Capitolo 18

JavaBeans

Andrea Gini

La programmazione a componenti

Uno degli obiettivi più ambiziosi dell'ingegneria del software è organizzare lo sviluppo di sistemi in maniera simile a quanto è stato fatto in altre branche dell'ingegneria, dove la presenza di un mercato di parti standard altamente riutilizzabili permette di aumentare la produttività riducendo nel contempo i costi. Nella meccanica, ad esempio, esiste da tempo un importante mercato di componenti riutilizzabili, come viti, dadi, bulloni e ruote dentate; ciascuno di questi componenti trova facilmente posto in centinaia di prodotti diversi.

L'industria del software, sempre più orientata alla filosofia dei componenti, sta dando vita a due nuove figure di programmatore: il progettista di componenti e l'assemblatore di applicazioni.

Il primo ha il compito di scoprire e progettare oggetti software di uso comune, che possano essere utilizzati con successo in contesti differenti. Produttori in concorrenza tra di loro possono realizzare componenti compatibili, ma con caratteristiche prestazionali differenti. L'acquirente può orientarsi su un mercato che offre una pluralità di scelte e decidere in base al budget o a particolari esigenze di prestazione.

L'assemblatore di applicazioni, d'altra parte, è un professionista specializzato in un particolare dominio applicativo, capace di creare programmi complessi acquistando sul mercato componenti standard e combinandoli con strumenti grafici o linguaggi di scripting.

Questo capitolo offre un'analisi approfondita delle problematiche che si incontrano nella creazione di componenti in Java; attraverso gli esempi verrà comunque offerta una panoramica su come sia possibile assemblare applicazioni complesse a partire da componenti concepiti per il riuso.

La specifica JavaBeans

JavaBeans è una specifica, ossia un insieme di regole seguendo le quali è possibile realizzare in Java componenti software riutilizzabili, che abbiano la capacità di interagire con altri componenti, realizzati da altri produttori, attraverso un protocollo di comunicazione comune.

Ogni Bean è caratterizzato dai servizi che è in grado di offrire e può essere utilizzato in un ambiente di sviluppo differente rispetto a quello in cui è stato realizzato. Quest'ultimo punto è cruciale nella filosofia dei componenti: sebbene i Java Beans siano a tutti gli effetti classi Java, e possano essere manipolati completamente per via programmatica, essi vengono spesso utilizzati in ambienti di sviluppo diversi, come tool grafici o linguaggi di scripting.

I tool grafici, tipo JBuilder, permettono di manipolare i componenti in maniera visuale. Un assemblatore di componenti può selezionare i Beans da una palette, inserirli in un apposito contenitore, impostarne le proprietà, collegare gli eventi di un Bean ai metodi di un altro, generando in tal modo applicazioni, Applet, Servlet e persino nuovi componenti senza scrivere una sola riga di codice.

I linguaggi di scripting, di contro, offrono una maggiore flessibilità rispetto ai tool grafici, senza presentare le complicazioni di un linguaggio generico. La programmazione di pagine web dinamiche, uno dei domini applicativi di maggior attualità, deve il suo rapido sviluppo a un'intelligente politica di stratificazione, che vede le funzionalità di più basso livello, come la gestione dei database, la Businnes Logic o l'interfacciamento con le risorse di sistema, incapsulate all'interno di JavaBeans, mentre tutto l'aspetto della presentazione viene sviluppato con un semplice linguaggio di scripting, tipo Java Server Pages o PHP.

Il modello a componenti JavaBeans

Un modello a componenti è caratterizzato da almeno sette fattori: proprietà, metodi, introspezione, personalizzazione, persistenza, eventi e modalità di deployment. Nei prossimi paragrafi si analizzerà il ruolo di ciascuno di questi aspetti all'interno della specifica Java Beans; quindi si procederà a descriverne l'implementazione in Java.

Proprietà

Le proprietà sono attributi privati, accessibili solamente attraverso appositi metodi `get` e `set`. Tali metodi costituiscono l'unica via di accesso pubblica alle proprietà, cosa che permette al progettista di componenti di stabilire per ogni parametro precise regole di accesso. Se si utilizzano i Bean all'interno di un programma di sviluppo visuale, le proprietà di un componente vengono visualizzate in un apposito pannello, che permette di modificarne il valore con un opportuno strumento grafico.

Metodi

I metodi di un Bean sono metodi pubblici Java, con l'unica differenza che essi risultano accessibili anche attraverso linguaggi di scripting e Builder Tools. I metodi sono la prima e più importante via d'accesso ai servizi di un Bean.

Introspezione

I Builder Tools scoprono i servizi di un Bean (proprietà, metodi ed eventi) attraverso un processo noto come introspezione, che consiste principalmente nell'interrogare il componente per conoscerne i metodi, e dedurre da questi le caratteristiche. Il progettista di componenti può attivare l'introspezione in due maniere: seguendo precise convenzioni nella formulazione delle firme dei metodi, o creando una speciale classe `BeanInfo`, che fornisce un elenco esplicito dei servizi di un particolare Bean.

La prima via è senza dubbio la più semplice: se si definiscono i metodi di accesso a un determinato servizio seguendo le regole di naming descritte dalla specifica JavaBeans, i tool grafici saranno in grado, grazie alla reflection, di individuare i servizi di un Bean semplicemente osservandone l'interfaccia di programmazione. Il ricorso ai BeanInfo, d'altro canto, torna utile in tutti quei casi in cui sia necessario mascherare alcuni metodi, in modo da esporre solamente un sottoinsieme dei servizi effettivi del Bean.

Personalizzazione

Durante il lavoro di composizione di Java Beans all'interno di un tool grafico, un apposito Property Sheet, generato al volo dal programma di composizione, mostra lo stato delle proprietà e permette di modificarle con un opportuno strumento grafico, tipo un campo di testo per valori `String` o una palette per proprietà `Color`. Simili strumenti grafici vengono detti editor di proprietà.

I tool grafici dispongono di editor di proprietà in grado di supportare i tipi Java più comuni, come i tipi numerici, le stringhe e i colori; nel caso si desideri rendere editabile una proprietà di un tipo diverso, è necessario realizzare un'opportuna classe di supporto, conforme all'interfaccia `PropertyEditor`. Quando invece si desideri fornire un controllo totale sulla configurazione di un Bean, è possibile definire un Bean Customizer, una speciale applicazione grafica specializzata nella configurazione di un particolare tipo di componenti.

Persistenza

La persistenza permette ad un Bean di salvare il proprio stato e di ripristinarlo in un secondo tempo. JavaBeans supporta la persistenza grazie all'Object Serialization, che permette di risolvere questo problema in modo molto rapido.

