JTrash

Nome: Giovanni
Cognome: Ferretti
Matricola: 2070106
Corso: teledidattica
Settembre 2023

INDICE

GESTIONE PROFILO UTENTE

Gestione Profilo Utente (dal Model)

L'architettura di un software è spesso il risultato di decisioni accurate e ponderate che mirano a massimizzare l'efficienza e la flessibilità del sistema. In particolare, quando si tratta di giochi, la gestione dei profili degli utenti può diventare una componente fondamentale, poiché determina l'esperienza personalizzata dell'utente e il suo progresso. Nella struttura di JTrash, ho pensato di attuare dei processi per gestire in modo efficiente la scelta e il salvataggio dei profili giocatore attraverso l'utilizzo di varie classi. Questa scelta non solo agevola una migliore organizzazione dei dati, ma offre anche una maggiore fluidità e coerenza all'intera applicazione.

Classe: *Partita (lato gestione profili)*

Nel cuore dell'ecosistema da me progettato, ho posizionato la classe *Partita*, che ho ideato con l'obiettivo di rappresentare e amministrare una singola sessione di gioco. Nell'ottica di analizzare il ruolo della classe Partita nella gestione del profilo utente, mi preme considerare che durante il processo di inizializzazione, ho programmato *Partita* per invocare il metodo *.assegnaProfili()* di *ProfiloAssigner*. Questa operazione ha lo scopo di determinare e distribuire i profili corretti ai partecipanti. Al termine di ciascuna sessione, ho previsto che *Partita* faccia affidamento sul metodo *.aggiornaProfiliPostPartita(AbstractGiocatore vincitore)* per rielaborare e aggiornare le statistiche dei giocatori. Con questa decisione, intendo garantire una conservazione e gestione accurata delle informazioni, collaborando strettamente con *ProfiloManager*.

E' essenziale sottolineare come la classe *Partita*, sia stata concepita per interagire con la gestione dei profili sia all'apertura che alla conclusione della sua esecuzione.

Classe: *ProfiloAssigner*

Quando ho progettato la classe *ProfiloAssigner*, ho inteso creare un'entità responsabile dell'assegnazione dei profili. Essa rappresenta una soluzione essenziale al problema di attribuire profili distinti ai giocatori durante una partita. Ho adottato il pattern Singleton nella sua progettazione, assicurando l'unicità dell'istanza di questa classe nell'intero ambito dell'applicazione. Questa decisione architetturale è fondamentale per garantire coerenza nell'attribuzione dei profili e prevenire possibili sovrapposizioni o duplicazioni.

All'interno della classe, ho predisposto un elenco privato di profili, che funge da registro dinamico per i profili pronti per l'assegnazione, sia per i giocatori umani che per le entità gestite dal sistema (*GiocatoreCpu*). Questa lista, denominata "*profili*", contiene i profili disponibili. Inoltre, ho introdotto un oggetto *rand* della classe *Random* per la generazione di numeri casuali, elemento cruciale nell'assegnazione aleatoria dei profili.

Ho deliberatamente scelto di rendere privato il costruttore di *ProfiloAssigner* per enfatizzare l'adozione del pattern Singleton. Durante l'esecuzione di questo costruttore, l'istanza di *Random* viene inizializzata e tento di caricare i profili da un file esterno, "*profili.dat*". Inoltre, nel caso di problemi nella lettura del file, ho previsto la cattura di un'eccezione.

Un elemento chiave della classe è il metodo *.setProfiloGiocatoreUmano(Profilo profilo) (Figura 1)* attraverso il quale gestisco l'attribuzione di un profilo specifico al giocatore umano. Questa funzione non si limita all'assegnazione: interviene anche sull'aggiornamento dello stato di gioco e rimuove il profilo assegnato dalla lista disponibile.

```
public void setProfiloGiocatoreUmano(Profilo profilo) {
            // Assegna il profilo al giocatore umano tramite GameManager e
GiocatoreUmano
            GameManager.getInstance().getGiocatoreUmano().setProfilo(profilo);
            GiocatoreUmano.getInstance().setProfilo(profilo);

            // Aggiunge il profilo alla lista degli elenchi dei profili
disponibili
            StatoGioco.getInstance().aggiungiProfiloGiocatore(profilo);

            // Utilizza uno stream per rimuovere il profilo assegnato dalla
lista degli
            // elenchi dei profili disponibili
            profili = profili.stream().filter(p -> !p.equals(profilo)) // Filtra
i profili diversi dal profilo assegnato
                        .collect(Collectors.toList()); // Raccoglie i profili
filtrati in una nuova lista
      }
```

Figura 1 Metodo .setProfiloGiocatoreUmano()

Nel metodo *.assegnaProfili()*, ho affrontato le problemataiche relative all'attribuzione dei profili ai giocatori Cpu, ovvero i giocatori gestiti dal sistema. Ho prestato particolare attenzione alla verifica della disponibilità dei profili per evitare duplicazioni.

```
public void assegnaProfili() {
            Profilo profiloUmano =
GameManager.getInstance().getGiocatoreUmano().getProfilo();

            GameManager.getInstance().getGiocatoriCpu().forEach(giocatoreCpu ->
{
```

```
                if (profili.isEmpty()) {
                      throw new IllegalStateException("Non ci sono profili
disponibili da assegnare.");
                }

                Profilo profiloCasuale = prendiProfiloCasuale();
                while (profiloCasuale.equals(profiloUmano) ||
profiloEAssegnato(profiloCasuale)) {
                      profiloCasuale = prendiProfiloCasuale();
                }

                giocatoreCpu.setProfilo(profiloCasuale);

      StatoGioco.getInstance().aggiungiProfiloGiocatore(profiloCasuale);
                profili.remove(profiloCasuale);
            });
      }
```

Figura 2 Metodo: .assegnaProfili()

Ho enfatizzato la modularità del codice integrando metodi ausiliari come *.prendiProfiloCasuale()* e *.profiloEAssegnato(Profilo profilo)*. Il primo agevola la selezione casuale, mentre il secondo assicura l'assegnazione univoca controllando se il profilo è stato assegnato ad un *GiocatoreUmano* o ad un *GiocatoreCpu*.

Classe: *ProfiloManager*

Situandosi al centro tra la logica applicativa e le operazioni di salvataggio e caricamento dei dati, la classe ProfiloManager svolge un ruolo da protagonista nella gestione dei profili. Ha la responsabilità di implementare e sovrintendere a tutte le operazioni collegate ai profili. Metodi come .salvaProfilo() e .salvaTuttiIProfili() sono stati progettati per agevolare il salvataggio e l'aggiornamento dei dati dei profili. Questi metodi sfruttano la serializzazione di Java per salvare l'oggetto profilo sul disco. Analogamente, il metodo .caricaTuttiIProfili() è responsabile della deserializzazione e del ripristino della lista dei profili memorizzata sul disco.

Il metodo *.aggiornaESalvaProfilo()* verifica prima i dati esistenti di un profilo con lo stesso nome utente e, se trovato, lo aggiorna; in caso contrario, aggiunge il nuovo profilo alla lista. Questo assicura che i dati degli utenti siano sempre accurati e aggiornati.

E' da notare che la classe *ProfiloManager* non implementa l'interfaccia *Serializable*, ma utilizza comunque *.writeObjec()t* e *.readObject()* che sono metodi usati per serializzare e deserializzare oggetti. Tuttavia, non è la classe *ProfiloManager* stessa ad essere serializzata, ma gli oggetti di tipo *Profilo* che vengono passati ai suoi metodi.

La classe *ProfiloManager* sfrutta la serializzazione attraverso l'utilizzo degli oggetti *ObjectOutputStream* e *ObjectInputStream*, ma è la classe *Profilo* che deve implementare l'interfaccia *Serializable* per garantire che tutto funzioni correttamente.

Classe: *Profilo*

La classe *Profilo* conserva le informazioni, le specifiche e altri dettagli pertinenti al giocatore. Un aspetto distintivo di questa classe è la sua implementazione dell'interfaccia *Serializable*. Tale interfaccia, in Java permette un efficace processo di serializzazione, ovvero la conversione dell'oggetto in una sequenza di byte, e la successiva deserializzazione. Nel contesto in esame, *Profilo* subisce una serializzazione per garantire il salvataggio su disco e viene deserializzato quando richiesto, il tutto viene gestito da *ProfiloManager* come abbiamo visto nella parte precedente.

Interfaccia: *Serializable*

L'interfaccia *Serializable* è un'interfaccia marker, il che significa che non contiene metodi da implementare. La sua sola presenza in una classe indica al Java Virtual Machine (JVM) che gli oggetti di quella classe sono candidati per la serializzazione e deserializzazione.

```
public interface Serializable {
}
```

Quando un oggetto viene serializzato, non solo lo stato dell'oggetto (i valori dei suoi attributi) viene conservato, ma anche metadati sulla classe stessa, che consentono una deserializzazione efficace in un secondo momento.
La classe *ObjectOutputStream* e la classe *ObjectInputStream* sono rispettivamente responsabili della serializzazione e deserializzazione degli oggetti in Java. Quando si invoca il metodo *.writeObject()* su un'istanza di *ObjectOutputStream*, l'oggetto fornito come argomento viene serializzato, sempre che implementi l'interfaccia *Serializable*. Allo stesso modo, il metodo *.readObject()* di *ObjectInputStream* restituisce un oggetto deserializzato.
Java fornisce un meccanismo di versioning per la serializzazione attraverso il campo *serialVersionUID*. Questo identificativo unico per ogni classe permette al JVM di verificare la compatibilità tra la versione della classe utilizzata per la serializzazione e quella usata per la deserializzazione.

Classe: GestioneDati

La classe *GestioneDati* fornisce metodi per salvare e caricare liste di oggetti *Profilo* su e da un file. Ho sfruttato questa classe semplicemente per popolare il file profili.dat infatti non viene utilizzata all'interno del software durante la sua esecuzione.

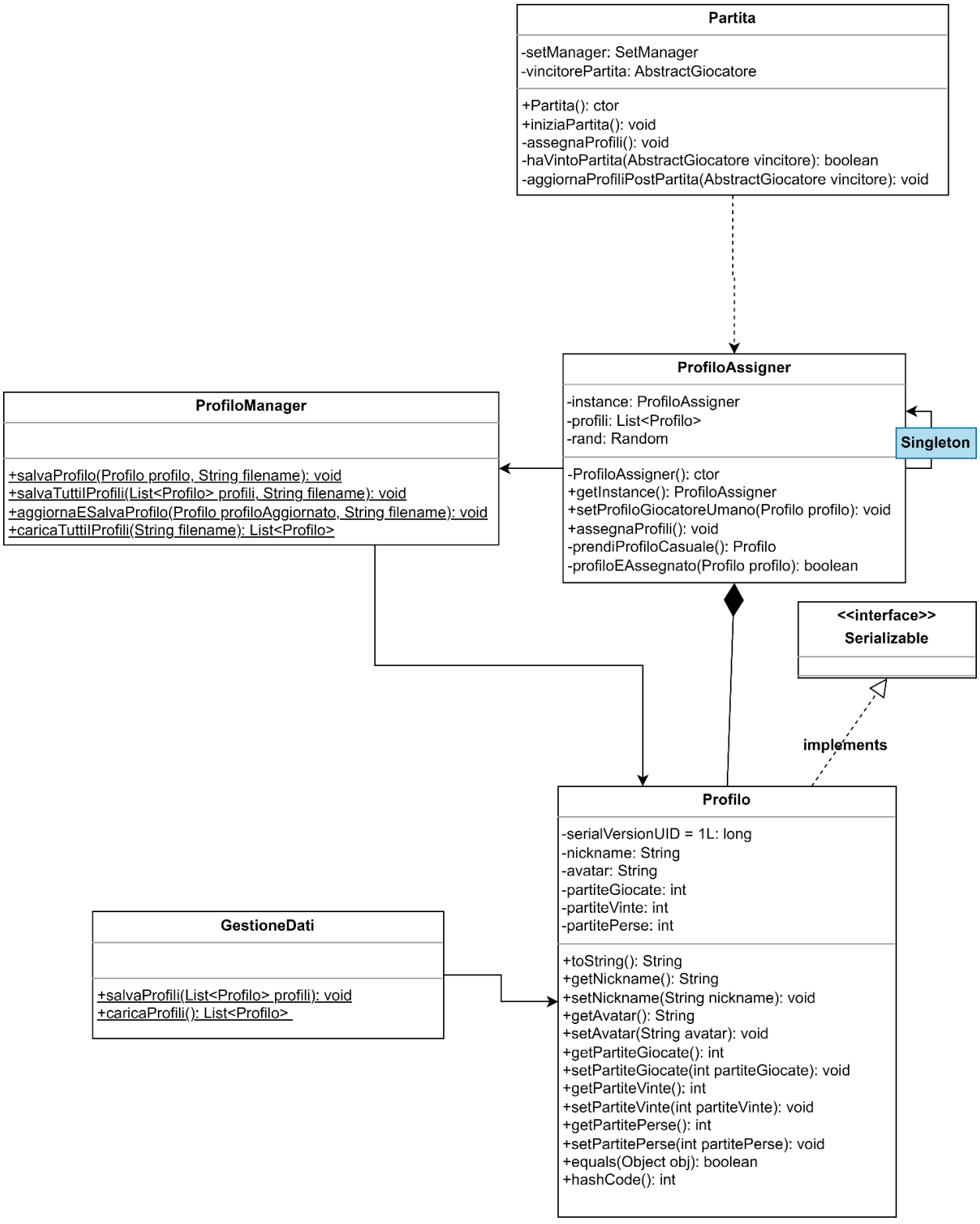

Figura 3 Diagramma UML Gestione Profili

Gestione Profilo Utente (dalla View)

Le classi della View interessate che si interessano alla gestione dei profili, alla selezione e alla consultazione sono le seguenti:

Classe: *StatistichePanel*

La classe *StatistichePanel* rappresenta un componente dell'interfaccia utente che si occupa di visualizzare le statistiche relative ai profili dei giocatori nel gioco.

