"Ludomania"

Valerii Sargov
Marco Pertegato
Carlo Michele Nicastro
Salvatore Zammataro

22 giugno 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software "Ludomania", sviluppato per il progetto di laboratorio del corso di Programmazione ad Oggetti dell'anno accademico 2024/2025 all'interno del corso di laurea triennale in Ingegneria e Scienze Informatiche dell'Università di Bologna, mira alla creazione di un mini-casinò. All'interno l'utente potrà scegliere tra varie attività disponibili tra cui Blackjack, Trente et Quarante e Roulette. Dal menù principale è possibile selezionare il gioco a cui si vuole partecipare, aprire un menù dei cosmetici dove si potranno selezionare i vari elementi estetici, cambiare le impostazioni dell'applicazione (risoluzione, lingua, volume).

Blackjack è un gioco che utilizza le carte francesi il cui obiettivo è ottenere una mano il cui punteggio sia il più vicino possibile a 21, senza superarlo. Le carte numerate valgono il loro valore facciale, mentre le figure (re, regina, fante) valgono 10, e gli assi possono essere contati come 1 o 11, a seconda della mano. Il giocatore riceve due carte e può scegliere di "chiedere" altre carte ("hit") o "stare" con quelle già in mano ("stand"). Se la somma delle carte supera 21, il giocatore perde automaticamente. Se il banco ha una mano migliore, il giocatore perde la puntata, ma se il giocatore ha una mano migliore, vince.

Trente et Quarante è anch'esso un gioco con le carte francesi in cui il giocatore scelglie la puntata, che può essere fatta su quattro zone: rosso o nero (rouge ou noire) e colore o inverso (couleur ou enverse). A questo punto il banco scopre una per volta le carte in gioco e le sistema su una fila, che giocherà per il nero e si fermerà quando il valore delle carte sarà maggiore di trenta, le carte numerate valgono il loro valore facciale, mentre le figure (re, regina, fante) valgono 10, gli assi invece valgono 1. In seguito inizierà a

stendere la fila del rosso, fermandosi quando raggiunge anche in questo caso un numero di punti superiore a trenta. Vince la fila che fa meno punti, in caso di parità il colpo è nullo, per stabilire il vincitore tra couleur e enverse si guarda il colore della prima carta della prima fila e si confronta con il colore vincente: se è dello stesso colore avrà vinto couleur, se è differente enverse.

La Roulette Francese consiste in un disco, diviso in 37 settori numerati da 0 a 36 e colorati alternativamente in rosso e nero, mentre lo zero (0) è normalmente colorato di verde; il disco viene fatto ruotare dal gestore del banco (il croupier) che successivamente vi lancia una pallina e si ferma cadendo in uno dei settori numerati, determinando il numero vincente. Le combinazioni su cui è possibile puntare sono svariate:

- Plein (singolo numero) con cui si vince 35 volte la somma puntata
- Cheval (cavallo o coppia di numeri) con cui si vince 17 volte la somma puntata
- Carré (quartina) con cui si vince 8 volte la somma puntata
- Douzaine (dozzina, prima, seconda o terza) con cui si vince 2 volte la somma puntata
- Colonne (colonna, prima, seconda o terza) con cui si vince 2 volte la somma puntata.

Ci sono poi due ulteriori tipi di puntata che in caso di vittoria, restituiscono una volta la somma puntata e sono:

- Pair ou Impair, o anche Even or Odd, ovvero numeri pari o dispari
- Rouge ou Noir, ovvero i numeri rossi o neri.
- Manque ou Passe, ovvero i numeri dall'1 al 18 e dal 19 al 36.

Requisiti funzionali

- All'avvio del gioco verrà visualizzata una schermata principale che include un menù in cui si può scegliere il tavolo cui vuole unirsi.
- Dal menù principale deve essere possibile aprire i vari menù: quello delle impostazione e quello dei cosmetici dove si potranno selezionare degli elementi cosmetici, inoltre deve essere possibile chiudere l'applicazione
- All'interno della finestra delle impostazioni potrà scegliere se attivare o meno gli effetti sonori e la musica e regolarne il volume, e cambiare la lingua in inglese o italiano.

- Durante il gioco l'utente deve poter tenere sotto controllo il proprio credito.
- Allatto di effettuare una puntata l'utente deve poter scegliere l'ammontare di fiches, in accordo col credito residuo, da giocare.

