Programmazione Orientata agli Oggetti in Linguaggio Java

Programmazione Grafica: Organizzazione del Codice Parte b

versione 1.0

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

- Riorganizzazione del Codice
  - ⇨Distribuzione delle Responsabilità
  - ⇨Raffinamento
- MVC

Riorganizzazione del Codice

- Le linee guida viste
  - ⇨rappresentano un passo in avanti rispetto allo stile “Swing” ha ancora dei limiti
- Ma...
  - ⇨anche avendo separato la logica applicativa nello strato del modello, resta aperto il problema di organizzare il codice nello strato di interfaccia e controllo

Riorganizzazione del Codice

- Struttura tipica dello strato di interfaccia
  - ⇨contiene la maggior parte del codice dell’applicazione per via della “verbosità” di Swing
  - ⇨molte classi interne (le azioni)
  - ⇨se l’applicazione ha un unico “schermo” visibile (es: Morra Cinese), c’è un’unica classe Principale che gestisce lo schermo e contiene moltissimo codice

Riorganizzazione del Codice

- Di conseguenza
  - ⇨questo stile funziona solo per applicazioni con GUI molto semplici
  - ⇨in applicazioni grafiche di medie dimensioni, il codice dello strato di interfaccia e controllo diventa presto impossibile da manutenere
- Di conseguenza
  - ⇨è necessario adottare strumenti per la riorganizzazione del codice

Riorganizzazione del Codice

- Riorganizzazione del codice
  - ⇨vogliamo individuare criteri per distribuire le responsabilità nello strato di interfaccia e controllo
  - ⇨per renderlo più modulare e quindi più facilmente gestibile e manutenibile
- Per farlo
  - ⇨procediamo ad un refactoring dell'applicazione della Morra Cinese

Distribuzione delle Responsabilità

- Individuiamo le responsabilità evidenti
  - ⇨I responsabilità: inizializzare il JFrame e visualizzarlo (creare i menu, impostare i parametri relativi al frame)
  - ⇨II responsabilità: gestire i pannelli che corrispondono a visualizzare gli schermi (disporre ed aggiornare i componenti)
  - ⇨III responsabilità: gestire le azioni (crearle, abilitarle, disabilitarle, avviare l'applicazione)

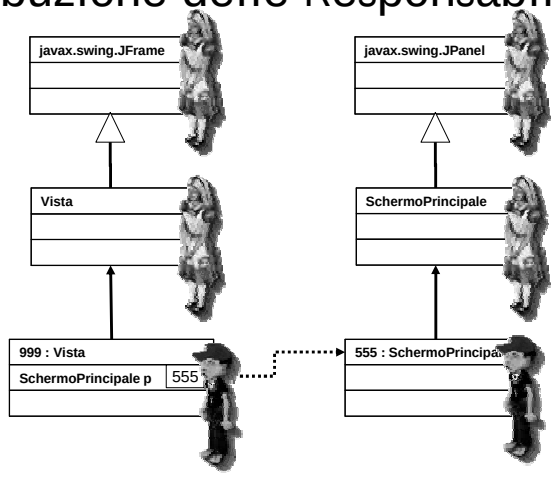
Distribuzione delle Responsabilità

- Questo suggerisce il seguente refactoring
  - ⇨riorganizziamo il codice della classe Principale in tre componenti separati
- Vista
  - ⇨ha la I responsabilità: inizializzare il JFrame e visualizzarlo
- SchermoPrincipale
  - ⇨ha la II responsabilità: gestire l'unico pannello necessario per l'unico schermo

Distribuzione delle Responsabilità

- Terminologia
  - ⇨vista = interfaccia; classi della vista = classi che gestiscono l'interfaccia
  - ⇨vista principale: la classe che gestisce il frame (JFrame)
  - ⇨viste secondarie: le classi che gestiscono i pannelli (JPanel) visualizzati sul frame ed eventuali finestre secondarie (JDialog)
  - ⇨la vista principale deve tenere riferimenti alle viste secondarie per poterle visualizzare

