SAPIENZA Università di Roma

a.a. 2010-2011

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Ingegneria Informatica ed Automatica
Corso di Laurea in Ingegneria dei Sistemi Informatici

Esercitazioni di Progettazione del Software

Massimo Mecella

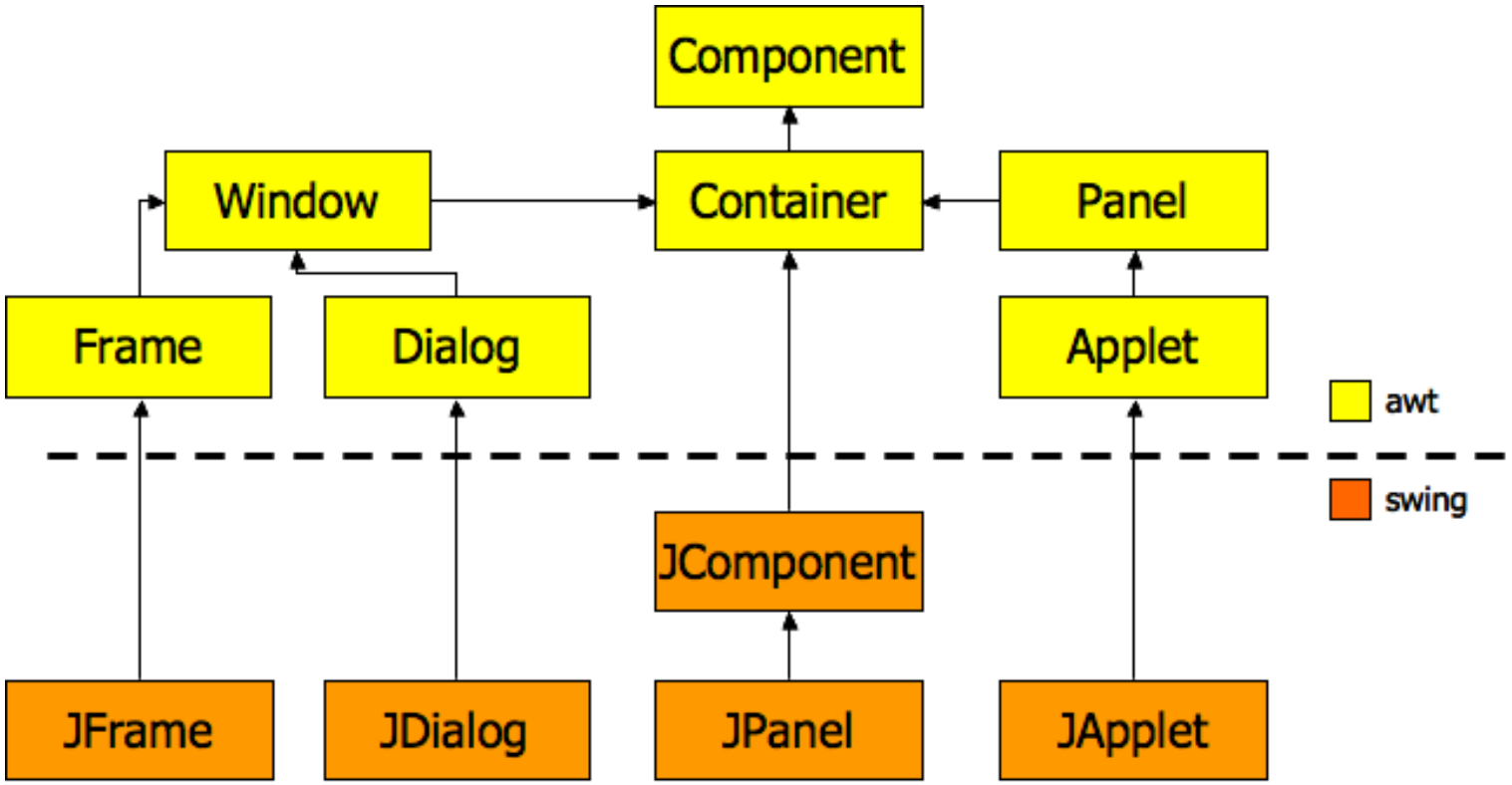
Sviluppo di Interfacce Grafiche in Java con il package Swing

(slide preparate a partire da versioni precedenti di Massimiliano de Leoni, Claudio Di Ciccio, Fabio Patrizi)

Il package Swing

- `javax.swing.*`
- è il package standard per lo sviluppo di interfacce grafiche
- deriva da `java.awt.*` (Abstract Window Toolkit)
- sviluppato per superare i problemi di portabilità di AWT su JVMs diverse
- elementi Swing possono essere usati al posto di AWT

Il package Swing: la gerarchia di base



I Top-Level Container

- Ogni interfaccia grafica è identificata da un Top-Level container
- Un Container è un tipo di `Component` che contiene e manipola altri componenti
- Ne esistono 4: `JApplet`, `JDialog`, `JFrame` (v. figura), `JWindow`
- Un T.L.C. contiene tutti gli elementi dell'interfaccia (e.g., titolo, pulsanti, checkboxes, aree di testo)
- JFrame rappresenta una finestra del programma
- Ci focalizziamo su `JFrame`

La classe `JFrame`

Esempio:

```
import javax.swing.*;

public class GUIApp {
        public static void main(String[] args){
                // Crea una finestra con titolo "Hello World"
                JFrame frm = new JFrame("Hello World!");
                // Rende la finestra visibile (di default non lo e')
                frm.setVisible(true);
                System.out.println("Ciao ciao");
        }
}
```

Cosa succede?

1. viene creato l'oggetto `frm` di classe `JFrame` la cui finestra corrispondente è inizialmente *nascosta*;

2. la finestra viene resa visibile;

3. il metodo `main` **continua la propria esecuzione** e stampa su schermo `Ciao ciao`

4. il metodo `main` **rimane in uno stato di attesa** (se terminasse dovrebbe chiudere la finestra attiva)

Chiusura di un programma con interfaccia grafica

- in questo corso identificheremo un'applicazione grafica con il frame principale

- alla chiusura del frame principale vogliamo far corrispondere la chiusura dell'applicazione

- per fare ciò aggiungiamo l'istruzione:

  ```
  frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  ```

  con cui comunichiamo al metodo `main` che deve terminare *incondizionatamente* quando il **pulsante di chiusura del frame principale viene premuto**

Esempio

```
import javax.swing.*;

public class GUIApp {
        public static void main(String[] args){
                // Crea una finestra con titolo "Hello World"
                JFrame frm = new JFrame("Hello World!");

