



Agent and Object Technology Lab
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma



Ingegneria del software A

Interfacce grafiche (in Java)

Michele Tomaiuolo

- ♦ La programmazione orientata agli oggetti si è sviluppata a fronte del successo delle **Graphical User Interface (GUI)**
- ♦ Capire il funzionamento delle GUI...
 - Consente di esplorare meglio il modello ad oggetti
 - È importante perchè oggi tutte le applicazioni hanno una GUI

- ♦ Progettare una GUI è un'attività molto complessa
- ♦ Il progettista deve:
 - Conoscere la tipologia degli utenti e i loro bisogni
 - Prevenire gli errori degli utenti, quando possibile
 - Snellire il più possibile l'accesso ai dati ed ai comandi
- ♦ Una GUI deve essere:
 - Auto-consistente, cioè avere un modo uniforme per presentare i dati e per accettare i comandi
 - Tenere presente le convenzioni del sistema in cui l'applicazione verrà eseguita

- ◆ **Abstract Windowing Toolkit (AWT)**, minimo comune denominatore tra i widget delle diverse piattaforme
 - Idea iniziale: fornire una library per realizzare GUI indipendentemente dalla piattaforma di esecuzione
 - Non sufficientemente potente per realizzare GUI complesse
- ◆ **Swing** introdotto nelle specifiche Java 2 (JDK 1.2)
 - Nuova library completamente riprogettata che si appoggia ad AWT solo per i servizi di base
- ◆ Attualmente, Java contiene sia AWT che Swing
 - Il progettista può scegliere quale utilizzare
 - Inoltre, **SWT** (Eclipse) è una alternativa abbastanza diffusa

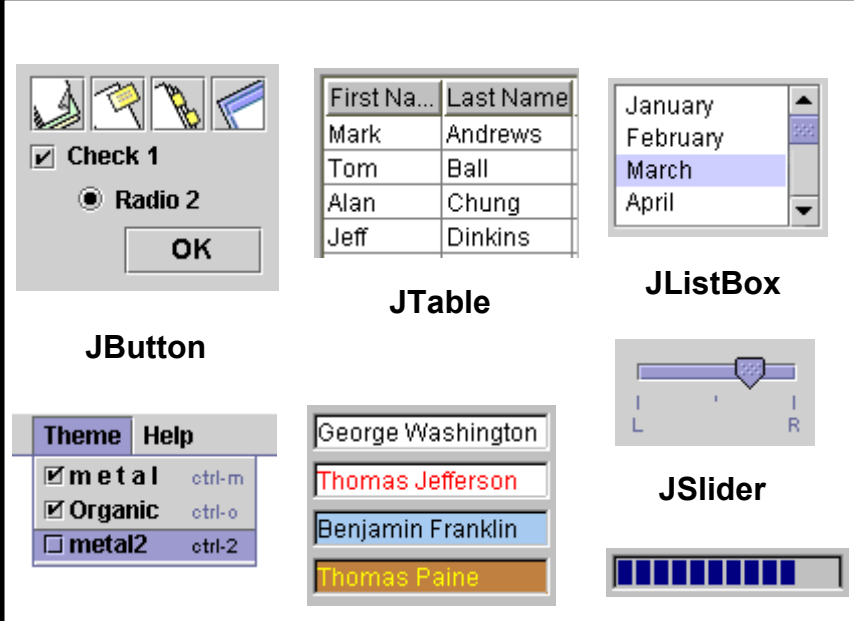
- ♦ Swing è un **framework** per creare delle GUI
 - Gli sviluppatori hanno fornito un ambiente generale, in cui aggiungere le parti specifiche di ogni particolare applicazione
- ♦ Un framework orientato agli oggetti facilita lo sviluppo tramite due diversi tipi di **riuso**
 1. Il riuso **black-box** di componenti già pronti
 - Classi Swing pronte da usare (es. `JButton`)
 2. Il riuso **white-box** di classi semi-lavorate, da completare
 - Interfacce o classi astratte Swing da specializzare (`Action`)

- ♦ Per usare Swing bisogna sapere due cose
 - Quali sono le principali caratteristiche di un framework orientato agli oggetti per GUI
 - Cosa offre Swing relativamente a ciascuna di esse
- ♦ Swing è paradigmatico
 - Sviluppato appositamente per creare interfacce grafiche complesse in maniera *indipendente dalla piattaforma*
- ♦ Tre caratteristiche principali di una GUI
 1. **Struttura** (*suddividere lo spazio con criterio*)
 2. **Reattività** (*reagire a eventi con azioni*)
 3. **Visualizzazione** (*disegnare con aspetto piacevole*)

- ◆ Swing fornisce i principali componenti di una GUI (bottoni, menu ecc.)
- ◆ In Swing, tutti i componenti estendono la classe `JComponent`
- ◆ Fornisce molti componenti che possono contenerne altri
 - Un contenitore (**container**) è uno speciale tipo di componente
 - Racchiude ed organizza altri componenti
 - Bottoni, pannelli, combo-box...

- ◆ Per dare una struttura ad una interfaccia grafica, si devono impostare delle *relazioni di contenimento* tra i vari componenti Swing
- ◆ La maggior parte dei contenitori possono essere contenuti anche in un altro contenitore
 - Si possono creare strutture *ricorsive* e *gerarchiche*
 - Una delle classi contenitore più usata è `JPanel`
- ◆ `JComponent` ha un metodo `add()`

Principali componenti



JButton: A button with four icons and labels 'Check 1' (checked) and 'Radio 2' (selected), with an 'OK' button below.

JTable: A table with two columns: 'First Na...' and 'Last Name'. Rows include Mark Andrews, Tom Ball, Alan Chung, and Jeff Dinkins.

JList: A list box showing months: January, February, March (selected), and April.

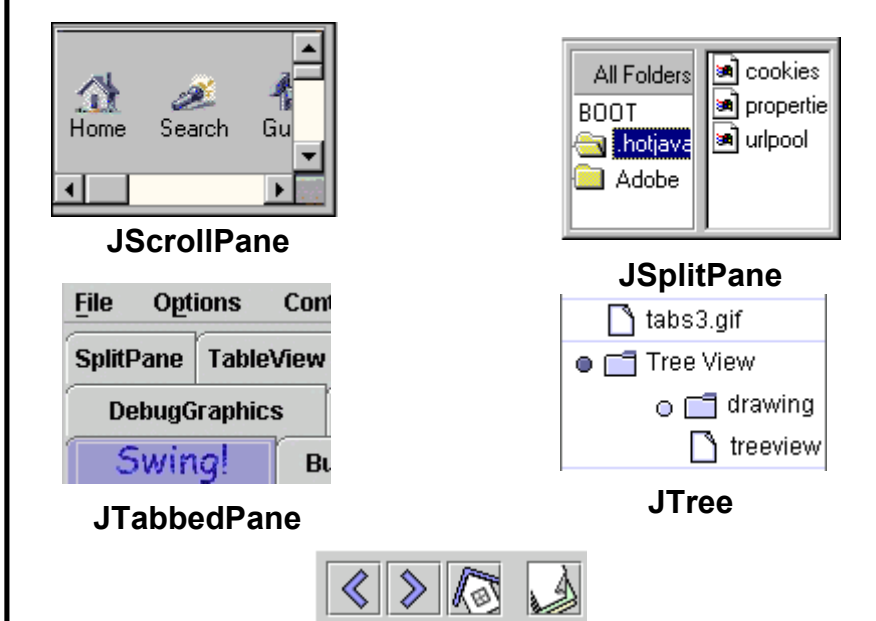
JMenu: A menu with 'Theme' and 'Help' sections. 'Theme' has 'metal' (checked, ctrl-m), 'Organic' (checked, ctrl-o), and 'metal2' (unchecked, ctrl-2).

JTextField: A text field containing 'George Washington', with a list of names below: Thomas Jefferson, Benjamin Franklin, and Thomas Paine.

JSlider: A horizontal slider with 'L' and 'R' labels.

JProgressBar: A progress bar with a blue fill and a vertical line.

Principali contenitori



JScrollPane: A scroll pane showing a home icon, a search icon, and a 'Gu' label.