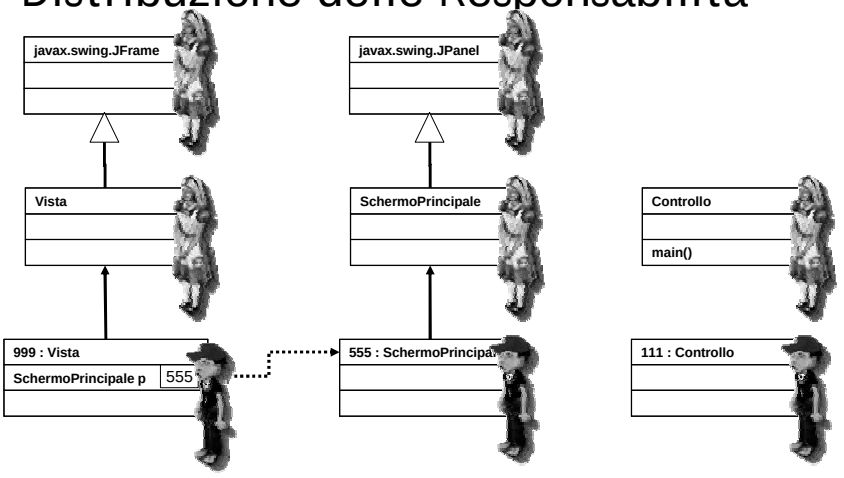
Distribuzione delle Responsabilità



Distribuzione delle Responsabilità

- Controllo
  - ha la terza responsabilità: gestisce le azioni
- Attenzione a questa scelta
  - si tratta di una scelta naturale perchè va nella direzione di separare il codice di interfaccia da quello di controllo
  - ma implica che le azioni non sono più classi interne delle classi di interfaccia
  - dovremo risolvere alcuni problemi di visibilità

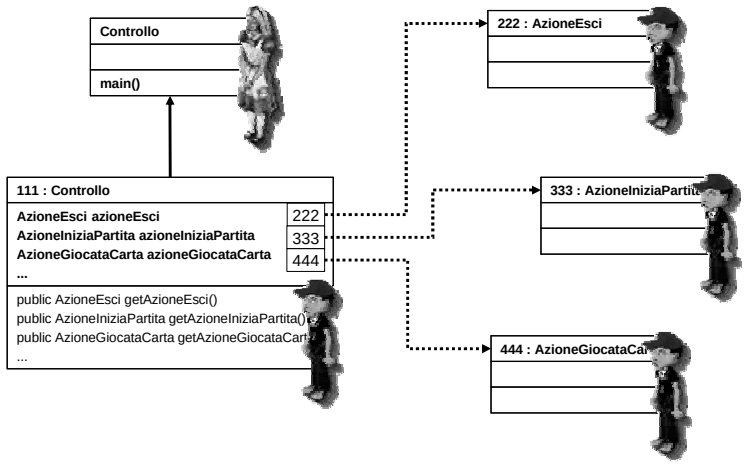
Distribuzione delle Responsabilità



Distribuzione delle Responsabilità

- I Problema di visibilità
  - ⇨nel codice della vista ho bisogno di accedere alle azioni (es: per creare i componenti)
- Soluzione
  - ⇨la classe Controllo ha una proprietà per ogni azione utilizzata nell'applicazione
  - ⇨e una serie di metodi get per consentire l'accesso alle azioni dall'esterno