Requisiti non funzionali

- Il gioco deve essere fluido e responsivo alle azioni dell'utente.
- Il programma deve funzionare correttamente nei tre principali sistemi operativi:Linux, Windows e MacOs.
- Il programma deve garantire un minimo di sicurezza per l'accesso ai wallet per garantire la consistenza dei dati

1.2 Modello del Dominio

Un tavolo potrà gestire ogni fase del gioco : l'inizio e la fine della partita, la distribuzione delle carte, le scommesse dei giocatori e la paga a fine gioco. Ogni giocatore avrà a disposizione un proprio wallet dal quale verrà prelevato denaro al momento del piazzamento di una scommessa e versato in caso di vincita. Per ogni gioco sarà presente un croupier che si occuperà di fornire quanto necessario per poter iniziare la partita, nel caso di Blackjack e Trente Et Quarante verranno fornite delle carte, mentre per quanto riguarda la roulette si occuperà dell'estrazione dei numeri della ruota. Prima che ciò avvenga ad ogni giocatore verrà data la possibilità di effettuare un tipo di bet, che varia in base al gioco da lui deciso, nella Roulette sono presenti 5 tipi di puntate con pagamenti differenti in base alla scelta, in Trente Et Quarante sono disponibili 4 bet che pagano in egual maniera, mentre nel Blackjack vi è una sola bet con un pagamento diverso in base all'esito del gioco, una retribuzione base o una speciale quando si verifica Blackjack. Il flusso dei vari giochi consiste nel susseguirsi di puntate da parte dei giocatori ed estrazione da parte dei croupier ad eccezione del Blackjack nel quale il giocatore potrà, durante il suo turno, richiedere una o più ulteriori carte finché non deciderà di smettere o sballerà. Infine quando il gioco verrà ritenuto terminato, spetta al croupier stabilire quali bet verranno pagate e riferirlo al tavolo che si occuperà del pagamento delle vincite ad ogni singolo giocatore.

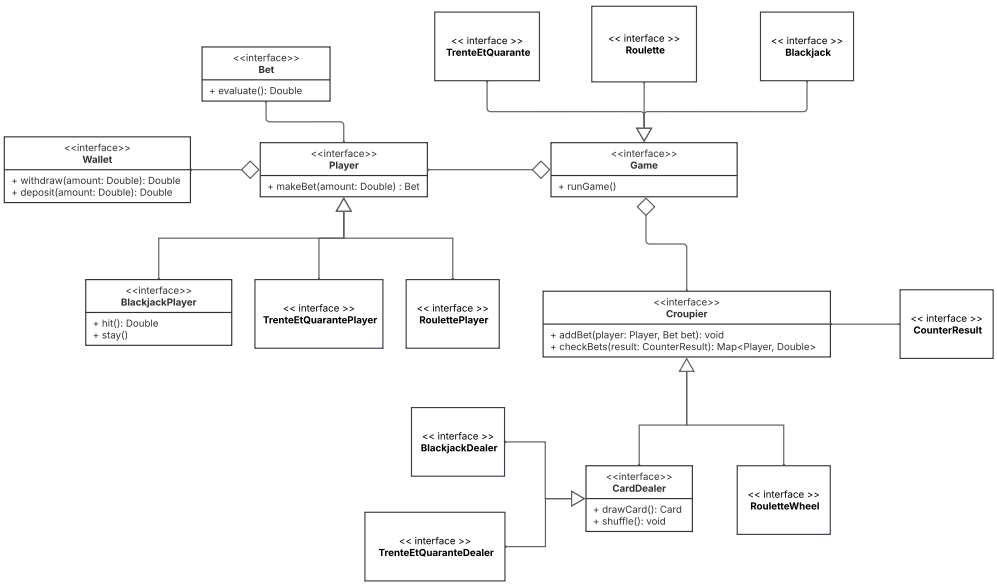


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettura adottata per il progetto Ludomania segue il pattern architetturale MVC (Model Control View). Ciò consente una chiara separazione delle responsabilità e una maggiore flessibilità in fase di sviluppo, permettendoci di sostituire in blocco la view senza richiedere alcuna modifica nel controller e nel Model. L'architettura si basa su cinque componenti principali:

- ViewBuilder
- Handler
- Controller
- Game
- Manager

Il ViewBuilder ha il ruolo di generare la User Interface quindi, di gestire tutto ciò che riguarda la visualizzazione dei componenti all'interno della finestra. L' Handler si occupa di gestire tutte le interazioni che ha l'utente con la View attraverso i suoi metodi handle e fa aggiornare la view di conseguenza. Il Controller implementa l'Handler e permette all'Handler di far comunicare View e Model Game e Manager rappresentano il modello dell'applicazione, rispettivamente la logica interna di uno dei 3 giochi o la logica di funzionamento dell'applicazione.

