Capitolo 15

Pannelli, accessori e decorazioni

Andrea Gini

Si è visto come sia possibile creare interfacce grafiche innestando pannelli uno all'interno dell'altro. La scelta di pannelli disponibili in Swing non è limitata al semplice JPanel, ma include pannelli specializzati nel trattamento di casi particolari. In questo capitolo verranno illustrati JSplitPane e JTabbedPane, due pannelli estremamente utili per creare interfacce grafiche. In un secondo tempo verrà spiegato l'uso della classe JOptionPane, che permette di creare finestre di dialogo di diverso genere, e sarà presentato il meccanismo di gestione dei look & feel di Swing. Per finire, si vedrà un esempio di applicazione grafica di una certa complessità.

Pannelli

JSplitPane

JSplitPane è un pannello formato da due aree, separate da una barra mobile. Al suo interno è possibile disporre una coppia di componenti, affiancati lateralmente o uno sopra l'altro. Il divisore può essere trascinato per impostare l'area da assegnare a ciascun componente, rispettandone la dimensione minima. Usando JSplitPane in abbinamento a JScrollPane si può ottenere una coppia di pannelli ridimensionabili. Il seguente programma crea una finestra con un JSplitPane al suo interno: nel pannello superiore monta un'immagine JPEG, in quello inferiore un'area di testo in cui si possono annotare commenti. Per avviarlo è necessario specificare sulla riga di comando il percorso di un file JPEG o GIF (per esempio, java JSplitDemo c:\immagine.jpg):

```
import javax.swing.*;
import java.awt.*;

public class JSplitDemo extends JFrame {

  public JSplitDemo(String fileName) {
    super("JSplitPane");
    setSize(300, 250);

    // costruisce un pannello contenente un'immagine
    ImageIcon img = new ImageIcon(fileName);
    JLabel picture = new JLabel(img);
    JScrollPane pictureScrollPane = new JScrollPane(picture);

    // crea un pannello che contiene un'area di testo
    JTextArea comment = new JTextArea();
    JScrollPane commentScrollPane = new JScrollPane(comment);

    // Crea uno SplitPane verticale con i due pannelli al suo interno
    JSplitPane splitPane= new JSplitPane(JSplitPane.VERTICAL_SPLIT, pictureScrollPane, commentScrollPane);
    splitPane.setOneTouchExpandable(true);
    splitPane.setDividerLocation(190);
    splitPane.setContinuousLayout(true);

    // aggiunge lo SplitPane al frame principale
    getContentPane().add(splitPane);
    setVisible(true);
  }
  public static void main(String argv[]) {

    if(argv.length == 1) {
      JSplitDemo b = new JSplitDemo(argv[0]);
    }
    else
      System.out.println("usage JSplitDemo <filename>");
  }
}
```

Figura 15.1 – *Un esempio di JSplitPane.*

JSplitPane API

Nell'esempio si è fatto ricorso a un costruttore che permette di impostare le più importanti proprietà dello SplitPane con un'unica istruzione:

```
public JSplitPane(int orientamento, Component leftComponent, Component rightComponent)
```

Il primo parametro serve a specificare l'orientamento: esso può assumere i valori JSplitPane.HORIZONTAL_SPLIT o JSplitPane.VERTICAL_SPLIT. Il secondo e il terzo parametro permettono di impostare i due componenti da disporre nel pannello. Questi parametri possono essere impostati anche con i seguenti metodi:

```
void setOrientation(int orientation)
void setBottomComponent(Component comp)
void setTopComponent(Component comp)
void setRightComponent(Component comp)
void setLeftComponent(Component comp)
```

Un gruppo di tre metodi permette di specificare la posizione del divisore:

```
void setDividerLocation(int location)
void setDividerLocation(double proportionalLocation)
void setResizeWeight(double d)
```

Il primo di questi metodi chiede di specificare con un intero la posizione assoluta del divisore, mentre il secondo imposta la posizione del divisore suddividendo in modo proporzionale lo spazio disponibile tra i due componenti (con un valore di 0,5 il divisore viene posto a metà). Infine, il metodo setResizeWeight(double d) consente di specificare come si desidera distribuire lo spazio che si viene a creare quando il componente viene ridimensionato, mediante un parametro reale compreso tra 0 e 1. Se si imposta un valore di 0,5 lo spazio in più viene diviso in misura uguale tra i due componenti.

Per modificare l'aspetto fisico del pannello sono disponibili le seguenti possibilità:

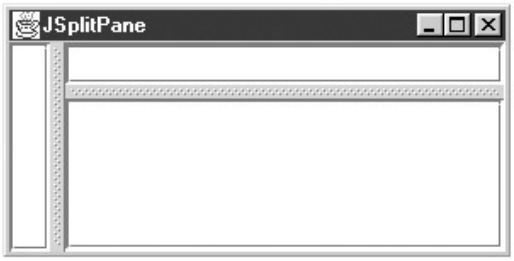
```
void setDividerSize(int)
void setOneTouchExpandable(boolean)
void setContinuousLayout(boolean)
```

Il primo serve a impostare la dimensione in pixel della barra di divisione; il secondo permette di attivare una coppia di pulsanti a freccia che consentono di espandere o collassare il divisore con un semplice clic; il terzo, infine, consente di specificare se si desidera che il pannello venga ridisegnato durante il posizionamento del divisore.

