

Un approccio pratico a Java Swing e MVC

Sommario

Java Swing	3
Wireframe	3
Implementazione passo passo di un wireframe	4
Creare una finestra con JFrame (doc)	4
Errori comuni	4
Aggiungere contenuti a una finestra con JPanel (doc)	5
Colori, font e personalizzare lo sfondo con <code>paintComponent(Graphics g)</code>	5
Centrare un elemento con GridBagLayout (doc)	6
Implementazione	7
Interfacce con Swing senza variabili d'appoggio	8
Modificare l'aspetto dell'interfaccia con UIManager (doc)	9
Layout Manager	10
BorderLayout (doc)	10
Barra delle statistiche e menu di gioco	11
Implementazione	12
Funzionamento del <code>BorderLayout</code>	14
CardLayout (doc)	15
Menu, impostazioni e partita (<i>cambiare schermata con Singleton e Observer</i>) ..	15
L'implementazione più rozza	16
Usando gli enum	16
Singleton + Observer	17
GridLayout (doc)	18
Implementazione	18
GridBagLayout (doc)	20
Il layout più flessibile	20
Implementazione	21
In conclusione (sui Layout)	23
Pulsanti	24
Immagini	24
Animazioni	24
Graphics	24
Timer	24
MVC	25
Minesweeper (<i>prato fiorito</i>)	26
UML e modello	26
In breve	26
Vincoli e regole del modello	27
Use Case e operazioni sul modello	28
Wireframe della vista	28
Codice	29
Model	29
Encapsulation con <code>public final</code>	30
Tipizzazione con <code>enum</code>	30
Tipi <code>null</code> con <code>Optional<T></code>	30
<code>switch</code> sotto steroidi (<i>switch expression</i>)	30
Observer Pattern	30
Gestire il timer	31
Alcuni esempi di <code>Stream</code>	31
Controller	33

View	33
git	34
Lavorare in gruppo	34
Merge conflict	34
GitHub Actions	34
Generare la documentazione in automatico	34
Generare l'eseguibile in automatico	34

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 *solo 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)`.

i Nota

`java.awt.Container` è un esempio di [Composite Pattern](#) (doc)

- `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

```
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)`

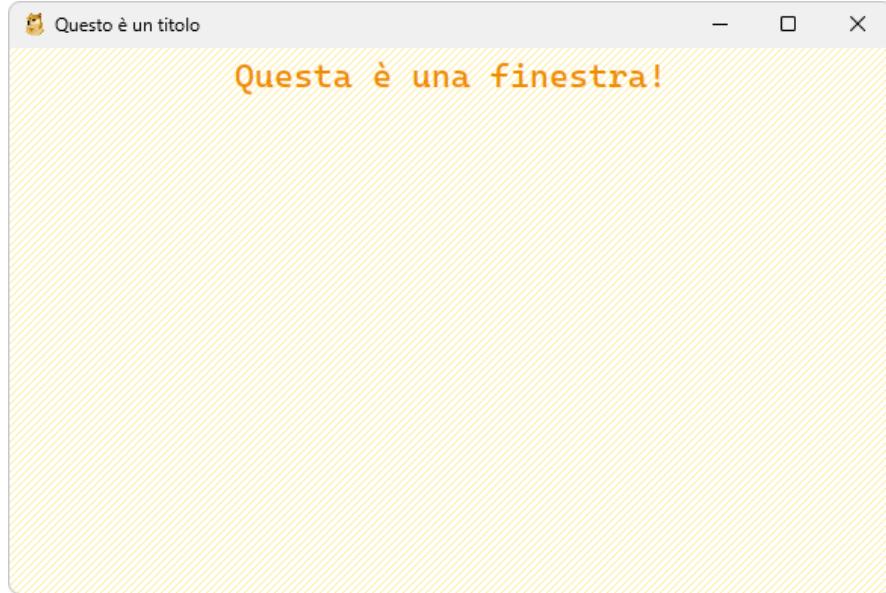
Creando una **classe anonima** possiamo sovrascrivere il comportamento di uno di questi metodi. Nel nostro esempio, sovrascriviamo `paintComponent(Graphics g)` per disegnare lo sfondo con le linee oblique in Figura 1.

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());
```

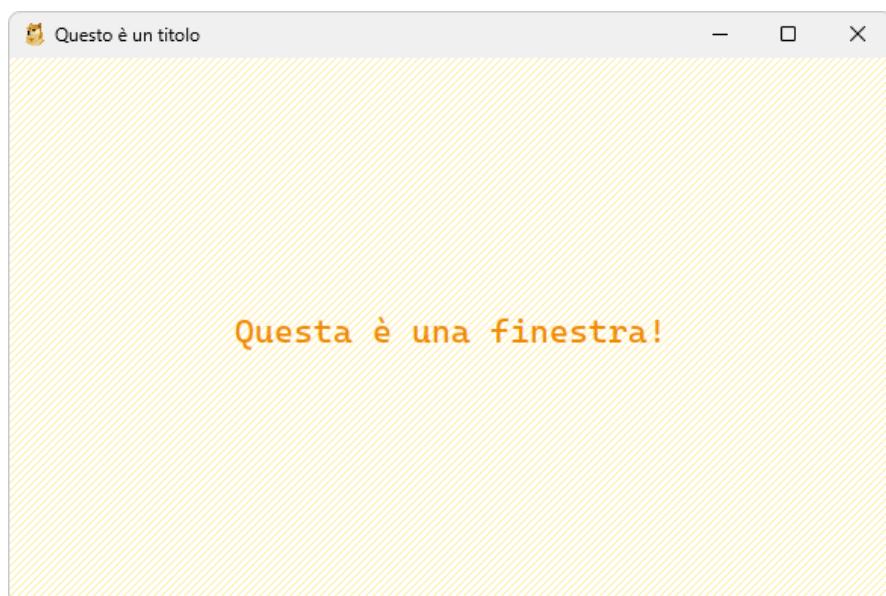
Il costruttore `JPanel(LayoutManager layout)` permette di specificare una **strategia** per **posizionare** e **dimensionare** il contenuto di un `JPanel` in **automatico**:

- 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

```
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);
```

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

```
public class App extends JFrame {
    static { UIManager.put("Panel.background", Color.WHITE); }

    App() {
        super("JGioco");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        try { setIconImage(ImageIO.read(new File("icon.png"))); } catch (IOException e) { }

        add(new JPanel(new BorderLayout(10, 10)) { // Sfondo
        {
            setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20)); // Spazio dal bordo della finestra

            add(new JPanel() { // Barra in alto
            {
                setBackground(new Color(255, 255, 255, 0)); // Trasparente
                setBorder(BorderFactory.createCompoundBorder(
                    BorderFactory.createLineBorder(Color.decode("#2f9e44")),
                    BorderFactory.createEmptyBorder(10, 10, 10, 10)));
            }
            }, BorderLayout.NORTH); // Posizionata a NORD

            add(new JPanel() { // Blocco centrale
            {
                setBackground(new Color(255, 255, 255, 0)); // Trasparente
                setBorder(BorderFactory.createCompoundBorder(
                    BorderFactory.createLineBorder(Color.decode("#2f9e44")),
                    BorderFactory.createEmptyBorder(10, 10, 10, 10)));
            }
            }, BorderLayout.CENTER); // Posizionato al CENTRO
        }

        @Override
        protected void paintComponent(Graphics g) { // Sfondo linee oblique verdi
            super.paintComponent(g);
            int density = 5;
            g.setColor(Color.decode("#b2f2bb"));
            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(); }
```

