Relazione Progetto JBlackJack

Francesco Zompanti
Matricola: 2012601
Corso: Presenza M-Z

1 Descrizione del Progetto

Il progetto JBlackJack è stato sviluppato per ricreare il gioco di carte BlackJack in Java.

2 Gestione Main

Il main, in questo caso denominato *JBlackJack*, gestisce l'inizializzazione della finestra per la creazione del profilo e procede al menu principale una volta creato un profilo utente.

3 Scelta Framework per l'Interfaccia Grafica

Per poter modellare l'interfaccia ho utilizzato il framework Java Swing.

4 Gestione Profilo Utente

Per la gestione del profilo utente ho modellato le seguenti classi:

4.1 ProfileCreationView (View)

Questa classe si occupa di creare la View per la creazione del profilo utente. Permette al giocatore di scegliere un nickname e un avatar tra i quattro disponibili, ognuno di essi ispirato ad un pilota di Formula 1.

Pattern utilizzati:

- **MVC**: serve per gestire l'interfaccia grafica di creazione del profilo utente;
- **Composite**: serve per organizzare i componenti dell'interfaccia.

Figura 1: Esempio di interfaccia per la creazione del profilo.

4.2 StatisticsView (View)

Questa classe permette la visualizzazione delle statistiche del giocatore:

- Nickname;
- Livello;
- Esperienza;
- Partite giocate, vinte e perse.

Pattern utilizzati:

- **MVC**: serve per gestire l'interfaccia grafica delle statistiche;
- **Composite**: serve per organizzare i componenti dell'interfaccia.

Figura 2: Esempio di interfaccia per la visualizzazione delle statistiche.

4.3 UserProfile (Model)

La classe *UserProfile* gestisce le informazioni e le statistiche di gioco del giocatore.
Pattern utilizzati:

- **Domain Model**: serve per rappresentare l'utente e gestisce la logica relativa a statistiche e progressi;
- **Serialization**: serve per salvare lo stato del profilo su un file txt;
- **Deserialization**: serve per caricare lo stato del profilo da un file txt.

Ogni giocatore ha un file *txt* con il nome del proprio profilo, che viene salvato nella directory *profiles* con il nome *NomeUtente_profilo.txt*. Il profilo parte dal livello zero ed ogni giocatore accumula 100 punti esperienza per ogni vittoria, 0 per ogni sconfitta. Per avanzare al livello successivo, è necessario moltiplicare il livello attuale per 1000 (ad es. se si vuole passare dal livello 1 al livello 2, sono necessari 1000 punti esperienza; dal livello 2 al livello 3, sono necessari 2000 punti esperienza; dal livello 3 al livello 4, sono necessari 3000 punti esperienza, e così via).

5 Gestione della Partita

Per la gestione di una partita completa, ho modellato le seguenti classi:

5.1 BlackjackController (Controller)

La classe *BlackjackController* gestisce l'interazione tra il *Model*, la *View* e l'audio di gioco.
Pattern utilizzati:

- **MVC**: serve per gestire la logica di controllo del gioco;
- **Singleton**: serve per garantire un'unica istanza del controller;
- **Observer**: serve per far interagire il *Model* e la *View*.

Il controller inizializza il *Model* e la *View*, implementa gli *ActionListener* per i tasti "Pesca Carta" e "Stai" per poter ricevere in input azioni dal giocatore, e gestisce l'audio di gioco tramite la classe *AudioManager*. Appena avviata la partita, viene inizializzata una musica di sottofondo e per ogni azione vengono riprodotti dei suoni:

- Per l'azione **Pesca Carta** viene riprodotto il suono di pesca di una carta;
- Per l'azione **Stai** viene riprodotto il suono di lancio di una chip in modo da far capire che si è terminato il proprio turno.

5.2 BlackjackModel (Model)

La classe *BlackjackModel* implementa l'inizializzazione dei giocatori, mescola il mazzo e distribuisce le carte. Inoltre, gestisce i turni dei giocatori e determina il vincitore a fine partita.

Nello specifico, sono inizializzati:

- Un giocatore umano;
- Due bot;
- Il banco.

Pattern utilizzati:

- **MVC**: serve per gestire la logica di gioco;

Utilizzo degli Stream:

- **Distribuzione delle carte**: ho implementato uno Stream per distribuire le due carte iniziali per ogni giocatore e per il banco;
- **Determinare il vincitore della partita**: ho implementato uno Stream per filtrare solo i giocatori che non hanno un valore superiore della mano pari a 21. Essi verranno confrontati col banco e poi verrà stabilito il vincitore con conseguente messaggio a schermo come negli esempi riportati qui sotto;

Figura 3: Messaggio mostrato quando il banco vince.

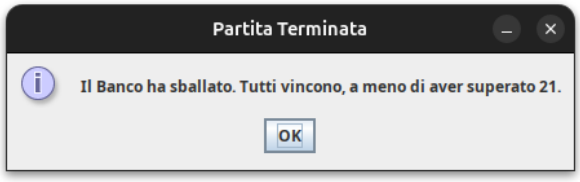

Figura 4: Messaggio mostrato quando il banco sballa.

