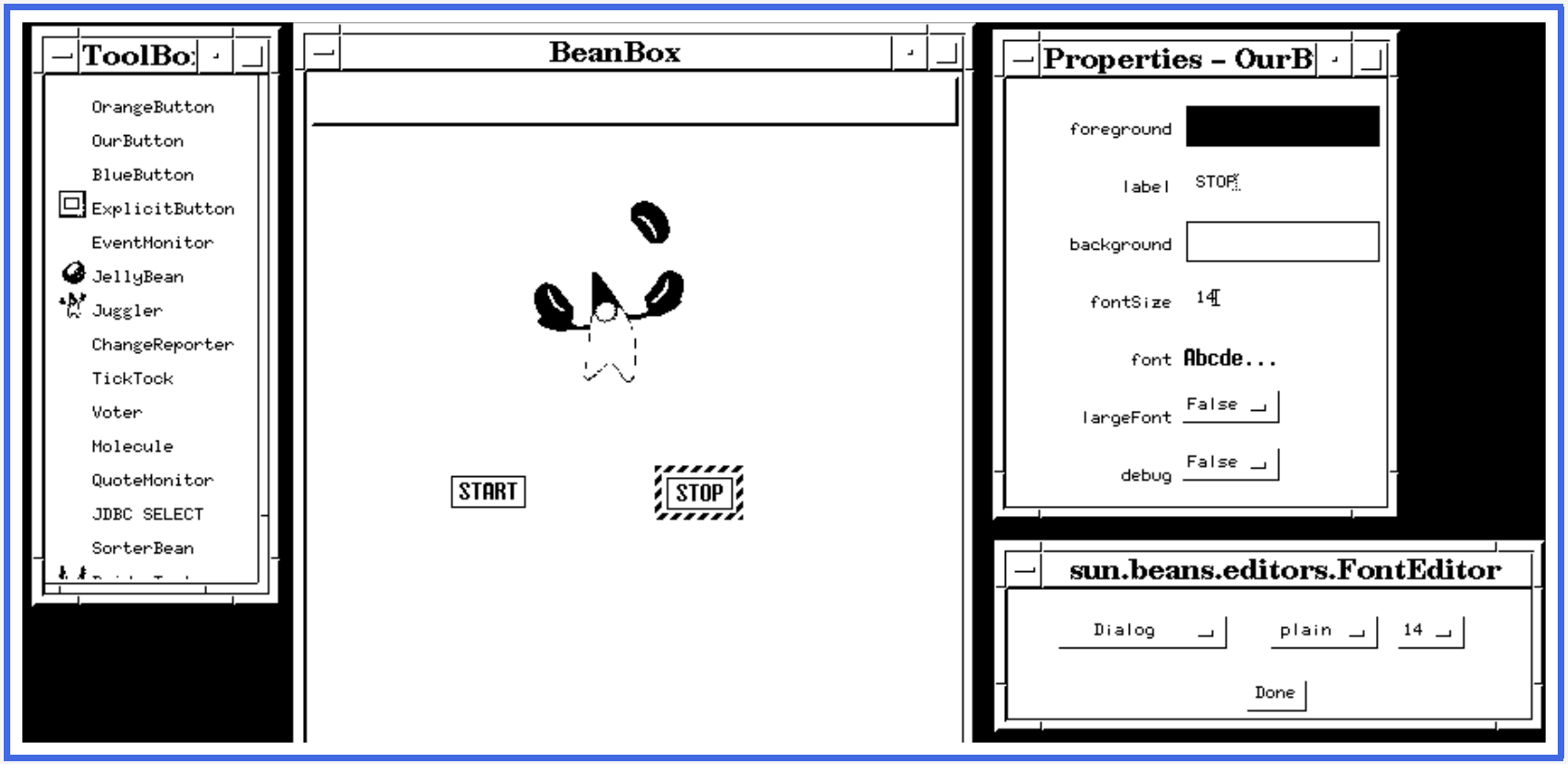

Figure (5) – *Dépose d'un second bouton et édition de sa propriété "label"*

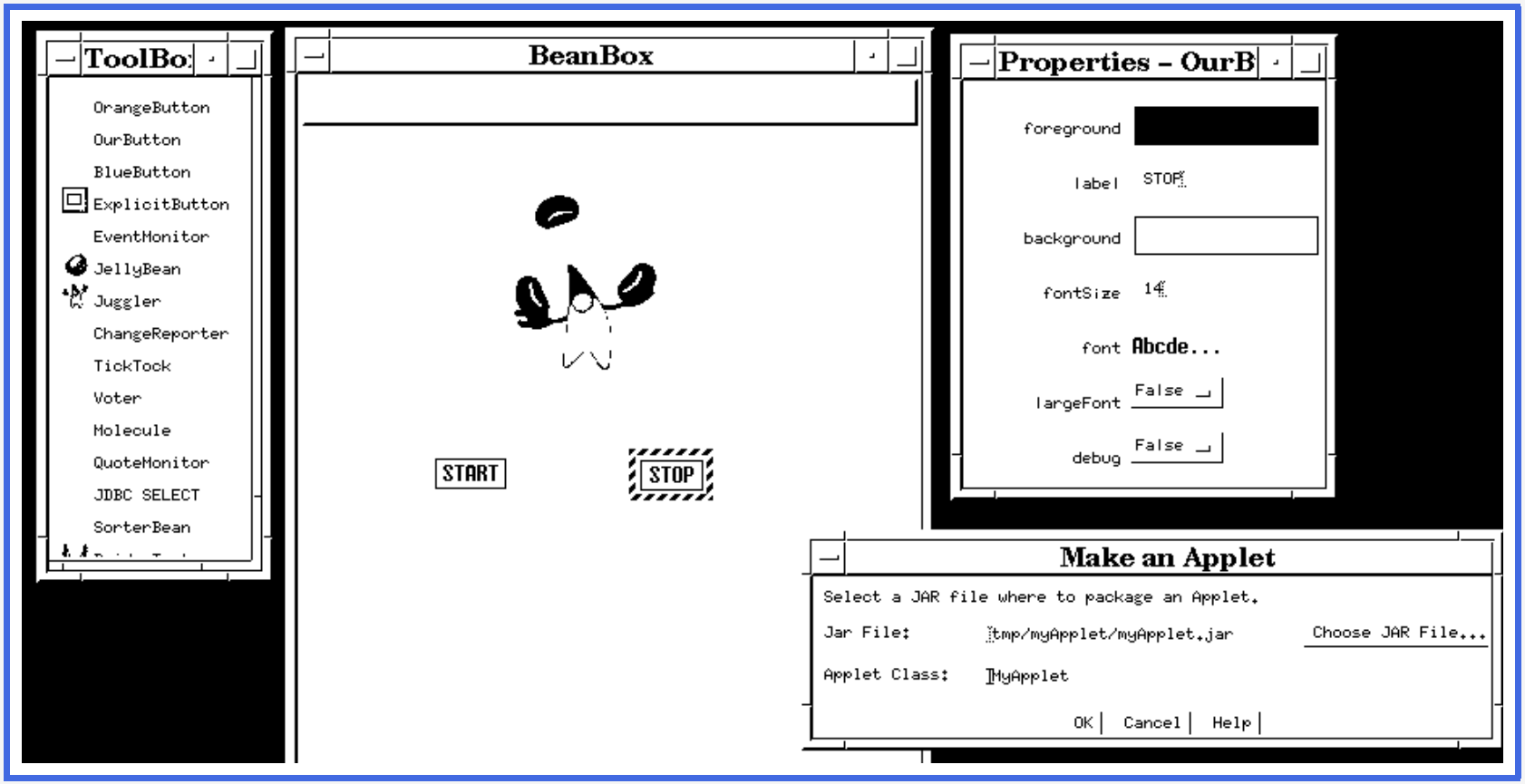

Figure (6) – *Déploiement de l'application, par exemple en tant qu'*"applet"