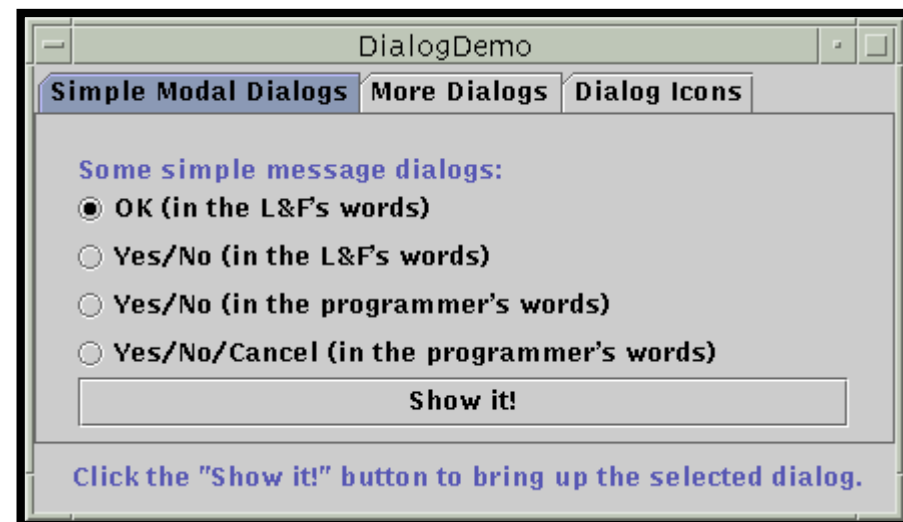
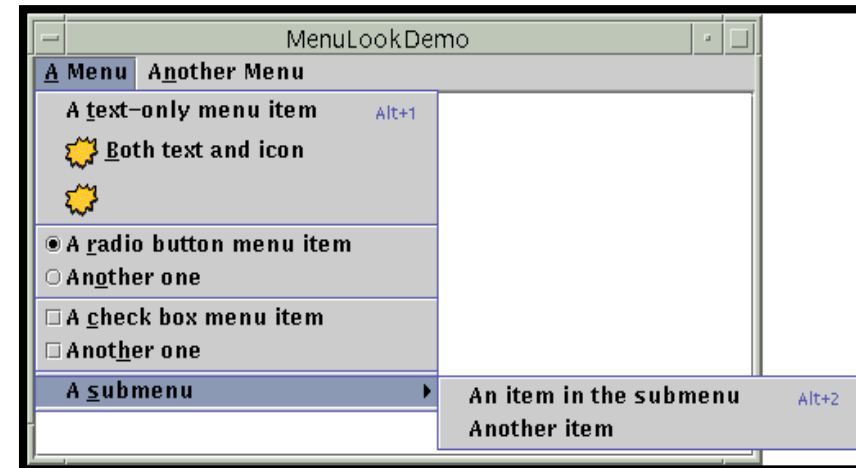
JTabbedPane: A tabbed pane with tabs for 'File', 'Options', 'Com', 'SplitPane', 'TableView', 'DebugGraphics', and 'Swing!' (selected).

JSplitPane: A split pane showing a file explorer view with 'All Folders' and a list of folders: cookies, proprietie, urlpool, and a file 'tabs3.gif'.

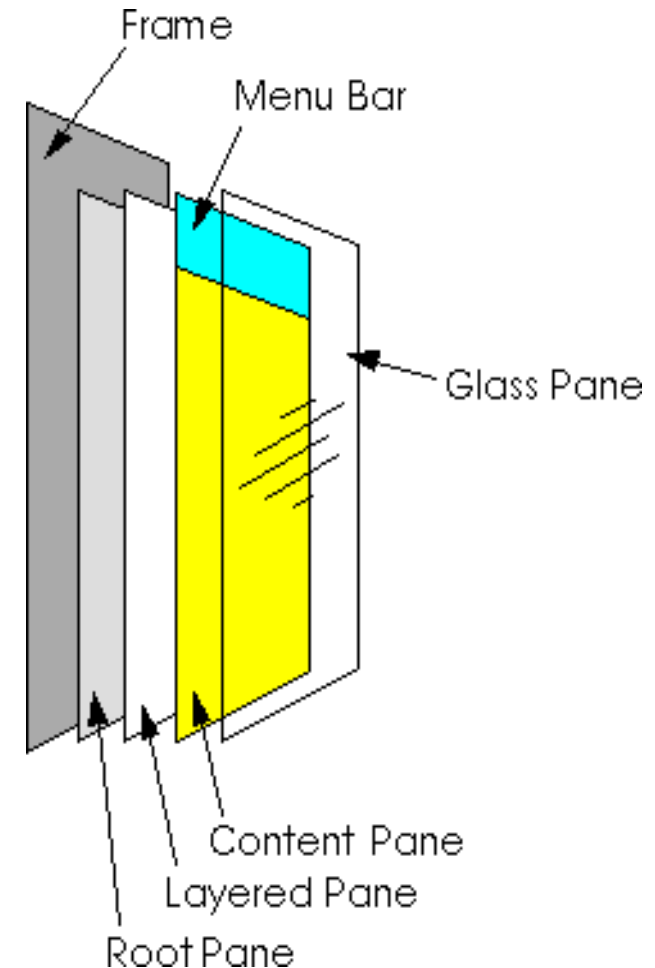
JTree: A tree view showing a 'Tree View' folder containing 'drawing' and 'treeview'.

JToolBar: A toolbar with four icons: a left arrow, a right arrow, a home icon, and a document icon.

- ◆ Solo pochi tipi di contenitori possono agire come finestre principali
- ◆ Questi non possono essere contenuti in altri contenitori (*top-level*)
- ◆ Esempi principali
 - **JFrame** (per le finestre della applicazione)
 - **JDialog** (per le finestre di dialogo)
 - **JApplet** (per l'area delle applet nei browser)



- ◆ I contenitori Swing *top-level* sono composti da diversi livelli
- ◆ I componenti figlio vanno aggiunti al *content pane*
- ◆ `frame.getContentPane().add(label);`
- ◆ `frame.setContentPane(panel);`
- ◆ Gli altri livelli sono usati solo per scopi particolari



- ♦ Un programma con una GUI deve:
 1. Costruire tutti i vari oggetti Java che rappresentano i componenti grafici
 2. Comporli in una struttura
 - Relazioni di contenimento
 - Gestione della disposizione (*layout management*)

- ♦ Dopo aver completato questa fase, cos'altro deve fare?

Niente!!! (Fino a nuovi ordini)

- ♦ Una GUI è composta da almeno tre tipi di oggetti
 - **Componenti**, come bottoni o menù, che vengono visualizzati
 - **Eventi**, che reificano le azioni dell'utente
 - **Listener**, che rispondono agli eventi sui componenti
- ♦ Una GUI è infatti un sistema reattivo
 - Significa che intraprende una *azione* solo quando riceve uno stimolo esterno (*evento*)
- ♦ Quello che bisogna fare, una volta costruita la struttura grafica, è specificare quale oggetto Java è il gestore di ciascun diverso stimolo
- ♦ In gergo Java, bisogna **installare i listener**

- ◆ Evento: un oggetto generato (*fire*) da un componente, che rappresenta un'attività dell'utente sulla GUI
 - Il mouse è stato mosso
 - Il bottone del mouse è stato premuto
 - È stata scelta una voce di menù

- ◆ Varie classi che rappresentano i più comuni tipi di eventi, organizzate in una gerarchia di ereditarietà

- ◆ Ogni componente genera eventi specifici
 - Per esempio, un `JButton` può generare `ActionEvent`

- ◆ Per gestire un certo evento, bisogna implementare una specifica interfaccia *listener*
- ◆ Un listener è un oggetto che aspetta che un componente generi un particolare tipo di evento
- ◆ Swing mette a disposizione una serie di interfacce corrispondenti a listener per i più comuni tipi di eventi
- ◆ Per reagire ad un evento si implementa l'interfaccia listener apposita e la si registra sul componente che potrebbe generare l'evento

- ♦ L'interfaccia `ActionListener` è utilizzata per implementare listener di eventi di molti componenti
 - Un bottone è stato premuto
 - Una voce di menù è stata selezionata
 - Un bottone toggle ha cambiato stato
- ♦ Quando un evento di questo tipo accade, su tutti i listener registrati viene invocato il metodo `actionPerformed(ActionEvent)`

- ♦ L'interfaccia `MouseListener` è quella da utilizzare per ricevere gli eventi del mouse (`MouseEvent`)

- `mousePressed` – il bottone del mouse è stato premuto
- `mouseReleased` – il bottone del mouse è stato rilasciato
- `mouseClicked` – il bottone del mouse è stato prima premuto e poi rilasciato senza muovere il mouse
- `mouseEntered` – il puntatore del mouse è entrato nel componente che ha attivato il listener
- `mouseExited` – il puntatore del mouse è uscito dal componente che ha attivato il listener

- ◆ Dopo aver scritto l'oggetto *handler*, bisogna agganciarlo allo specifico componente che è la fonte dell'evento
- ◆ Un componente può avere più di un *listener*
- ◆ Si possono usare i seguenti metodi
 - `addXYZListener(...)`
 - `removeXYZListener(...)`