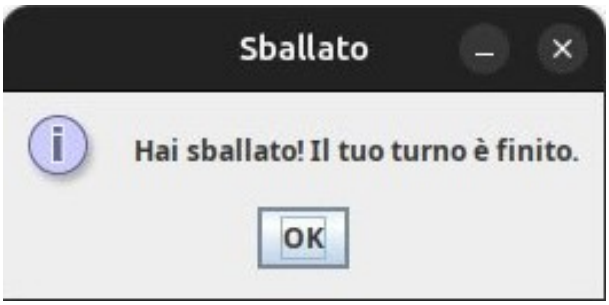

Figura 5: Messaggio mostrato quando il giocatore sballa.

Figura 6: Messaggio mostrato quando il bot vince.

5.3 MainMenuView (View)

La classe **MainMenuView** gestisce il menu principale del gioco.
Pattern utilizzato:

- **MVC**: serve per visualizzare il menu principale.

Inizializza l'interfaccia del menu, carica il profilo utente nella sezione *Statistiche*, tramite il pulsante *Gioca* permette di avviare una nuova partita e tramite il pulsante *Crea Profilo* permette di tornare alla schermata di creazione profilo per poter cambiare il nostro nickname o avatar. Per poter gestire questi pulsanti sono stati implementati degli **Action Listener** in modo da rispondere all'input del giocatore.

Figura 7: Menu Principale.

5.4 BlackjackView (View)

La classe *BlackjackView* gestisce l'interfaccia grafica del gioco ed anche l'aggiornamento di quest'ultima in base ai cambiamenti di stato nel **BlackjackModel**.

Pattern utilizzati:

- **MVC**: serve per gestire l'interfaccia grafica della partita;
- **Observer**: serve per ricevere aggiornamenti dal **BlackjackModel**.

In particolare viene inizializzata l'interfaccia grafica con i giocatori e le loro rispettive carte già distribuite ed i pulsanti per interagire con il gioco come: **Pesca Carta**, **Stai** e **Ricomincia Partita**.

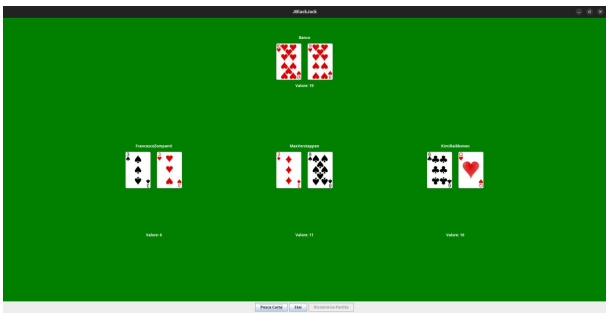

Figura 8: BlackjackView.

6 Modellazione delle Carte

Le carte sono modellate attraverso le seguenti classi ed interfacce:

6.1 Interfaccia Card (Model)

L'interfaccia *Card* definisce una carta.

Pattern utilizzato:

- **Template Method**: serve per definire i metodi base che le classi concrete devono implementare.

Metodi implementati:

- **getValue()**: restituisce il valore della carta (es. valore tra 2 e 10);
- **getSuit()**: restituisce il seme della carta (es. "hearts", "clubs", "spades", "diamonds");
- **getRank()**: restituisce il rango della carta (es. "ace", "king", "queen", ecc.);
- **getImageFileName()**: metodo di default che restituisce il nome del file immagine associato alla carta. Costruisce il nome del file immagine utilizzando rango e seme della carta (es. per un Asso di Cuori, il nome del file sarà "ace_of_hearts.png").

6.2 Classe CardFactory (Model)

La classe *CardFactory* si occupa di creare oggetti di tipo *Card*.

Pattern utilizzato:

- **Factory Method**: serve per determinare il tipo di carta da creare (Ace, Face o Numeric) basandosi sul rango fornito.

Metodo implementato:

- **createCard(String suit, String rank)**: serve per creare e restituire un oggetto di tipo Card basato sul seme e rango forniti.

6.3 Classi AceCard, FaceCard, NumericCard (Model)

Queste classi rappresentano rispettivamente le carte Ace, Figure e Numeric, implementando l'interfaccia *Card*.

Pattern utilizzato:

- **Composite**: serve per modellare le singole carte, ossia di tipo Ace, Face e Numeric, implementando l'interfaccia *Card*;

Nello specifico:

- **AceCard**: rappresenta l'Asso, con un valore fisso di 11 in questa implementazione;
- **FaceCard**: rappresenta le carte figura (Jack, Queen, King), con un valore fisso di 10;
- **NumericCard**: rappresenta le carte numeriche, con valori compresi tra 2 e 10.

6.4 Classe Deck (Model)

La classe *Deck* rappresenta il mazzo di carte.

