Relazione JBlackJack

Federico Raponi 1963339

Canale presenza MZ

Contents

1 Introduzione

Per la realizzazione del progetto JBlackJack ho scelto di utilizzare la libreria grafica JavaFx.

> JavaFX is an open source, next generation client application platform for desktop, mobile and embedded systems built on Java. It is a collaborative effort by many individuals and companies with the goal of producing a modern, efficient, and fully featured toolkit for developing rich client applications.

Grazie a questa libreria è stato possibile realizzare il progetto seguendo l'MVC pattern. I principali pattern utilizzati per questo progetto sono:

- **MVC Pattern** per la gestione della GUI
- **Observer Pattern** per la gestione del ritiro delle puntate da parte del Dealer
- **Singleton Pattern** per creare delle singole istanze di una determinata classe

1.1 Model View Controller Pattern

Il pattern MVC (Model View Controller) prevede la suddivisione di un'interfaccia grafica in tre parti:

- **Model**: che contiene i dati e la logica dell'applicazione, che nel mio caso sono le varie classi, interfaccie ed enumerazioni che sono alla base del progetto.
- **View**: sono i file dedicati alla rappresentazione grafica dei dati che vengono elaborati durante l'esecuzione del software.
- **Controller**: sono dei file dedicati alla comunicazione tra *View* e *Model*.

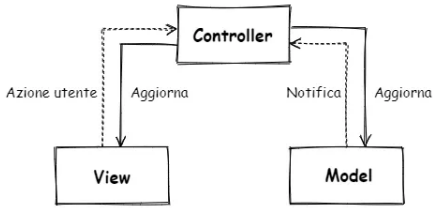

Figure 1: MVC

1.2 Observer Pattern

Questo pattern viene utilizzato per creare una relazione di tra vari oggetti, basandosi sul concetto di *Observer* e *Subject*. L'*Observer* è l'oggetto che osserva il *Subject*. Un *Subject* può essere osservato da più observer contemporaneamente, e il suo compito sarà quello di notificare a questi ultimi un determinato cambiamento di stato.

1.2.1 Observer

```
1 public interface Observer {
2     void update();
3 }
```

questa interfaccia impone a chiunque la estenda di implementare il seguente metodo di ***update*** che esegue del codice in risposta ad un cambiamento di stato notificato dal subject.

1.2.2 Subject

```
public interface Subject {
    void attach(Observer observer);

    void detach(Observer observer);

    void notifyObservers();
}
```

questa interfaccia impone a chiunque la estenda di implementare i seguenti metodi:

- ***attach*** che serve per aggiungere un observer alla lista del subject.
- ***detach*** che serve per rimuovere un observer dalla lista del subject.
- ***notifyObservers*** che serve per notificare tutti gli observer del cambio di stato avvenuto nel subject.

1.3 Singleton Pattern

Il pattern Singleton garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. Questo è particolarmente utile quando si desidera controllare l'accesso a una risorsa condivisa. Per implementare un Singleton, si segue solitamente questo schema:

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // creazione dell'istanza
    }