                // Comunica al metodo main() che deve terminare quando
                // il pulsante di chiusura del frame viene attivato
                frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                // Rende la finestra visibile (di default non lo e')
                frm.setVisible(true);
                System.out.println("Ciao ciao");
        }
}
```

Chiusura delle finestre

Il metodo `setDefaultCloseOperation()` imposta il comportamento da adottare quando **il pulsante di chiusura di un frame viene attivato**

Abbiamo 4 possibili alternative, corrispondenti all'assegnazione di una delle seguenti costanti come parametro attuale del metodo:

1. `JFrame.DO_NOTHING_ON_CLOSE`: l'attivazione del pulsante di chiusura della finestra non ha alcun effetto

2. `JFrame.HIDE_ON_CLOSE`: nasconde la finestra, **mantenendola in memoria** per possibili accessi futuri (e.g., usando `setVisible(true)`)

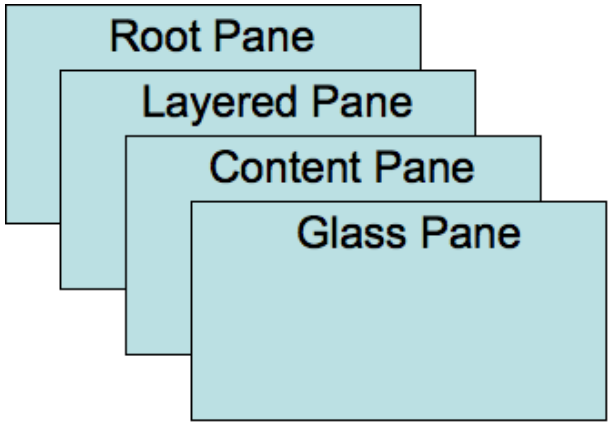
3. `JFrame.DISPOSE_ON_CLOSE`: elimina la finestra **liberando la memoria allocata** (non sarà più possibile accedervi)

4. `JFrame.EXIT_ON_CLOSE`: forza la chiusura incondizionata del programma (corrisponde a `System.exit(0)`)

NOTA: in realtà, il metodo `main()` rimane nello stato di attesa finché esiste una finestra in memoria. Potremmo anche non definire un frame principale ed assegnare a tutte le finestre il comportamento 3. In tal modo, il `main()` terminerebbe automaticamente alla chiusura dell'ultima finestra (l'argomento è piuttosto avanzato e non sarà affrontato in dettaglio)

Struttura a strati di un oggetto `JFrame`

Per creare finestre più ricche con `JFrame`, che contengano pulsanti, aree di testo, caselle opzione, etc., occorre accedere al `Content Pane` dell'oggetto `JFrame`.



- Ciascuno strato svolge una funzione particolare;
- Per i nostri scopi è sufficiente il Content Pane;
- Gli altri strati permettono di creare funzionalità avanzate (non usate in questo corso);

Popolare un oggetto `JFrame`

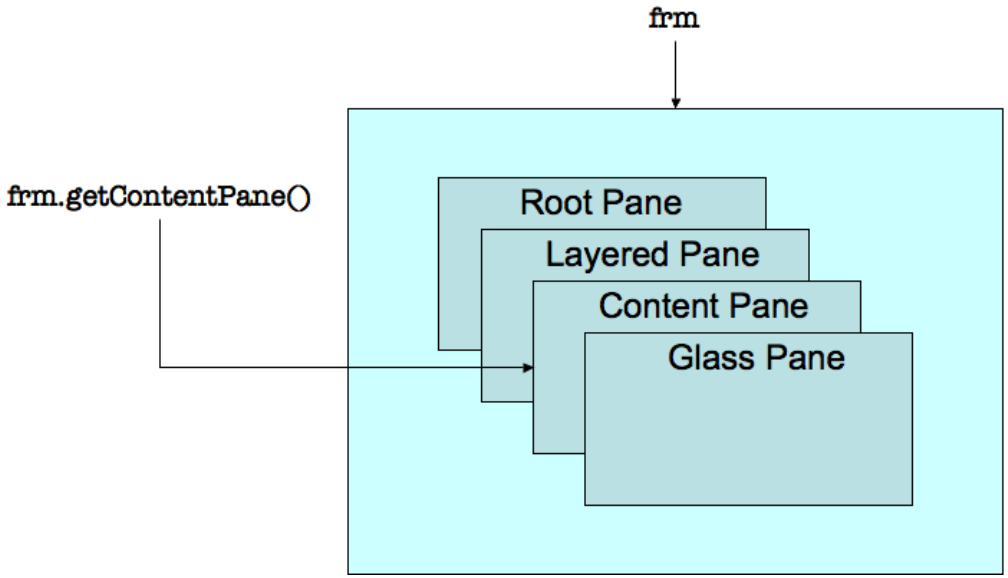
- Popolare un oggetto `JFrame` significa aggiungervi nuovi elementi grafici quali pulsanti, aree di testo, etc.

- Un oggetto `JFrame` viene popolato aggiungendo elementi grafici al suo Content Pane

- Per fare ciò dobbiamo innanzitutto ottenere un riferimento al Content Pane del JFrame

- Il metodo `getContentPane()` della classe `JFrame` svolge esattamente questa funzione, restituendo un riferimento ad un oggetto, il content Pane, appunto, di tipo `Container` (definito nel package AWT, v. sopra)

Popolare un oggetto `JFrame` (cont.)



Possiamo ora aggiungere elementi all'oggetto `frm` tramite il metodo di classe `Container`:

```
Component add(Component c)
```

che aggiunge il `Component` passato come argomento al Content Pane oggetto di invocazione

Popolare un oggetto `JFrame` (cont.)

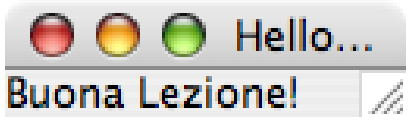
```
import javax.swing.*;
// Importa AWT per usare la classe Container
import java.awt.*;

public class GUIApp {
        public static void main(String[] args){
                JFrame frm = new JFrame("Hello World!");
                frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                // Ottieni il riferimento al Content Pane
                Container frmContentPane = frm.getContentPane();

