Un approccio pratico a Java Swing e MVC

Sommario

Java Swing

L'obiettivo di *questa guida* è dare, tramite esempi pratici, gli **strumenti fondamentali** per lo *sviluppo agevole* di interfacce grafiche.

Wireframe

Per sviluppare un'interfaccia grafica (per un sito web, un'applicazione, un gioco etc…) è utile disegnare un **wireframe** fatto di **rettangoli**, **testo** e **icone** come in Figura 1.

Il **wireframe** serve perché è difficile progettare un'interfaccia **intuitiva** e **funzionale**. Una volta progettata l'interfaccia **scrivere il codice è semplice**, perché abbiamo un'idea chiara di quello che vogliamo, e dobbiamo solo disegnarlo.

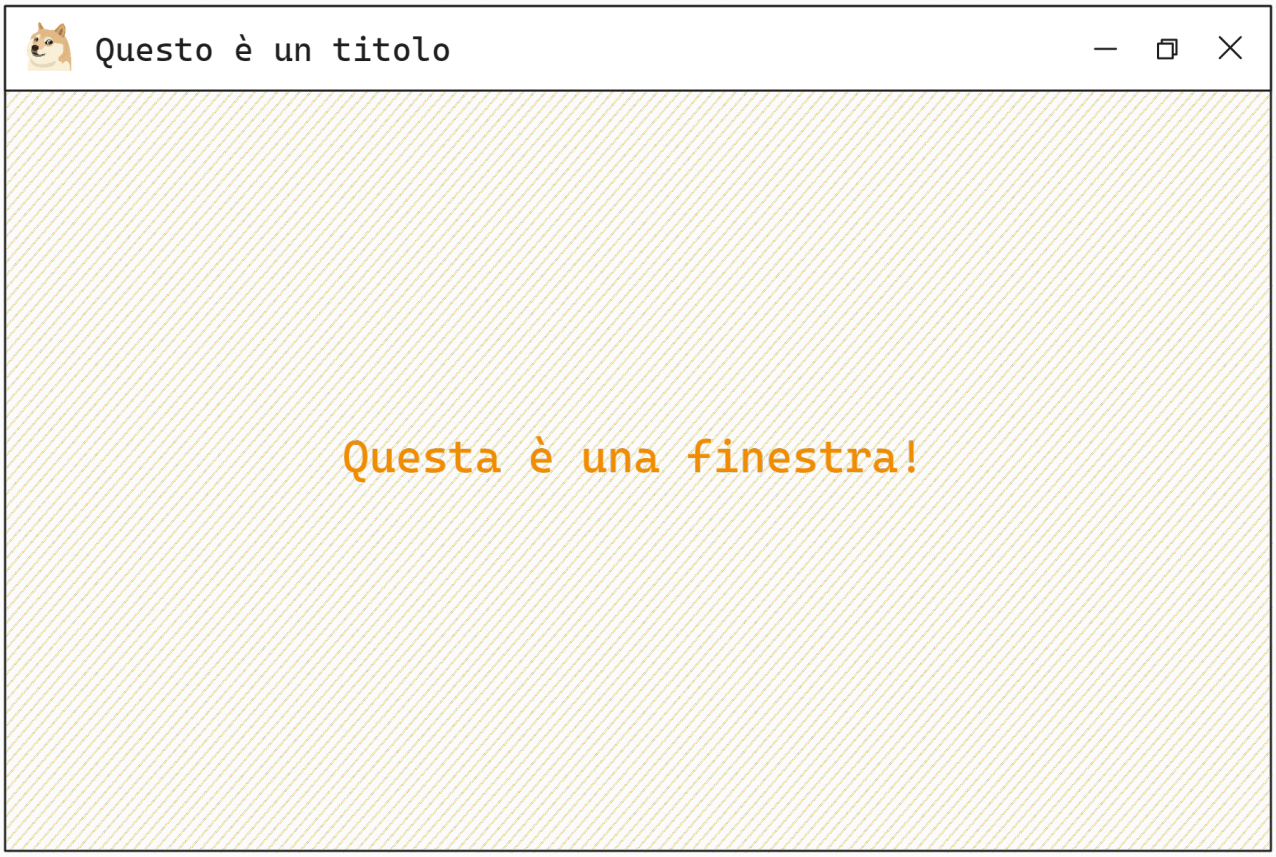


Figura 1: esempio di wireframe disegnato con excalidraw (doc)

Proviamo ad implementare il **wireframe** in Figura 1

Implementazione passo passo di un wireframe

Il codice completo è **in fondo alla spiegazione**

Creare una finestra con JFrame (doc)

```
JFrame frame = new JFrame("Questo \u00E8 un titolo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

try {
  frame.setIconImage(ImageIO.read(new File("icon.png")));
} catch (IOException e) { }

frame.setSize(600, 400);
frame.setLocationRelativeTo(null);
frame.setVisible(true);
```

- `new JFrame(String title)` crea una finestra *invisibile* con il titolo specificato
- `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` termina il programma quando la finestra viene chiusa (di default la finestra viene *nascosta*)
- `setIconImage(Image image)` imposta l'icona in alto a sinistra della finestra