Eventi

Nella programmazione a oggetti tradizionale non esiste nessuna convenzione su come modellare lo scambio di messaggi tra oggetti. Ogni programmatore adotta un proprio sistema, creando una fitta rete di dipendenze che rende molto difficile il riutilizzo di oggetti in contesti differenti da quello di partenza. Gli oggetti Java progettati secondo la specifica Java Beans adottano un meccanismo di comunicazione basato sugli eventi, simile a quello utilizzato nei componenti grafici Swing e AWT. L'esistenza di un unico protocollo di comunicazione standard garantisce l'intercomunicabilità tra componenti, indipendentemente da chi li abbia prodotti.

Deployment

I JavaBeans possono essere consegnati, in gruppo o singolarmente, attraverso file JAR, speciali archivi compressi in grado di trasportare tutto quello di cui un Bean ha bisogno, come classi, immagini o altri file di supporto. Grazie ai file .jar è possibile consegnare i Beans con una modalità del tipo "chiavi in mano": l'acquirente deve solamente caricare un file JAR nel proprio ambiente di sviluppo e i Beans in esso contenuti verranno subito messi a disposizione. L'impacchettamento di classi Java all'interno di file JAR segue poche semplici regole, che verranno descritte negli esempi del capitolo.

Guida all'implementazione dei JavaBeans

Realizzare un componente Java Bean è un compito alla portata di qualunque programmatore Java che disponga di buone conoscenze di sviluppo Object Oriented. Nei paragrafi seguenti verranno descritte dettagliatamente le convenzioni di naming dettate dalla specifica, e verranno fornite le istruzioni su come scrivere le poche righe di codice necessarie a implementare i meccanismi che caratterizzano i servizi Bean. Infine verranno presentati degli esempi, che permetteranno di impratichirsi con il processo di implementazione delle specifiche.

Le proprietà

Le proprietà sono attributi che descrivono l'aspetto e il comportamento di un Bean, e che possono essere modificate durante tutto il ciclo di vita del componente. Di base, le proprietà sono attributi privati, ai quali si accede attraverso una coppia di metodi della forma:

```
public <PropertyType> get<PropertyName>()
public void set<PropertyName>(<PropertyType> property)
```

La convenzione di aggiungere il prefisso get e set ai metodi che forniscono l'accesso a una proprietà, permette ad esempio ai tool grafici di rilevare le proprietà Bean, determinarne le regole di accesso (Read Only o Read/Write), dedurne il tipo, visualizzare le proprietà su un apposito Property Sheet e individuare l'editor di proprietà più adatto al caso.

Se ad esempio un tool grafico scopre, grazie all'introspezione, la coppia di metodi

```
public Color getForegroundColor() { ... }
public void setForegroundColor(Color c) { ... }
```

da questi conclude che esiste una proprietà chiamata foregroundColor (notare la prima lettera minuscola), accessibile sia in lettura che in scrittura, di tipo Color. A questo punto, il tool può cercare un editor di proprietà per parametri di tipo Color, e mostrare la proprietà su un property sheet in modo che possa essere vista e manipolata dal programmatore.

Proprietà indicizzate (Indexed Property)

Le proprietà indicizzate permettono di gestire collezioni di valori accessibili attraverso indice, in maniera simile a come si fa con un vettore. Lo schema di composizione dei metodi di accesso di una proprietà indicizzata è il seguente:

```
public <PropertyType>[] get<PropertyName>();
public void set<PropertyName>(<PropertyType>[] value);
```

per i metodi che permettono di manipolare l'intera collection, mentre per accedere ai singoli elementi, si deve predisporre una coppia di metodi del tipo:

```
public <PropertyType> get<PropertyName>(int index);
public void set<PropertyName>(int index, <PropertyType> value);
```

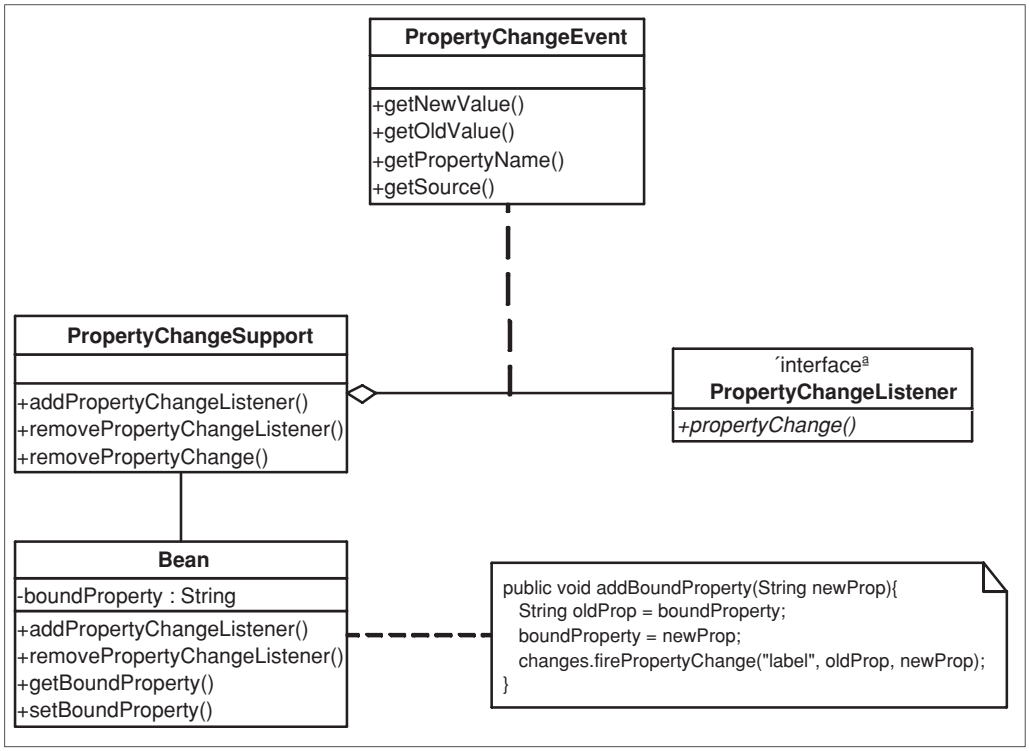
Proprietà bound

Le proprietà semplici, così come sono state descritte nei paragrafi precedenti, seguono una convenzione radicata da tempo nella normale programmazione a oggetti. Le proprietà bound, al contrario, sono caratteristiche dell'universo dei componenti, dove si verifica molto spesso la necessità di collegare il valore delle proprietà di un componente a quelli di un'altro, in modo tale che si mantengano aggiornati. I metodi set delle proprietà bound, inviano una notifica a tutti gli ascoltatori registrati ogni qualvolta viene alterato il valore della proprietà. Il meccanismo di ascolto–notifica, simile a quello degli eventi Swing e AWT, segue il pattern Observer.

Le proprietà bound, a differenza degli eventi Swing, utilizzano un unico tipo di evento, ChangeEvent, cosa che semplifica il processo di implementazione. La classe PropertyChangeSupport, presente all'interno del package java.bean, fornisce i metodi che gestiscono la lista degli ascoltatori e quelli che producono l'invio degli eventi.