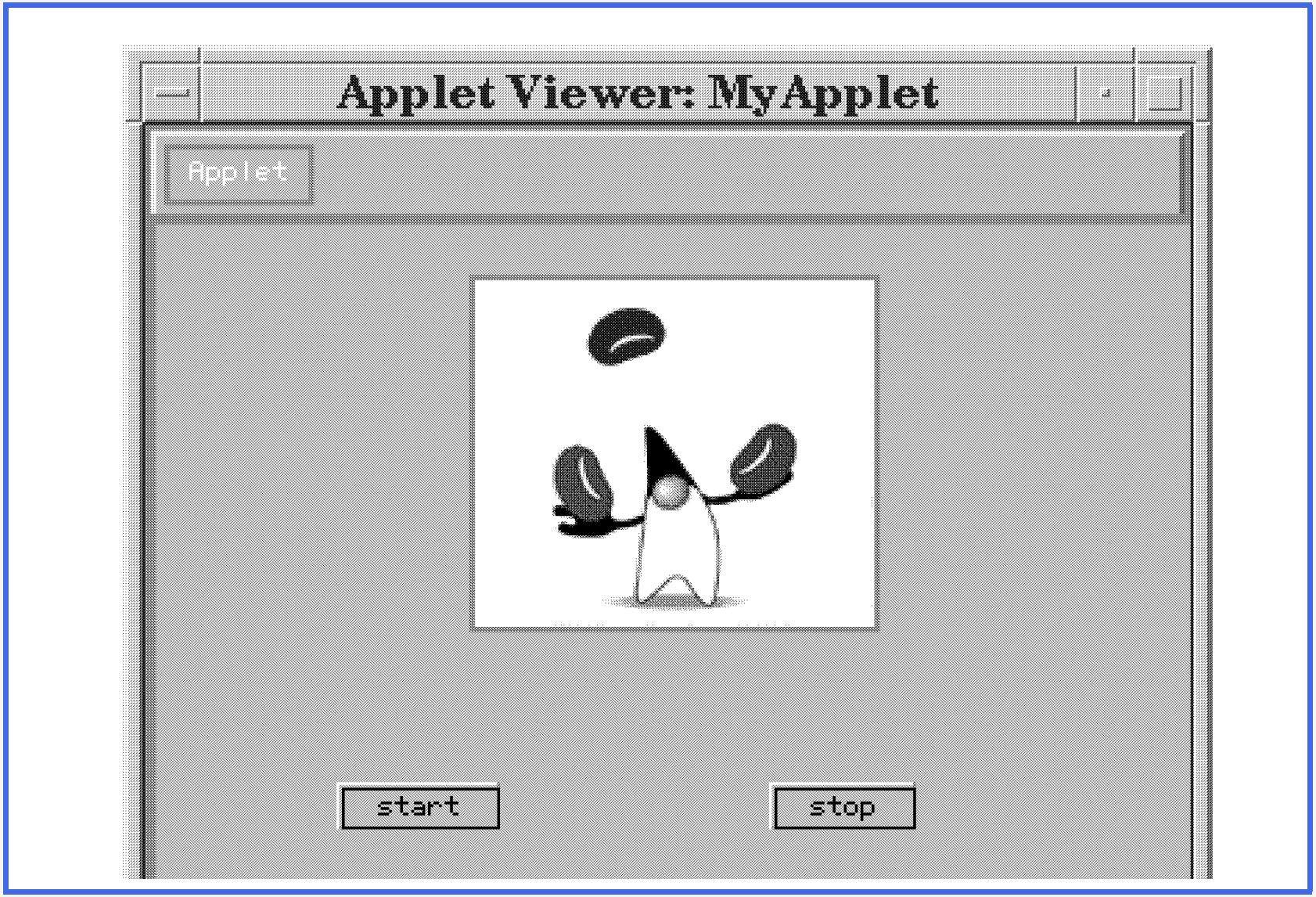

Figure (7) – *Exécution de l'application déployée*

Modèle de connection : écouteur/écouté selon le schéma “Observateur”

`../cpatterns/cObserver-MVC.s.pdf`

5 Génération automatique d'adapteurs

Le composant Jongleur n'a pas été écrit pour être un écouteur d'évènements. Le schéma "Adapteur" est utilisé pour pallier à ce problème.

La figure 8 explicite la solution d'implantation de ce *pattern* (version "adaptation par composition") dand le contexte de l'application précédente.

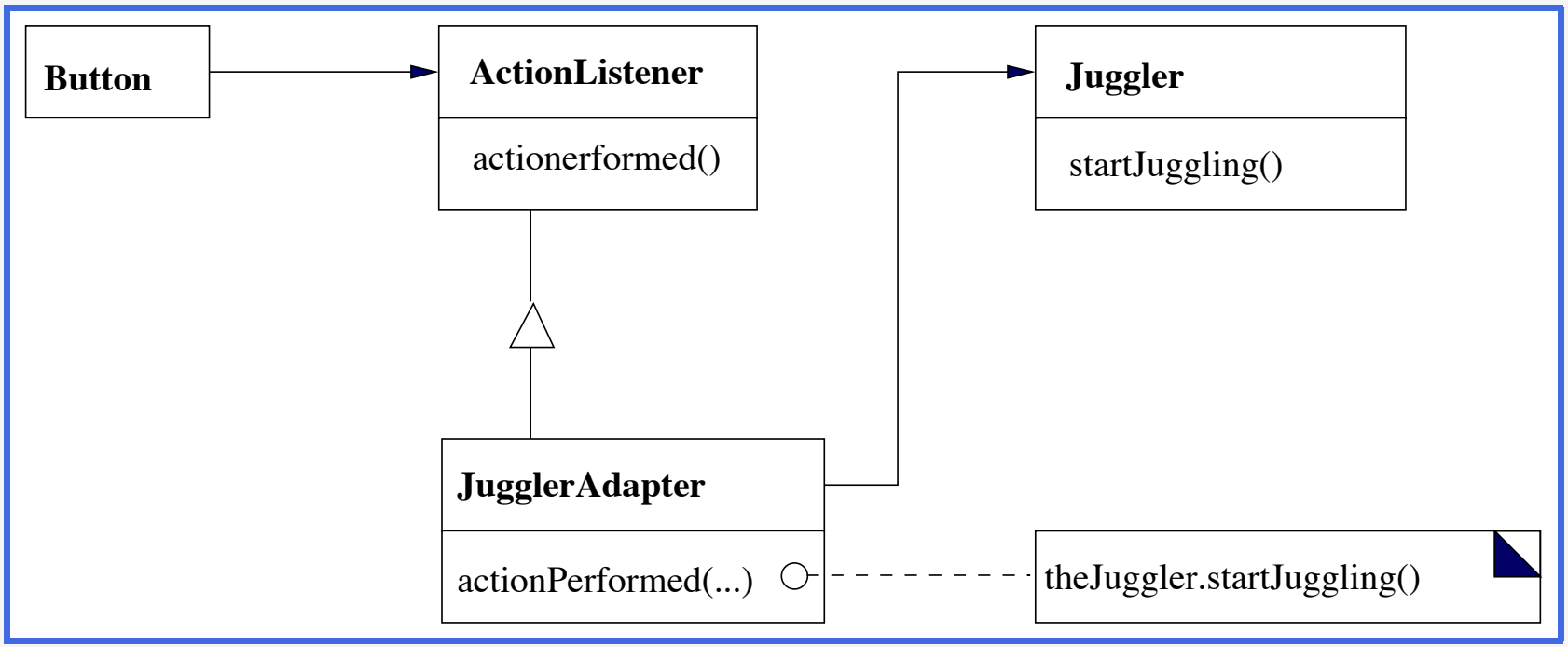

Figure (8) – *Adaptation par composition*

code généré [a] pour l'adaptation :

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

4  package tmp.sunw.beanbox;
5  import sunw.demo.juggler.Juggler;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;

9  public class ___Hookup_16add0c757 implements java.awt.event.ActionList
10 ener, java.io.Serializable {
11     private sunw.demo.juggler.Juggler target;
12     public void setTarget(sunw.demo.juggler.Juggler t) {
13         target = t; }
14     public void actionPerformed(java.awt.event.ActionEvent arg0) {
15         target.startJuggling(arg0); }
16 }
```

a. Voir répertoire $BINDIR/beanbox/tmp/sunw/beanbox

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

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

Un composant déployé dans ce modèle est soit un objet sérialisé qui pourra être cloné (copié) soit une classe qui pourra être instanciée.

Cas a) : une classe.

a. On 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.

```
1  import java.awt;
2
3  public class simplestBean extends Canvas {
4      public SimplestBean(){
5          this.setBackground(Color.red);
6      }
7
8      public Dimension getMinimumSize(){
9          return new Dimension(50,50);
10     } }
```

6.2 Finalisation, Description, Paquetage, Archivage

1. **Finalisation** : mise en état d'être archivé : dans ce contexte, simple compilation et production d'un fichier de type ".class".
2. **Description** : création d'un fichier appelé "*manifeste*" décrivant le composant.