> **i Nota**
>
> Non lasciate l'icona di default! Fate vedere un po' di attenzione ai dettagli :)

- `setSize(int width, int height)` imposta la dimensione della finestra

> **i Nota**
>
> Esistono altre strategie per dimensionare la finestra:
> - `frame.pack()` imposta larghezza e altezza al valore *minimo* che rispetta il contenuto della finestra
>
> *Se la finestra non ha contenuto*, la larghezza e l'altezza vengono impostati a 0

- `setLocationRelativeTo(null)` posiziona la finestra al centro dello schermo
- `setVisible(true)` rende la finestra *visibile*

Errori comuni

Se la finestra **non rispetta la dimensione impostata** con `setSize(int width, int height)` probabilmente state usando anche `pack()` nel codice (non vanno usati entrambi).

Se usando `pack()` **larghezza e altezza sono impostati a 0** è perché state usando `pack()` *prima* di aggiungere il contenuto alla finestra.

Aggiungere contenuti a una finestra con `JPanel` (doc)

```
JPanel panel = new JPanel();
JLabel label = new JLabel("Questa \u00E8 una finestra!");

panel.add(label);
frame.add(panel);
```

Sia `JFrame` sia `JPanel` sono `java.awt.Container`, quindi possiamo aggiungere contenuto (testo, immagini, pulsanti etc…) al loro interno tramite `add(Component comp)`.

- `JPanel` occuperà l'intero spazio disponibile nella finestra
- `JLabel` serve a visualizzare testo (`"Questa è una finstra"` Figura 1)

Colori, font e personalizzare lo sfondo con `paintComponent(Graphics g)`

Ora l'obiettivo è quello di colorare lo sfondo e il testo come in Figura 1

> **i Nota**
>
> Normalmente un **wireframe** *non prevede* colori o scelte stilistiche, ma nel caso di un progetto piccolo possiamo permetterci di usarlo come se fosse un design

```
JPanel panel = new JPanel() {
  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    int density = 5;
    g.setColor(Color.decode("#ffec99"));
    for (int x = 0; x <= getWidth() + getHeight(); x += density)
      g.drawLine(x, 0, 0, x);
  }
};
panel.setBackground(Color.WHITE);

JLabel label = new JLabel("Questa \u00E8 una finestra!");
label.setForeground(Color.decode("#f08c00"));
label.setFont(new Font("Cascadia Code", Font.PLAIN, 22));

panel.add(label);
frame.add(panel);
```

`JPanel` viene visualizzato invocando il metodo `paint(Graphics g)` tramite `repaint()`. A sua volta `paint(Graphics g)` invoca in ordine:

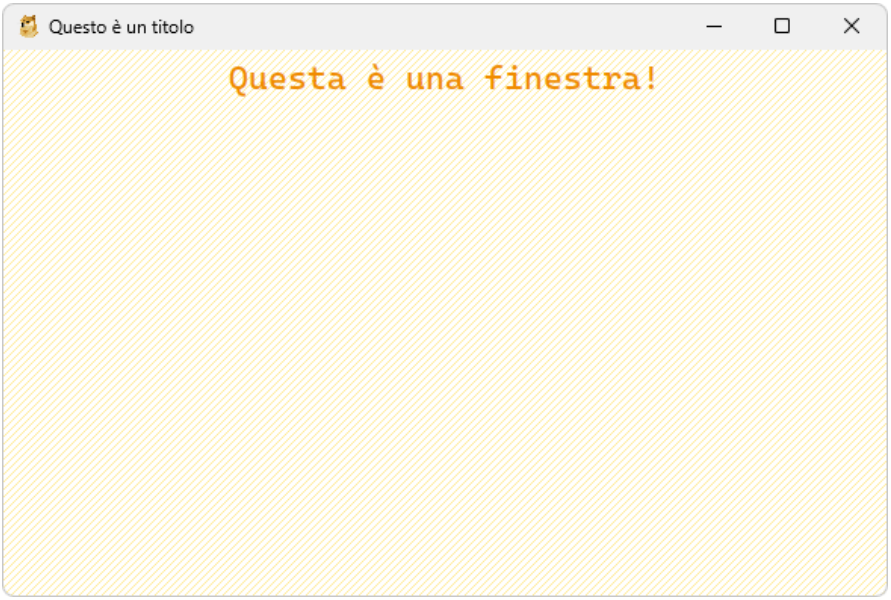
- `paintComponent(Graphics g)`
- `paintBorder(Graphics g)`
- `paintChildren(Graphics g)`

Creando una **classe anonima** possiamo sovrascrivere il comportamento di uno di questi metodi. Nel nostro esempio, sovrascriviamo `paintComponent(Graphics g)` per disegnare lo sfondo con le linee oblique in Figura 1.

Questo approccio è molto flessibile, perché con `Graphics g` possiamo disegnare immagini, testo e figure programmaticamente (quindi eventualmente **animazioni**).