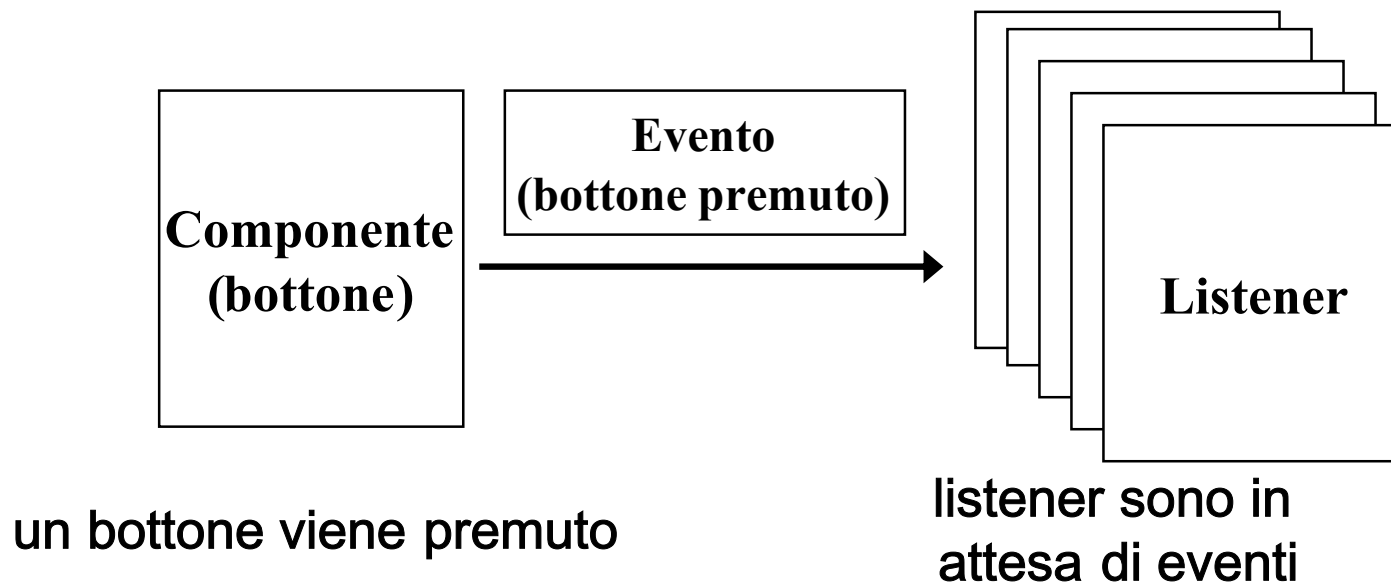
```
class X implements
    ActionListener {
    public void actionPerformed(
       (ActionEvent ae) {
        // Handle button click ...
    }
}

//...

Jbutton b = new Jbutton("OK");
X handler = new X();

b.addActionListener(handler);
```

- ◆ Quando si verifica un *evento*, il *componente* genera un oggetto che viene passato a tutti i *listener* registrati
- ◆ Tutti gli *event handler* del sistema sono eseguiti in un singolo thread (*Event Dispatcher Thread*)



- ♦ Per scrivere i gestori di eventi (*event handler*), possono dimostrarsi molto utili le *Anonymous Inner Classes*
 - Permettono di scrivere le linee di codice strettamente necessario
 - Preservano l'incapsulamento, evitando di scrivere classi separate, aperte all'accesso esterno

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

public class SwingHello {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hey!");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        JLabel label = new JLabel("Hello, world!");
        frame.getContentPane().add(label);
        frame.pack();
        frame.setVisible(true);
    }
}
```

```
javac SwingHello.java  
java SwingHello
```

```
public class ButtonDemo extends JPanel
    implements ActionListener {
    private JButton b1, b2, b3;
    //...
    public ButtonDemo() {
        b1 = new JButton("Disable middle button");
        b1.setActionCommand("disable");

        b2 = new JButton("Middle button");

        b3 = new JButton("Enable middle button");
        b3.setActionCommand("enable");
        b3.setEnabled(false);

        //Listen for actions on buttons 1 and 3.
        b1.addActionListener(this); b3.addActionListener(this);
        add(b1); add(b2); add(b3);
    }
}
```

