PROGETTO JUNO

Corso di Metodologie di Programmazione in Teledidattica
Sapienza Universita' di Roma

Nome: Giorgio Carlo Roberto Albieri
Matricola: 2009437
Data Consegna: 14/04/2023

1. Descrizione

Il progetto JUno consiste nello sviluppo di un'applicazione java che simula il gioco di carte dell'UNO nella sua modalità classica e facendo uso della libreria Java Swing per la realizzazione della GUI.
Il codice è strutturato secondo il modello del MVC Pattern che consente di mantenere disaccoppiato il Model dalla View al fine di consentire un facile riutilizzo del model su qualsivoglia nuova interfaccia grafica.

2. Specifiche progettuali

Le specifiche progettuali che sono state sviluppate nel presente progetto sono le seguenti e verranno trattate una ad una nel presente documento.

1. Gestione del profilo utente
2. Gestione di una partita completa in modalità classica con un giocatore umano e 3 artificiali
3. Uso appropriato di MVC, Observer Observable e altri Design Patterns
4. Adozione di Java Swing per la GUI
5. Utilizzo appropriato degli Streams
6. Riproduzione di audio sample

3. Gestione del profilo utente

L'applicazione è stata sviluppata al fine di consentire il caricamento e il salvataggio di qualsivoglia profilo utente al fine di conservarne i principali dati di gioco quali il nome, il punteggio, il numero di partite giocate, vinte e perse.

Ciò è conseguito tramite la serializzazione e la deserializzazione della classe *ProfiloUtente.java*, che è contenuta nel package *model*, ed implementa l'interfaccia *Serializable*.

Il tutto è gestito dalla classe *GestoreProfilo*, contenuta nel package *Controller,* e che, grazie all'utilizzo del ***Facade pattern***, è costruita in modo tale da nascondere il processo all'utente all'interno di 3 soli metodi (caricaProfilo, creaProfilo e aggiornaProfilo).

La serializzazione e la deserializzazione della classe *ProfiloUtente.java* sono conseguite utilizzando le classi `java.io.FileInputStream, java.io.FileOutputStream, java.io.ObjectInputStream` e `java.io.ObjectOutputStream` che consentono di leggere e salvare file in formato *.ser.

Non appena l'applicazione viene lanciata, il menu iniziale consente all'utente di scegliere se caricare un profilo utente esistente (selezionando un file *.ser dal disco) o crearne uno nuovo semplicemente inserendo il nome del nuovo profilo. A seconda della scelta dell'utente, la classe *GestoreProfilo* carica il file *.ser selezionato convertendolo in una istanza di classe *ProfiloUtente* (*GestoreProfilo.caricaProfilo)* oppure crea una nuova istanza di *ProfiloUtente* avente il nuovo nome inserito (*GestoreProfilo.creaProfilo*).

Il nome del profilo viene assegnato all'istanza di classe *GiocatoreHuman*, contenuta nel package *model*, che ha il compito di modellare il comportamento dell'utente durante la partita immagazzinando informazioni sul punteggio, il numero di round vinti etc.

Una volta terminata la partita, l'istanza di classe *GestoreProfilo* preleva i dati di gioco di *GiocatoreHuman* e li usa per aggiornare l'istanza di classe *ProfiloUtente* per poi serializzarlo e salvarlo come file *.ser nella cartella *profiles/* del progetto java.

Il diagramma delle classi *GestoreProfilo* e *ProfiloUtente*, estratto dal diagramma UML del progetto, e' riportato qui di seguito.

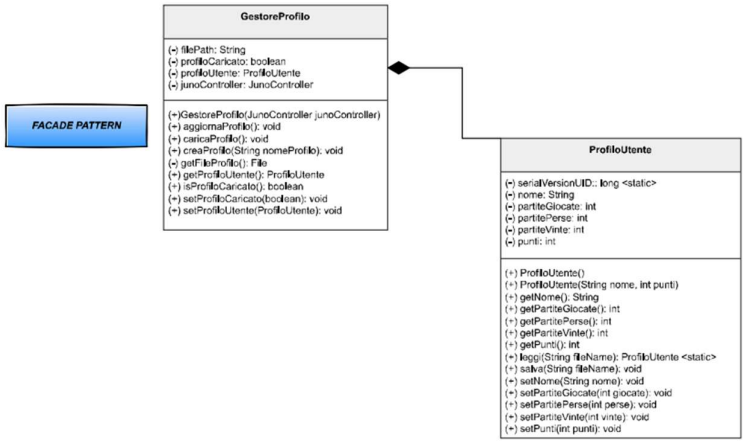

Figura 1: Estratto Diagramma UML classi GestoreProfilo.java e ProfiloUtente.java

4. Gestione partita

La gestione della partita è conseguita tramite l'utilizzo del Model-View-Controller Compound Pattern in abbinamento con l'Observer Pattern. Una descrizione dettagliata dei Design Patterns utilizzati nel progetto è riportata nel paragrafo Design Patterns del presente report.