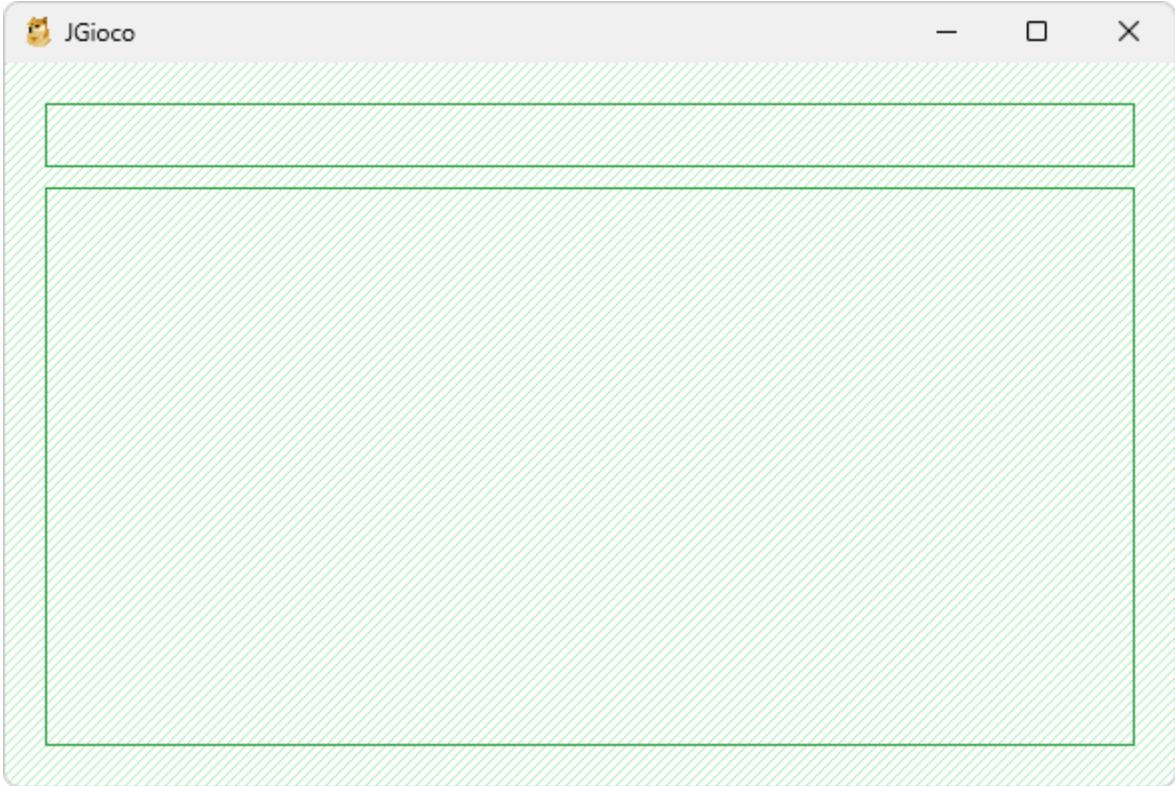
- `BorderLayout(int vgap, int hgap)` imposta uno “spazio” verticale e orizzontale fra due componenti.
- con `setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20))` impostiamo un bordo trasparente (per lasciare uno spazio dal bordo della finestra)

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](#))

Risultato



Implementazione

i Nota

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

```
public class App extends JFrame {  
  
    static {  
        UIManager.put("Label.font", new Font("Cascadia Code", Font.PLAIN, 17));  
        UIManager.put("Label.foreground", Color.decode("#2f9e44"));  
        UIManager.put("Label.background", Color.WHITE);  
  
        UIManager.put("Button.font", new Font("Cascadia Code", Font.PLAIN, 17));  
        UIManager.put("Button.foreground", Color.decode("#2f9e44"));  
        UIManager.put("Button.background", Color.WHITE);  
        UIManager.put("Button.border", BorderFactory.createCompoundBorder(  
            BorderFactory.createLineBorder(Color.decode("#2f9e44")),  
            BorderFactory.createEmptyBorder(5, 10, 5, 10)));  
        UIManager.put("Button.highlight", Color.decode("#b2f2bb"));  
        UIManager.put("Button.select", Color.decode("#b2f2bb"));  
        UIManager.put("Button.focus", Color.WHITE);  
  
        UIManager.put("Panel.background", Color.WHITE);  
    }  
}
```

```

App() {
    add(new JPanel(new BorderLayout(10, 10)) {
        {
            // ...
        }

        add(new JPanel() {
            {
                // ...

                String[] labels = { "Pincopallino", "partite: 10", "vittorie: 2" };
                for (String label : labels)
                    add(new JLabel(label) {
                        {
                            setOpaque(true);
                            setBorder(BorderFactory.createCompoundBorder(
                                BorderFactory.createLineBorder(Color.decode("#2f9e44")),
                                BorderFactory.createEmptyBorder(5, 10, 5, 10)));
                        }
                    });
            }
        }, BorderLayout.NORTH);

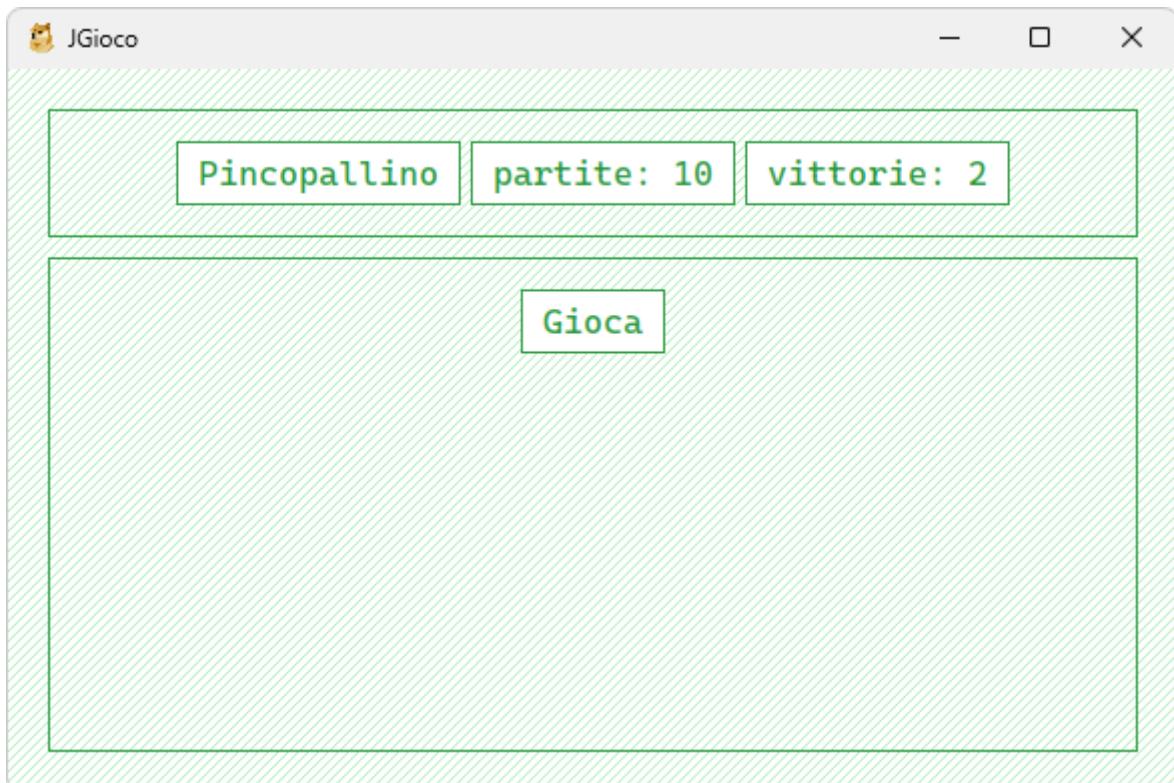
        add(new JPanel() { { ...; add(new JButton("Gioca")); } }, BorderLayout.CENTER);
    }

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

// ...
}

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