Funzionalità Principali:
Visualizzazione dei Profili dei Giocatori: Questo pannello presenta le informazioni dei profili dei giocatori in un formato ordinato e strutturato. Le statistiche visualizzate includono l'avatar del giocatore, il nickname, il numero di partite giocate, vinte e perse.

La classe *StatistichePanel* riceve le informazioni sui profili dei giocatori attraverso il suo costruttore e le utilizza per popolare l'interfaccia grafica con le relative statistiche.

```
public StatistichePanel(List<Profilo> profili, String gifPath, ActionListener
backListener) {
…
}
```

Figura 4 Classe: StatistichePanel

La classe utilizza la classe *Profili* per generare l'interfaccia grafica.

All'interno del costruttore, il metodo *.creaSingoloPannelloGiocatore(Profilo profilo)* viene chiamato per ogni Profilo presente nella lista profili. Questo metodo utilizza le informazioni del Profilo fornito per creare e configurare un pannello individuale per quel giocatore:

```
for (Profilo profilo : profili) {
                    JPanel singleProfilePanel =
creaSingoloPannelloGiocatore(profilo);
                    profilesGridPanel.add(singleProfilePanel);
                }
```

Figura 5 Crea e configura per ogni Giocatore

La classe *StatistichePanel* riceve le informazioni sugli utenti direttamente attraverso il suo costruttore sotto forma di una lista di oggetti *Profilo*. Utilizza queste informazioni per popolare l'interfaccia grafica e mostrare le statistiche di ciascun giocatore in un formato ordinato.

Figura 6 Immagine schermata Statistiche Di Gioco

Classe: *SelezionaProfiloPanel*

Questa classe è stata creata per consentire all'utente di selezionare un profilo esistente dal database dei profili utente.

Figura 7 Immagine Selezione del Profilo

Funzionalità Principali:

Il costruttore della classe prende in ingresso una lista di profili (*List<Profilo>*), un percorso verso una gif da usare come sfondo e un *ActionListener* per gestire l'evento di conferma della selezione. Centralmente, possiede una *JComboBox* che elenca i profili disponibili e permette all'utente di effettuare la selezione. La visualizzazione di ciascun profilo all'interno della *JComboBox* è personalizzata grazie all'override del metodo *.getListCellRendererComponent()*, mostrando l'avatar e il nickname dell'utente. Una volta effettuata la selezione, l'utente può confermare la sua scelta attraverso un *JButton*, che, al momento della pressione, innesca una serie di operazioni, tra cui l'assegnazione del profilo scelto al giocatore e l'inizio di una nuova partita.

Figura 8 Schermata di Selezione del Profilo

GESTIONE PARTITA

Gestione PARTITA (dal Model)

Le carte

Dall'analisi delle carte del gioco **Trash** sorge evidente la presenza di differenti attributi che contraddistinguno i differenti tipi di carte. Il diagramma UML nella figura della pagina successiva mostra le relazioni che intercorrono tra le diverse entità software che prendono parte nella realizzazione di un oggetto *Carta*. La classe *Carta,* che implementa l'interfaccia *InterfaceAction*, definisce il modello costitutivo di una carta all'interno del programma. Gli attributi di una carta del gioco **Trash** sono i seguenti:

- Attributo "*InterfaceAction*": stabilisce l'azione che la carta può esibire.
- Attibuto "*Carte*": determina il valore e il seme della carta.
- Attributo "*Scoperta*": determina se la carta è scoperta o coperta.

L'attibuto *InterfaceAction* viene impostato dal metodo *.setInterfaceAction* della classe *Carta*; questo metodo viene chiamato nel metodo costruttore della classe Carta per impostare 3 tipi di Azione: *AzioneMatch*, *AzioneScarto* e *AzioneJolly*.

L'attributo "Carte" è una enumerazione che accoppia i valori di altre due enumerazioni che sono *Semi* e *Valori*. Ogni carta quindi è composta dalla coppia *Semi Valori* che permette in fase di creazione di stabilire quale *InterfaceAction* assegnare alla carta.

```
/**
       * Imposta l'azione della carta in base al suo valore.
       */
      public void setInterfaceAction() {
            // Creazione di set di valori per le diverse azioni
            Set<Valori> matchValori = new HashSet<>(Arrays.asList(Valori.ASSO,
Valori.DUE, Valori.TRE, Valori.QUATTRO,
                        Valori.CINQUE, Valori.SEI, Valori.SETTE, Valori.OTTO,
Valori.NOVE, Valori.DIECI));
            Set<Valori> scartoValori = new HashSet<>(Arrays.asList(Valori.JACK,
Valori.QUEEN));

            // Recupero del valore della carta
            Valori cartaValori = carte.getValore();

            // Impostazione dell'azione basandosi sul valore della carta
            if (matchValori.contains(cartaValori)) {
                  azione = new AzioneMatch();
            } else if (scartoValori.contains(cartaValori)) {
                  azione = new AzioneScarto();
            } else {
                  azione = new AzioneJolly();
            }
      }
```

Figura 9 Metodo: setInterfaceAction()

Per quanto riguarda l'attributo "Scoperta", le carte quando vengono generate vengono tutte impostate *this.scoperta = false;* di default in modo che siano tutte coperte all'inizio del gioco.

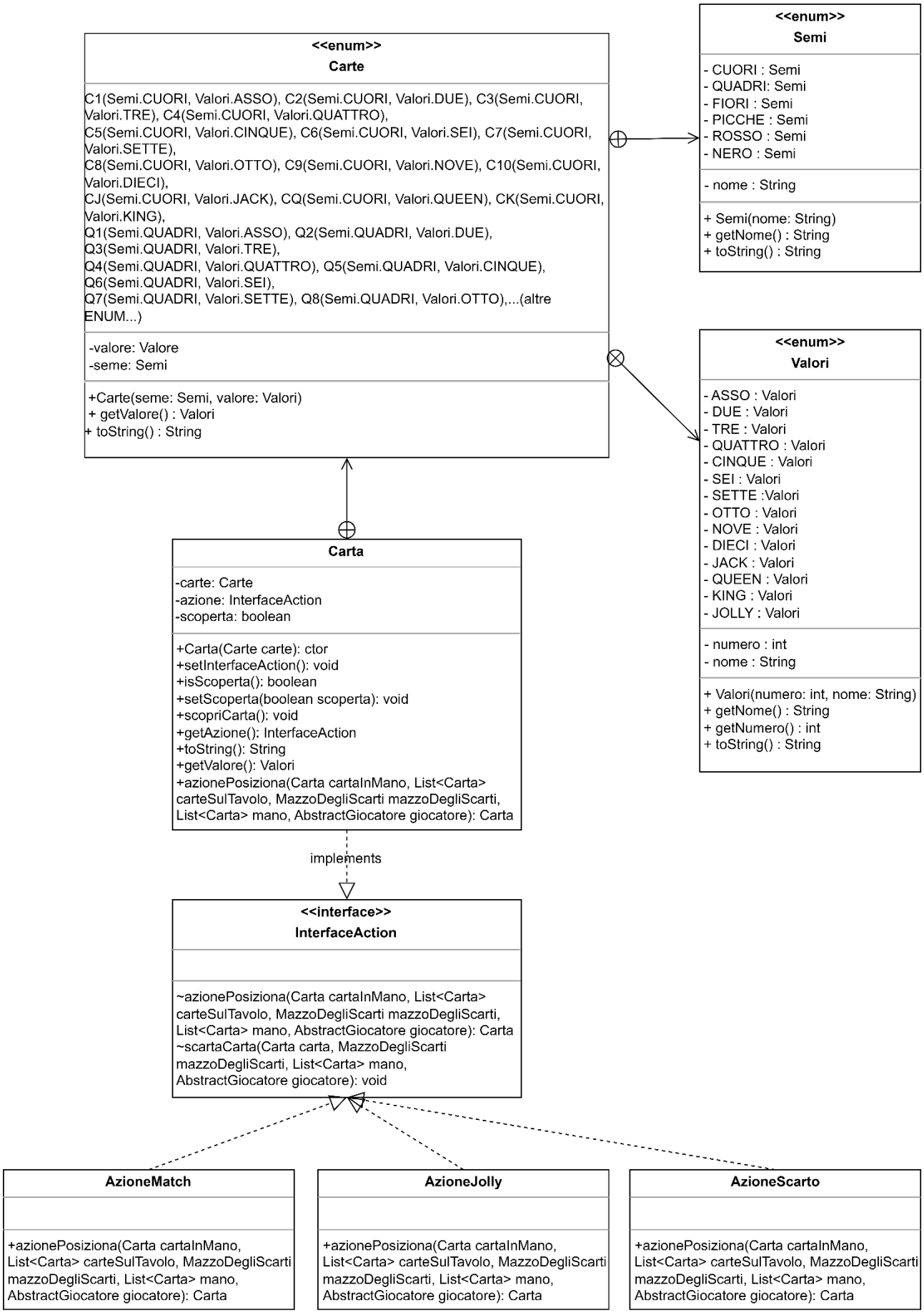

Figura 10 Diagramma UML Gestione Carta

La Partita

Una partita viene inizializzata dal *Controller* che ha una classe *EseguiPartita* che serve appunto a iniziare una partita.
Al centro di questo ecosistema di gioco troviamo la classe *GameManager*, che è l'amministratore del gioco. L'adozione del pattern Singleton da parte di questa classe fa in modo che, in ogni momento, vi sia soltanto una singola istanza operativa di essa. Una delle sue funzioni di spicco, *.inizializzaGioco()*, non solo si occupa della determinazione del numero di giocatori CPU, ma anche della creazione e inizializzazione del mazzo di carte e dell'istanziazione dei giocatori. Inoltre, la classe fornisce metodi essenziali per la gestione e l'accesso ai giocatori e alle carte.

Parallelamente, la classe *NumeroGiocatoriManager* svolge la funzione di regolare il numero dei giocatori CPU. Adottando anch'essa il pattern Singleton, siamo sicuri che il numero dei giocatori CPU sia sempre limitato tra 1 e 3, interfacciandosi con *StatoGioco* per aggiornare lo stato attuale della partita.

Avanzando nella struttura, la classe *Partita* incarna una singola sessione di gioco. Essa si occupa dell'assegnazione e gestione dei profili dei giocatori come abbiamo detto nelle pagine precedenti, e coordina anche l'intero ciclo di gioco. All'interno di questo ciclo, la classe *SetManager* assume la responsabilità di gestire un singolo set, organizzando le carte, avviando e concludendo ogni set, e individuando i vincitori.

