Capitolo 12

La grafica in Java

Andrea Gini

Uno dei problemi più grossi emersi durante la progettazione di Java fu senza dubbio la realizzazione di un toolkit grafico capace di funzionare con prestazioni di buon livello su piattaforme molto differenti tra loro. La soluzione adottata nel 1996 fu AWT, un package grafico che mappa i componenti del sistema ospite con apposite classi dette peer, scritte in gran parte in codice nativo. In pratica, ogni volta che il programmatore crea un componente AWT e lo inserisce in un'interfaccia grafica, il sistema AWT posiziona sullo schermo un oggetto grafico della piattaforma ospite, e si occupa di inoltrare ad esso tutte le chiamate a metodo effettuate sull'oggetto Java corrispondente, ricorrendo a procedure scritte in buona parte in codice nativo; nel contempo, ogni volta che l'utente manipola un elemento dell'interfaccia grafica, un'apposita routine (scritta sempre in codice nativo) crea un apposito oggetto `Event` e lo inoltra al corrispondente oggetto Java, in modo da permettere al programmatore di gestire il dialogo con il componente e le azioni dell'utente con una sintassi completamente Object Oriented e indipendente dal sistema sottostante.

A causa di questa scelta progettuale, il set di componenti grafici AWT comprende solamente quel limitato insieme di controlli grafici che costituiscono il minimo comune denominatore tra tutti i sistemi a finestre esistenti: un grosso limite rispetto alle reali esigenze dei programmatori. In secondo luogo, questa architettura presenta un grave inconveniente: i programmi grafici AWT assumono un aspetto ed un comportamento differente a seconda della JVM su cui vengono eseguite, a causa delle macroscopiche differenze implementative esistenti tra le versioni di uno stesso componente presenti nelle diverse piattaforme. Spesso le interfacce grafiche realizzate su una particolare piattaforma mostrano grossi difetti se eseguite su un sistema differente, arrivando in casi estremi a risultare inutilizzabili.

Nel 1998, con l'uscita del JDK 1.2, venne introdotto il package Swing, i cui componenti erano stati realizzati completamente in Java, ricorrendo unicamente alle primitive di disegno più semplici, tipo "traccia una linea" o "disegna un cerchio", accessibili attraverso i metodi dell'oggetto Graphics, un oggetto AWT utilizzato dai componenti Swing per interfacciarsi con la piattaforma ospite. Le primitive di disegno sono le stesse su tutti i sistemi grafici, e il loro utilizzo non presenta sorprese: il codice java che disegna un pulsante Swing sullo schermo di un PC produrrà lo stesso identico risultato su un Mac o su un sistema Linux. Questa architettura risolve alla radice i problemi di uniformità visuale, visto che la stessa identica libreria viene ora utilizzata, senza alcuna modifica, su qualunque JVM. Liberi dal vincolo del "minimo comune denominatore", i progettisti di Swing hanno scelto di percorrere la via opposta, creando un package ricco di componenti e funzionalità spesso non presenti nella piattaforma ospite.

Applet e AWT

A quasi 8 anni dalla sua introduzione, Swing ha completamente rimpiazzato AWT nello sviluppo di applicazioni stand alone. Esiste tuttavia un contesto applicativo in cui l'uso di AWT risulta essere una scelta obbligatoria: la creazione di Applet Java compatibili con tutti i browser presenti sul mercato.