Per suddividere un'area in più di due aree ridimensionabili, è possibile inserire i JSplitPane uno dentro l'altro. Nel seguente esempio vengono creati un JSplitPane orizzontale, contenente una JTextArea, e un JSplitPane verticale, che contiene a sua volta due ulteriori JTextArea.

```
JScrollPane scroll1 = new JScrollPane(new JTextArea());
JScrollPane scroll2 = new JScrollPane(new JTextArea());
JScrollPane scroll3 = new JScrollPane(new JTextArea());

JSplitPane internalSplit
= new JSplitPane(JSplitPane.VERTICAL_SPLIT, scroll1, scroll2);
JSplitPane externalSplit
= new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, scroll3, internalSplit );
```

Figura 15.2 – *È possibile creare JSplitPane multipli e annidarli l'uno dentro l'altro.*



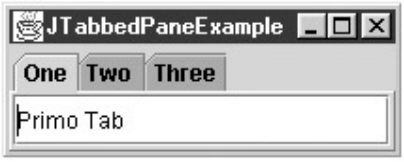
JTabbedPane

JTabbedPane permette a diversi componenti di condividere lo stesso spazio sullo schermo: l'utente può scegliere su quale componente operare premendo il Tab corrispondente. Il tipico uso di questo componente è nei pannelli di controllo, nei quali si assegna a ogni Tab la gestione di un insieme di funzioni differenti. Oltre all'innegabile utilità, questo componente presenta una modalità di impiego straordinariamente semplice: è sufficiente creare il pannello e aggiungervi i vari componenti usando il metodo addTab(String title, Component c), in cui il primo parametro specifica l'etichetta del Tab, e il secondo passa il componente. Il passaggio da un Tab all'altro viene ottenuto cliccando con il mouse sul Tab desiderato, senza bisogno di gestire gli eventi in modo esplicito. Nell'esempio seguente viene creato un JTabbedPane al quale vengono aggiunti tre Tab, ognuno dei quali contiene un componente grafico diverso. L'esempio mostra anche un esempio di gestione degli eventi: al cambio di Tab viene aggiornato il titolo della finestra.

```
import javax.swing.*;
import javax.swing.event.*;

public class JTabbedPaneExample extends JFrame {
   private JTabbedPane tabbedPane;
   public JTabbedPaneExample() {
     super("JTabbedPaneExample");
     tabbedPane = new JTabbedPane();

     JTextField tf = new JTextField("primo Tab");
     JButton b = new JButton("secondo Tab");
     JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 60, 15);
     tabbedPane.addChangeListener(new TabListener());
     tabbedPane.addTab("uno", tf );
     tabbedPane.addTab("due", b );
     tabbedPane.addTab("tre", slider);

     getContentPane().add(tabbedPane);
```

```
        pack();
        setVisible(true);
    }
    public class TabListener implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
            int pos = tabbedPane.getSelectedIndex();
            String title = tabbedPane.getTitleAt(pos);
            setTitle(title);
        }
    }
    public static void main(String[] args) {
     JTabbedPaneExample te = new JTabbedPaneExample();
    }
}
```

Figura 15.3 – *Un semplice esempio di JTabbedPane.*



Nell'esempio è stato creato un JTabbedPane e sono stati inseriti al suo interno tre Tab, ognuno dei quali contiene un componente. Naturalmente, è possibile inserire all'interno di un Tab un intero pannello con tutto il suo contenuto:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
panel.add(BorderLayout.NORTH, new Button("nord"));
....
panel.add(BorderLayout.SOUTH, new Button("sud"));
tabbedPane.addTab("Pannello", panel);
....
```

JTabbedPane API

Per creare un JTabbedPane è possibile ricorrere ai costruttori riportati di seguito:

```
JTabbedPane()
JTabbedPane(int tabPlacement)
```

Il secondo permette di specificare il posizionamento dei tab attraverso un parametro che può assumere i valori JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT o JTabbedPane.RIGHT.

Per aggiungere o togliere componenti è disponibile un gruppo di metodi simili a quelli di una nomale collection, tipo Vector. Ogni pannello può essere associato a un titolo con o senza un'icona:

```
void addTab(String title, Component component)
void addTab(String title, Icon icon, Component component, String tip)
void remove(Component component)
void removeAll()
int getTabCount()
```

Per operare sui componenti presenti all'interno dei tab sono disponibili i seguenti metodi;

```
Component getSelectedComponent()
void setSelectedComponent(Component c)
int getSelectedIndex()
void setSelectedIndex(int index)
int indexOfComponent(Component component)
String getTitleAt(int index)
```

La gestione degli eventi su JTabbedPane è abbastanza limitata, dal momento che gli oggetti di tipo ChangeEvent non contengono nessuna informazione sul tipo di evento che li ha generati (in pratica, è impossibile, per un ascoltatore, distinguere tra eventi di selezione, di aggiunta o di rimozione di tab). Per aggiungere o rimuovere un ascoltatore si utilizza la caratteristica coppia di metodi:

```
void addChangeListener(ChangeListener l)
void removeChangeListener(ChangeListener l)
```

Gli ascoltatori di tipo ChangeListener sono caratterizzato dal metodo void stateChanged (ChangeEvent e); gli eventi di tipo ChangeEvent contengono il solo metodo Object getSource(), che permette di ottenere un riferimento all'oggetto che ha generato l'evento. Per conoscere i dettagli dell'evento (numero di Tab, titolo e così via) è necessario interrogare direttamente il componente sorgente.

Accessori e decorazioni

JOptionPane

La classe JOptionPane permette di realizzare facilmente finestre modali di input, di allarme o di scelta multipla, ossia quel genere di finestre che vengono utilizzate qualora sia necessario segnalare un malfunzionamento, o presentare all'utente un insieme di scelte su come procedere nell'esecuzione di un programma. L'API JOptionPane mette a disposizione tre tipi di pannelli:

confirm dialog, input dialog e message dialog. Il primo tipo di pannello viene usato quando si deve chiedere all'utente di effettuare una scelta tra un ventaglio di possibilità; il secondo torna utile per richiedere l'inserimento di una stringa di testo; il terzo, infine, viene usato per informare l'utente di un evento. La classe JOptionPane fornisce un gruppo di metodi statici che permettono di creare facilmente questi pannelli ricorrendo a una sola riga di codice. Ecco un esempio di confirm dialog:

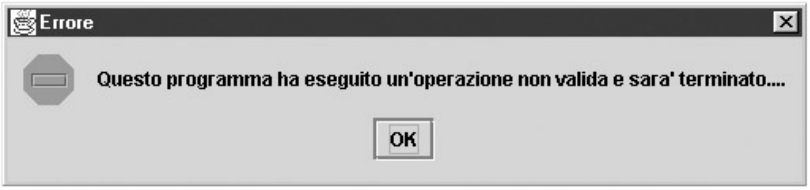
```
JOptionPane.showConfirmDialog(null, "Salvare le modifiche?");
```

e uno di message dialog:

```
JOptionPane.showMessageDialog(null,
 "Questo programma ha eseguito un'operazione non valida e sarà terminato...", "Errore", JOptionPane.ERROR_MESSAGE);
```

Figura 15.4 – *Un pannello di conferma può aiutare a evitare guai.*

Figura 15.5 – *Un minaccioso pannello di notifica annuncia che ormai è troppo tardi.*



Come si può vedere, non è stato necessario creare esplicitamente alcun oggetto di tipo JOptionPane: in entrambi i casi è bastato richiamare un metodo statico che ha provveduto a creare un oggetto grafico con le caratteristiche specificate dai parametri. In questo paragrafo ci si concentrerà sull'utilizzo di un sottoinsieme di tali metodi, che permettono di affrontare la stragrande maggioranza delle situazioni in modo compatto ed elegante:

```
static int showConfirmDialog(Component parentComponent, Object message)
static int showConfirmDialog(Component
```

```
parentComponent, Object message, String title, int optionType, int messageType)

static String showInputDialog(Component parentComponent, Object message)
static String showInputDialog(Component parentComponent, Object message, String title, int messageType)

static void showMessageDialog(Component parentComponent, Object message)
static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)
```

Le finestre di conferma permettono di scegliere tra un gruppo di opzioni (tipicamente Yes, No e Cancel), mentre le finestre di input servono a inserire del testo; infine, i message dialog si usano per informare l'utente di una particolare situazione. Per ognuna di queste situazioni sono disponibili una chiamata generica e una che permette invece di specificare un gran numero di parametri. Vediamo quali sono i più importanti:

Parent Component

Questo parametro serve a specificare il frame principale. Esso verrà bloccato fino al termine dell'interazione. Ponendo a null questo parametro, la finestra verrà visualizzata al centro dello schermo e risulterà indipendente dal resto dell'applicazione.

Message

Questo campo permette di specificare una stringa da visualizzare come messaggio. In alternativa a String, si può passare una Icon o una qualsiasi sottoclasse di Component.

OptionType

I confirm dialog possono presentare diversi gruppi di opzioni a seconda del valore di questo parametro. Esso può assumere i seguenti valori costanti:

```
JOptionPane.YES_NO_OPTION
JOptionPane.YES_NO_CANCEL_OPTION
JOptionPane.OK_CANCEL_OPTION
```

Message Type

Mediante questo parametro è possibile influenzare l'aspetto complessivo della finestra, per quanto attiene al tipo di icona, al titolo e al layout. Il parametro può assumere uno dei seguenti valori:

```
JOptionPane.ERROR_MESSAGE
JOptionPane.INFORMATION_MESSAGE
JOptionPane.WARNING_MESSAGE
JOption-Pane.QUESTION_MESSAGE
JOptionPane.PLAIN_MESSAGE
```

Le finestre create con i metodi showConfirmDialog e showMessageDialog restituiscono un intero che fornisce informazioni su quale scelta è stata effettuata dall'utente. Esso può assumere uno dei seguenti valori:

```
JOptionPane.YES_OPTION
JOptionPane.NO_OPTION
JOptionPane.CANCEL_OPTION
JOption-Pane.OK_OPTION
JOptionPane.CLOSED_OPTION
```

Nel caso di showInputDialog() viene invece restituita una stringa di testo, o null se l'utente ha annullato l'operazione. Il programmatore può prendere decisioni sul modo in cui proseguire leggendo e interpretando la risposta in maniera simile a come si vede in questo esempio:

```
int returnVal = JOptionPane.showConfirmDialog(null, "Salvare le modifiche?");
if(returnVal == JOptionPane.YES_OPTION)
        // procedura da eseguire in caso affermativo
else if(returnVal == JOptionPane.NO_OPTION)
        // procedura da eseguire in caso negativo
else;
        // operazione abortita
```

JFileChooser

Un file chooser è un oggetto grafico che permette di navigare il file system e di selezionare uno o più file su cui eseguire una determinata operazione. Qualsiasi applicazione grafica ne utilizza uno per facilitare le operazioni su disco. JFileChooser offre questa funzionalità tramite un'accessoriata finestra modale.

Si può creare un'istanza di JFileChooser utilizzando i seguenti costruttori:

```
JFileChooser()
JFileChooser(File currentDirectory)
```

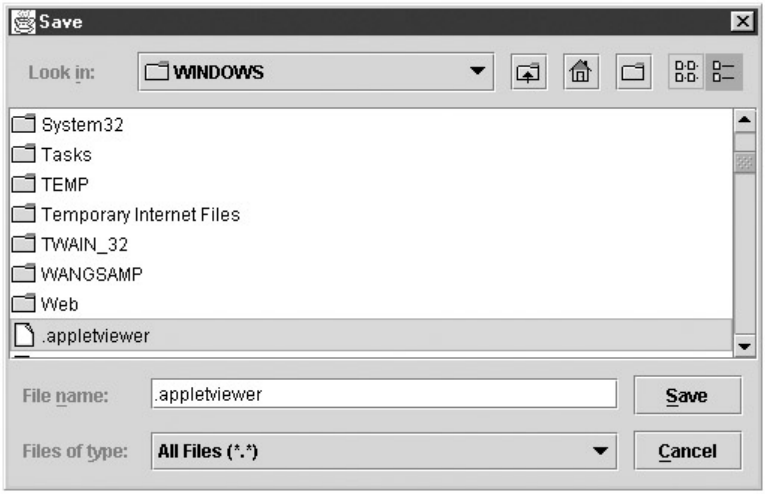
Crea un JFileChooser che punta alla directory specificata dal parametro. Per visualizzare un JFileChooser è possibile ricorrere alla seguente coppia di metodi, a seconda che si desideri aprire una finestra di apertura o di salvataggio file:

```
int showOpenDialog(Component parent)
int showSaveDialog(Component parent)
```

Entrambi i metodi restituiscono un intero che può assumere uno dei tre valori:

```
JFile-Chooser.CANCEL_OPTION
JFileChooser.APPROVE_OPTION
JFile-Chooser.ERROR_OPTION
```

Figura 15.6 – *Un esempio di JFileChooser.*



Il programmatore può decidere cosa fare dei file selezionati basandosi su queste risposte. Come di consueto per le finestre modali, entrambi i metodi richiedono un reference a un componente, in modo da bloccare il JFrame principale per tutta la durata dell'operazione. Passando null come parametro, il JFrame verrà visualizzato al centro dello schermo e risulterà indipendente dalle altre finestre. Per conoscere il risultato dell'interrogazione, è possibile usare i seguenti metodi:

```
File getCurrentDirectory()
File getSelectedFile()
```

Tali parametri possono essere impostati per via programmatica attraverso la seguente coppia di metodi:

```
void setCurrentDirectory(File dir)
void setSelectedFile(File file)
```

Alcuni metodi consentono un uso più avanzato di JFileChooser. Il metodo setDialogTitle(String dialogTitle) permette di impostare il titolo del JFileChooser; setFileSelectionMode(int mode) consente di abilitare il JFileChooser a selezionare solo file, solo directory o entrambi a seconda del valore del parametro:

```
JFileChooser.FILES_ONLY
JFileChooser.DIRECTORIES_ONLY
JFileChooser.FILES_AND_DIRECTORIES
```

Il metodo setMultiSelectionEnabled(boolean b) abilita o disabilita la possibilità di selezionare più di un file per volta. In questa modalità, per interrogare o modificare lo stato del componente si ricorrerà ai due metodi che seguono:

```
void setSelectedFiles(File[] selectedFiles)
File[] getSelectedFiles()
```

Nelle righe seguenti si può osservare una tipica procedura che fa uso di JFileChooser:

```
class MyFrame extends JFrame {
....
   fileChooser = new JFileChooser();
   int response = fileChooser.showOpenDialog(this);
   if(response == JFileChooser.APPROVE_OPTION) {
      File f = fileChooser.getSelectedFile();
      // qui viene eseguita l'operazione sul file
   }
....
}
```

Colori e JColorChooser

Un parametro molto importante nei programmi grafici è senza dubbio il colore. Ogni componente grafico Java presenta una quadrupla di metodi che permettono di leggere o impostare i colori di sfondo (background) e di primo piano (foreground):

```
Color getBackground()
void setBackground(Color c)
Color getForeground()
void setForeground(Color c)
```

I componenti Swing usano per default uno sfondo trasparente. Se si desidera impostare un colore di sfondo, bisogna prima rendere opaco il componente con il metodo setOpaque(true).

La classe Color permette di descrivere i colori mediant le tre componenti cromatiche red, green e blue (RGB). Il costruttore principale permette di specificare le tre componenti come interi compresi tra 0 e 255:

```
public Color(int r , int g , int b)
```

I colori più comuni sono disponibili anche sotto forma di costanti della classe Color. Di seguito viene fornito un elenco di tali costanti e il rispettivo valore RGB:

```
public final static Color WHITE      =  new Color(255, 255, 255);  // Bianco
public final static Color LIGHT_GRAY =  new Color(192, 192, 192);  // Grigio Chiaro
public final static Color GRAY       =  new Color(128, 128, 128);  // Grigio
public final static Color DARK_GRAY  =  new Color(64, 64, 64);     // Grigio Scuro
public final static Color BLACK      =  new Color(0, 0, 0);        // Nero
public final static Color RED        =  new Color(255, 0, 0);      // Rosso
public final static Color PINK       =  new Color(255, 175, 175);  // Rosa
public final static Color ORANGE     =  new Color(255, 200, 0);    // Arancio
public final static Color YELLOW     =  new Color(255, 255, 0);    // Giallo
public final static Color GREEN      =  new Color(0, 255, 0);      // Verde
public final static Color MAGENTA    =  new Color(255, 0, 255);    // Magenta
public final static Color CYAN       =  new Color(0, 255, 255);    // Ciano
public final static Color BLUE       =  new Color(0, 0, 255);      // Blu
```

Le costanti appena illustrate sono presenti nelle versioni del JDK a partire dalla 1.4. Le versioni precedenti dispongono di un insieme di costanti equivalenti, con il nome in lettere minuscole e convenzione CamelCase (per esempio Color.white, Color.lightGray e così via). Tali costanti, ora deprecated, sono disponibili anche nelle versioni del JDK più recenti.

Il pacakge Swing dispone un color chooser, un componente che permette di navigare i colori di sistema con una pratica interfaccia grafica e di selezionarne uno. JColorChooser può essere utilizzato sia come componente separato sia come finestra di dialogo. Nel primo caso bisogna creare un JColorChooser e inserirlo all'interno dell'interfaccia grafica come un qualsiasi altro componente. I metodi void setColor(Color c) e Color getColor() permettono di impostare il colore iniziale o di leggere quello selezionato dall'utente. È anche possibile utilizzare un PropertyChangeListener in modo da essere informati non appena l'utente seleziona un colore dalla palette:

```
JColorChooser c = new JColorChooser();
c.addPropertyChangeListener(new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent e) {
    if(e.getPropertyName().equals("color"))
      System.out.println("Hai selezionato il colore " + e.getNewValue());
  }
});
```

L'uso dei PropertyChangeListener verrà approfondito nel capitolo 18.

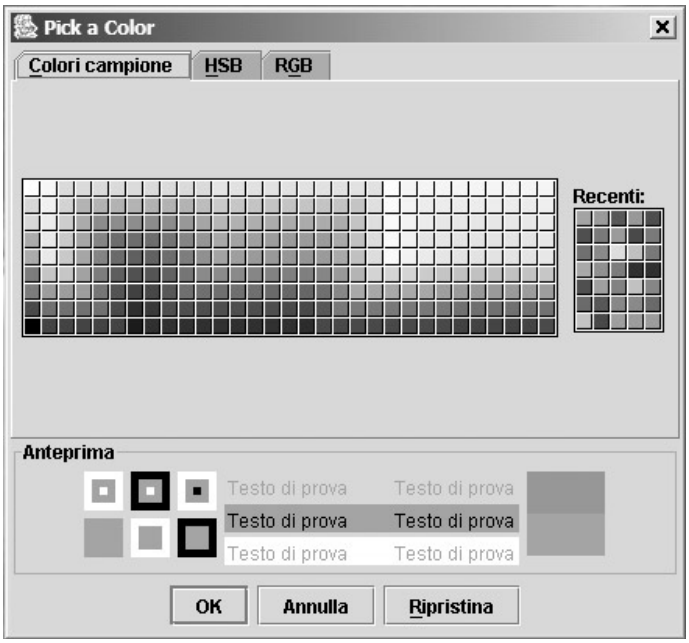
Per utilizzare JColorChooser sotto forma di finestra di dialogo, è disponibile un comodo metodo statico, che rende l'utilizzo del componente particolarmente rapido:

```
static Color showDialog(Component component, String title, Color initialColor)
```

In questo caso il color chooser viene visualizzato all'interno di una finestra modale che scompare non appena l'utente ha effettuato una scelta. Come parametri è necessario specificare la

finestra da bloccare, il titolo da dare alla finestra e il colore su cui impostare il color chooser al momento della visualizzazione. L'oggetto restituito corrisponde al colore selezionato dall'utente. Se l'utente ha annullato l'operazione, invece, viene restituito null.

Figura 15.7 – *JColorChooser permette di selezionare un colore utilizzando tre tipi di palette diverse.*



Font e FontChooser

La classe Font incapsula tutte le informazioni relative ai font della piattaforma ospite. Il costruttore permette di specificare il nome di un font, il suo stile e la sua dimensione in punti tipografici:

```
Font(String name, int style, int size)
```

Il parametro style può assumere quattro valori, a seconda che si desideri un carattere normale, grassetto, corsivo o grassetto e corsivo insieme:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD | Font.ITALIC
```

Il nome del font può essere di due tipi: logico o fisico. Il nome logico può assumere i seguenti valori: Serif, Sans-serif, Monospaced, Dialog e DialogInput. Tutte le piattaforme che supportano Java forniscono un mapping tra questi nomi logici e i font reali presenti sul sistema, in modo da permettere al programmatore di scrivere applicazioni portabili. Il nome fisico, invece, identifica un preciso font di sistema, e per questo un font valido su una particolare macchina può non esistere su un'altra. A partire dal JDK 1.2, la distribuzione standard di Java comprende comunque i seguenti font fisici:

```
LucidaBrightDemiBold.ttf
LucidaBrightDemiItalic.ttf
LucidaBrightItalic.ttf
LucidaBrightRegular.ttf
LucidaSansDemiBold.ttf
LucidaSansDemiOblique.ttf
LucidaSansOblique.ttf
LucidaSansRegular.ttf
LucidaTypewriterBold.ttf
LucidaTypewriterBoldOblique.ttf
LucidaTypewriterOblique.ttf
LucidaTypewriterRegular.ttf
```

Esiste un sistema per conoscere il nome fisico di tutti i font presenti nel sistema. Si tratta dell'oggetto GraphicEnvironment, accessibile tramite il metodo statico:

```
GraphicEnvironment.getLocalGraphicsEnvironment()
```

Esso dispone di un metodo statico in grado di restituire l'elenco dei font installati:

```
Font[] getAllFonts()
```

Grazie a questa chiamata è possibile scrivere con poche righe un programma che stampi a console i nomi di tutti i font di sistema:

```
import java.awt.*;

public class FontEstractor {
  public static void main(String argv[]) {
    Font[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts();
    for ( int i = 0; i < fonts.length; i++)
      System.out.println(fonts[i].toString());
  }
}
```

Dal momento che il package Swing non contiene un componente JFontChooser, verrà ora mostrato come crearne uno. Si definisce una classe JFontChooser, sottoclasse di JComponent,

provvista di due JComboBox e due JCheckBox che permettono di selezionare un font, la sua dimensione e lo stile. Questo programma è la dimostrazione di come sia semplice creare dal niente un nuovo componente grafico da utilizzare in numerosi contesti:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class FontChooser extends JComponent {

 private JComboBox fontNameBox;
 private JComboBox fontSizeBox;
 private JCheckBox boldCheckBox;
 private JCheckBox italicCheckBox;

 public FontChooser() {
  fontNameBox = new JComboBox();
  fontSizeBox = new JComboBox();
  boldCheckBox = new JCheckBox("Bold", false);
  italicCheckBox = new JCheckBox("Italic", false);

  Font[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts();
  for ( int i = 0; i < fonts.length; i++)
   fontNameBox.addItem(fonts[i].getName());
  for ( int i = 6; i < 200; i++)
   fontSizeBox.addItem(new Integer(i));

  fontSizeBox.setSelectedIndex(12);
  setLayout(new GridLayout(0, 1));

  JPanel comboBoxPanel = new JPanel();
   comboBoxPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
   comboBoxPanel.add(fontNameBox);
   comboBoxPanel.add(fontSizeBox);
  add(comboBoxPanel);