Un oggetto che voglia mettersi in ascolto di una proprietà, deve implementare l'interfaccia PropertyChangeListener e deve registrarsi presso la sorgente di eventi. L'oggetto PropertyChangeEvent incapsula le informazioni riguardo alla proprietà modificata, alla sorgente e al valore della proprietà.

Figura 18.1 – *Il meccanismo di notifica di eventi bound segue il pattern Observer*



Come implementare il supporto alle proprietà bound

Per aggiungere a un Bean il supporto alle proprietà bound, bisogna anzitutto importare il package java.beans.*, in modo da garantire l'accesso alle classi PropertyChange-Support e PropertyChangeEvent. Quindi bisogna creare un oggetto PropertyChangeSupport, che ha il compito di mantenere la lista degli ascoltatori e di fornire i metodi che gestiscono l'invio degli eventi.

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

A questo punto bisogna realizzare, nella propria classe, i metodi che permettono di gestire la lista degli ascoltatori. Tali metodi sono dei semplici metodi Wrapper che fanno riferimento a metodi con la stessa firma, presenti nel PropertyChangeSupport:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
 changes.addPropertyChangeListener(l);
}
```

```
public void removePropertyChangeListener(PropertyChangeListener l) {
 changes.removePropertyChangeListener(l);
}
```

La presenza dei metodi addPropertyChangeListener() e removeProperty-ChangeListener() permette ai tool grafici di riconoscere un oggetto in grado di inviare proprietà bound e di mettere a disposizione un'opportuna voce nel menù di gestione degli eventi.

L'ultimo passaggio consiste nel modificare i metodi set relativi alle proprietà che si vuole rendere bound, per fare in modo che venga generato un PropertyChangeEvent ogni volta che la proprietà viene reimpostata

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 changes.firePropertyChange("color", oldColor , newColor);
}
```

Nel caso di proprietà read only, prive di metodo set, l'invio dell'evento dovrà avvenire all'interno del metodo che attua la modifica della proprietà. Un aspetto interessante del meccanismo di invio di PropertyChangeEvent, è che essi trasportano sia il nuovo valore che quello vecchio. Questa scelta dispensa chi implementa un ascoltatore dal compito di mantenere una copia del valore, qualora questo fosse necessario, dal momento che l'evento viene propagato *dopo* la modifica della relativa proprietà. Il metodo fireChangeEvent() della classe PropertyChangeListener fornisce il servizio di Event Dispatching:

```
firePropertyChange(String propertyName, Object oldValue, Object newValue)
```

In pratica esso impacchetta i parametri in un oggetto PropertyChangeEvent, e chiama il metodo propertyChange(PropertyChangeEvent p) su tutti gli ascoltatori registrati. I parametri vengono trattati come Object, e nel caso si debbano inviare proprietà espresse in termini di tipi primitivi, occorre incapsularle nell'opportuno Wrapper (Integer per valori int, Double per valori double e così via). Per facilitare questo compito, la classe propertyChangeSupport prevede delle varianti di firePropertyChange per valori int e boolean.

Come implementare il supporto alle proprietà bound su sottoclassi di JComponent

La classe JComponent, superclasse di tutti i componenti Swing, dispone del supporto nativo alla gestione di proprietà bound. Di base essa fornisce i metodi addProperty-ChangeListener e removePropertyChangeListener, oltre a una collezione di metodi firePropertyChange adatta ad ogni tipo primitivo. In questo caso l'implementazione di una proprietà bound richiederà solo una modifica al metodo set preposto, similmente a come descritto nell'ultimo passaggio del precedente paragrafo, con la differenza che non è necessario ricorrere a un oggetto propertyChangeSupport per inviare la proprietà:

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 firePropertyChange("color", oldColor , newColor);
}
```

Ascoltatori di proprietà

Se si desidera mettersi in ascolto di una proprietà, occorre definire un opportuno oggetto PropertyChangeListener e registrarlo presso il Bean. Un Property-ChangeListener deve definire il metodo propertyChange(PropertyChangeEvent e), che viene chiamato quando avviene la modifica di una proprietà bound.

Un PropertyChangeListener viene notificato quando avviene la modifica di *una qualunque* proprietà bound: per questa ragione esso deve, come prima cosa, verificare, che la proprietà appena modificata sia quella alla quale si è interessati. Una simile verifica richiede una chiamata al metodo getPropertyName di PropertyChangeEvent, che restituisce il nome della proprietà. Per convenzione, i nomi di proprietà vengono estratti dai nomi dichiarati nei metodi get e set, con la prima lettera minuscola. Il seguente frammento di codice presenta un tipico PropertyChange-Listener, che ascolta la proprietà foregroundColor:

```
public class Listener implements PropertyChangeListener() {
 public void propertyChange(PropertyChangeEvent e) {
    if(e.getPropertyName().equals("foregroundColor"))
       System.out.println(e.getNewValue());
 }
}
```

Un esempio di Bean con proprietà bound

Un Java Bean rappresenta un mattone di un programma. Ogni componente è un'unità di utilizzo abbastanza grossa da incorporare una funzionalità evoluta, ma piccola rispetto ad un programma fatto e finito. Il concetto del riuso può essere presente a diversi livelli del progetto: il seguente Bean fornisce un esempio di elevata versatilità

Il Bean PhotoAlbum è un pannello grafico al cui interno vengono caricate delle immagini. Il metodo showNext() permette di passare da un'immagine all'altra, in modo ciclico. Il numero ed il tipo di immagini viene determinato al momento dell'avvio: durante la fase di costruzione viene letto il file comment.txt, presente nella directory images, che contiene una riga di commento per ogni immagine presente nella cartella. Le immagini devono essere nominate in modo progressivo (img0.jpg, img1.jpg, img2.jpg…) e devono essere presenti in numero uguale alle righe del file comment.txt. Questa scelta progettuale consente di introdurre il riuso a un livello abbastanza alto: qualunque utente, anche con scarse conoscenze del linguaggio, può personalizzare il componente, inserendo le sue foto preferite, senza la necessità di alterare il codice sorgente.

Il Bean PhotoAlbum ha tre proprietà:

- imageNumber, che restituisce il numero di immagini contenute nell'album. Essendo una quantità immutabile, tale proprietà è stata implementata come proprietà semplice.

- imageIndex: restituisce l'indice dell'immagine attualmente visualizzata. Al cambio di immagine viene inviato un PropertyChangeEvent.

- imageComment: restituisce una stringa di commento all'immagine. Anche in questo caso, al cambio di immagine viene generato un PropertyChangeEvent.

Il Bean viene definito come sottoclasse di JPanel: per questo motivo non vengono dichiarati i metodi addPropertyChangeListener e removeProperty-ChangeListener, già presenti nella superclasse. L'invio delle proprietà verrà messo in atto grazie al metodo firePropertyChange di JComponent.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class PhotoAlbum extends JPanel {

 private Vector comments = new Vector();
 private int imageIndex;

 public PhotoAlbum() {
    super();
    setLayout(new BorderLayout());
    setupComments();
    imageIndex = 0;
    showNext();
 }
 private void setupComments() {
    try {
       URL indexUrl = getClass().getResource("images/" + "comments.txt");
       InputStream in = indexUrl.openStream();
       BufferedReader lineReader = new BufferedReader(new InputStreamReader(in));
       String line;
       while((line = lineReader.readLine())!=null)
          comments.add(line);
    }
    catch(Exception e) {
       e.printStackTrace();
```