Un'Applet è un'applicazione Java incorporata in una pagina web, che viene scaricata assieme alla pagina ed eseguita da una JVM presente nel browser del client. Nel '96, quando Java venne introdotto sul mercato, la Netscape si accordò con la Sun Microsystems per incorporare una Java Virtual Machine nel proprio browser, allora leader di mercato. Questa decisione diede un enorme impulso alla diffusione del nuovo linguaggio, che permetteva di aggiungere alle pagine internet piccoli giochi, animazioni o vere e proprie applicazioni client. Nel '97, la Microsoft decise di adottare una politica aggressiva per conquistare la nascente Web Economy: con il lancio del browser Internet Explorer diede vita a quella che venne definita dalla stampa "La Guerra dei Browser". Verso la fine del '98, la Netscape, stremata dalla competizione, venne rilevata dal provider americano AOL, un evento che sancì la vittoria di Microsoft in una battaglia che purtroppo lasciò sul campo numerose vittime. Una delle vittime più illustri di questa competizione fu proprio Java, una tecnologia che la Microsoft intendeva ottimizzare per i propri sistemi operativi a discapito della natura multi piattaforma, vero e proprio asso nella manica del linguaggio della Sun Microsystem, Questa politica commerciale spinse la Sun Microsystems a chiamare in causa il gigante di Redmond, ottenendo una sentenza che imponeva alla Microsoft di rendere la propria Virtual Machine compatibile al 100% con le specifiche del linguaggio. La Microsoft a questo punto decise di sospendere lo sviluppo della Virtual Machine inclusa in Internet Explorer, che non venne aggiornata alle successive release del JDK. Data la posizione dominante di Microsoft nel campo dei browser, i programmatori Java sono costretti a sviluppare Applet conformi alla specifica 1.1 del linguaggio, la versione precedente all'introduzione di Swing, un fattore che ha contribuito all'arresto della diffusione di Applet nei siti web. Incidentalmente, la piattaforma Java ha finito per conquistare una posizione dominante come linguaggio per lo sviluppo di applicazioni lato Server, grazie, alle tecnologie Servlet, JSP ed EJB. Nel '97 un simile sviluppo era semplicemente impensabile, ma già in passato la tecnologia ha finito per intraprende delle strade non previste dai loro stessi creatori.

Le versioni più recenti dell'ambiente di sviluppo Java comprendono un framework per la creazione di Applet Swing, ma la loro esecuzione all'interno di un browser richiede l'installazione da parte dell'utente di un apposito plug-in da 10 MB, una circostanza che di fatto ne blocca la diffusione nel Web, a vantaggio di tecnologie alternative (Servlet, JSP, ASP e PHP) che permettono di realizzare siti Web dinamici senza problemi di compatibilità.

Queste considerazioni hanno condotto alla decisione di non trattare le tecnologie Applet ed AWT in questo manuale. Chi per ragioni professionali dovesse affrontare lo studio di tali tecnologie (esiste tuttora una grossa base installata di Applet introdotte nella seconda metà degli anni novanta in siti web tuttora attivi), non incontrerà comunque alcuna difficoltà, data l'enorme disponibilità di documentazione gratuita su web (anche in italiano), e la sostanziale somiglianza tra le API Swing ed AWT.

In questo capitolo verranno illustrati i principi della programmazione di interfacce grafiche in Java; a partire dal prossimo verranno introdotti i principali componenti grafici.

I top level container

I top level container sono i componenti all'interno dei quali si creano le interfacce grafiche: ogni programma grafico ne possiede almeno uno, di solito un JFrame, che rappresenta la finestra principale. Ogni top level container possiede un pannello (accessibile tramite il metodo getContentPane()) all'interno del quale vanno disposti i controlli dell'interfaccia grafica. Esistono due tipi principali di finestra: JFrame e JDialog. Il primo viene solitamente usato come finestra principale per il programma, mentre il secondo serve a creare le finestre di dialogo con l'utente. Queste classi sono presenti nel package javax.swing, che deve essere importato all'inizio dell'applicazione. Data la dipendenza di Swing da alcune classi presenti nel package AWT (principalmente la classe Graphics e i LayoutManager), è quasi sempre necessario necessario importare anche il package java.awt.

È sempre possibile impostare il titolo ricorrendo al metodo setTitle(String s). Due importanti proprietà dell'oggetto sono la dimensione e la posizione, che possono essere impostate sia specificando le singole componenti sia mediante oggetti Dimension e Point:

```
public void setSize(Dimension d)
public void setSize(int width, int height)
public void setLocation(Point p)
public void setLocation(int x,int y)
```

Tali proprietà possono essere impostate anche tramite il metodo setBounds(), che accetta come parametri sia una quadrupla di interi sia un oggetto Rectangle:

```
public void setBounds(int x, int y, int width, int height)
public void setBounds(Rectangle r)
```