Qualsivoglia interazione dell'utente con la GUI, viene recepita dal Controller per mezzo degli ActionListeners (EventHandlers) assegnati ai componenti della View. Questi hanno il compito di chiamare i principali metodi del Model che consentono di sviluppare il gioco della partita tramite turni gestiti da un Timer all'interno del Controller. Non appena il turno di gioco viene concluso e il Model aggiornato, il Model stesso lancia il metodo .notifyObservers() che chiama il metodo .update() degli observers (JFrames) della View. Questi aggiornano quindi tutte le proprie componenti grafiche per mostrare il nuovo stato del Model e del gioco.

Il Model

Le classi principali del Model, essenziali per lo sviluppo del gioco, sono JunoModel, Giocatore e Carta.

JunoModel

JunoModel è la classe di riferimento per l'aggiornamento dei dati di gioco.

La lista dei giocatori, il mazzo di carte, il giocatore del turno, il colore e il simbolo del turno, i parametri di controllo (RoundOver, GameOver etc.) …tutte le principali entità del gioco sono registrate come attributi privati della classe JunoModel (vedi diagramma UML).

I metodi setter e getter consentono la manipolazione e la lettura degli attributi che viene eseguita direttamente dal Model, all'interno dei metodi di gestione del gioco implementati dall'interfaccia JunoModelInterface (inizializza(), giocatTurno(), giocaRound(), giocaPartita()), e indirettamente dal Controller.

Come già accennato, l'aggiornamento della GUI viene eseguito tramite notifica degli observers registrati nel model (***Observer pattern***).
Per garantire l'unicità dell'istanza di classe JunoModel, si fa ricorso al ***Singleton Pattern*** (costruttore privato e metodo statico .getInstance()).

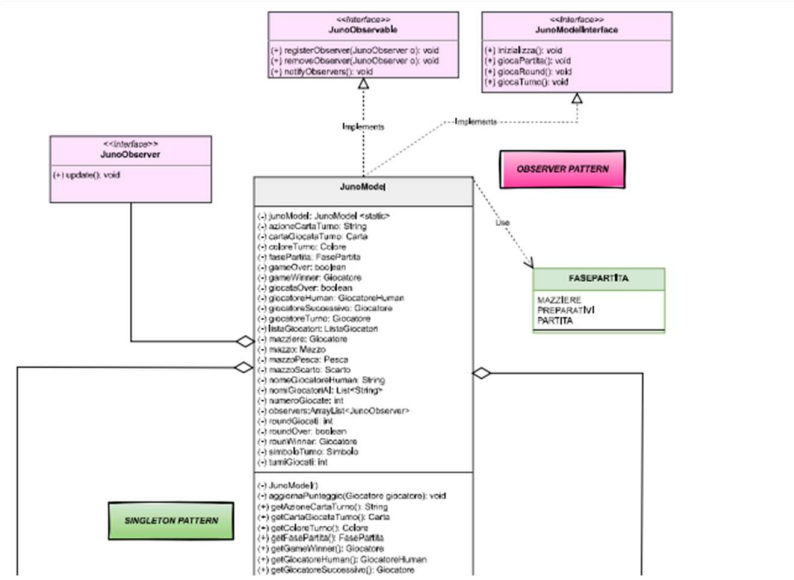

Figura 2: Estratto Diagramma UML classe JunoModel.java

Carta

Carta è la classe astratta che contiene le proprietà di ciascuna carta del gioco (colore, simbolo, nome, valore, faceUp…) e la definizione del suo corrispondente comportamento/azione (tramite l'uso dello ***Strategy Pattern***). Qualsiasi classe concreta che rappresenti una particolare tipologia di carta (CartaNumerica, CartaSalto, CartaJolly…) estende la classe astratta Carta ereditandone attributi e metodi.

Per garantire l'ordinabilita' e il confronto fra le carte all'interno di strutture dati ordinabili, l'interfaccia Comparable è implementata e i metodi hashCode(), equals() e compareTo(T o) sono sovrascritti opportunamente.