  JPanel checkBoxPanel = new JPanel();
   checkBoxPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
   checkBoxPanel.add(boldCheckBox);
   checkBoxPanel.add(italicCheckBox);
  add(checkBoxPanel);
  setBorder(BorderFactory.createTitledBorder("Choose Font"));
  ActionListener eventForwarder = new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    setFont(new Font((String)fontNameBox.getSelectedItem(), (boldCheckBox.isSelected()
```

```
? Font.BOLD : Font.PLAIN) | (italicCheckBox.isSelected()
? Font.ITALIC : Font.PLAIN), ((Integer)fontSizeBox.getSelectedItem()).intValue()));
    }
  };
  fontNameBox.addActionListener(eventForwarder);
  fontSizeBox.addActionListener(eventForwarder);
  boldCheckBox.addActionListener(eventForwarder);
  italicCheckBox.addActionListener(eventForwarder);
 }

 public void setFont(Font f) {
  super.setFont(f);  // Questa chiamata genera un PropertyChangeEvent
  fontNameBox.setSelectedItem(f.getName());
  fontSizeBox.setSelectedItem(new Integer(f.getSize()));
  boldCheckBox.setSelected(f.isBold());
  italicCheckBox.setSelected(f.isItalic());
 }
 public static Font showFontChooser(Component parent,String title , Font initialFont) {
  FontChooser fc = new FontChooser();
  fc.setFont(initialFont);
  int answer = JOptionPane.showConfirmDialog(parent, fc, title, JOptionPane.OK_CANCEL_OPTION);
  if(answer != JOptionPane.OK_OPTION)
   return null;
  else
   return fc.getFont();
 }
}
```

Figura 15.8 – *Utilizzando i concetti esposti finora, è possibile creare un pratico JFontChooser.*

JFontChooser presenta due modalità d'uso del tutto simili a quelle di JColorChooser: da una parte è possibile crearlo come componente separato da usare all'interno di interfacce grafiche, dall'altra è possibile creare una finestra di dialogo mediante il seguente metodo statico:

```
static Font showFontChooser(Component parent , String title , Font initialFont)
```

Tale metodo richiede come parametri il componente parent, il titolo della finestra e il font con il quale impostare il componente al momento dell'apertura. Il valore di ritorno riporta il font selezionato o null se l'utente ha annullato l'operazione:

```
    Font f = showFontChooser(null,"Scegli un Font",new Font("monospaced",0,15));
    if(f != null) {
      JLabel label = new JLabel("Test");
      label.setFont(f);
      JOptionPane.showMessageDialog(null,label);
  }
```

Anche questo componente genera un PropertyChangeEvent ogni volta che l'utente seleziona un nuovo stile. Per gestire l'evento si può utilizzare un frammento di codice del tipo:

```
FontChooser fc = new FontChooser("FontChooser");
final JLabel label = new JLabel("Test");
fc.setFont(new Font("Times New Roman", 3, 120));
fc.addPropertyChangeListener(new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent e) {
    if(e.getPropertyName().equals("font"))
      label.setFont((Font)e.getNewValue());
  }
});
```

Pluggable look & feel

Ogni ambiente a finestre è caratterizzato da due proprietà fondamentali: l'aspetto dei componenti (ossia la loro sintassi), e la maniera in cui essi reagiscono alle azioni degli utenti (la loro semantica). L'insieme di queste proprietà viene comunemente definito look & feel.

Chiunque abbia provato a lavorare con un sistema Linux dopo anni di pratica su piattaforma Windows si sarà reso conto di quanto sia difficile abituarsi a una nuova semantica: le mani tendono a comportarsi come sulla vecchia piattaforma, ma la reazione che osserviamo con gli occhi non è quella che ci aspettavamo. Per esempio, in Linux è normale che le finestre si espandano verticalmente invece che a pieno schermo, e che i menu scompaiano quando si rilascia il pulsante del mouse.

La natura multi piattaforma di Java ha spinto i progettisti di Swing a separare le problematiche di disegno grafico dei componenti da quelle inerenti al loro contenuto informativo, con la sorprendente conseguenza di permettere agli utenti di considerare il look & feel come una proprietà del componente da impostare a piacere. La distribuzione standard del JDK comprende di base due alternative: Metal e Motif. La prima definisce un look & feel multipiattaforma, progettato per risultare il più possibile familiare a chiunque. La seconda implementa una vista

familiare agli utenti Unix. Le distribuzioni di Java per Windows e Mac includono anche un look & feel che richiama quello della piattaforma ospite. Per motivi di copyright Sun non può proporre queste due scelte su piattaforme diverse. Alcune software house indipendenti distribuiscono, sotto forma di file JAR, dei package contenenti dei look & feel alternativi, che è possibile aggiungere alla lista dei look & feel di sistema.

Per impostare da programma un particolare look & feel, è sufficiente chiamare il metodo UIManager.setLookAndFeel(String className) passando come parametro il nome di un look & feel nstallato nel sistema. Quindi, è necessario chiamare il metodo statico updateComponentTreeUI(Component c) della classe SwingUtilities sulla finestra principale, in modo da forzare tutti i componenti dell'interfaccia ad aggiornare il proprio look & feel. Per esempio:

```
try {
  UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
  SwingUtilities.updateComponentTreeUI(frame);
}
catch (Exception e) {}
```

Di seguito si presentano le stringhe relative ai quattro look & feel descritti sopra:

```
"javax.swing.plaf.metal.MetalLookAndFeel"
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
"javax.swing.plaf.mac.MacLookAndFeel"
```

L'ultima definisce il look & feel Mac, disponibile solo su piattaforma Apple.

Se si desidera interrogare il sistema per conoscere il nome e la quantità dei look & feel installati, si può ricorrere ai seguenti metodi statici di UIManager:

```
static String getSystemLookAndFeelClassName()
```

Restituisce il nome del look & feel che implementa il sistema a finestre della piattaforma ospite (Windows su sistemi Microsoft, Mac su macchine Apple e Motif su piattaforma Solaris). Se non esiste una scelta predefinita, viene restituito il nome del look & feel Metal.