```
1 cat > SimplestBean.mf
2 Name: SimplestBean.class
3 Java—Bean: True
```

3. **Paquetage**

   Création d'un fichier *.jar*) regroupant le code finalisé et la description :

```
1 jar —cfm simplestBean.jar simplestBean.mf SimplestBean.class
```

4. **Archivage**

   Spécifique à chaque environnement :

   assemblage

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

   Selon le contexte, le terme "composant" est appliqué à l'objet ou à l'archive rangée le contenant (e.g. composant OSGI).

6.3 Utilisation : sélection, paramétrage

1. **Sélection d'un composant sur étagère (ou palette)**.

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

2. **Paramétrage**.

   Les propriétés du composant prennent les valeurs spécifiées par le texte de son programme (valeurs par défaut, constructeurs),

   - et sont aussi éditables via un éditeur interactif spécialisé (cf. fig. 9).

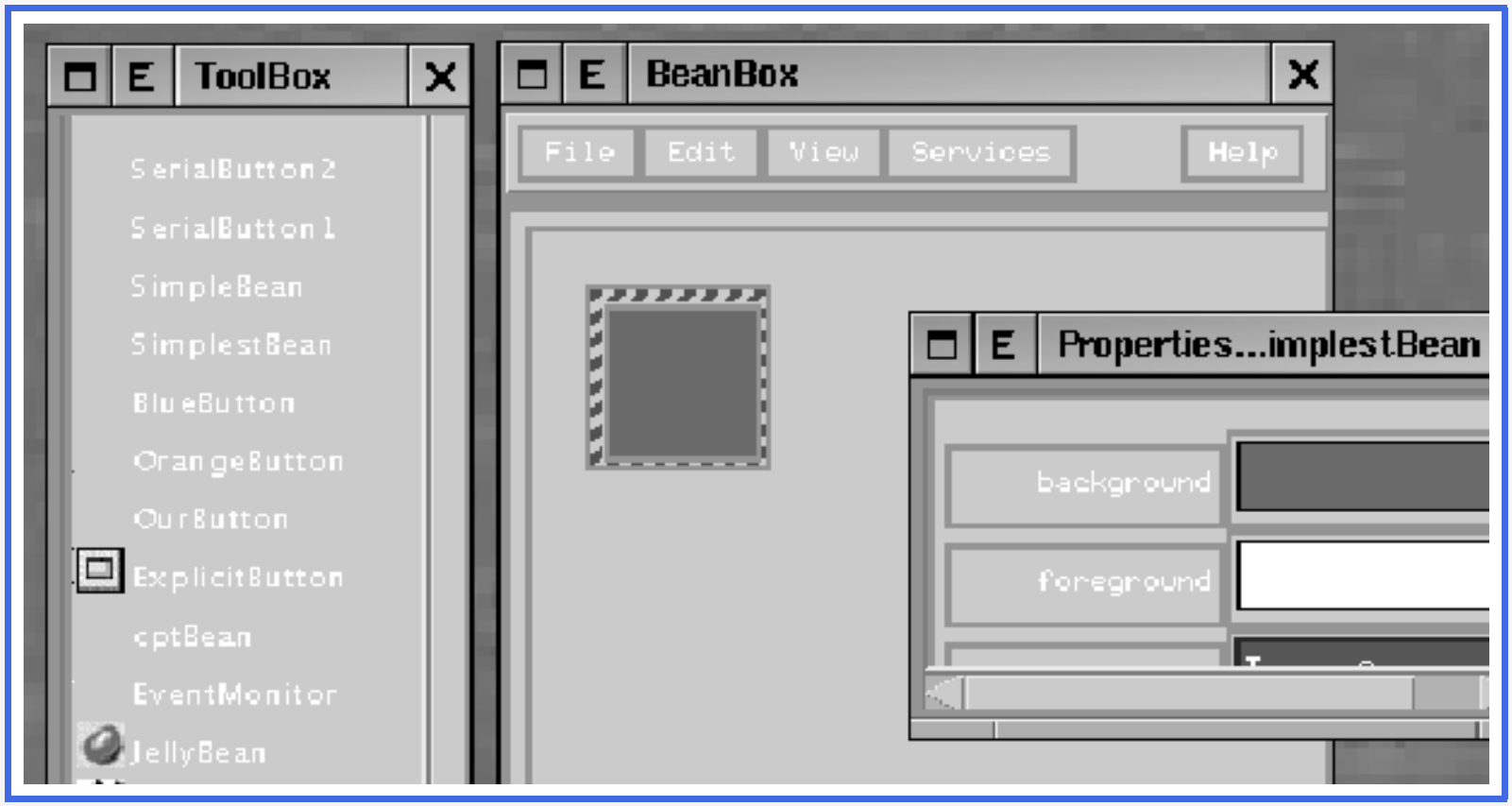

Figure (9) – *Sélection et paramétrage d'un composant*

7 Apparté : 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`.

```
1  import java.applet.*;
2  import java.awt.*;
3  import java.io.*;

5  public class SerialButton{

7      static void serialize(Object o, String filename){
8          try{
9              FileOutputStream f = new FileOutputStream(filename);
10             ObjectOutputStream s = new ObjectOutputStream(f);
11             s.writeObject(o);
12             s.flush(); }
13         catch (Exception e) { System.out.println(e); }
14     }
```

instantiation et **serialisation** :

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

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

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

Déclaration des composants - fichier "manifest"

```
1 Name: SerialButton1.ser
2 Java—Bean: True
3 Name: SerialButton2.ser
4 Java—Bean: True
```

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

```
1 jar —cfm SerialButtons.jar
2         SerialButton.mf
3         SerialButton1.ser SerialButton2.ser
4 cp SerialButtons.jar /home/dony/obj/java/beans/jars
```

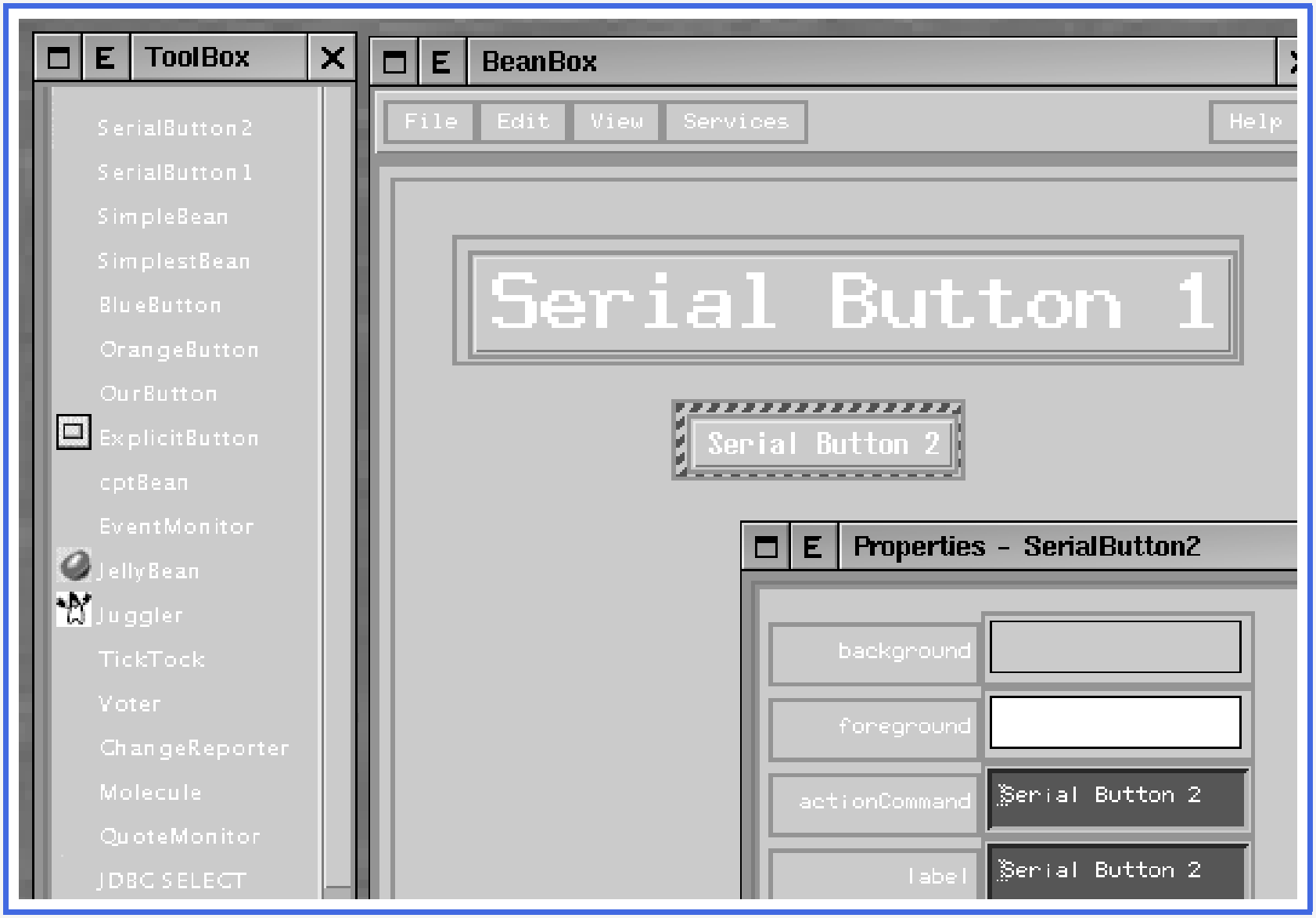

Figure (10) – *Composants sérialisés sur étagère*

8 Deux sortes de propriétés

Propriété : caractéristique nommée et paramétrable d'un composant.

Avec le modèle `JavaBeans`, une propriété est implantée par un attribut et des accesseurs.

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

9 Exemple de réalisation d'un composant possédant une propriété

Cette section montre la création d'un composant de type "compteur graphique" doté d'une propriété nommée *Value* implémentée avec une variable d'instance nommée *val*.

9.1 Définition de GraphicCptBean

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

4  public class GraphicCptBean extends Panel implements Serializable{
5      private int val;
6      private TextField textfield;

8      public GraphicCptBean(){ //Constructor
9          val = 0;
10         setLayout(new FlowLayout(FlowLayout.CENTER, 30, 10));
11         textfield = new TextField(this.newLabel());
12         add(textfield); }
```

```
14      public int getValue(){ return val; }

16      public void setValue(int newValue) {
17          val = newValue;
18          textfield.setText(this.newLabel());
19          this.repaint();}
```