> **i Nota**
>
> Il font `"Cascadia Code"` non è installato di default, provate anche con altri font

Risultato



Centrare un elemento con GridBagLayout (doc)

```
JPanel panel = new JPanel(new GridBagLayout());
```

Il costruttore `JPanel(LayoutManager layout)` permette di specificare una **strategia** per **posizionare** e **dimensionare** il contenuto di un `JPanel` in **automatico**:
- non bisogna calcolare a mano `x`, `y`, `width` e `height` dei componenti, lo fa il `LayoutManager`
- funziona anche quando la **finestra viene ridimensionata**

> **i Nota**
>
> `LayoutManager` è un esempio di Strategy Pattern (doc)

Per **centrare un elemento** in un panel si usa un `GridBagLayout`

Implementazione

```java
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.GridBagLayout;
import java.io.File;
import java.io.IOException;

public class App {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Questo \u00E8 un titolo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    try {
      frame.setIconImage(ImageIO.read(new File("icon.png")));
    } catch (IOException e) {
    }

    JPanel panel = new JPanel(new GridBagLayout()) {
      @Override
      protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        int density = 5;
        g.setColor(Color.decode("#ffec99"));
        for (int x = 0; x <= getWidth() + getHeight(); x += density)
          g.drawLine(x, 0, 0, x);
      }
    };
    panel.setBackground(Color.WHITE);

    JLabel label = new JLabel("Questa \u00E8 una finestra!");
    label.setForeground(Color.decode("#f08c00"));
    label.setFont(new Font("Cascadia Code", Font.PLAIN, 22));

    panel.add(label);
    frame.add(panel);

    frame.setSize(600, 400);
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
  }
}
```

Interfacce con Swing senza variabili d’appoggio

Java mette a disposizione uno strumento che si chiama **instance initialization block**, un blocco di codice che viene eseguito dopo aver invocato il costruttore. È specialmente comodo quando si istanziano **classi anonime**.

```
class Persona {
  String nome;
}

class Main {
  public static void main(String[] args) {
    Persona rossi = new Persona() {
      {
        nome = "Rossi";
      }
    };
    System.out.println(rossi.nome); // Rossi
  }
}
```

Possiamo sfruttare questa strategia per riscrivere il codice di prima senza variabili.

```
public class App extends JFrame {
  App() {
    super("Questo \u00E8 un titolo");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    try { setIconImage(ImageIO.read(new File("icon.png"))); } catch (IOException e) { }

    add(new JPanel(new GridBagLayout()) {
      {
        setBackground(Color.WHITE);
        add(new JLabel("Questa \u00E8 una finestra!") {
          {
            setForeground(Color.decode("#f08c00"));
            setFont(new Font("Cascadia Code", Font.PLAIN, 22));
          }
        });
      }

      @Override
      protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        int density = 5;
        g.setColor(Color.decode("#ffec99"));
        for (int x = 0; x <= getWidth() + getHeight(); x += density)
          g.drawLine(x, 0, 0, x);
      }
    });

    setSize(600, 400);
    setLocationRelativeTo(null);
    setVisible(true);
  }

  public static void main(String[] args) {
    new App();
  }
}
```

Modificare l'aspetto dell'interfaccia con UIManager (doc)

Per gestire l'aspetto dei componenti e il loro comportamento in Java Swing viene usata la classe LookAndFeel (doc). È possibile modificare **globalmente** l'aspetto del `LookAndFeel` impostato tramite la classe `UIManager`.

Ad esempio, per impostare il `Font` di default di tutti i `JLabel` a `"Cascadia Code"`

```
UIManager.put("Label.font", new Font("Cascadia Code", Font.PLAIN, 14));
```

> **i Nota**
>
> `UIManager` è un esempio di Singleton Pattern (doc)

```
import java.awt.Color;
import java.awt.Font;

import javax.swing.UIManager;

public class App {
  static {
    UIManager.put("Label.font", new Font("Cascadia Code", Font.PLAIN, 14));
    UIManager.put("Label.foreground", Color.DARK_GRAY);
    UIManager.put("Label.background", Color.WHITE);

    UIManager.put("Button.font", new Font("Cascadia Code", Font.PLAIN, 14));
    UIManager.put("Button.foreground", new Color(224, 49, 49));
    UIManager.put("Button.background", new Color(255, 201, 201));

    UIManager.put("Button.highlight", Color.WHITE);
    UIManager.put("Button.select", Color.WHITE);
    UIManager.put("Button.focus", Color.WHITE);

    UIManager.put("Panel.background", new Color(233, 236, 239));
  }
}
```

> **i Nota**
>
> Un elenco delle possibili chiavi (doc)

Per stampare l'elenco di chiavi disponibili:

```
javax.swing.UIManager.getDefaults().keys().asIterator().forEachRemaining(System.out::println);
```

Pulsanti

Immagini

Animazioni

Layout Manager

Il costruttore `JPanel(LayoutManager layout)` permette di specificare una **strategia** per **posizionare** e **dimensionare** il contenuto di un `JPanel` in **automatico**:
- non bisogna calcolare a mano `x`, `y`, `width` e `height` dei componenti, lo fa il `LayoutManager`
- funziona anche quando la **finestra viene ridimensionata**

> **i Nota**
>
> `LayoutManager` è un esempio di Strategy Pattern (doc)

BorderLayout (doc)

Supponiamo di voler implementare questo wireframe

Figura 2: caso d'uso di un `BorderLayout`

Abbiamo un rettangolo con le statistiche in alto, e il restante spazio è occupato da un rettangolo centrale con un pulsante.