```
public class ButtonDemo extends JPanel
    implements ActionListener {

    //...
    public void actionPerformed(ActionEvent e) {
        if ("disable".equals(e.getActionCommand())) {
            b2.setEnabled(false);
            b1.setEnabled(false);
            b3.setEnabled(true);
        } else {
            b2.setEnabled(true);
            b1.setEnabled(true);
            b3.setEnabled(false);
        }
    }
}
```

- ♦ Nello sviluppo di sistemi software, tra le altre qualità, ***maintainability***, ***modularity*** e ***extensibility*** sono spesso caratteristiche richieste
 - Le *-ilities* dell'ingegneria del software...
- ♦ Nella progettazione di una GUI, la prima regola è:

*“In un sistema complesso,
l'interfaccia utente non dovrebbe essere
accoppiata alla logica di elaborazione”*

- ◆ Conseguentemente, la GUI deve essere tenuta separata da:
 - Modello dei dati elaborati (per esempio, la struttura di un database)
 - Politiche di gestione dei dati stessi (per esempio, le *business rules* di una applicazione)
- ◆ Swing supporta il primo sforzo definendo classi separate per il *modello* e la *vista*
- ◆ I modelli di Swing sono legati ai singoli componenti (*micro-modelli*)

<http://java.sun.com/products/jfc/tsc/articles/architecture/>

- ♦ In Swing, la maggior parte dei componenti sono associati ad un micro-modello separato
 - Per esempio, un `JButton` ha il suo proprio `ButtonModel`, che permette al programma di “pilotare” il bottone
 - Più i componenti sono complessi, più si dimostrano utili (`JList`, `JTable`, `JTree`)

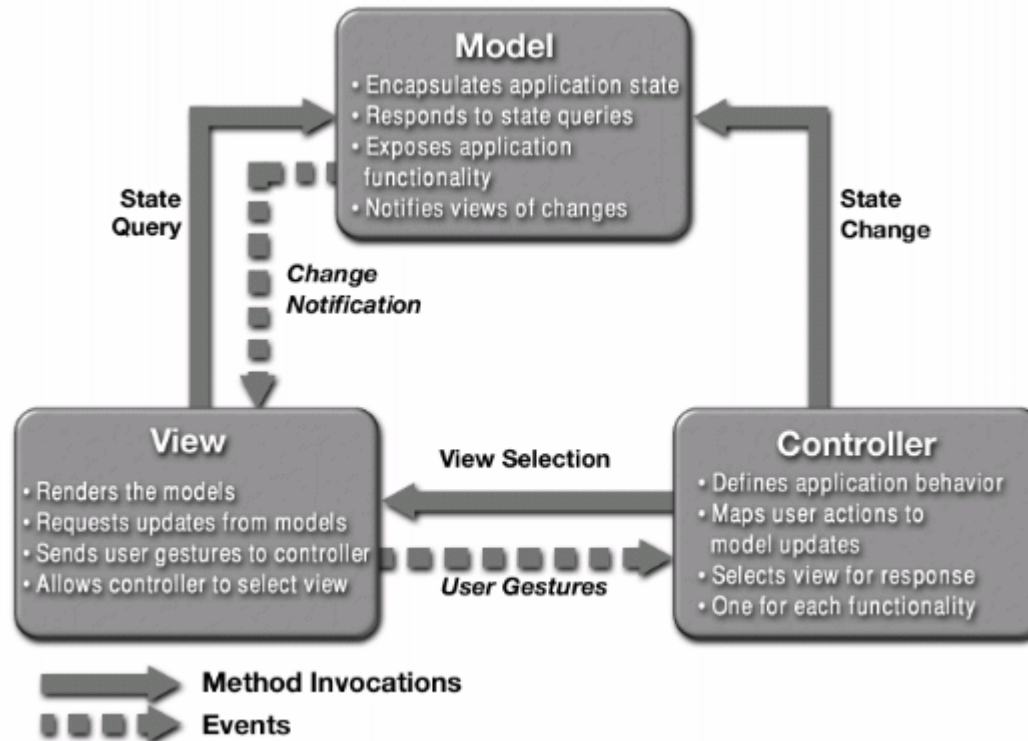
- ♦ I componenti grafici Swing sono sincronizzati automaticamente con i rispettivi micro-modelli
 - Quindi, se un elemento viene aggiunto ad un `JTreeModel` chiamando il suo metodo `insertNodeInto()`, allora un nuovo ramo apparirà nel suo `JTree`

- ♦ Con le classi di micro-modello di Swing, le modifiche apportate ai dati sono rese visibili nella GUI
- ♦ Tuttavia non gestiscono il modello completo dei dati di una applicazione
- ♦ Non dovrebbero *duplicare* i dati, ma essere implementati come *adapter* (o filtri) rispetto al modello completo
- ♦ L'altro requisito è di separare la gestione delle politiche e le elaborazioni dei dati dalla GUI

⇒ ***Le classi listener devono delegare il “vero” lavoro alle classi del dominio dell'applicazione***

- ♦ L'architettura ritenuta migliore per progettare una GUI è detta **Model-View-Controller (MVC)**
- ♦ La parte dell'applicazione dedicata alla GUI viene spezzata in tre categorie di classi
 - Classi *model*: implementano il modello di quello che si vuole rappresentare, senza dire nulla su come verrà rappresentato
 - Classi *view*: utilizzano le classi *model* per dare una veste grafica, una vista, al modello
 - Classi *controller*: descrivono come il modello cambia in reazione agli eventi che l'utente genera sulla GUI; ad ogni cambiamento significativo del modello, anche la vista viene informata

Architettura MVC classica



<http://java.sun.com/developer/technicalArticles/javase/mvc/>

1. La *vista* si registra come listener sul *modello*

- Ogni cambiamento nei dati del modello sottostante provoca immediatamente una notifica in broadcast del cambiamento, che la vista riceve (modello push)
- Si noti che il modello non ha nozione della vista o del controller: semplicemente invia in broadcast le notifiche di cambiamento a tutti i listener interessati

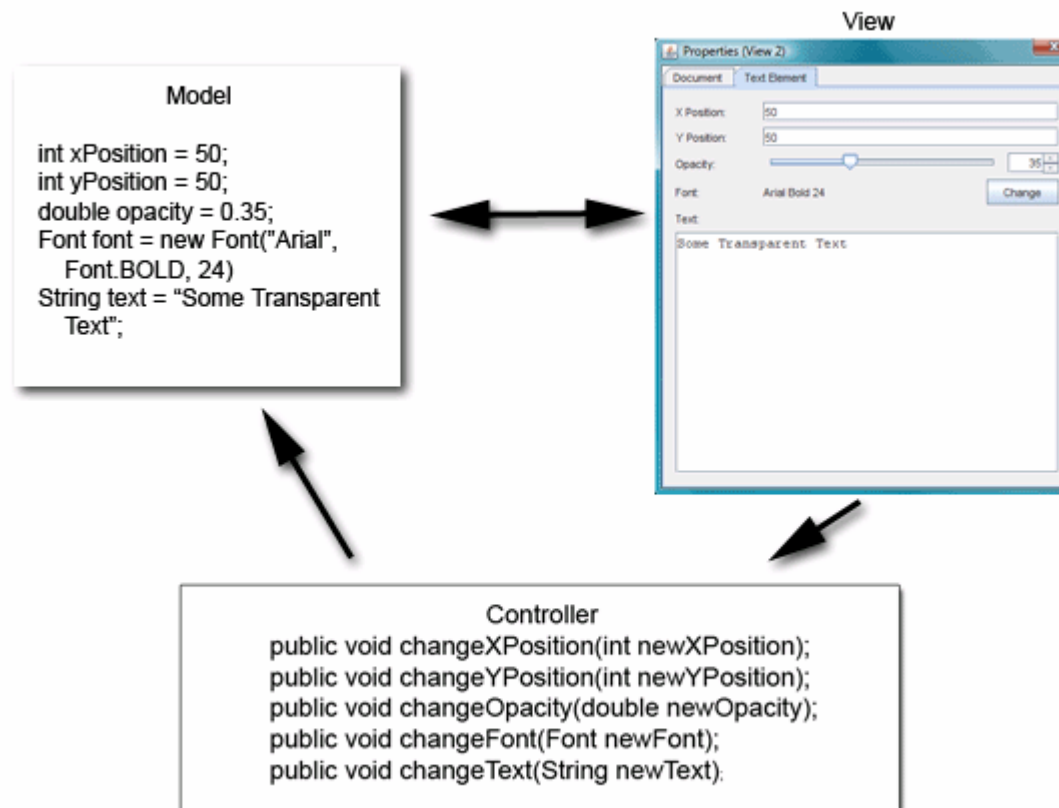
2. Il *controller* è collegato alla *vista*

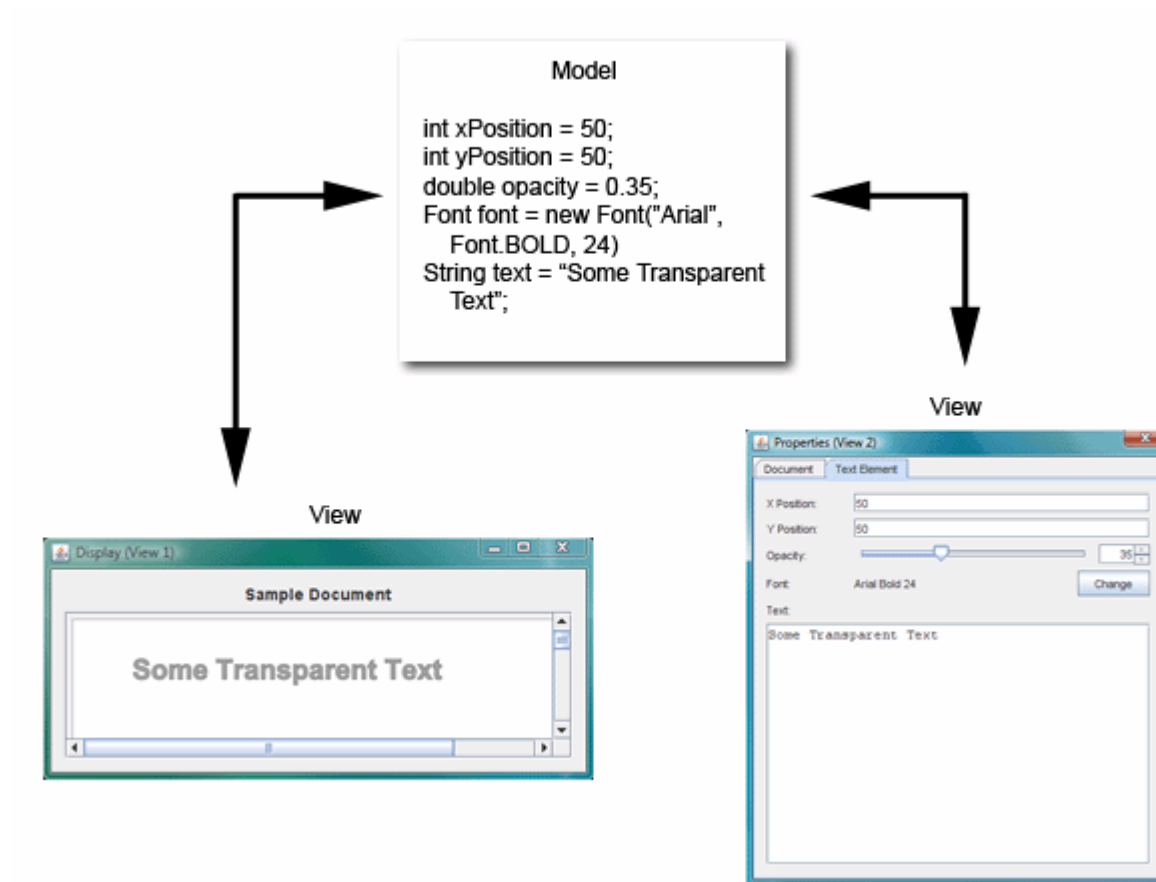
- Questo tipicamente significa che ogni azione dell'utente eseguita sulla vista invocherà un metodo nella classe controller registrato come listener

3. Al *controller* viene dato un riferimento al sottostante *modello*

1. La *vista* riconosce qualche attività dell'utente sulla GUI
 - Es. bottone premuto o movimento scroll bar
 - La vista ha un metodo listener registrato per essere invocato quando ha luogo una tale azione, e poi chiama l'appropriato metodo del controller
 - A volte il controller è registrato direttamente come listener
2. Il *controller* accede al modello, possibilmente aggiornandolo in un modo appropriato rispetto all'azione dell'utente
3. Se il *modello* è stato modificato, notifica del cambiamento i listener interessati, come la vista
 - In alcune architetture, il controller potrebbe essere responsabile anche dell'aggiornamento della vista

Esempio con vista singola

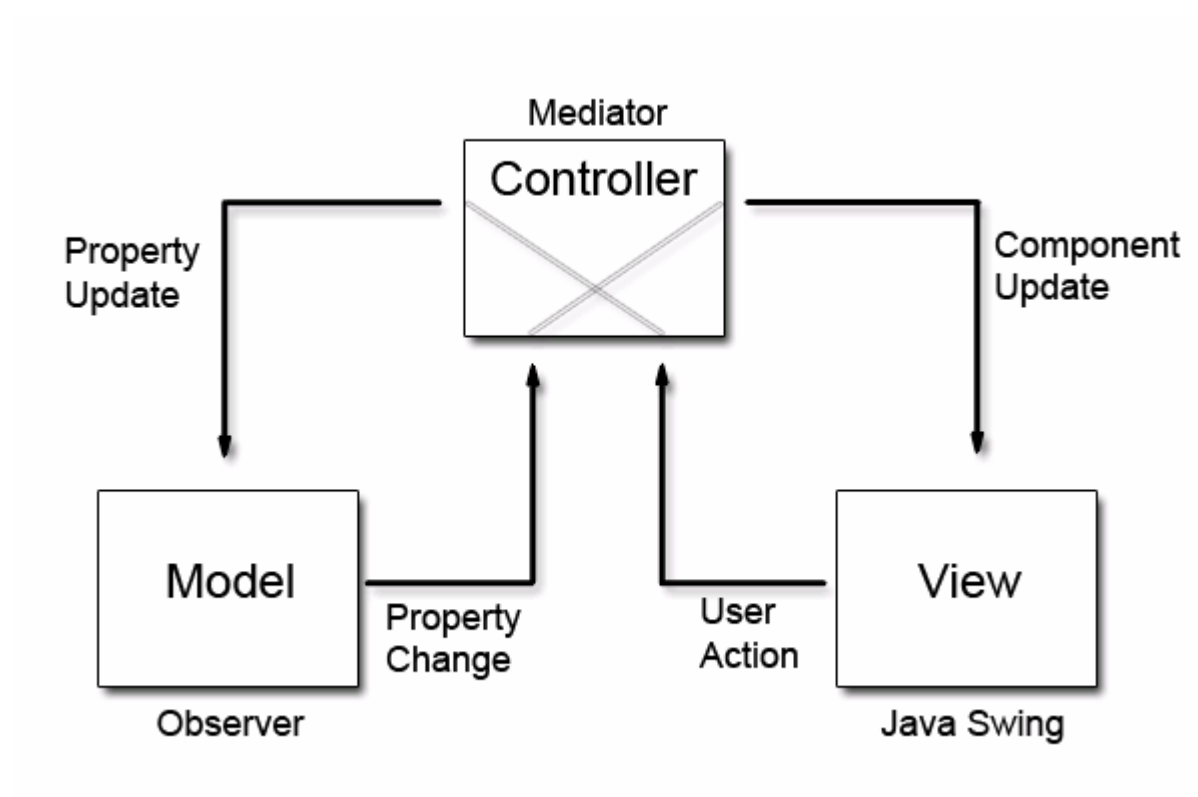




