Introducción a la Programación

Tema 18. Introducción a la programación de interfaces de usuario

1 Introducción

Las interfaces gráficas son un elemento esencial en cualquier programa de ordenador. Son los mecanismos para aceptar las entradas de un usuario humano, presentarle las salidas y mostrar el estado del programa (tiempo restante, posibles errores, etc.)

En muchos casos el usuario de un programa mide la calidad del mismo por la calidad de la interfaz del programa. Es por lo tanto necesario esmerarse en el diseño y acabado de estas interfaces.

En este capítulo vamos a ver los elementos de las interfaces gráficas en general y especificación usando la tecnología Swing proporcionada por Java.

Para comprender las interfaces de usuario es necesario tener unos buenos conocimientos de los principios del diseño orientado a objetos. Esto es debido a que una interfaz gráfica se compone de un conjunto de objetos que se han diseñado con unos determinados patrones.

También es conveniente, para comprender el diseño de las interfaces gráficas, tener nociones sobre concurrencia. Este aspecto lo veremos en el próximo capítulo.

En cualquier caso el capítulo estará dedicado a la tecnología de las interfaces gráficas no a los principios estilísticos de los mismos ni a su usabilidad. Esos temas se tratarán en otro lugar.

Otras tecnologías son posibles para la construcción de interfaces gráficas. Una muy importante es la necesaria para construir interfaces para aplicaciones en la Web. Estas aplicaciones son

muy importantes pero los detalles de las correspondientes interfaces y la tecnología necesaria no las veremos en este capítulo.

2 Concepto de bean. Tipos de propiedades y eventos

El diseño *beans* toma coma base el diseño orientado a objetos que hemos visto en capítulos anteriores y le añade algunos elementos de estandarización. Estos elementos permiten usar los *beans* de una forma más sistemática y diseñar herramientas para manipularlos.

Los *beans* son, por lo tanto, objetos que siguen determinadas normas para dar nombre a las propiedades, eventos que pueden producir, etc. Veamos sus características.

Como los objetos los *beans* tienen propiedades que pueden ser **consultables** o también **modificables**. En general una propiedad tendrá un identificador (ejemplo *Edad*) y un tipo (en ese caso *Integer*). Cada propiedad está representada en vean por un método con un nombre que se fija a partir de la propiedad. Si la propiedad es consultable de tipo *T* e identificador *Id* entonces el bean tiene el método *T getId().* Si esa propiedad es modificable entonces el *bean* tiene el método *void setId(T a).* Por lo tanto observando los nombres de los métodos podemos deducir las propiedades del *bean*, si son consultables y si también son modificables. Los detalles de implementación no juegan aquí ningún papel. Es decir no tenemos información sobre si la propiedad se implementa mediante un atributo o mediante un cálculo basado sobre otros atributos.

Puede haber **propiedades indexadas**. Son aquellas propiedades cuyo tipo es *T[].* En este caso el *bean* tiene métodos adicionales para recuperar o modificar un elemento concreto del array de valores de la propiedad. Así si tenemos la propiedad modificable *Coeficientes* de tipo *Double[]* el vean tendrá los métodos.

- *Double[] getCoeficientes();*
- *void setCoeficientes(Double[] c);*
- *Double getCoeficientes(int index);*
- *void setCoeficientes(int index, Double c);*

En general puede haber **propiedades observadas**. Son propiedades que admiten objetos interesados en ser notificados del cambio de la propiedad. Esos objetos oyentes los denominaremos **listeners.** Si el bean tiene una propiedad observada entonces ofrece métodos para que los posibles *listeners* indiquen que están interesados en la notificación o que ya han dejado de estarlo. A una propiedad observada con identificador **Id** le corresponden los métodos:

- *void addIdChangeListener(IdChangeListener listener);*
- *void removeIdChangeListener(IdChangeListener listener);*

La idea es que cuando la propiedad Id cambia por alguna razón entonces el objeto observado envía un evento del tipo *IdChangeEvent* a todos los *listeners* que indicaron su interés. Aquí los tipos *IdChangeEvent* y *IdChangeListener* tienen que ser implementados.

Una forma más general, válida para todas las propiedades es parametrizar los métodos anteriores con el identificador de la propiedad. Entonces los métodos serían de la forma:

- *void addPropertyChangeListener(PropertyChangeListener listener):* El *listener* indica que está interesado en el cambio de todas las propiedades.
- *void addPropertyChangeListener(String property, PropertyChangeListener listener):* El *listener* indica que está interesado en el cambio de la propiedad *property*.
- *void removePropertyChangeListener(PropertyChangeListener listener):* El *listener* indica que ya no está interesado en el cambio de las propiedades.
- *void removePropertyChangeListener(String property, PropertyChangeListener listener):* El *listener* indica que ya no está interesado en el cambio de la propiedad *property*.

Como antes una propiedad cambia entonces el objeto observado envía un evento del tipo *PropertyChangeEvent* a todos los *listeners* que indicaron su interés. Ahora los tipos *PropertyChangeEvent* y *PropertyChangeListener* pueden ser reutilizados y son ofrecidos en el API de java en el paquete *java.beans.* En ese paquete se ofrece, también, la clase de utilidad *PropertyChangeSupport* que nos será de gran ayuda para implementar el mecanismo de notificación de las propiedades observables en los *beans* que las tengan. Esta clase dispone de métodos adecuados para implementar los anteriores y otros que hacer la notificación. Son los métodos:

- *void firePropertyChange(String property, Object oldValue, Object newValue):* Envía un evento de tipo *PropertyChangeEvent* a todos los *listeners* de la propiedad property.
- Hay disponibles variantes de este método

El tipo *PropertyChangeListener* tiene el método

- *void propertyChange(PropertyChangeEvent evt)*

Cuando una propiedad cambia los listeners interesados son notificados mediante la llamada de este método. Y el tipo *PropertyChangeEvent* los métodos

- *Object getNewValue()*
- *Object getNewValue()*
- *String getPropertyName()*
- *Object getSource()*

Posteriormente veremos algún ejemplo.

Un último tipo de propiedades son las **propiedades observables cuyo cambio puede ser restringido** por los posibles oyentes.

Si el *bean* tiene una propiedad observada cuyo cambio puede ser restringido entonces ofrece métodos para que los posibles *listeners* indiquen que están interesados en la notificación o que ya han dejado de estarlo. A una propiedad de este tipo con identificador **Id** le corresponden los métodos:

- *void addIdVetoableChangeListener(IdVetoableChangeListener listener);*
- *void removeIdVetoableChangeListener(IdVetoableChangeListener listener);*

La idea es que cuando la propiedad Id cambia por alguna razón entonces el objeto observado envía un evento del tipo *IdVetoableChangeEvent* a todos los *listeners* que indicaron su interés. Aquí los tipos *IdVetoableChangeEvent* y *IdVetoableChangeListener* tienen que ser implementados.

Como antes una forma más general, válida para todas las propiedades es parametrizar los métodos anteriores con el identificador de la propiedad. Entonces los métodos serían de la forma:

- *void addVetoableChangeListener(VetoableChangeListener listener):* El *listener* indica que está interesado en el cambio de todas las propiedades.
- *void addVetoableChangeListener(String property, VetoableChangeListener listener):* El *listener* indica que está interesado en el cambio de la propiedad *property*.
- *void removePropertyChangeListener(VetoableChangeListener listener):* El *listener* indica que ya no está interesado en el cambio de las propiedades.
- *void removePropertyChangeListener(String property, VetoableChangeListener listener):* El *listener* indica que ya no está interesado en el cambio de la propiedad *property*.