Ricorrendo al metodo setResizable(boolean b) è possibile stabilire se si vuole permettere all'utente di ridimensionare la finestra manualmente. Infine, vi sono tre metodi piuttosto importanti:

```
public void pack()
public void setVisible(boolean b)
public void setDefaultCloseOperation(int operation)
```

Il primo ridimensiona la finestra tenendo conto delle dimensioni ottimali di ciascuno dei componenti presenti all'interno. Il secondo permette di visualizzare o di nascondere la finestra. Il terzo imposta l'azione da eseguire alla pressione del bottone close, con quattro impostazioni disponibili: WindowConstants.DO_NOTHING_ON_CLOSE (nessun effetto), WindowConstants.HIDE_ON_CLOSE (nasconde la finestra), WindowConstants.DISPOSE_ON_CLOSE (chiude la finestra e libera le risorse di sistema) e JFrame.EXIT_ON_CLOSE (chiude la finestra e conclude l'esecuzione del programma).

Un breve programma è sufficiente a illustrare come si possa: creare un JFrame; assegnargli un titolo, una posizione sullo schermo e una dimensione; stabilirne il comportamento in chiusura; renderlo visibile.

```
import javax.swing.*;

public class JframeExample {
   public static void main(String argv[]) {
      JFrame j = new JFrame();
      j.setTitle("JFrameExample");
      j.setBounds(10, 10, 300, 200);
      j.setDefaultCloseOperation(JFrame. EXIT_ON_CLOSE);
      j.setVisible(true);
   }
}
```

JDialog

Le finestre di dialogo si usano per consentire l'inserimento di valori, o per segnalare all'utente una situazione anomala. Ogni finestra di dialogo *appartiene* a un'altra finestra: se si chiude il frame principale, anche i JDialog di sua proprietà verranno chiusi. Se si definisce come *modale* un JDialog, alla sua comparsa esso bloccherà il frame di appartenenza, in modo da costringere l'utente a portare a termine l'interazione. Per creare una finestra di dialogo, è necessario specificare tra i parametri del costruttore il reference alla finestra principale, definire il titolo e indicare se si tratta o meno di una finestra modale:

```
JDialog(Dialog owner, String title, boolean modal)
JDialog(Frame owner, String title, boolean modal)
```

Esistono anche costruttori con un numero inferiore di parametri. I metodi presentati per JFrame sono validi anche con JDialog. Naturalmente, non è possibile selezionare l'opzione EXIT_ON_CLOSE con il metodo setDefaultCloseOperation().

Gerarchia di contenimento

Un'interfaccia grafica è composta da un top level container, tipicamente un JFrame, e da un insieme di componenti disposti al suo interno. Esistono alcuni componenti che hanno il preciso compito di fungere da contenitori per altri componenti. Il più usato di questi è senza dubbio JPanel, un pannello di uso estremamente generale. Con poche righe si può creare un JPanel e inserire un JButton al suo interno:

```
JPanel p = new JPanel();
JButton b = new JButton("Button");
p.add(Component c)
```

Più in generale, è possibile creare un JPanel, disporre alcuni controlli grafici al suo interno grazie al metodo add(Component c) e quindi inserirlo in un altro JPanel o nel content pane di un top level container. Un breve programma permetterà di illustrare meglio i concetti appena esposti e di introdurre i successivi: si tratta di un esempio abbastanza avanzato, e non c'è da preoccuparsi se alcuni aspetti all'inizio dovessero risultare poco chiari:

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

public class FirstExample {

  public static void main(String argv[]) {
    // Componenti
    JLabel label = new JLabel("Un programma Swing");
    JCheckBox c1 = new JCheckBox("Check Box 1");
    JCheckBox c2 = new JCheckBox("Check Box 2");
    JButton okButton = new JButton("OK");
    JButton cancelButton = new JButton("Cancel");

    // Pannello NORTH
    JPanel northPanel = new JPanel();
    northPanel.add(label);

    // Pannello CENTER
    JPanel centralPanel = new JPanel();
    centralPanel.setLayout(new GridLayout(0,1));
    centralPanel.setBorder(BorderFactory.createEmptyBorder(20, 20, 50, 50));
    centralPanel.add(c1);
    centralPanel.add(c2);