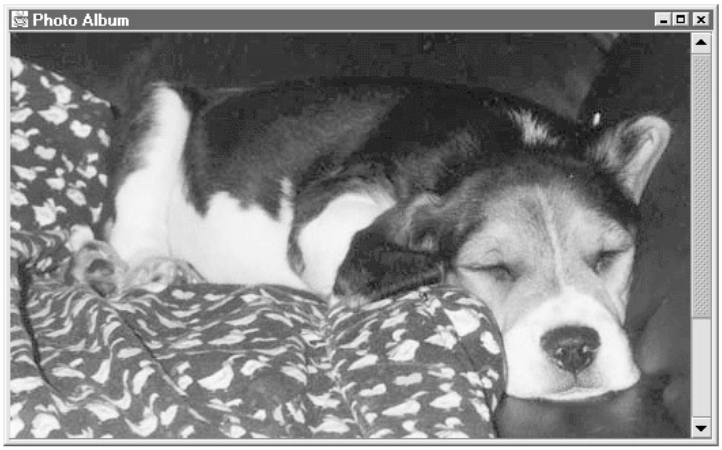
```
        }
    }
    public int getImageNumber() {
        return comments.size();
    }
    public int getImageIndex() {
        return imageIndex;
    }
    public String getImageComment() {
        return (String)comments.elementAt(imageIndex);
    }
    public void showNext() {
        int oldImageIndex = imageIndex;
        imageIndex = ((imageIndex +1 ) % comments.size());
        String imageName = «img» + Integer.toString(imageIndex) + «.jpg»;
        showImage(getClass().getResource(„images/“ + imageName));
        String oldImageComment = (String)comments.elementAt(oldImageIndex);
        String currentImageComment = (String)comments.elementAt(imageIndex);
        firePropertyChange(“imageComment”, oldImageComment, currentImageComment);
        firePropertyChange(“imageIndex”, oldImageIndex, imageIndex);
    }
    private void showImage(URL imageUrl) {
        ImageIcon img = new ImageIcon(imageUrl);
        JLabel picture = new JLabel(img);
        JScrollPane pictureScrollPane = new JScrollPane(picture);
        removeAll();
        add(BorderLayout.CENTER,pictureScrollPane);
        validate();
    }
}
```

È possibile testare il Bean come fosse una normale classe Java, utilizzando queste semplici righe di codice:

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import com.mokabyte.mokabook.javaBeans.photoAlbum.*;
import java.beans.*;
import javax.swing.*;

public class PhotoAlbumTest {
public static void main(String argv[]) {
    JFrame f = new JFrame(“Photo Album”);
    PhotoAlbum p = new PhotoAlbum();
    f.getContentPane().add(p);
    p.addPropertyChangeListener(new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent e) {
```

```
            System.out.println(e.getPropertyName() + ": " + e.getNewValue());
        }
    });
f.setSize(500,400);
f.setVisible(true);

while(true)
    for(int i=0;i<7;i++) {
        p.showNext();
        try {Thread.sleep(1000);}catch(Exception e) {}
    }
}
}
```

Figura 18.2 – *Un programma di prova per il Bean PhotoAlbum*



Creazione di un file JAR

Prima di procedere alla consegna del Bean entro un file JAR, bisogna anzitutto compilare le classi PhotoAlbum.java e PhotoAlbumTest.java, che devono trovarsi nella cartella com\mokabyte\mokabook\javaBeans\

```
javac com\mokabyte\mokabook\javaBeans\photoAlbum\*.java
```

A questo punto bisogna creare, ricorrendo a un semplice editor di testo tipo Notepad, un file photoAlbumManifest.tmp con il seguente contenuto

```
Main-Class: com.mokabyte.mokabook.javaBeans.photoAlbum.PhotoAlbumTest
```

```
Name: com/mokabyte/mokabook/javaBeans/photoAlbum/PhotoAlbum.class
Java-Bean: True
```

Le prime due righe, opzionali, segnalano la presenza di una classe dotata di metodo main.

Le ultime due righe del file manifest specificano che la classe PhotoAlbum.class è un Java Bean. Se l'archivio contiene più di un Bean, è necessario elencarli tutti.

Per generare l'archivio photoAlbum.jar, bisogna digitare la riga di comando:

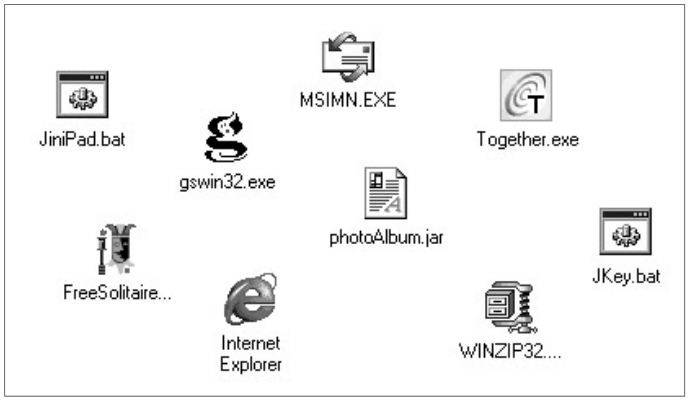
```
jar cfm photoAlbum.jar photoAlbumManifest.tmp
 com\mokabyte\mokabook\javaBeans\photoAlbum\*.class
 com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

Il file così generato contiene tutte le classi e le immagini necessarie a dar vita al Bean PhotoAlbum. Tale file potrà essere utilizzato facilmente all'interno di tool grafici o di pagine web, racchiuso dentro una Applet.

Il file .jar potrà essere avviato digitando

```
java PhotoAlbum.jar
```

Figura 18.3 – *Un file JAR opportunamente confezionato può essere aperto con un opportuno tool come Jar o WinZip*



Le istruzioni fornite sono valide per la piattaforma Windows. Su piattaforma Unix, le eventuali occorrenze del simbolo "\", che funge da path separator su piattaforme Windows, andranno sostituite col simbolo "/". Le convenzioni adottate all'interno del file manifest valgono invece su entrambe le piattaforme.