Como antes una propiedad cambia entonces el objeto observado envía un evento del tipo *VetoableChangeEvent* a todos los *listeners* que indicaron su interés. Ahora los tipos *VetoableChangeEvent* y *VetoableChangeListener* pueden ser reutilizados y son ofrecidos en el API de java en el paquete *java.beans.* En ese paquete se ofrece, también, la clase de utilidad *VetoableChangeSupport* que nos será de gran ayuda para implementar el mecanismo de notificación de las propiedades observables en los *beans* que las tengan. Esta clase dispone de métodos adecuados para implementar los anteriores y otros que hacer la notificación. Son los métodos:

- *void fireVetoableChange(String property, Object oldValue, Object newValue) throws PropertyVetoException*
  Envía un evento de tipo *VetoableChangeEvent* a todos los *listeners* de la propiedad *property*. Si alguno de los listeners veta el cambio se dispara excepción *PropertyVetoException*
- Hay disponibles variantes de este método

El tipo *VetoableChangeListener* tiene el método

- *void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException*

Cuando una propiedad cambia los *listeners* interesados son notificados mediante la llamada de este método. Si el *listener* veta el cambio dispara la excepción.

En el ejemplo siguiente implementamos un Punto con propiedades observables *X, Y, DistanciaAlOrigen.* Como se ve en el ejemplo se reutiliza por delegación la clase *PropertyChangeSupport* y se hace uso de método firePropertyChange dentro del nuevo diseño de los métodos setX y setY que son los que pueden cambiar esas propiedades.

```
public class ListenablePunto extends Punto {

      final private PropertyChangeSupport boundProperty =
         new PropertyChangeSupport(this);
      private static int n = 1;
      public int id;

      ListenablePunto() {
            super();  id = n;  n++;
      }

      ListenablePunto(Double x, Double y) {
            super(x, y); id = n; n++;
      }

      public void addPropertyChangeListener(String property,
             PropertyChangeListener listener) {
          boundProperty.addPropertyChangeListener(property, listener);
      }

      public void removePropertyChangeListener(String property,
            PropertyChangeListener listener) {
        boundProperty.removePropertyChangeListener(property,listener);
      }

      @Override
      public void setX(Double x) {
            Double oldX = getX();
            Double oldD = super.getDistanciaAlOrigen();
            super.setX(x);
            boundProperty.firePropertyChange("x", oldX, x);
            boundProperty.firePropertyChange(
              "distanciaAlOrigen", oldD, super.getDistanciaAlOrigen());
      }

      @Override
      public void setY(Double y) {
            Double oldY = getY();
            Double oldD = super.getDistanciaAlOrigen();
            super.setY(y);
            boundProperty.firePropertyChange("y", oldY, y);
            boundProperty.firePropertyChange(
              "distanciaAlOrigen", oldD, super.getDistanciaAlOrigen());
```

```
	}

	@Override
	public String toString() {
		return "ListenablePunto [id = "+id+", getX()=" +
		   getX() + ", getY()=" + getY() + "]";
	}
}
```

```
public class VetoablePunto extends Punto {

	final private VetoableChangeSupport vetoableProperty =
	  new VetoableChangeSupport(this);

	private static int n = 1;
	public int id;

	VetoablePunto() {
		super(); id = n; n++;
	}

	VetoablePunto(Double x, Double y) {
		super(x, y); id = n; n++;
	}

	public void addVetoableChangeListener(String property,
			VetoableChangeListener listener) {
	 vetoableProperty.addVetoableChangeListener(property, listener);
	}

	public void removeVetoableChangeListener(String property,
			VetoableChangeListener listener) {
	vetoableProperty.removeVetoableChangeListener(property,listener);
	}

	@Override
	public void setX(Double x) {
		Double oldX = getX();
		Double oldD = super.getDistanciaAlOrigen();
		super.setX(x);
		try {
			vetoableProperty.fireVetoableChange("x", oldX, x);
			vetoableProperty.
			    fireVetoableChange("distanciaAlOrigen", oldD,
				super.getDistanciaAlOrigen());
		} catch (PropertyVetoException e) {
			super.setX(oldX);
		}
	}

	@Override
	public void setY(Double y) {
		Double oldY = getY();
```

```
            Double oldD = super.getDistanciaAlOrigen();
            super.setY(y);
            try {
                  vetoableProperty.fireVetoableChange("y", oldY, y);
                  vetoableProperty.
                     fireVetoableChange("distanciaAlOrigen", oldD,
                          super.getDistanciaAlOrigen());
            } catch (PropertyVetoException e) {
                  super.setY(oldY);
            }
      }

      @Override
      public String toString() {
            return "VetoablePunto [id = "+id+", getX()=" + getX() + ",
getY()=" + getY() +", distancia ="+getDistanciaAlOrigen()+"]";
      }
}
```

Un posible programa principal para comprobar lo anterior puede ser de la forma:

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

            VetoablePunto p1 = BeansPuntos.createVetoable(2., 200.);
            VetoablePunto p2 = BeansPuntos.createVetoable(11., 10.);
            ConsolaConVeto c1 = Consolas.createConsolaConVeto();
            ConsolaConVeto c2 = Consolas.createConsolaConVeto();

            p1.addVetoableChangeListener("x",c1);
            p1.addVetoableChangeListener("distanciaAlOrigen",c2);

            p1.setX(-3.0);
            p1.setY(2.);
            p2.setY(4.0);

            System.out.println(p1+","+p2);
      }
```

Se ha usado un objeto de tipo *ConsolaConVeto* que implementa el tipo *VetoableChangeListener* de la forma:

```
public class ConsolaConVeto implements VetoableChangeListener {

      private static int nc = 1;
      private int id;

      ConsolaConVeto() {
            super(); id = nc; nc++;
      }

      @Override
      public void vetoableChange(PropertyChangeEvent  evt)
```

```
        throws PropertyVetoException {
           String property = evt.getPropertyName();
           if(property.equals("x")){
                 Double x = (Double) evt.getNewValue();
                 if(x < 0.){
                       System.out.println(…);
                       throw new PropertyVetoException(property, evt);
                 }
           }else if(property.equals("y")){
                 Double y = (Double) evt.getNewValue();
                 if(y < 100.) {
                       System.out.println(…);
                       throw new PropertyVetoException(property, evt);
                 }
           }else if(property.equals("distanciaAlOrigen")){
                 Double d = (Double) evt.getNewValue();
                 if(d < 10.) {
                       System.out.println(…);
                       throw new PropertyVetoException(property, evt);
                 }
           }