    // Pannello SOUTH
    JPanel southPanel = new JPanel();
    southPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
```

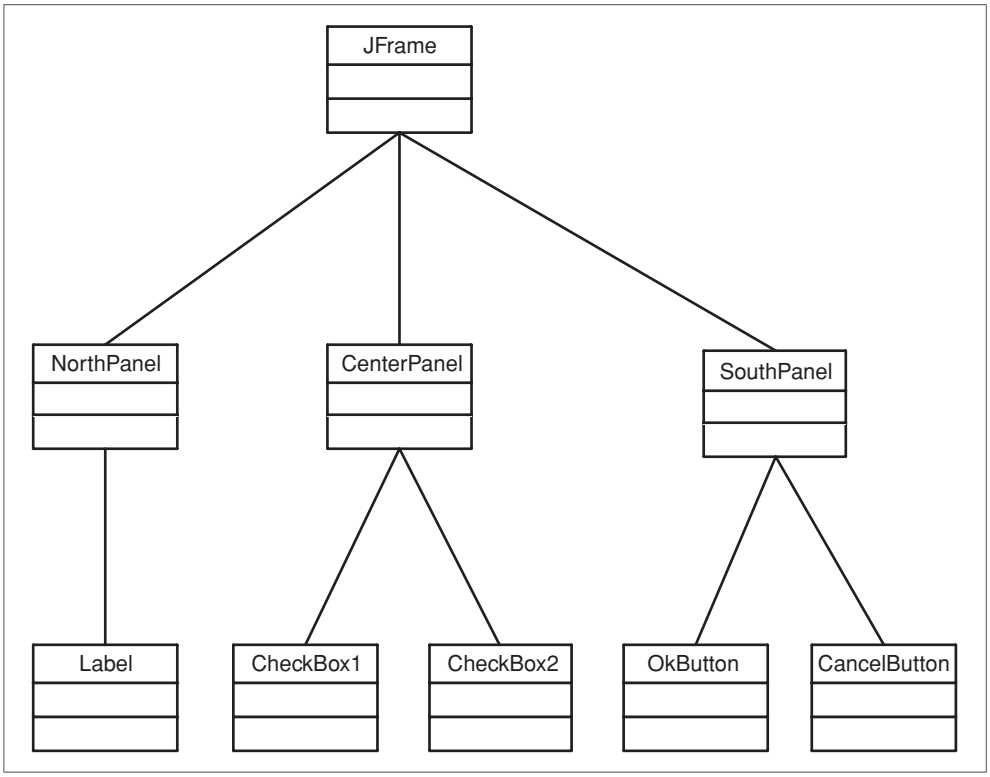
```
    southPanel.add(cancelButton);
    southPanel.add(okButton);

    // Top Level Container
    JFrame f = new JFrame("First Example");
    f.getContentPane().setLayout(new BorderLayout());
    f.getContentPane().add(BorderLayout.NORTH, northPanel);
    f.getContentPane().add(BorderLayout.CENTER, centralPanel);
    f.getContentPane().add(BorderLayout.SOUTH, southPanel);
    f.pack();
    f.setVisible(true);
  }
}
```

Figura 12.1 – *Un programma di esempio.*

Osservando il sorgente si può notare che dapprima vengono creati i componenti, poi sono montati su pannelli e infine tali pannelli vengano disposti all'interno di un JFrame. Per ogni componente vengono impostati alcuni attributi, come il bordo, il testo o il layout manager. È possibile rappresentare con un albero la disposizione gerarchica dei componenti di questo programma, come si vede in figura 12.2.

Figura 12.2 – *Gerarchia di contenimento dell'interfaccia grafica del programma di esempio.*



Layout management

Quando si dispongono i componenti all'interno di un contenitore sorge il problema di come gestire il posizionamento: infatti, sebbene sia possibile specificare le coordinate assolute di ogni elemento dell'interfaccia, queste possono cambiare nel corso della vita del programma allorquando la finestra principale venga ridimensionata.

Per semplificare il lavoro di impaginazione e risolvere questo tipo di problemi possibile ricorrere ai layout manager, oggetti che si occupano di gestire la strategia di posizionamento dei componenti all'interno di un contenitore.

Il metodo setLayoutManager(LayoutManager m) permette di assegnare un layout manager a ogni pannello. Combinando gli effetti dei layout manager possibile ottenere la disposizione desiderata.

FlowLayout

Questo semplice layout manager dispone i componenti in maniera ordinata da sinistra a destra e dall'alto in basso, assegnando a ciascun componente la dimensione minima necessaria a disegnarlo.

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

public class FlowLayoutExample {
  public static void main(String argv[]) {
    JFrame f = new JFrame();
    f.getContentPane().setLayout(new FlowLayout());
    f.getContentPane().add(new JButton("Primo"));
    f.getContentPane().add(new JButton("Secondo"));
    f.getContentPane().add(new JButton("Terzo"));
    f.getContentPane().add(new JButton("Quarto"));
    f.getContentPane().add(new JButton("Quinto"));
    f.pack();
    f.setVisible(true);
  }
}
```