                // Usa frmContentPane per aggiungere elementi grafici:
                frmContentPane.add(new JLabel("Buona Lezione!"));

                frm.setVisible(true);
        }
}
```



NOTA: le dimensioni della finestra sono quelle di default ed il titolo non vi è contenuto per intero

Metodi essenziali della classe `JFrame`

Principali metodi della classe `JFrame`, alcuni dei quali ereditati da classi antenate (`Component`, `Container`, `Window`, `Frame`):

- `JFrame()`: costruttore senza argomenti, crea un oggetto di classe `JFrame` senza titolo e ne restituisce il riferimento
- `JFrame(String titolo)`: costruttore ad un argomento, crea un oggetto di classe `JFrame` con titolo `titolo` e ne restituisce il riferimento
- `setTitle(String titolo)`: ereditato da `Frame`, assegna il titolo all'oggetto di invocazione
- `void setSize(int lrg, int alt)`: ereditato da `Component`, assegna le dimensioni specificate (misurate in px) alla finestra
- `void setLocation(int x, int y)`: ereditato da `Component`, assegna la posizione specificata (coordinate in px), con origine in alto a sinistra
- `void pack()`: ereditato da `Window`, ottimizza le dimensioni della finestra, in base alle *dimensioni preferite dei suoi contenuti*

- `void setVisible(boolean b)`: ereditato da `Component`, visualizza (`b==true`) o nasconde (`b==false`) la finestra

- `void setDefaultCloseOperation(int op)`: imposta il comportamento da adottare all'attivazione del pulsante di chiusura della finestra

Altri metodi saranno introdotti nel seguito mediante esempi

Qui riportiamo solo quelli utili ai fini del corso

Per una lista esaustiva, consultare il sito ufficiale `java.sun.com` (in inglese)

Esempio

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

public class GUIApp{
        public static void main(String args[]){
                JFrame frm = new JFrame("Prima finestra");
                frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frm.setSize(200,200);
                Container frmContentPane = frm.getContentPane();
                frmContentPane.add(new JLabel("Buona Lezione"));
                frm.setVisible(true);
        }
}
```

Modularizzazione delle interfacce grafiche

Vogliamo evitare di fare gestire al modulo `main()` tutti gli aspetti dell'interfaccia grafica

Per fare ciò, decidiamo di **costruire una classe per ogni finestra** e delegare a ciascuna di esse le operazioni interne che conivolgono dettagli implementativi

Le nuove classi saranno ottenute estendendo `JFrame`, in modo da averne a disposizione tutte le funzionalità

NOTA: La progettazione delle interfacce grafiche è un argomento piuttosto articolato che esula dagli scopi del corso. Qui, proponiamo un approccio semplificato.

Modularizzazione delle interfacce grafiche (cont.)

Con questo approccio, l'esempio visto sopra diventa:

```
// File MyFrame.java
import javax.swing.*; // AWT non serve (perche'?)
public final class MyFrame extends JFrame{
        // Costanti (no "numero magici"!)
        private static final String titolo = "Prima Finestra";
        private static final JLabel testo = new JLabel("Buona Lezione");
        private static final int larghezza=200, altezza=200;

        public MyFrame(){
                super(titolo);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setSize(larghezza,altezza);
                this.getContentPane().add(testo);// Abbreviato: evitiamo il Container
                this.setVisible(true);
        }
}

//File GUIApp.java
public class GUIApp {
        public static void main(String[] args){
                MyFrame myFrm = new MyFrame();
        }
}
```

Layout Manager

La disposizione di default degli elementi in un container è definita all'interno del container stesso

Per intervenire su di essa usiamo oggetti di classe `LayoutManager`

Un `LayoutManager` è un'interfaccia contenente un insieme di funzionalià dedicate al posizionamento degli elementi all'interno di un `Container`

Esistono numerose implementazioni predefinite di `Layout Manager` (cf. `java.sun.com`), ciascuna corrispondente ad una politica di posizionamento degli elementi. Qui ne vedremo solo alcune (`FlowLayout`, `GridLayout`, `BorderLayout`)

Non abbiamo bisogno di conoscere i dettagli interni dei `LayoutManager`: ci limiteremo semplicemente ad usarli!

Ogni `Container` ha un'implementazione di `LayoutManager` associata che ne definisce la disposizione di default. Tipicamente, gli oggetti sono disposti da sinistra verso destra, secondo l'ordine di inserimento

Layout Manager (cont.)

Per assegnare un particolare `LayoutManager` (diverso da quello di default), la classe `Container` mette a disposizione il metodo:

```
void setLayout(LayoutManager)
```

Ad esempio, se vogliamo disporre gli elementi secondo la politica definita dal `FlowLayout`:

```
//...
frm = new JFrame(titolo);
Container frmContentPane = frm.getContentPane();
frm.setLayout(new FlowLayout());
//...
```

FlowLayout

Inserisce gli elementi "riga per riga"

Costruttori: `FlowLayout()`, `FlowLayout(int align)`

Il Layout Manager `FlowLayout()` realizza la seguente politica di posizionamento:

- ogni elemento viene inserito in una riga finché vi è sufficiente spazio
- il primo elemento che non ha spazio viene posizionato nella riga successiva

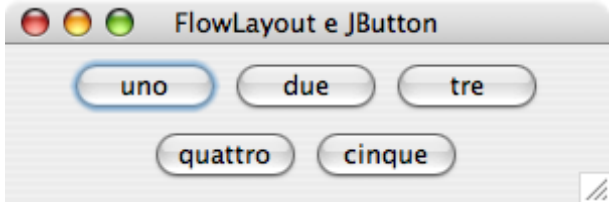
Se viene usato il costruttore ad un argomento, il parametro `align` può assumere i valori: `FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`, che allineano ciascuna riga, rispettivamente a sinistra, al centro o a destra

Il costruttore senza argomenti allinea al centro

Esempio

```
public final class MyFrame extends JFrame{
        private static final String titolo = "FlowLayout e JButton";
        private static final int larghezza=300, altezza=100;
        private static final JButton uno = new JButton("uno");
        // due, tre, quattro
        private static final JButton cinque = new JButton("cinque");

        public MyFrame(){
                super(titolo);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setSize(larghezza,altezza);
                Container frmContentPane=this.getContentPane();
                frmContentPane.setLayout(new FlowLayout());
                frmContentPane.add(uno);
                // due, tre, quattro
                frmContentPane.add(cinque);
                this.setVisible(true);
        }
}
```



GridLayout

Dispone gli elementi secondo una griglia

Costruttore: `GridLayout(int righe, int colonne)`

Il Layout Manager `GridLayout()` posiziona gli elementi in una griglia, riga per riga, da sinistra verso destra e dall'alto verso il basso:

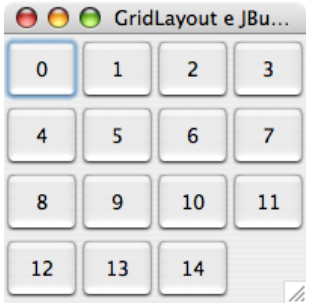
1	2	3
4	5	6
7	8	9

Tutti gli elementi hanno **stessa** dimensione, stabilita dall'elemento di dimensione massima e comunque limitata dalla dimensione della finestra