     }

     @Override
     public String toString() {
           return "ConsolaConVeto [id=" + id + "]";
     }
}
```

De forma similar diseñaríamos los objetos que implementan *PropertyChangeListener* y el resto de detalles.

3 Sincronización de propiedades entre beans

En el diseño e implementación de interfaces de usuario es necesario mantener sincronizados los valores de propiedades de distintos *beans*. Además es conveniente generar el código que mantiene la sincronización de la forma más automática posible. Para conseguir esto puede ser de gran ayuda el uso del software producido en proyecto *Swing Data Binding (JSR 295).* Para usarlo hay que añadir la librería *beansbinding-1.2.1.jar* al proyecto que estamos diseñando.

En ese paquete tenemos disponible varias clases de gran utilidad. Veamos algunas de ellas.

Para hacer más automática la gestión de propiedades de un *bean* podemos usar el tipo ***BeanProperty<S,V>.*** Este tipo nos permite gestionar una propiedad de tipo *V* de un *bean* de tipo *S*.

Parámetros del tipo: S (tipo del objeto fuente), V (tipo de la propiedad)

Objetos de este tipo se crean con un método estático que toma como parámetro un *String* con el nombre de la propiedad (puede ser una secuencia de identificadores de propiedades separadas por puntos para indicar un camino hacia una propiedad)

```
static <S,V> BeanProperty<S,V> create(String path)

BeanProperty<PrimeNumbersTask, Integer> pm=
         BeanProperty.create("progress");
BeanProperty<JProgressBar, Integer> jp = BeanProperty.create("value");
```

El tipo ***ELProperty<S,V>*** nos permite gestionar una propiedad derivada de tipo *V* de un *bean* de tipo *S*.

Parámetros del tipo: S (tipo del objeto fuente), V (tipo de la propiedad)

Objetos de este tipo se crean con un método estático que toma como parámetro un *String* con una expresión que calcula el valor de la propiedad (la expresión admite los operadores del ..)

```
static <S,V> ELProperty<S,V> create(String expression)

ELProperty<Person,String>  el1 =
       ELProperty.create("${firstName} ${lastName}");
ELProperty<Person,Boolean> el2  =
       ELProperty.create("${mother.age > 65}");
```

El tipo *AutoBinding<SS,SV,TS,TV>* nos permite mantener sincronizados los valores de dos propiedades. La estrategia de sincronización puede ser de varios tipos: lectura una vez, lectura o lectura escritura. El tipo tiene varios parámetros para indicar el tipo del objeto fuente, el tipo des su propiedad, el tipo del objeto destino y de su propiedad.

Los parámetros del tipo son:

- `SS` – tipo del objeto fuente
- `SV` – tipo de la propiedad del objeto fuente
- `TS` - tipo del objeto destino
- `TV` - tipo de la propiedad del objeto destino

Teniendo en cuenta los objetos que desempeñan el papel de objeto fuente y destino las estrategias de sincronización posibles son:

- `AutoBinding.UpdateStrategy.READ_ONCE`
- `AutoBinding.UpdateStrategy.READ`
- `AutoBinding.UpdateStrategy.READ_WRITE`

La estrategias posibles son entonces: lectura (cuando cambia la propiedad del objeto fuente cambia la del destino) o lectura escritura (cuando cambia una de las propiedades cambia la otra.

La clase *Bindings* es una factoría para crear objetos del tipo anterior.

- *static <SS,SV,TS,TV> AutoBinding<SS,SV,TS,TV> createAutoBinding(*
  AutoBinding.UpdateStrategy strategy,
  SS sourceObject,
  Property<SS,SV> sourceProperty,
  TS targetObject,
  Property<TS,TV> targetProperty)

Un ejemplo es:

```
BeanProperty<PrimeNumbersTask, Integer> pm =
             BeanProperty.create("progress");
BeanProperty<JProgressBar, Integer> jp =
             BeanProperty.create("value");
AutoBinding<PrimeNumbersTask, Integer, JProgressBar, Integer>
   autoBinding = Bindings.createAutoBinding(UpdateStrategy.READ, task,
       pm, progressBar, jp);
autoBinding.bind();
```

En muchos casos los tipos de las propiedades fuente y destino no son los mismos. Es necesario introducir mecanismos de cambio de tipo. Es podemos hacerlo con objetos de la clase *Converter<S,T>*. En este caso S es tipo de la propiedad fuente y T el tipo de la propiedad destino. Sus métodos son:

```
T convertForward(S value)
S convertReverse(T value)
```

También podemos introducir objetos que validen los valores de la propiedad del objeto fuente mediante la clase *Validator<S>.* Tiene el método:

```
Validator.Result validate(S value)
```

Dónde los objetos de tipo *Result* codifican los posibles errores en la validación. Si la validación es correcta el resultado del método *validate* debe ser *null*.

Por último para que todo funcione los objetos de tipo *Autobinding* ofrecen la posibilidad a los listeners interesados en ser notificados sin ha existido un error en la validación o en la sincronización.

Los *listeners* específicos de los objetos de tipo *Autobinding* son del tipo *BindingListener.* Este tipo tiene, entre otros, los siguientes métodos:

- *void syncFailed(Binding binding,Binding.SyncFailure failure):* Ha habido un fallo en la sincronización codificado en el parámetro failure. El fallo puede ser debido a la validación entre otras razones.
- *void synced(Binding binding):* Las propiedades se han sincronizado

Un ejemplo de estas ideas podemos ver en el trozo de código siguiente donde se quiere sincronizar el valor de la propiedad *X* de tipo *Double* de un objeto de tipo *Punto* con una propiedad de tipo *String* de un objeto de tipo *JTextField.* Para que un objeto de tipo Punto pueda ocupar el papel de fuente debe ser capaz de notificar a los listeners interesados los cambios en la propiedad. El tipo *ListenablePunto* hace ese trabajo.

```
BeanProperty<ListenablePunto, Double> listenablePuntoBeanProperty =
     BeanProperty.create("x");
BeanProperty<JTextField, String> jTextFieldBeanProperty =
     BeanProperty.create("text");
AutoBinding<ListenablePunto, Double, JTextField, String> autoBinding =
     Bindings.createAutoBinding(
         UpdateStrategy.READ_WRITE,
         p1,
         listenablePuntoBeanProperty,
         xText,
         jTextFieldBeanProperty);
autoBinding.setConverter(cvDoubleString);
autoBinding.setValidator(validator);
autoBinding.addBindingListener(listener);
autoBinding.bind();
```

Como los tipos de las propiedades a sincronizar son diferentes necesitamos un convertidor de tipo. Es el objeto *cvDoubleString*. Si queremos validar los valores posibles en la propiedad *X* y los posibles errores de formato en el texto de campo de texto usamos el validador *validator*.
Para poder imprimir en una etiqueta (en este caso la variable mensaje abajo) los posibles errores que puedan aparecer usamos el objeto *listener.*

```
private Converter<Double,String> cvDoubleString =
  new Converter<Double,String>(){
            public String convertForward(Double a) {
                  return a.toString();
            }

            public Double convertReverse(String a) {
                  Double r;
                  try {
                        r = new Double(a);
                  } catch (NumberFormatException e) {
                        r = Double.NEGATIVE_INFINITY;
                  }
                  return r;
            }
      };
```

```
Validator<Double> validator = new Validator<Double>(){
            @Override
            public Validator<Double>.Result validate(Double a) {
                  Result r = null;
                  if (a == Double.NEGATIVE_INFINITY) {
                        r = new Result(null, "Formato no correcto");
                  } else if (a <0.){
                        r = new Result(null, "Valor no correcto");
                  }
                  return r;
            }
      };