```

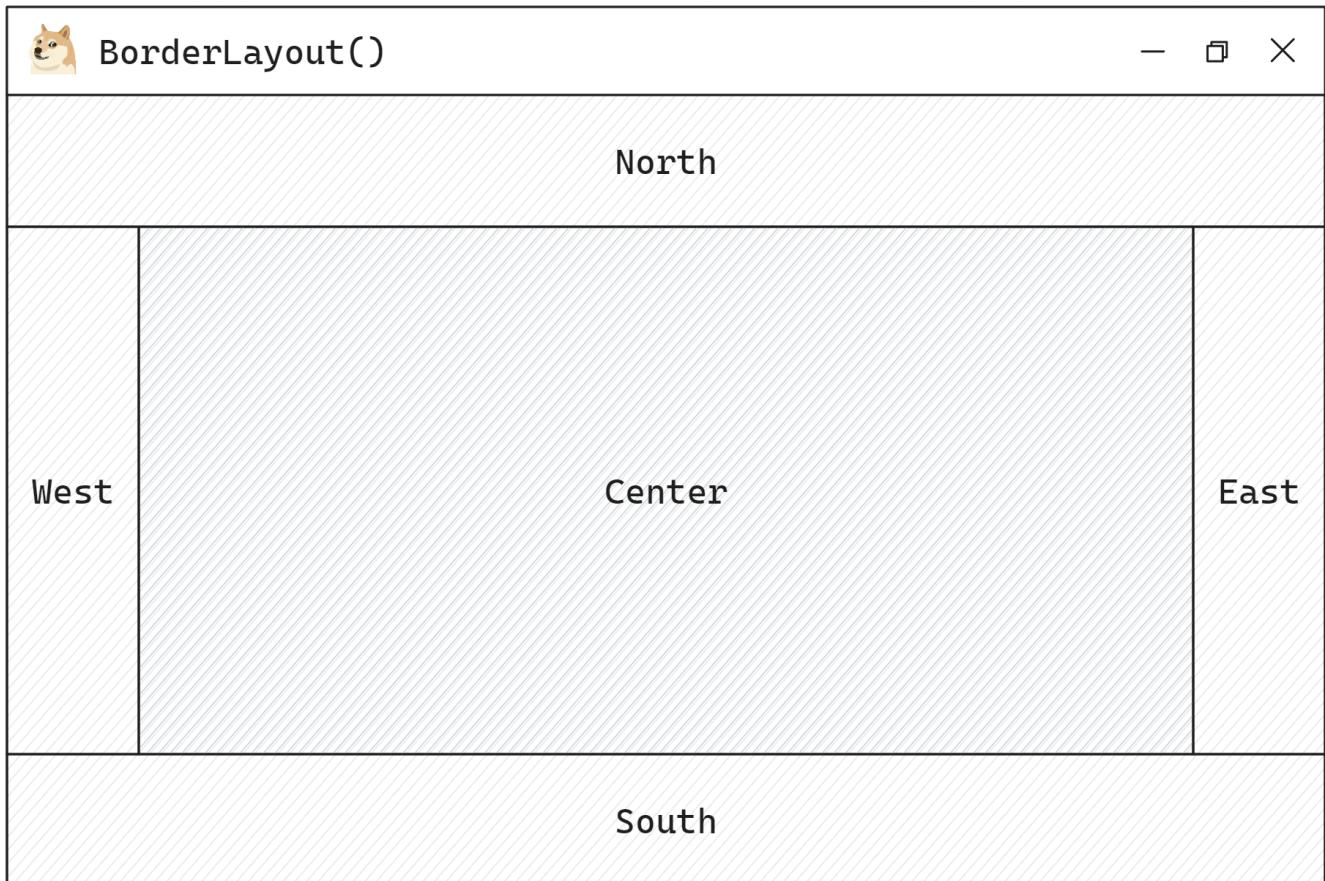


Funzionamento del BorderLayout

Il `BorderLayout` permette di specificare come posizionare gli elementi:

- l'elemento `CENTER` occuperà tutto lo spazio possibile
- gli elementi `NORTH` e `SOUTH` avranno larghezza massima (indipendentemente dalla larghezza impostata) e avranno altezza minima, o, se impostata, l'altezza impostata
- gli elementi `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.



```
public class App extends JFrame {  
    App() {  
        //...  
  
        add(new JPanel(new BorderLayout())) {  
            {  
                add(new DecoratedPanel("Nord"), BorderLayout.NORTH);  
                add(new DecoratedPanel("Sud"), BorderLayout.SOUTH);  
                add(new DecoratedPanel("West") {  
                    {  
                        setPreferredSize(new Dimension(120, 0)); // Larghezza custom  
                    }  
                }, BorderLayout.WEST);  
                add(new DecoratedPanel("East"), BorderLayout.EAST);  
                add(new DecoratedPanel("Center"), BorderLayout.CENTER);  
            }  
        );  
    }  
  
    public static void main(String[] args) { new App(); }  
}
```

[CardLayout](#) (doc)

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

Il [CardLayout](#) è molto utile quando abbiamo più schermate (menu principale, impostazioni, partita etc...)