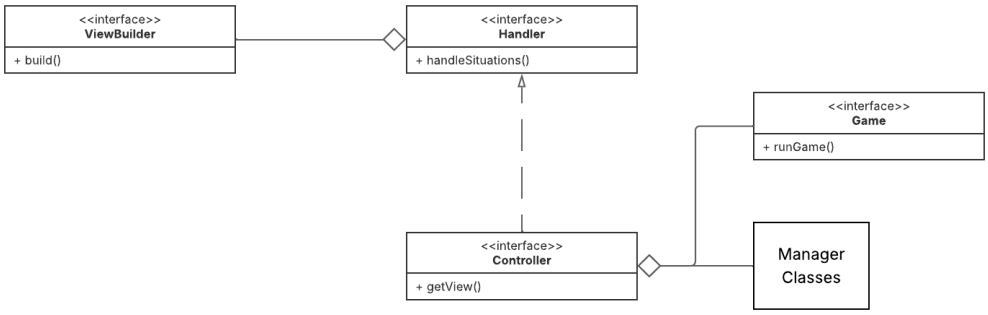
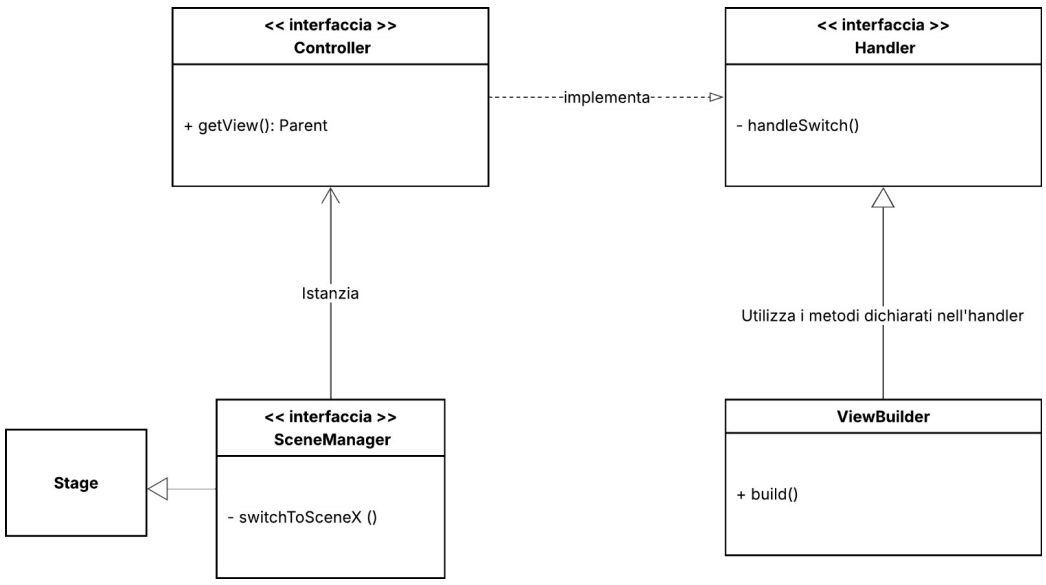


Figura 2.1: Rappresentazione UML dell'architettura MVC realizzata per Ludomania.

2.2 Design dettagliato

2.2.1 Valerii Sargov

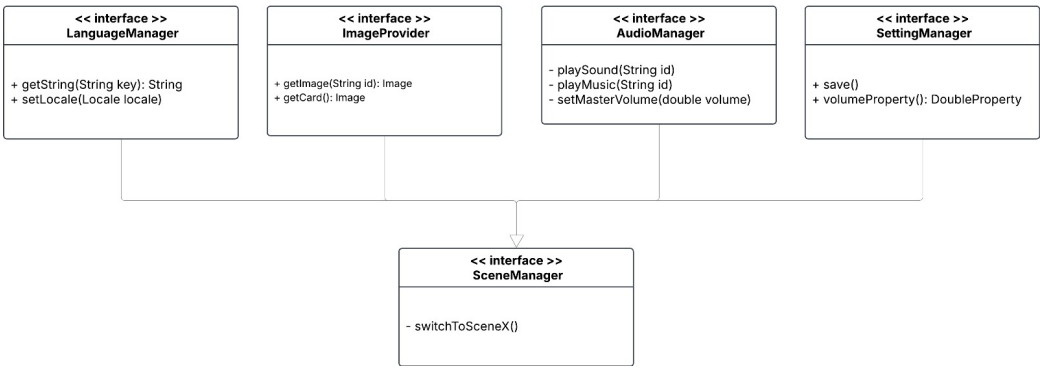
Gestione delle schermate



Problema In applicazioni con molte viste, delegare il cambio di schermata a ciascun controller porta rapidamente a codice duplicato e difficile da mantenere. Inoltre, ogni modifica alla navigazione richiede interventi in più punti del sistema, compromettendo la scalabilità e la coerenza dell'interfaccia.