Barra delle statistiche e menu di gioco

```
frame.add(new JPanel(new BorderLayout(10, 10)) {
    {
        setBackground(new Color(240, 255, 240)); // verdignolo
        setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        add(
          new JPanel() {{ setBackground(new Color(220, 220, 255)); }},  // bluastro
          BorderLayout.NORTH
        );

        add(
          new JPanel() {{ setBackground(new Color(220, 220, 255)); }},
          BorderLayout.CENTER
        );
    }
});
```

Il costruttore `BorderLayout(int vgap, int hgap)` imposta uno “spazio” verticale e orizzontale fra due componenti.

Con `setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10))` impostiamo un bordo trasparente (per lasciare uno spazio dal bordo della finestra)

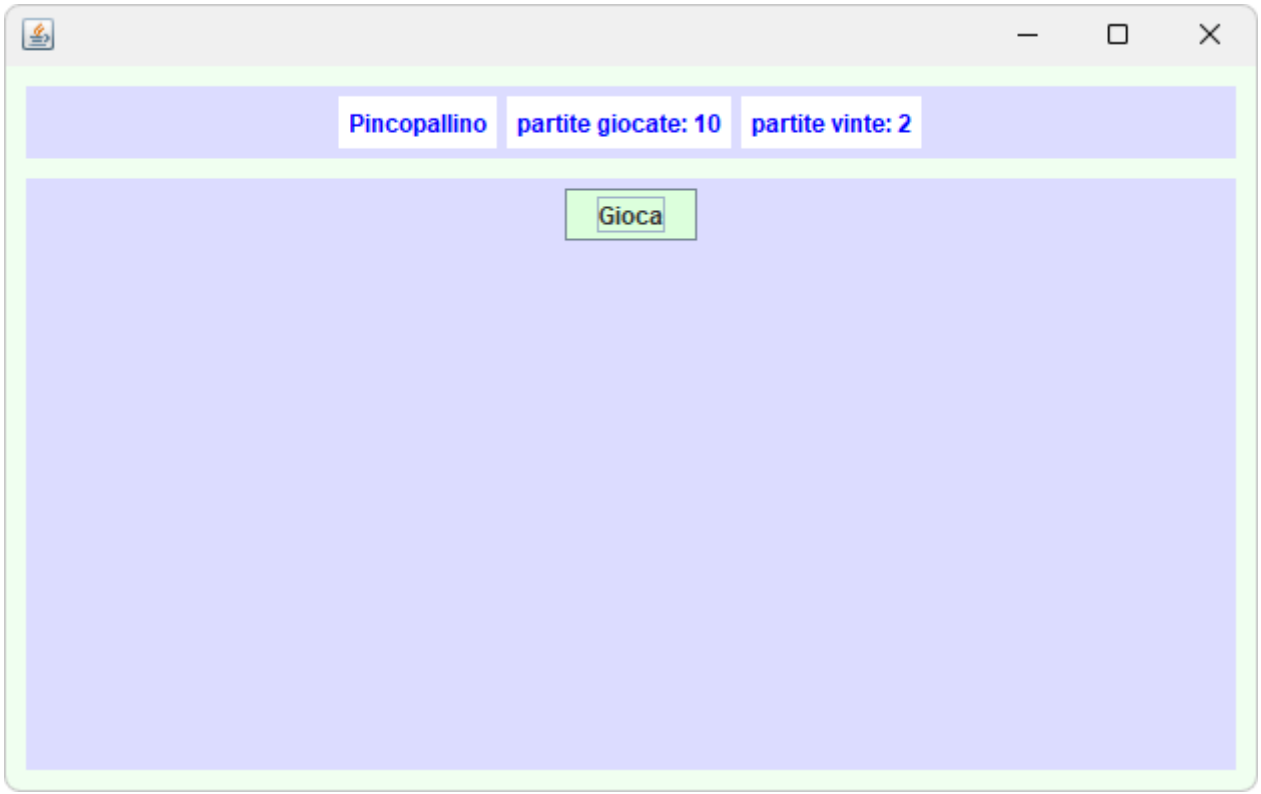
Quando aggiungo un elemento ad un container, posso specificare come deve essere trattato tramite il metodo `add(Component comp, Object constraints)`: in base al layout del container, `Object constraints` avrà un significato diverso.