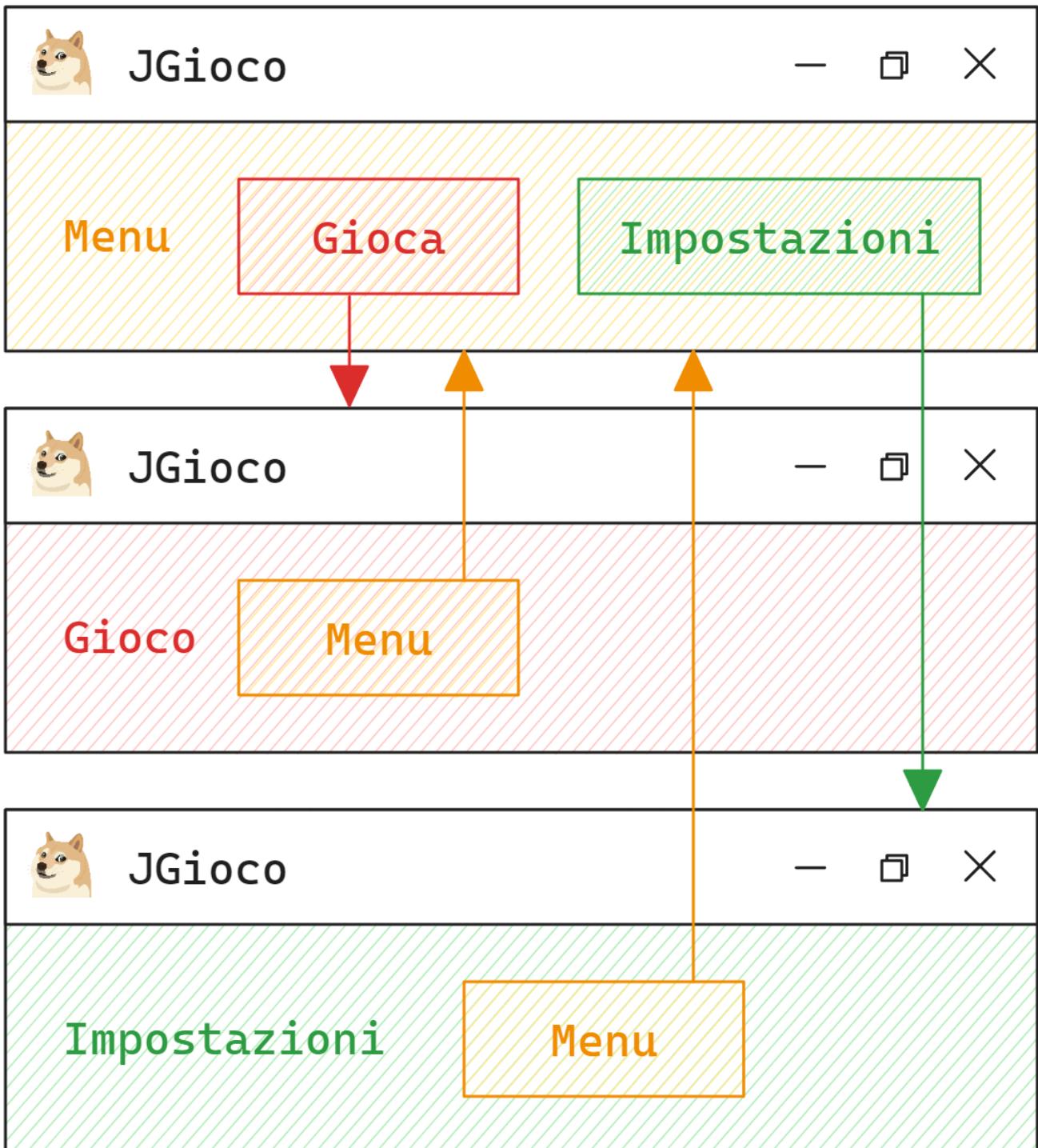


Figura 3: caso d'uso di un [CardLayout](#)

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());
}

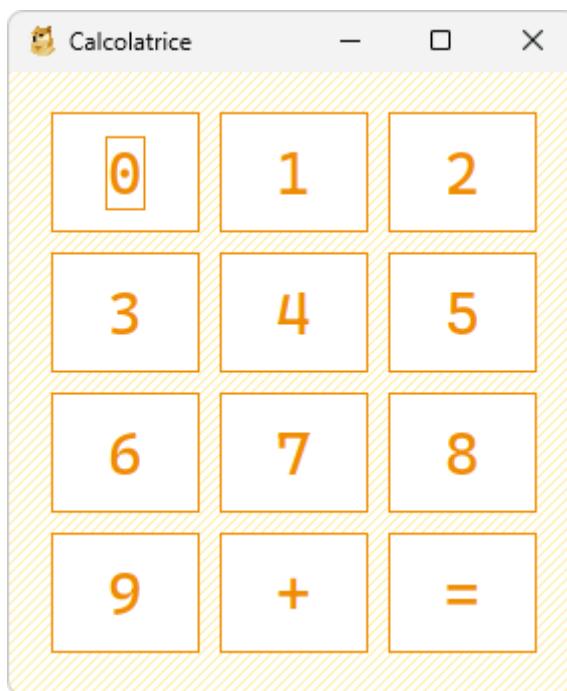
public static void main(String[] args) {
    new App();
    Navigator.getInstance().navigate(Screen.Settings);
}
}
```

- `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(); }
}
```

[GridLayout](#) (doc)

Il layout più flessibile

Il `GridLayout` permette di dividere il panel in una griglia (nell'esempio, le celle della griglia sono 3, e sono divise dalla linea gialla) in maniera molto flessibile:

- una cella può essere **posizionata** in qualunque x e y della griglia
- la **dimensione** di una cella può essere definita in diversi modi
 - numero di righe e numero di colonne
 - può essere specificato che deve occupare “tutto lo **spazio rimanente**”
 - in larghezza
 - in altezza
 - in entrambe le dimensioni

La particolarità (rispetto agli altri layout) è che gli elementi **non vengono ridimensionati** per occupare tutto lo spazio di una cella, ma vengono posizionati all'interno della cella rispettando le loro dimensioni (nella cella a sinistra, si può vedere che il label "testo" è posizionato a `NORTH_EAST`)