Soluzione Per risolvere questo problema, è stato introdotto un componente centrale chiamato SceneManager, incaricato di gestire tutte le transizioni tra le schermate. I controller non cambiano direttamente la scena, ma delegano le azioni dell'utente a interfacce Handler, che vengono implementate da componenti esterni. Questi, a loro volta, possono richiamare lo SceneManager per effettuare la transizione. Questo approccio riduce l'accoppiamento tra controller e logica di navigazione, favorendo la modularità e la manutenibilità.

Gestione centralizzata dei servizi



Problema Alcune funzionalità trasversali, come la lingua dell'interfaccia o le preferenze (settaggi) utente, non possono essere delegate a singoli componenti senza rischiare codice duplicato o comportamenti incoerenti.

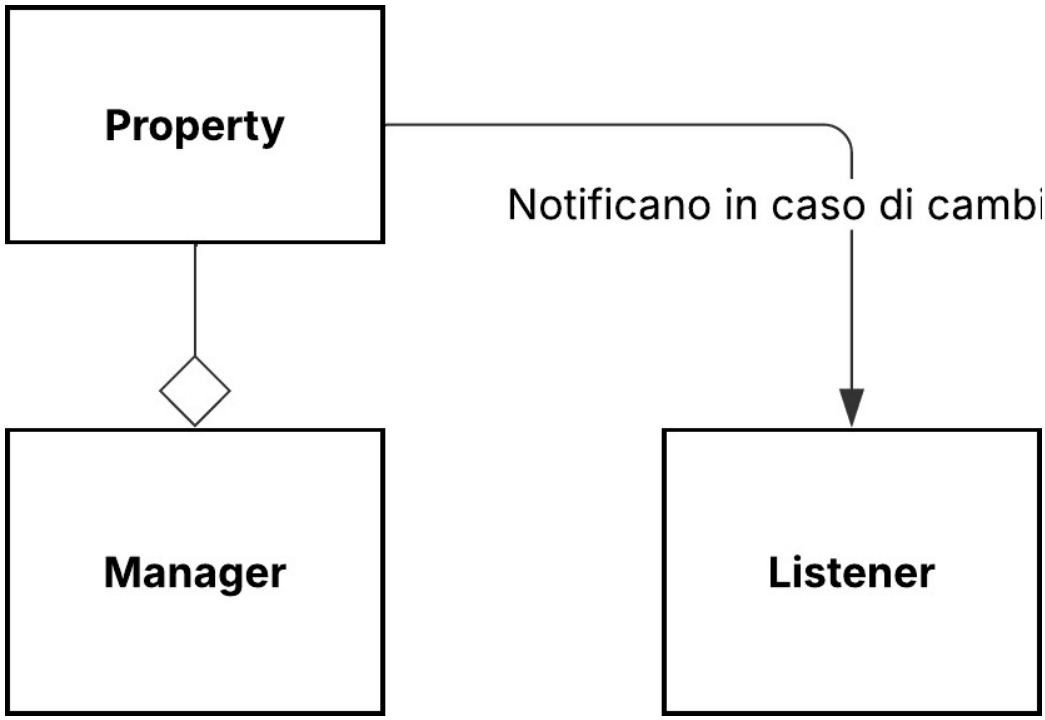
Soluzione Per migliorare l'organizzazione, sono stati introdotti manager dedicati a ciascuna funzionalità generale dell'applicazione:

- LanguageManager: gestisce la lingua dell'interfaccia tramite ResourceBundle, notificando i componenti interessati a ogni cambiamento.
- ImageProvider: fornisce risorse grafiche coerenti con il tema selezionato (es. sfondi, carte, fiche), nascondendo la complessità interna della gestione visiva [vedremo meglio dopo].
- AudioManager: controlla musica ed effetti sonori, mantenendo separata la logica audio dalla presentazione.
- SettingsManager: salva e carica le preferenze utente (lingua, volume, risoluzione, fullscreen), utilizzando l'API Preferences di Java.

Tutti i manager vengono creati nella classe principale dell'applicazione (Ludomania) e passati ai componenti che ne hanno bisogno tramite injection manuale, evitando l'uso di singleton. Il SceneManager funge da coordinatore, distribuendo i manager dove necessario (ad esempio a ViewBuilder e Handler). Alcuni manager legati alla UI, come LanguageManager e ImageProvider, possono essere forniti direttamente alle viste, che così si aggiornano autonomamente. Questa architettura offre diversi vantaggi:

- Inversion of Control: i componenti ricevono le dipendenze dall'esterno e non le istanziano direttamente.
- Single Responsibility Principle: ogni manager ha un compito ben definito.
- Open/Closed Principle: è possibile estendere i comportamenti dei manager senza modificarne l'implementazione.

