Principi di base della programmazione di interfacce grafiche in Java

Riccardo Silvestri

4 giugno 2010

Introduzione

L'interfaccia grafica (*Graphical User Interface*, in breve *GUI*) facilita l'interazione tra l'utente e l'applicazione, cercando di rendere naturale e intuitivo l'accesso alle funzionalità offerte. Fin dagli inizi degli anni '80 i personal computers sono stati dotati di dispositivi come il mouse e le applicazioni hanno adottato interfacce utente basate su componenti grafici come finestre, menu, bottoni, ecc. La programmazione di tali interfacce è, in generale, più difficile rispetto a quella di interfacce testuali. Tutti i sistemi (Linux, Windows, MacOS X, ecc.) mettono a disposizione del programmatore delle librerie che forniscono i strumenti di base per la costruzione e la gestione di interfacce grafiche. Le librerie e le interfacce di un sistema non sono compatibili con quelle di un altro sistema, nel senso che non sono facilmente portabili da un sistema ad un altro. Una delle cause che ostacolano la portabilità deriva dal fatto che tali librerie sono implementate in modo nativo, cioè, sono fortemente dipendenti dall'architettura del sistema operativo sottostante. La piattaforma Java fornisce delle librerie che permettono la programmazione di interfacce grafiche portabili su qualsiasi sistema che ha una JVM.

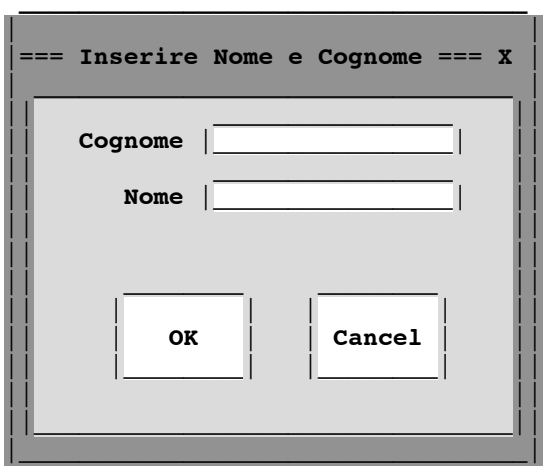
Già dalla sua prima versione (1.0) Java dispone di una libreria per la programmazione delle interfacce grafiche. La libreria si chiama *AWT* (*Abstract Window Toolkit*) e "garantisce" la portabilità delegando la creazione e la gestione dei componenti grafici dell'interfaccia alla libreria nativa del sistema su cui sta operando (ad es. Linux o MacOS X). Questo significa che la stessa applicazione Java se eseguita sotto il sistema Windows avrà una interfaccia grafica realizzata con i componenti grafici propri di Windows e se è eseguita sotto MacOS X avrà un'altra interfaccia realizzata con i componenti nativi di MacOS X. Ovviamente, passando da un sistema ad un altro, cambierà solamente l'aspetto visivo dei componenti, mentre le loro funzionalità dovrebbero rimanere invariate. Per garantire che le funzionalità rimangano invariate è necessario che ogni componente grafico (sopratutto se rappresenta un controllo, come un bottone, un menu, ecc.) abbia un comportamento che è condiviso dai vari sistemi. Però, le differenze tra i componenti nativi dei vari sistemi sono piuttosto ampie e per mantenere la portabilità è quindi necessario che la libreria di Java si basi solamente sulle caratteristiche comuni a tutti i sistemi. Questo porta ad avere una libreria (AWT) che su quasi tutti i sistemi non permette di usare al meglio i componenti grafici nativi. In altre parole, l'interfaccia grafica di un'applicazione Java che usa AWT non è all'altezza di un'interfaccia grafica di un'applicazione nativa (proprio perché non può sfruttare al meglio i componenti nativi). Inoltre, AWT contiene degli errori subdoli che si manifestano in modo differente nei diversi sistemi. Per queste ragioni l'approccio adottato dalla libreria AWT non dà una risposta adeguata al problema della portabilità delle interfacce grafiche.