Integrazione con altri Bean

Nonostante il Bean PhotoAlbum fornisca un servizio abbastanza evoluto, non è ancora classificabile come applicazione. Esso, opportunamente integrato con altri Beans, può comunque dar vita a numerosi programmi; di seguito, ecco qualche esempio: collegato a un CalendarBean, PhotoAlbum può dar vita a un simpatico calendario elettronico; collegando un bottone Bean al metodo showNext() è possibile creare un album interattivo, impacchettarlo su un'Applet e pubblicarlo su Internet; impacchettando il Bean PhotoAlbum con foto natalizie, e collegandolo con un Bean Carillon, si può ottenere un biglietto di auguri elettronico.

Figura 18.4 – *Combinando, all'interno del Bean Box, il Bean PhotoAlbum con un pulsante Bean, si ottiene una piccola applicazione*

A questi esempi se ne possono facilmente aggiungere altri; altri ancora diventano possibili aggiungendo al Bean nuovi metodi, come previousImage() e setImageAt(int i); un compito ormai alla portata del lettore che fornisce un ottimo pretesto per esercitarsi.

Eventi Bean

La notifica del cambiamento di valore delle proprietà bound è un meccanismo di comunicazione tra Beans. Se si vuole che un Bean sia in grado di propagare eventi di tipo più generico, o comunque eventi che non è comodo rappresentare come un cambiamento di stato, è possibile utilizzare un meccanismo di eventi generico, del tutto simile a quello pre-

sente nei componenti grafici Swing e AWT. I prossimi paragrafi servono a illustrare le tre fasi dell'implementazione: creazione dell'evento, definizione dell'ascoltatore e infine creazione della sorgente di eventi.

Creazione di un evento

Per implementare un meccanismo di comunicazione basato su eventi, occorre anzitutto definire un'opportuna sottoclasse di EventObject, che racchiuda tutte le informazioni relative all'evento da propagare.

```
public class <EventType> extends EventObject {
 private <ParamType> param
 public <EventType>(Object source,<ParamType> param) {
    super(source);
    this.param = param;
 }
 public <ParamType> getParameter() {
    return param;
 }
}
```

La principale variazione sul tema si ha sul numero e sul tipo di parametri: tanto più complesso è l'evento da descrivere, maggiori saranno i parametri in gioco. L'unico parametro che è obbligatorio fornire è un reference all'oggetto che ha generato l'evento: tale reference, richiamabile con il metodo getSource() della classe EventObject, permetterà all'ascoltatore di interrogare la sorgente degli eventi qualora ce ne fosse bisogno.

Destinatari di eventi

Il secondo passaggio è quello di definire l'interfaccia di programmazione degli ascoltatori di eventi. Tale interfaccia deve essere definita come sottoclasse di EventListener, per essere riconoscibile come ascoltatore dall'Introspector. Lo schema di sviluppo degli ascoltatori segue lo schema

```
import java.awt.event.*;

public Interface <EventListener> extends EventListener {
 public void <EventType>Performed(<EventType> e);
}
```

Le convenzioni di naming dei metodi dell'interfaccia non seguono uno schema standard: la convenzione descritta nell'esempio, <EventType>performed, può essere seguita o meno. L'importante è che il nome dei metodi dell'interfaccia Listener suggeriscano il tipo di azione sottostante, e che accettino come parametro un evento del tipo giusto.

Sorgenti di eventi

Se si desidera aggiungere a un Bean la capacità di generare eventi, occorre implementare una coppia di metodi

```
add<EventListenerType>(<EventListenerType> l)
remove<EventListenerType>(<EventListenerType> l)
```

La gestione della lista degli ascoltatori e l'invio degli eventi segue una formula standard, descritta nelle righe seguenti:

```
private Vector listeners = new Vector();

public void add<EventListenerType>(<EventListenerType> l) {
 listeners.add(l);
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listeners.remove(l);
}
protected void fire<Eventype>(<EvenType> e) {
 Enumeration listenersEnumeration = listeners.elements();
 while(listenersEnumeration.hasMoreElements()) {
 <EventListenerType> listener = (<EventListenerType>)listenersEnumeration.nextElement();
    listener.<EventType>Performed(e);
 }
}
```

Sorgenti unicast

In alcuni casi occorre definire sorgenti di eventi capaci di servire un unico ascoltatore. Per implementare tali classi, che fungono da sorgenti unicast, si può seguire il seguente modello

```
private <EventListenerType> listener;

public void add<EventListenerType>(
<EventListenerType> l) throws TooManyListenersException {
 if(listener == null)
    listener = l;
 else
    throw new TooManyListenerException();
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listener = null;
}
protected void fire<Eventype>(<EvenType> e) {
```

```
if(listener! = null)
    listener.<EventType>Performed(e);
}
```

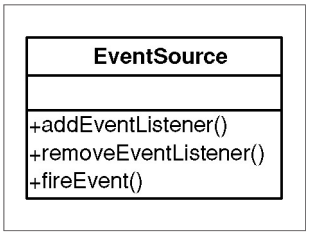
Ascoltatori di eventi: Event Adapter

Se si vuole che un evento generato da un Bean scateni un'azione su un altro Bean, è necessario creare un oggetto che realizzi un collegamento tra i due. Tale classe, detta Adapter, viene registrata come ascoltatore presso la sorgente dell'evento, e formula una chiamata al metodo destinazione ogni volta che riceve una notifica dal Bean sorgente.

Gli strumenti grafici tipo JBuilder generano questo tipo di classi in maniera automatica: tutto quello che l'utente deve fare è collegare, con pochi click di mouse, l'evento di un Bean sorgente a un metodo di un Bean target. Qui di seguito viene riportato il codice di un Adapter, generato automaticamente dal Bean Box, che collega la pressione di un pulsante al metodo startJuggling(ActionEvent e) del Bean Juggler.

```
// Automatically generated event hookup file.

public class ___Hookup_172935aa26 implements     java.awt.event.ActionListener, java.io.Serializable {

 public void setTarget(sunw.demo.juggler.Juggler t) {
    target = t;
 }

 public void actionPerformed(java.awt.event.ActionEvent arg0) {
    target.startJuggling(arg0);
 }

 private sunw.demo.juggler.Juggler target;
}
```

Figura 18.5 – *Un Adapter funge da ponte di collegamento tra gli eventi di un Bean e i metodi di un altro*



Un esempio di Bean con eventi

Il prossimo esempio è un Bean Timer, che ha il compito di generare battiti di orologio a intervalli regolari. Questo componente è un tipico esempio di Bean non grafico.