    public static Singleton getInstance() {
        if (instance == null) {
            // Crea l'istanza se non esiste già
            instance = new Singleton();
        }
        return instance;
    }
}
```

Questa struttura comprende:

- **Costruttore privato:** Il costruttore della classe viene dichiarato come privato, impedendo così la creazione diretta di nuove istanze da parte del codice esterno.

- **Istanza statica:** Viene definita una variabile statica all'interno della classe per memorizzare l'unica istanza che verrà creata. Questa variabile è tipicamente privata e accessibile solo attraverso un metodo pubblico.
- **Metodo di accesso pubblico:** Si implementa un metodo pubblico e statico che controlla se l'istanza è già stata creata. Se non lo è, crea l'istanza e la restituisce; altrimenti, restituisce semplicemente l'istanza esistente. Questo metodo è il punto di accesso globale all'istanza del Singleton.

2 Struttura del progetto

Ho deciso di suddividere questo progetto in due package principali:

- **java package**: nella quale risiedono tutti i file java, a loro volta suddivisi in:
  - ***classes:*** al suo interno ci sono tutte le classi create per il funzionamento del gioco.
  - ***enums***: al suo interno ci sono le enumerazioni utilizzate.
  - ***interfaces***: al suo interno ci sono le interfacce utilizzate.
  - ***exceptions***: al suo interno ci sono le eccezioni personalizzate.
  - Il resto dei file presenti sono i vari Controller delle pagine, tra cui il file JBlackJack in cui risiede il *main* del Software.
- ***jesources package***: nella quale risiedono tutti i file dedicati all'intefaccia grafica, come i file.fxml e le immagini utilizzate.

2.1 Classes

Analizzando nello specifico le classi realizzate:

2.1.1 AbstractGiocatore

Questa classe astratta rappresenta un giocatore di BlackJack. Questa classe astratta, ovviamente, non può essere istanziata ma, viene utilizzata come struttura base da estendere con le altre classi concrete. All'interno di questa classe troviamo i seguenti attributi:

- nickname
- avatar
- partiteVinte
- partitePareggiate
- partitePerse

- livello
- mano
  - Questo attributo è l'istanza di un'altra classe realizzata per il gioco che rappresenta la mano, intesa come lista di carte in possesso dal giocatore.

Questa classe espone i vari metodi *setter e getter* per permettere l'accesso ai vari attributi privati. Sono presenti inoltre dei metodi per l'aggiornamento del livello del giocatore tramite i punti esperienza guadagnati da i vari esiti delle partite giocate.

Per aumentare di livello un giocatore dovrà ottenere 1000 punti, che verranno assegnati al termine della vittoria sulla base del risultato nel seguente modo:

- Vittoria: 300 punti
- Pareggio: 150 punti
- Sconfitta: 100 punti

Solitamente il pareggio in questo genere di giochi non è contemplato, ma leggendo il regolamento esiste questo caso.

> *Il giocatore che fa 21 con le prime carte assegnategli dal mazziere (cioè riceve un asso e una figura) forma il cosiddetto "black jack" e ha diritto al pagamento di 3 a 2 (una volta e mezzo la posta) o in alcuni casi di 6 a 5 ; se anche il mazziere realizza il black jack, la mano è considerata alla pari.*

2.1.2 GiocatoreNonDealer

Questa classe astratta è stata creata per fornire attributi aggiuntivi per le istanze del Bot e del Giocatore che non devono essere presenti all'interno dell'istanza del Dealer. Questi attributi sono:

- secondaMano
  - utilizzata nel caso un cui il giocatore decida di dividere la prima mano, possibile solo se il dealer ha distribuito all'inizio della partita due carte dello stesso valore, ad esempio due 7.
- puntata
- puntataSecondaMano
- saldo
  - quantità totale di *"coins"* disponibili per il giocatore

Oltre a questi attributi è presente la lista degli *Observer* in quanto questa classe astratta implementa l'interfaccia *Subject*.

In questa classe sono quindi presenti tutti i metodi relativi alla gestione delle puntate e della seconda mano, oltre a quelli necessari all'Observer pattern.

2.1.3 Bot

La classe concreta Bot estende la classe astratta GiocatoreNonDealer e permette di istanziare un Bot con un nickname e un avatar scelti casualmente tra quelli disponibili.

2.1.4 Giocatore

La classe concreta Giocatore estende la classe astratta GiocatoreNonDealer. Questa classe sfrutta il Singleton Pattern, permettendo di istanziare il giocatore una sola volta, in quanto il giocatore sarà l'utente che utilizzerà il software. Questa classe espone quindi un metodo di inizializzazione del giocatore che chiama il costruttore privato e il metodo pubblico per ottenere l'istanza di questo giocatore.