Distribuzione delle Responsabilità



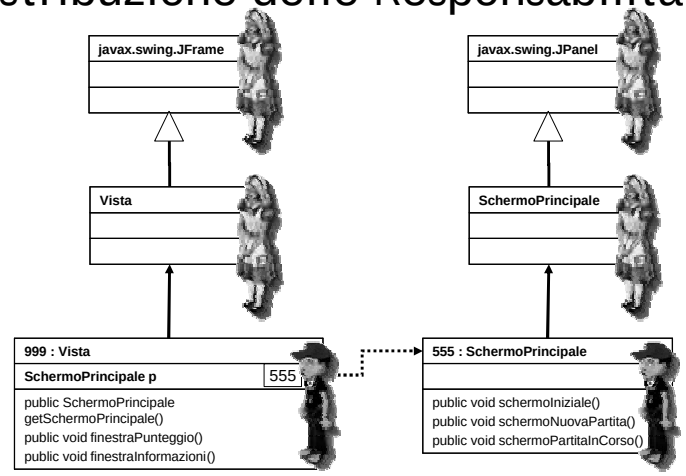
Distribuzione delle Responsabilità

- Il Problema di visibilità
  - ⇨nel codice delle azioni ho bisogno di accedere ai controlli dell'interfaccia
  - ⇨per acquisirne i valori forniti dall'utente
  - ⇨e per aggiornarne lo stato a seguito dell'evoluzione della logica applicativa
- Componenti per fornire valori in ingresso
  - ⇨tipicamente JTextField, JTextArea, JCheckBox, JRadioButton

Distribuzione delle Responsabilità

- Soluzione
  - ⇨le classi della vista forniscono metodi get per accedere ai valori dei componenti attraverso cui l'utente fornisce valori
  - ⇨inoltre forniscono metodi di "schermo" per aggiornarne lo stato a seguito di un'azione
  - ⇨l'azione acquisisce il valore del componente, elabora il valore, poi chiama il metodo di schermo per aggiornare l'interfaccia

Distribuzione delle Responsabilità



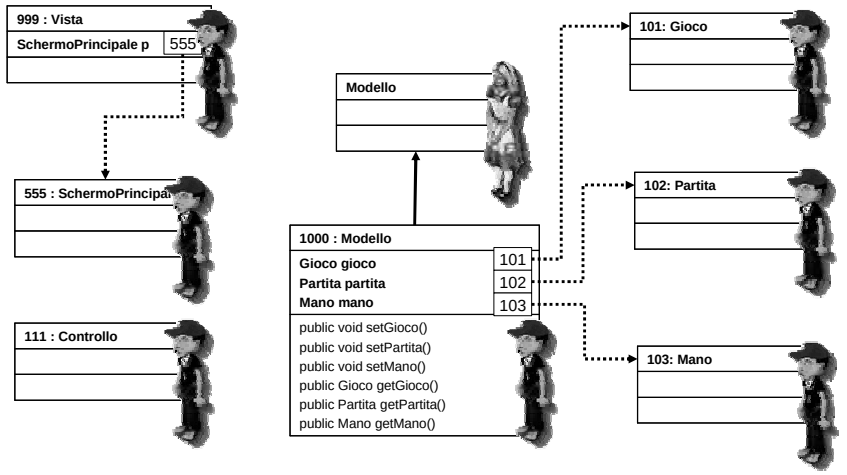
Distribuzione delle Responsabilità

- III Problema di visibilità
  - ⇨nelle azioni è necessario accedere ai bean del modello per chiamarne i metodi
  - ⇨lo stesso vale nella vista, per prelevarne i valori da mostrare sullo schermo
  - ⇨ho bisogno da tutte e due le parti di acquisire riferimenti ai bean
  - ⇨questi riferimenti possono essere molti nel caso in cui il modello sia complesso

Distribuzione delle Responsabilità

- Soluzione
  - ⇨utilizzo una classe Modello come “deposito” centralizzato di riferimenti ai bean
  - ⇨una proprietà per ciascun bean e metodi get e set per le proprietà
  - ⇨ogni volta che un’azione crea un bean la “salva” nel modello chiamando un metodo set
  - ⇨successivamente il bean è possibile lavorare con il bean utilizzando il metodo set

Distribuzione delle Responsabilità



Distribuzione delle Responsabilità

- Terminologia
  - ⇨la classe modello mantiene lo “stato della sessione di lavoro”
  - ⇨ovvero in questo caso lo stato del modello durante lo svolgimento del gioco
- Stato della sessione
  - ⇨è dato dai valori dei componenti del modello presenti nello heap