Tuttavia, già dal 1996 Netscape aveva affrontato sostanzialmente lo stesso problema seguendo però un approccio differente da quello adottato da AWT. Invece di basarsi sui componenti grafici nativi, i componenti sono creati e gestiti direttamente dalla libreria. Così le sole librerie native usate sono quelle per la gestione di finestre, per il disegno in una finestra e per la ricezione degli eventi a basso livello. La libreria creata da Netscape si chiamava *IFC* (*Internet Foundation Classes*). Grazie

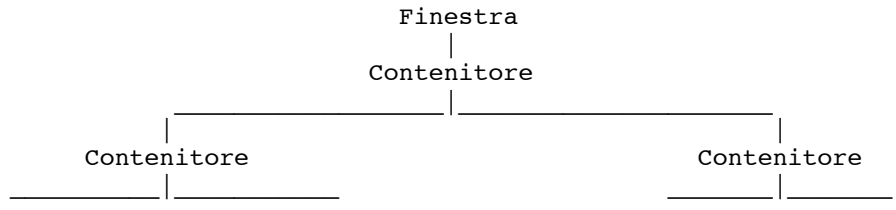
all'approccio adottato dalla libreria IFC l'interfaccia grafica appare e si comporta sempre nello stesso modo indipendentemente dal sistema su cui è eseguita. La Sun collaborò con Netscape per perfezionare tale approccio e introdusse, a partire dalla versione 1.2, la libreria *Swing*. Ora, Swing è parte delle *Java Foundation Classes* (*JFC*) che comprendono anche librerie per la grafica 2D e per il drag-and-drop. Purtroppo, la libreria Swing non sostituisce completamente AWT perché è costruita su di essa e, in particolare, la gestione degli eventi richiede l'uso esplicito di AWT. Nel seguito introdurremo i principi di base della programmazione di interfacce grafiche in Java tramite Swing. La libreria AWT sarà trattata limitatamente a ciò che è strettamente necessario per l'uso di Swing.

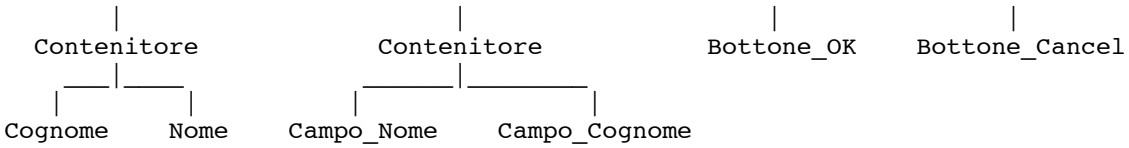
Architettura di Swing/AWT

Una interfaccia grafica consiste di vari elementi o componenti. Prima di tutto, c'è almeno una finestra. Questa costituisce un componente che potremmo considerare primario, nel senso che né dipende né è contenuto in altri componenti e anzi tutti gli altri componenti sono contenuti in una finestra. Come già accennato, le finestre sono fornite dal sistema sottostante e Swing si limita a "pilotarle" e a disegnarci sopra tutti gli altri componenti. Una finestra contiene tutti i componenti, come menu, bottoni, campi di testo ecc., che permettono di interagire con l'applicazione. Questi componenti sono creati e gestiti da Swing (e in parte da AWT). In realtà non è la finestra a contenerli direttamente ma è un componente contenitore (di tipo `Container`). Infatti, i componenti possono essere divisi in due categorie: i componenti atomici e i componenti contenitori. Un componente atomico non contiene altri componenti e assolve alla sua funzione direttamente senza bisogno di altri componenti (ad es. un bottone o un campo di testo). Un componente contenitore invece può contenere altri componenti e la sua funzione principale sta proprio nell'organizzare la visualizzazione dei componenti contenuti in esso (ad es. disponendoli in una linea orizzontale). I componenti che sono visualizzati in una finestra o sono contenuti nel contenitore della finestra o sono contenuti in altri componenti contenitori contenuti a loro volta nel contenitore della finestra a un qualsiasi livello di annidamento. Quindi i componenti visualizzati in una finestra sono, in generale, organizzati in un albero la cui radice è il componente contenitore della finestra, i nodi interni sono altri componenti contenitori e le foglie sono componenti atomici. Ad esempio, una finestra come quella qui sotto schematicamente mostrata



potrebbe avere i componenti organizzati come nel seguente albero:





I componenti atomici possono a loro volta essere suddivisi in due categorie: passivi e attivi. Quelli passivi non reagiscono alle azioni dell'utente (tasti premuti, click del mouse, ecc.). Esempi ne sono i componenti la cui unica funzione è la visualizzazione di scritte o immagini. I componenti attivi, come ad esempio bottoni, menu, campi di testo, ecc., sono invece in grado di rispondere alle azioni dell'utente. Quest'ultimi rappresentano la parte "viva" dell'interfaccia grafica. E sono anche i componenti più complessi. Un componente attivo ha tre caratteristiche:

- il *contenuto* (ad es. lo stato di un bottone, il testo di un campo);
- l'*aspetto visivo* (ad es. la forma, il colore, la dimensione, ecc.);
- il *comportamento* (come reagisce alle azioni dell'utente).

Chiaramente queste tre caratteristiche interagiscono fra loro. Ad esempio, lo stato di un bottone (premuto o no) influenza l'aspetto visivo e il comportamento influenza il contenuto (ad es. il mouse premuto sul bottone ne cambia lo stato). Tuttavia, le tre caratteristiche sono anche in larga misura indipendenti. Per organizzare il codice che gestisce queste tre caratteristiche e le loro interazioni Swing (e anche AWT) usa un ben conosciuto design pattern: *model-view-controller* (*MVC*).

Model-View-Controller Il design pattern MVC assegna la responsabilità di ognuna delle tre caratteristiche ad una apposita classe:

- il `Model`, che gestisce il contenuto;
- la `View`, che gestisce l'aspetto visivo;
- il `Controller`, che gestisce il comportamento.

Inoltre, MVC specifica precisamente come gli oggetti delle tre classi interagiscono. Tra i vari vantaggi di questo design pattern c'è la possibilità di fornire facilmente differenti views dello stesso contenuto (mantenendo quindi lo stesso `Model` e variando la `View` e forse anche il `Controller`) e di cambiare il *look and feel* dell'intera interfaccia grafica semplicemente agendo sulle `View` (senza quindi toccare `Model` e `Controller`). L'uso del design pattern MVC è, in molti casi, invisibile al programmatore che usa Swing perchè ogni componente (bottone, menu, campo di testo, ecc.) è rappresentato tramite un'unica classe che internamente usa le tre classi (`Model`, `View` e `Controller`) ma che espone una interfaccia programmativa pubblica unitaria (cioè, le cosidette API). Per i componenti più complesssi come quelli che gestiscono liste, tabelle, alberi o documenti con stili (ad es. RTF o HTML), l'interfaccia programmativa pubblica comprende anche l'accesso diretto al `Model`.

Gestione degli eventi I componenti attivi devono rispondere alle azioni dell'utente e queste sono rappresentate da *eventi*. Quando l'utente, ad esempio, fa click su un bottone ciò provoca la produzione, da parte di AWT, di un oggetto-evento che è comunicato, a cascata, ai vari componenti coinvolti: prima di tutto è comunicato alla finestra la quale lo comunicherà al contenitore diretto il quale lo comunicherà a un sotto-contenitore che contiene il bottone e così via lungo il cammino nell'albero dei componenti della finestra fino ad arrivare al componente atomico bottone. Arrivato al bottone, sempre AWT/Swing, si occuperà di aggiornare opportunamente il `Model` e la `View` del bottone tramite il `Controller` e infine l'evento potrà essere notificato all'applicazione (in realtà la notifica dell'evento all'applicazione può avvenire anche in congiunzione ad ogni componente coinvolto lungo il cammino a partire dalla finestra). Ma come viene notificato all'applicazione? Il meccanismo adottato dalle librerie AWT/Swing è quello prescritto dal design pattern chiamato *Observer*. La sua descrizione è molto semplice. I partecipanti sono due classi, che chiameremo `Subject` e `Observer`, e due corrispondenti

istanze *subjectInstance* e *observerInstance*:

Subject
: È la classe degli oggetti "osservati". Implementa una interfaccia per registrare e de-registrare gli "osservatori" (*Observer*).

Observer
: È la classe degli "osservatori". Implementa una interfaccia per la notifica di eventi.

subjectInstance
: Mantiene i riferimenti agli "osservatori" registrati. Quando si verifica un evento, invia una notifica agli "osservatori" registrati.

observerInstance
: Risponde alla notifica di eventi.