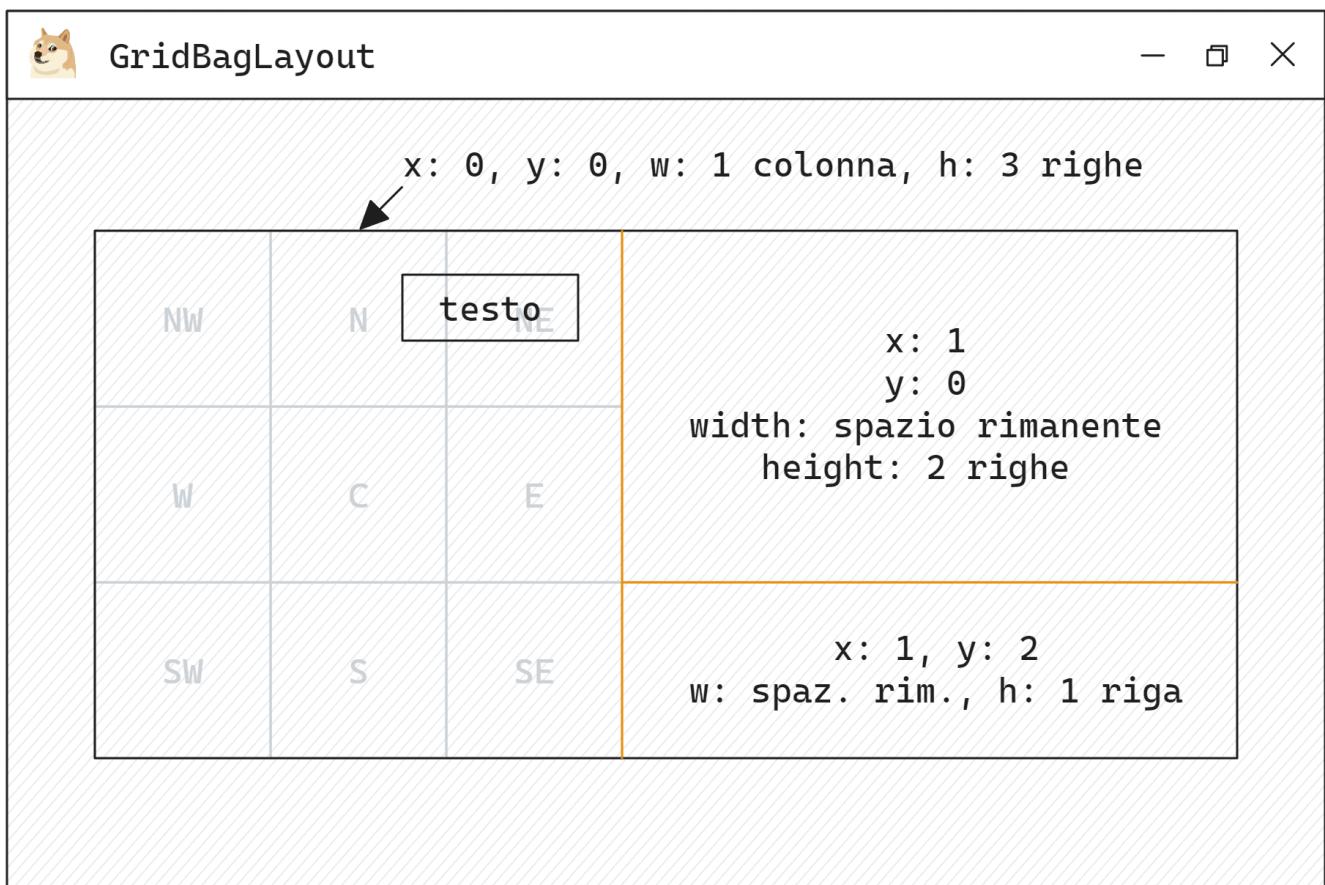


Figura 4: caso d'uso di un `GridLayout`

Implementazione

```
public class App extends JFrame {
    static Color TRANSPARENT = new Color(0, 0, 0, 0);

    static {
        UIManager.put("Label.font", new Font("Cascadia Code", Font.PLAIN, 14));
        UIManager.put("Panel.background", Color.WHITE);
    }

    App() {
        super("JGioco");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

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

        JFrame frame = this;
        setSize(600, 400);

        add(new JPanel(new GridLayout(1, 1)) {
        {
            setBorder(BorderFactory.createEmptyBorder(50, 30, 50, 30));

            add(new JPanel(new GridBagLayout()) {
            {
                setBackground(TRANSPARENT);

                setBorder(BorderFactory.createCompoundBorder(BorderFactory.createLineBorder(Color.DARK_GRAY),
                    BorderFactory.createEmptyBorder(30, 30, 30, 30)));

                GridBagConstraints constraints = new GridBagConstraints();

                constraints.weightx = 1;
                constraints.weighty = 1;

                constraints.anchor = GridBagConstraints.NORTHEAST;
                constraints.gridx = 0;
                constraints.gridy = 0;
                constraints.gridwidth = 1;
                constraints.gridheight = 3;

                add(new JLabel("testo"), constraints);

                constraints.anchor = GridBagConstraints.CENTER;
                constraints.gridx = 1;
                constraints.gridy = 0;
                constraints.gridwidth = GridBagConstraints.REMAINDER;
                constraints.gridheight = 2;

                add(new JLabel("x: 1,\n y: 0, w: spaz. rim., h: 2 righe"), constraints);

                constraints.gridx = 1;
                constraints.gridy = 2;
                constraints.gridwidth = GridBagConstraints.REMAINDER;
                constraints.gridheight = 1;

                add(new JLabel("x: 1, y: 0, w: spaz. rim., h: 1 riga"), constraints);
            }
        }
    }

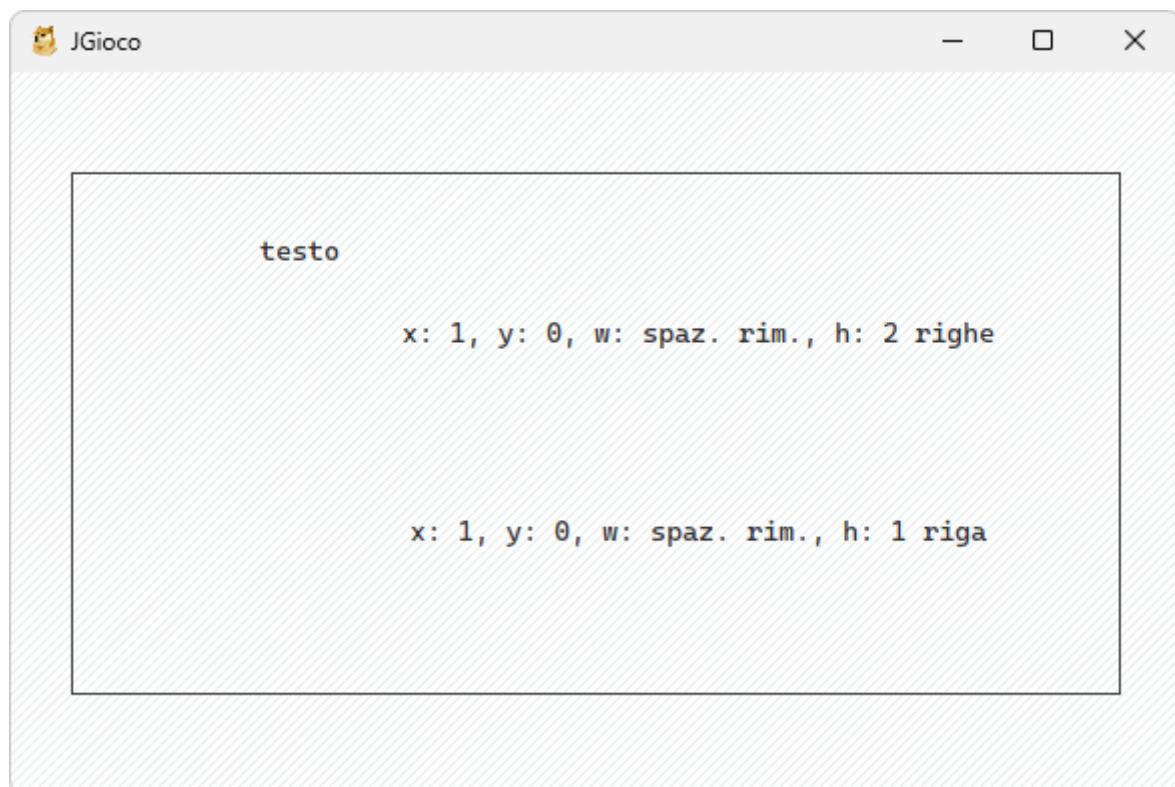
    @Override
    protected void paintComponent(Graphics g) {
        setPreferredSize(new Dimension(frame.getWidth() - 80, frame.getHeight() - 80));
    }
}
```

```
        super.paintComponent(g);
    }
});

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    int density = 5;
    g.setColor(Color.decode("#e9ecef"));
    for (int x = 0; x <= getWidth() + getHeight(); x += density)
        g.drawLine(x, 0, 0, x);
}
});

setLocationRelativeTo(null);
setVisible(true);
}

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



In conclusione (sui Layout)

Dato un **wireframe** ora dovreste avere gli strumenti per implementare il layout

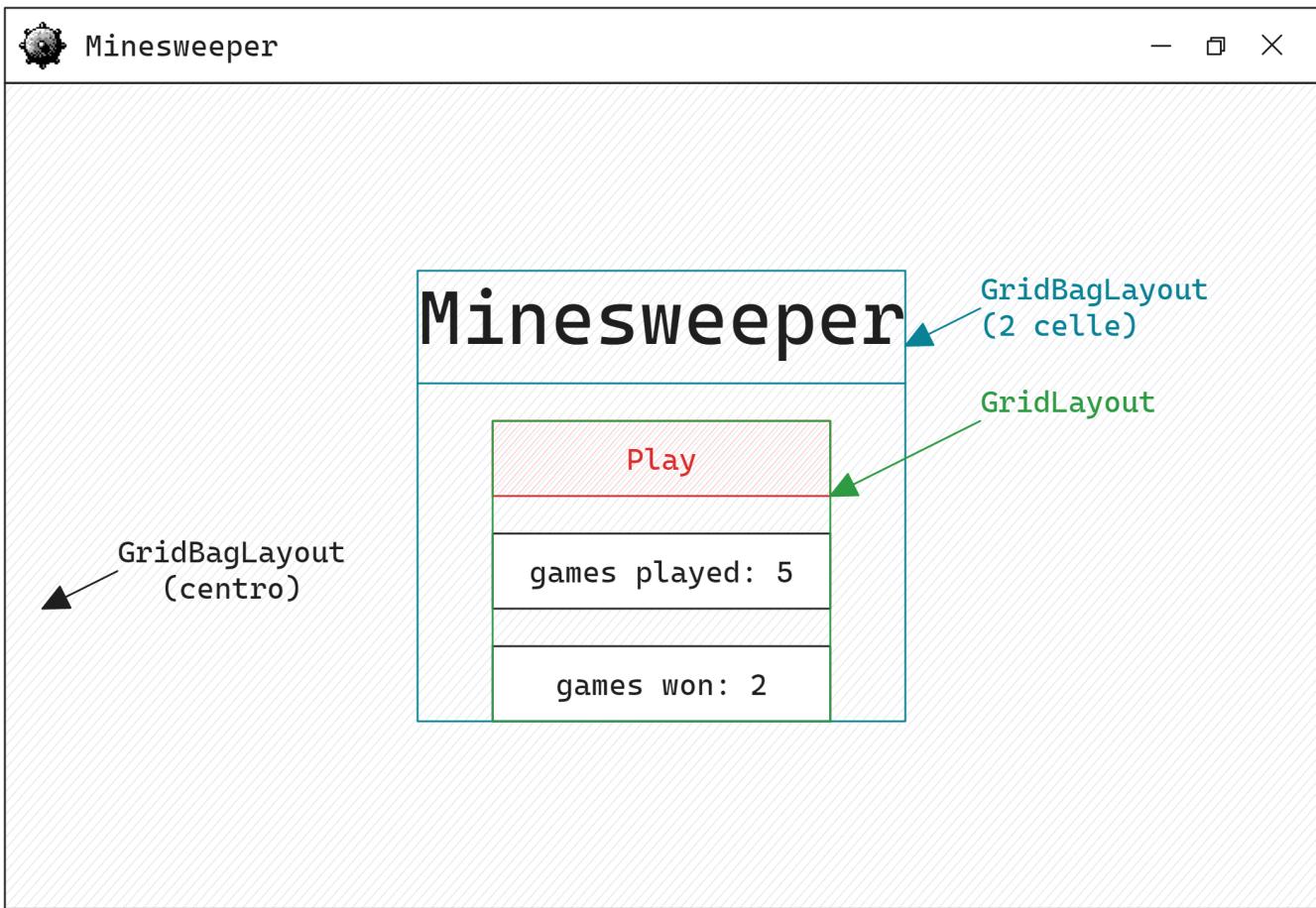


Figura 5: esempio di wireframe per il gioco “Minesweeper”