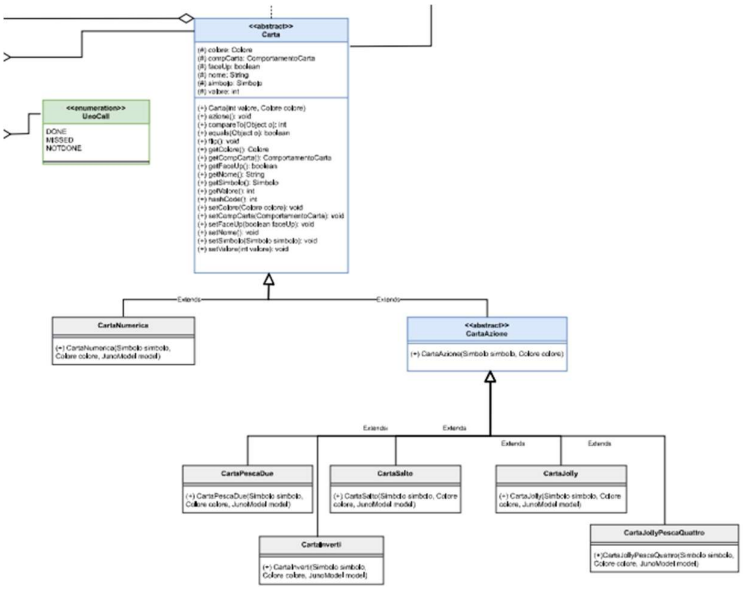

Figura 3: Estratto Diagramma UML classe Carta.java

Giocatore

La classe Giocatore contiene tutte le informazioni riguardanti lo status del giocatore durante la partita (nome, posizione, mano di carte, carta giocata, turni giocati, chiamata dell'UNO…) come anche i metodi che ne definiscono le principali azioni di gioco (scegliCarta, pescaCarta, scartaCarta, giocaCarta…).

Gli unici metodi su cui il comportamento del giocatore AI si differenzia dal giocatore umano sono quelli che richiedono la diretta interpretazione della scelta effettuata dall'utente, ovvero scegliCarta(), controllaUno() e callUno() (implementati dall'interfaccia ComportamentoGiocatore).

Mentre i giocatori AI sono programmati per eseguire le operazioni sopracitate secondo una determinata logica definita in corrispondenti algoritmi (contenuti all'interno dei metodi stessi), il giocatore umano risponde alle scelte libere dell'utente.

Per tale ragione i due tipi di giocatore sono modellati tramite due diverse sottoclassi (GiocatoreAI e GiocatoreHuman) della superclasse Giocatore da cui ereditano tutti i metodi tramite quelli sopra citati e per i quali definiscono diverse implementazioni.

Per consentire il confronto e l'ordinabilità delle diverse istanze di Giocatore all'interno di strutture dati ordinabili, la classe Giocatore implementa l'interfaccia Comparable e sovrascrive opportunatamente i metodi hashCode(), equals() e compareTo(T o).
La lista dei giocatori della partita è quindi incapsulata nella classe ListaGiocatori la quale la decora (***Decorator Pattern***) con metodi di utilità specifici per la gestione dei turni della partita (aggiornaOrdine(), invertiOrdine(), assegnaPosizioni() etc.)

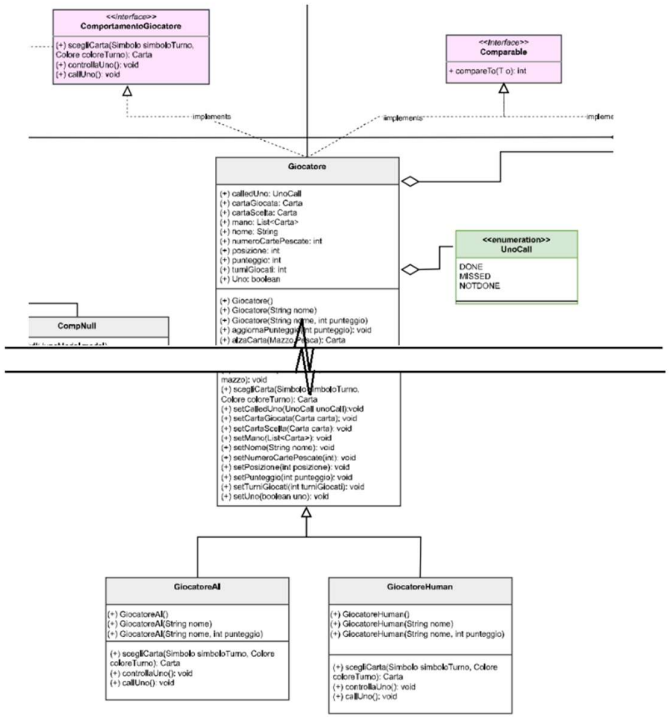

Figura 4: Estratto Diagramma UML classe Giocatore.java

La classe JunoModel, quindi, sulla base degli input provenienti dalla View tramite il Controller, va a chiamare i principali metodi di comportamento dei giocatori con conseguente modifica di punteggi, mano di carte, mazzo di scarto ecc. fino alla conclusione del turno in corrispondenza del quale aggiorna la View. Il tutto continua fino alla conclusione del gioco con la vittoria da parte di uno dei giocatori

La View

Le classi principali della View, essenziali per lo sviluppo del gioco, sono JunoView, ViewMenu, ViewGame, PlayerPanel e CartaPanel.
La quasi totalità delle classi della View ereditano da classi della libreria java Swing.