```
static String getCrossPlatformLookAndFeelClassName()
```

Restituisce il nome del look & feel multipiattaforma: il Java look & feel (JLF).

```
static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()
```

Restituisce un vettore di oggetti che forniscono alcune informazioni sui look & feel installati nel sistema, come per esempio il nome (accessibile con il metodo getName()).

Il seguente esempio crea una finestra con tanti JRadioButton quanti sono i look & feel dispo-

nibili nel sistema. Ogni volta che l'utente preme uno dei pulsanti, il look & feel viene aggiornato di conseguenza:

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

public class LookAndFeelExample extends JFrame {
  public LookAndFeelExample() {
    super("LookAndFeelExample");
    getContentPane().setLayout(new GridLayout(0, 1));
    ButtonGroup group = new ButtonGroup();
    ActionListener buttonListener = new ButtonListener();
    getContentPane().add(new JLabel("Scegli un Look & Feel"));

    UIManager.LookAndFeelInfo[] lookAndFeelList = UIManager.getInstalledLookAndFeels();
    for ( int i = 0; i < lookAndFeelList.length; i++) {
      JRadioButton b = new JRadioButton(lookAndFeelList[i].getClassName());
      b.addActionListener(buttonListener);
      group.add(b);
      getContentPane().add(b);
    }
    pack();
  }
  public void changeLookAndFeel(String s) {
    try {
      UIManager.setLookAndFeel(s);
      SwingUtilities.updateComponentTreeUI(this);
    }
    catch (Exception ex) {}
  }
  class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JRadioButton b = (JRadioButton)e.getSource();
      changeLookAndFeel(b.getText());
    }
  }

  public static void main(String argv[]) {
    LookAndFeelExample e = new LookAndFeelExample();
    e.setVisible(true);
  }
}
```

Figura 15.9 – *Un programma di esempio visto con tre look & feel diversi.*

Border

Una caratteristica che Swing offre in esclusiva è la possibilità di assegnare un bordo diverso a ogni singolo componente grafico, sia esso un pannello, un pulsante o una Tool Bar. Per aggiungere un bordo a un componente, è sufficiente chiamare il metodo setBorder(Border b), passando come parametro un'istanza di una qualsiasi delle classi descritte di seguito.

Il package javax.swing.border offre ben sette tipi di bordo, il più semplice dei quali è composto da una singola riga dello spessore specificato.

```
LineBorder(Color color, int thickness, boolean roundedCorners)
```

Il tipo mostrato sopra crea un bordo a linea, con il colore, lo spessore e il tipo di bordo specificati. I bordi seguenti, invece, ricreano effetti tridimensionali. Essi richiedono come parametro un intero che può assumere il valore BevelBorder.LOWERED o BevelBorder.RAISED, a seconda che si desideri un effetto in rilievo o rientrante:

```
BevelBorder(int bevelType)
```

Crea un bordo in rilievo, del tipo specificato dal parametro.

`SoftBevelBorder(int bevelType)`

Crea un bordo in rilievo sfumato, del tipo specificato dal parametro.

`EtchedBorder(int etchType)`

Crea un bordo scolpito, del tipo specificato dal parametro.

Qualora si desideri creare attorno a un componente una vera e propria cornice di spessore arbitrario, è possibile ricorrere ai seguenti oggetti, che permettono creare bordi vuoti, a tinta unita o decorati con un'immagine GIF o JPEG:

`EmptyBorder(int top, int left, int bottom, int right)`

Crea un bordo vuoto dello spessore specificato.

`MatteBorder(Icon tileIcon)`

Crea un bordo utilizzando un'immagine.

`MatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`

Crea un bordo delle dimensioni specificate, utilizzando un'icona.

`MatteBorder(int top, int left, int bottom, int right, Color matteColor)`

Crea un bordo delle dimensioni e del colore specificati.

Per finire, è disponibile una coppia di bordi che permette di creare composizioni a partire da altri bordi:

`TitledBorder(Border border, String title, int titleJustification, int titlePosition, Font titleFont, Color titleColor)`

Crea una cornice, composta dal bordo che viene passato come primo parametro e dal titolo specificato dal secondo parametro. Il terzo parametro può assumere i valori `TitledBorder.CENTER`, `TitledBorder.LEFT` o `TitledBorder.RIGHT`. Il quarto, che specifica la posizione del titolo, può assumere invece i valori `Titled-Border.ABOVE_BOTTOM`, `TitledBorder.ABOVE_TOP`, `Titled-Border.BELOW_BOTTOM`, `TitledBorder.BELOW_TOP`. Gli ultimi due parametri specificano font e colore del titolo. Sono disponibili anche costruttori più semplici, per esempio uno che richiede solo i primi due parametri e uno che omette gli ultimi due.

`CompoundBorder(Border outsideBorder, Border insideBorder)`

Crea una cornice componendo i due bordi passati come parametro.