```
public class ExampleModel {
    // ...
    private PropertyChangeSupport propertyChangeSupport;
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertyChangeSupport.addPropertyChangeListener(listener);
    }
    public void removePropertyChangeListener(PropertyChangeListener listener) {
        propertyChangeSupport.removePropertyChangeListener(listener);
    }
    protected void firePropertyChange(String propertyName,
                                       Object oldVal, Object newVal) {
        propertyChangeSupport.firePropertyChange(propertyName, oldVal, newVal);
    }

    private String text;
    public String getText() {
        return text;
    }
    public void setText(String text) {
        String oldText = this.text;
        this.text = text;
        firePropertyChange("Text", oldText, text);
    }
}
```

Controller come mediatore



1. La *vista* riconosce qualche attività dell'utente sulla GUI
 - Es. bottone premuto o movimento scroll bar
 - La vista ha un metodo listener registrato per essere invocato quando ha luogo una tale azione, e poi chiama l'appropriato metodo del controller
 - A volte il controller è registrato direttamente come listener
2. Il *controller* accede al modello, possibilmente aggiornandolo in un modo appropriato rispetto all'azione dell'utente
3. Se il *modello* è stato modificato, notifica del cambiamento i listener interessati...

Tuttavia, in questo caso, il cambiamento è inviato al controller, che si occupa di aggiornare la vista

*Cicli infiniti: errore abbastanza frequente
con entrambe le architetture MVC*

1. Un componente Swing nella *vista* viene modificato
 - ♦ Presumibilmente da un'azione dell'utente
2. Il metodo appropriato del *controller* viene invocato
3. Il *modello* viene aggiornato
 - ♦ Esso notifica il controller (o la vista) delle modifiche
4. La *vista* riceve un evento di cambiamento
 - ♦ Dal controller (o dal modello)
 - ♦ La vista tenta di impostare il valore dei componenti appropriati
 - ♦ Anche il componente che ha originato la modifica
5. Il metodo appropriato del *controller* viene invocato (***di nuovo...***)
6. Il *modello* viene aggiornato (***di nuovo...***)

1. Il componente che ha lanciato la modifica iniziale rifiuta di aggiornarsi per la seconda volta
 - Nota che il suo stato non può essere aggiornato mentre sta ancora notificando ai listener la modifica iniziale
 - Succede quando si usano i componenti di testo Swing
2. Il modello rifiuta di inviare una notifica di cambiamento
 - Nota che il valore del secondo aggiornamento coincide con il primo (il suo valore attuale)
 - È sempre una buona pratica di programmazione sicura
 - Succede automaticamente se si usa la classe `PropertyChangeSupport`, nel package `java.beans`
 - Però non impedisce al modello di ricevere un'aggiornamento inutile
3. Nessuna salvaguardia nel modello o nei componenti
 - ⇒ Il programma entra in un ciclo infinito

- ♦ Il *modello* è una lista di poligoni che l'utente ha introdotto nel suo disegno
- ♦ La *vista* è come questi poligoni vengono disegnati
 - Con quali colori, in quale ordine
- ♦ Il *controller* è responsabile di...
 - Modificare il modello cambiando la posizione di un poligono quando l'utente trascina il mouse
 - Informare la vista che qualcosa sta cambiando

- ♦ Una finestra è realizzata creando un oggetto di classe `JFrame`
- ♦ Per disegnare all'interno della finestra è possibile creare un pannello disegnabile
- ♦ In Swing, i pannelli sono oggetti di classe `JPanel`
 - È possibile disegnare al loro interno
 - Sono contenitori di altri componenti

- ♦ Un `JFrame` consiste di quattro piani
- ♦ Normalmente si lavora con il piano detto *content pane*
- ♦ Es. aggiungere un `JPanel` al `JFrame`

```
JFrame f = new JFrame("Title");  
JPanel p = new JPanel();  
Container contentPane = f.getContentPane();  
contentPane.add(p);
```

- ♦ `JPanel` è una sotto-classe di `JComponent`
- ♦ `JComponent` **contiene** `paintComponent(Graphics)`
 - Viene invocato dalla JVM tutte le volte che si presenta la necessità di ridisegnare un componente
 - Le sotto-classi di `JComponent` devono re-implementare questo metodo per fornire un algoritmo di disegno del componente

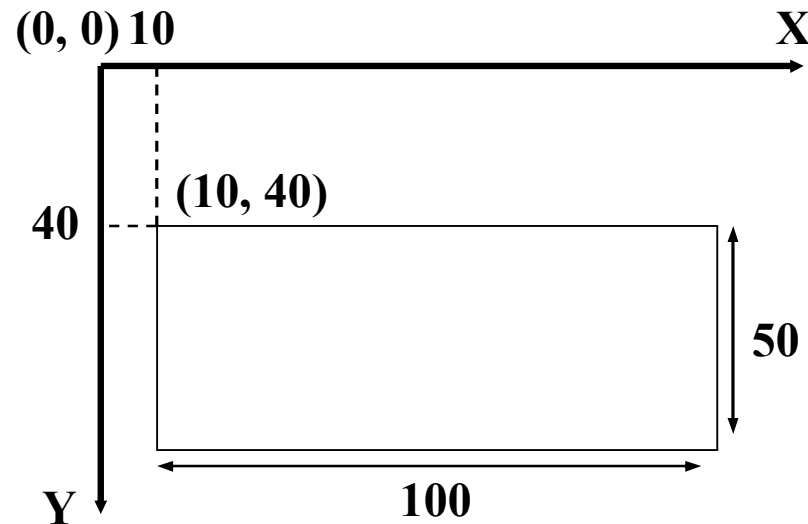
⇒ Per disegnare nel `JPanel`, ne costruiamo una sotto-classe dove re-implementiamo opportunamente `paintComponent(Graphics)`

- ♦ `paintComponent(Graphics)` riceve un oggetto `Graphics` che utilizza per disegnare
- ♦ `Graphics` offre tutti i metodi necessari per disegnare e per gestire i colori ed i font di caratteri

```
public class EditorView extends JPanel {  
    public void paintComponent(Graphics g) {  
        // disegna lo sfondo  
        super.paintComponent(g);  
  
        /*...utilizza il modello per disegnare sul JPanel  
        sfruttando g...*/  
    }  
}
```

- ◆ Ogni **pixel** (**picture element**) all'interno del `JPanel` è identificato da due numeri interi
- ◆ `Graphics` utilizza il sistema di coordinate detto **raster**