Figura 12.3 – *FlowLayout dispone i componenti da sinistra a destra.*

Al termine di una riga, i componenti vengono inseriti nella successiva. In ogni riga, inoltre, i componenti vengono centrati. Si noti come ogni pulsante assuma una dimensione diversa, corrispondente alla sua dimensione minima.

Figura 12.4 – *Ridimensionando il contenitore, i componenti vengono disposti su più righe.*

possibile specificare nel costruttore un criterio di allineamento dei componenti diverso da quello di default: i valori possibili sono FlowLayout.LEFT, FlowLayout.CENTER e FlowLayout.RIGHT.

GridLayout

GridLayout suddivide il contenitore in una griglia di celle di uguali dimensioni. Le dimensioni della griglia vengono definite mediante il costruttore:

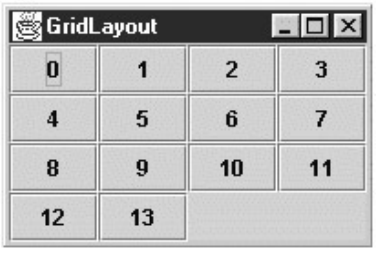
```
public GridLayout(int rows, int columns)
```

in cui i parametri rows e columns specificano rispettivamente le righe e le colonne della griglia. A differenza di quanto avviene con FlowLayout, i componenti all nterno della griglia assumono automaticamente la stessa dimensione, dividendo equamente lo spazio disponibile. Un semplice esempio permette di illustrare il funzionamento di questo pratico layout manager:

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

public class GridLayoutExample {
  public static void main(String argv[]) {
    JFrame f = new JFrame("GridLayout");
    f.getContentPane().setLayout(new GridLayout(4, 4));
    for ( int i = 0; i < 14; i++)
      f.getContentPane().add(new JButton(String.valueOf(i)));
    f.pack();
    f.setVisible(true);
  }
}
```

Figura 12.5 – *GridLayout.*

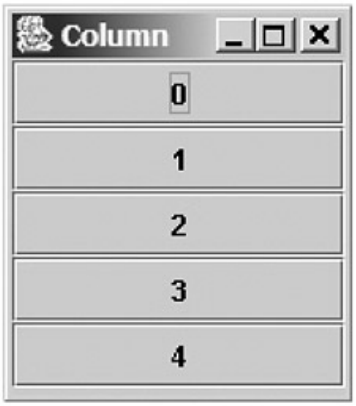


È possibile utilizzare GridLayout anche per disporre i componenti in riga o in colonna. Per ottenere questo, è sufficiente impostare i parametri row e column uno a 0 e l'altro a 1:

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

public class ColumnLayoutExample {
  public static void main(String argv[]) {
    JFrame f = new JFrame("Column");
    f.getContentPane().setLayout(new GridLayout(0, 1));
    for ( int i = 0; i < 5; i++)
      f.getContentPane().add(new JButton(String.valueOf(i)));
    f.pack();
    f.setVisible(true);
  }
}
```