Ogni set si conclude con la determinazione di uno o più vincitori, i quali vedono il numero delle loro carte iniziali ridotto di uno. La partita prosegue fino a quando un giocatore raggiunge una condizione di vittoria, che in questo caso è avere una sola carta iniziale rimasta.

```
private boolean haVintoPartita(AbstractGiocatore vincitore) {
        return vincitore.getCarteIniziali() == 1;
    }
```

Ai fini della esecuzione di una partita conviene porre l'attenzione sui seguenti metodi di *SetManager*:

Metodo: *iniziaSet()*

Si tratta di un punto di inizializzazione critico per ogni set di gioco. Si evidenzia un'interazione strettamente accoppiata con *StatoGioco* e *GameManager*, entrambi fondamentali per l'istanziazione corretta delle variabili di gioco e l'inizializzazione dei giocatori. Questo metodo fa in modo che i giocatori inizino il gioco.

Il metodo .*iniziaGioco()* sia di *GiocatoreUmano* che di *GiocatoreCpu* (il metodo viene overridato) serve come punto di partenza per un turno o una sessione di gioco del giocatore. Quando chiamato, questo metodo assicura che tutte le carte sul tavolo siano rimosse, fornendo così una "tavola pulita" per iniziare. Successivamente, il giocatore pesca un certo numero di carte dal suo mazzo, corrispondente al numero delle sue carte iniziali. Queste carte vengono poi posizionate sul tavolo.

Metodo: *getGiocatori()*

E’ il metodo che serve per ottenere una lista completa di entità giocatore, amalgamando sia il giocatore umano sia le entità CPU. Questo metodo esemplifica un approccio di astrazione, trattando ogni giocatore come un'entità di tipo *AbstractGiocatore*.

Metodo: *eseguiSet()*

Questo metodo riveste una particolare rilevanza, in quanto rappresenta la logica centrale del flusso di gioco per un intero set. Esso integra cicli, condizioni e invocazioni esterne in un’unica funzione.

La logica dietro .*eseguiSet()* è la seguente:

- Un ciclo *while* continua finché non vengono determinati dei vincitori o finché tutti i giocatori non hanno completato il loro turno finale.
- Durante ogni iterazione del ciclo, il metodo itera su tutti i giocatori. Per ogni giocatore:
  - Se un vincitore non è stato ancora trovato o se il giocatore è ancora nella lista *giocatoriFinalTurn*, il giocatore esegue il suo turno.
  - Se durante il turno di un giocatore si determina che è un vincitore e non è stato ancora identificato un vincitore, il metodo segna che un vincitore è stato trovato e riproduce un suono (la voce di un uomo che dice “Trash”).
  - Se un vincitore è stato trovato, il giocatore viene rimosso dalla lista *giocatoriFinalTurn*.
  - Una volta che tutti i giocatori hanno completato il loro turno finale, i vincitori vengono determinati tramite .*determinaVincitoriSet().*

Metodo: .*giocatoreVincitore(AbstractGiocatore giocatore)*

Opera come un verificatore condizionale, determinando se un giocatore ha soddisfatto le condizioni per essere dichiarato vincitore. L'uso del paradigma di programmazione funzionale, attraverso l'invocazione di streams, rende l'operazione concisa e chiara.

Metodo: .*determinaVincitoriSet()*

Altra manifestazione dell'utilizzo del paradigma funzionale, questo metodo impiega le stream per elencare i vincitori del set attuale, fornendo un esempio di come le operazioni su insiemi di dati possano essere realizzate in maniera dichiarativa e concisa.

Classe: *Turno*

Ogni set, poi, si compone di turni distinti, e qui entra in gioco la classe *Turno*. Essa gestisce le azioni di un singolo giocatore durante un set, dall'azione di pescare carte alle azioni specifiche legate ad ogni singola carta.

Quando ho pensato la classe *Turno* il mio scopo era quello di creare una classe che potesse gestire il flusso di un turno di gioco.

Campo: *giocatoreCorrente*

Un'istanza di *AbstractGiocatore* rappresentante il giocatore il cui turno è attualmente in corso. Ho deciso di utilizzare una classe astratta in modo che diversi tipi di giocatori possano esistere (*GiocatoreUmano* e *GiocatoreCpu*), permettendo una maggiore flessibilità nell'implementazione di varie strategie o comportamenti di gioco.

Campo: *haScartato*

Una variabile booleana che funge da segnaposto per tracciare se il giocatore ha scartato una carta durante il suo turno.

Metodo: .*eseguiTurno()*

Rappresenta il cuore della logica del turno. Esso inizializza la variabile *haScartato* a *false* e poi avvia una serie di azioni sequenziali:

1. Verifica quale carta si trova in cima al mazzo degli scarti.
2. Decide se il giocatore dovrebbe gestire una carta specifica o pescare una nuova carta dal mazzo.
3. Processa la mano del giocatore, eseguendo le azioni delle carte in essa contenute.

Metodo: .*gestisciPescaGiocatore()*

Questa funzione decide se il giocatore corrente dovrebbe gestire una carta specifica o pescare una carta. Ciò dipende dalla natura della carta in cima al mazzo degli scarti. Se tale carta non richiede alcuna azione di scarto, il giocatore decide da quale mazzo pescare. In caso contrario, il giocatore pesca direttamente.

Metodo: .*processaManoGiocatore()*

Qui, la classe itera attraverso la mano del giocatore fino a quando non ha scartato una carta o finché ha carte disponibili. Durante ogni iterazione, recupera l'ultima carta dalla mano del giocatore e ne gestisce l'azione. Questa è una rappresentazione chiara del flusso del gioco, dove ogni carta può avere un'azione associata che influisce sullo stato del gioco.

Infine, non possiamo trascurare le classi *MazzoDegliScarti* e *MazzoDiCarte*, fondamentali per la manipolazione e la gestione delle carte all'interno dei mazzi. Entrambe le classi sono dotate di metodi specifici per l'aggiunta, la rimozione e il controllo delle carte, assicurando una gestione delle carte fluida e senza interruzioni.

UTILIZZO DI MODEL-VIEW-CONTROLLER E OBSERVER-OBSERVABLE

JTrash sfrutta il pattern architetturale Model-View-Controller.

Il pattern MVC (MODEL – VIEW – CONTROLLER) è un Pattern Architetturale che ci permette di progettare l'architettura del software dividendo il codice in una parte che gestisce il Modello, una parte che gestisce la View e una parte che gestisce il Controller.

Il Model è il "motore" del nostro software, rappresenta i dati e li utilizza. Al suo interno troviamo tutta la logica di creazione e memorizzazione dei dati. Inoltre si occupa di notificare alla View quando i suoi dati cambiano in modo che la visualizzazione possa essere aggiornata. Rappresenta la entità del mio problema e ha un altissimo riuso.

La View è ciò che viene utilizzato per poter visualizzare i dati (ad esempio una GUI), basso riuso perché la view va ogni volta ricreata in base al sistema dove verrà eseguito il software.

Il Controller in questo tipo di pattern è un package che dipende da Model e da View, fa da "coordinatore".

Per una corretta implementazione, il Model non deve contenere riferimenti alla View e sarebbe consigliabile farlo il più possibile anche nell'altro senso; il pattern che può aiutarci a dividere queste entità si chiama Observer-Observable dove il Model estende la classe Observable e la View implementa l'interfaccia Observer.

Allego il diagramma UML del mio progetto visto a distanza, per capire come queste entità sono separate per rispettare il pattern architetturale MVC

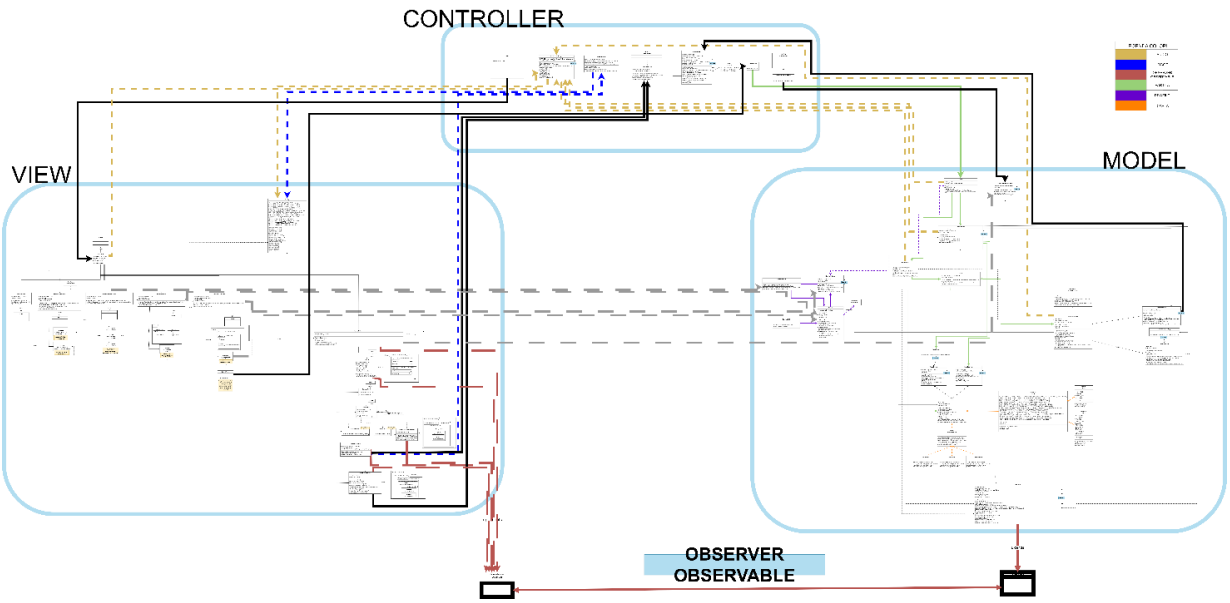

Figura 11 Diagramma UML Completo

I collegamenti diretti più “forti” sono fra Model e Controller , e fra View e Controller. Model e View comunicano fra loro tramite pattern Observer-Observable.
L’unico punto del codice dove la View usa direttamente il Model me lo sono concesso nella parte dove la View utilizza i profili dei giocatori, per semplificare il codice ho valutato che questa fosse un’opzione utilizzabile direttamente.

Per quanto riguarda il pattern Observer-Observable, durante lo sviluppo mi sono accorto che sarebbe stato molto complesso avere molte classi osservabili, quindi per semplificare la progettazione, all’interno del Model mi sono creato una classe *StatoGioco* che ha lo scopo di rappresentare lo stato del gioco in tempo reale. All’interno di essa ho incluso tutti i campi del model che voglio che vengano osservati dalla View per aggiornarsi. Infatti durante la esecuzione del software quando nel model viene eseguita una operazione, spesso includo anche i cambiamenti che vanno apportati su *StatoGioco*. Ogni volta che lo stato del gioco cambia, gli *Observer* (o meglio, le classi nella vista che implementano *Observer*) vengono notificati. Ad esempio, quando la mano del giocatore umano cambia (*.setManoGiocatoreUmano(Carta manoGiocatoreUmano)*), dopo aver impostato la nuova carta, gli observer vengono notificati attraverso l'invocazione dei metodi *.setChanged()* e *.notifyObservers(Object arg).* Questo schema è adottato per diversi metodi, garantendo che ogni modifica al modello rifletta immediatamente nella vista.
MazzoEScartiPanel , *GiocatoreButtons* , *ManoGiocatoreUmanoPanel* , *SideBar* , e *TavoloDiGioco* sono le classi della View che implementano l'interfaccia Observer, permettendo loro di reagire ai cambiamenti di stato in *StatoGioco*. Queste classi monitorano specificamente le variazioni dello stato del gioco e aggiornano i componenti dell'interfaccia utente di conseguenza tramite il metodo *.update().* Ad esempio, se la mano del giocatore cambia, una delle classi vista potrebbe aggiornare la visualizzazione delle carte del giocatore.