Quando si verifica un evento relativo ad un *subjectInstance* quest'ultimo lo notifica a tutti gli "osservatori" (*observerInstance*) registrati. Le comunicazioni tra gli oggetti delle classi *Subject* e *Observer* avvengono tramite una opportuna interfaccia per la notifica degli eventi, così che una classe per diventare un "osservatore" è sufficiente che implementi tale interfaccia. In Swing/AWT gli osservatori di eventi sono chiamati *listener* (cioè, ascoltatori). Consideriamo il caso di un bottone. La classe di Swing che rappresenta un bottone si chiama `JButton` (in `javax.swing`). La classe che rappresenta un evento relativo al "premere" un bottone (ma rappresenta anche molti altri tipi di eventi) si chiama `ActionEvent` (in `java.awt.event`). L'interfaccia per l'"ascolto" di eventi di questo tipo si chiama `ActionListener` (in `java.awt.event`) ed è così definita:

```
public interface ActionListener extends EventListener {
    void actionPerformed(ActionEvent e);
}
```

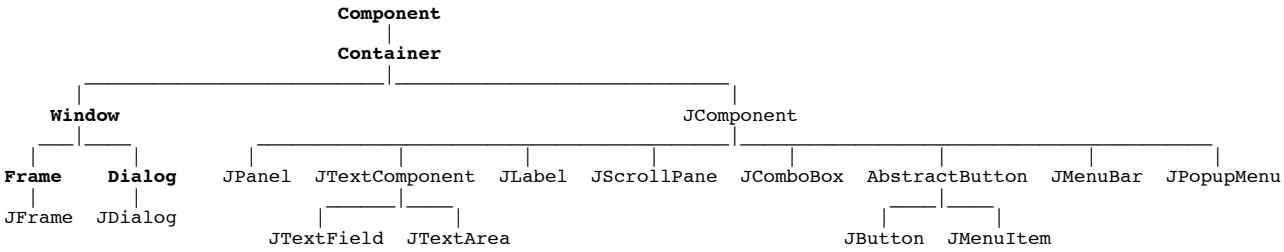
L'interfaccia `EventListener` è priva di metodi e serve solamente a marcare "ascoltatori di eventi". La classe `JButton` ha il seguente metodo:

```
public void addActionListener(ActionListener l)
```

per registrare un "ascoltatore". Se un oggetto *L* che implementa l'interfaccia `ActionListener` viene registrato, tramite il metodo `addActionListener()`, di un certo bottone *B*, quando il bottone *B* sarà "premuto" il metodo `actionPerformed(e)` di *L* sarà invocato con l'oggetto `e` contenente informazioni circa l'evento (la sorgente dell'evento, in questo caso il bottone *B*, i tasti modificatori premuti, il tempo dell'evento, ecc.). Quindi un'applicazione per gestire un bottone deve solamente istanziare un oggetto di tipo `JButton`, creare un oggetto che implementa opportunamente l'interfaccia `EventListener` e infine aggiungere l'oggetto come "ascoltatore" di `ActionEvent` del bottone.

Lo stesso meccanismo ma con interfacce e classi differenti è usato da Swing/AWT per tutti i tipi di eventi e tutti i tipi di componenti. Inoltre, è anche usato per notificare eventi relativi a cambiamenti di stato o di forma.

I componenti principali Il seguente diagramma mostra la gerarchia delle principali classi di Swing/AWT.



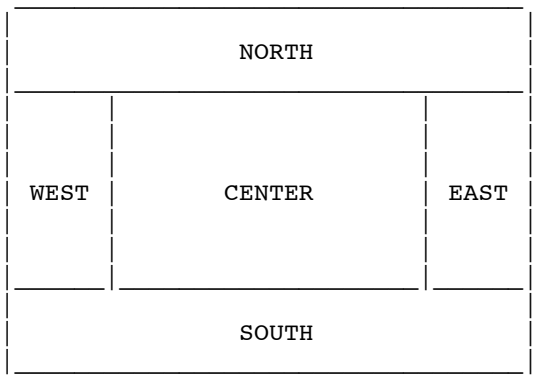
In grassetto sono evidenziate le classi di AWT (package `java.awt`) mentre le altre appartengono a Swing (package `javax.swing`). Il diagramma mostra solamente quelle più comuni, ma Swing offre molte altre classi: liste (`JList`), tabelle (`JTable`), alberi (`JTree`), documenti con stile (`JEditorPane`), vari tipi di bottoni (`JCheckBox`, `JRadioButton`, ecc.), ecc.