La prima classe che si definisce è quella che implementa il tipo di evento

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerEvent extends EventObject implements Serializable {

 public TimerEvent(Object source) {
    super(source);
 }
}
```

Come si può vedere, l'implementazione di un nuovo tipo di evento è questione di poche righe di codice. L'unico particolare degno di nota è che il costruttore del nuovo tipo di evento deve invocare il costruttore della superclasse, passando un reference alla sorgente dell'evento.

L'interfaccia che rappresenta l'ascoltatore deve estendere l'interfaccia EventListener; a parte questo, al suo interno si può definire un numero arbitrario di metodi, la cui unica costante è quella di avere come parametro un reference all'evento da propagare.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public interface TimerListener extends java.util.EventListener {
 public void clockTicked(TimerEvent e);
}
```

Per finire, ecco il Bean vero e proprio. Come si può notare, esso implementa l'interfaccia Serializable che rende possibile la serializzazione.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerBean implements Serializable {
```

```
private int time = 1000;
private transient TimerThread timerThread;
private Vector timerListeners = new Vector();

public void addTimerListener(TimerListener t) {
 timerListeners.add(t);
}
public void removeTimerListener(TimerListener t) {
 timerListeners.remove(t);
}
protected void fireTimerEvent(TimerEvent e) {
 Enumeration listeners = timerListeners.elements();
 while(listeners.hasMoreElements())
    ((TimerListener)listeners.nextElement()).clockTicked(e);
}
public synchronized void setMillis(int millis) {
 time = millis;
}
public synchronized int getMillis() {
 return time;
}
public synchronized void startTimer() {
 if(timerThread!=null)
    forceTick();
 timerThread = new TimerThread();
 timerThread.start();
}
public synchronized void stopTimer() {
 if(timerThread == null)
    return;

 timerThread.killTimer();
 timerThread = null;
}
public synchronized void forceTick() {
 if(timerThread!=null) {
    stopTimer();
    startTimer();
 }
 else
    fireTimerEvent(new TimerEvent(this));
}

 class TimerThread extends Thread {
    private boolean running = true;

    public synchronized void killTimer() {
```

```
        running = false;
    }
    private synchronized boolean isRunning() {
        return running;
    }
    public void run() {
        while(true)
            try {
                if(isRunning()) {
                    fireTimerEvent(new TimerEvent(TimerBean.this));
                    Thread.sleep(getMillis());
                }
                else
                    break;
        catch(InterruptedException e) {}
        }
    }
 }
}
```

I primi tre metodi servono a gestire la lista degli ascoltatori. Il terzo e il quarto gestiscono la proprietà millis, ossia il tempo, in millisecondi, tra un tick e l'altro. I due metodi successivi, startTimer, stopTimer, servono ad avviare e fermare il timer, mentre forceTick lancia un tick e riavvia il timer, se questo è attivo. Il timer vero e proprio viene implementato grazie a una classe interna TimerThread, sottoclasse di Thread. Si noti il metodo killTimer, che permette di terminare in modo pulito la vita del thread: questa soluzione è da preferire al metodo stop (deprecato a partire dal JDK 1.1), che in certi casi può provocare la terminazione del thread in uno stato inconsistente.

Per compilare le classi del Bean, bisogna usare la seguente riga di comando

```
javac com\mokabyte\mokabook\javaBeans\timer\*.java
```

Per impacchettare il Bean in un file .jar, è necessario per prima cosa creare con un editor di testo il file timerManifest.tmp, con le seguenti righe

```
Name: com/mokabyte/mokabook/javaBeans/timer/TimerBean.class
Java-Bean: True
```

Per creare l'archivio si deve quindi digitare il seguente comando

```
jar cfm timer.jar timerManifest.tmp com\mokabyte\mokabook\javaBeans\timer\*.class
```

Per testare la classe TimerBean, si può usare il seguente programma, che crea un oggetto TimerBean e registra un TimerListener il quale stampa a video una scritta ad ogni tick del timer.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public class TimerTest {
 public static void main(String argv[]) {


TimerBean t = new TimerBean();
    t.addTimerListener(new TimerListener() {
      public void clockTicked(TimerEvent e) {
        System.out.println("Tick");
      }
    });
    t.startTimer();
 }
}
```

Introspezione: l'interfaccia BeanInfo

Le convenzioni di naming descritte nei paragrafi precedenti permettono ai tool grafici abilitati ai Beans di scoprire i servizi di un componente grazie alla reflection. Questo processo automatico è certamente comodo, ma ha il difetto di non offrire nessun tipo di controllo sul numero e sul tipo di servizi da mostrare. In alcune occasioni può essere necessario mascherare un certo numero di servizi, specie quelli ereditati da una superclasse.

I Beans creati a partire dalla classe JComponent, ad esempio, ereditano automaticamente più di dieci attributi (dimensioni, colore, allineamento…) e ben dodici tipi diversi di evento (ComponentEvent, MouseEvent, HierarchyEvent…). Un simile eccesso provoca di solito disorientamento nell'utente; in questi casi è preferibile fornire un elenco esplicito dei servizi da associare al nostro Bean, in modo da "ripulire" gli eccessi.

Per raggiungere questo obiettivo, bisogna associare al Bean una classe di supporto, che implementi l'interfaccia BeanInfo. Una classe BeanInfo permette di fare un certo numero di cose: esporre solamente i servizi che si desidera rendere visibili, aggirare le convenzioni di naming imposte dalle specifiche Java Beans, associare al Bean un'icona e attribuire ai servizi nomi più descrittivi di quelli rilevabili con il processo di analisi delle firme dei metodi.

Creare una classe BeanInfo

Per creare una classe BeanInfo bisogna anzitutto definire una classe con lo stesso nome del Bean, a cui si deve aggiungere il suffisso BeanInfo. Per semplificare il lavoro si può estendere SimpleBeanInfo, una classe che fornisce un'implementazione nulla di tutti i metodi dell'interfaccia. In questo modo ci si limiterà a sovrascrivere solamente i metodi che interessano, lasciando tutti gli altri con l'impostazione di default.

Per ridefinire il numero ed il tipo dei servizi Bean, occorre agire in modo appropriato a restituire le proprietà, i metodi o gli eventi che si desidera esporre. Opzionalmente, si può associare

un'icona al Bean, definendo il metodo public java.awt.Image getIcon(int iconKind). Per finire, si può specificare la classe del Bean e il suo Customizer, qualora ne esista uno, con il metodo public BeanDescriptor getBeanDescriptor().

La classe BeanInfo così prodotta deve essere messa nello stesso package che contiene il Bean. In assenza di una classe BeanInfo, i servizi di un Bean vengono trovati con la reflection.

Feature Descriptors

Una classe di tipo BeanInfo restituisce, tramite i seguenti metodi, vettori di *descriptors* che contengono informazioni relative ad ogni proprietà, metodo o evento che il progettista di un Bean desidera esporre.

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

Ogni Descriptor fornisce una precisa rappresentazione di una classe di servizi Bean. Il package java.bean implementa le seguenti classi:

- FeatureDescriptor: è la classe base per tutte le altre classi Descriptor, e definisce gli aspetti comuni a tutta la famiglia.
- BeanDescriptor: descrive il tipo e il nome della classe Bean associati, oltre a fornire il Customizer, se ne esiste uno.
- PropertyDescriptor: descrive le proprietà del Bean.
- IndexedPropertyDescriptor: è una sottoclasse di PropertyDescriptor, e descrive le proprietà indicizzate.
- EventSetDescriptor: descrive gli eventi che il Bean è in grado di inviare.
- MethodDescriptor: descrive i metodi del Bean.
- ParameterDescriptor: descrive i parametri dei metodi.