BindingListener listener = new BindingListener(){

      public void bindingBecameBound(Binding arg0) { }

      public void bindingBecameUnbound(Binding arg0) {}

      public void sourceChanged(Binding arg0, PropertyStateEvent a) {}

      public void syncFailed(Binding b, SyncFailure fallo) {
        mensaje.setText(fallo.getValidationResult().getDescription());
      }

      public void synced(Binding arg0) {
        mensaje.setText("Correcto");
      }
      public void targetChanged(Binding a, PropertyStateEvent b) { }
    };
```

4 Catálogo de componentes gráficos

JLabel: Componente adecuado para mostrar texto e imágenes. Las etiquetas pueden formar grupos.

Propiedades:

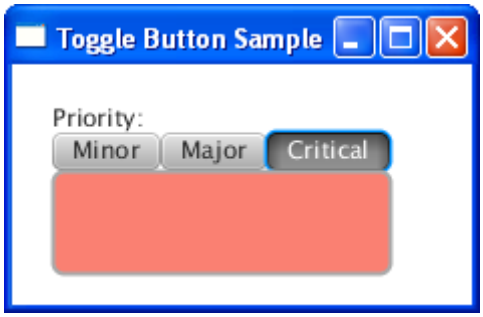
- *Text: String*. Texto que se muestra
- *Icon: Icon*. Icono que se muestra

JButton: Similar a una etiqueta pero con la capacidad adicional de poder asociar acciones a eventos (como ser presionado, etc)

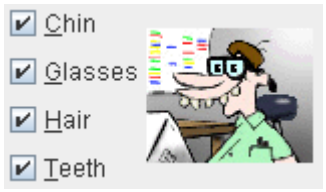
Propiedades:

- *Text: String*. Texto que se muestra
- *Icon: Icon*. Icono que se muestra

JToggleButton: Un botón con estado de presionado o no



JCheckBox: Elemento gráfico dónde se puede escoger una opción verdadera o falsa. En muchos casos se agregan varios elementos de este tipo para poder escoger a la vez varias alternativas.



Propiedades:

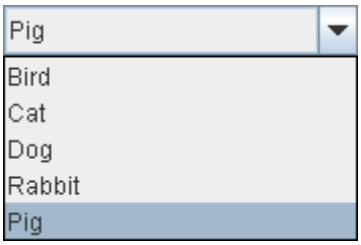
- *Text: String*. Texto que se muestra
- *Icon: Icon*. Icono que se muestra
- *Selected: Boolean.*

JRadioButton: Elemento gráfico, similar al anterior, dónde se puede escoger una opción verdadera o falsa. En muchos casos se agregan varios elementos de este tipo para poder escoger a la vez varias alternativas.

Propiedades:

- *Text: String*. Texto que se muestra
- *Icon: Icon*. Icono que se muestra
- *Selected: Boolean.*

JComboBox<E>: Componente adecuado para escoger una opción entre varias dadas a priori.



Propiedades

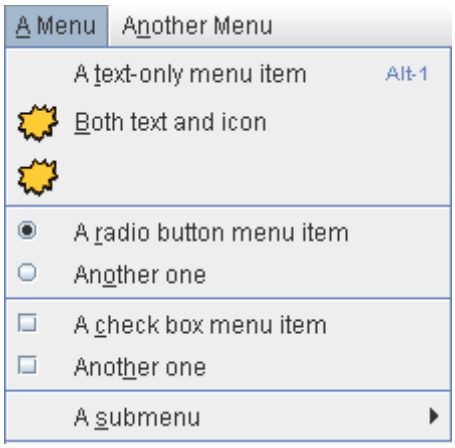
- *Model:* --. Lista de ítems entre los que se puede escoger
- *ItemAt(int p): E.* Ítem en la posición p.
- *SelectedItem: E.* El ítem seleccionado.
- *SelectedIndex: int.* Índice del ítem seleccionado

JList<E>: Componente que presenta una lista de ítem de entre se puede escoger uno o varios.

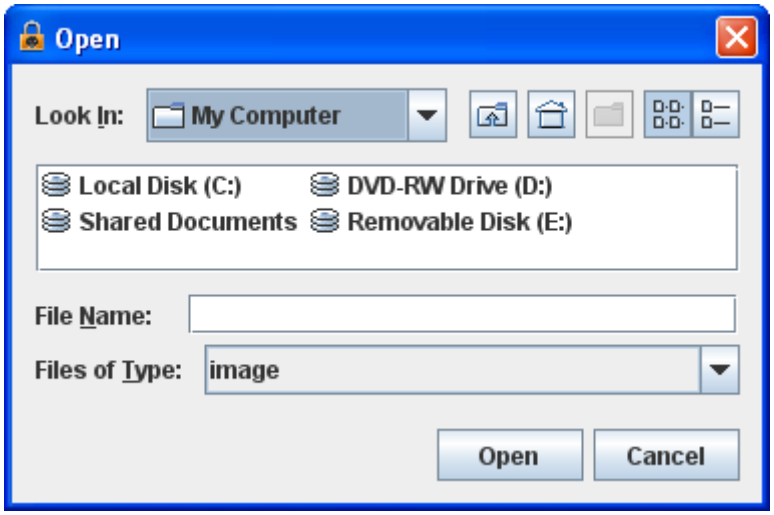
Propiedades

- *Model:* --. Lista de ítems entre los que se puede escoger
- *ItemAt(int p): E.* Ítem en la posición p.
- *SelectedValuesList: List<E>.* Lista de ítems seleccionados.
- *SelectedIndices: int[].* Índices de los ítems seleccionados