UTILIZZO DI ALTRI DESIGN PATTERN

STRATEGY PATTERN

Strategy Pattern per la gestione delle azioni di Carta

Lo Strategy Pattern è un design pattern comportamentale che permette di definire una famiglia di algoritmi, incapsularli individualmente e renderli intercambiabili. Esso consente agli algoritmi di variare indipendentemente dai clienti che ne fanno uso.
L'obiettivo principale dello Strategy Pattern è quello di isolare gli algoritmi dalle classi che li utilizzano, fornendo modalità alternative per eseguire un'azione. In termini pratici, il pattern si compone di una famiglia di classi (strategie), ciascuna delle quali rappresenta un'implementazione diversa di un algoritmo. Tali algoritmi possono essere cambiati dinamicamente a runtime.
Nella classe *Carta*, lo Strategy Pattern è implementato tramite l'interfaccia *InterfaceAction*, che rappresenta l'azione che una carta può eseguire. L'interfaccia definisce un insieme di metodi che tutte le classi che la implementano devono fornire. Questa interfaccia viene poi implementata da diverse classi concrete, in particolare *AzioneMatch*, *AzioneScarto*, e *AzioneJolly*.
La classe *Carta* fa uso dello Strategy Pattern attraverso il campo *private InterfaceAction azione*;. Questo campo determina l'azione specifica che una *Carta* può eseguire. La funzione .*setInterfaceAction()* nella classe *Carta* è incaricata di definire quale azione (o strategia) deve essere associata a una particolare `Carta` in base al suo valore.

La logica all'interno di .*setInterfaceAction()* utilizza dei set per raggruppare diversi valori di carte e associarli a un'azione specifica. Ad esempio, se il valore della carta appartiene al set *matchValori*, all'oggetto *Carta* viene associata l'azione *AzioneMatch*.

Il vantaggio di questo approccio è che permette di estendere facilmente le azioni disponibili per le carte senza dover modificare la classe *Carta* stessa. Se in futuro si desidera aggiungere una nuova azione, basta creare una nuova classe che implementa *InterfaceAction* e aggiornare la logica in .*setInterfaceAction().*

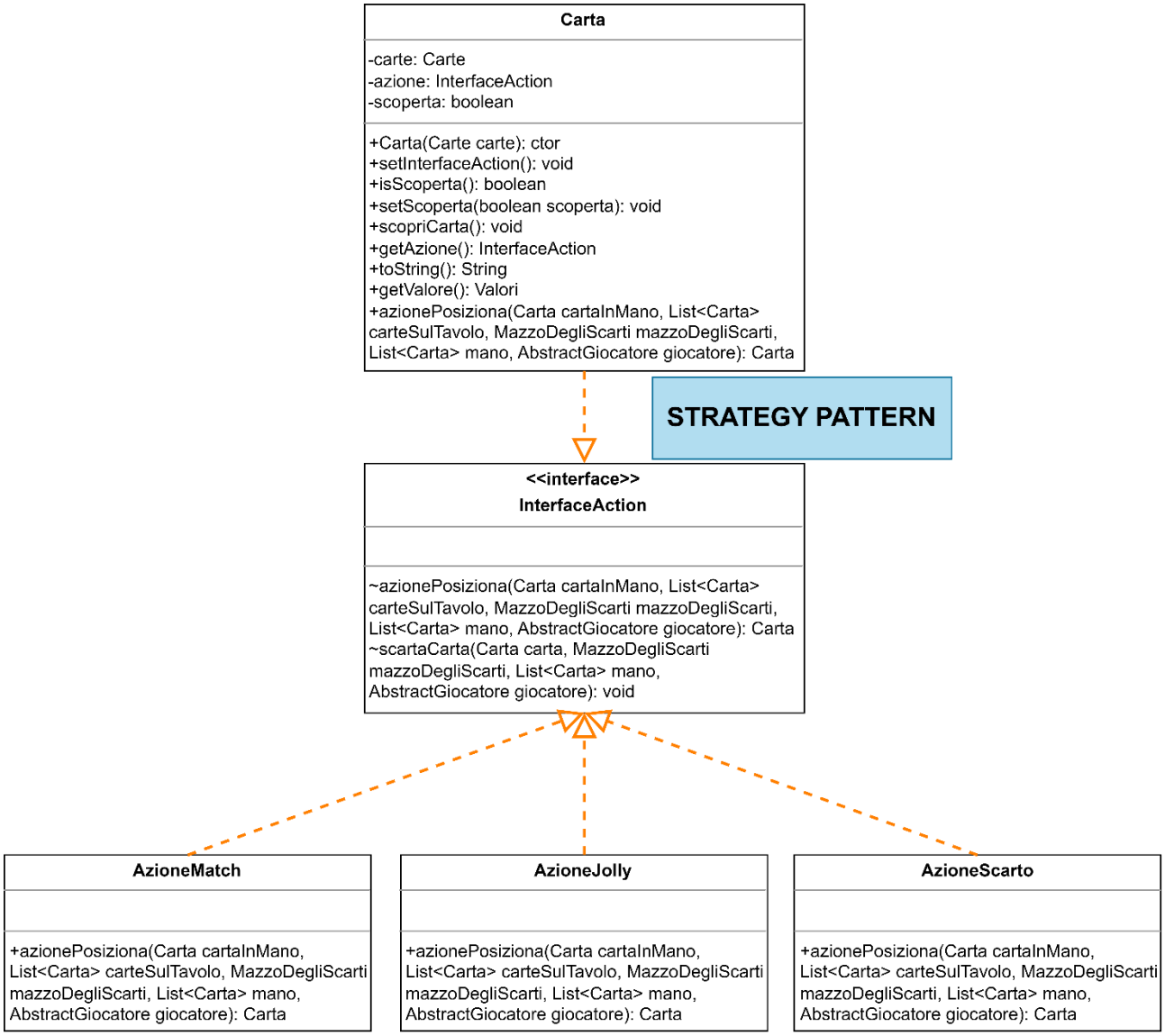

Figura 12 Diagramma UML Strategy Pattern

Attraverso l'uso dell'interfaccia *InterfaceAction* e delle relative classi di implementazione, la classe *Carta* può facilmente cambiare il suo comportamento a runtime in base al valore della carta, offrendo così una flessibilità notevole.

FACTORY PATTERN

Factory Pattern per la creazione di MazzoDiCarte

In ambito di programmazione orientata agli oggetti, il Factory Pattern è uno dei design pattern creazionali che mira a fornire un mezzo per la creazione di oggetti, evitando la necessità di specificare la classe esatta dell'oggetto che verrà creato. Questo processo viene delegato a sottoclassi specifiche o a metodi statici che decidono quale oggetto creare. Questo pattern astrae la logica di creazione, consentendo al codice chiamante di rimanere indipendente dalla logica di inizializzazione e rappresentazione.

L'implementazione tipica di un Factory Pattern include una classe factory, che contiene metodi per creare oggetti delle classi concrete.

Ho deciso di utilizzare questo pattern nell'implementazione del mazzo di carte (classe *MazzoDiCarte*) e nella sua factory associata (classe *MazzoFactory*).

MazzoFactory svolge il ruolo da protagonista nel Factory Pattern. La sua funzione principale è *creaMazzo(int numeroDiGiocatoriCpu),* che si occupa di creare e inizializzare un'istanza di *MazzoDiCarte* in base al numero di giocatori CPU fornito. Il metodo utilizza l'unicità del Singleton *MazzoDiCarte* per ottenere un'istanza e poi la inizializza in base ai requisiti.

Nello specifico

```
public class MazzoFactory {

	/**
	 * Crea un mazzo di carte basandosi sul numero di giocatori CPU fornito.
	 *
	 * @param numeroDiGiocatoriCpu Il numero di giocatori CPU per decidere
	 *                             l'inizializzazione del mazzo.
	 * @return Un'istanza del mazzo di carte, gia' inizializzata in base al
numero
	 *         di giocatori CPU.
	 */
	public static MazzoDiCarte creaMazzo(int numeroDiGiocatoriCpu) {
		MazzoDiCarte mazzoDiCarte = MazzoDiCarte.getInstance();
		mazzoDiCarte.inizializza(numeroDiGiocatoriCpu);
		return mazzoDiCarte;
	}
}
```

Figura 13 Classe MazzoFactory

Mostro anche il diagramma UML della parte dove ho gestito questo pattern:

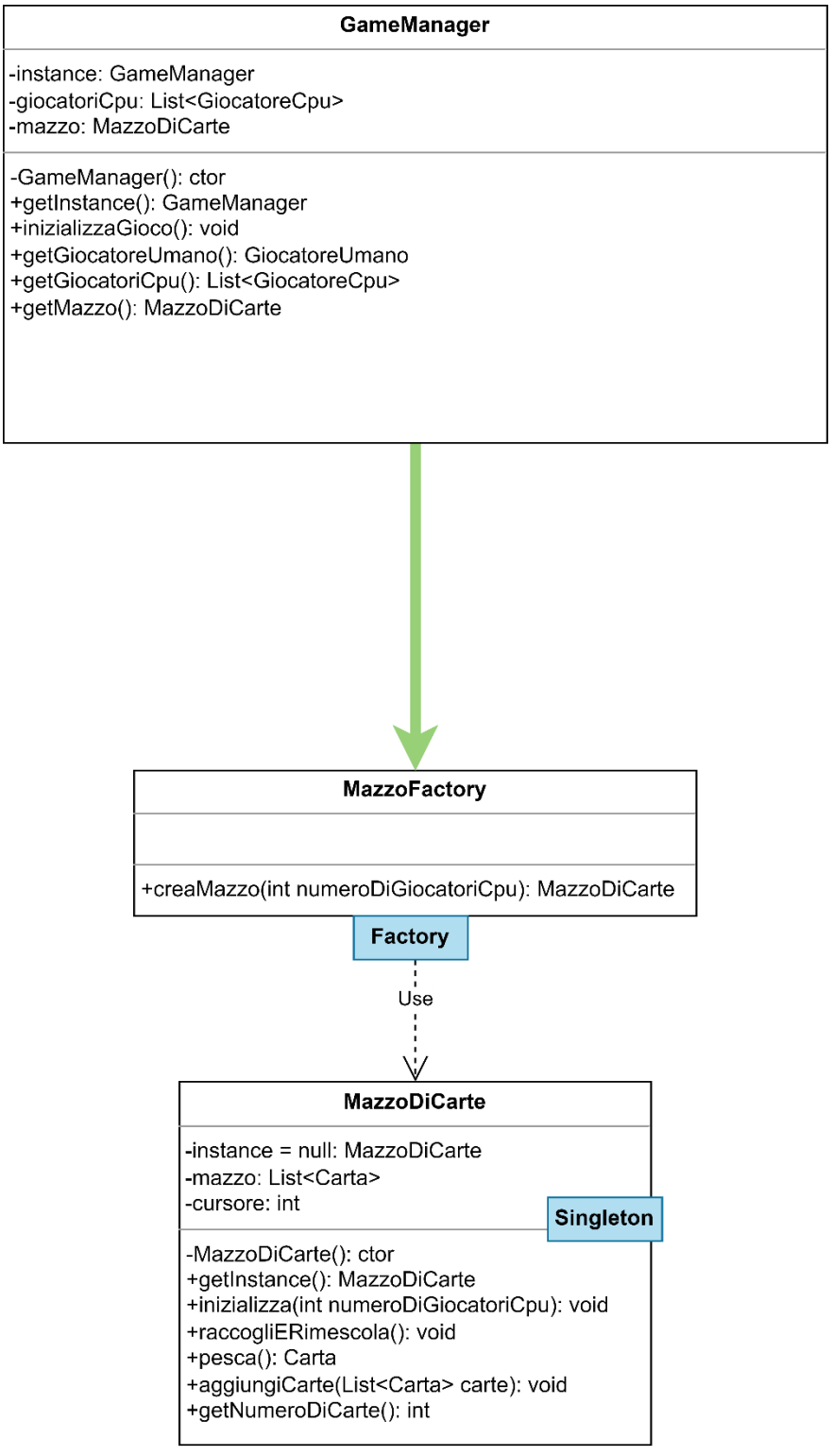

Figura 14 Diagramma UML Factory Pattern

Factory Pattern per la creazione di *Giocatore Cpu*

Ho utilizzato lo stesso tipo di Pattern anche per la generazione di *GiocatoreCpu*.