> **i Nota**
>
> `BorderFactory` è un esempio di Factory Pattern (doc)

Aggiungere uno sfondo ai `JLabel`

```
frame.add(new JPanel(new BorderLayout(10, 10)) {
    {
        setBackground(new Color(240, 255, 240));
        setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        add(new JPanel() {
            {
                setBackground(new Color(220, 220, 255));

                String[] labels = {"Pincopallino", "partite giocate: 10", "partite vinte: 2"};

                for (String label : labels)
                    add(new JLabel(label) {
                        {
                            setForeground(Color.BLUE);
                            setBackground(Color.WHITE);
                            setOpaque(true);
                            setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
                        }
                    });
            }
        }, BorderLayout.NORTH);

        add(new JPanel() {
            {
                setBackground(new Color(220, 220, 255));
                add(new JButton("Gioca") {{ setBackground(new Color(220, 255, 220)); }});
            }
        }, BorderLayout.CENTER);
    }
});
```

Nota interessante: di default, lo sfondo di un `JLabel` è trasparente, per renderlo visibile bisogna usare `setOpaque(true)`



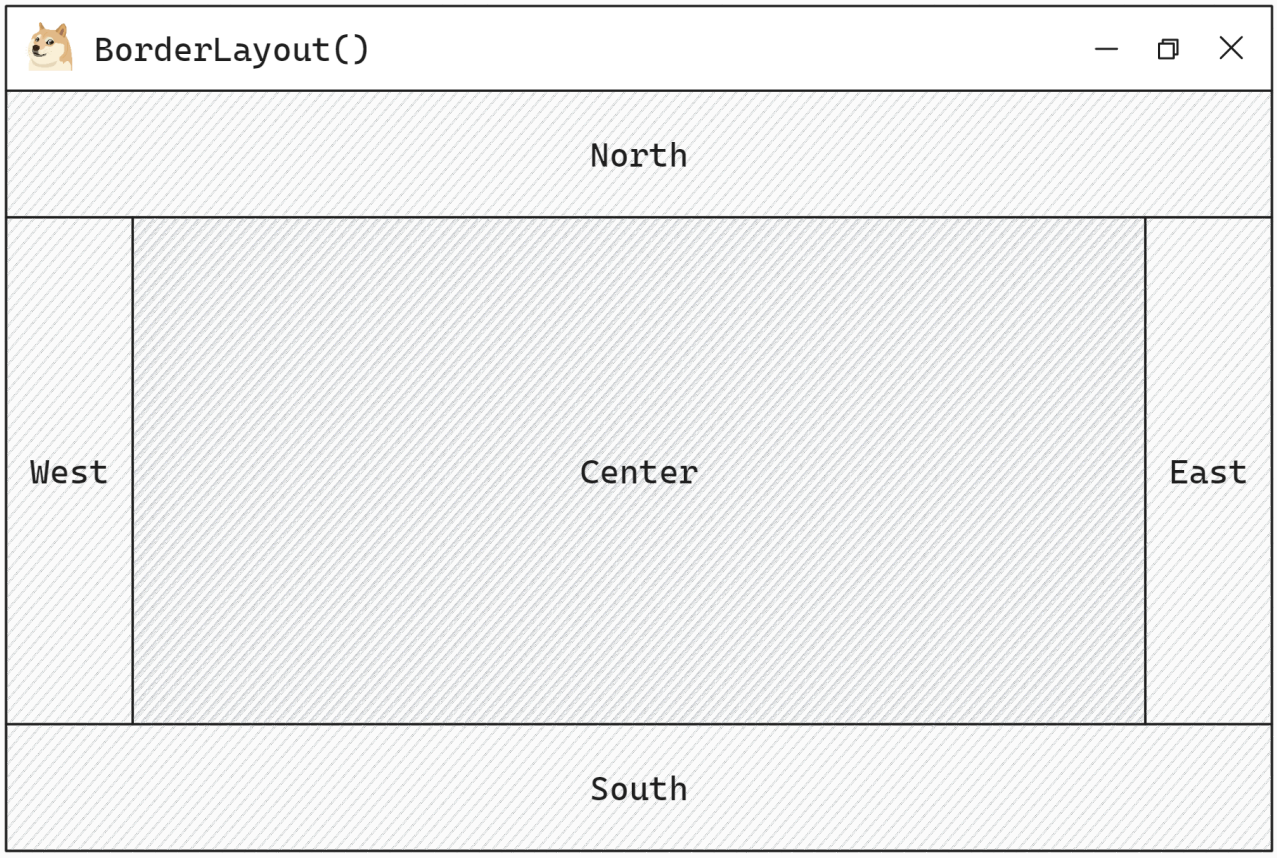
Come funziona il `BorderLayout` in generale?

```
new JPanel(new BorderLayout()) {
  {
    add(new JPanel(), BorderLayout.CENTER);
    add(new JPanel(), BorderLayout.NORTH);
    add(new JPanel(), BorderLayout.SOUTH);
    add(new JPanel(), BorderLayout.WEST);
    add(new JPanel(), BorderLayout.EAST);
  }
}
```

Il `BorderLayout` permette di specificare in quale posizione mettere un componente, secondo certe regole:

- il componente `CENTER` occuperà tutto lo spazio possibile
- i componenti `NORTH` e `SOUTH` avranno larghezza massima (indipendentemente dalla larghezza impostata) e avranno altezza minima, o, se impostata, l'altezza impostata
- i componenti `WEST` e `EAST` avranno altezza massima (indipendentemente dall'altezza impostata) e avranno larghezza minima, o, se impostata, la larghezza impostata

Il costruttore `BorderLayout(int vgap, int hgap)` imposta uno "spazio" verticale e orizzontale fra due componenti.