JMenu: Componente adecuado para escoger una alternativa entre varias alternativas y subalternativas. Un menú se construye como un agregado de componentes (ítems de menú, radio button, check box, etc.). Entre los componentes para construir el menú podemos usar otro menú.



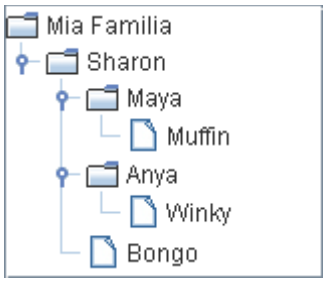
JFileChooser: Componente adecuado para escoger un fichero desde un conjunto de carpetas



JTable: Componente adecuado para visualizar tablas de datos. Cada columna puede tener un identificador y ser de un tipo. Existen un número dado de columnas y de filas. Los detalles sobre el número de columnas y sus etiquetas, el número de filas y sus contenidos pueden ser editados a través del modelo asociado al componente. Existe una fila seleccionada.

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

JTree: Componente adecuado para mostrar una jerarquía de datos. El componente dispone de un modelo que indica las etiquetas de los hijos y demás descendientes. Editando el modelo se edita la estructura del árbol. Un ítem del árbol puede ser seleccionado y un subárbol puede ser expandido.



JProgressBar: Componente adecuado para mostrar el progreso de una actividad.

JTextField: Componente con capacidad para editar una línea de texto. Se puede combinar con etiquetas.

Propiedades:

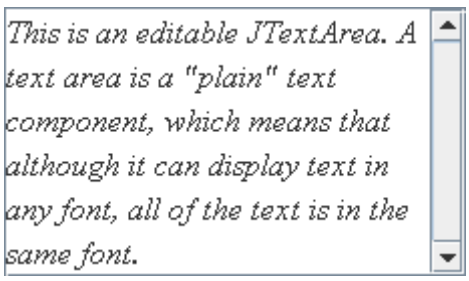
- *Text: String*. Texto que se muestra

JPasswordField: Componente con capacidad para editar una línea de texto sin que sea visible su contenido directamente. Se puede combinar con etiquetas.

Propiedades

- *Text: String*. Texto que se muestra

JTextArea: Componente adecuado para visualizar varias líneas de texto.



Propiedades

- *Text: String*. Texto que se muestra
- *Rows: int*. Número de filas
- *Columns*: Número de columnas

Métodos

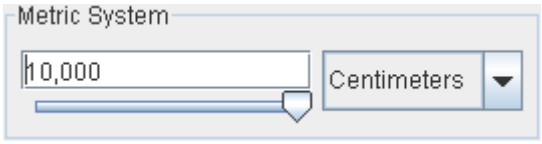
- *void append (String s):* Añade el texto al final del que se muestra en el componente.

5 Agregados de componentes gráficos

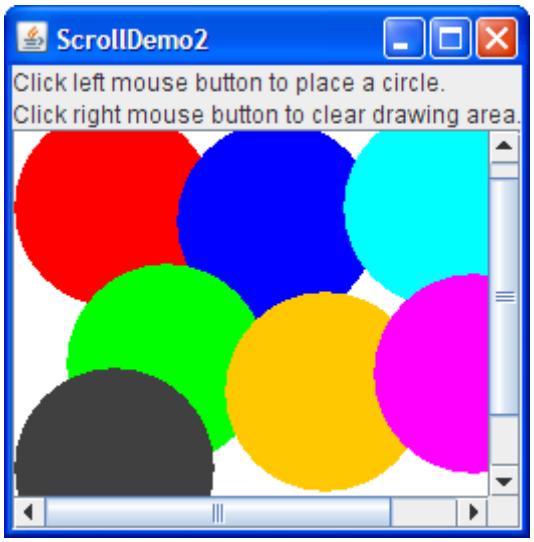
Una interfaz gráfica puede ser considerado como un conjunto de componentes gráficos básicos (como los vistos anteriormente) agrupados mediante contenedores de diversos tipos. Dentro de cada contenedor los componentes pueden tener varias formas de disponerse. Estas formas son denominadas *layout* (disposición del contenido) del contenedor. Un contenedor puede tener como componente otro contenedor.

Veamos los contenedores más usuales.

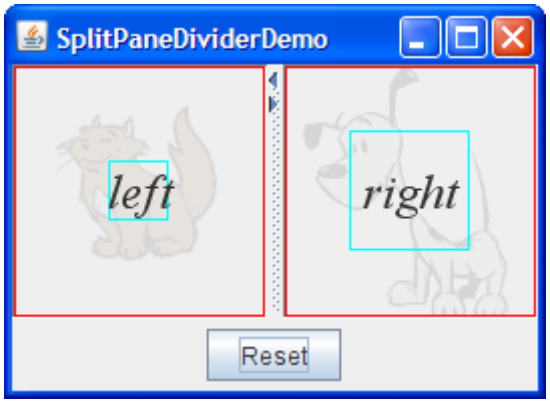
JPanel: Proporciona un contenedor genérico. Con él podemos agregar componentes y disponerlos de la forma que consideremos conveniente.