La classe *GiocatoreCpu* estende *AbstractGiocatore* e rappresenta un giocatore gestito dalla CPU. Essa contiene diversi metodi che permettono al giocatore di iniziare un gioco, pescare carte, decidere da quale mazzo pescare, gestire le azioni delle carte e molto altro.

Il costruttore di *GiocatoreCpu* accetta un parametro *indice* che rappresenta l'indice del giocatore CPU. Questo indice è particolarmente significativo perché distingue i diversi giocatori CPU tra loro.

La classe *GiocatoreCpuFactory* è la classe al centro dell'implementazione del Factory Pattern. Questa classe contiene un metodo statico *.creaGiocatoriCpu()* che accetta un parametro *numeroCpu*, rappresentando il numero di giocatori CPU che si desidera creare.

```
public class GiocatoreCpuFactory {

	/**
	 * Crea una lista di giocatori CPU in base al numero specificato.
	 *
	 * @param numeroCpu Il numero di giocatori CPU da creare.
	 * @return Una lista di giocatori CPU.
	 * @throws IllegalArgumentException Se il numero di CPU specificato e'
inferiore
	 *                                  a 1 o superiore a 3.
	 */
	public static List<GiocatoreCpu> creaGiocatoriCpu(int numeroCpu) {
		if (numeroCpu < 1 || numeroCpu > 3) {
			throw new IllegalArgumentException("Numero di CPU non
valido");
		}

		List<GiocatoreCpu> giocatoriCpu = new ArrayList<>();
		for (int i = 0; i < numeroCpu; i++) {
			giocatoriCpu.add(new GiocatoreCpu(i + 1)); // Passa l'indice
come parametro
		}

		return giocatoriCpu;
	}
}
```

Figura 15 Classe: GiocatoreCpuFactory

Il metodo inizia con un controllo di validità per assicurarsi che il numero di giocatori CPU richiesti sia compreso tra 1 e 3. Successivamente, itera per il numero specificato, creando

nuovi oggetti *GiocatoreCpu* e assegnando loro un indice incrementale. Alla fine, restituisce una lista di oggetti *GiocatoreCpu*.

Mostro anche il diagramma UML della parte dove ho gestito questo pattern:

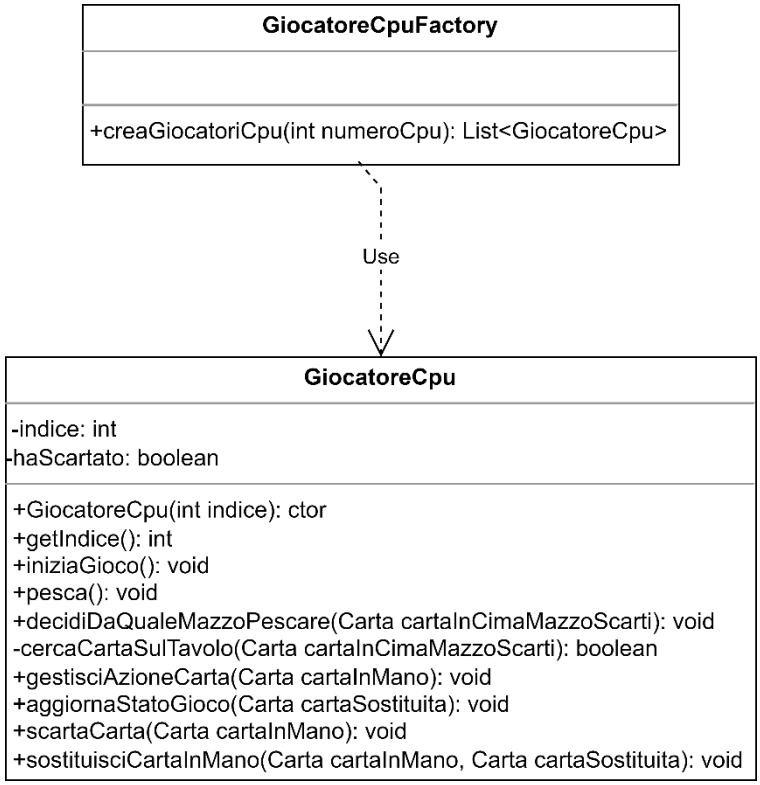

Figura 16 Diagramma UML Factory Pattern

SINGLETON PATTERN

Il Singleton Pattern è un pattern che viene utilizzato ogni volta che abbiamo l'esigenza di avere una sola istanza di una determinata classe, e fornire un modo per accedere a tale istanza da qualsiasi punto del codice. La necessità di tale pattern può emergere in vari contesti, come ad esempio quando si ha a che fare con risorse globali o condivise, come connessioni a database, configurazioni di sistema o, come nel nostro caso, rappresentazioni uniche di determinate entità nel contesto di un gioco.

Ho usato questo pattern in varie classi:
-*GiocatoreUmano*
-*SimpleAudioManager*
-*InputManager*
-*NumeroGiocatoriManager*
-*GameManager*
-*ProfiloAssigner*
-*MazzoDiCarte*
-*MazzoDegliScarti*
-*StatoGioco*

Al fine di relazionare il modo in cui l'ho utilizzato, prenderò in esame solo una classe, in particolare la classe *GiocatoreUmano*.

Singleton Pattern per *GiocatoreUmano* (esempio)

La classe *GiocatoreUmano* implementa il pattern Singleton attraverso i seguenti passi:
1. Istanza privata statica: L'attributo privato statico *instance* è inizializzato come *null*. Questo attributo conterrà la singola istanza della classe.

```
private static GiocatoreUmano instance = null;
```

Figura 17 Campo: instance = null

2. Costruttore privato: Il costruttore della classe è reso privato. Questo impedisce la creazione di nuove istanze di *GiocatoreUmano* al di fuori della classe stessa.

```
/**
       * Costruttore privato per il Singleton.
       */
      private GiocatoreUmano() {
            super();
      }
```

Figura 18 Costruttore privato Giocatore Umano

3. Metodo di accesso: La classe fornisce un metodo pubblico statico *.getInstance()*, che consente di ottenere l'istanza unica di *GiocatoreUmano*. Se tale istanza non esiste (ovvero *instance* è *null*), viene creata; altrimenti, viene semplicemente restituita.

```
/**
	 * Ottieni l'istanza Singleton del giocatore umano.
	 *
	 * @return L'istanza Singleton del giocatore umano.
	 */
	public static GiocatoreUmano getInstance() {
		if (instance == null) {
			instance = new GiocatoreUmano();
		}
		return instance;
	}
```

Figura 19 Implementazione Singleton Pattern

All'interno del gioco, può esistere un solo giocatore umano attivo per sessione di gioco. Pertanto, l'utilizzo del pattern Singleton garantisce che non vengano create istanze multiple del giocatore umano, evitando ambiguità e potenziali errori di implementazione.

Il Singleton fornisce inoltre un punto di accesso globale all'istanza, garantendo che tutte le parti del codice che necessitano di interagire con il giocatore umano possano farlo attraverso un riferimento univoco e condiviso.
Questa scelta progettuale assicura l'unicità dell'entità del giocatore umano durante l'esecuzione del gioco JTrash.

Mostro anche il diagramma UML della parte dove ho gestito questo pattern:

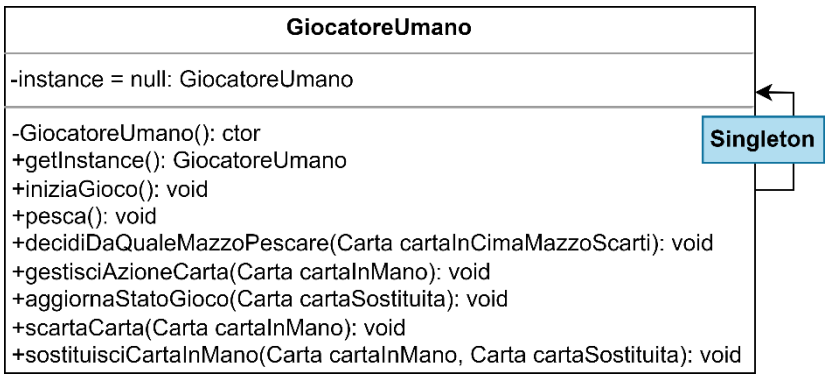

Figura 20 Diagramma UML Singleton Pattern

Utilizzo di JAVA *SWING* per la GUI

Java *Swing* è una potente libreria per la creazione di interfacce grafiche in Java. Essa fornisce una vasta gamma di componenti pre-costruiti che possono essere facilmente integrati in un'applicazione Java per realizzare interfacce utente ricche e interattive.
Il Frame principale della mia applicazione è la classe *MainFrame*

Classe: *MainFrame*

La classe *MainFrame* estende *JFrame*, che rappresenta la finestra principale dell'applicazione. Questo è il contenitore principale dove vengono aggiunti e visualizzati tutti gli altri componenti grafici.

CardLayout e Pannelli

Un concetto fondamentale nella progettazione dell'interfaccia è l'uso di *CardLayout*. Questo layout manager permette di alternare la visualizzazione di diversi pannelli (*JPanel*) all'interno di un unico contenitore. Praticamente *CardLayout* permettere di gestire la visualizzazione di componenti come fossero “carte” in un mazzo. *CardLayout* permette di avere diversi pannelli in uno stesso spazio e di mostrarne solo uno alla volta.

Creo un *CardLayout* e lo associo al container *JPanel cardsPanel*:

```
public MainFrame() {
            setTitle(Config.getTitolo()); // Imposta il titolo del frame.
            cardLayout = new CardLayout(); // Inizializzazione del CardLayout e
del pannello principale.
            cardsPanel = new JPanel(cardLayout);
```

Figura 21 Creazione CardLayout e associazione

Si aggiunge i componenti al container:

```
// Imposta il pannello GIF come pannello visibile iniziale.
            cardsPanel.add(gifInizialePanel, "GIF_PANEL");
            cardsPanel.add(gifBackgroundPanel, "BACKGROUD_PANEL");
            cardsPanel.add(selezioneGiocatoriPanel, "PLAYER_SELECTION_PANEL");
```

Figura 22 Aggiunta componenti al cardsPanel

poi si utilizza il metodo *.show()* per mostrare una carta specifica (Esempio):

```
cardLayout.show(cardsPanel, "PROFILE_SELECTION_PANEL");
```

Figura 23 Metodo: .show()

La variabile *cardLayout* gestisce l'alternanza tra i diversi pannelli e *cardsPanel* è il pannello principale che contiene questi pannelli secondari.
L'applicazione inizia con la visualizzazione di un pannello contenente una GIF, gestito dalla classe *GifInizialePanel*. Una volta terminata la riproduzione della GIF, grazie all'uso di un timer, si passa al pannello di sfondo, *GifBackgroundPanel*.

Un altro elemento chiave dell'interfaccia è la selezione dei giocatori e dei profili. *NumeroGiocatoriController* gestisce la logica per la selezione del numero dei giocatori. Successivamente, il pannello *SelezioneGiocatoriPanel* consente all'utente di selezionare i giocatori, che poi porta alla selezione dei profili attraverso il pannello *SelezioneProfiloPanel*.