Esempio

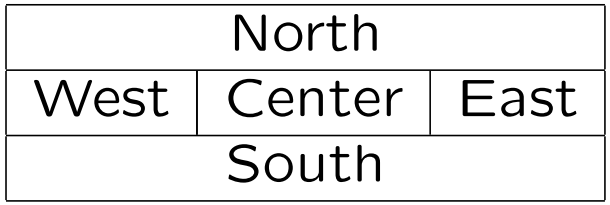
```
public final class MyFrame extends JFrame{
        private static final String titolo = "GridLayout e JButton";
        private static final int larghezza=200, altezza=200;
        private static final int righe=4, colonne=4;

        public MyFrame(){
                super(titolo);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setSize(larghezza,altezza);
                Container frmContentPane=this.getContentPane();
                frmContentPane.setLayout(new GridLayout(righe,colonne));
                for(int i = 0; i<15; i++){
                        frmContentPane.add(new JButton(String.valueOf(i)));
                }
                this.setVisible(true);
        }
}
```



BorderLayout

Permette di disporre gli elementi suddividendo il `Container` in cinque aree:



Costruttore: `BorderLayout()`

Con il metodo `void add(Component c, String s)` si seleziona l'area a cui assegnare un elemento

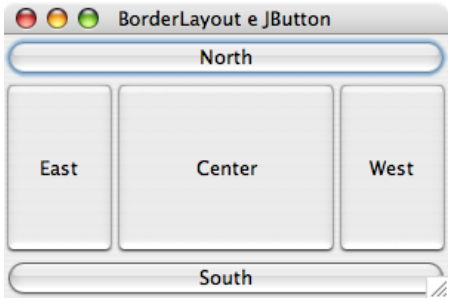
Aree senza elementi non vengono visualizzate e la finestra viene adattata di conseguenza

Il parametro `s` può assumere i valori: `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.EAST`

Esempio

```
public final class MyFrame extends JFrame{
        private static final String titolo = "BorderLayout e JButton";
        private static final int larghezza=300, altezza=200;
        private static final JButton north = new JButton("North");
        // south, east,...

        public MyFrame(){
                super(titolo);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setSize(larghezza,altezza);
                Container frmContentPane=this.getContentPane();
                frmContentPane.setLayout(new BorderLayout());
                frmContentPane.add(north,BorderLayout.NORTH);
                frmContentPane.add(south,BorderLayout.SOUTH);
                // etc...
                this.setVisible(true);
        }
}
```



Container Annidati

Per creare finestre più complesse si possono aggiungere container diversi, ciascuno con un proprio `LayoutManager` che ne definisce la disposizione degli elementi

Per fare ciò usiamo la classe `JPanel`

Un oggetto di classe `JPanel` è un `JComponent` che pu essere aggiunto ad un `JFrame`, in particolare al suo Content Pane

Essendo anche un oggetto di tipo `Container`, un `JPanel` può contenere, a sua volta, elementi grafici

Infine, anche un oggetto `JPanel` può avere un proprio `LayoutManager`

Progettazione della GUI

Adottiamo un approccio **Top-Down** alla progettazione: partendo dal contenitore principale, procediamo verso i contenitori più interni

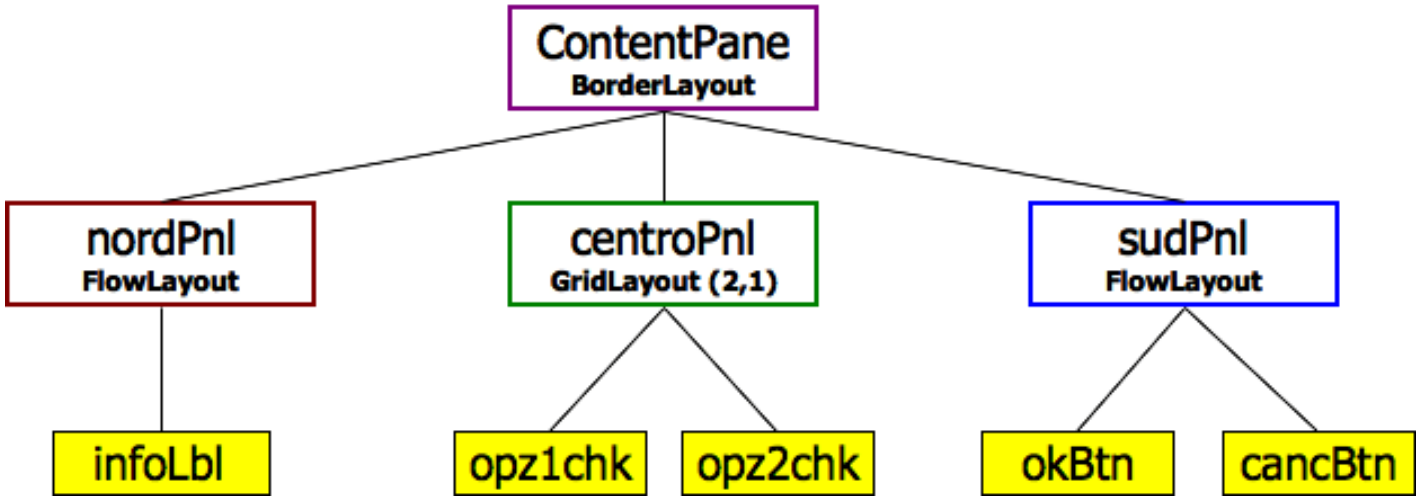
1. Si assegna un Layout Manager al Content Pane del `JFrame` che rappresenta la finestra da realizzare (il default è `BorderLayout`)

2. Per ciascuna area che si vuole aggiungere viene creato un `JPanel`, assegnando a ciascuno di essi un Layout Manager (il default è `FlowLayout` con allineamento centrale)

3. Ciascun pannello potrà contenere a sua volta altri pannelli o elementi grafici più semplici (`JButton`, `JLabel`, etc.)

Rappresentiamo il risultato di questa fase con un **albero della GUI** annotato con informazioni utili alla realizzazione (e.g., Layout Manager adottato, elementi grafici contenuti, etc.)

Progettazione della GUI (cont.)



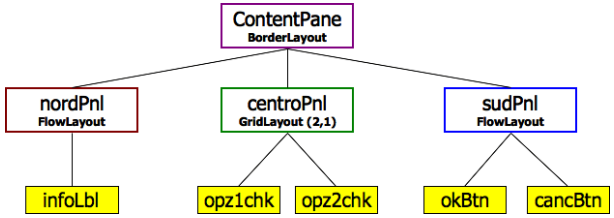
Realizzazione della GUI Progettata

Si procede **Bottom-Up**:

1. costruiamo i componenti atomici

2. popoliamo i contenitori partendo da quelli più profondi, fino ad arrivare al Content Pane radice

Esempio




```
public final class MyFrame extends JFrame{
        private static final String titolo = "Contenitori Multipli";

        //Costruiamo Prima gli elementi atomici (foglie dell'albero):
        //Pannello Nord:
        private static final JPanel nordPnl = new JPanel();
        private static final JLabel infoLbl = new JLabel("Selezionare");
        //Pannello Centrale:
        private static final JPanel centroPnl = new JPanel(new GridLayout(2,1));
        private static final JCheckBox opz1Chk = new JCheckBox("Opz1");
        private static final JCheckBox opz2Chk = new JCheckBox("Opz2");
        //Pannello Sud:
        private static final JPanel sudPnl = new JPanel();
        private static final JButton okBtn=new JButton("OK");
        private static final JButton cancBtn=new JButton("Annulla");
```