Distribuzione delle Responsabilità

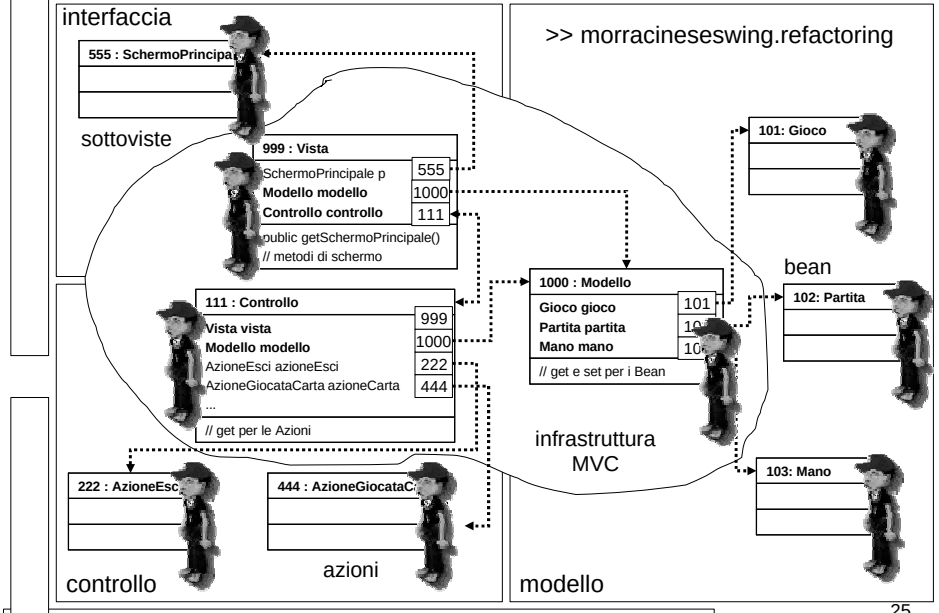
- Nota
  - ⇨il componente Modello non è indispensabile
  - ⇨ma è particolarmente utile per risolvere i problemi di visibilità collegati all’accesso ai bean
  - ⇨è sufficiente che le classi di vista e di controllo abbiano un riferimento al modello per avere accesso a tutti i bean
  - ⇨invece che tanti riferimenti a ciascun bean

Distribuzione delle Responsabilità

- Riassumendo
  - ⇨tre nuove classi significative
- Controllo
  - ⇨che mantiene lo stato delle azioni
- Vista (con le relative sottoviste)
  - ⇨che mantiene lo stato della gui
- Modello
  - ⇨che mantiene lo stato dei bean

Distribuzione delle Responsabilità

- Questi tre componenti
  - ⇨rappresentano l'infrastruttura fondamentale di un'applicazione con interfaccia grafica
- Relazione tra i tre componenti
  - ⇨il controllo mantiene un riferimento alla vista ed uno al modello
  - ⇨la vista mantiene un riferimento al controllo ed uno al modello



Distribuzione delle Responsabilità

- Riassumendo
  - ⇨per organizzare le responsabilità e semplificare la scrittura del codice abbiamo adottato un'infrastruttura standard
  - ⇨fatta delle classi Modello, Vista e Controllo
  - ⇨Modello: coordina l'accesso ai bean
  - ⇨Vista: coordina l'accesso alle sottoviste
  - ⇨Controllo: coordina l'accesso alle azioni

Distribuzione delle Responsabilità

- Nota
  - ⇨negli esempi le azioni continuano ad essere classi interne al controllo
  - ⇨questo però per pura comodità (accedono ai riferimenti al modello e alla vista)
  - ⇨sarebbe però possibile facilmente trasferirli in classi esterne fornendo nel costruttore il riferimento al controllo