Esempio

In questo esempio si analizzerà un BeanInfo per il Bean PhotoAlbum, che permette di nascondere una grossa quantità di servizi Bean che per default vengono ereditati dalla superclasse JPanel.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.beans.*;
```

```
import com.mokabyte.mokabook.javaBeans.photoAlbum.*;

public class PhotoAlbumBeanInfo extends SimpleBeanInfo {

private static final Class beanClass = PhotoAlbum.class;

 public PropertyDescriptor[] getPropertyDescriptors() {
    try {
       PropertyDescriptor imageNumber
       = new PropertyDescriptor("imageNumber", beanClass, "getImageNumber", null);
       PropertyDescriptor imageIndex = new PropertyDescriptor("imageIndex", beanClass, "getImageIndex", null);
       PropertyDescriptor imageComment
       = new PropertyDescriptor("imageComment", beanClass, "getImageComment", null);

       imageIndex.setBound(true);
       imageComment.setBound(true);

       PropertyDescriptor properties[]
       = {imageNumber, imageIndex, imageComment};
       return properties;
    } catch (IntrospectionException e) {
        throw new Error(e.toString());
    }
 }
 public EventSetDescriptor[] getEventSetDescriptors() {
    try {
       EventSetDescriptor changed
       = new EventSetDescriptor(beanClass, "propertyChange", PropertyChangeListener.class, "propertyChange");
       changed.setDisplayName("Property Change");
       EventSetDescriptor events[] = {changed};
       return events;
    } catch (IntrospectionException e) {
       throw new Error(e.toString());
    }
 }
 public MethodDescriptor[] getMethodDescriptors() {
    try {
       MethodDescriptor showNext
       = new MethodDescriptor(beanClass.getMethod("showNext", null));

          MethodDescriptor methods[] = {showNext};
          return methods;
    } catch (Exception e) {
           throw new Error(e.toString());
    }
 }
 public java.awt.Image getIcon(int iconKind){
```

```
    if(iconKind == SimpleBeanInfo.ICON_COLOR_16x16)
      return loadImage("photoAlbumIcon16.gif");
    else
      return loadImage("photoAlbumIcon32.gif");
  }
}
```

La classe viene definita come sottoclasse di SimpleBeanInfo, in modo da rendere il processo di sviluppo più rapido.

Il primo metodo, getPropertyDescriptors, restituisce un array con un tre PropertyDescriptor, uno per ciascuna delle proprietà che si vogliono rendere visibili. Il costruttore di PropertyDescriptor richiede quattro argomenti: il nome della proprietà, la classe del Bean, il nome del metodo getter e quello del metodo setter: quest'ultimo è posto a null, a significare che le proprietà sono di tipo Read Only. Si noti, in questo metodo e nei successivi, che la creazione dei Descriptors deve essere definita all'interno di un blocco try–catch, dal momento che può generare IntrospectionException.

Il secondo metodo, getEventSetDescriptors(), restituisce un vettore con un unico EventSetDescriptor. Quest'ultimo viene inizializzato con quattro parametri: la classe del Bean, il nome della proprietà, la classe dell'ascoltatore e la firma del metodo che riceve l'evento. Si noti la chiamata al metodo setDisplayName(), che permette di impostare un nome più leggibile di quello che viene normalmente ottenuto dalle firme dei metodi.

Il terzo metodo, getMethodDescriptors, restituisce un vettore contenente un unico MethodDescriptor, che descrive il metodo showNext(). Il costruttore di MethodDescriptor richiede come unico parametro un oggetto di classe Method, che in questo esempio viene richiesto alla classe PhotoAlbum ricorrendo alla reflection.

Infine il metodo getIcon() restituisce un'icona, che normalmente viene associata al Bean all'interno di strumenti visuali.

Per impacchettare il Bean PhotoAlbum con le icone e il BeanInfo, si può seguire la procedura già descritta, modificando la riga di comando dell'utility jar in modo da includere le icone nell'archivio.

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp com\mokabyte\mokabook\
javaBeans\photoAlbum\*.class com\mokabyte\mokabook\javaBeans\
photoAlbum\*.gif com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

Personalizzazione dei Bean

L'aspetto e il comportamento di un Bean possono essere personalizzati in fase di composizione all'interno di un tool grafico abilitato ai Beans. Esistono due strumenti per personalizzare un Bean: gli Editor di proprietà e i Customizer. Gli Editor di proprietà sono componenti grafici specializzati nell'editing di un particolare *tipo* di proprietà: interi, stringhe, files… Ogni Editor di proprietà viene associato a un particolare tipo Java, e il tool grafico compone automaticamente un Property Sheet analizzando le proprietà di un Bean, e ricorrendo agli Editor più adatti alla circostanza. In fig. 18.6 si può vedere un esempio di Property Sheet, realizzato dal Bean Box: ogni riga presenta, accanto al nome della proprietà, il relativo Editor.

Figura 18.6 – *Un Property Sheet generato in modo automatico a partire dalle proprietà di un pulsante Bean*

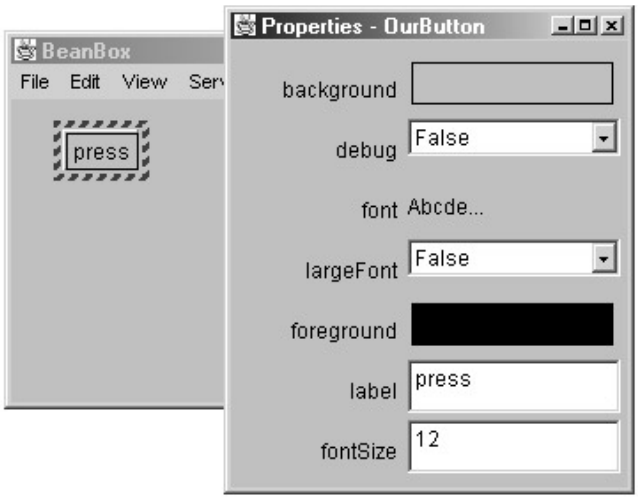
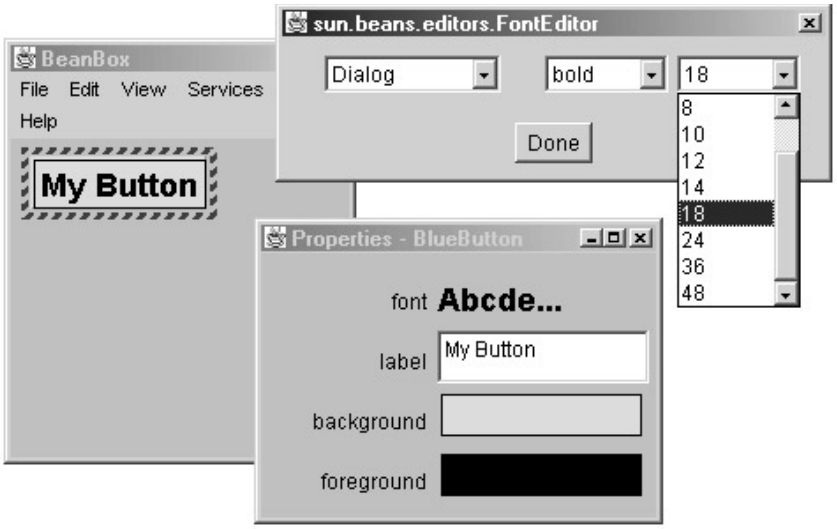


Figura 18.7 – *Il Property Sheet relativo a un pulsante Bean. Si noti il pannello ausiliario FontEditor*



Un Customizer, d'altra parte, è un pannello di controllo specializzato per un particolare Bean: in questo caso è il programmatore a decidere cosa mostrare nel pannello e in quale maniera. Per questa ragione un Customizer viene associato, grazie al BeanInfo, a un particolare Bean e non può, in linea di massima, essere usato su Bean differenti.

Come creare un Editor di proprietà

Un Editor di proprietà deve implementare l'interfaccia PropertyEditor, o in alternativa, estendere la classe PropertyEditorSupport che fornisce un'implementazione standard dei metodi dell'interfaccia. L'interfaccia PropertyEditor dispone di metodi che premettono di specificare come una proprietà debba essere rappresentata in un property sheet. Alcuni Editor consistono in uno strumento direttamente editabile, altri presentano uno strumento a scelta multipla, come un ComboBox; altri ancora, per permettere la modifica, aprono un pannello separato, come nella proprietà font dell'esempio, che viene modificata grazie al pannello ausiliario FontEditor.

Per fornire il supporto a queste modalità di editing, bisogna implementare alcuni metodi di PropertyEditor, in modo che ritornino valori non nulli.

I valori numerici o String possono implementare il metodo setAsText(String s), che estrae il valore della proprietà dalla stringa che costituisce il parametro. Questo sistema permette di inserire una proprietà con un normale campo di testo.

Gli Editor standard per le proprietà Color e Font usano un pannello separato, e ricorrono al Property Sheet solamente per mostrare l'impostazione corrente. Facendo click sul valore, viene aperto l'Editor vero e proprio. Per mostrare il valore corrente della proprietà, è necessario sovrascrivere il metodo isPaintable() in modo che restituisca true, e sovrascrivere paintValue in modo che dipinga la proprietà attuale in un rettangolo all'interno del Property Sheet.

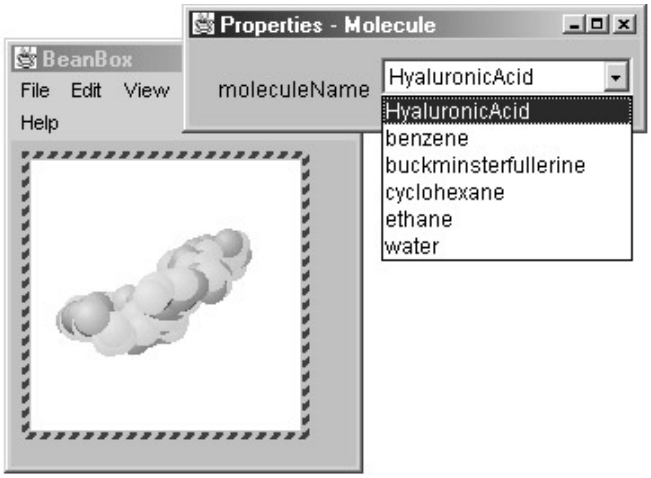
Per supportare l'Editor di Proprietà personalizzato occorre sovrascrivere altri due metodi della classe PropertyEditorSupport: supportsCustomEditor, che in questo caso deve restituire true, e getCustomEditor, in modo che restituisca un'istanza dell'Editor.

Registrare gli Editor

I Property Editor vengono associati alle proprietà attraverso un'associazione esplicita, all'interno del metodo getPropertyDescriptors() del BeanInfo, con una chiamata al metodo setPropertyEditorClass(Class propertyEditorClass) del PropertyDescriptor corrispondente, come avviene nel Bean Molecule