CardLayout (doc)

Menu, impostazioni e partita *(cambiare schermata con Sigleton e Observer)*

Il `CardLayout` è molto utile quando abbiamo più schermate (menu principale, impostazioni, partita etc…)

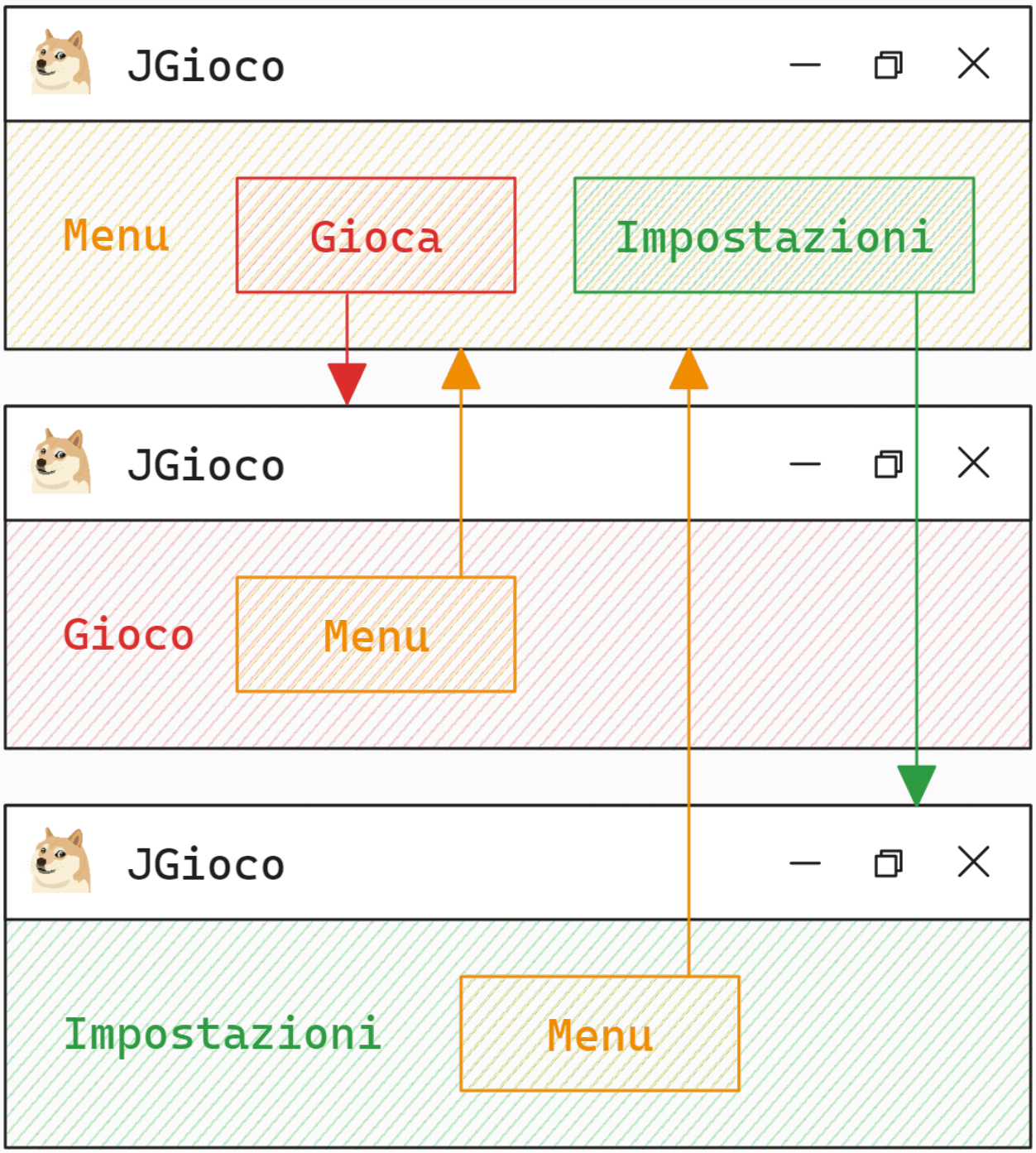


Figura 3: caso d'uso di un `CardLayout`

L’implementazione più rozza

```
public class App extends JFrame {
  App() {
    super("JGioco");
    // ...

    JPanel panel;

    add(panel = new JPanel(new CardLayout()) {
      {
        add(new MenuPanel(), "Menu");
        add(new SettingsPanel(), "Settings");
        add(new GamePanel(), "Game");
      }
    });

    ((CardLayout) panel.getLayout()).show(panel, "Menu");
    // ...
  }

  public static void main(String[] args) { new App(); }
}
```