JunoView

La classe JunoView è la classe di riferimento di tutto il package View.
Essa contiene il riferimento al Model e al Controller, lancia i metodi per la creazione delle diverse finestre della GUI (i.e. ViewMenu e ViewGame) e della classe Notificatore che crea le notifiche testuali di aggiornamento per l'utente tramite il ***Builder Pattern.***

ViewMenu

La Classe ViewMenu eredita dalla classe JFrame ed implementa l'interfaccia ViewUnoInterface che definisce le costanti di tipo stringa contenenti i filepath rilevanti e l'unico metodo che tutte le finestre della GUI devono sempre implementare: il metodo creaComponenti().
Il metodo creaComponenti() ha il compito di creare tutte le componenti contenute all'interno del JFrame corrispondente andando a nascondere all'utente la miriade di chiamate di funzioni java swing necessarie per ottenere ciò (***Facade Pattern***).
La ViewMenu è la prima finestra visualizzata e ha il compito di consentire all'utente di creare/caricare un profilo utente, scegliere i giocatori AI contro cui giocare, scegliere la colonna sonora ecc.
Tramite gli actionListeners assegnati dal Controller ai bottoni CARICAPROFILO e GIOCAPARTITA della VeiwMenu è possibile caricare il profilo serializzato e lanciare il JFrame ViewGame rispettivamente.

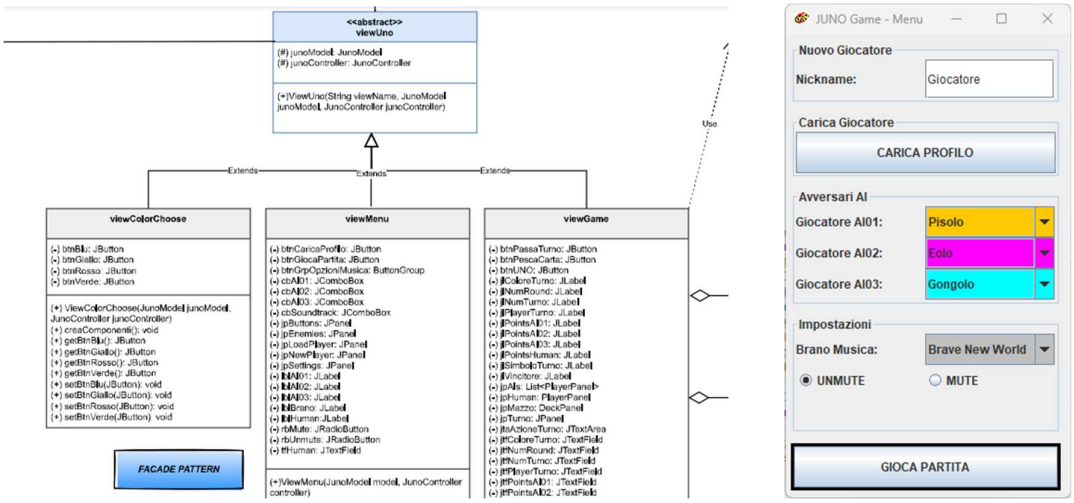

Figura 5: Estratto Diagramma UML classi JFrame (sinistra), Finestra ViewMenu (destra)

ViewGame

La classe ViewGame è la seconda e principale finestra del gioco e che consente lo sviluppo della partita dall'inizio alla fine.
Tramite l'utilizzo di diversi LayoutManagers provvisti dalla libreria Java Swing, essa si suddivide in due parti fondamentali: Il tavolo da gioco sulla sinistra, e il pannello informativo dei dati del gioco sulla destra.
Il tavolo da gioco è costituito da 5 JPanels: uno per la rappresentazione grafica del mazzo e 4 per la rappresentazione grafica delle mani di carte dei giocatori.
Il pannello informativo mostra, invece, il numero, il simbolo e il colore del turno, il numero di round giocati come anche una descrizione testuale di quanto avvenuto nel turno appena conclusosi.
Infine, i 3 bottoni "Pesca", "Passa Turno" e "UNO" in fondo al pannello informativo consentono all'utente di compiere le corrispondenti azioni durante il proprio turno.
Ulteriore interazione consentita all'utente consiste nella selezione della carta da giocare cliccando su di essa nel corrispondente JPanel. Se la carta è consentita in base al simbolo e al colore del turno vigente essa viene spostata nel mazzo scarto e il suo effetto viene visualizzato graficamente come anche descritto testualmente all'interno della JTextArea corrispondente e contenuta all interno del pannello informativo.

La classe ViewGame viene aggiornata sulle modifiche del model al termine di ciascun turno tramite l'***Observer Pattern*** e il metod .update() implementato dall'interfaccia JunoObservable.