Screenshot schermata iniziale:

Figura 24 Immagine schermata iniziale

Struttura GUI

Struttura generale dell’interfaccia grafica e le sue classi

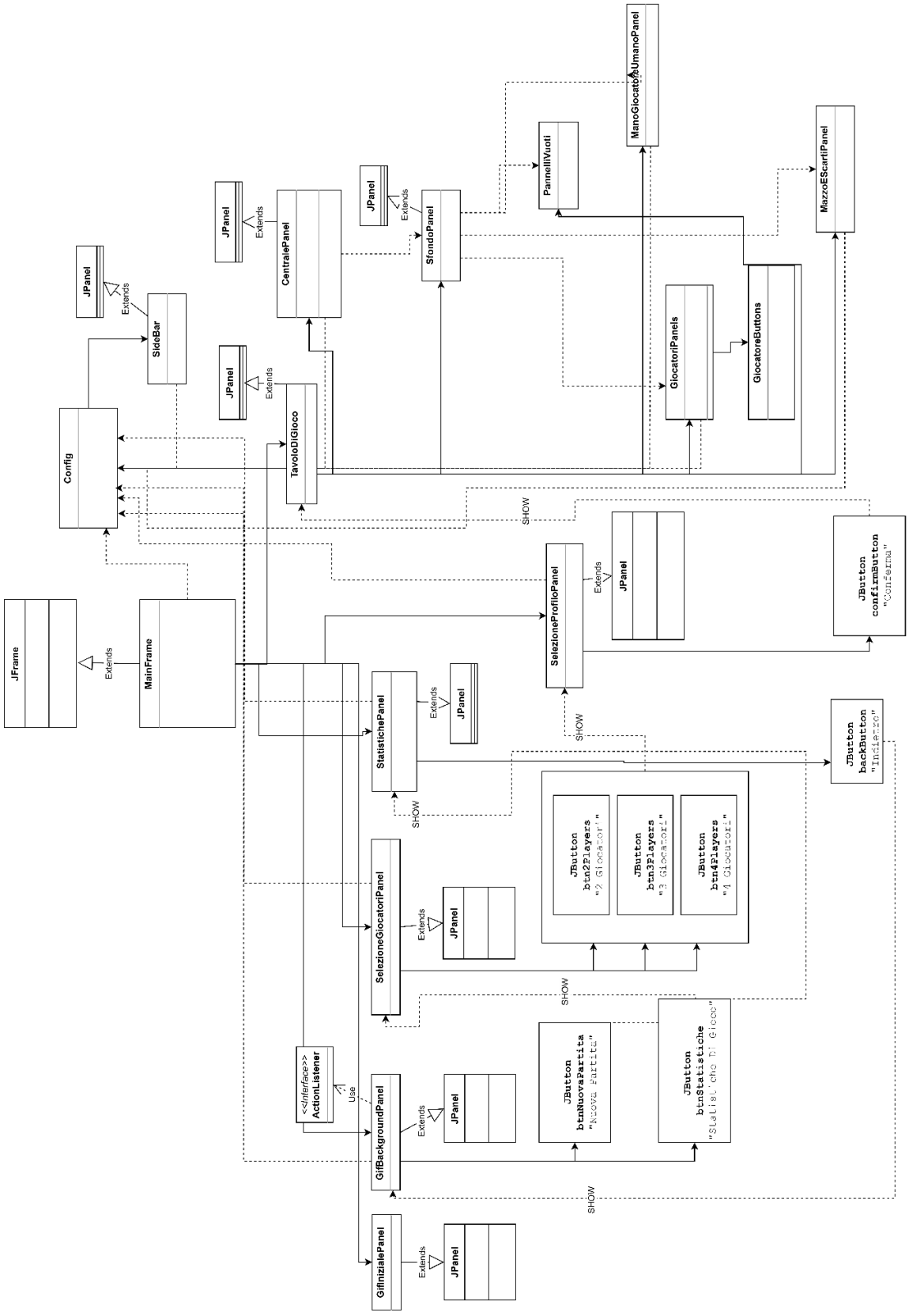

Figura 25 Diagramma UML Struttura View

DiagrammaUML che riguarda solo il menù iniziale e le selezioni di impostazione di gioco:

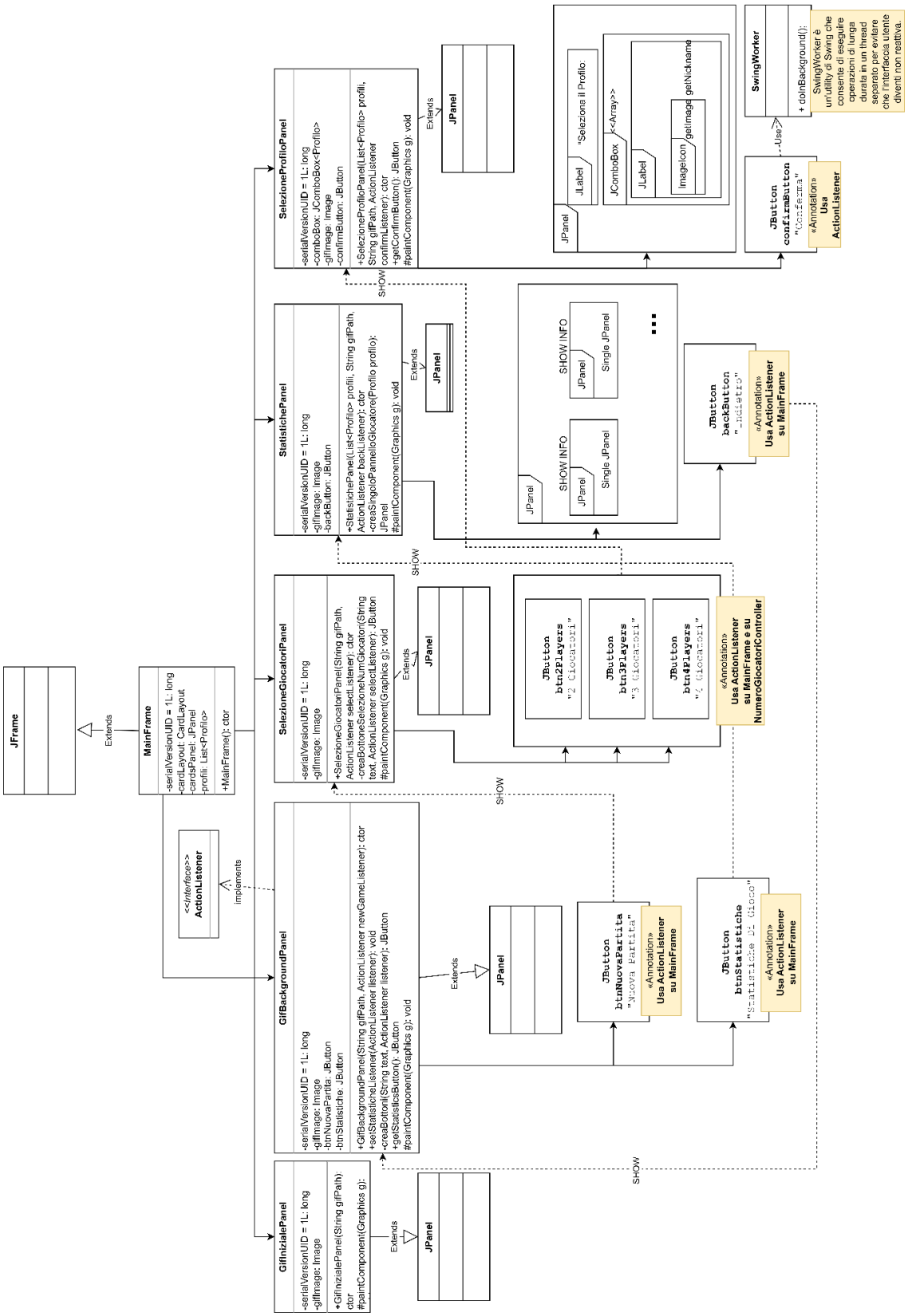

Figura 26 Diagramma UML Menu Iniziale View

Struttura dell'interfaccia grafica della schermata di gioco:

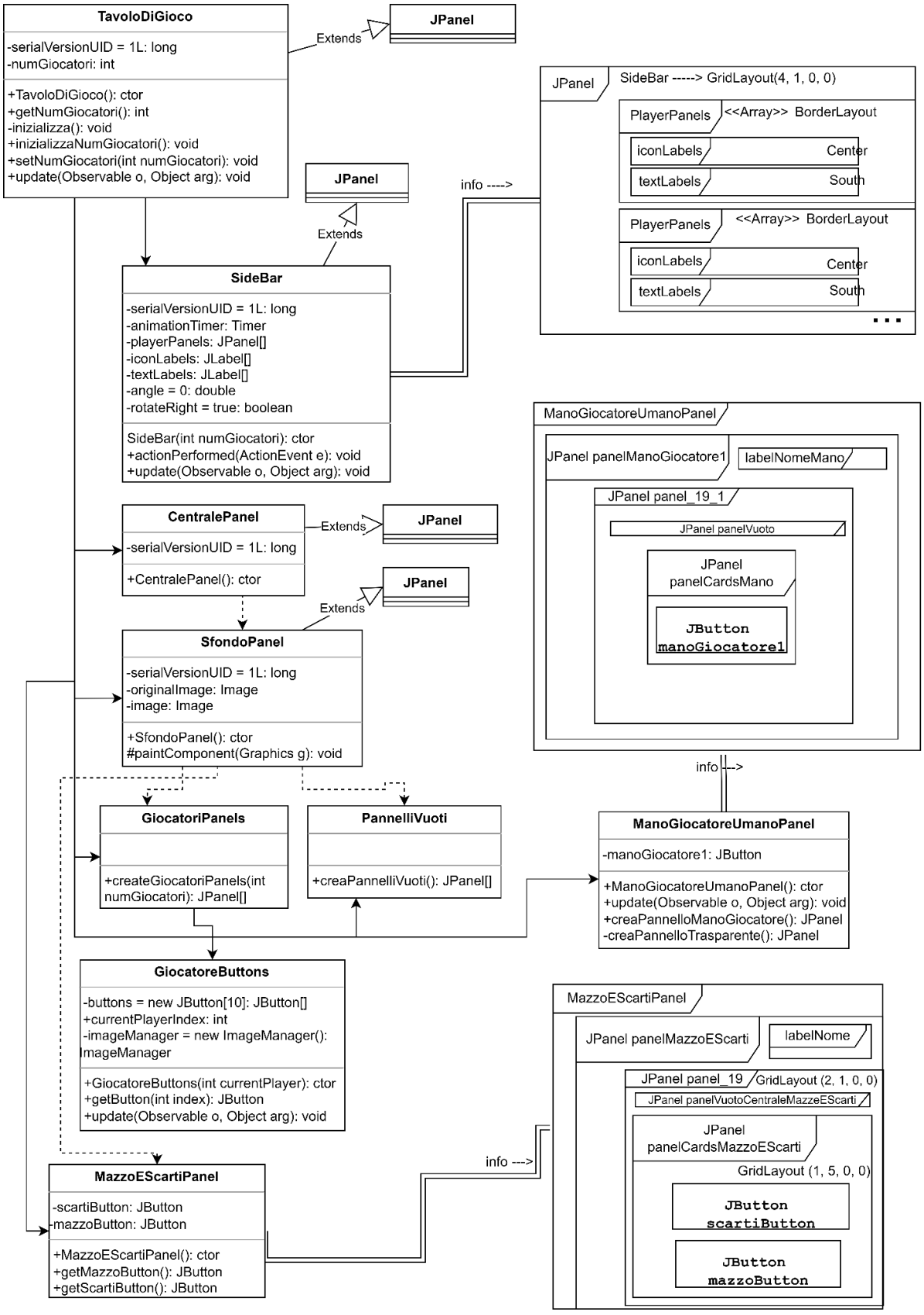

Figura 27 Diagramma UML Schermata di Gioco

Screenshot della schermata di Gioco:

Figura 28 Immagine Schermata Di Gioco

Utilizzo di *STREAM*

Le *Stream* API rappresentano una delle principali aggiunte a Java 8 e permettono una manipolazione elegante e funzionale dei dati, in particolare delle collezioni, attraverso operazioni dichiarative.

Metodo: *.inizializza()* della classe *MazzoDiCarte*

Il metodo utilizza le *Stream* per generare un mazzo di carte. L'operazione centrale qui è la creazione di uno stream di interi, che rappresenta il numero di mazzi, mappato in seguito ad uno stream di carte. Questi stream sono poi appiattiti in un singolo stream e raccolti in una lista.

Praticamente viene preso il mazzo che è stato svuotato, e si genera le carte in base al numero dei giocatori avversari.

Sfrutto il metodo *.addAll()* della classe *Collection* per aggiungere tutti gli elementi di *mazzo* alla fine di questo elenco, nell'ordine in cui sono restituiti dall'iteratore della collezione specificata. A questo punto genero uno stream; nello specifico utilizzo *IntStream* in quanto questo è specializzato per gli *int*; non posso usare *.stream()* a causa dei dati che sto

trattando in quanto sto cercando di creare uno stream che contiene un numero variabile di copie dello stesso set di dati (un mazzo di carte) infatti sfrutto il metodo *.range()* che mi permette di creare uno stream sequenziale di numeri interi primitivi in un intervallo specifico (nel mio caso l'intervallo va da 0 al numero di mazzi totale). Per ogni mazzo crea uno stream di carte utilizzando *.mapToObj().*

La firma tipica di *.mapToObj()* per *IntStream* è
<U> Stream<U> mapToObj(IntFunction>? extends U> mapper)
Dove
-'*U*' è il tipo di oggetto risultante
-*mapper* è una funzione che accetta un valore per *IntStream* e ritorna un oggetto di tipo '*U*' infatti la sua funzione è quello di trasformare un elemento in un altro, cioè di mapparlo a un nuovo valore. E' un concetto fondamentale nella programmazione funzionale e consente di manipolare i dati in modo dichiarativo.