I layout visti in questa guida [non sono gli unici \(doc\)](#) (e non sono state coperte tutte le loro funzionalità), ma, capiti i concetti chiave, si può iniziare ad usare la [documentazione \(doc\)](#).

Pulsanti

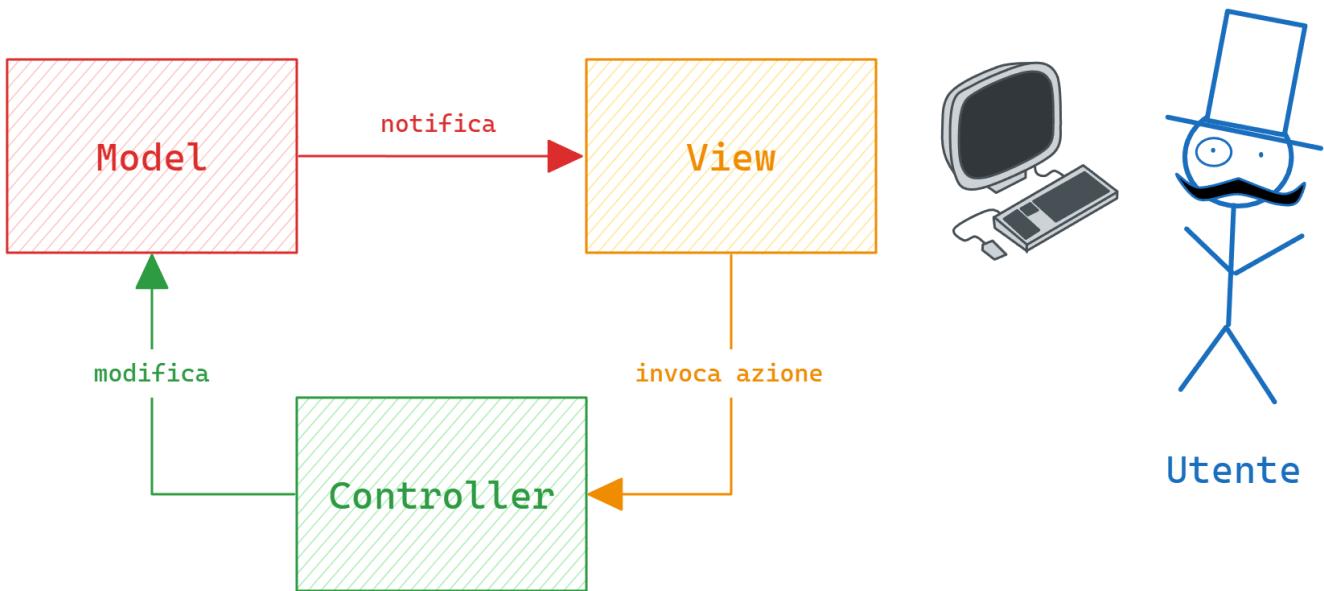
Immagini

Animazioni

Graphics

Timer

MVC



MVC è un **pattern architettonale** per sviluppare diversi tipi di applicazioni: servizi web, programmi GUI, programmi per terminale etc...

L'idea è quella di **separare la logica del programma dall'interfaccia**. In questo modo:

- la logica e i dati sono **definiti in modo chiaro e pulito**
- la logica è **riusabile** (più interfacce diverse possono condividere la logica)

Ad esempio, posso definire la logica del gioco **Solitario** una volta sola, e usarla per costruire un sito web, un'applicazione per Windows, un'API REST, una TUI (terminal user interface) etc... Il **Model** è condiviso da più **View**.

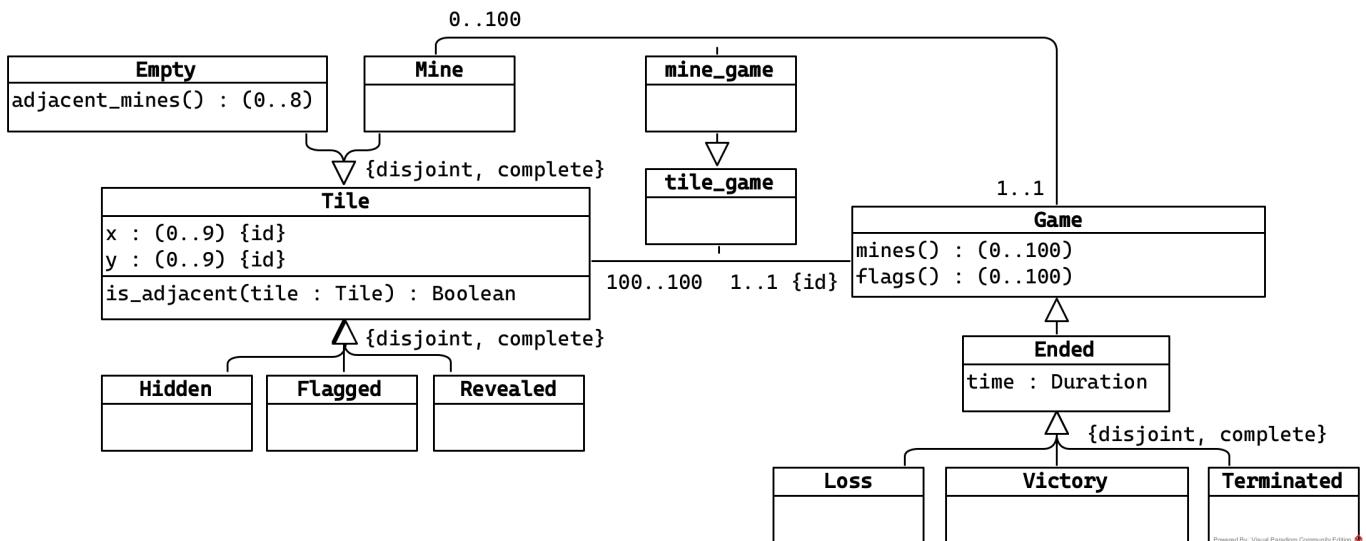
Fin'ora abbiamo discusso solo della **View**, ora l'obiettivo è quello di **progettare una semplice applicazione** e mostrare un possibile modo di strutturare il codice.

Minesweeper (*prato fiorito*)

Trovate il codice completo del progetto [su GitHub \(doc\)](#)

UML e modello

Iniziamo definendo il modello



i Nota

Attenzione! Questo NON è il tipo di UML che avete visto a lezione.

È definito tramite la [logica di primo ordine \(doc\)](#) ed è agnostico. Si vede in corsi come **Basi di dati 2** e **Ingegneria del software**. Lo uso per brevità e semplicità.

In breve

- vogliamo gestire le partite (Game)
 - ogni partita ha 100 caselle (Tile)
 - sono disposte in una griglia 10x10
 - c'è un numero variabile di mine da 0 a 100
 - una partita può essere finita (Ended)
 - caso in cui va segnata la durata
 - può avere uno di tre esiti
 - sconfitta (Loss)
 - vittoria (Victory)
 - terminata dall'utente (Terminated)
 - di ogni partita si devono poter calcolare
 - il numero di mine
 - il numero di bandiere
- ogni casella (Tile)
 - può essere di uno dei due tipi
 - vuota (Empty)
 - con una mina (Mine)
 - può trovarsi in uno dei tre stati
 - nascosta (Hidden)
 - con una bandiera (Flagged)
 - scoperta (Revealed)
 - di ogni casella si devono poter calcolare le caselle adiacenti
 - di ogni casella vuota si deve conoscere il numero di mine vicine

Vincoli e regole del modello

Il modello che abbiamo definito è **incompleto**: permette **alcuni stati che non vogliamo**

- una partita potrebbe essere una vittoria anche con una mina scoperta
- una partita potrebbe essere una vittoria con tutte le caselle nascoste

Ci servono dei vincoli (qui scritti in *logica di primo ordine*) per definire le **regole**.

i Nota

Di seguito sono riportati alcuni vincoli, [qui c'è il documento completo \(doc\)](#)

Una partita è una vittoria in uno di due casi

- tutte le mine hanno una bandiera e tutte le caselle vuote sono senza bandiera
- tutte le caselle vuote sono scoperte

[V.Game.victory_condition]