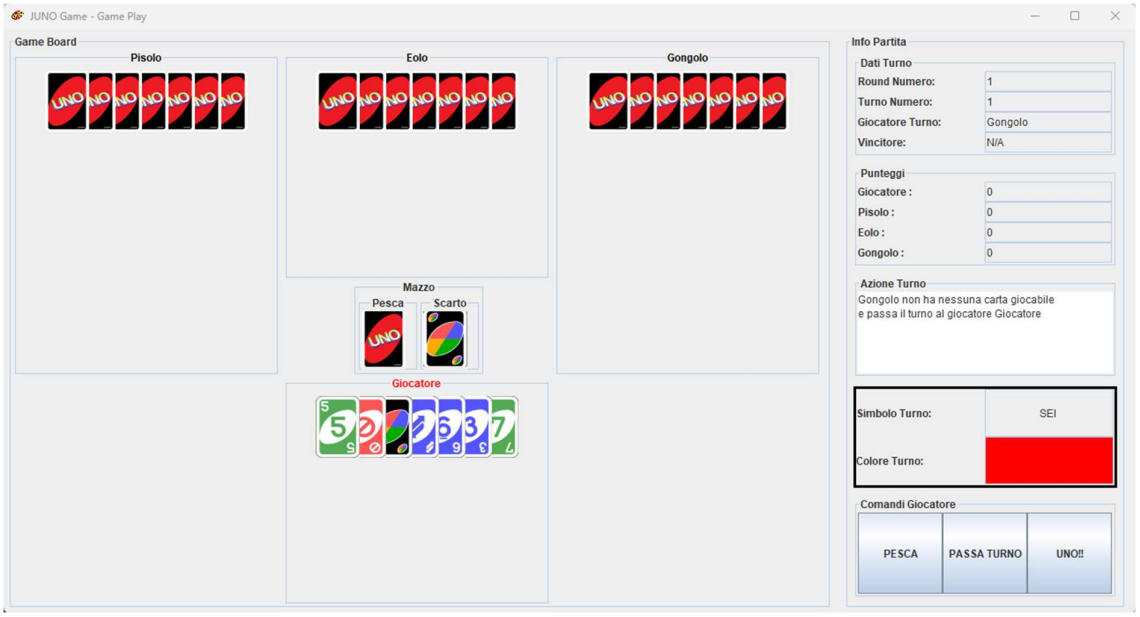

Figura 6: Finestra ViewGame durante partita

CartaPanel

CartaPanel è la classe che definisce la componente di base di tutta la GUI ed è costrutita attorno alla clase Carta tramite il ***Decorator Pattern***.
In base alle proprietà dell'istanza di classe Carta che incapsula, CartaPanel definisce la visualizzazione grafica della stessa tramite funzioni java Swing contenute all'interno del Costruttore e di metodi privati di utilità (setImage()).
A sua volta, le istanze della classe CartaPanel sono componenti costitutive delle classi DeckPanel e PlayerPanel le quali decorano analogamente e rispettivamente le classi Mazzo e Giocatore del Model facendo uso del Decorator Pattern.

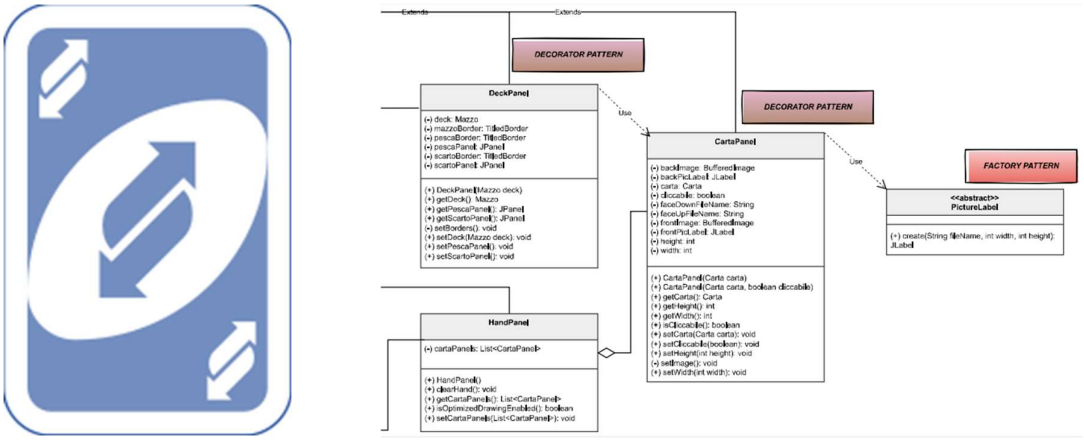

Figura 7: Classe Carta Panel- Visualizzazione grafica (sinistra), Diagramma UML (destra)

Il Controller

Le classi principali del Controller, essenziali per lo sviluppo del gioco, sono JunoController, TurnoTimerListener, CardsMouseListener e GestoreProfilo.

JunoController

La classe JunoController è la classe di riferimento dell'intero package Controller.
Nel costruttore, essa crea il Model e la View oltre agli AudioManagers (SoundManager e MusicManager), gli ActionListeners e gli ExceptionHandlers.
Tramite il metodo .inizializza(), junoController lancia la JFrame ViewMenu della View che consente all'utente di selezionare le impostazioni di gioco.
Tramite il metodo .iniziaPartita(), invece, lancia la JFrame ViewGame e inizializza il TurnoTimerListener che è il motore dei turni della partita.