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

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

7     private String newLabel(){
8         return(String.valueOf(this.getValue()));} }
```

9.2 Utilisation : assemblage

Voir figures 11 à 13

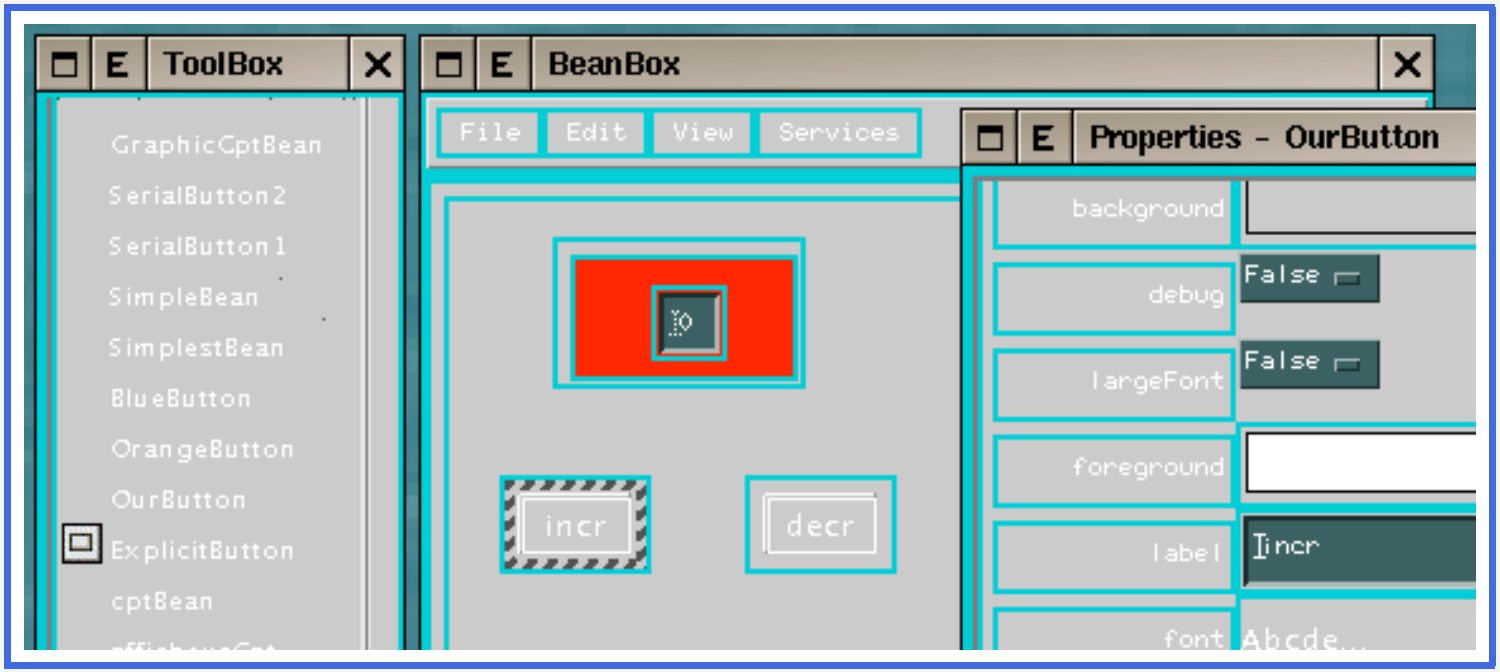

Figure (11) – *Edition d'un bouton*

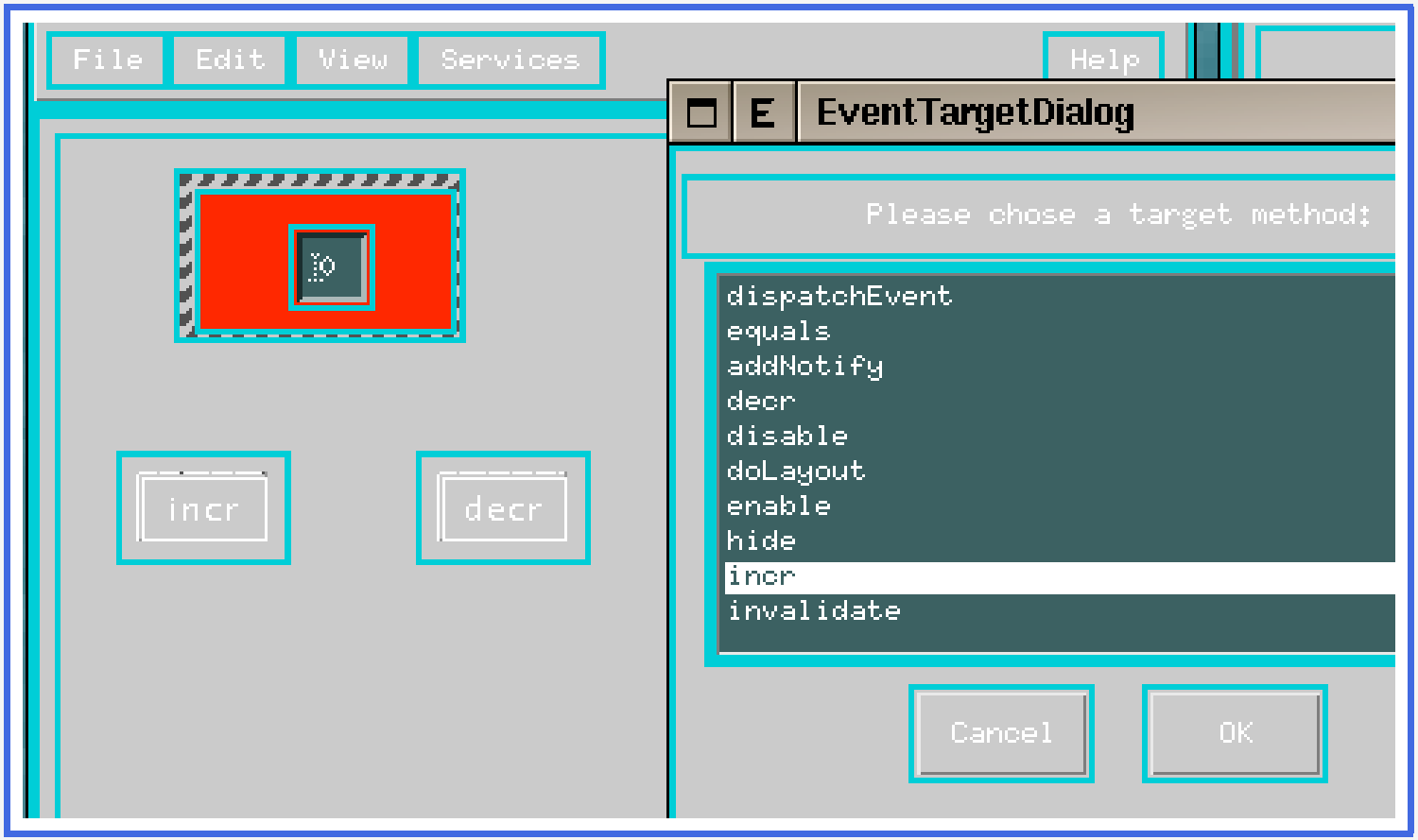

Figure (12) – *Assemblage du bouton et d'un compteur*

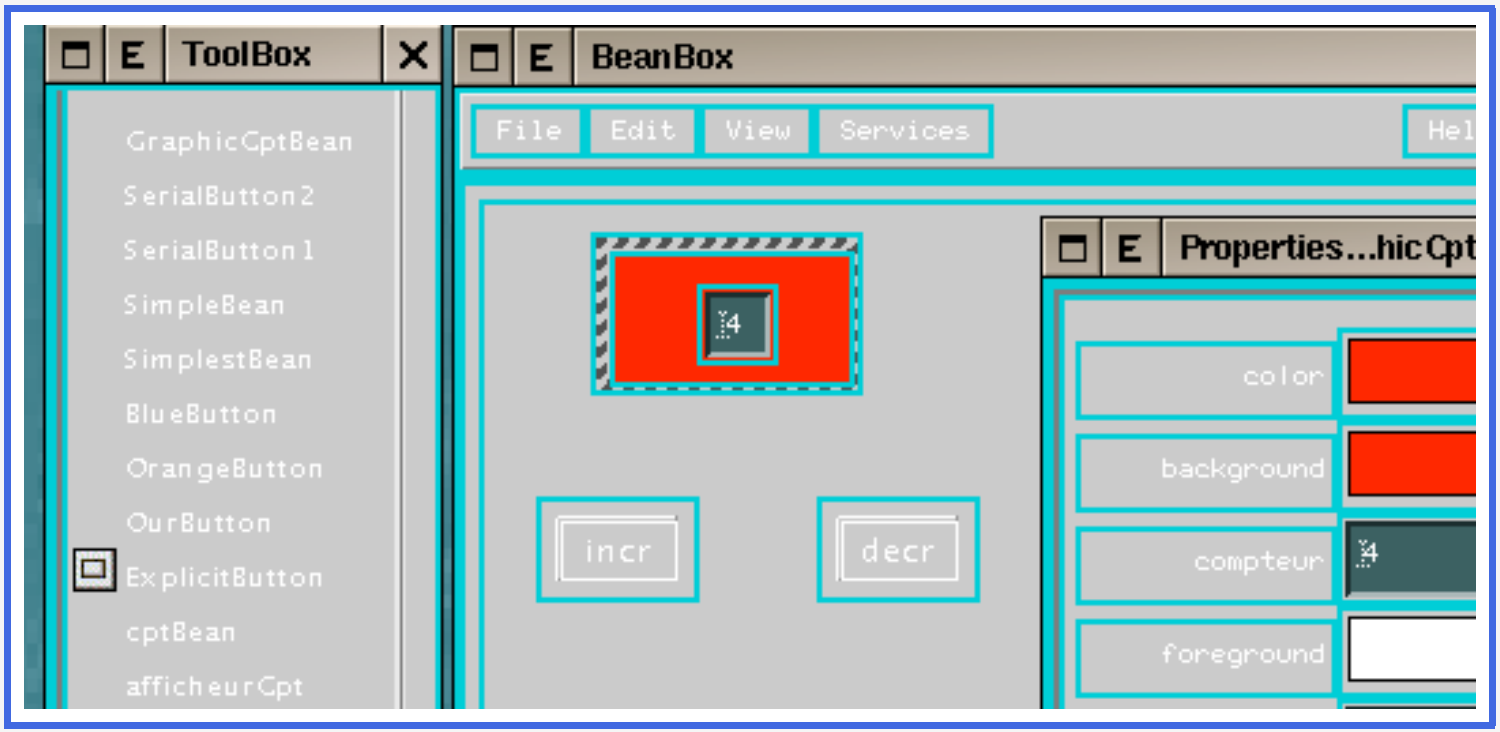

Figure (13) – *Utilisation de l'application résultante*

10 Protocole d'assemblage : Observateur

Les JavaBeans sont des composants que l'on assemble via un protocole de type "écouteur-écouté" ou "publish-subscribe" modélisés par le schéma *Observateur*.

On a en conséquence des composants écoutés et des écouteurs.

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.

11 Composants de type “écoutés”

Composants possédant un moyen d’indiquer à qui est intéressé qu’ils ont changé. (Exemple : tout modèle du MVC).

11.1 Propriétés liées

Définition

Propriété liée (*bound property*) : propriété dont le changement de valeur est signalé aux écouteurs potentiels du composant qui la détient.

Gestion des propriétés liées

L'API `javaBeans` (java.beans.*) propose un protocoles prédéfini pour gérer les propriétés liées.

- La classe `PropertyChangedEvent`