Pattern utilizzato:

- **Singleton**: serve per assicurarci che esista una sola istanza del mazzo durante l'esecuzione del gioco.

Metodi implementati:

- **initializeDeck()**: inizializza il mazzo di carte creando tutte le combinazioni possibili di semi e ranghi tramite la classe *CardFactory*;
- **shuffle()**: mescola le carte del mazzo;
- **drawCard()**: estrae una carta dal mazzo. Se il mazzo è vuoto, viene reinizializzato e mescolato prima di estrarre una carta.

Utilizzo degli Stream:

- **Creare le combinazioni**: ho implementato due Stream per creare tutte le combinazioni possibili di semi e ranghi, creando poi le carte tramite la classe **CardFactory**.

7 Gestione dei Giocatori

Le seguenti classi ed interfacce sono state modellate per la gestione dei giocatori:

7.1 Classe Player (Model)

La classe *Player* rappresenta un giocatore e gestisce la sua mano, la strategia di gioco e fornisce metodi per interagire in partita.

Metodi implementati:

- **addCard(Card card)**: per aggiungere una carta alla mano del giocatore;
- **wantsToHit()** : per determinare se il giocatore desidera chiedere un'altra carta;
- **clearHand()** : per rimuovere tutte le carte dalla mano del giocatore una volta terminata la partita;
- **isHuman()** : per stabilire se il giocatore è un umano oppure un bot o il banco;
- **getHandValue()** : per restituire il valore della somma delle carte nella mano del giocatore, la quale è mostrata a schermo nella *BlackjackView*;
- **getHand()** : per restituire la mano attuale del giocatore;
- **getStrategy()** : attraverso l'interfaccia *PlayerStrategy*, serve per determinare la strategia da applicare per il giocatore che sta svolgendo il turno attuale;

Utilizzo degli Stream:

- **Ottenere il valore della mano**: ho implementato uno Stream per poter restituire il valore della somma delle carte nella mano del giocatore.

7.2 Interfaccia PlayerStrategy (Model)

Questa interfaccia definisce la strategia di gioco per i vari giocatori in partita.

Pattern utilizzato:

- **Strategy**: permette di modellare una strategia di gioco, creando un'interfaccia generica per essere poi estesa nei singoli casi.

Metodo implementato:

- **wantsToHit(List<Card>hand)** : per determinare se il giocatore desidera chiedere un'altra carta basandosi sul valore della sua mano.

7.3 Classe HumanPlayerStrategy (Model)

Questa classe implementa *PlayerStrategy* e definisce la strategia di gioco per il giocatore umano.

Pattern utilizzato:

- **Strategy**: permette di cambiare la strategia di gioco del giocatore umano senza modificare l'interfaccia *PlayerStrategy*.

Metodo implementato proveniente da PlayerStrategy:

- **wantsToHit(List<Card>hand)** : restituisce sempre *false*, indicando che il giocatore umano non desidera chiedere un'altra carta. Questo serve da segnaposto per interagire con l'input proveniente dal giocatore stesso in base alle scelte che egli vuole compiere.

7.4 Classe BotStrategy (Model)

Questa classe implementa *PlayerStrategy* e definisce la strategia di gioco per i bot.

Pattern utilizzato:

- **Strategy**: permette di cambiare la strategia di gioco dei bot senza modificare l'interfaccia *PlayerStrategy*.

Metodo implementato proveniente da PlayerStrategy:

- **wantsToHit(List<Card>hand)** : il bot continua a pescare finché il valore della mano è inferiore a 17.

Utilizzo degli Stream:

- **Conteggio carte**: utilizzo uno Stream che restituisce la somma delle carte nella mano del bot.

7.5 Classe DealerStrategy (Model)

Questa classe implementa *PlayerStrategy* e definisce la strategia di gioco per il banco.

Pattern utilizzato:

- **Strategy**: permette di cambiare la strategia di gioco del banco senza modificare l'interfaccia *PlayerStrategy*.

Metodo implementato proveniente da PlayerStrategy:

- **wantsToHit(List<Card>hand)** : segue la regola standard del Blackjack per il banco, ossia pescare se il valore totale della mano è inferiore a 17.

Utilizzo degli Stream:

- **Conteggio carte**: utilizzo uno Stream che restituisce la somma delle carte nella mano del banco.

8 Gestione Animazioni

La gestione delle animazioni è molto semplice:

- Nel momento in cui il giocatore, il bot o il banco chiedono una carta essa verrà aggiunta mettendola vicino alle altre che possiede nella sua mano;
- Appena verrà aggiunta la carta che il giocatore pesca, le altre che sono già presenti nella sua mano, verranno spostate verso sinistra in modo da fare spazio per le nuove.

9 Note Progettuali Aggiuntive

Questa struttura modulare permette una facile estendibilità e manutenzione del codice, oltre a favorire una separazione e gestione delle responsabilità tra le diverse componenti del progetto.

Inoltre l'utilizzo dei design pattern ha facilitato l'implementazione di funzionalità complesse in modo organizzato.