```
1  public class Giocatore extends GiocatoreNonDealer {
2      private static Giocatore instance;
3
4      private Giocatore(String nickname, String avatar) {
5          super(nickname, avatar);
6      }
7
8      public static Giocatore getInstance() throws
       ↪  GiocatoreException {
9          if (instance == null) {
10             throw new GiocatoreException("Giocatore fisico non
               ↪  inizializzato");
11         }
12         return instance;
13     }
14
15     public static void inizializzaGiocatore(String nickname,
       ↪  String avatar) throws GiocatoreException {
16         if (instance != null) {
17             throw new GiocatoreException("Giocatore fisico già
               ↪  inizializzato");
18         }
19         instance = new Giocatore(nickname, avatar);
20     }
21
22     public static void logout() {
23         instance = null;
24     }
25 }
```

2.1.5 Dealer

La classe Dealer estende AbstractGiocatore ereditando tutte le funzioni base di un giocatore, ma a differenza del Giocatore e del Bot, possiede questi attributi:

- mazzo
- cartaCoperta

Il Dealer in quanto gestore della partita, necessita di avere il mazzo da gioco, su cui vengono effettuate varie operazioni, come mescolare o distribuire le carte agli altri giocatori. Da regolamento il dealer possiede una carta coperta e una scoperta fino all'inizio del suo turno, quindi una volta iniziato quest'ultimo la sua carta coperta verrà messa nella mano effettiva e concorrerà al punteggio totale. Anche il dealer sfrutta il Singleton pattern, ma a differenza delle altre classi dei giocatori il Dealer è un Observer.

Ho strutturato in questo modo l'observer pattern per fare in modo che il il giocatore possa notificare a quest'ultimo quando il suo punteggio è superiore a 21 *(sballato)*, in modo tale che il dealer possa ritirare la sua puntata immediatamente, senza dover aspettare la fine dalla partita. Questo avviene perché secondo il regolamento se un giocatore dovesse *sballare* durante il suo turno perderebbe immediatamente la partita, anche se il dealer dovesse sballare nel suo turno.

2.1.6 Mano

La classe della Mano, rappresenta la lista delle carte in possesso da giocatore, e contiene tutti i metodi per la gestione di quest'ultime, come:

- aggiungiCarta
- rimuoviCarta
- calcolaPunteggio
- getCarte
- resetMano
- isBlackJack

2.1.7 Mazzo

La classe Mazzo rappresenta il mazzo di gioco, che come da regolamento può essere composto da 2 a 6 mazzi di carte francesi. Questa classe quindi ha un costruttore pubblico che richiede come parametro il numero di mazzi di carte francesi con la quale si vuole comporre il mazzo di gioco. Se non vine passato questo parametro tramite l'*overload* del metodo viene generato un mazzo da gioco composto da 2 mazzi di carte francesi.

2.1.8 Carta

La classe Carta rappresenta una singola carta da gioco che ha come attributi:

- valore
- seme
- path
  - Il path è il percorso all'immagine relativa alla carta che serve poi per mostrare la carta in gioco.

2.1.9 Tavolo

La classe Tavolo rappresenta l'effettivo tavolo di gioco, quindi possiede la lista di giocatori. Dato che il anche questa classe sfrutta il Singleton Pattern il costruttore è privato, delegando la creazione dell'istanza quando viene chiamato il metodo pubblico ***inizializzaTavolo***. Questo metodo riceve come parametri, il Giocatore effettivo, il numero di mazzi francesi che comporranno il mazzo di gioco, e il numero di bot contro la quale si vuole giocare.

Dato che il giocatore non può giocare da solo, il numero minimo di bot contro il quale potrà giocare è 1, che sarà proprio il Dealer. Si può arrivare fino ad un massimo di 3 bot in una partita, quindi un tavolo potrà essere al minimo da:

- Giocatore
- Dealer

Al massimo da:

- Giocatore
- Dealer
- Bot1
- Bot2

2.2 Enums

Le enumerazioni utilizzate in questo progetto sono.

2.2.1 Seme

Enumerazione che rappresenta i semi delle carte da gioco francesi.

2.2.2 Valore

Enumerazione che rappresenta i valori delle carte da gioco francesi.

2.3 Exceptions

Ho realizzato delle eccezioni personalizzate da lanciare in specifiche situazioni. Tutte le successive eccezioni estendono la classe Exception che a sua volta Throwable rendendole quindi lanciabili tramite la parola chiave ***throw***.

2.3.1 GiocatoreException

Questa eccezione accetta un messaggio di errore e serve quando si verifica un errore relativo al giocatore. Ad esempio:

```
public static Giocatore getInstance() throws GiocatoreException {
    if (instance == null) {
        throw new GiocatoreException("Giocatore fisico non
        ↪  inizializzato");
    }
    return instance;
}
```

2.3.2 MazzoException

Questa eccezione accetta un messaggio di errore e serve quando si verifica un errore relativo al mazzo. Ad esempio:

```
public Mazzo(int numMazzi) throws MazzoException {
    if (numMazzi < 2 || numMazzi > 6) {
        throw new MazzoException("Il numero di mazzi deve essere
        ↪  compreso tra 2 e 6");
    }
    //resto del codice
}
```

2.3.3 TavoloExceptions

Questa eccezione accetta un messaggio di errore e serve quando si verifica un errore relativo al tavolo Ad esempio:

```
public static Tavolo getInstance() throws TavoloExceptions {
    if (instance == null) {
        throw new TavoloExceptions("Tavolo non inizializzato");
    }
    return instance;
}
```

2.4 Interfaces

Le interfacce che sono state sfruttate sono quelle necessarie, come detto in precedenza, per l'implementazione dell'Observer Pattern. Sono state quindi implementate:

- Observer
- Subjet

2.5 Controllers

Una parte fondamentale della struttura del progetto sono i Controllers, classi dedicate alla comunicazione tra Model e View.

2.5.1 BaseController

Questa classe astratta serve come base per tutti gli altri controller, implementando il metodo ***switchScene*** che riceve come parametro il model del file .fxml da caricare per il cambio scena e l'evento che innesca il cambiamento.