Esempio (cont.)

```
//Popoliamo i Container "dal basso verso l'alto" (nel costruttore)
public MyFrame(){
        //Pannello Nord
        nordPnl.add(infoLbl);

        //Pannello Centro
        centroPnl.add(opz1Chk);
        centroPnl.add(opz2Chk);

        //Pannello Sud
        sudPnl.add(okBtn);
        sudPnl.add(cancBtn);

        //Container Principale
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container frmContentPane=this.getContentPane();
        frmContentPane.setLayout(new BorderLayout());
        frmContentPane.add(nordPnl,BorderLayout.NORTH);
        frmContentPane.add(centroPnl,BorderLayout.CENTER);
        frmContentPane.add(sudPnl,BorderLayout.SOUTH);
```

Esempio (cont.)

```
        //Impostiamo le proprieta' di visualizzazione
        this.pack();
        // Imposta la dimensione minima
        // necessaria a visualizzare tutti i componenti

        // Usiamo la classe java.awt.Dimension
        // contiene due proprieta': Width Height
        // che impostiamo alle dimensioni dello schermo
        Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();

        // Posizioniamo la finestra al centro dello schermo
        //(necessario il cast double -> int):
        this.setLocation((int)(dim.getWidth()-this.getWidth())/2,
                                 (int)(dim.getHeight()-this.getHeight())/2);

        this.setVisible(true);
}
```

I Menù

Permettono un'organizzazione gerarchica delle funzionalità offerte dalla GUI

Ciascun `JFrame` può contenere al più una barra Menu, identificata da un oggetto `JMenuBar`, che viene creato con il costruttore senza parametri

`JMenuBar()`

ed assegnato ad un oggetto `JFrame` tramite il metodo (della classe `JFrame`):

`void setJMenuBar(JMenuBar menubar)`

Esempio:

```
JFrame frm = new JFrame("Il mio Frame");
JMenuBar jb = new JMenuBar();
frm.setJMenuBar(jb);
```

I Menù (Cont.)

Un oggetto `JMenuBar` può contenere un numero qualsiasi di oggetti `JMenu`, che corrispondono ai sottomenù

Un oggetto `JMenuBar` è costruito a partire dal costruttore ad un argomento, al quale si fornisce il nome del menu in input

`JMenu(Edit)`

e viene inserito in un oggetto `JMenuBar` tramite il metodo:

`JMenu add(JMenu m)`

Esempio:

```
JMenuBar jb = new JMenuBar();
JMenu jm = new JMenu("Edit");
jb.add(jm);
```

I Menù (Cont.)

Oggetti di classe `JMenu` possono contenere altri oggetti `JMenu`, in modo da realizzare menù annidati (sottomenù)

Un sottomenù viene aggiunto ad un menù tramite il metodo

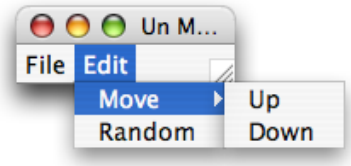
`Component add(Component)` della classe `JMenu`

Inoltre, oggetti di classe `JMenu` possono contenere oggetti di tipo `JMenuItem` che rappresentano le voci del menù

Un oggetto `JMenuItem` può essere costruito con il costruttore ad un argomento: `JMenuItem(String voce)`

e può essere aggiunto ad un menù con il metodo `JMenuItem add(JMenuItem item)`

Esempio




```
// Procediamo "dal basso verso l'alto"
public class MyMenu extends JFrame{
        private static final String titolo = "Un Menu di Esempio";
        //Menu Bar
        private final JMenuBar menuBar = new JMenuBar()
        //Menu Move
        private static final JMenu moveMenu = new JMenu("Move");
        private static final JMenuItem upIt = new JMenuItem("Up");
        private static final JMenuItem dwnIt = new JMenuItem("Down");
        //Menu di Editing
        private static final JMenu editMenu = new JMenu("Edit");
        private static final JMenuItem rndmIt = new JMenuItem("Random");
        //Menu File
        private static final JMenu fileMenu = new JMenu("File");
```

Esempio (cont.)

```
        public MyMenu(){
                super(titolo);
                //Move Menu
                moveMenu.add(upIt);
                moveMenu.add(dwnIt);

                // Edit Menu
                editMenu.add(moveMenu);
                editMenu.add(rndmIt);

                //Menu  File
                menuBar.add(fileMenu);
                menuBar.add(editMenu);
                //Container Principale
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                //Imposta la Barra Menu
                this.setJMenuBar(menuBar);
                //Impostazioni di visualizzazione
                this.pack();
                Dimension dim=Toolkit.getDefaultToolkit().getScreenSize();
                this.setLocation((int)(dim.getWidth()-this.getWidth())/2,
                                                (int)(this.getHeight()-this.getHeight())/2);
                this.setVisible(true);
        }
}
```

Alcuni `JComponent` Comuni

Nel seguito presentiamo gli elementi atomici di uso più comune. La libreria Swing definisce più di 50 diversi componenti, per una lista esustiva, rimandiamo al sito ufficiale (`java.sun.com`)

- `JTextField`: casella di testo
  - es. `JTextField casella = new JTextField(5)`, crea una casella di testo di dimensione pari a 5 caratteri (possono esserne inseriti anche di più)
- `JLabel`: etichetta (v. esempi precedenti)
- `JButton`: pulsante (v. esempi precedenti)
- `JCheckBox`: crea una casella di spunta (v. esempi precedenti)
- `JRadioButton`: permettono la selezione di una tra diverse opzioni
- `JComboBox`: permettono la selezione di uno tra diversi elementi

Rimandiamo al testo *C.S. Horstmann, Concetti di Informatica e Fondamenti di Java 2 (Ed. Apogeo), Capp. 9 e 11* per i dettagli sui componenti: `JLabel`, `JTextField`, `JButton`, `JTextArea`, `JComboBox`, `JRadioButton`, `JCheckBox`, `JTable` e `JOptionPane`

È inoltre disponibile una dispensa didattica pubblicata sul sito del corso

Funzionalità di un'interfaccia

Fin qui, non siamo in grado di dare una semantica alle interfacce grafiche

Ad es., all'attivazione di un pulsante vogliamo far corrispondere l'apertura di una nuova finestra

Per dotare le interfacce di qualche funzionalità abbiamo bisogno delle interfacce Java Listener