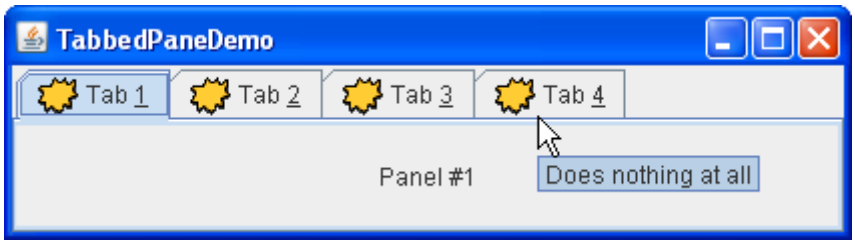
JScrollPane*:* Proporciona un contenedor con capacidad para desplazar la imagen que contiene



JSplitPane: Contenedor con dos partes que se pueden mostrar una al lado de otra o una sobre otra.

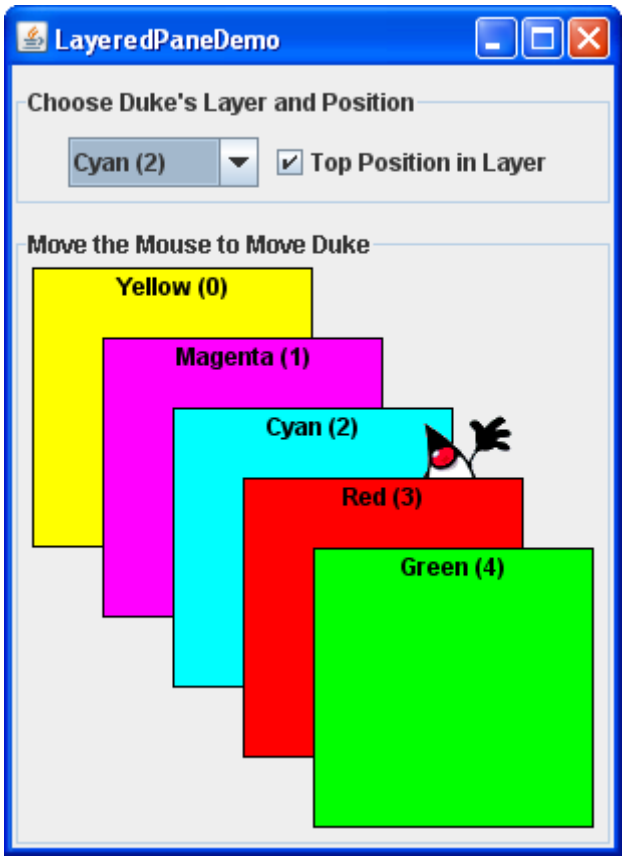


JTabbedPane: Un contenedor con varias partes que se muestran una sobre otra. El número de partes y sus etiquetas pueden ser editadas.



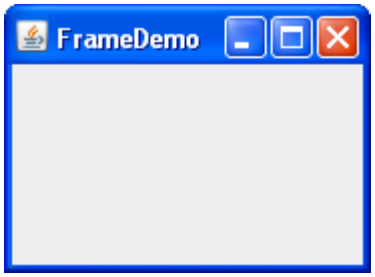
JToolBar: Un Contenedor que agrupa usualmente botones para llevar a cabo determinadas acciones asociadas

JLayeredPane: Un contedor que aporta una tercera dimensión para mostrar las partes de que consta.

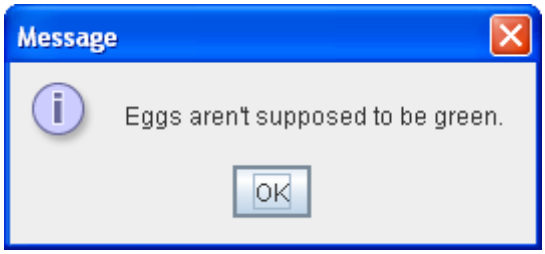


Por último tenemos los contenedores de primer nivel.

JFrame: Es un contenedor de primer nivel con un título y un borde



JDialog: Es un componente de primer nivel que nos permite crear ventanas temporales para mostrar una información o pedir una respuesta.

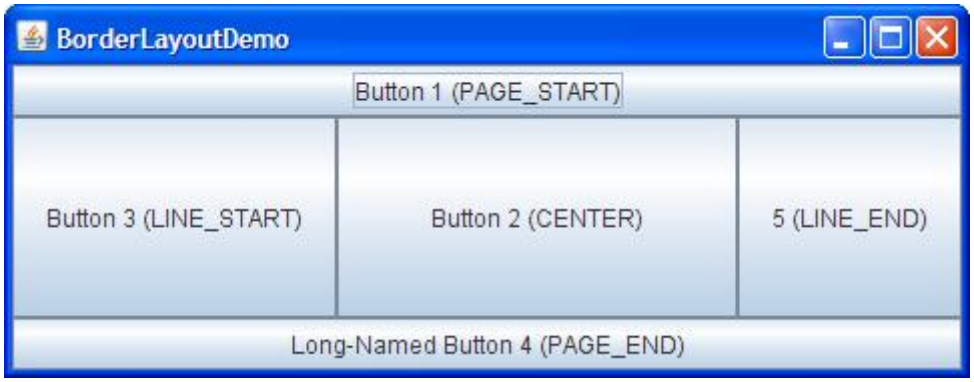


JAppelt: Un componente adecuado para mostrar imágenes en movimiento e interactivas.

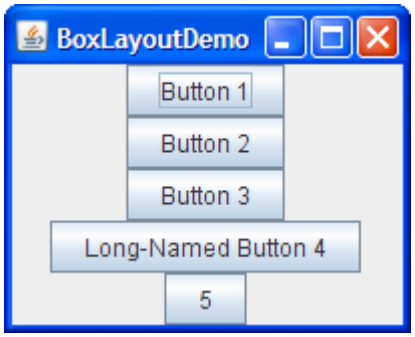
6 Layout

Dentro de cada contenedor los componentes se distribuyen según disposiciones especificadas que denominaremos **layout**. Algunas de estas disposiciones son:

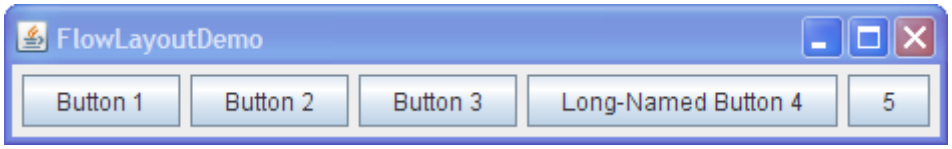
BorderLayout: Dispone los componentes en cinco áreas como se muestran en el ejemplo.



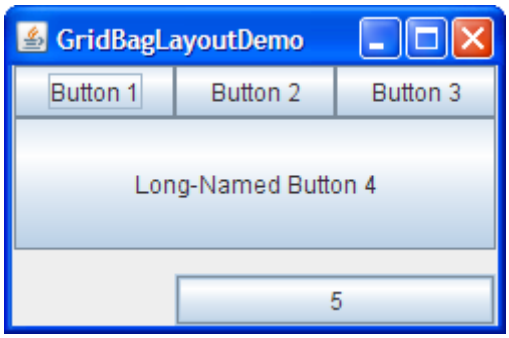
BoxLayout: Dispone los componentes en una columna o una fila. Respeta los tamaños requeridos por cada componente. Puede alinear los componentes.



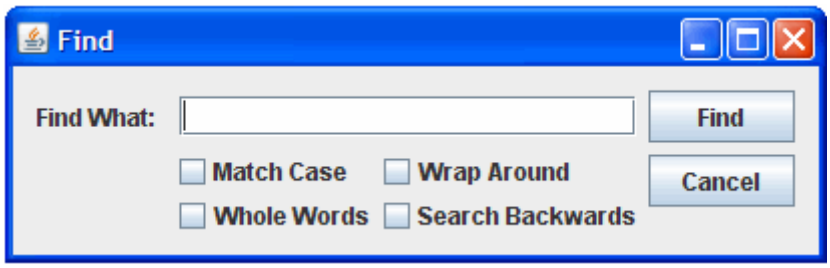
FlowLayout: Dispone los componentes en una fila uno detrás de otro y si no caben los pasa a la fila siguiente.



GridBagLayout: Es una disposición sofisticada y flexible. Dispone cada componente en el vértice de una rejilla de celdas. Es el más adecuado si usamos herramientas para diseñar y construir la interfaz gráfica.



GroupLayout: Dispone los componentes en grupos verticales y horizontales



7 Herramientas

Para construir interfaces gráficas es casi imprescindible el uso de herramientas. Estas nos permiten generar de forma cómoda los diversos componentes que componen la interfaz, estructurarlos agrupándolos en paneles, darles una determinada disposición dentro de un panel (*layout*), configurar algunas propiedades y establecer las ligaduras entre unas propiedades y otras.

Aunque hay otras herramientas disponibles aquí vamos a usar *WindowBuilder*

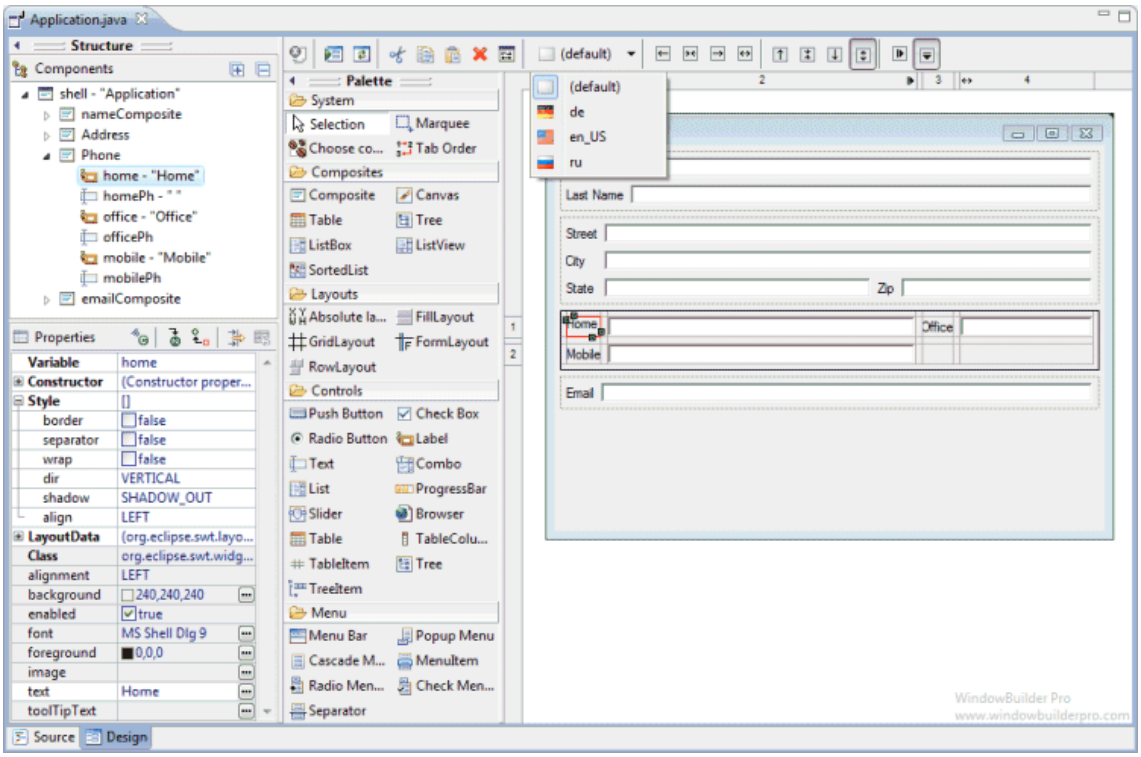