```
1 void switchScene(String fxml, ActionEvent event) throws
  ↪ IOException {
2     Parent root =
      ↪ FXMLLoader.load(Objects.requireNonNull(getClass().getResource(fxml)));
3     Stage stage = (Stage) ((Node)
      ↪ event.getSource()).getScene().getWindow();
4     Scene scene = new Scene(root);
5     stage.setScene(scene);
6     stage.show();
7 }
```

2.5.2 LoginController

Questo controller è adibito all'inizializzazione del Giocatore. Gestisce quindi il campo di testo *(TextField)* nella quale l'utente inserirà in nickname, effettua anche un controllo sulla validità di quest'ultimo, infatti non potrà essere selezionato un nickname vuoto. In quel caso comparirà un *Alert* di errore.

In caso di corretto inserimento del nickname, verrà inizializzato il giocatore e ci sarà un cambio di scena che porterà alla *Home*.

2.5.3 HomeController

Questo controller oltre ad estendere BaseController, implementa l'interfaccia *Initializable*, che impone alla classe di effettuare l'*Override* del metodo *initialize*.

Il metodo initialize Questo metodo è utilizzato per i seguenti scopi:

1. Configurazione iniziale dei componenti: initialize viene utilizzato per impostare lo stato iniziale dei componenti dell'interfaccia utente (come bottoni, etichette, tabelle, ecc.) prima che l'utente interagisca con l'applicazione.

2. Popolamento di dati iniziali: Ad esempio, inserimento di testo all'interno di un *Label*.

3. Connessione tra FXML e controller: Quando si utilizza il markup FXML per progettare l'interfaccia utente, initialize è il punto in cui puoi fare riferimento ai componenti definiti nel file FXML e collegarli alla logica del controller.

In questo controller il metodo *initialize* viene utilizzato per impostare delle Label nelle quali comparirà il nickname del giocatore e il suo saldo attuale.

Ci sono poi tre bottoni che permettono di navigare dalla Home verso le altre scene.

Intuitivamente il bottone *Esci* permette di tornare alla schermata di *Login* andando ad eliminare l'istanza del Giocatore.

Il bottone *Profilo* condurrà alla scena dove, il giocatore potrà personalizzare il suo avatar e il suo nickname. Potrà inoltre effettuare una ricarica di *coins* nel caso la sfortuna avesse preso il sopravvento durante le fasi di gioco.

Il bottone *Gioca* condurrà alla pagina di inizializzazione del gioco.

2.5.4 ProfiloController

Questo controller come il precedente implementa l'interfaccia Initializable permettendo di inserire all'interno del *TextField* adibito alla modifica del nickname il nickname corrente del giocatore. Serve inoltre a popolare il Label con la quantità di *coins* e l'avatar scelto dal giocatore.

Una volta inizializzata la scena l'utente potrà appunto, personalizzare il proprio avatar scegliendone uno tra quelli disponibili, personalizzare il proprio nickname o ricaricare il proprio saldo.

Cliccando sul bottone *Indietro* potrà tornare alla scena *Home* pronto per giocare.

2.5.5 PreGiocoController

Questo controller serve al giocatore per decidere come affrontare la sua partita, dando la possibilità di decidere di quanti mazzi di carte francesi si comporrà il mazzo di gioco e contro quanti bot verrà giocata la partita.

Come detto in precedenza il numero minimo di mazzi è 2 e il massimo è 6, mentre il numero minimo di Bot è 1 e il massimo è 3.

Tramite la pressione dei relativi bottoni potrà selezionare le impostazioni desiderate, cliccando quindi su *Avanti* si entrerà nella vera e propria schermata di gioco, mentre cliccando *Indietro* si tornerà alla scena *Home*

2.5.6 GiocoController

Questo controller è sicuramente il più complesso tra tutti, in quanto deve gestire molti eventi e simulare una partita di BlackJack.

Partendo con in metodo *initialize* abbiamo la configurazione base della schermata, mostrando sulla scena i dati relativi al giocatore, e i dati relativi ai bot, se presenti. In questo momento la scena permette di effettuare una sola operazione, ovvero effettuare la puntata di *coins* per poter entrare in gioco.

Una volta effettuata correttamente la puntata, il dealer distribuirà due carte ciascuno. In questo momento vengono calcolati i punteggi delle varie mani dei giocatori e mostrati in gioco, nel caso del Dealer il punteggio calcolato è parziale in quanto non è dato sapere il valore della carta coperta a questo punto della partita.

Una volta effettuata la distribuzione iniziale delle carte, il giocatore ha davanti a se quattro possibili mosse selezionabili con i relativi bottoni:

- Passa: permette di non chiedere nessuna carta e andare al prossimo giocatore
- Chiedi carta: permette al giocatore di ricevere una carta dal mazzo che si andrà a sommare al punteggio totale della sua mano. Può chiedere carte fin tanto che il suo punteggio sia ¡= 21.
- Raddoppia: permette di raddoppiare la puntata, ricevendo obbligatoriamente una sola carta per poi passare subito dopo. Questa operazione può essere eseguita solo se la non sono state eseguite altre operazioni in precedenza.
- Dividi: permette di dividere la mano corrente se non sono state eseguite altre operazioni e le due carte presenti nella mano del giocatore sono dello stesso valore.
  - Se il giocatore decide di dividere può in questo modo giocare due mani separate potendo scegliere su quale delle due chiedere carta.

Al termine del turno del giocatore verranno eseguirti i turni dei bot non Dealer, se presenti in partita. Quest'ultimi effettueranno delle scelte causali per progredire nella partita.

Quando sarà il turno del dealer, in primis viene scoperta la sua carta coperta, e successivamente pescherà carte dal mazzo fintanto che il suo punteggio sia minore di 17.

Al termine di tutti i turni dei giocatori verranno calcolati i punteggi e pagate le rispettive puntate in caso di vittoria.

Al termine di ciò il giocatore potrà effettuare un altra partita premendo su *Rigioca*.