Raffinamento

- Un altro esempio
  - ⇨it.unibas.indovinaswing
- Caratteristiche dell'applicazione
  - ⇨complessità comparabile alla Morra Cinese
  - ⇨ma ci sono alcune differenze
  - ⇨necessità di effettuare convalide dei dati
  - ⇨due schermi (nome e poi partita)
  - ⇨utilizzo di gestori di layout

Raffinamento

- Dimensioni delle due versioni
  - ⇨versione console: 360 linee di codice
  - ⇨versione swing: 744 linee di codice (+52%)
- Il package it.unibas.indovinaswing.gui
  - ⇨complessivamente 454 linee di codice
  - ⇨organizzato secondo l'infrastruttura MVC
  - ⇨con un raffinamento

Raffinamento

- Raffinamento dell'infrastruttura
  - ⇨separiamo le azioni dalla classe di controllo
  - ⇨introduzione di mappe di riferimenti al posto dei metodi get/set
- Separiamo le azioni
  - ⇨le azioni non sono più classi interne al controllo
  - ⇨ricevono un riferimento al controllo attraverso il costruttore e quindi anche a vista e modello

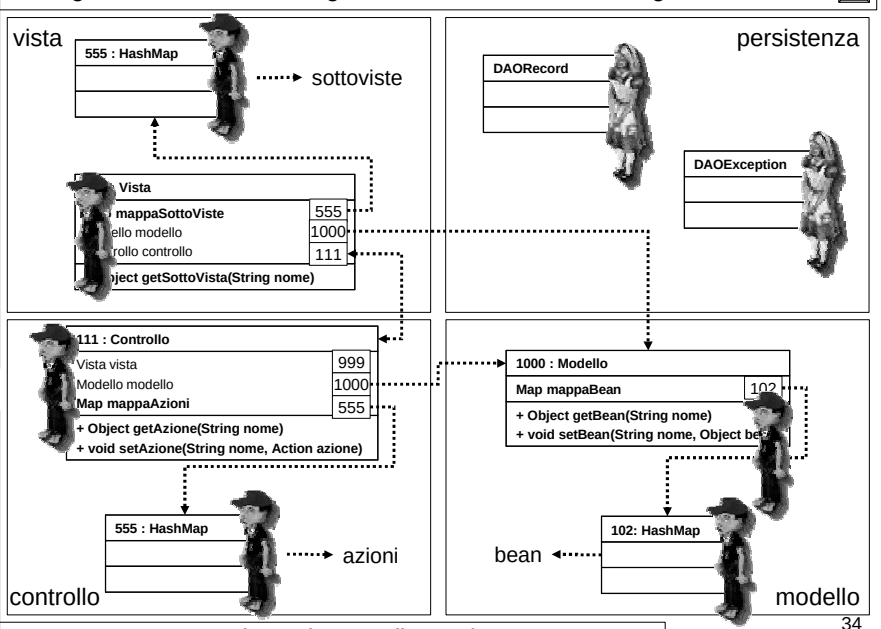
```
package it.unibas.indovinaswing.controllo;

public class AzioneInformazioni extends javax.swing.AbstractAction {

    private Controllo controllo;

    public AzioneInformazioni(Controllo controllo) {
        this.controllo = controllo;
        this.putValue(javax.swing.Action.NAME, "Informazioni sul gioco");
        this.putValue(javax.swing.Action.SHORT_DESCRIPTION,
                        "Informazioni sul gioco");
        this.putValue(javax.swing.Action.MNEMONIC_KEY,
                        new Integer(java.awt.event.KeyEvent.VK_M));
    }

    public void actionPerformed(java.awt.event.ActionEvent e) {
        this.controllo.getVista().finestraInformazioni();
    }
}
```


Raffinamento

- Mappe dei riferimenti
  - ⇨i bean nel modello possono essere molti
  - ⇨le azioni nel controllo possono essere molte
  - ⇨le sottoviste nella vista possono essere molte
  - ⇨per ciascuna dovrei definire un metodo get (e nel modello anche il metodo set)
  - ⇨codice lungo e noioso da manutenere