È da notare che *.mapToObj(i -> Arrays.stream(carteValues).map(Carta::new))* per ogni numero in questa sequenza, crea un nuovo stream di carte. Senza *IntStream*, non avrei avuto un modo facile per generare quel numero variabile di mazzi.

Poi sfrutto *.flatMap().*
.flatMap(cardStream -> cardStream) prende tutti questi stream separati di carte e li "appiattisce" in un singolo stream. Questo è necessario perché senza *.flatMap()*, avrei avuto uno *Stream<Stream<Carta>>* (un stream di stream di carte) piuttosto che uno *Stream<Carta>.*

```
/**
       * Inizializza il mazzo di carte utilizzando STREAM.
       *
       * @param numeroDiGiocatoriCpu Il numero di giocatori CPU per decidere il
numero
       *                             di mazzi.
       */
      public void inizializza(int numeroDiGiocatoriCpu) {
            // Ottiene l'elenco di tutte le carte possibili
            Carte[] carteValues = Carte.values();

            // Svuota il mazzo corrente
            mazzo.clear();

            // Determina il numero di mazzi da creare in base al numero di
giocatori CPU
```

```
            int numeroDiMazzi = numeroDiGiocatoriCpu == 1 ? 1 : 2;

            // genero le carte e le aggiungo al mazzo
            mazzo.addAll(
                        // Crea uno Stream di interi da 0 a numeroDiMazzi - 1
                        IntStream.range(0, numeroDiMazzi)
                                    // Per ogni numero nel range, crea uno
Stream di carte utilizzando .map
                                    .mapToObj(i ->
Arrays.stream(carteValues).map(Carta::new))
                                    // Appiattisci gli Stream di carte in un
singolo Stream
                                    .flatMap(cardStream -> cardStream)
                                    // Raccogli tutte le carte in una lista
                                    .collect(Collectors.toList()));
      }
```

Figura 29 Metodo .inizializza() di MazzoDiCarte

Quindi, l'uso di *IntStream* e di *.mapToObj()* è un modo per generare dinamicamente un numero variabile di copie dello stesso set di dati e appiattirli in un unico stream. Se avessi cercato di fare ciò con uno Stream normale, avrei dovuto trovare un altro meccanismo per generare quelle copie multiple, il che avrebbe potuto risultare meno intuitivo e più complicato.

Metodo: *.raccogliERimescola()* della classe *MazzoDiCarte*

Qui, le *Stream* vengono utilizzate per iterare su ogni carta nel mazzo (*.forEach()*) e impostare la sua proprietà "scoperta" su false, il che indica che la carta è stata "girata" e non è visibile.

Ho fatto in modo di eseguire questa operazione in modo da assicurarmi che ogni volta che rimescolo la carta venga effettivamente coperta.

```
/**
       * Raccoglie e rimescola le carte nel mazzo usando STREAM.
       */
      public void raccogliERimescola() {
            cursore = 0;

            // Utilizza Stream API per impostare tutte le carte come non
scoperte
            mazzo.stream().forEach(carta -> carta.setScoperta(false));

            // Rimescola il mazzo
            Collections.shuffle(mazzo);
      }
```

Figura 30 Metodo .raccogliERImescola() di MazzoDiCarte

Metodo: *azionePosiziona()* della classe *AzioneJolly*

Viene utilizzata una *Stream* per filtrare le carte sul tavolo che non sono scoperte (ossia quelle coperte). Successivamente, la carta Jolly può sostituire una di queste carte coperte a seconda della logica di gioco.

Viene creato uno stream a partire dalle carte sul tavolo, poi si filtra tenendo nella lista solo le carte coperte. Nel caso in cui ci siano delle carte coperte, se ne sceglie una casualmente e si posiziona la carta in mano nella relativa posizione che rispetta i predicati.
Se non ci sono posti liberi , la carta viene semplicemente scartata.

```
/**
	 * Esegue l'azione associata al posizionamento della carta Jolly.
	 *
	 * @param cartaInMano      La carta che il giocatore ha scelto di
posizionare.
	 * @param carteSulTavolo   La lista di carte attualmente sul tavolo.
	 * @param mazzoDegliScarti Il mazzo degli scarti dove vengono messe le
carte
	 *                         scartate.
	 * @param mano             La mano attuale del giocatore.
	 * @param giocatore        Il giocatore che sta eseguendo l'azione.
	 * @return La carta che è stata sostituita dalla carta Jolly, o null se la
carta
	 *         Jolly viene scartata.
	 */
	@Override
	public Carta azionePosiziona(Carta cartaInMano, List<Carta>
carteSulTavolo, MazzoDegliScarti mazzoDegliScarti,
			List<Carta> mano, AbstractGiocatore giocatore) {
		if (giocatore instanceof GiocatoreUmano) {
			// Per GiocatoreUmano, stiamo gestendo l'azione direttamente
in
			// gestisciAzioneCarta
			return null;
		} else if (giocatore instanceof GiocatoreCpu) {
			// Utilizzo le Stream API per ottenere le carte coperte.
			List<Carta> carteCoperte =
carteSulTavolo.stream().filter(carta -> !carta.isScoperta())
					.collect(Collectors.toList());

			// Se ci sono carte coperte, scegli una casualmente
			if (!carteCoperte.isEmpty()) {
				Carta cartaSostituita = carteCoperte.get(new
Random().nextInt(carteCoperte.size()));
				int indice = carteSulTavolo.indexOf(cartaSostituita);
				cartaSostituita.scopriCarta();
				carteSulTavolo.set(indice, cartaInMano);
				cartaInMano.setScoperta(true);
				mano.add(cartaSostituita);
				mano.remove(cartaInMano);
				return cartaSostituita;
			} else {
				// Se non ci sono carte coperte, scarta la carta Jolly
				scartaCarta(cartaInMano, mazzoDegliScarti, mano,
giocatore);
				return null;
			}
		}
		return null;
	}
```

Figura 31 Metodo: azionePosiziona() della classe AzioneJolly

Metodo: *.aggiornaProfiliPostPartita()* della classe *Partita*

Dopo ogni partita, questo metodo aggiorna i profili dei giocatori. Utilizza le Stream per iterare su ogni giocatore e aggiornare le relative statistiche del profilo.

All'inizio viene creato uno stream a partire dalla lista di giocatori inclusi, per ogni giocatore (*.forEach()*) si aggiorna il numero di partite giocate. Se il giocatore è il vincitore della partita incrementa il numero di partite vinte, altrimenti incrementa il numero di partite perse. Per concludere salva il profilo aggiornato del giocatore su il file *profili.dat*

```
/**
	 * Aggiorna i profili dei giocatori al termine della partita utilizzando
uno
	 * stream.
	 *
	 * @param vincitore Il giocatore che ha vinto la partita.
	 * @throws IOException            Se si verifica un errore di I/O durante
il
	 *                                salvataggio del profilo.
	 * @throws ClassNotFoundException Se si verifica un errore di class not
found
	 *                                durante il caricamento del profilo.
	 */
	private void aggiornaProfiliPostPartita(AbstractGiocatore vincitore)
throws IOException, ClassNotFoundException {
		// Creazione di una lista contenente tutti i giocatori, inclusi il
giocatore
		// umano e i giocatori CPU
		List<AbstractGiocatore> tuttiIGiocatori = new ArrayList<>();
		tuttiIGiocatori.add(GameManager.getInstance().getGiocatoreUmano());
		tuttiIGiocatori.addAll(GameManager.getInstance().getGiocatoriCpu());

		// Creazione di uno stream a partire dalla lista di giocatori e
iterazione su
		// ciascun giocatore
		tuttiIGiocatori.stream().forEach(giocatore -> {
			// Aggiornamento del numero di partite giocate per il
giocatore

	giocatore.getProfilo().setPartiteGiocate(giocatore.getProfilo().getPartite
Giocate() + 1);

			// Verifica se il giocatore è il vincitore della partita
			if (giocatore.equals(vincitore)) {
				// Se sì, incrementa il numero di partite vinte

	giocatore.getProfilo().setPartiteVinte(giocatore.getProfilo().getPartiteVi
nte() + 1);
			} else {
				// Altrimenti, incrementa il numero di partite perse

	giocatore.getProfilo().setPartitePerse(giocatore.getProfilo().getPartitePe
rse() + 1);
			}

			// Salva il profilo aggiornato del giocatore su file
			try {
```

```
      ProfiloManager.aggiornaESalvaProfilo(giocatore.getProfilo(),
"profili.dat");
                  } catch (ClassNotFoundException | IOException e) {
                        // Gestione delle eccezioni in caso di errore durante il
salvataggio del profilo
                        e.printStackTrace();
                  }
            });
```

Figura 32 Metodo: .aggiornaProfiliPostPartita() della classe Partita

Metodo: *.setProfiloGiocatoreUmano()* della classe *ProfiloAssigner*

Dopo aver assegnato un profilo a un giocatore umano, questo metodo utilizza una Stream per rimuovere tale profilo dalla lista dei profili disponibili.

Per svolgere questa operazione, all'interno dello stream si filtra i profili diversi dal profilo assegnato e alla fine si utilizza *.collect()* per raccogliere i profili filtrati in una nuova lista.

```
/**
       * Assegna un profilo al giocatore umano e aggiorna l'elenco dei profili
       * disponibili utilizzando Stream. Questo metodo assegna un profilo al
giocatore
       * umano, aggiunge il profilo alla lista degli elenchi dei profili
disponibili e
       * rimuove il profilo assegnato dalla lista degli elenchi dei profili
       * disponibili utilizzando le stream.
       *
       * @param profilo Il profilo da assegnare al giocatore umano.
       */
      public void setProfiloGiocatoreUmano(Profilo profilo) {
            // Assegna il profilo al giocatore umano tramite GameManager e
GiocatoreUmano
            GameManager.getInstance().getGiocatoreUmano().setProfilo(profilo);
            GiocatoreUmano.getInstance().setProfilo(profilo);

            // Aggiunge il profilo alla lista degli elenchi dei profili
disponibili
            StatoGioco.getInstance().aggiungiProfiloGiocatore(profilo);

            // Utilizza uno stream per rimuovere il profilo assegnato dalla
lista degli
            // elenchi dei profili disponibili
            profili = profili.stream().filter(p -> !p.equals(profilo)) // Filtra
i profili diversi dal profilo assegnato
                        .collect(Collectors.toList()); // Raccoglie i profili
filtrati in una nuova lista
      }
```

Figura 33 Metodo: .setProfiloGiocatoreUmano() della classe ProfiloAssigner

Metodo: *.giocatoreVincitore()* della classe *SetManager*

In questo caso eseguo una semplice verifica se tutte le carte sul tavolo di un giocatore sono scoperte (utilizzando una *Stream* e *.allMatch()*).