```
∀ game
    Victory(game) ⇔
        ∀ tile mine_game(tile, game) ⇒ Flagged(tile) ∧
        ¬ ∃ tile tile_game(tile, game) ∧ Empty(tile) ∧ Flagged(tile)
        ∨
        ∀ tile (tile_game(tile, game) ∧ Empty(tile)) ⇒ Revealed(tile))
```

Una partita è una sconfitta se e solo se c'è una mina scoperta

[V.Game.loss_condition]

```
∀ game
    Loss(game) ⇔
        ∃ mine mine_game(mine, game) ∧ Revealed(mine)
```

i Nota

Vedremo che in Java questi vincoli sono comodi da scrivere con gli [Stream](#)

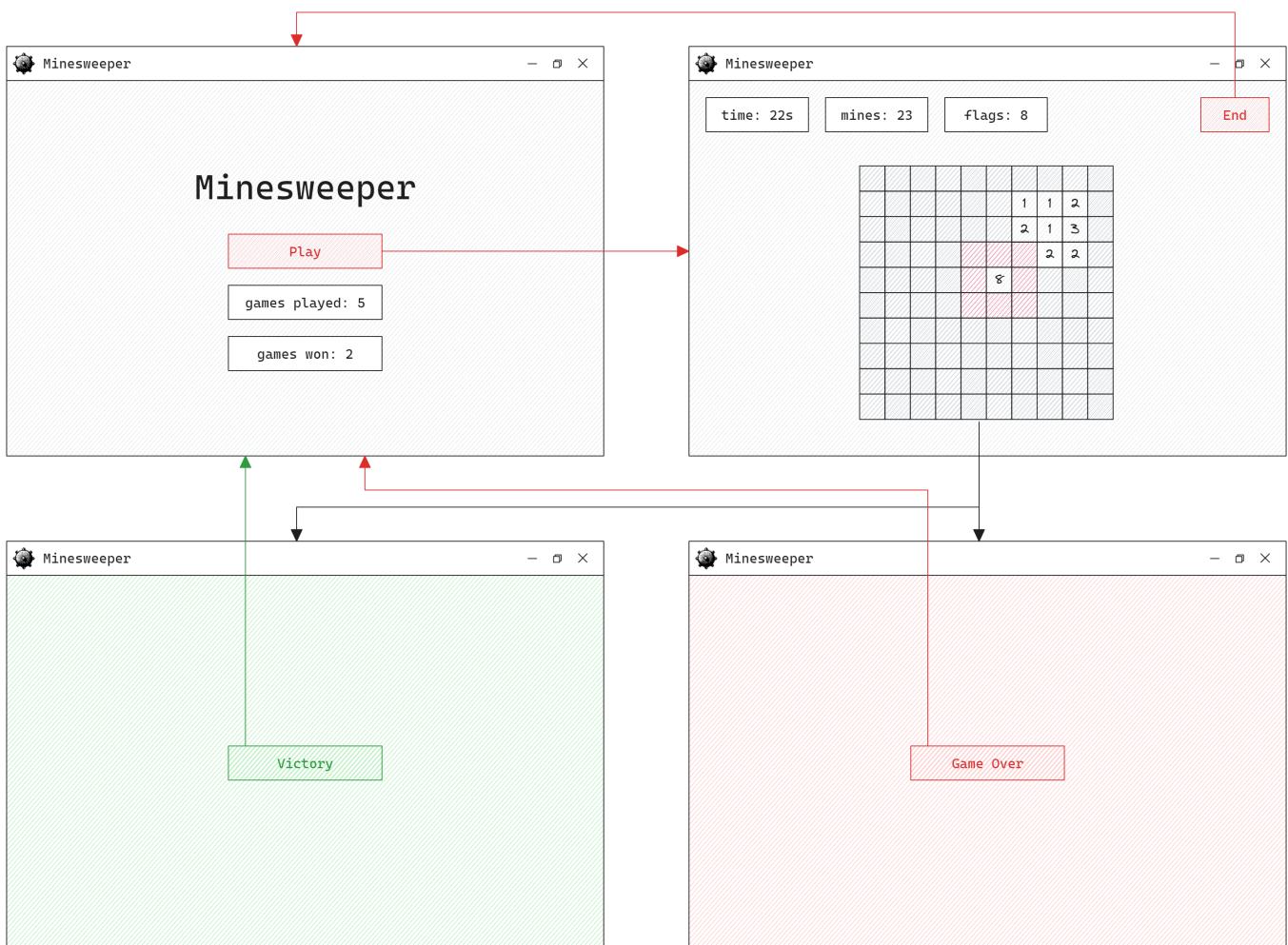
Use Case e operazioni sul modello

Gli use case sono un insieme di **operazioni** che un **utente** deve poter effettuare sul **model**. Sono le funzionalità che dobbiamo garantire nella progettazione del **wireframe** e che verranno tradotte in parte in operazioni del **Controller**.