```
PropertyDescriptor pd = new PropertyDescriptor("moleculeName", Molecule.class);
pd.setPropertyEditorClass(MoleculeNameEditor.class);
```

Figura 18.8 – *Il Bean Molecule associa alla proprietà moleculeName di un Editor di proprietà personalizzato*



In alternativa si può registrare l'Editor con il seguente metodo statico

```
PropertyEditorManager.registerEditor(Class targetType, Class editorType)
```

che richiede come parametri la classe che specifica il tipo e quella che specifica l'Editor.

Customizers

Con un Bean Customizer è possibile fornire un controllo completo sul modo in cui configurare ed editare un Bean. Un Customizer è in pratica una piccola applicazione specializzata nell'editing di un particolare Bean, ogni volta che la configurazione di un Bean richiede modalità troppo sofisticate per il normale processo di creazione automatica del Property Sheet.

Le uniche regole a cui ci si deve attenere per realizzare un Customizer sono:

- deve estendere la classe `Component`, o una delle sue sottoclassi;
- deve implementare l'interfaccia `java.bean.Customizer`;
- deve implementare un costruttore privo di parametri.

Per associare il Customizer al proprio Bean, bisogna sovrascrivere il metodo `getBeanDescriptor` nella classe `BeanInfo`, in modo che restituisca un opportuno `BeanDescriptor`, il quale a sua volta dovrà restituire la classe del Customizer alla chiamata del metodo `getCustomizerClass`.

Serializzazione

Per rendere serializzabile una classe Bean è di norma sufficiente implementare l'interfaccia Serializable, sfruttando così l'Object Serialization di Java. L'interfaccia Serializable non contiene metodi: essa viene usata dal compilatore per marcare le classi che possono essere serializzate. Esistono solo poche regole per implementare classi Serializable: anzitutto è necessario dichiarare un costruttore privo di argomenti, che verrà chiamato quando l'oggetto verrà ricostruito a partire da un file .ser; in secondo luogo una classe serializzabile deve definire al suo interno solamente attributi serializzabili.

Se si desidera fare in modo che un particolare attributo non venga salvato al momento della serializzazione, si può ricorrere al modificatore transient. La serializzazione standard, inoltre, non salva lo stato delle variabili static.

Per tutti i casi in cui la serializzazione standard non risultasse applicabile, occorre procedere all'implementazione dell'interfaccia Externalizable, fornendo, attraverso i metodi readExternal(ObjectInput in) e writeExternal(ObjectOutput out), delle istruzioni esplicite su come salvare lo stato di un oggetto su uno stream e come ripristinarlo in un secondo tempo.