```
g.drawRect(10, 40, 100, 50);
```



- ♦ Una GUI viene organizzata mediante componenti contenitori e componenti contenuti
 - I contenitori consentono di organizzare i loro contenuti mediante degli oggetti **layout manager**
 - I contenuti offrono funzionalità all'utente

- ♦ L'aspetto della GUI è determinato da
 - Gerarchia di contenimento
 - Layout manager associati ad ogni contenitore
 - Tipo dei singoli componenti e loro proprietà

The entire interface is governed by a border layout with a panel in each area.

North: two labels

East: two labels, a slider, and a combo box with vertical spacing in a box layout

West: four smaller panels in a vertical box layout. One panel for each label / text field combination and another for the gender radio buttons

Center: several check boxes, a label, and a text field

South: two buttons preceded by horizontal glue in a box layout

Survey

Demographic Survey

Please complete this form. Press the Submit button when complete.

First Name:

Last Name:

Middle Initial:

Gender

☐ Male

☐ Female

Hobbies

☐ Reading

☐ Hiking

☐ Woodworking

☐ Origami

☐ Classic Sports

☐ Xtreme Sports

Other:

Age:

Salary Range:

Clear Submit

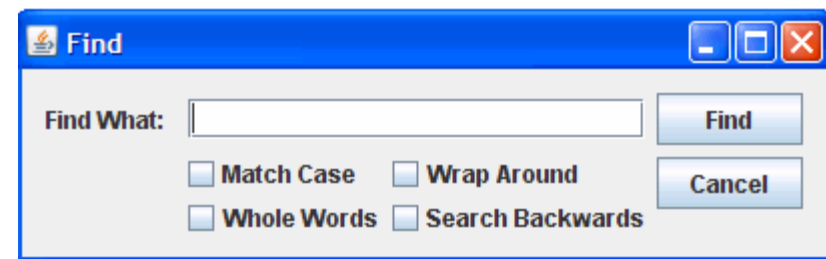
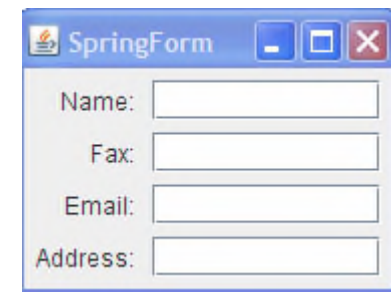
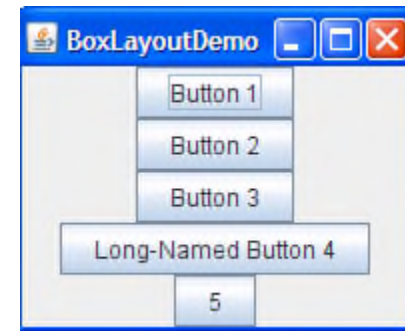
- ♦ Quando un contenitore ospita più di un componente, occorre specificare il modo in cui i figli dovrebbero essere sistemati
- ♦ In diversi contesti, si potrebbe volere usare lo stesso contenitore con gli stessi componenti, ma sistemati in modo diverso

*La responsabilità della disposizione (layout)
deve essere posta in una classe separata*

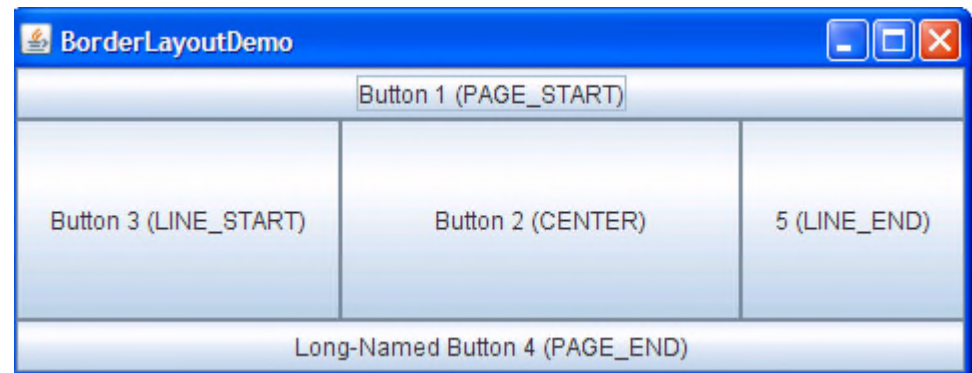
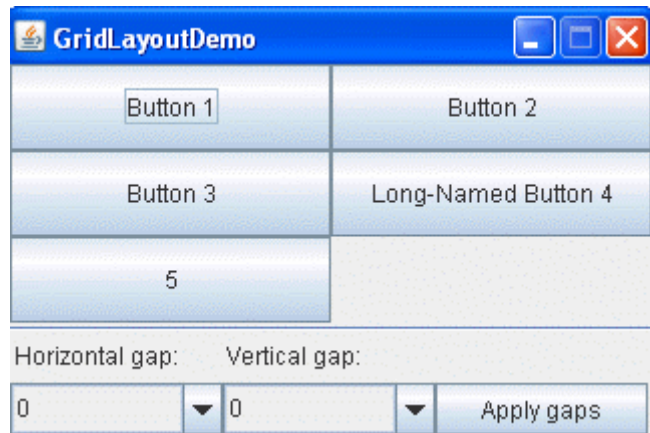
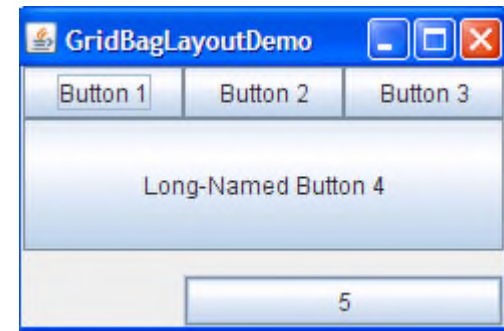
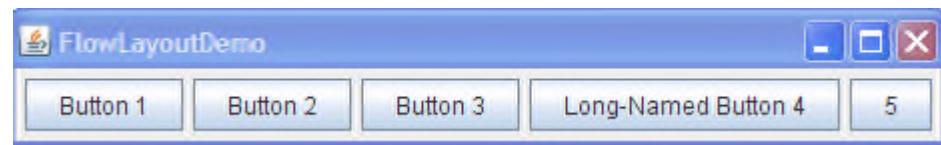
- ♦ La gestione del layout è il processo che determina la dimensione e la posizione dei componenti
- ♦ Ogni contenitore ha un *layout manager*
- ♦ Oggetto che determina il modo in cui i componenti sono disposti all'interno di un contenitore
- ♦ I componenti possono suggerire misura e allineamento
- ♦ Ma è il layout manager ad avere l'ultima parola sulla loro effettiva misura e posizione
- ♦ Ogni contenitore ha un layout manager di default
- ♦ Un nuovo layout manager può essere impostato mediante `setLayout(LayoutManager)`

- ♦ Ogni layout manager ha le sue regole per disporre i componenti all'interno del contenitore
- ♦ Alcuni utilizzano le dimensioni preferite dei componenti, altri utilizzano le dimensioni massime o minime
- ♦ Il layout manager di un contenitore dispone gli oggetti contenuti tutte le volte che un componente è aggiunto al contenitore o che ne cambiano le dimensioni

- ♦ Java fornisce vari layout manager di uso comune
 - Progettati per gestire più componenti assieme
- ♦ In `javax.swing`
 - `BoxLayout`
 - `OverlayLayout`
 - `SpringLayout`
 - `GridLayout`



- ◆ In `java.awt`
 - `FlowLayout`, `GridBagLayout`, `GridLayout`, `BorderLayout`, `CardLayout`



- ◆ Quando si usa il metodo `add` di un contenitore per inserire un componente, bisogna tenere in conto il layout manager del contenitore
- ◆ Alcuni layout manager richiedono di specificare la posizione relativa del componente nel contenitore, usando un argomento aggiuntivo per il metodo `add`
 - Es. `BorderLayout`
- ◆ Occasionalmente, un layout manager può richiede procedure di setup elaborate
 - Es. `GridBagLayout`
- ◆ Molti layout manager, tuttavia, dispongono i componenti semplicemente in base all'ordine con cui sono inseriti nel contenitore

- ♦ In generale, ci sono due diversi approcci per gestire il layout di un contenitore
- 1. Gestire tutti i suoi figli allo stesso tempo
 - Più efficiente, ma richiede di esprimere vincoli di layout complicati (es. `GridBagLayout`)
- 2. Strutturare i suoi figli in gruppi gerarchici
 - Diversi pannelli possono essere usati per costruire la gerarchia e tenere i componenti
 - Aggiunge livelli di struttura, ma tiene semplici i vincoli di layout (es. `BoxLayout`)

- ◆ Si può facilmente cambiare il layout manager usato da un contenitore

- `JPanel pane = new JPanel();`
`pane.setLayout(new BorderLayout());`

- ◆ La classe `Box` fornisce metodi statici per creare componenti (tipo pannelli) con layout già impostato

- `Box hBox = Box.createHorizontalBox();`
 - `Box vBox = Box.createVerticalBox();`

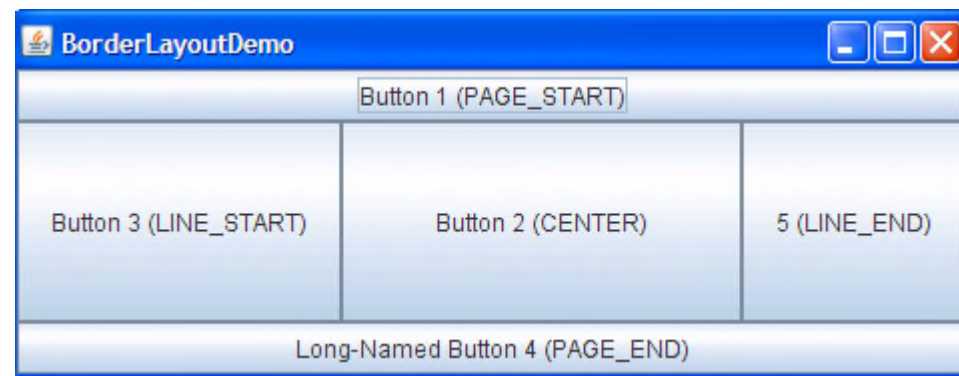
- ◆ Anche se è raccomandato l'uso di un layout manager, si può anche farne a meno
- ◆ Se la proprietà layout di un contenitore viene impostata a `null`, il contenitore non userà alcun layout manager
- ◆ Con questa strategia, chiamata posizionamento assoluto, bisogna specificare la dimensione e la posizione di ciascun componente nel contenitore
- ◆ Svantaggi
 - Non si adatta bene quando il contenitore di primo livello (finestra) viene ridimensionato
 - Non si adatta bene alle differenze tra utenti e sistemi, es. diverse dimensioni dei caratteri

- ♦ A volte bisogna personalizzare le dimensioni suggerite da un componente al layout manager del suo contenitore, in modo che il componente sia disposto bene
 - `setMinimumSize`, `setPreferredSize`, `setMaximumSize`
- ♦ Oltre a fornire suggerimenti sulla sua dimensione, un componente può fornirne anche per il suo allineamento
 - Per esempio, si può specificare che due componenti abbiano i bordi in alto allineati
 - `setAlignmentX`, `setAlignmentY`
- ♦ Non tutti i layout prestano attenzione dimensioni e allineamento suggeriti

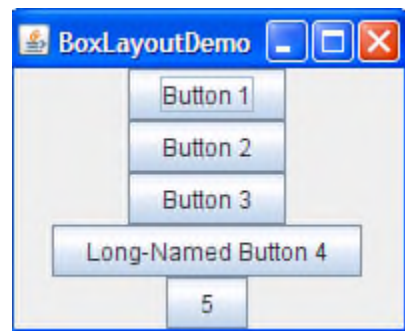
- ◆ Mette tutti i componenti possibili su una riga e poi continua in quella successiva
- ◆ Componenti disposti nell'ordine con cui sono aggiunti
- ◆ Il default è che i componenti sono centrati sulle righe



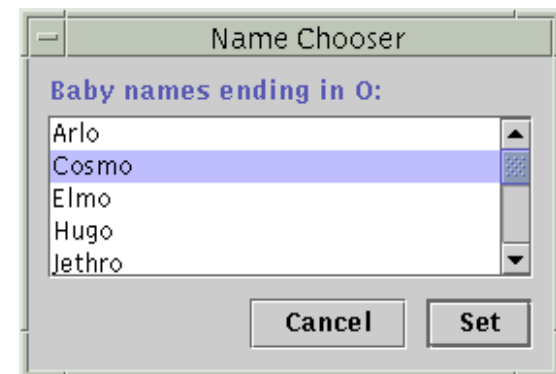
- ◆ Cinque aree in cui si può aggiungere un componente
- ◆ L'area di centro si allarga al massimo in modo da riempire tutto lo spazio non utilizzato



- ◆ Layout manager di utilità generale incluso in Swing
- ◆ Può essere considerato una versione più avanzata di `FlowLayout`
- ◆ Componenti disposti uno sotto l'altro (con il primo componente in cima)
- ◆ Oppure in in una riga, procedendo da sinistra a destra



- ◆ Creando uno o più contenitori “leggeri”, che usano `BoxLayout`, si possono ottenere layout per i quali spesso si usava il più complesso `GridBagLayout`
- ◆ La figura seguente mostra una GUI che usa due istanze di `BoxLayout`
 - Prima di tutto, un box layout dispone dall’alto in basso un’etichetta, una lista con scorrimento e un pannello
 - In basso, quest’ultimo pannello ha un altro box layout che dispone i due bottoni uno di fianco all’altro, da sinistra a destra

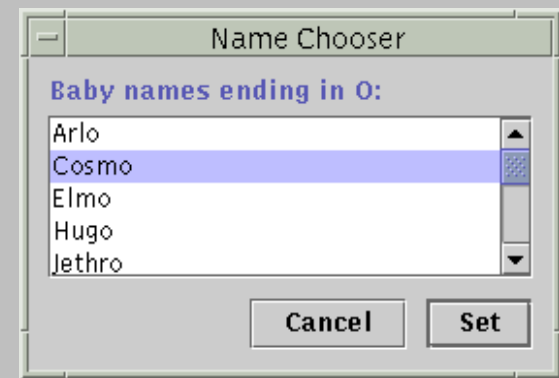