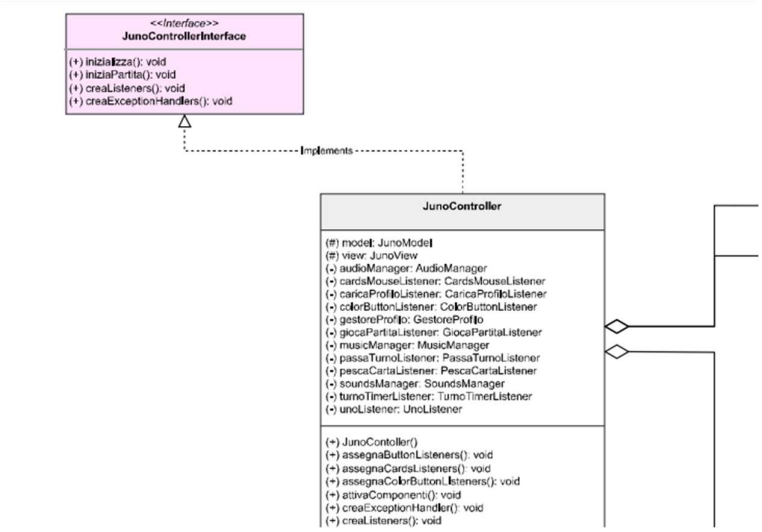

Figura 8: Estratto Diagramma UML classe JunoController

TurnoTimerListener

La classe TurnoTimerListener estende la classe astratta AbstractTimerListener che, a sua volta, implementa l'interfaccia TimerListenerInterface, sottoclasse di ActionListener.
TurnoTimerListener contiene un riferimento alla classe timer e un riferimento alla classe JunoController.
Il primo le consente di attivare e disattivare il timer in base alle diverse fasi di gioco mentre il secondo le consente di avere accesso diretto a tutti i metodi e attributi del controller con i quali viene gestita la comunicazione tra la View e il Model.
Tutte le operazioni descritte sopra sono contenute all'interno del metodo actionPerformed(ActionEvent e), il quale viene chiamato ogni volta che il timer viene attivato.

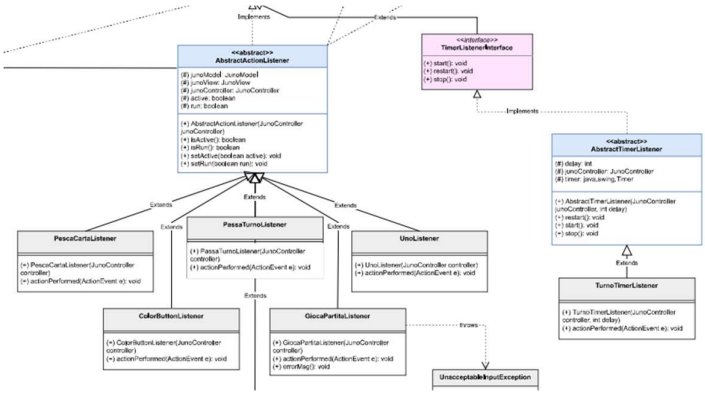

Figura 9: Estratto Diagramma UML classi ActionListener e MouseListener

CardsMouseListener

La classe CardsMouseListener, tra tutte le classi di tipo Action/MouseListener contenute nel package Controller, è probabilmente la più importante in quanto consente all'utente di selezionare la carta da giocare nella JFrame ViewGame, cliccando sul JPanel corrispondente.
Come tutti gli altri listeners, esso viene assegnato alle componenti java Swing corrispondenti direttamente da JunoController.

GestoreProfilo

La classe GestoreProfilo è fondamentale per la gestione del profilo utente interno al Model.
Per ulteriori informazioni a riguardo, vedi paragrafo precedente riguardo alla gestione del profilo utente.

La Classe JUno

Infine, il main per il lancio del gioco e' contenuto nella classe esterna JUno il cui unico scopo e' appunto di contenere il main da eseguire per il lancio del programma.

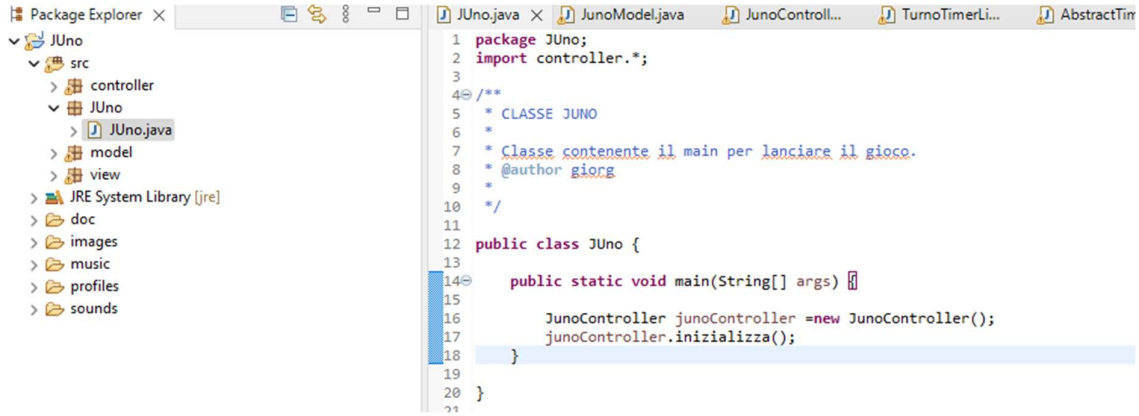

Figura 10: Classe JUno con il main per il lancio del programma

5. Design Patterns

In forma piu' sintetica rispetto al paragrafo precedente, si riporta di seguito l'elenco dei principali Design Patterns usati nel progetto (vedi anche diagrammi UML).