Event Delegation

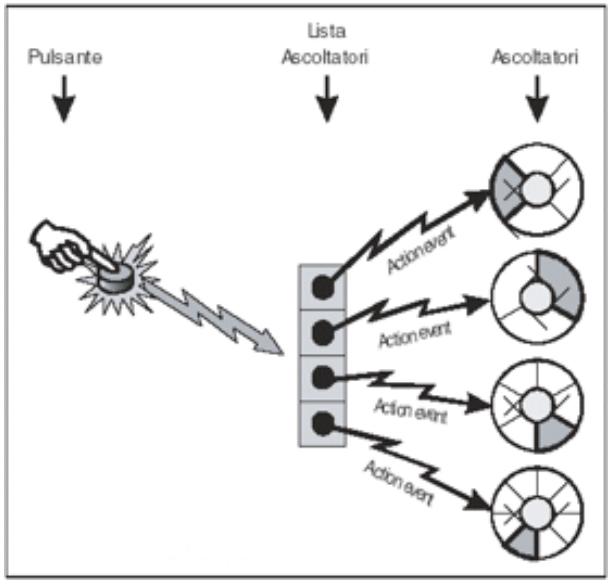
Ciascuno dei componenti visti fin qui è progettato per interagire con l'utente (ad es. un pulsante viene premuto, si può passare sopra una finestra con il mouse, etc.)

Ogni volta che un'interazione ha luogo, viene generato un evento

L'evento generato viene **notificato** ai moduli interessati, **delegando** loro il compito di eseguire le azioni opportune

Per fare ciò, è necessario associare ad ogni componente (e.g., un pulsante) la **lista dei suoi ascoltatori** (Listener)

Event Delegation (cont.)



Ciascun componente può avere più ascoltatori associati ad uno stesso evento

Diversi componenti possono avere uno stesso ascoltatore associato ad uno stesso evento

Gestione degli Eventi

Per dotare un'interfaccia di funzionalità effettive, occorre:

1. Implementare un'opportuna interfaccia Listener con metodi che realizzino le funzionalit

2. Associare un'istanza di tale interfaccia al componente desiderato

Esempio: l'interfaccia MouseListener

Permette di gestire gli eventi scatenati dal mouse

```
public interface MouseListener{
    void mouseClicked(MouseEvent e);
    // Invocato quando si clicca sul componente associato

    void mouseEntered(MouseEvent e);
    // Invocato quando il puntatore del mouse raggiunge il componente

    void mouseExited (MouseEvent e);
    // Invocato quando il puntatore del mouse abbandona il componente

    void mousePressed(MouseEvent e);
    // Invocato quando un pulsante (dx o sx) del mouse viene premuto

    void mouseReleased(MouseEvent e);
    // Invocato quando un pulsante (dx o sx) del mouse viene rilasciato
}
```

Esempio: la classe `MouseEvent`

Un oggetto di classe `MouseEvent` viene istanziato ogni volta che un evento del mouse ha luogo

La classe `MouseEvent` contiene informazioni sull'evento scatenato, accessibili tramite opportuni metodi:

- i metodi `int getX()` e `int getY()` restituiscono le coordinate del mouse relative al momento in cui l'evento è stato scatenato

- Il metodo `int getButton()` permette di conoscere il pulsante premuto (`MouseEvent.NOBUTTON`, `MouseEvent.BUTTON1`, `MouseEvent.BUTTON2`, `MouseEvent.BUTTON3`)

Esempio

```
import java.awt.event.*;// necessaria per la gestione degli eventi

public class MouseSpy implements MouseListener{
        public void mouseClicked(MouseEvent e) {
                System.out.println("Click su ("+e.getX()+","+e.getY()+")");
        }
        public void mousePressed(MouseEvent e) {
                System.out.println("Premuto su ("+e.getX()+","+e.getY()+")");
        }
        public void mouseReleased(MouseEvent e) {
                System.out.println("Rilasciato su ("+e.getX()+","+e.getY()+")")
        }
        public void mouseEntered(MouseEvent e) {} // evento non gestito
        public void mouseExited(MouseEvent e) {} // evento non gestito
}

public class MyFrame extends JFrame{
        public MyFrame(){
                super("MouseTest");
                this.addMouseListener(new MouseSpy());
                setSize(200,200);
                setVisible(true);
        }
}
```

Principali Listener

Interfaccia	Descrizione
`ActionListener`	Definisce un metodo per gestire eventi "azione" (v. sotto)
`ComponentListener`	Definisce 4 metodi per riconoscere quando un componente viene nascosto, spostato, mostrato o ridimensionato
`FocusListener`	Definisce 2 metodi per riconoscere quando un componente ottiene o perde il focus
`KeyListener`	Definisce 3 metodi per riconoscere quando un pulsante viene premuto, rilasciato o cliccato
`MouseMotionListener`	Definisce 2 metodi per riconoscere quando il puntatore del mouse è spostato o trascinato
`MouseListener`	Gestisce gli eventi del mouse (v. sopra)
`TextListener`	Definisce 1 metodo per riconoscere quando il valore di un campo testuale cambia
`WindowListener`	Definisce 7 metodi per riconoscere quando una finestra viene attivata, chiusa, disattivata, ripristinata, ridotta a icona, ecc.

Gli Eventi Azione

Ogni componente Swing genera eventi di tipo `ActionEvent`, gestibili dall'interfaccia `ActionListener`

```
public interface ActionListener{
        public void actionPerformed(ActionEvent ae);
}
```

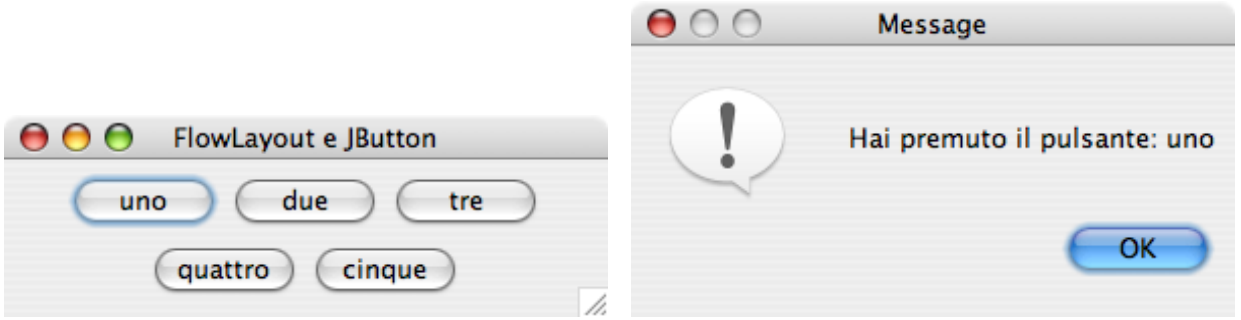
Ad esempio, le seguenti azioni generano un `ActionEvent`:

- pressione del tasto `Invio` all'interno di un campo testuale
- selezione di un `CheckBox`, `RadioButton` o una voce in un `ComboBox`
- selezione di una voce da un menù o click su un pulsante

Esempio

Quando viene premuto un pulsante, vogliamo che si apra una finestra con un messaggio (e.g., che ci dica quale pulsante è stato premuto)

Es:



Esempio (cont.)