```
JScrollPane listScroller = new JScrollPane(list);
listScroller.setPreferredSize(new Dimension(250, 80));
listScroller.setMinimumSize(new Dimension(250, 80));
listScroller.setAlignmentX(LEFT_ALIGNMENT);

// Lay out the buttons from left to right.
JPanel buttonPane = new JPanel();
buttonPane.setLayout(new BoxLayout(buttonPane, BoxLayout.X_AXIS));
buttonPane.add(Box.createHorizontalGlue());
buttonPane.add(cancelButton);
buttonPane.add(Box.createRigidArea(new Dimension(10, 0)));
buttonPane.add(setButton);
buttonPane.setAlignmentX(LEFT_ALIGNMENT);

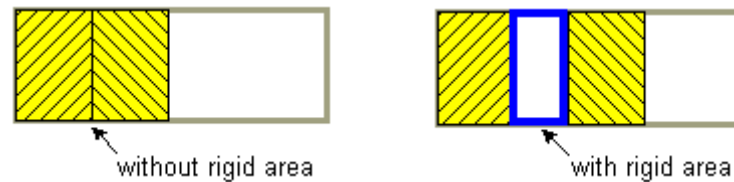
// Lay out label, scroll pane and button panel from top to bottom.
JPanel listPane = new JPanel();
listPane.setLayout(new BoxLayout(listPane, BoxLayout.Y_AXIS));
listPane.add(new JLabel(labelText));
listPane.add(Box.createRigidArea(new Dimension(0, 5)));
listPane.add(listScroller);
listPane.add(Box.createRigidArea(new Dimension(0, 10)));
listPane.add(buttonPane);
listPane.setBorder(
    BorderFactory.createEmptyBorder(10, 10, 10, 10));

setContentPane(listPane);
```



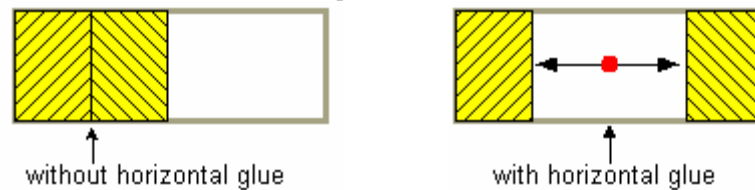
- ◆ Spazio di dimensione fissa tra due componenti
- ◆ Es. inserire 5 pixel tra due componenti in un box da sinistra a destra

```
container.add(firstComponent);  
container.add(Box.createRigidArea(  
    new Dimension(5,0)));  
container.add(secondComponent);
```



- ◆ Specifica dove dovrebbe andare lo spazio in eccesso
- ◆ Da immaginare come una colla gelatinosa
 - Di base, non richiede spazio, ma è elastica ed espandibile
 - Si espande al massimo tra i componenti a cui è attaccata
- ◆ Es. in un box da sinistra a destra, inserire spazio tra due componenti, invece che alla loro destra

```
container.add(firstComponent);  
container.add(Box.createHorizontalGlue());  
container.add(secondComponent);
```



- ◆ Aggiunto in Java 6
- ◆ Si imposta separatamente la disposizione lungo i due assi: orizzontale e verticale
- ◆ Non bisogna curarsi della disposizione verticale quando si definisce quella orizzontale, e viceversa
- ◆ Focalizzandosi su una sola dimensione, si ha da risolvere solo metà del problema alla volta
- ◆ Più facile che gestire entrambe le dimensioni assieme
- ◆ Però ciascun componente deve essere definito due volte nel layout, altrimenti viene generata un'eccezione

- ◆ Due tipi di gruppi: sequenziali e paralleli, che possono essere combinati in una gerarchia annidata

- I gruppi possono contenere componenti o altri gruppi

1. Gruppo sequenziale

- ◆ I componenti sono semplicemente posti in ordine l'uno dopo l'altro, Come in BoxLayout

- Dimensione = somma delle dimensioni degli elementi contenuti

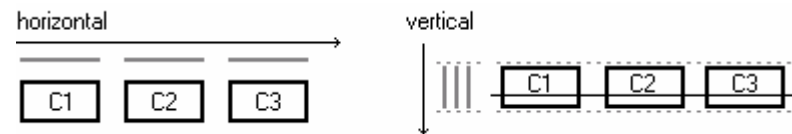
2. Gruppo parallelo

- ◆ I componenti sono posti nello stesso spazio
- Dimensione = dimensione dell'elemento più ampio

1. Allineati in alto, basso o baseline lungo l'asse verticale
2. Allineati a sinistra, destra o centro lungo l'asse orizzontale

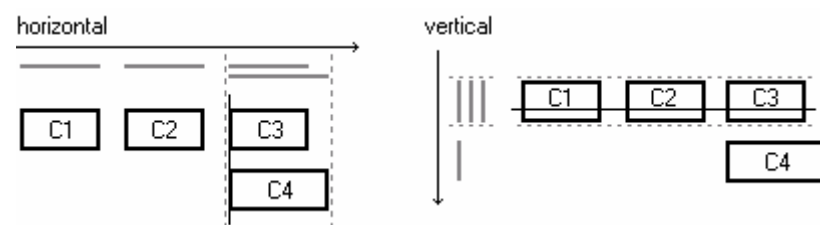
◆ Esempio 1