```
1 PropertyChangedEvent(
2     Object source,
3     String PropertyName,
4     Object oldValue,
5     Object newValue)
```

- l'interface `PropertyChangeListener` définissant la méthode `propertyChanged(...)` utilisable pour signaler un changement de valeur
- 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.

---

a. La 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.

12 Un Exemple plus complet avec NetBeans ...

Le composant précédant `GraphicCptBean` ne propose pas un découplage de type MVC.

Le code suivant propose une solution plus modulaire avec deux composants :

- Un composant visible (Afficheur de nombre entier), de type écouteur (via adaptation)
- Un composant invisible (Compteur), de type écouté, possédant une propriété liée (`Value`)

12.1 Code des composants

12.1.1 Un beans compteur - partie 1

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

4  public class CounterBean implements Serializable {

6      public static final String PROP2 = "value";

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

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

14     public Integer getValue() { return value;}

16     public void setValue(Integer v) {
17         Integer oldValue = value;
18         this.value = v;
19         propertySupport.firePropertyChange(PROP2, oldValue, value);}
```

12.1.2 Un beans compteur - partie 2

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

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

7     public void addPropertyChangeListener(PropertyChangeListener
          listener) {
8         propertySupport.addPropertyChangeListener(listener);
9     }

11    public void removePropertyChangeListener(PropertyChangeListener
          listener) {
12        propertySupport.removePropertyChangeListener(listener);
13    }
14 }
```

12.1.3 Un composant afficheur d'entiers