Riprendiamo la classe `MyFrame` (v. sopra)

```
public final class MyFrame extends JFrame{
        private static final String titolo = "FlowLayout e JButton";
        private static final int larghezza=300, altezza=100;
        private static final JButton uno = new JButton("uno");
        // due, tre, quattro
        private static final JButton cinque = new JButton("cinque");

        public MyFrame(){
                super(titolo);
                this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                this.setSize(larghezza,altezza);
                Container frmContentPane=this.getContentPane();
                frmContentPane.setLayout(new FlowLayout());
                frmContentPane.add(uno);
                // due, tre, quattro
                frmContentPane.add(cinque);
                this.setVisible(true);
        }
}
```

Esempio (cont.)

Per prima cosa, implementiamo un `ActionListener`

Il metodo `actionPerformed()` stampa l'etichetta del pulsante che ha generato l'evento (gi sappiamo che questo Listener sarà associato ad almeno un pulsante)

```
import java.awt.event.*; // Necessario alla gestione degli eventi
import javax.swing.*;

public class MyFrameListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
         // getSource() restituisce un riferimento all'oggetto che ha
         // generato l'evento
         JButton b = (JButton) ev.getSource();
         // getText() restituisce il testo inserito come etichetta del pulsante
         // (gi sappiamo che assoceremo il Listener ad un pulsante)
         JOptionPane.showMessageDialog(null,"Hai premuto il pulsante: "+b.getText());
    }
}
```

Esempio (cont.)

Infine, è sufficiente associare un oggetto di classe `MyFrameListener` a ciascun pulsante definito nell'interfaccia (grafica) originaria

La classe `MyFrame`, pertanto, diventa:

```
public final class MyFrame extends JFrame{
        /* private static final String titolo, ... larghezza, altezza ...
        private static final JButton uno = new JButton("uno"); // due, ..., cinque
// Creo un'istanza di MyFrameListener:
private static final MyFrameListener listener = new MyFrameListener();

        public MyFrame(){
                super(titolo);
                /*... */
                // Associa listener al pulsante uno
                uno.addActionListener(listener);
                // due, ...
                // Associa listener al pulsante cinque
                cinque.addActionListener(listener);

                frmContentPane.add(uno);
                // due, ...
```

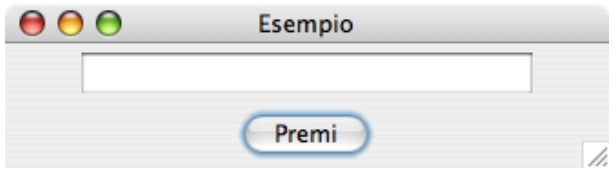
```
                frmContentPane.add(cinque);
                this.setVisible(true);
        }
}
```

Accedere ad altri elementi del JFrame

Consideriamo la seguente finestra

```
public class EsempioTextArea extends JFrame{
        private static final JPanel centro = new JPanel();
        private static final JPanel sud = new JPanel();
        private static final JTextField areaTesto = new JTextField(20);
        private static final JButton button = new JButton("Premi");
        private static final String titolo = "Esempio";

        public EsempioTextArea() {
                super(titolo);
                centro.add(areaTesto);
                sud.add(button);
                getContentPane().add(centro,BorderLayout.CENTER);
                getContentPane().add(sud,BorderLayout.SOUTH);
                // Associa un opportuno Listener al pulsante "button"
                button.addActionListener(new EsempioTextAreaListener());
                setVisible(true);
        }
}
```



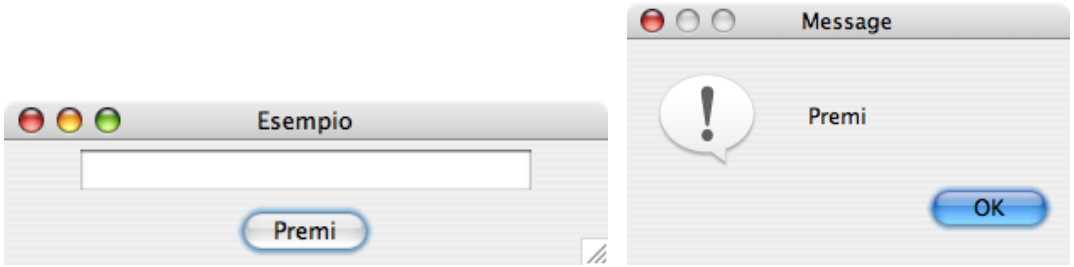
Accedere ad altri elementi del JFrame (cont.)

Vogliamo che, dopo aver riempito la casella di testo, la pressione del pulsante produca una finestra di messaggio che mostri il contenuto della casella

`ActionEvent` permette di accedere **solo all'oggetto che ha generato l'evento**

Se usassimo il metodo `getText()` all'interno di un listener associato al pulsante otterremmo l'etichetta del pulsante, **non il testo della casella**:

```
public class EsempioTextAreaListener implements ActionListener {
        public void actionPerformed(ActionEvent a){
                JButton pulsanteOrigine = (JButton) a.getSource();
                JOptionPane.showMessageDialog(null,pulsanteOrigine.getText());
        }
}
```



Soluzione 1: Listener come inner class (evitare!)

NOTA: L'implementazione con inner class riportata solo per completezza, poich uno stile molto diffuso, tuttavia, l'uso di un'altra classe (interna!) non giustificato da alcuna considerazione ragionevole.

Possiamo definire `EsempioTextAreaListener` come **inner class** (classe interna) di `EsempioTextArea`

La inner class `EsempioTextAreaListener` **ha accesso a tutti i campi (anche privati)** della classe in cui è definita, ovvero `EsempioTextArea` ed in particolare al campo `JTextField areaTesto`, che rappresenta la casella di testo d'interesse:

```
public class EsempioTextArea extends JFrame{
// Definizione di: centro, sud, areaTesto, button, titolo (come sopra)
// In particolare:
private static final JTextField areaTesto = new JTextField(20);

public EsempioTextArea() {
        //Creazione e impostazione finestra (come sopra)
        // ...
        button.addActionListener(new EsempioTextAreaListener());
        setVisible(true);
        }
        // Inner class:
        private class EsempioTextAreaListener implements ActionListener {
                public void actionPerformed(ActionEvent a){
                        // Accede direttamente al campo "areaTesto"
                        JOptionPane.showMessageDialog(null,areaTesto.getText());
                }
        }
}
```

Soluzione 2: la finestra implementa il Listener

Otteniamo lo stesso risultato con la seguente dichiarazione (da preferire!):

```
public class EsempioTextArea extends JFrame implements ActionListener{
  // Definizione di: centro, sud, areaTesto, button, titolo (come sopra)
  // In particolare:
  private static final JTextField areaTesto = new JTextField(20);

  public EsempioTextArea() {
    //Creazione e impostazione finestra (come sopra)
    // ...
    button.addActionListener(this);// il listener  la classe stessa
    setVisible(true):

    // (Stesso metodo della inner class mostrata sopra:
    public void actionPerformed(ActionEvent a){
      // Accede direttamente al campo "areaTesto"
      JOptionPane.showMessageDialog(null,areaTesto.getText());
    }
}
```

Questo approccio utile per gestire finestre con poche e semplici funzionalit (per la gestione di pi componenti da parte di uno stesso listener v. sotto).