Model-View-Controller Pattern

Il MVC Pattern è il pattern su cui si base l'intero funzionamento del programma.
Esso gestice la comunicazione dalla GUI (view) al DataBase (Model) tramite il controller in modo da disaccoppiare la View dal Model e massimizzare il riutilizzo del Model stesso.
Il Controller ha il compito di creare il Model e la View passando alla View il riferimento al Model (essenziale per l'uso dell'Observer Pattern) e a se stesso.

Classi Interessate: *JunoController, JunoModel, JunoView*.

Observer Pattern

L'Observer Pattern è l'attore principale nella gestione della comunicazione dal Model verso la View.
Le diverse finestre (JFrame) della View vengono registrate Controller come observers del Model.
Ogni volta che un turno di gioco si è concluso, il Model lancia il metodo notifyObservers() che va a chiamare il corrispondete metodo update() contenuto in ciascuno di essi.
Nel metodo update(), la View, usando il riferimento al Model passatogli dal Controller, può estrarre agevolmente i dati aggiornati del modello e assegnarli alle componenti java Swing della GUI.

Classi Interessate: *JunoModel, JunoView. ViewGame, ViewMenu*

Strategy Pattern

Lo Strategy Pattern è stato utilizzato nella definizione e gestione dei comportamenti delle carte da gioco.
Ciascun tipo di carta compie un'azione con effetti differenti sia sul gioco che sugli altri giocatori.
Utilizzare l'ereditarietà per il metodo azione() delle diverse carte riduce la flessibilità in quanto bisognerebbe lanciare un'eccezione nel caso in cui una carta sia caratterizzata da azione nulla (es. carte numeriche).
Utilizzare l'implementazione tramite interfaccia costringe a ridefinire il metodo azione di ripetere l'implementazione di tale metodo in tutte le classi anche quando essa non cambia.
Per tali ragioni, si fa ricorso allo Strategy Pattern.
Si crea un'interfaccia funzionale ComportamentoCarta caratterizzata da un unico metodo azione().
Quindi si crea una classe astratta Comportamento che implementa l'interfaccia ComportamentoCarta e contiene un riferimento diretto al Model in modo da consentire di operare direttamente su di esso in ciascuna implementazione specifica del metodo azione().
Infine, si creano le classi di comportamento concrete (CompPescaDue, CompInverti…) che implementano il metodo azione andando ad operare direttamente sul Model in base alla tipologia di carta corrispondente.

In tal modo, ciascuna classe concreta di tipo Carta, in base alla tipologia di carta che rappresenta, viene assegnata con un riferimento all'istanza di classe di comportamento carta corrispondente.

Il metodo azione() lanciato dalla carta, si limita quindi a chiamare il metodo azione() del comportamento carta corrispondente.

Classi interessate: *Comportamento, CompPescaDue, CompInverti, CompSalto, CompNull, CompJolly, CompJollyPescaQuattro*

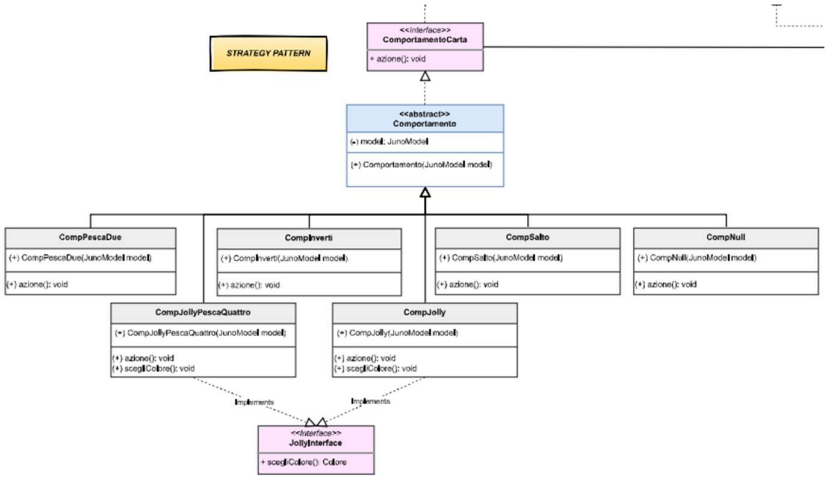

Figura 11: Estratto Diagramma UML classi di Comportamento dello Strategy Pattern

Singleton Pattern

Il Singleton Pattern viene utilizzato per assicurarsi che alcune classi chiave del progetto siano vincolate ad avere una ed una sola istanza.
Il costruttore è privato e per accedere all'unica istanza si fa uso del metodo statico getInstance()

Classi interessate: *JunoModel, MusicManager, SoundsManager*

Factory Pattern

Il Factory Pattern viene impiegato nel progetto nei casi in cui si debbano generare numerose istanze di classi differenti e in cui si voglia incapsulare e nascondere all'utente l'algoritmo necessario per la loro creazione.
Il Factory Pattern è stato utilizzato per la creazione delle carte del Model e dei picture labels dei CartaPanels della View.