https://developers.google.com/java-dev-tools/wbpro/

Solo veremos los aspectos más relevantes de la herramienta y sólo para interfaces construidas con *Swing*.

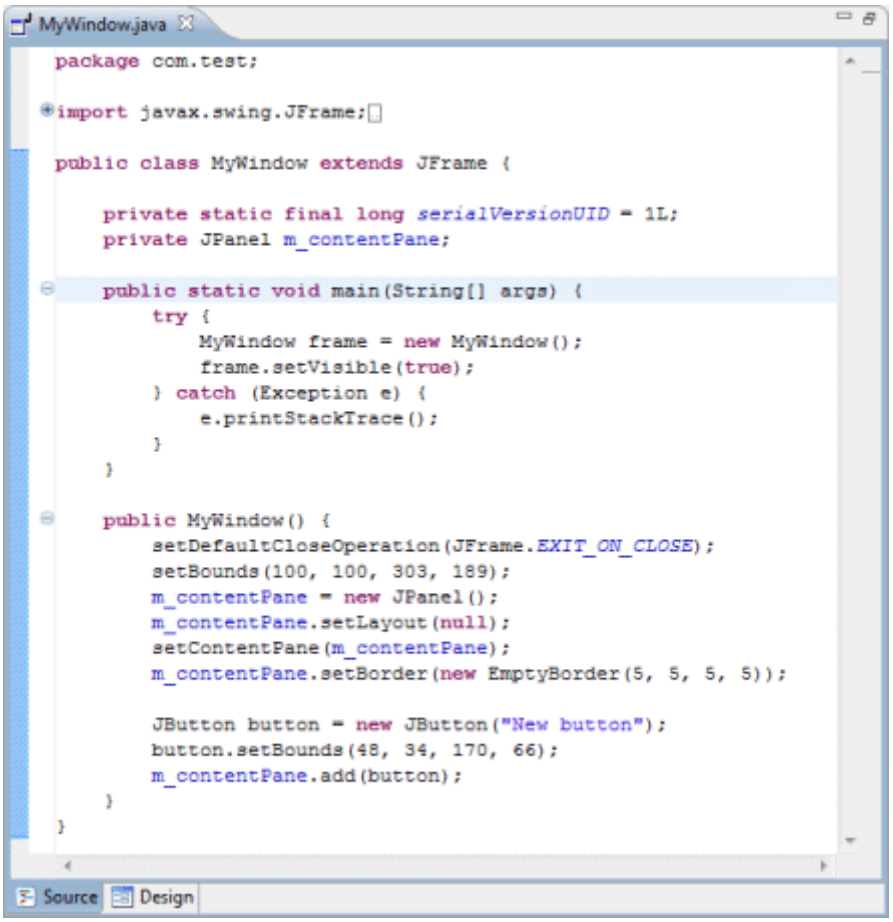
La herramienta tiene varias vistas. Una es la vista de diseño que nos permite añadir o eliminar componentes, fijar propiedades para los mismos, agruparlos mediante paneles, fijar el *layout*, y ver el aspecto resultante del diseño.

También es posible cambiar fácilmente el tipo de un componente, hacer que la variable que representa un componente sea un atributo en vez de una variable local, crear factorías específicas para componentes con propiedades especiales, etc.

Un ejemplo concreto es de la forma:

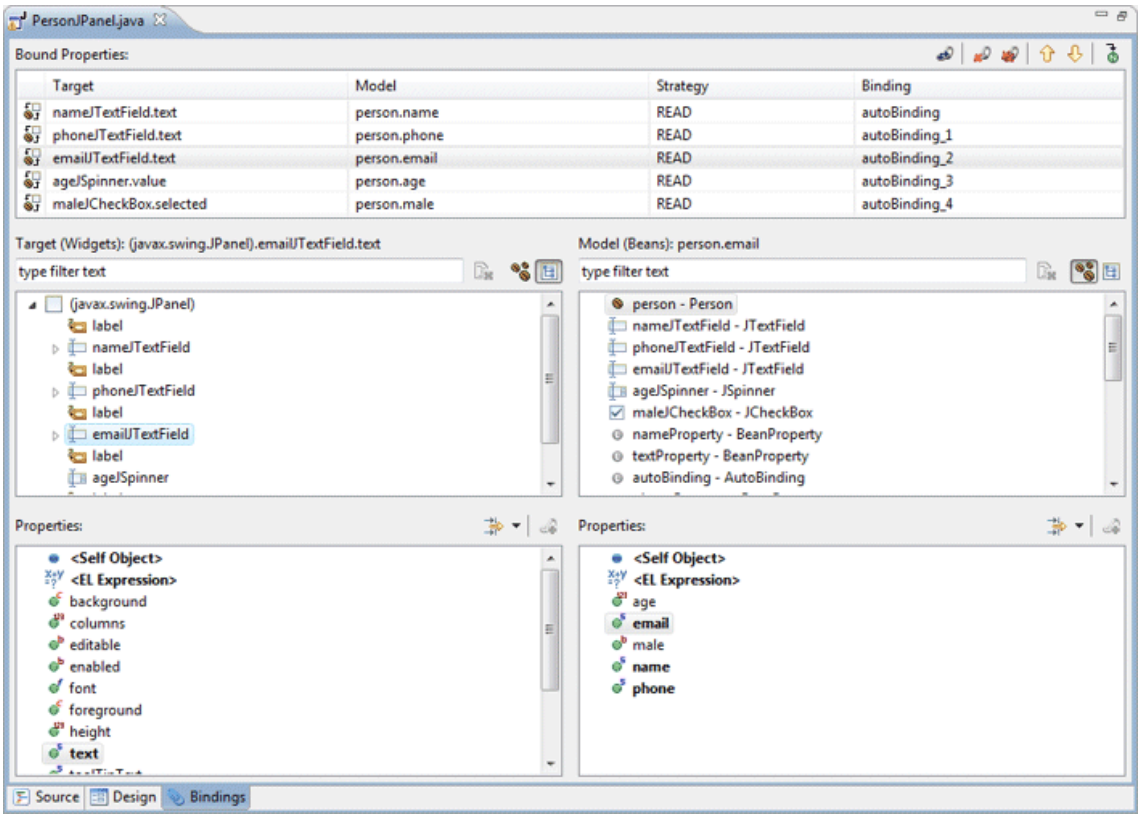


La vista del código fuente nos permite ver el código que ha generado la herramienta. Esta vista nos permite añadir código manualmente. Especialmente las acciones asociadas a los eventos de los diversos componentes etc.



Una tercera vista es la que nos permite construir automáticamente ligaduras entre las propiedades de distintos componentes.

Es la vista denominada *Binding*. Usando esta vista es posible generar código basado en los objetos *BeanProperty*, *ELProperty* y *AutoBinding* vistos más arriba.

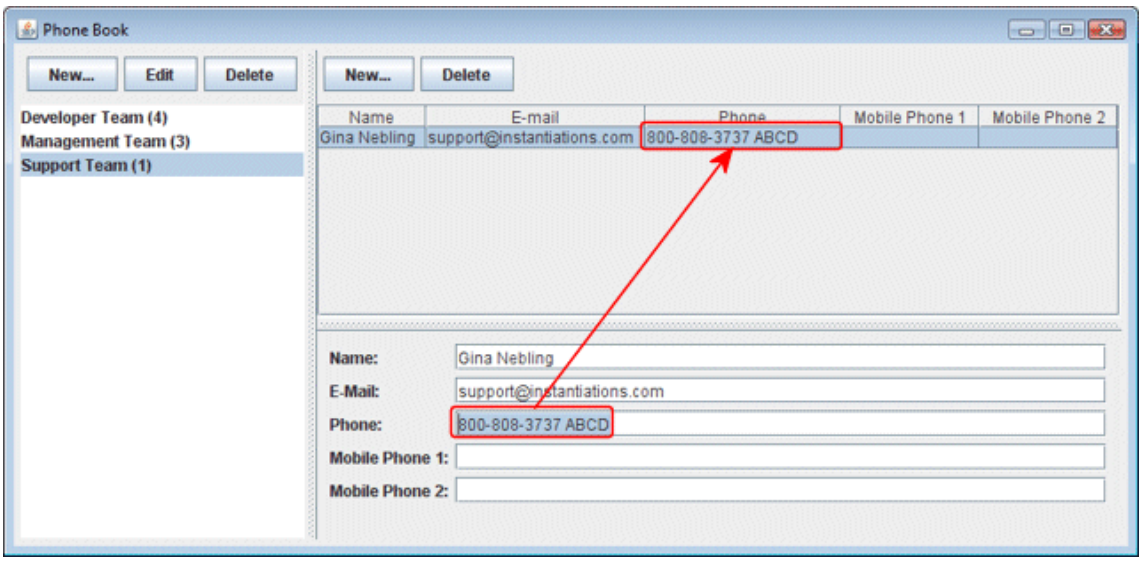


8 Un ejemplo

Como ejemplo veamos el que se muestra con la herramienta.

https://developers.google.com/java-dev-tools/wbpro/features/swing/data_binding/example

Se pretende diseñar una interfaz gráfica que se compone de dos partes separadas como se indica abajo. Ambas partes tienen una hilera de botones en la parte superior con distintas acciones asociadas.



Se trata de visualizar y editar varios grupos de personas, cada grupo compuesto por un número variable de las mismas y cada persona con un conjunto de propiedades que pueden ser editadas. Se pueden crear nuevos grupos, nuevas personas dentro de un grupo, editar la propiedad de una persona, eliminar un grupo o una persona.

La parte izquierda está asociada a los grupos. La parte derecha a un grupo de personas y una persona concreta seleccionada. En cada momento el grupo seleccionado en la parte izquierda aparecerá desplegado en la parte derecha y la persona seleccionada en la parte de abajo derecha para poder ser editada.

El código completo del ejemplo puede verse junto con la herramienta en:

https://developers.google.com/java-dev-tools/wbpro/features/swing/data_binding/example

Como resumen general podemos sugerir unos pasos para construir la interfaz y una estructura para la clase principal generada.