Gestione reattiva delle proprietà

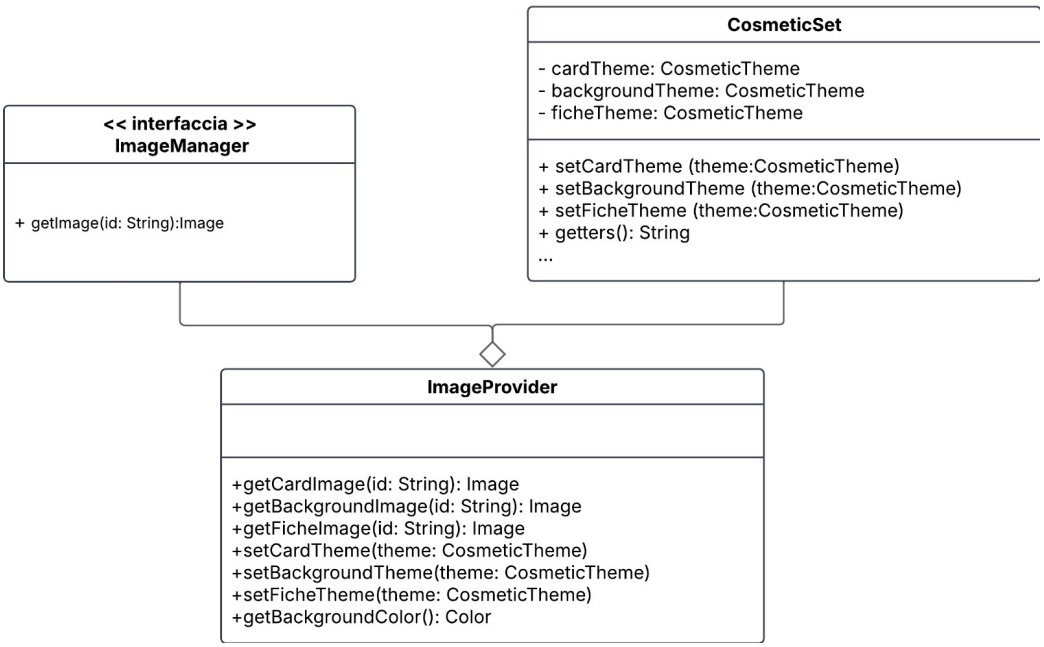


Problema Quando una proprietà condivisa (come lingua, volume o risoluzione) cambia, è necessario aggiornare automaticamente i componenti

interessati. Farlo manualmente renderebbe il sistema fragile e soggetto a errori.

Soluzione Per affrontare questa esigenza, è stato adottato l'Observer Pattern. I manager che espongono proprietà dinamiche (come LanguageManager e SettingsManager) notificano automaticamente i listener registrati quando una proprietà cambia. Nel caso di proprietà compatibili con il sistema di binding di JavaFX (es. testi delle etichette o dimensioni dello stage), viene utilizzato direttamente il data binding, così da ottenere aggiornamenti automatici e reattivi. Ad esempio, quando l'utente cambia la lingua, le etichette della UI si aggiornano istantaneamente senza ulteriori istruzioni. Anche impostazioni come la modalità fullscreen o la risoluzione sono modellate con javafx.beans.property, e i componenti interessati reagiscono ai cambiamenti in modo automatico. Lo stesso approccio è stato esteso alla gestione dei temi cosmetici: le preferenze dell'utente relative al tema attivo vengono osservate centralmente. Quando il tema cambia, le view vengono notificate e possono aggiornare esplicitamente la loro grafica. In questo modo, si evita di duplicare la logica di aggiornamento nei singoli componenti, pur richiedendo un refresh controllato della vista per riflettere correttamente il nuovo tema. Questa gestione reattiva garantisce un comportamento coerente, migliora l'affidabilità del sistema e semplifica il codice dell'interfaccia.