```
start_game(): Game
terminate_game(game: Game): Terminated
reveal(tile: Hidden): Revealed
flag(tile: Hidden): Flagged
remove_flag(tile: Flagged): Hidden
games_played(): Integer ≥ 0
games_won(): Integer ≥ 0
time(): Duration
mines(): (0..100)
flags(): (0..100)
```

Wireframe della vista

Ora dobbiamo solo progettare la vista dell'applicazione con un **wireframe**



Codice

Nel [codice \(doc\)](#) del progetto ho deciso di creare un **package** principale chiamato **minesweeper** e di suddividerlo a sua volta in 3 **package** per **model**, **view** e **controller**. Il progetto è piccolo, quindi bastano questi tre. I **package** sono lo strumento più importante per l'**encapsulation**.

Model

Vediamo alcune particolarità degne di nota.

```
package minesweeper.model;

import static minesweeper.model.Tile.Visibility.Flagged;
import static minesweeper.model.Tile.Visibility.Hidden;
import static minesweeper.model.Tile.Visibility.Revealed;

import java.util.Observable;
import java.util.Optional;

@SuppressWarnings("deprecation")
public class Tile extends Observable {

    public enum Kind {
        Mine,
        Empty
    }

    public enum Visibility {
        Hidden,
        Flagged,
        Revealed
    }

    public final int x, y;
    public final Kind kind;
    private Visibility visibility = Visibility.Hidden;
    Optional<Integer> adjacentMines = Optional.empty();

    Tile(int x, int y, Kind kind) {
        this.x = x;
        this.y = y;
        this.kind = kind;
    }

    public Visibility visibility() { return visibility; }

    public Optional<Integer> adjacentMines() { return adjacentMines; }

    public void flag() {
        visibility = switch (visibility) {
            case Hidden -> {
                setChanged();
                yield Flagged;
            }
            case Flagged -> {
                setChanged();
                yield Hidden;
            }
            case Revealed -> Revealed;
        };
        notifyObservers(visibility);
    }
}
```

```

public void reveal() {
    if (visibility != Hidden)
        return;

    visibility = Revealed;

    setChanged();
    notifyObservers(Revealed);
}

}

```

Encapsulation con `public final`

La strategia di ridurre la visibilità di un attributo a `private` e di definire `getter` e `setter` non è l'unico modo di fare encapsulation.

Un'altra strategia è quella di segnare un attributo come `public final`, e di rendere il costruttore visibile solo all'interno del package (visibilità di default)

In questo modo, le `Tile` possono essere istanziate con valori per `x`, `y` e `kind` all'interno del package (quindi si spera in modo corretto), e questi valori possono essere letti (ma non modificati) all'esterno.

Tipizzazione con `enum`

Ho deciso di codificare i tipi di caselle e i possibili stati con degli `enum`: usare l'ereditarietà in questo caso sarebbe stato inutilmente complesso e verboso.

Gli `enum` sono interni alla classe per poter usare la notazione `Tile.Kind` e `Tile.Visibility` che trovo più leggibile (e rende il codice più semplice da navigare non dovendo creare altri due file).

Tipi `null` con `Optional<T>`

Nel modello che abbiamo definito solo le caselle `Empty` hanno il numero di mine adiacenti. Per poterlo fare in Java possiamo aggiungere un attributo `Integer adjacentMines` a tutte le caselle, e impostarlo a `null` per le caselle non `Empty`.

Il problema di questo approccio è che chi usa la libreria deve sapere in qualche modo che quell'attributo potrebbe essere `null`.

Per risolvere il problema, basta rendere l'attributo `Optional<Integer>`, in questo modo stiamo esplicitamente dichiarando nel codice che quell'attributo potrebbe non avere un valore impostato (`Optional.empty()`), e chi usa l'attributo deve gestire il caso in cui il valore non c'è.

`switch` sotto steroidi (`switch expression`)

In Java 14 sono state ufficializzate le `switch` expression che possono restituire un valore (vedere metodo `flag()`).

Hanno anche altre funzionalità (permettono di determinare il tipo di un oggetto e castarlo senza usare `instanceof`, vedere `update(Observable o, Object arg)` in `model.Game`)

Observer Pattern

Per la casella ho deciso di usare l'`Observer` per due motivi:

- notificare la `View` (per ridisegnare la casella)
- notificare le altre classi del model (la classe `Game` deve sapere quando una `Tile` cambia stato, per decidere se terminare la partita o scoprire le caselle adiacenti)

```

@SuppressWarnings("deprecation")
public class Game extends Observable implements Observer {

    // ...

    final Tile[] tiles = new Tile[100];
    public final int mines;
    int flags = 0;
    Duration time = Duration.ofSeconds(0);

    public Game() {
        Random random = new Random();

        for (int y = 0; y < 10; y++)
            for (int x = 0; x < 10; x++)
                tiles[y * 10 + x] = new Tile(x, y, random.nextInt(100) >= 15 ? Empty : Mine);

        for (Tile tile : tiles) {
            tile.addObserver(this);

            int adjacentMines = (int) adjacent(tile.x, tile.y)
                .filter(t -> t.kind == Mine)
                .count();

            if (tile.kind == Empty && adjacentMines > 0)
                tile.adjacentMines = Optional.of(adjacentMines);
        }
    }

    mines = (int) Stream.of(tiles)
        .filter(t -> t.kind == Mine)
        .count();
}

public void updateTime() {
    time = time.plusSeconds(1);
    setChanged();
    notifyObservers(Message.Timer);
}

// ...

```

Un errore che ho visto è quello di creare un attributo per ogni variabile. Ad esempio: alcuni hanno creato un attributo `Random random` per la classe `Game`, nonostante questo venga usato solo nel costruttore. **NON c'è bisogno di creare un attributo per ogni variabile**.

Gestire il timer

Il **Model** non dovrebbe gestire l'eventuale **timer**, quello è un compito del **Controller**. Mettendo il **timer** fuori dal **Model** diventa più facile mettere in pausa il gioco.

Il **Model** dovrebbe semplicemente avere un metodo `update()` (nel mio caso `updateTime()`) che deve essere invocato dal **Controller** quando scatta il **timer**.

Alcuni esempi di `Stream`

Nella classe `model.Game` ci sono alcuni esempi di `Stream` oltre a quelli sopra.

```
// ...
@Override
public void update(Observable o, Object arg) {
    switch (o) {
        case Tile tile -> {
            switch (arg) {
                case Tile.Visibility visibility -> {
                    switch (visibility) {
                        case Flagged -> flags++;
                        case Hidden -> flags--;
                        case Revealed -> {
                            if (tile.kind == Mine) {
                                setChanged();
                                notifyObservers(Result.Loss);
                                deleteObservers();
                                return;
                            }

                            if (tile.adjacentMines().isEmpty())
                                adjacent(tile.x, tile.y).forEach(Tile::reveal);

                            boolean allEmptyRevealed = Stream.of(tiles)
                                .allMatch(t -> t.visibility() == Revealed || t.kind == Mine);

                            boolean allMinesFlagged = Stream.of(tiles)
                                .allMatch(t -> (t.kind == Mine && t.visibility() == Flagged)
                                    || (t.kind == Empty && t.visibility() != Flagged));

                            if (allEmptyRevealed || allMinesFlagged) {
                                setChanged();
                                notifyObservers(Result.Victory);
                                deleteObservers();
                                return;
                            }
                        }
                    }
                }
            }
        default -> {}
    }
}
default -> {}
}
```

Controller

View

git

Lavorare in gruppo

Merge conflict

GitHub Actions

Generare la documentazione in automatico

Generare l'eseguibile in automatico