Facciamo ora una breve panoramica delle classi della gerarchia. Tutti i componenti (comprese le finestre) sono sottoclassi di `Component` che è la classe astratta di base. Essa definisce tutti i metodi (centinaia) che sono comuni ai vari componenti grafici: registrazione di "ascoltatori" relativamente a vari tipi di eventi (movimenti del mouse, tasti, focus, ecc.), dimensioni, allineamento, colori, ecc. Poi c'è la sottoclasse (concreta) `Container` che rappresenta la base dei componenti contenitori. Come si può notare tutti i componenti, compresi i componenti atomici, come bottoni, campi di testo ecc., sono sottoclassi di `Container`. Questo può sembrare strano ma è un effetto del fatto che Java non ammette l'ereditarietà multipla di classi. La classe `Container` definisce parecchie decine di metodi che sono comuni ai vari tipi di contenitori: aggiunta e rimozione di componenti, impostazione del *layout* (cioè, la configurazione visiva dei componenti), e tanti altri. La classe `Window` è la base di tutte le classi che rappresentano finestre (comprese le cosidette finestre di dialogo). Mentre la classe astratta `JComponent` è la base di tutti i componenti e i contenitori di Swing (ridefinisce molti metodi delle superclassi `Component` e `Container`). Diamo ora una breve descrizione di ognuna delle classi rimanenti.

`Frame` e `JFrame`
: La classe `Frame` di AWT aggiunge i bordi e la barra di menu alla finestra rappresentata dalla classe base `Window`. Mentre la sottoclasse `JFrame` rende la finestra adatta per essere usata nel framework di Swing.

`Dialog` e `JDialog`
: Una finestra di dialogo è una finestra specializzata per comunicare all'utente delle informazioni o per ricevere qualche tipo di input sempre dall'utente. La classe `Dialog` è la versione di AWT mentre `JDialog` è quella di Swing.

`JPanel`
: Rappresenta un contenitore di Swing. I costruttori permettono di impostare un `LayoutManager` (un'interfaccia in `java.awt`) per il pannello `JPanel`. Un `LayoutManager` determina quale sarà la configurazione visiva dei componenti (direttamente) contenuti nel pannello. Il layout di default è il `FlowLayout` (una classe che implementa l'interfaccia `LayoutManager`) che dispone i componenti in un flusso continuo come parole in un testo. Uno dei layout più utili è il `BorderLayout` che permette di disporre i componenti in 5 zone (`NORTH`, `SOUTH`, `WEST`, `EAST` e `CENTER`) illustrate nel seguente schema:



Quando il pannello viene ridimensionato (ad esempio, quando la finestra viene allargata), le zone `NORTH` e `SOUTH` crescono orizzontalmente, le zone `WEST` e `EAST` crescono verticalmente e la zona

`CENTER` cresce sia orizzontalmente che verticalmente. Ci sono molti altri `LayoutManager` alcuni dei quali piuttosto sofisticati.

`JTextComponent`, `JTextField` e `JTextArea`
: La classe astratta `JTextComponent` (in `javax.swing.text`) e la base di tutti i componenti di Swing per gestire il testo. La sottoclasse `JTextField` (in `javax.swing`) è un componente che permette di gestire la visualizzazione e l'editing di una linea di testo. Mentre la sottoclasse `JTextArea` gestisce la visualizzazione e l'editing di un testo su più linee.

`JLabel`
: Una classe per visualizzare una scritta e/o una piccola immagine (icona). È un componente passivo che non reagisce alle azioni dell'utente.

`JScrollPane`
: Fornisce una vista su un'altro componente anche con l'ausilio di scroll bars.

`JComboBox`
: Un componente che, quando attivato, visualizza una lista di elementi che l'utente può selezionare.