Un'applicazione grafica complessa

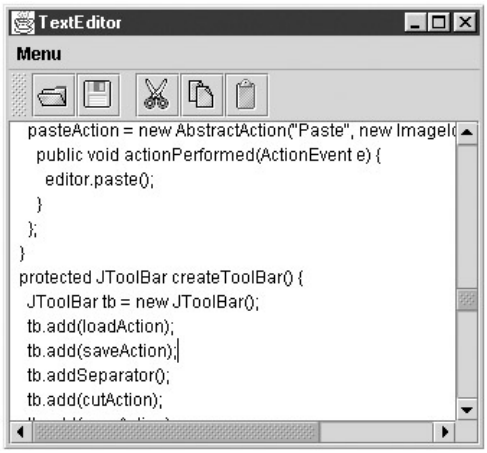
Le nozioni apprese finora permettono di affrontare lo studio di un'applicazione grafica di una discreta complessità. Le seguenti righe permettono di realizzare un piccolo editor di testo perfettamente funzionante, utilizzando una JTextArea, una JToolBar, una JMenuBar e un JFileChooser, e mostrando un utilizzo pratico delle Action. Viene inoltre illustrato, all'interno dei metodi loadText() e saveText(), come sia possibile inizializzare un JTextComponent a partire da un file su disco.

```
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class TextEditor extends JFrame {
   private JTextComponent editor;
   private JFileChooser fileChooser;
   protected Action loadAction;
   protected Action saveAction;
   protected Action cutAction;
   protected Action copyAction;
   protected Action pasteAction;
   public TextEditor() {
     super("TextEditor");
     setSize(300, 300);
     createActions();
     JMenuBar menuBar = createMenuBar();
     JToolBar toolBar = createToolBar();
     editor = createEditor();
     JComponent centerPanel = createCenterComponent();
     getContentPane().add(BorderLayout.NORTH, toolBar);
     getContentPane().add(BorderLayout.CENTER, centerPanel);
     setJMenuBar(menuBar);
     fileChooser = new JFileChooser();
     setVisible(true);
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
   protected void createActions() {
     loadAction = new AbstractAction("Open", new ImageIcon("Open24.gif")) {
        public void actionPerformed(ActionEvent e) {
           loadText();
        }
     };
     saveAction = new AbstractAction("Save", new ImageIcon("Save24.gif")) {
        public void actionPerformed(ActionEvent e) {
           saveText();
```

```
        }
    };
    cutAction = new AbstractAction("Cut", new ImageIcon("Cut24.gif")) {
        public void actionPerformed(ActionEvent e) {
            editor.cut();
        }
    };
    copyAction = new AbstractAction("Copy", new ImageIcon("Copy24.gif")) {
        public void actionPerformed(ActionEvent e) {
            editor.copy();
        }
    };
    pasteAction = new AbstractAction("Paste", new ImageIcon("Paste24.gif")) {
        public void actionPerformed(ActionEvent e) {
            editor.paste();
        }
    };
}
protected JToolBar createToolBar() {
    JToolBar tb = new JToolBar();
    tb.add(loadAction);
    tb.add(saveAction);
    tb.addSeparator();
    tb.add(cutAction);
    tb.add(copyAction);
    tb.add(pasteAction);
    return tb;
}
protected JMenuBar createMenuBar() {
    JMenu menu = new JMenu("Menu");
    menu.add(loadAction);
    menu.add(saveAction);
    menu.addSeparator();
    menu.add(cutAction);
    menu.add(copyAction);
    menu.add(pasteAction);
    JMenuBar menuBar = new JMenuBar();
    menuBar.add(menu);
    return menuBar;
}
protected JComponent createCenterComponent() {
    if(editor == null)
        editor = createEditor();
    return new JScrollPane(editor);
}
protected JTextComponent createEditor() {
    return new JTextArea();
}
```

```
    public void loadText() {
      int response = fileChooser.showOpenDialog(this);
      if(response == JFileChooser.APPROVE_OPTION) {
        try {
          File f = fileChooser.getSelectedFile();
          Reader in = new FileReader(f);
          editor.read(in, null);
          setTitle(f.getName());
        }
        catch(Exception e) {}
      }
    }
    public void saveText() {
      int response = fileChooser.showSaveDialog(this);
      if(response == JFileChooser.APPROVE_OPTION) {
        try {
          File f = fileChooser.getSelectedFile();
          Writer out = new FileWriter(f);
          editor.write(out);
          setTitle(f.getName());
        }
        catch(Exception e) {}
      }
    }
    public static void main(String argv[]) {
      TextEditor t = new TextEditor();
    }
}
```

Figura 15.10 – *Con appena un centinaio di righe è possibile realizzare un editor di testi completo.*



Il metodo createActions() riesce a definire cinque classi in appena trenta righe di codice, facendo uso delle classi anonime. L'uso di classi anonime in questo contesto è giustificato dal proposito di rendere il programma molto compatto. Questo programma vuole anche dare una dimostrazione di costruzione modulare di interfacce grafiche. Come si può notare, il costruttore genera gli elementi dell'interfaccia grafica ricorrendo a un gruppo di metodi factory, vale a dire metodi protected caratterizzati dal prefisso create, come createToolBar(), createMenuBar(), createCenterComponent() e createEditor(), i quali restituiscono il componente specificato dal loro nome. Questa scelta offre la possibilità di creare sottoclassi del programma che implementino una differente composizione della GUI semplicemente sovrascrivendo questi metodi, e lasciando inalterato il costruttore. Ridefinendo i metodi factory è possibile modificare in misura evidente l'aspetto dell'applicazione, aggiungendo un bordo alla Menu Bar, alla Tool Bar e al pannello centrale, senza bisogno di alterare il costruttore del programma:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class BorderedTextEditor extends TextEditor {
  protected JMenuBar createMenuBar() {
    JMenuBar mb = super.createMenuBar();
    mb.setBorder(new MatteBorder(7, 12, 7, 12, new ImageIcon("Texture_wood_004.jpg")));
    mb.setBackground(new Color(224, 195, 96));
    return mb;
  }
    protected JToolBar createToolBar() {
      JToolBar tb = super.createToolBar();
      tb.setBorder(new MatteBorder(7, 12, 7, 12, new ImageIcon("Texture_wood_004.jpg")));
      tb.setBackground(new Color(224, 195, 96));
      return tb;
    }
    protected JComponent createCenterComponent() {
      JComponent c = super.createCenterComponent();
      c.setBorder(new MatteBorder(7, 12, 7, 12, new ImageIcon("Texture_wood_004.jpg")));
      return c;
    }
  public static void main(String argv[]) {
    BorderedTextEditor t = new BorderedTextEditor();
  }
}
```

Figura 15.11 – *Sovrascrivendo i metodi factory è possibile modificare il comportamento di un'applicazione senza alterarne la struttura.*