Figura 12.6 – *Layout in colonna con GridLayout.*



BorderLayout

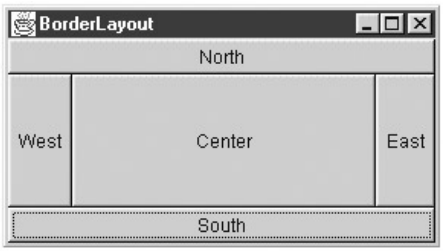
BorderLayout suddivide il contenitore esattamente in cinque aree, disposte a croce. Il programmatore può decidere in quale posizione aggiungere un controllo utilizzando il metodo add(component c,String s), presente nei Container, dove il primo parametro specifica il componente da aggiungere e il secondo indica la posizione. I valori validi per il secondo parametro sono le costanti BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.CENTER, BorderLayout.EAST e BorderLayout.WEST. Si osservi un esempio:

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

public class BorderLayoutExample {
```

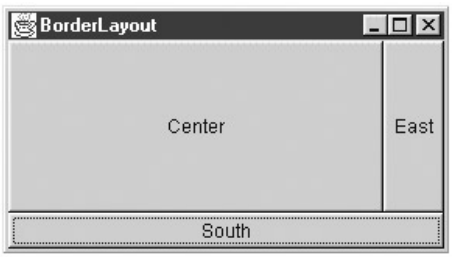
```
  public static void main(String argv[]) {
    JFrame f = new JFrame("BorderLayout");
    f.getContentPane().setLayout(new BorderLayout());
    f.getContentPane().add(new Button("North"), BorderLayout.NORTH);
    f.getContentPane().add(new Button("South"), BorderLayout.SOUTH);
    f.getContentPane().add(new Button("East"), BorderLayout.EAST);
    f.getContentPane().add(new Button("West"), BorderLayout.WEST);
    f.getContentPane().add(new Button("Center"), BorderLayout.CENTER);
    f.setSize(500,400);
    f.setVisible(true);
  }
}
```

Figura 12.7 – *Disposizione standard dei componenti all'interno di un pannello con BorderLayout.*



Si noti che non è obbligatorio riempire tutti e cinque gli spazi: se qualcuno di essi viene lasciato vuoto, i componenti presenti riempiranno lo spazio disponibile.

Figura 12.8 – *Comportamento di BorderLayout quando alcuni spazi vengono lasciati vuoti.*



BorderLayout particolarmente indicato per gestire l'impaginazione complessiva di una finestra. Infatti, la sua conformazione ripropone il layout di una tipica applicazione a finestre, con la

barra degli strumenti in alto, una barra di stato in basso, un browser a sinistra o a destra e il pannello principale al centro.

Progettazione top down di interfacce grafiche

Durante la progettazione delle interfacce grafiche, può essere utile ricorrere a un approccio top down, descrivendo l'insieme dei componenti a partire dal componente più esterno per poi procedere a mano a mano verso quelli più interni. Si può sviluppare una GUI come quella dell'esempio precedente seguendo questa procedura:

1. Si definisce il tipo di top level container su cui si vuole lavorare (tipicamente un JFrame).

2. Si assegna un layout manager al content pane del JFrame, in modo da suddividerne la superficie in aree più piccole.

3. Per ogni area messa a disposizione dal layout manager è possibile definire un nuovo JPanel. Ogni sotto pannello può utilizzare un layout manager differente.

4. Ogni pannello identificato nel terzo passaggio può essere sviluppato ulteriormente, creando al suo interno ulteriori pannelli o disponendo dei controlli.

Una volta conclusa la fase progettuale, si può passare a scrivere il codice relativo all'interfaccia: in questo secondo momento, è opportuno adottare un approccio bottom up, realizzando dapprima il codice relativo ai componenti atomici, quindi quello dei contenitori e infine quello del JFrame.

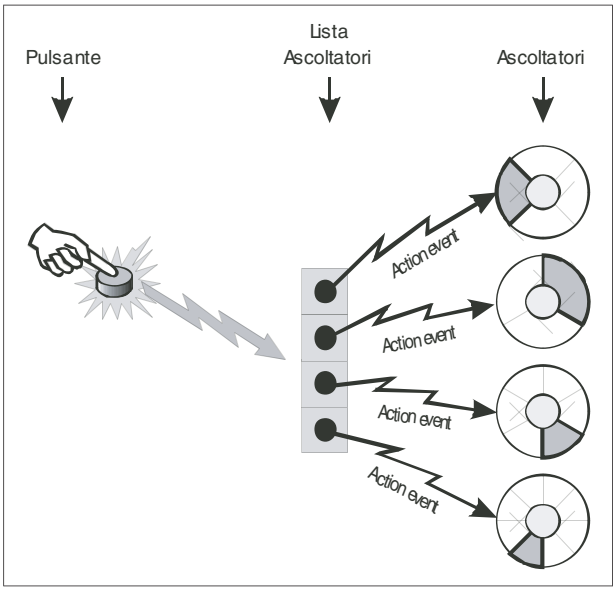
La gestione degli eventi