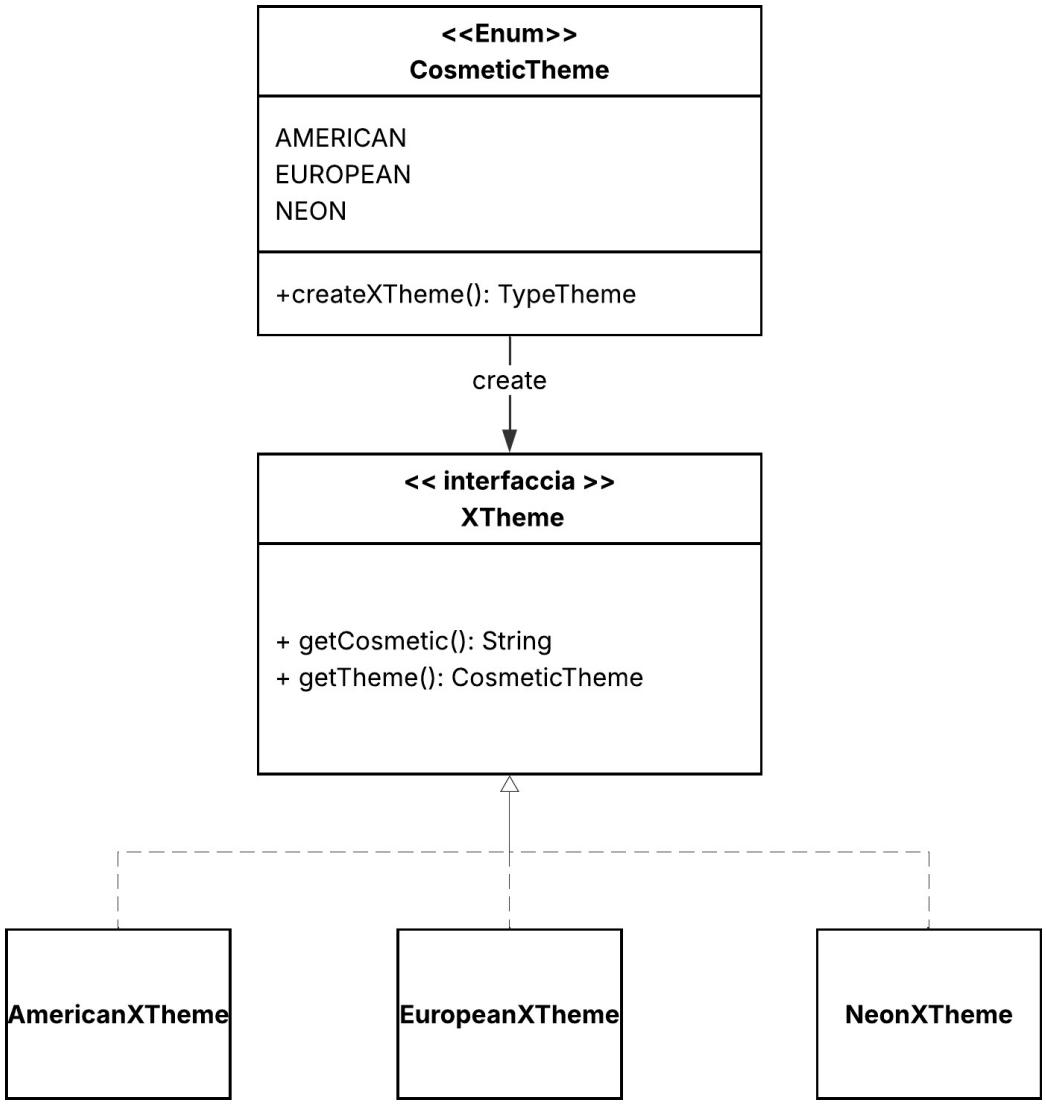
Gestione dei temi cosmetici



Problema L'ImageManager, da solo, non era sufficiente a gestire temi grafici complessi. Caricare manualmente le immagini per ogni tema avrebbe introdotto duplicazioni e una logica frammentata, rendendo difficile aggiungere o modificare i temi visivi.

Soluzione Per affrontare questo problema è stato introdotto il componente ImageProvider, che funge da facciata per la gestione centralizzata delle risorse grafiche. Questo approccio segue il Facade Pattern: le viste interagiscono con un'interfaccia semplice, mentre la complessità della selezione delle risorse grafiche in base al tema è incapsulata all'interno del ImageProvider. Le viste ottengono così gli elementi grafici necessari (carte, fiche, sfondi) tramite metodi come getCard(String type), getFiche(int value) e getBackground(). In questo modo, l'interfaccia utente resta indipendente dal tema attivo e dalla logica di caricamento delle risorse, migliorando la modularità e facilitando l'estensione del sistema con nuovi temi.

Estendibilità dei temi grafici



Problema L'aggiunta di nuovi temi grafici doveva essere semplice e non doveva compromettere il codice esistente. Un approccio rigido o basato su condizionali avrebbe reso il sistema difficile da estendere e mantenere.