```
public class JPhoneBook extends JFrame {

      Declaración de variables del modelo;
      Declaración de variables gráficas;

      public static void create(final PhoneGroups g,
                                final List<String> names) {
           EventQueue.invokeLater(new Runnable() {
              public void run() {
                  try {
                    JPhoneBook frame = new JPhoneBook(g,names);
                    frame.setVisible(true);
                  } catch (Exception e) {
                    e.printStackTrace();
                  }
              }
           });
      }

      public void setDefaultValues(PhoneGroups g,List<String> names){
            Inicialización de variables del modelo;
            Código manual;
      }

      protected void initDataBindings() {
            Ligaduras entre variables;
            Código generado por la herramienta;
      }

      protected void saveVariablesModel(){
            Guarda el valor de las variables del modelo;
            Código manual;
            Acción a asociar a un botón de la interfaz gráfica;
```

```
    }
    public JPhoneBook(PhoneGroups g, List<String> names) {
          setDefaultValues(g,names);

          Inicialización de variables gráficas y código asociado a
          Eventos. Código generado por la herramienta más código
          asociado a manual asociado a los eventos.

          initDataBindings();
    }
```

Arriba hemos mostrado la estructura del código generado. Hay varios métodos que podemos usar siempre:

- *setDefaultValues(…):* Método encargado de inicializar las variables del modelo que aparecen en la clase. Se implementa manualmente.
- *initDataBindings():* Método para generar las ligaduras entre las variables (gráficas y del modelo). Se genera automáticamente por la herramienta en su vista *DataBinding.*
- *saveVariablesModel():* Método manual para guardar, si estamos interesados, las variables del modelo. La acción correspondiente hay que asociarla a un botón de la interfaz gráfica.

La secuencia de pasos a seguir es:

- Dentro del paquete correspondiente abrir uno de los elementos de alto nivel proporcionados por *WindowsBuilder (JFrame, JDialog, JApplet, …)*
- En la vista de diseño ir añadiendo los elementos gráficos necesarios agrupados en paneles, sus propiedades y los respectivos *layouts.*
- En la vista de código añadir las variables del modelo en la zona de declaración de variables e inicializarlas en el método *setDefaultValues.*
- En la vista de *Bindings* generar las ligaduras adecuadas.
- En la vista de diseño asociar el código que corresponda a los botones
- Crear métodos de factoría para los objetos de la clase que estamos diseñando. En este caso el método *create.*
- Escribir el código para guardar los valores de las variables del modelo si fuera necesario.
- En muchos casos es conveniente que la interfaz gráfica implemente un interface que será la vista que ofrecerá al resto del programa y que permita ignorar los detalles internos (elementos gráficos, eventos, ligaduras de variables, etc.). Esta interfaz ofrecerá métodos para gestionar las propiedades del modelo.
- Si es necesario realizar cálculos costosos en tiempo tenemos que tener en cuenta las posibilidades de *Swing* para usar varias hebras. Los detalles los veremos en el siguiente capítulo.

En este ejemplo concreto las variables del modelo (según se describe con detalle en el código citado) son:

```
private PhoneGroups m_groups = new PhoneGroups();
private List<String> m_names = new ArrayList<String>();
```

Es decir varios grupos de personas (de tipo *PhoneGroups*) y una lista de nombres disponibles para posibles nuevos grupos. Los objetos de tipo *PhoneGroups* tienen que ofrecer la posibilidad de ser *listenables* con las ideas vistas al principio del capítulo. La interfaz que ofrece al resto del código puede tener dos propiedades:

- *Categories: PhoneGroups*, Consultable y los valores iniciales proporcionados en el constructor.
- *NamesOfCategories: List<String>,* Consultable y los valores iniciales proporcionados en el constructor.

La interfaz puede ser entonces de la forma:

```
public interface PhoneBook {

	public PhoneGroups getCategories();

	public List<String> getNamesOfCategories();
}
```

Los tipos que se usan en esta interfaz y que debe ser implementados en el modelo son:

- *Person* con propiedades:
  - *Name: String*
  - *Email: String*
  - *Phone: String*
  - *MobilePhone1: String*
  - *MobilePhone2: String*
- *PhoneGroup* con propiedades:
  - *Name: String*
  - *Persons: List<Person>*
- *PhoneGroups* con propiedades:
  - *Groups: List<PhoneGroup>*

En resumen en el ejemplo se pretende proporcionar los valores iniciales para las propiedades *Categories* y *NameOfCategories.* La interfaz gráfica se ejecutará modificándose las categorías, las personas en ellas o los detalles de algunas personas. El valor final de esas propiedades podrá ser consultado a través de la interfaz anterior o guardado con la acción correspondiente.