Classi interessate: *CardsFactory, UnoCardsFactory, PictureLabel*

Facade Pattern

Il Facade Pattern viene impiegato nel progetto nei casi in cui bisogna chiamare un grande numero di metodi su un grande numero di oggetti per svolgere determinate operazioni.
Poter nascondere tutte queste chiamate all'interno di un singolo metodo di una classe apposita rende il codice più snello e maggiormente leggibile.

Questa strategia è stata utilizzata nella creazione delle componenti dei JFrames della View (metodo creaComponenti()) e nella gestione del profilo utente ad opera della classe GestoreProfilo nel Controller (metodi caricaProfilo() e aggiornaProfilo()).

Classi interessate: *ViewMenu, ViewGame, ViewColorChoose, GestoreProfilo*

6. Streams

L'utillizzo degli Streams è stato di notevole aiuto nello snellire il codice in particolar modo nelle operazioni di ricerca all'interno dei dati contenuti nel Model.

Una lista concisa dei frangenti in cui si sono utilizzati gli Streams all'interno del progetto è riportata di seguito insieme al riferimento alle classi in cui sono stati usati.

Uso degli Streams in tutto il progetto (fare lista di classi/metodi in cui vengono usati)

- Identificazione del vincitore della partita
  - Classe interessata: *TurnoTimerListener*
- Identificazione del vincitore del round
  - Classe interessata: *TurnoTimerListener*
- Aggiornamento punteggio giocatori
  - Classe interessata: *JunoModel.aggiornaPunteggio()*
- Ricerca della carta migliore da giocare per il giocatore AI
  - Classe interessata: *GiocatoreAI*
- Identificazione del mazziere in base al valore della carta pescata da ciascun giocatore
  - Classe interessata: *JunoModel*
- Inversione ordine giocatori
  - Classe interessata: *ListaGiocatori*
- Creazione delle carte da gioco
  - Classe interessata: *UnoCardsFactory*
- Ricavare lista nomi brani musicali da cartella esterna
  - Classe interessata: *ViewMenu*

Si riporta inoltre qui di seguito un'estratto del codice sorgente per mostrare un esempio di Stream utilizzato nel progetto.

```
// aggiornaPunteggiolistaGiocatori
public void aggiornaPunteggio(Giocatore giocatore) {

     int punteggio=
     // Creazione Stream su Lista listaGiocatori tramite STREAMS
     this.listaGiocatori.getGiocatori().stream()
        // 1. Trattieni tutti gli altri Giocatori
        .filter((gci)->!gci.equals(giocatore))
        // 2. Mappa i Giocatori alle loro Mano corrispondenti
        .map((gci)->gci.getMano())
        // 3. Converti Stream<List<Carta>> in Stream<Carta>
        .flatMap((glist)->glist.stream())
        // 4. Somma i valori di tutte le carte
        .collect(Collectors.summingInt((crt)->(int)crt.getValore()));

     // Aggiungi il punteggio calcolato al giocatore corrispondente
     giocatore.aggiornaPunteggio(punteggio);}
```

Figura 12: Esempio di utilizzo degli Streams

7. Riproduzione Audio

Nel progetto si utilizzano due diverse classi concrete per la gestione degli effetti sonori: SoundsManager e MusicManager. Si è scelto di usare due distinte classi per lo gestione dei suoni e della colonna sonora per consentire di disaccoppiare le impostazioni corrispondenti (volume, mute/unmute…).

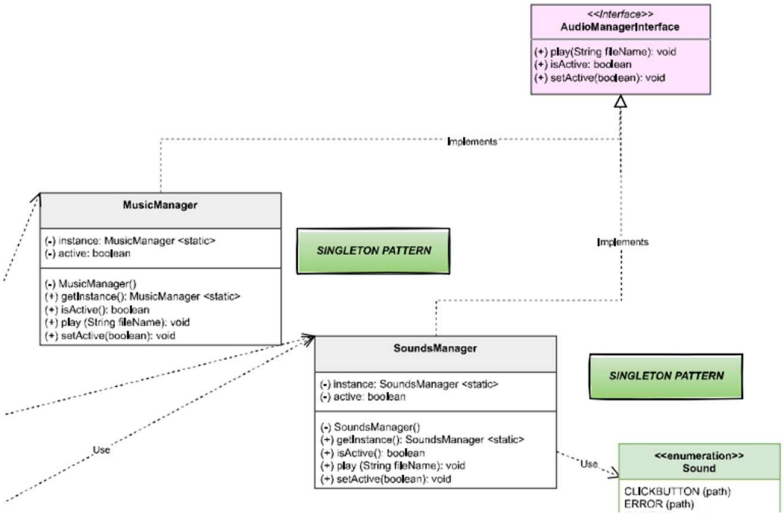

Figura 13: Estratto Diagramma UML classi AudioManager