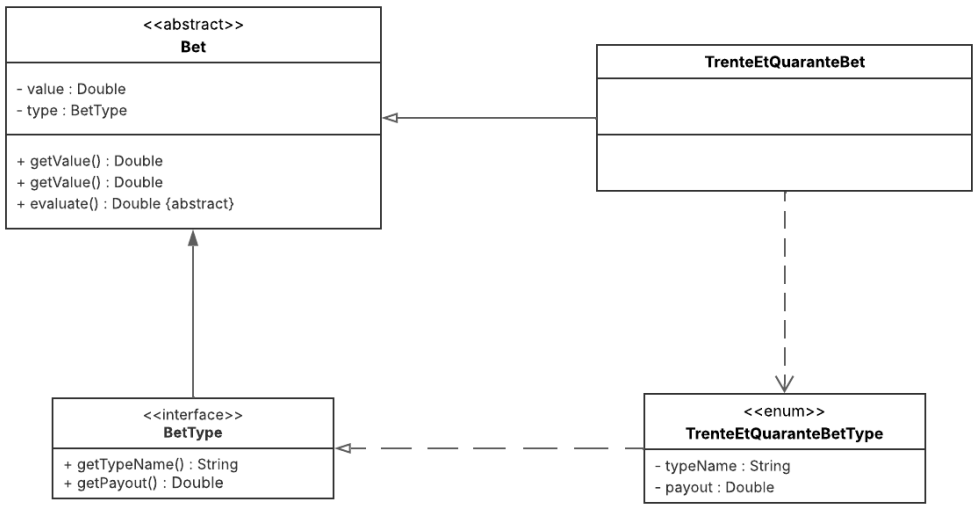
Soluzione Per garantire flessibilità, è stato introdotto un enum polimorfo chiamato CosmeticTheme, che rappresenta i diversi temi disponibili (EUROPEAN, AMERICAN, NEON, ...). Ogni valore dell'enum implementa metodi specifici per creare i componenti grafici principali:

- CardTheme per le carte
- BackgroundTheme per gli sfondi
- FicheTheme per le fiche

Ogni tema funge da factory, fornendo le risorse grafiche corrispondenti. Questo approccio rispetta il Single Responsibility Principle, perché le classi concrete sono responsabili solo della grafica di uno specifico elemento. L'ImageProvider si basa su CosmeticTheme per fornire le risorse richieste dalle viste. Questo consente di aggiungere nuovi temi senza modificare il codice esistente, evitare condizionali sparsi nel sistema e mantenere l'interfaccia pulita e stabile per le viste. Grazie a questa soluzione, il sistema è pronto ad accogliere nuovi temi o aggiornamenti grafici senza richiedere interventi invasivi, migliorando l'estensibilità e la coesione del codice.

2.2.2 Marco Pertegato

Astrazione delle Scommesse e Tipi di Scomessa

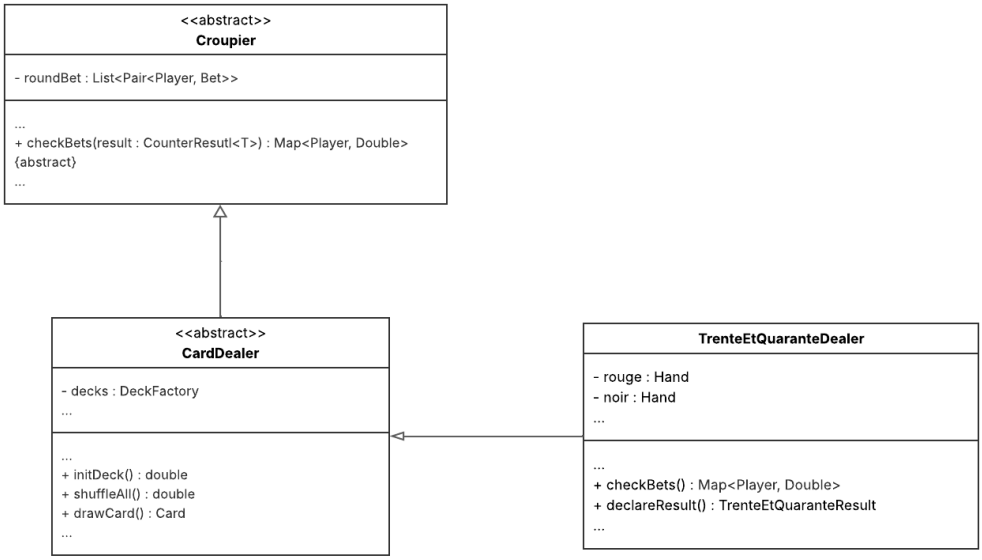


Problema Il gioco di Trente et Quarante ha diversi tipi di scommesse ognuno con una propria logica di valutazione e payout (Rouge, Noir, Couleur, Enverse, Draw), e se dovessi creare una classe per ogni scommessa il codice diventerebbe ripetitivo e difficile da estendere per nuovi tipo di scommesse.