Ad ogni schermata bisogna associare un **nome** quando viene aggiunta al `JPanel` con il `CardLayout`. Per visualizzare la schermata che vogliamo basta usare il metodo
- `show(Container parent, String name)` del `CardLayout`

```
((CardLayout) panel.getLayout()).show(panel, "Menu");
```

Questo approccio ha 2 problemi:
- è facile sbagliare il nome della schermata, essendo una stringa
- non c’è un elenco esplicito delle schermate disponibili

Usando gli enum

Per ovviare a questi problemi, si può usare un `enum`

```
enum Screen { Menu, Settings, Game }

public class App extends JFrame {
  App() {
    super("JGioco"); // ...
    JPanel panel;

    add(panel = new JPanel(new CardLayout()) {
      {
        add(new MenuPanel(), Screen.Menu.name());
        add(new SettingsPanel(), Screen.Settings.name());
        add(new GamePanel(), Screen.Game.name());
      }
    });

    ((CardLayout) panel.getLayout()).show(panel, Screen.Game.name());
  }

  public static void main(String[] args) { new App(); }
}
```

Il problema è che per poter **cambiare schermata**, bisogna passare ai vari componenti l’istanza di `App` di cui vogliamo cambiare la schermata, creando un groviglio di **spaghetti code**. Ma c’è una soluzione per ovviare anche a questo problema.

Singleton + Observer

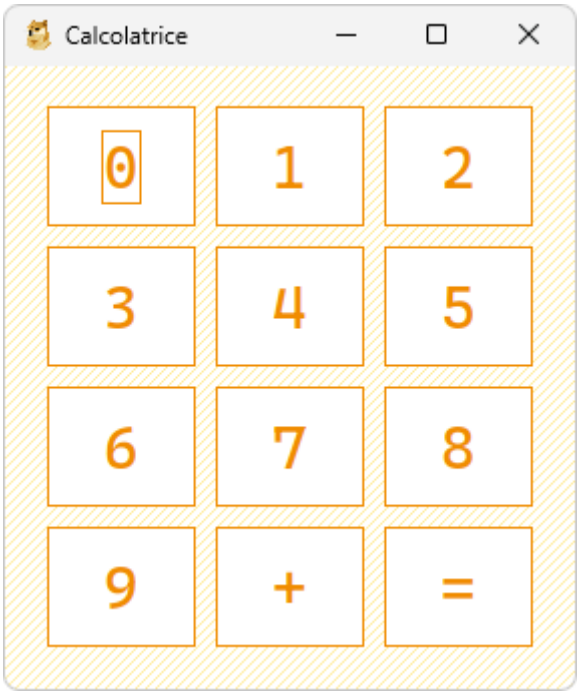
```
enum Screen { Menu, Settings, Game }

@SuppressWarnings("deprecation")
class Navigator extends Observable {
  private static Navigator instance;

  private Navigator() { }

  public static Navigator getInstance() {
    if (instance == null)
      instance = new Navigator();
    return instance;
  }

  public void navigate(Screen screen) {
    setChanged();
    notifyObservers(screen);
  }
}

@SuppressWarnings("deprecation")
public class App extends JFrame implements Observer {
  JPanel panel;

  App() {
    // ...

    Navigator.getInstance().addObserver(this);

    add(panel = new JPanel(new CardLayout()) {
      {
        add(new MenuPanel(), Screen.Menu.name());
        add(new SettingsPanel(), Screen.Settings.name());
        add(new GamePanel(), Screen.Game.name());
      }
    });

    // ...
  }

  @Override
  public void update(Observable o, Object arg) {
    if (o instanceof Navigator && arg instanceof Screen)
      ((CardLayout) panel.getLayout()).show(panel, ((Screen) arg).name());
  }

  public static void main(String[] args) {
    new App();
    Navigator.getInstance().navigate(Screen.Settings);
  }
}
```

- `Navigator` è la classe che permette di cambiare schermata
  - usa il pattern **Singleton** perché deve avere una sola istanza globale
  - usa il pattern **Observer** per notificare gli `Observer` dei cambiamenti di schermata
  - per cambiare schermata, *da qualsiasi parte del codice*, basta usare `Navigator.getInstance().navigate(Screen.Schermata);`
- `App`
  - è un **Observer** per poter essere notificato tramite `update(Observable o, Object arg)` dei cambiamenti di schermata
  - usa `Navigator.getInstance().addObserver(this);` per osservare l'unica istanza di `Navigator`

GridLayout (doc)

Non è un layout particolarmente complesso: permette di specificare il numero di righe, il numero di colonne, e lo spazio fra due componenti.

```
frame.add(new JPanel(new GridLayout(4, 3, 10, 10)) {
  {
    setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));

    for (int digit = 0; digit <= 9; digit++)
      add(new JButton(String.valueOf(digit)));
    add(new JButton("+"));
    add(new JButton("="));
  }
});
```