`AbstractButton` e `JButton`
: La classe astratta `AbstractButton` (in `javax.swing`) è la base per tutti i tipi di bottoni e le voci di menu (`JMenuItem`). La sottoclasse `JButton` rappresenta il tipo di bottone più comune.

`JMenuItem` e `JMenu`
: La classe `JMenuItem` gestisce la voce di un menu. Infatti, la voce di un menu è vista come un bottone che risiede in una lista (cioè il menu). La sottoclasse `JMenu` gestisce un menu: sostanzialmente è un bottone che quando premuto fa apparire una lista di voci di menu (che è un `JPopupMenu`). Potrebbe sembrare strano che `JMenu` sia una sottoclasse di `JMenuItem` ma in effetti ciò facilita la creazione e gestione di sotto-menu (cioè, voci di menu che rendono visibili altri menu).

`JMenuBar`
: Rappresenta una barra di menu. Di solito è associata ad una finestra (`JFrame`). I menu mantenuti dalla barra sono di tipo `JMenu`.

`JPopupMenu`
: Gestisce un popup menu, cioè una piccola finestra che appare quando attivata e che mostra una serie di scelte (generalmente, dei `JMenuItem`).

Le librerie Swing e AWT offrono, come si è detto, molte altre classi, oltre a quelle menzionate, per la programmazione di interfacce grafiche.

Event Dispatch Thread C'è ancora un importante aspetto di Swing che bisogna sempre tener presente. La libreria Swing non è *thread safe*. Cosa significa questo? Prima di tutto, bisogna ricordare che Java è un linguaggio multi-threading il che significa che in un programma Java possono essere eseguiti concorrentemente più processi o threads. I threads fra loro possono comunicare ma bisogna immaginarli come se fossero dei programmi diversi che vengono eseguiti "simultaneamente" sulla stessa macchina indipendentemente l'uno dall'altro (se la macchina ha un solo processore, questo alternerà l'esecuzione di un po' di istruzioni di un thread con l'esecuzione di un po' di istruzioni di un altro thread). Tutta la gestione interna degli eventi di Swing (movimenti del mouse, tasti premuti, ecc.) e tutti gli aggiornamenti automatici della visualizzazione dei componenti grafici sono effettuati in un thread specializzato che si chiama *Event Dispatch Thread* (*EDT*). Un programma che usa componenti di Swing automaticamente innesca l'esistenza dell'EDT. Così conviveranno il main thread (quello in cui è eseguito il metodo `main()`) e l'EDT. Il guaio è che Swing non è thread safe e questo comporta che se la manipolazione dei componenti di Swing avviene in un thread differente dall'EDT, si corre il rischio che l'interfaccia grafica non si comporti come ci si aspetta. Per garantire che tutte le manipolazioni di componenti di Swing avvengano nell'EDT, si può usare il seguente metodo statico

```
public static void invokeLater(Runnable doRun)
```

della classe `SwingUtilities` (sempre in `javax.swing`) che fa sì che il metodo `doRun.run()` sia invocato in modo asincrono nell'EDT (`Runnable` è una interfaccia con il solo metodo `run()`, in `java.lang`). Allora, è sufficiente che la classe che definisce il main dell'applicazione segua lo schema:

```
public class MyApp {

    . . .

        //Sarà invocato nell'EDT
    public void createAndShowGUI() {
        //Crea e inizializza tutti componenti dell'interfaccia grafica
        //e li visualizza.
    }

    public static void main(String[] args) {
            //Fa sì che un compito sia eseguito in modo asincrono nell'EDT
        SwingUtilities.invokeLater(new Runnable() {
                //Il compito che sarà eseguito nell'EDT
            public void run() {
                new MyApp().createAndShowGUI();
            }
        });
    }
}
```

Con questo schema il main thread avrà una vita molto breve perché terminerà subito dopo aver invocato il metodo `invokeLater()` (che ritorna immediatamente). Dopo di ciò rimarrà in vita solamente l'EDT nel quale sarà eseguito il codice che crea, inizializza e rende visibili tutti i componenti dell'interfaccia grafica. La successiva interazione dell'utente con l'iterfaccia grafica avverrà sempre nell'EDT, fino alla terminazione dell'applicazione che, tipicamente, sarà provocata dalla chiusura dell'ultima finestra dell'applicazione.