```
horizontal layout = sequential group { c1, c2, c3 }  
vertical layout = parallel group (BASELINE) { c1, c2, c3 }
```



◆ Esempio 2

```
horizontal layout = sequential group { c1, c2, parallel group (LEFT) { c3, c4 } }  
vertical layout = sequential group { parallel group (BASELINE) { c1, c2, c3 }, c4 }
```



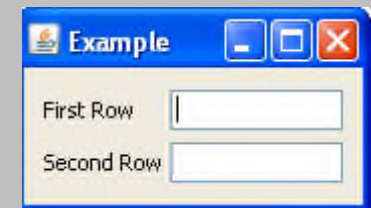
```
GroupLayout layout = new GroupLayout(panel);
panel.setLayout(new GroupLayout(panel));
layout.setAutoCreateGaps(true);
layout.setAutoCreateContainerGaps(true);

SequentialGroup h = layout.createSequentialGroup();
h.addComponent(c1).addComponent(c2);
h.addGroup(layout.createParallelGroup(LEADING)
    .addComponent(c3).addComponent(c4));
layout.setHorizontalGroup(h);

SequentialGroup v = layout.createSequentialGroup();
v.addGroup(layout.createParallelGroup(BASELINE)
    .addComponent(c1).addComponent(c2).addComponent(c3));
v.addComponent(c4);
layout.setVerticalGroup(v);
```

```
//...
SequentialGroup h = layout.createSequentialGroup();
h.addGroup(layout.createParallelGroup().
    addComponent(label1).addComponent(label2));
h.addGroup(layout.createParallelGroup().
    addComponent(tf1).addComponent(tf2));
layout.setHorizontalGroup(h);

SequentialGroup v = layout.createSequentialGroup();
v.addGroup(layout.createParallelGroup(BASELINE).
    addComponent(label1).addComponent(tf1));
v.addGroup(layout.createParallelGroup(BASELINE).
    addComponent(label2).addComponent(tf2));
layout.setVerticalGroup(v);
```



- ♦ Se si crea e usa la GUI nella maniera giusta...
non c'è di che preoccuparsi per i thread
 - Per esempio, nel caso di una applet, è corretto costruire la GUI nel metodo `init`
 - Nel caso di una applicazione, vedremo alcuni pattern comuni che sono corretti

- ♦ Invece, si può incorrere in **problemi (!)**:
 - Se il programma manipola la GUI dal thread principale
 - Se crea thread che hanno effetto diretto sulla GUI
 - O se manipola la GUI già visibile in risposta a qualsiasi cosa che non sia un evento standard dell'interfaccia

- ◆ Regola del singolo thread

Una volta che un componente Swing è stato realizzato, tutto il codice che influisce o dipende da quel componente dovrebbe essere eseguito nell'event-dispatching thread

- ◆ Questa regola può sembrare ostica, ma per molti programmi semplici non c'è affatto da preoccuparsi per i thread

- ♦ Prima di continuare, occorre definire il termine *realizzato*
- ♦ Realizzato significa che il componente è stato disegnato a schermo, o che è pronto per essere disegnato
- ♦ Una finestra è realizzata dopo l'invocazione di:
 - `setVisible(true)`, `show()`, `pack()`
 - Nota: il metodo `show()` fa la stessa cosa di `setVisible(true)`
- ♦ Una volta che una finestra è realizzata, tutti i componenti che contiene sono pure realizzati
- ♦ Un'altra maniera di realizzare un componente è di aggiungerlo ad un contenitore che è già realizzato

- ◆ Ci sono alcune eccezioni alla regola del singolo thread
 - ... Secondo cui tutto il codice che usa un componente realizzato deve essere eseguito nell'event-dispatching thread

1. Alcuni metodi sono *thread safe*

- Nella documentazione Java, questi metodi riportano il testo:
- “*This method is thread safe, although most Swing methods are not*”

2. La GUI di una applicazione può spesso essere costruita e mostrata nel thread principale

- Finchè nessun componente (Swing o altro) è stato realizzato nell'attuale ambiente di esecuzione, si può costruire e mostrare una GUI nel thread principale di una applicazione
- In generale, si può costruire (ma non mostrare) una GUI in qualsiasi thread, a patto di non fare chiamate che usino o manipolino componenti già realizzati
- Se il thread principale non esegue codice di GUI dopo la chiamata a `setVisible`, questo significa che tutto il lavoro della GUI si sposta dal thread principale all'event-dispatching thread, e in pratica il codice è thread safe

```
// Thread-safe example
public class MyApplication {
    public static void main(String[] args) {
        JFrame f = new JFrame("...");
        //Add components to the frame here...
        f.pack();
        f.setVisible(true);
        //Don't do any more GUI work here!
    }

    //All manipulation of the GUI -- setText, getText, etc. --
    //is performed in event handlers such as actionPerformed().
}
```

3. La GUI di una applet può essere costruita nel metodo `init`

- I browser non disegnano una applet se non dopo averne chiamati i metodi `init` e `start`
- Quindi, costruire la GUI nel metodo `init` di una applet è corretto
- ... Finchè non vengono invocati `show()` o `setVisible(true)` sull'oggetto applet

4. Due metodi di `JComponent` sono sicuri da chiamare da qualsiasi thread: `repaint` e `revalidate`
- Questi metodi accodano le richieste affinché siano eseguite dall'event-dispatching thread

5. Le liste di listener possono essere modificate da qualsiasi thread
- È sempre sicuro chiamare i metodi `addXYZListener` e `removeXYZListener`
 - Le operazioni di aggiunta/rimozione non influiscono sul dispatch di un evento che sia in corso di elaborazione

- ♦ La maggior parte del lavoro dell'interfaccia si svolge in maniera naturale nell'event-dispatching thread
 - Una volta che la GUI è visibile, la maggior parte dei programmi è guidata dagli eventi – azioni di bottoni o click del mouse – che sono sempre gestiti nell'event-dispatching thread
- ♦ Tuttavia, alcuni programmi devono eseguire del lavoro sulla GUI dopo che questa è visibile, ma a partire da thread diversi
 - Programmi che devono eseguire compiti lunghi
 - Programmi le cui interfacce devono essere aggiornate in risposta a eventi non standard – ossia esterni all'interfaccia

- ♦ Alcuni programmi richiedono l'esecuzione di lunghe computazioni
 - Questo genere di programmi dovrebbe generalmente mostrare una certa interfaccia mentre si svolge il lungo compito, e poi aggiornare o cambiare l'interfaccia
 - Il compito non dovrebbe svolgersi nell'event-dispatching thread; altrimenti, si fermerebbe il ridisegno e la gestione degli eventi
 - Tuttavia, dopo l'inizializzazione, gli aggiornamenti e cambi di GUI dovrebbero svolgersi nell'event-dispatching thread, per ragioni di *thread-safety*

- ♦ Alcuni programmi devono aggiornare l'interfaccia in risposta ad eventi non-standard
 - Per esempio, si supponga che un programma server possa ricevere richieste da altri programmi in esecuzione su macchine remote
 - Queste richieste possono arrivare in qualsiasi momento, e provocano l'invocazione di qualche metodo del server in qualche thread, possibilmente sconosciuto all'interfaccia
 - Come può quel metodo aggiornare la GUI? Eseguendo il codice di aggiornamento della GUI nell'event-dispatching thread

- ◆ La classe `SwingUtilities` fornisce due metodi che aiutano ad eseguire codice nell' event-dispatching thread
- ◆ `invokeLater`
 - Richiede che del codice sia eseguito nell'event-dispatching thread
 - Questo metodo ritorna immediatamente, senza aspettare che il codice sia eseguito
- ◆ `invokeAndWait`
 - Funziona come `invokeLater`, eccetto per il fatto che questo metodo aspetta che il codice sia eseguito
 - Come regola, si dovrebbe usare di preferenza `invokeLater` piuttosto che questo metodo

```
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        component.doSomething();  
    }  
});
```

```
SwingUtilities.invokeAndWait(new Runnable() {  
    public void run() {  
        JOptionPane.showMessageDialog(  
            myMainFrame, "Hello There");  
    }  
});
```

```
void printTextField() throws Exception {
    final String[] strs = new String[2];

    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            strs[0] = textField0.getText();
            strs[1] = textField1.getText();
        }
    });

    System.out.println(strs[0] + " " + strs[1]);
}
```

```
private static void createAndShowGUI() {
    JFrame frame = new JFrame("...");
    // Add components to the frame here...
    frame.pack();
    frame.setVisible(true);
}

public static void main(String[] args) {
    // Schedule a job for the event-dispatching thread:
    // creating and showing this application's GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
```