La gestione degli eventi grafici in Java segue il paradigma event forwarding (conosciuto anche come event delegation). Ogni oggetto grafico predisposto a essere sollecitato in qualche modo dall'utente genera eventi che vengono inoltrati ad appositi ascoltatori, che reagiscono agli eventi secondo i desideri del programmatore. l'event forwarding presenta il vantaggio di separare la sorgente degli eventi dal comportamento a essi associato: un componente *non sa* (e non è interessato a sapere) cosa avverrà al momento della sua sollecitazione: esso si limita a *notificare* ai propri ascoltatori che l'evento che essi attendevano è avvenuto, e questi provvederanno a produrre l'effetto desiderato.

Ogni componente dispone di un metodo addXxxListener() e removeXxxListener() per ogni evento supportato: per esempio, i pulsanti hanno i metodi addActionListener() e removeActionListener() per gestire gli ascoltatori di tipo ActionListener. Mediante questi metodi è possibile registrare un ascoltatore presso il componente. Nel momento in cui il componente viene sollecitato, esso chiama gli ascoltatori in modalità call back, passando come parametro della chiamata un'apposita sottoclasse di Event che contiene tutte le informazioni significative per l'evento stesso. Ogni ascoltatore è caratterizzato da una particolare interfaccia Java: grazie a questa particolarità,

qualsiasi classe può comportarsi come un ascoltatore, a patto che fornisca un'implementazione per i metodi definiti dalla corrispondente interfaccia listener.

Figura 12.9 – *Gestione degli eventi secondo il modello event forwarding.*



Le classi che permettono di gestire gli eventi generati dai componenti Swing sono in gran parte gli stessi utilizzati dai corrispondenti componenti AWT, e si trovano nel package java.awt.event. Alcuni componenti Swing, tuttavia, non hanno un omologo componente AWT: in questo caso le classi necessarie a gestirne gli eventi sono presenti nel package javax.swing.event,

Nel seguente esempio viene creato un pulsante; ad esso viene abbinato un ascoltatore grazie al metodo addActionListener(). Si noti che il gestore dell'evento è la classe EventHandlingExample stessa: essa infatti implementa la classe ActionListener e definisce il metodo actionPerformed(ActionEvent e), al cui interno si trova il codice che verrà richiamato in modalità call back ogni volta che il pulsante viene premuto.

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

public class EventHandlingExample extends JFrame implements ActionListener {

  public EventHandlingExample() {
    super("Eventi");
```

```
    setSize(400,300);
    JButton b = new JButton("Button");
    getContentPane().add(b);
    b.addActionListener(this);  // aggiunta dell'ascoltatore
  }
  // metodo call back per la gestione dell'evento
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this,"pulsante premuto");
  }
  public static void main(String argv[]) {
    EventHandlingExample e = new EventHandlingExample();
    e.setVisible(true);
  }
}
```

Ogni componente grafico può avere più di un ascoltatore per un determinato evento. In questo caso, gli ascoltatori saranno chiamati uno per volta secondo l'ordine in cui si sono registrati. Gli eventi generati dall'utente vengono accodati automaticamente. Se l'utente cerca di scatenare un nuovo evento prima che il precedente sia stato consumato (per esempio premendo ripetutamente un pulsante), ogni nuovo evento verrà accodato, e l'azione corrispondente sarà eseguita solo al termine della precedente.