Implementazione

```
public class App extends JFrame {

  static {
    Color YELLOW = Color.decode("#f08c00");

    UIManager.put("Button.font", new Font("Cascadia Code", Font.PLAIN, 30));

    UIManager.put("Button.foreground", YELLOW);
    UIManager.put("Button.background", Color.WHITE);

    UIManager.put("Button.border", BorderFactory.createCompoundBorder(
        BorderFactory.createLineBorder(YELLOW),
        BorderFactory.createEmptyBorder(10, 15, 10, 15)));

    UIManager.put("Button.highlight", Color.decode("#ffec99"));
    UIManager.put("Button.select", Color.decode("#ffec99"));
    UIManager.put("Button.focus", YELLOW);

    UIManager.put("Panel.background", Color.WHITE);
  }
```

```
  App() {
    super("Calcolatrice");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    try {
      setIconImage(ImageIO.read(new File("icon.png")));
    } catch (IOException e) {
    }

    add(new JPanel(new GridLayout(4, 3, 10, 10)) {
      {
        setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
        for (int digit = 0; digit <= 9; digit++)
          add(new JButton(String.valueOf(digit)));
        add(new JButton("+"));
        add(new JButton("="));
      }

      @Override
      protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        int density = 5;
        g.setColor(Color.decode("#ffec99"));
        for (int x = 0; x <= getWidth() + getHeight(); x += density)
          g.drawLine(x, 0, 0, x);
      }
    });

    setSize(300, 350);
    setLocationRelativeTo(null);
    setVisible(true);
  }

  public static void main(String[] args) { new App(); }
}
```

GridBagLayout (doc)

Il layout più flessibile

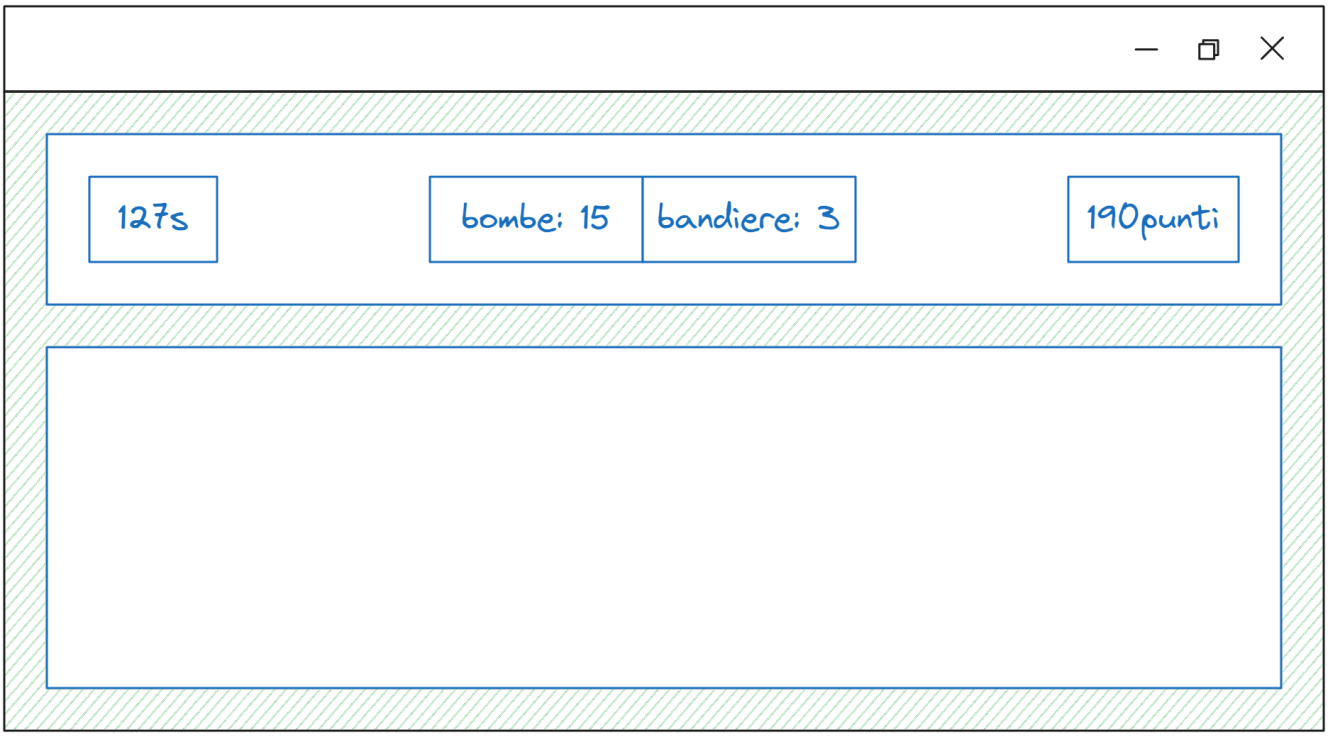


Figura 4: esempio di wireframe per il gioco “Minesweeper”

MVC

Minesweeper *(prato fiorito)*

Model

Controller

View

git

Lavorare in gruppo

Merge conflict

GitHub Actions

Generare la documentazione in automatico

Generare l’eseguibile in automatico