```
1  public class AfficheurBean extends Panel{
2
3      public AfficheurBean() {
4          super();
5          setLayout(new FlowLayout(FlowLayout.CENTER, 30, 10));
6          setSize(60,40);
7          setBackground(color);
8          textfield = new TextField("0");
9          add(textfield); }
10
11     private TextField textfield;
12     private Color color = Color.red;
13
14     public void affiche(String s){
15         textfield.setText(s); this.repaint();}
16
17     public void affiche(Integer i){
18         this.affiche(String.valueOf(i));}
19 }
```

12.2 Réalisation d'une application "counter game" sans écrire de code

Voir figures 14, 15, 16, 17 et 18.

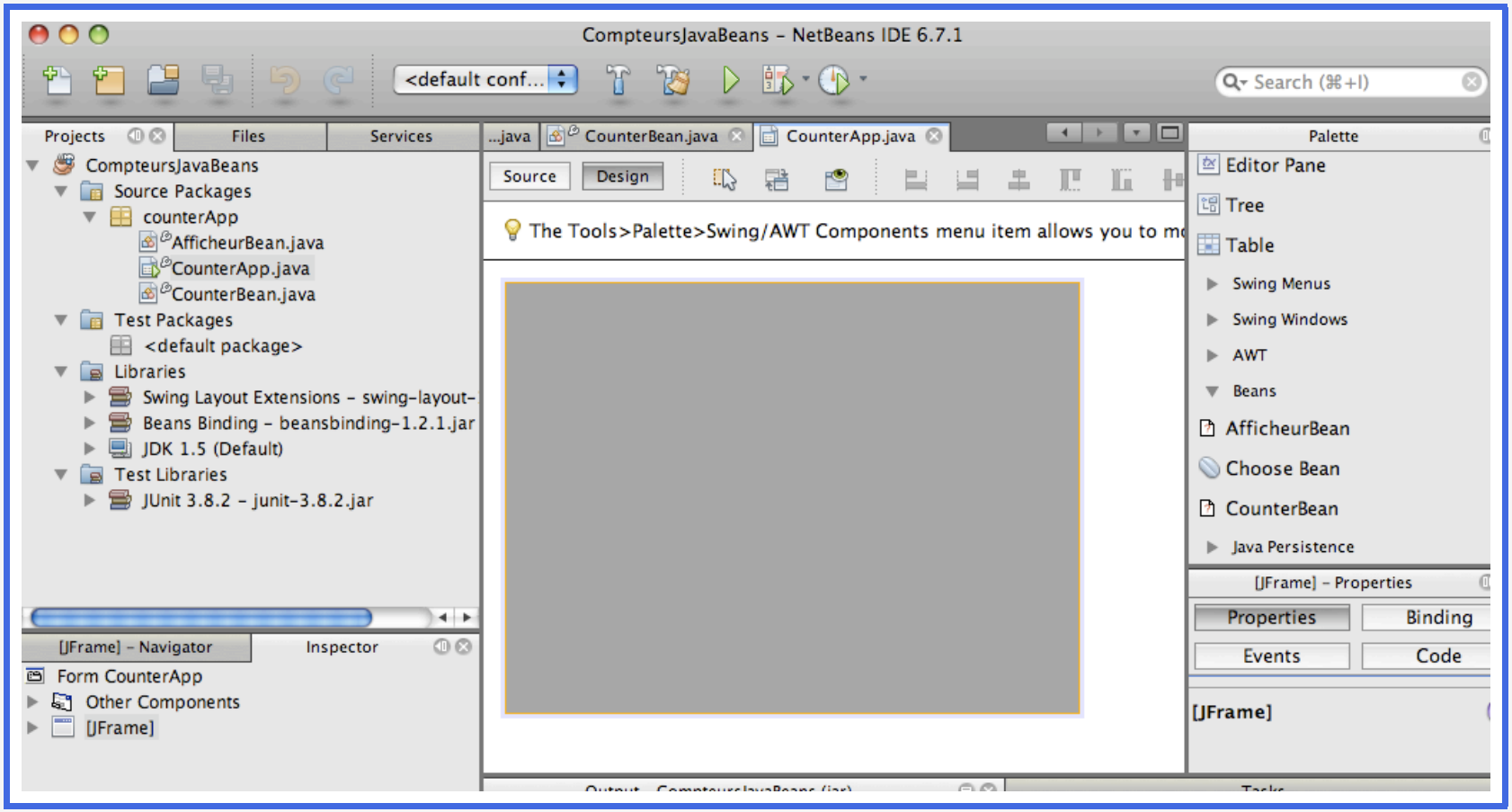

Figure (14) – *Nouveau projet et Palette*

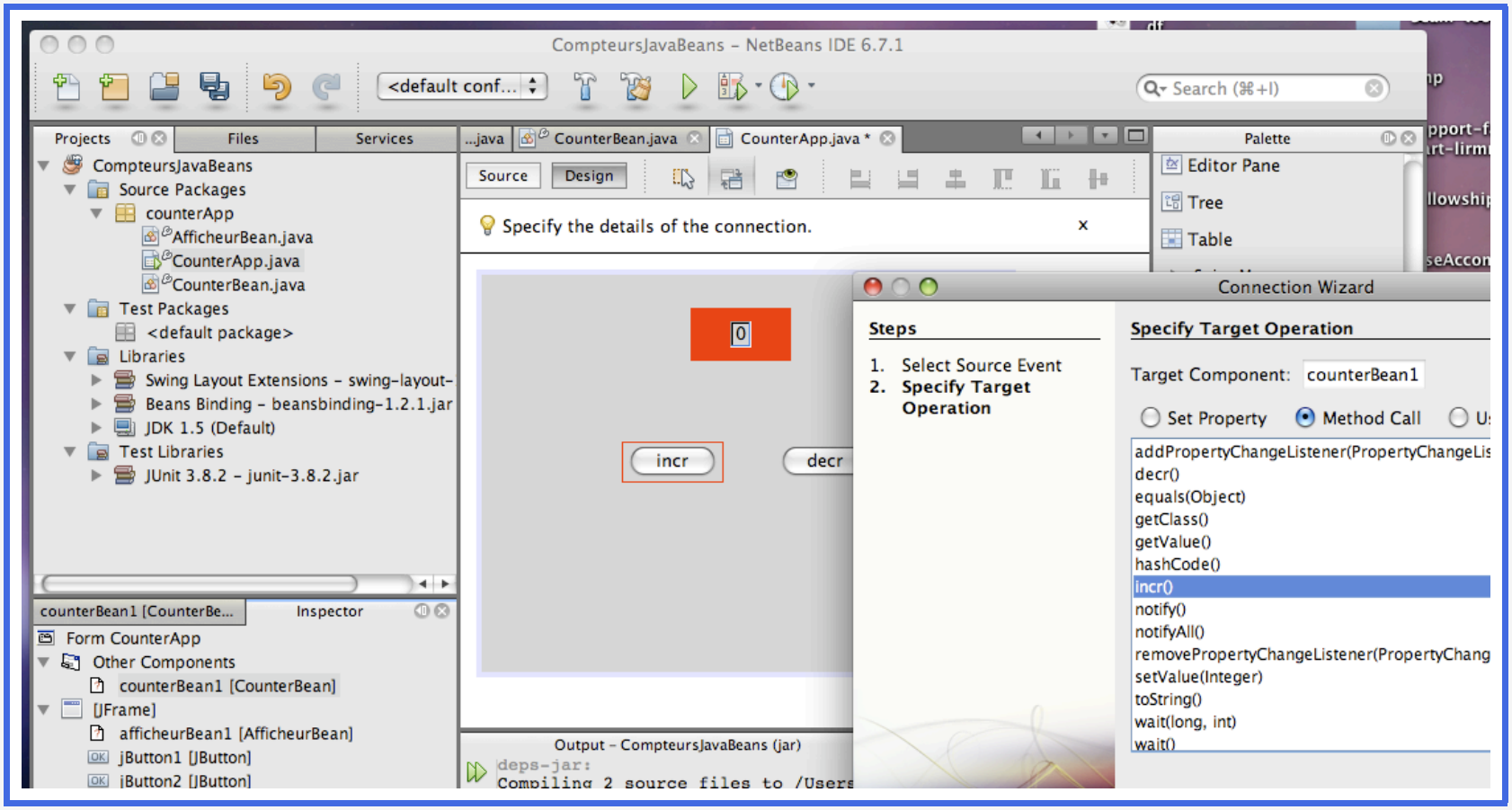

Figure (15) – *Connexion du bouton au compteur*

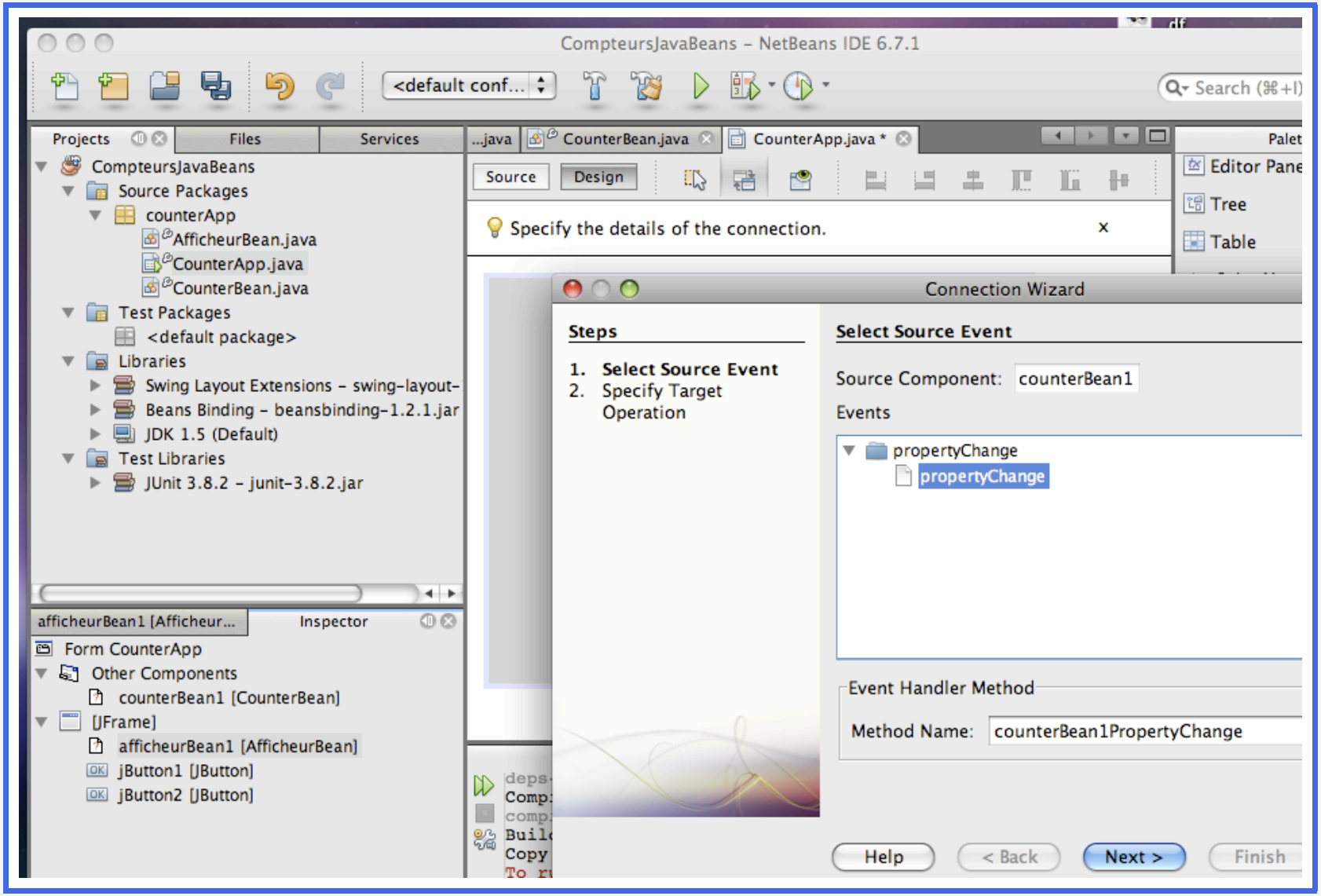

Figure (16) – *Connexion du compteur à l'afficheur - étape 1 - définir quel évènement de l'écouté va être utilisé*

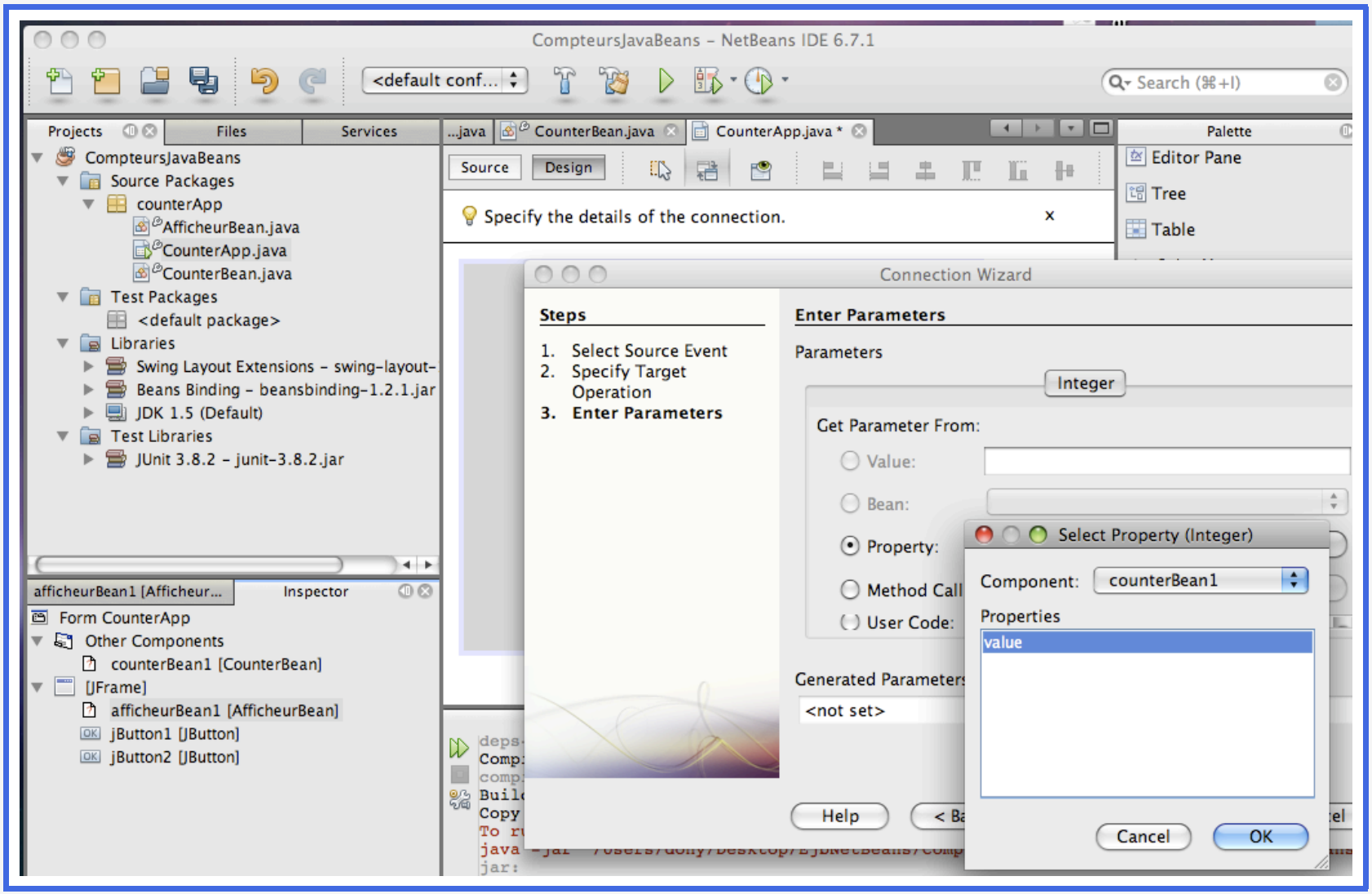

Figure (17) – *Connexion du compteur à l'afficheur - étape 2 - dire ce que va faire l'écouteur quand il reçoit l'évènement*

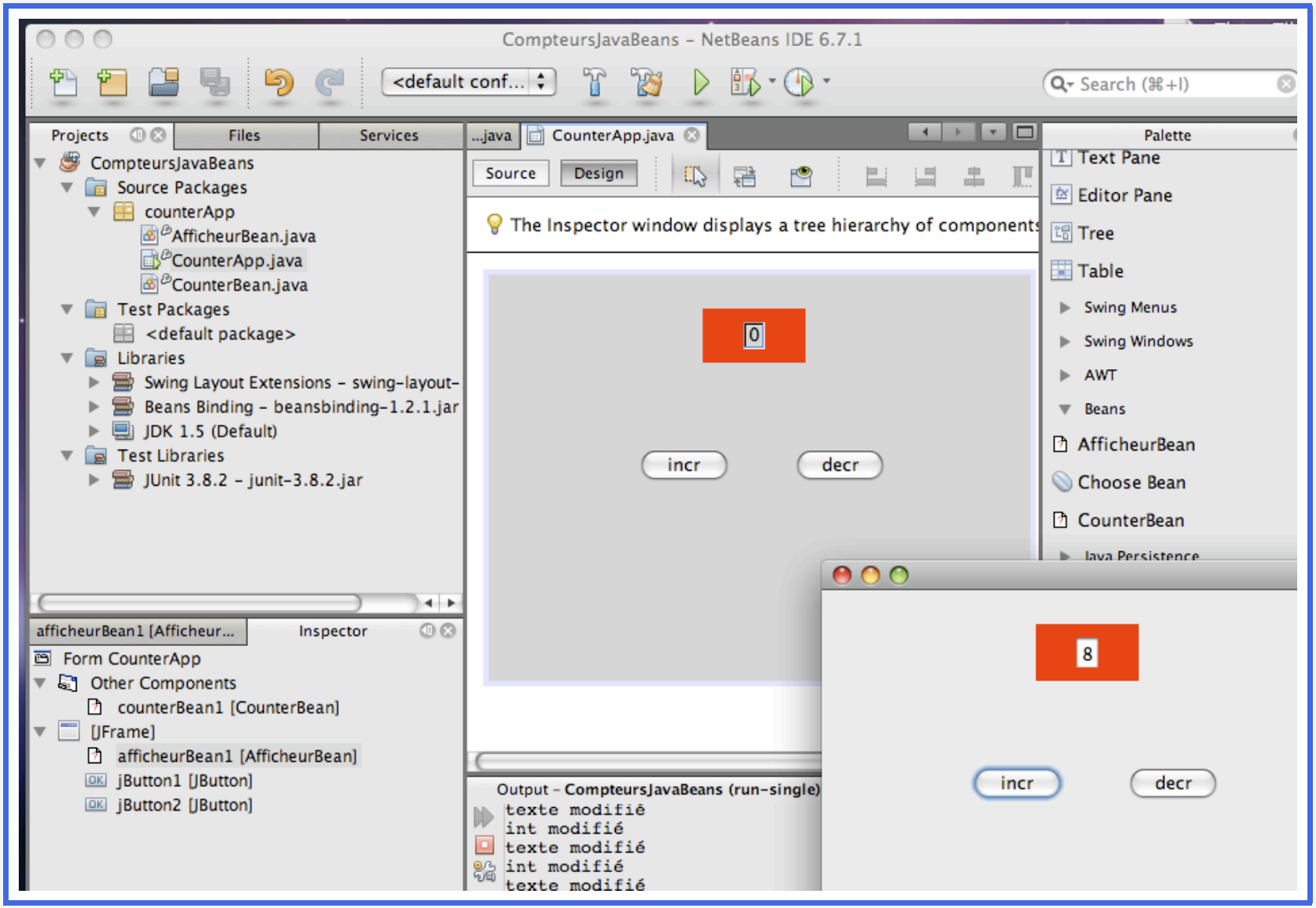

Figure (18) – *Execution*

12.3 Etude du code généré par l'environnement

Classe CounterApp, Adapteurs générés, Partie 1 :