.allMatch() è un metodo di *Stream* che accetta come argomento una interfaccia funzionale nello specifico *Predicate<T>* che è una funzione che prende un singolo input e restituisce un valore booleano, si tratta di una funzione che può essere espressa come una lamba o

un metodo di riferimento. Quindi l'operazione *.allMatch()* itera su tutti gli elementi dello stream e applica il predicato a ciascuno di essi. Se tutti gli elementi soddisfano la condizione imposta dal predicato, *.allMatch()* restituirà *true*. Se almeno un elemento non soddisfa la condizione, restituirà false.

```
/**
       * Controlla se un giocatore ha vinto.
       *
       * @param giocatore Il giocatore da verificare.
       * @return `true` se il giocatore ha vinto, altrimenti `false`.
       */
      private boolean giocatoreVincitore(AbstractGiocatore giocatore) {
            return
giocatore.getCarteSulTavolo().stream().allMatch(Carta::isScoperta);
      }
```

Figura 34 Metodo: .giocatoreVincitore() della classe SetManager

Metodo: *.determinaVincitoriSet()* della classe *SetManager*

Utilizzando le *Stream*, questo metodo filtra i giocatori controllando se tutte le loro carte sul tavolo sono scoperte. Per trovare i vincitori, si filtra la lista dei giocatori, prendendo solo i giocatori che rispettano la condizione di avere tutte le carte scoperte.

```
/**
       * Determina i vincitori del set attuale.
       *
       * @return Una lista di giocatori che hanno vinto il set.
       */
      private List<AbstractGiocatore> determinaVincitoriSet() {

            /*
             * Ottieni la lista di tutti i giocatori, inclusi il giocatore umano
e i
             * giocatori CPU.
             */
            List<AbstractGiocatore> giocatori = getGiocatori();

            /*
             * Utilizza una stream per filtrare i giocatori in base a una
condizione.
             * La condizione e' che tutte le carte sul tavolo del giocatore
siano scoperte.
             * (isScoperta() restituisce true per tutte le carte).
             */
            List<AbstractGiocatore> vincitori = giocatori.stream()
                        /*
                         * giocatori.stream() inizia una stream sulla lista
giocatori.
                         */

                        .filter(giocatore ->
giocatore.getCarteSulTavolo().stream().allMatch(Carta::isScoperta))
                        /*
                         * E' un'operazione di filtro che applica una condizione
a ciascun elemento
                         * della stream. La condizione e' definita tramite la
lambda giocatore ->
                         *
giocatore.getCarteSulTavolo().stream().allMatch(Carta::isScoperta). Questa
```

```
                        * lambda prende ogni giocatore nella stream, quindi
giocatore e' una variabile
                        * che rappresenta un elemento della lista giocatori.
                        */
                       .collect(Collectors.toList());

            /* Restituisce la lista dei vincitori del set.*/
            return vincitori;
      }
```

Figura 35 Metodo: .determinaVincitoriSet() della classe SetManager

Metodo: *.cercaCartaSulTavolo()* della classe *GiocatoreCpu*

Attraverso l'uso delle *Stream*, questo metodo verifica se una carta specifica (non scoperta) è presente sul tavolo in base a una determinata logica.

Viene creato uno stream prendendo le carte sul tavolo in input, queste vengono filtrate per prendere solo quelle che sono coperte, attraverso *.anyMatch()* verifichiamo se almeno un elemento soddisfa il predicato, altrimenti restituisce *false*; *.anyMatch()* si interrompe non appena si trova un elemento che soddisfa il predicato quindi significa che è stata trovata la carta sul tavolo. Il predicato in questo caso è che *.getValore().getNumero()* sia uguale a *getCarteSulTavolo().indexOf(cartaSulTavolo + 1) ;*

Nota Bene: sto usando == perché sto confrontando due valori primitivi di tipo *int*.

```
/**
       * Cerca la carta sul tavolo e restituisce true se trovata.
       *
       * @param cartaInCimaMazzoScarti La carta in cima al mazzo degli scarti.
       * @return True se la carta viene trovata sul tavolo, altrimenti false.
       */
      private boolean cercaCartaSulTavolo(Carta cartaInCimaMazzoScarti) {
            return getCarteSulTavolo().stream().filter(cartaSulTavolo ->
!cartaSulTavolo.isScoperta())
                        .anyMatch(cartaSulTavolo ->
cartaInCimaMazzoScarti.getValore()
                                    .getNumero() ==
getCarteSulTavolo().indexOf(cartaSulTavolo) + 1);
      }
```

Figura 36 Metodo: .cercaCartaSulTavolo() della classe GiocatoreCpu

Il codice sfrutta la potenza delle *Stream* API in Java per effettuare operazioni su collezioni in modo funzionale e dichiarativo. Questo non solo rende il codice più leggibile e manutenibile ma permette anche di esprimere complesse operazioni su dati in modo conciso.

Riproduzione di AUDIO SAMPLE

Per la gestione di audio sample durante l'esecuzione del software, ho creato la classe *SimpleAudioManager*.

Questa classe incorpora principi di progettazione di alto livello, in particolare il pattern Singleton, garantendo l'unicità dell'istanza durante tutto il ciclo di vita dell'applicazione.

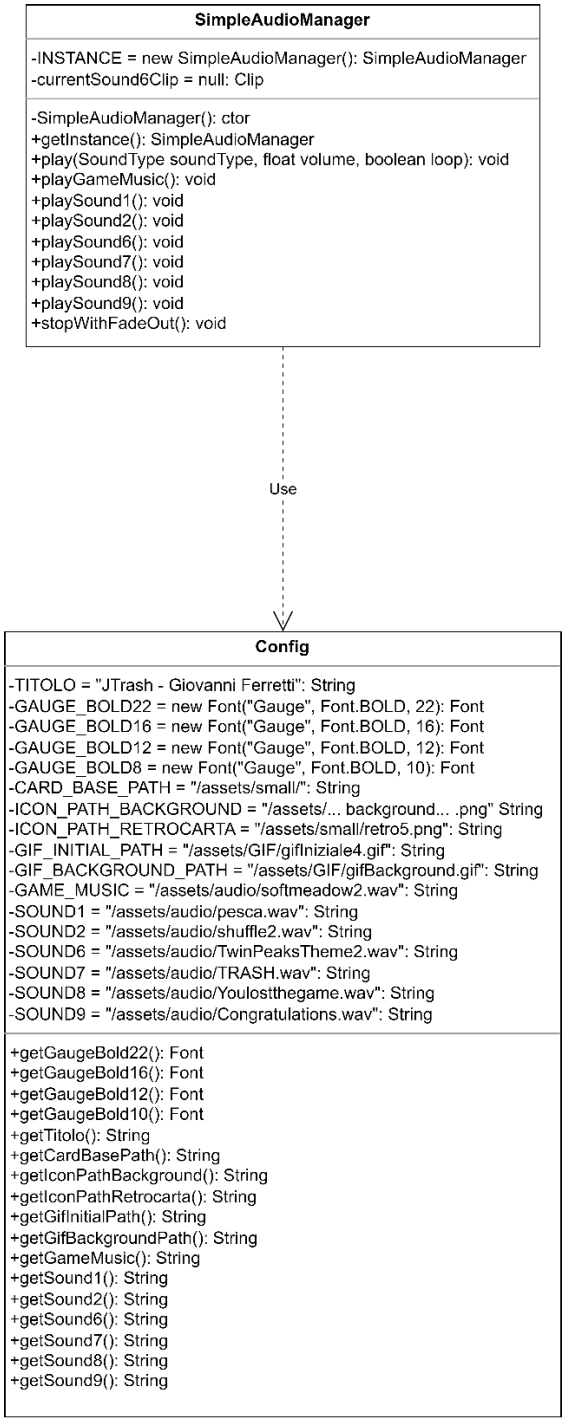

Figura 37 Diagramma UML Riproduzione Audio Sample

Enumerazione: *SoundType*

Centrale alla classe è l'enumerazione *SoundType*, che definisce diversi tipi di suoni disponibili all'interno del gioco. Questa enumerazione serve come riferimento chiaro e semantico alle risorse audio, associando ogni suono a un percorso di file specifico, e facilitando l'estensibilità, poiché l'aggiunta di nuovi suoni richiede solamente l'introduzione di nuovi elementi all'interno dell'enumerazione. Ogni suono viene preso dal file *Config.java* che si trova all'interno della View.

Funzionalità di Riproduzione

La classe presenta una serie di metodi per la riproduzione dei suoni, tra cui il metodo *.play()* , che rappresenta il cuore del sistema audio. Questo metodo si occupa di caricare e riprodurre una specifica risorsa audio, regolando il volume e decidendo se il suono debba essere riprodotto in loop o no. Questo tipo di modularità e riutilizzo del codice promuove una struttura pulita e manutenibile.
Un particolare degno di nota è l'implementazione della funzione *.stopWithFadeOut()*. Questa funzione offre un'esperienza audio più piacevole all'utente, riducendo gradualmente il volume di un suono fino alla sua interruzione, evitando interruzioni brusche che potrebbero disturbare l'esperienza dell'utente.

Da notare che nel software ogni volta che voglio eseguire un file audio posso chiamare l'istanza della classe con *SimpleAudioManager.getInstance()* e a seguire il metodo che voglio utilizzare per l'esecuzione del file audio.

Riproduzione di ANIMAZIONI

Le animazioni che ho deciso di includere nel software si trovano in 2 punti del software: all'inizio nella schermata iniziale, e durante l'esecuzione della partita nella barra laterale della schermata principale.

Classe: *GifInizialePanel*

La classe *GifInizialePanel* estende *JPanel*, che è un componente fondamentale del package *Swing* in Java. Questo consente al pannello di ereditare tutte le funzionalità basilari di un *JPanel*, ma con la capacità aggiuntiva di visualizzare un'immagine GIF.

L'elemento centrale della classe è l'attributo *gifImage*, che conserva l'immagine GIF che sarà mostrata all'interno del pannello. È essenziale notare che l'immagine viene caricata una sola volta attraverso il costruttore della classe, utilizzando *new ImageIcon(gifPath).getImage()*. Questo assicura un efficiente utilizzo delle risorse, evitando di ricaricare l'immagine ogni volta che viene visualizzata.

La personalizzazione principale avviene nel metodo *paintComponent*, che sovrascrive il comportamento standard di *JPanel*. Questo metodo viene invocato automaticamente ogni volta che il pannello ha bisogno di essere ridisegnato, come durante le fasi di ridimensionamento della finestra. Utilizzando il contesto grafico g, l'immagine GIF viene ridimensionata alle dimensioni correnti del pannello, garantendo che l'animazione appaia fluida e si adatti dinamicamente alle dimensioni del pannello.

```
// Disegna l'immagine ridimensionata
            g.drawImage(gifImage, 0, 0, width, height, this);
```

Figura 38 Disegna l'immagine ridimensionata

Durata dell'Animazione: la classe fornisce un metodo *.getDurataGif()* che restituisce un valore fisso di 7650 millisecondi.

```
public int getDurataGif() {
            return 7650;
      }
```

Figura 39 Restituisce la durata della GIF

Classe: *SideBar*

L'animazione di rotazione viene gestita attraverso un Timer (*animationTimer*) che invoca il metodo *.actionPerformed()* ad intervalli regolari. All'interno di questo metodo, l'angolo di rotazione (*angle*) viene incrementato o decrementato a seconda della direzione corrente della rotazione, determinata dalla variabile booleana *rotateRight*.

```
@Override
      public void actionPerformed(ActionEvent e) {
            if (rotateRight) {
                  angle += 0.03;
                  if (angle >= 0.2) {
                        rotateRight = false;
                  }
            } else {
                  angle -= 0.03;
                  if (angle <= -0.2) {
                        rotateRight = true;
                  }
            }
            repaint();
      }
```

Figura 40 Metodo .actionPerformed() di SideBar

La rotazione effettiva dell'avatar viene gestita nel metodo *.paintComponent()* della classe interna anonima di *JLabel*. Utilizzando l'oggetto *Graphics2D*, l'immagine dell'avatar viene ruotata attorno al suo centro. Questa rotazione dinamica offre un effetto visivo gradevole, dando vita all'interfaccia utente.

```
/**Sovrascrive il metodo paintComponent per permettere la rotazione dell'icona.
 * Se l'icona è presente, verrà ruotata in base all'angolo specificato;
* altrimenti verra' chiamato il metodo paintComponent della superclasse.
*
* @param g L'oggetto Graphics usato per disegnare componenti.
*/
      @Override
      protected void paintComponent(Graphics g) {
      Graphics2D g2d = (Graphics2D) g.create();

      (getIcon() != null) {
      int iconWidth = getIcon().getIconWidth();
      int iconHeight = getIcon().getIconHeight();
      AffineTransform original = g2d.getTransform();

      g2d.rotate(angle, getWidth() / 2, getHeight() / 2);
      g2d.drawImage(((ImageIcon) getIcon()).getImage(), (getWidth() - iconWidth)
/ 2,
       (getHeight() - iconHeight) / 2, this);
      g2d.setTransform(original);
      } else {
      super.paintComponent(g);
      }

      g2d.dispose();
      }
```