Soluzione 3: classe esterna

Possiamo anche implementare un Listener esterno che abbia un riferimento alla finestra contenente il componente che ha scatenato l'evento

In tal modo, il Listener avrebbe accesso ai soli campi pubblici della finestra

Nel nostro caso, quindi, è necessario dotare la classe `EsempioTextArea` di un metodo `getButton()` che restituisca un riferimento alla casella `AreaTesto`

Questo approccio utile quando uno stesso listener interagisce con i componenti di finestre diverse

Soluzione 3: classe esterna (cont.)

```
//Classe esterna:
public class EsempioTextAreaListener implements ActionListener {
        EsempioTextArea frame;

        EsempioTextAreaListener(EsempioTextArea frame){
                this.frame = frame;
        }

        public void actionPerformed(ActionEvent a){
                // Accede direttamente al campo "areaTesto"
                JOptionPane.showMessageDialog(null,frame.getAreaTesto().getText());
        }
}

public class EsempioTextArea extends JFrame{
        // Definizione di: centro, sud, areaTesto, button, titolo (come sopra)
        // In particolare:
        private static final JTextField areaTesto = new JTextField(20);
        public EsempioTextArea() {
                //Creazione e impostazione finestra (come sopra)
                // ...
                button.addActionListener(new EsempioTextAreaListener(this));// Passiamo  this al costruttore!
                setVisible(true);
        }

        getAreaTesto(){
                return areaTesto;
        }
}
```

Soluzione 4: package per l'interfaccia grafica

Se non specifichiamo il livello di accesso di un campo dati o di un metodo, esso sar accessibile (solo) a tutte le classi all'interno dello stesso package. Ovviamente, i campi privati rimangono accessibili solo dall'interno della classe cui appartengono.

Sfruttiamo questa caratteristica per garantire l'accesso ai campi dato di un JFrame da parte dei suoi Listener (potremmo dire che trattiamo i Listener come metodi della classe JFrame)

```
package app.gui.myframe; // package di appartenenza della classe
public class myFrame extends JFrame{
  final JTextField areaTesto = new JTextField(20); // livello d'accesso non specificato!
  public EsempioTextArea() { // Creazione e impostazione finestra (come sopra)  ...
     button.addActionListener(new EsempioTextAreaListener(this));
     setVisible(true);
  }
}
```

Listener come classe esterna (dello stesso package):

```
//stesso package: pu accedere ai campi non privati delle altre classi in app.gui.myframe
package app.gui.myframe;
class EsempioTextAreaListener implements ActionListener {// Nota: la classe non  pubblica!
        EsempioTextArea frame;
        EsempioTextAreaListener(EsempioTextArea frame){
                this.frame = frame;
        }
        public void actionPerformed(ActionEvent a){
                // Accede a tutti i campi non private di "areaTesto"
                JOptionPane.showMessageDialog(null,frame.getAreaTesto().getText());
        }
}
```

Soluzione 4: considerazioni

Adottiamo il seguente approccio:

1. creiamo un package app.gui che conterr tutte le classi relative all'interfaccia grafica

2. aggiungiamo un sotto-package app.gui.myframe contenente una classe che implementa la finestra ed i relativi ascoltatori

3. se vi sono ascoltatori che interagiscono con pi finestre, inseriamo tutto nello stesso sotto-package

Questo approccio **riduce l'information hiding** ed **aumenta l'accoppiamento**

Possiamo ancora nascondere campi interni al Frame dichiarandoli private

Questa soluzione da preferirsi quando si ha a che fare con listener molto complessi e/o che interagiscono con pi di una finestra

Limitare la proliferazione dei Listener

Come si è visto, associamo un Listener ad ogni componente al quale vogliamo assegnare una funzionalità

Al crescere del numero di componenti, crescerà il numero di Listener

Come limitarne la proliferazione?

Possiamo creare un solo Listener che offra pi funzionalit, ovvero raggruppare insiemi di funzionalit tra loro omogenee in uno stesso listener

Limitare la proliferazione dei Listener (cont.)

(Esempio con classi esterne, immediatamente applicabile agli altri casi)

Il metodo `setActionCommand(String c)` permette di associare un comando (stringa) all'`ActionEvent` generato da un componente

```
public class MyFrame extends JFrame {
        //...
        JMenuItem UpOpt = new JMenuItem("Up");
        JMenuItem DownOpt = new JMenuItem("Down");
        JMenuItem RandomOpt = new JMenuItem("Random");
        Listener ascolt = new Listener();
        public MyFrame() {
                //...
                UpOpt.addActionListener(ascolt);
                // UpOpt scrive nell'ActionEvent generato la stringa Listener.UPOPT
                UpOpt.setActionCommand(Listener.UPOPT);
                DownOpt.addActionListener(ascolt);
                DownOpt.setActionCommand(Listener.DOWNOPT);
                RandomOpt.addActionListener(ascolt);
                RandomOpt.setActionCommand(Listener.RANDOMOPT)
                //...
        }
}
```

Limitare la proliferazione dei Listener (cont.)

Il metodo `String getActionCommand()` restituisce la stringa "comando" associata all'evento ricevuto (v. sopra)

```
public class Listener implements ActionListener {
        // Costanti usate quando la stringa "comando" viene assegnata
        public final String UPOPT = "up";
        public final String DOWNOPT = "down";
        public final String RANDOMOPT = "random";

        public void actionPerformed(ActionEvent e) {
                String com = e.getActionCommand();
                // Qui, viene gestito il comando:
                // (se non e' una stringa valida, nessun comando viene eseguito)
                if (com == UPOPT)
                     upOpt();
                else if (src == DOWNOPT)
                        downOpt();
                else if (src == RANDOMOPT)
                        randomOpt();
        }
        // Metodi specifici associati alla stringa "comando" ricevuta
        private void upOpt(){ ... }
        private void randomOpt(){ ... }
        private void downOpt(){ ... }
```