Raffinamento

- Soluzione
  - ⇨sostituisco i riferimenti ordinari con una mappa di riferimenti
  - ⇨nel modello i metodi get/set diventano Object getBean(String nome) void putBean(String nome, Object bean)
  - ⇨nella vista Object getSottoVista(String nome)
  - ⇨nel controllo Action getAzione(String nome)
  - ⇨ATTENZIONE a inizializzare le mappe



Raffinamento

- Inizializzazione del controllo
  - ⇨crea tutte le azioni e le aggiunge alla mappa

```
//  Controllo.java
private void inizializzaAzioni() {
    mappaAzioni.put("azioneEsci", new AzioneEsci(this));
    mappaAzioni.put("azioneRecord", new AzioneRecord(this));
    mappaAzioni.put("azioneAbout", new AzioneInformazioni(this));
    mappaAzioni.put("azioneNuovaPartita",
                            new AzioneNuovaPartita(this));
    ...
}
```


Raffinamento

- La gerarchia delle viste
  - ⇨in questo caso, a differenza che nel precedente, ho due diversi schermi che vengono visualizzati all'utente
  - ⇨Vista: gestisce il frame principale
  - ⇨SchermoIniziale: pannello che corrisponde allo schermo iniziale (per fornire il nome)
  - ⇨SchermoPartita: pannello che corrisponde allo schermo su cui si gioca la partita

Raffinamento

- Inoltre
  - ⇨vengono visualizzate varie finestre di dialogo (finestre secondarie) standard
  - ⇨attraverso metodi di JOptionPane
- Inizializzazione della mappa
  - ⇨è indispensabile che la vista inizializzi correttamente la mappa inserendo i riferimenti a tutte le sottoviste ed inizializzandole opportunamente

Raffinamento

```
// Vista.java
public void inizializza() {
        this.mappaSottoViste.put("schermoIniziale",
                                 new SchermoIniziale(this));
        this.mappaSottoViste.put("schermoPartita",
                                 new SchermoPartita(this));
        this.getContentPane().setBackground(new Color(255, 241, 167));
        creaFrame();
        creaMenu();
        this.pannelloPrincipale = (JPanel)getContentPane();
}
```