```
1  class CounterApp {
2      private void initComponents() {

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

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

13         jButton1.setText("incr");
14         jButton1.addActionListener(new java.awt.event.ActionListener() {
15             public void actionPerformed(java.awt.event.ActionEvent evt) {
16                 jButton1ActionPerformed1(evt); } });
17     }
18     ...
```

Partie 2

```
2   ...
3   private void jButton1ActionPerformed1(java.awt.event.ActionEvent evt) {
4      counterBean1.incr(); }

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

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

13 Autres supports à l'assemblage "écouteur/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 écouteurs 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.

```
1  public class JellyBean extends Canvas {

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

6      public synchronized int getPrice() {
7          return ourPrice; }

9      public void setPrice(int newPrice) throws PropertyVetoException {
10         int oldPrice = ourPrice;
11         vetos.fireVetoableChange("price",
12                                  new Integer(oldPrice),
13                                  new Integer(newPrice));
14         ourPrice = newPrice; }

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

19     public void removeVetoableChangeListener(VetoableChangeListener l) {
20         vetos.removeVetoableChangeListener(l); } }
```

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*.

```
1  import java.lang.Reflect;
2
3  public class TestReflect {
4
5      public static void main (String[] args) throws NoSuchMethodException{
6          GraphicCptBean g = new GraphicCptBean();
7
8          Class gClass = g.getClass();
9
10         Class gSuperclass = gClass.getSuperclass();
11
12         Method gMeths[] = gClass.getDeclaredMethods();
13
14         Method getCompteur = gClass.getDeclaredMethod("getCompteur",
               null);
15
16         try {System.out.println(getCompteur.invoke(g, null));}
17         catch (Exception e) {}
18     }
19 }
```

Trace de l'exécution :

```
1 class GraphicCptBean
2 class java.awt.Panel
3 [Ljava.lang.reflect.Method;@5a2edaa1
4 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 accessseurs *get* et *set*, tous les évènements 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 recencées, 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* :

```
1 public BeanDescriptor getBeanDescriptor();
2 public PropertyDescriptor[] getPropertyDescriptors()
3 public EventSetDescriptor[] getEventSetDescriptors()
4 public MethodDescriptor[] getMethodDescriptors()
```

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

```
1  import java.beans.*;
2
3  class cptBeanBeanDescriptor extends SimpleBeanInfo {
4
5      public BeanDescriptor getBeanDescriptor() {
6          BeanDescriptor bd = new BeanDescriptor(cptBean.class);
7          bd.setDisplayName("Composant compteur");
8          bd.setShortDescription("Peut être incrémenté et décrémenté");
9          return bd;}
```

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

```
1    public PropertyDescriptor[] getPropertyDescriptors() {
2        try {PropertyDescriptor[] pds = {
3                new PropertyDescriptor("cpt", cptBean.class)};
4            return pds; }
5        catch(IntrospectionException e) {e.printStackTrace();}
6    }
```

La description des évènements qu'il emet. 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.

```
1   public EventSetDescriptor[] getEventSetDescriptors() {
2       try {EventSetDescriptor[] eds = {
3               new EventSetDescriptor
4                   (cptBean.class,
5                   "propertyChange",
6                   PropertyChangeListener.class,
7                   "propertyChange") };
8           return eds;}
9       catch(IntrospectionException e) {e.printStackTrace();}
10  }
```

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

```
1    public MethodDescriptor[] getMethodDescriptors() {
2        try {MethodDescriptor[] mds = {
3                new MethodDescriptor(cptBean.class.getMethod("incr", null)),
4                new MethodDescriptor(cptBean.class.getMethod("decr",
                     null))};
5            return mds; }
6        catch(NoSuchMethodException e) {e.printStackTrace();}
7    }
```

16 Java-FX

“Avec l’apparition de Java 8 en mars 2014, JavaFX devient la bibliothèque de création d’interface graphique officielle du langage Java, pour toutes les sortes d’application (applications mobiles, applications sur poste de travail, applications Web), le développement de son prédécesseur Swing étant abandonné (sauf pour les corrections de bogues).” [`https://fr.wikipedia.org/wiki/JavaFX`]

JavaFX est développé à partir du projet Open Source OpenJFX.

`https://openjfx.io/javadoc/11/javafx.base/module-summary.html`

`http://what-when-how.com/javafx-2/understanding-the-javafx-beans-convention-properties-and-bindings-part-1/`

17 Les propriétés liées dans d'autres modèles

17.1 JavaFX

Q : how are bound properties implemented in JavaFX

R : In JavaFX, bound properties are implemented using the Property and Observable interfaces, providing a mechanism for automatic notification when the value of a property changes. Here's an overview of how bound properties are implemented in JavaFX :

```
1   import javafx.beans.property.SimpleIntegerProperty;

3   public class BoundPropertiesExample {
4       public static void main(String[] args) {
5           SimpleIntegerProperty xProperty = new SimpleIntegerProperty(10);
6           SimpleIntegerProperty yProperty = new SimpleIntegerProperty(20);

8           // Bind zProperty to the sum of xProperty and yProperty
9           SimpleIntegerProperty zProperty = new SimpleIntegerProperty();
10          zProperty.bind(xProperty.add(yProperty));

12          System.out.println("x: " + xProperty.get() + ", y: " + yProperty.get() + ", z: " + zProperty.get());

14          xProperty.set(15); // Change xProperty and yProperty

16          // Print updated values, and zProperty is automatically updated
17          System.out.println("x: " + xProperty.get() + ", y: " + yProperty.get() + ", z: " + zProperty.get());
18      }
19  }
```

Listing (1) –

Property Interface : JavaFX properties extend the Property interface, which defines methods like getValue() and setValue(). This interface is implemented by various property types such as SimpleProperty, ReadOnlyProperty, and more.

Observable Interface