Il sistema grafico di Java definisce un gran numero di ascoltatori: tra i più importanti vale la pena di segnalare:

- quelli che gestiscono gli eventi legati alle finestre (WindowFocusListener, WindowListener, WindowStateListener);
- quelli relativi ai contenitori (ComponentListener, ContainerListener, FocusListener);
- i listener per gli eventi del mouse (MouseListener, MouseMotionListener, MouseWheelListener);
- e quelli per la tastiera (KeyListener).

Nei prossimi capitoli verranno descritti i principali componenti Java, e per ogni componente verrà illustrato in dettaglio il relativo ascoltatore.

Uso di adapter nella definizione degli ascoltatori

Alcuni componenti grafici generano eventi così articolati da richiedere, per la loro gestione, ascoltatori caratterizzati da più di un metodo. Per esempio, l'interfaccia MouseMotionListener, che ascolta i movimenti del mouse, ne dichiara due, mentre MouseListener, che ascolta gli eventi relativi ai pulsanti del mouse, ne definisce tre. Infine, l'interfaccia WindowListener ne dichiara addirittura sette, ossia uno per ogni possibile cambiamento di stato della finestra:

```
public interface WindowListener extends EventListener {
   public void windowOpened(WindowEvent e);
   public void windowClosing(WindowEvent e);
   public void windowClosed(WindowEvent e);
   public void windowIconified(WindowEvent e);
   public void windowDeiconified(WindowEvent e);
   public void windowActivated(WindowEvent e);
   public void windowDeactivated(WindowEvent e);
}
```

Il linguaggio Java obbliga a fornire un'implementazione per ciascun metodo definito da un'interfaccia; se si desidera creare un ascoltatore interessato a un solo evento (per esempio uno che intervenga quando la finestra viene chiusa), esso dovrà comunque fornire un'implementazione vuota anche dei metodi cui non è interessato. Nei casi come questo è possibile ricorrere agli adapter: classi di libreria che forniscono un'implementazione vuota di un determinato ascoltatore. Per esempio, il package java.awt.event contiene la classe WindowAdapter, che fornisce un'implementazione vuota di tutti i metodi previsti dall'interfaccia. Una sottoclasse di WindowAdapter, pertanto, è un valido WindowListener, con in più il vantaggio di una maggior concisione:

```
class WindowClosedListener extends WindowAdapter {
  public void windowClosed(WindowEvent e) {
    // codice da eseguire alla chiusura della finestra
  }
}
```

All'interno dei package java.awt.event e javax.swing.event sono presenti adapter per ogni ascoltatore dotato di più di un metodo:

```
ComponentAdapter
ContainerAdapter
FocusAdapter
HierarchyBoundsAdapter
InternalFrameAdapter
KeyAdapter
MouseAdapter
MouseInputAdapter
MouseMotionAdapter
WindowAdapter
```

Classi anonime per definire gli ascoltatori

Le classi anonime forniscono un metodo molto sintetico per creare ascoltatori, utile nelle situazioni in cui sia necessario crearne parecchie decine. Le classi anonime sono classi prive di nome che vengono definite nello stesso momento in cui vengono create. Ecco un esempio di poche righe, che mostra come creare un pulsante e aggiungervi un ActionListener:

```
JButton b = new JButton("Button");
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this,"pulsante premuto");
  }
});
```

Si osservi come all'interno del metodo addActionListener() venga creata un'istanza di una classe di tipo ActionListener, il cui codice è contenuto tra le parentesi graffe che seguono. Una classe di questo tipo viene detta anonima proprio a causa del fatto che non ne viene definito il nome, ma solamente il corpo. Le classi anonime sono state introdotte nel linguaggio Java a partire dal JDK 1.1, proprio a supporto del sistema di event forwarding: esse, infatti, consentono di gestire in modo rapido numerose situazioni molto comuni nella programmazione a eventi.