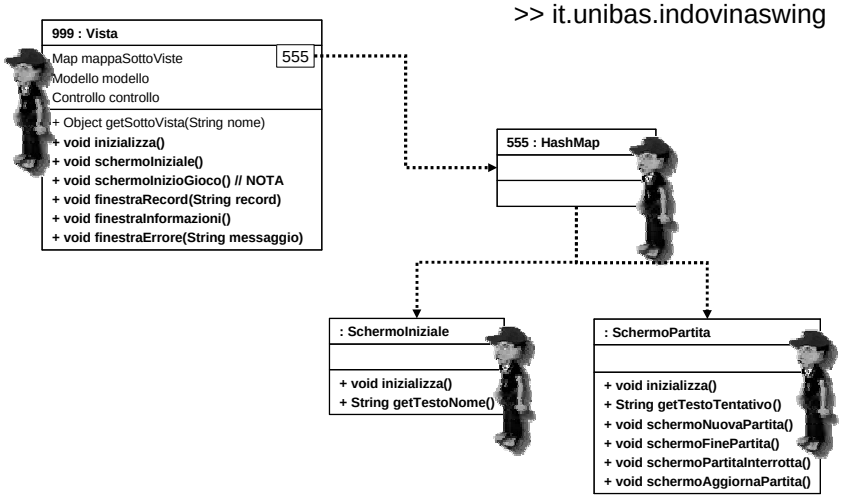
Raffinamento

- Un metodo interessante
  - ⇨il metodo schermoAvviaGioco()
  - ⇨è il metodo attraverso il quale viene sostituito lo schermo
- Idea
  - ⇨viene eseguito dalla vista principale
  - ⇨rimuove dal content pane il pannello iniziale
  - ⇨e lo sostituisce con l'altro
  - ⇨al termine richiama il metodo pack()

```
// Vista.java
 public void schermoIniziale() {
       SchermoIniziale schermoIniziale =
              (SchermoIniziale)this.mappaSottoViste.get("schermoIniziale");
       this.getContentPane().add(schermoIniziale);
       this.pack();
       this.setVisible(true);
   }

public void schermoAvviaGioco() {
      SchermoIniziale schermoIniziale =
              (SchermoIniziale)this.mappaSottoViste.get("schermoIniziale");
      SchermoPartita schermoPartita =
              (SchermoPartita)this.mappaSottoViste.get("schermoPartita");
      this.getContentPane().remove(schermoIniziale);
      this.getContentPane().add(schermoPartita);
      schermoPartita.schermoNuovaPartita();
      this.pack();
      // this.repaint();  NON E' INDISPENSABILE: forza il frame a ridisegnarsi
}
```

Raffinamento

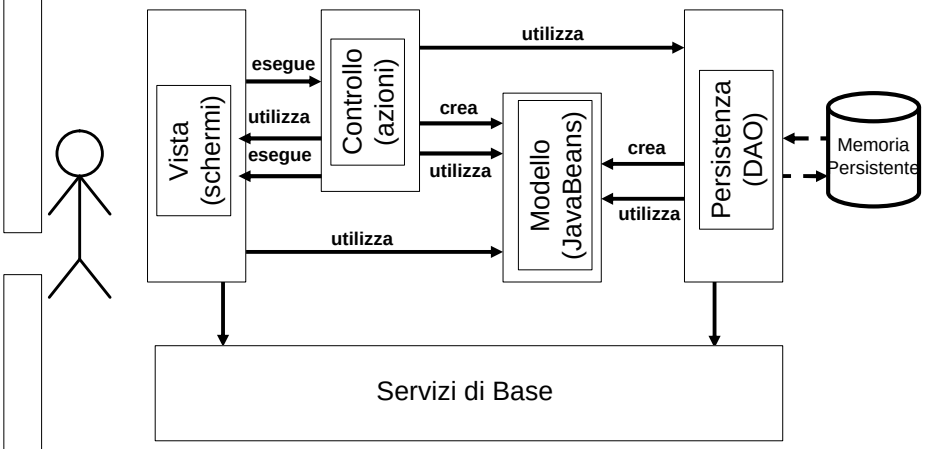
- Rapporto tra vista e controllo
  - ⇨in questo caso, a differenza che nel precedente, le azioni devono leggere i valori forniti dall'utente (nome e tentativo)
  - ⇨è necessario che la vista fornisca metodi opportuni
  - ⇨String getTestoNome() in SchermoIniziale
  - ⇨String getTestoTentativo() in SchermoPartita



MVC

- Riassumendo
  - ⇨abbiamo separato completamente i quattro strati fondamentali dell'architettura
- Architettura Modello-Vista-Controllo
  - ⇨architettura in cui il codice dei tre strati di modello, interfaccia e controllo è separato
  - ⇨può esserci un ulteriore strato di persistenza (che può anche essere accorpato al modello)

MVC



MVC

- Vantaggi della separazione degli strati
  - ⇨utile in applicazioni di dimensioni medio/grandi e di struttura articolata (es: Swing; con la console sarebbe inutile)
  - ⇨aiuta ad organizzare il codice
  - ⇨riduce l'accoppiamento tra gli strati se vengono rispettate le regole
  - ⇨e quindi semplifica la manutenzione

MVC

- Ma...
  - ⇨la separazione tra vista e controllo che abbiamo sviluppato è imperfetta
  - ⇨i due strati restano fortemente accoppiati (unico caso di frecce di comunicazione bidirezionali)
  - ⇨sarà necessario intervenire ulteriormente

Riassumendo

- Riorganizzazione del Codice
  - ⇨Distribuzione delle Responsabilità
  - ⇨Raffinamento
- MVC

Termini della Licenza