Soluzione

- Viene definita un'interfaccia BetType con metodi getTypeName() e getPayout().
- Un'enumerazione TrenteEtQuaranteBetType implementa BetType e definisce i vari tipi di scommessa specifici del gioco, ciascuno con il proprio nome e payout.
- Una classe astratta Bet gestisce il valore della scommessa e il tipo di scommessa generico (BetType).
- La classe TrenteEtQuaranteBet estende Bet e implementa il metodo evaluate() , utilizzando il getPayout() del TrenteEtQuaranteBetType specifico per calcolare il guadagno. Questo permette di aggiungere facilmente nuovi tipi di scommessa estendendo l'enumerazione TrenteEtQuaranteBetType o creando nuove implementazioni di BetType e Bet senza modificare il codice esistente che gestisce le scommesse.

Gestione del Mazziere e delle Carte



Problema Diversi giochi di casinò basati su carte condividono funzionalità comuni per un mazziere (es. gestione dei mazzi, estrazione di carte, collezione di scommesse), ma hanno logiche specifiche per la valutazione delle mani e delle scommesse.

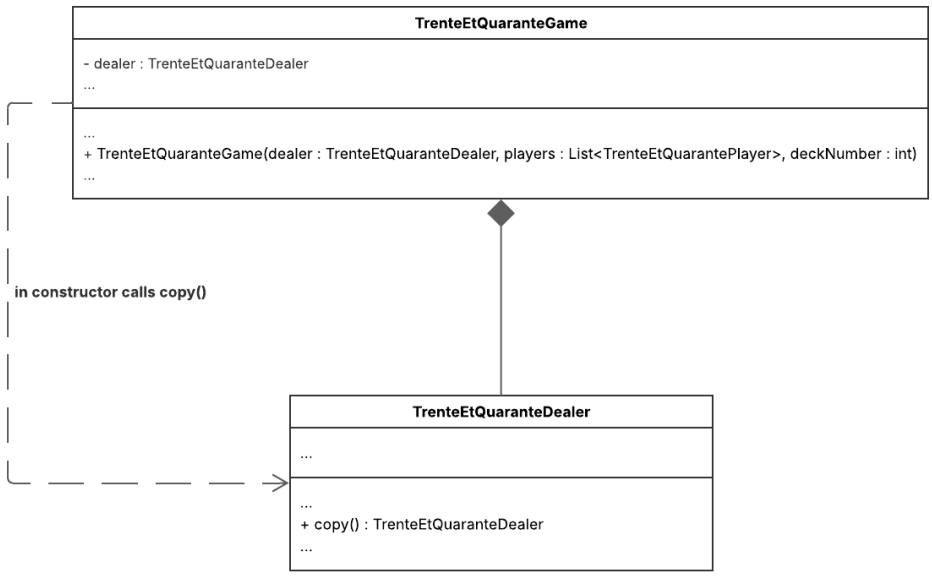
Soluzione Viene creata una gerarchia di classi astratte per i mazzieri.

- Croupier<T>(Classe Astratta Base): Definisce il comportamento generico di un "croupier" che gestisce le scommesse. Ha metodi per aggiungere e cancellare scommesse. Contiene un metodo astratto checkBets, che deve essere implementato dalle sottoclassi per valutare le scommesse in base al risultato specifico del gioco (CounterResult<T>).

- CardDealer<T>(Sottoclasse Astratta): Estende Croupier e aggiunge funzionalità comuni per i giochi di carte, come la gestione di un DeckFactory ,initDeck() , shuffleAll() e drawCard()

- TrenteEtQuaranteDealer (Implementazione Concreta): Estende CardDealer e fornisce l'implementazione concreta per il gioco Trente et Quarante. Gestisce le mani specifiche del gioco (Rouge e Noir) , calcola i totali delle mani , e implementa la logica di valutazione delle scommesse (checkBets) e la dichiarazione del risultato (declareResult). Questo approccio gerarchico permette il riutilizzo del codice comune a tutti i giochi di carte e la specializzazione per ogni gioco. Il metodo checkBets è un esempio del Template Method, dove la struttura generale della valutazione delle scommesse è definita, ma i dettagli specifici del risultato sono lasciati all'implementazione concreta

Copia dello Stato del Dealer



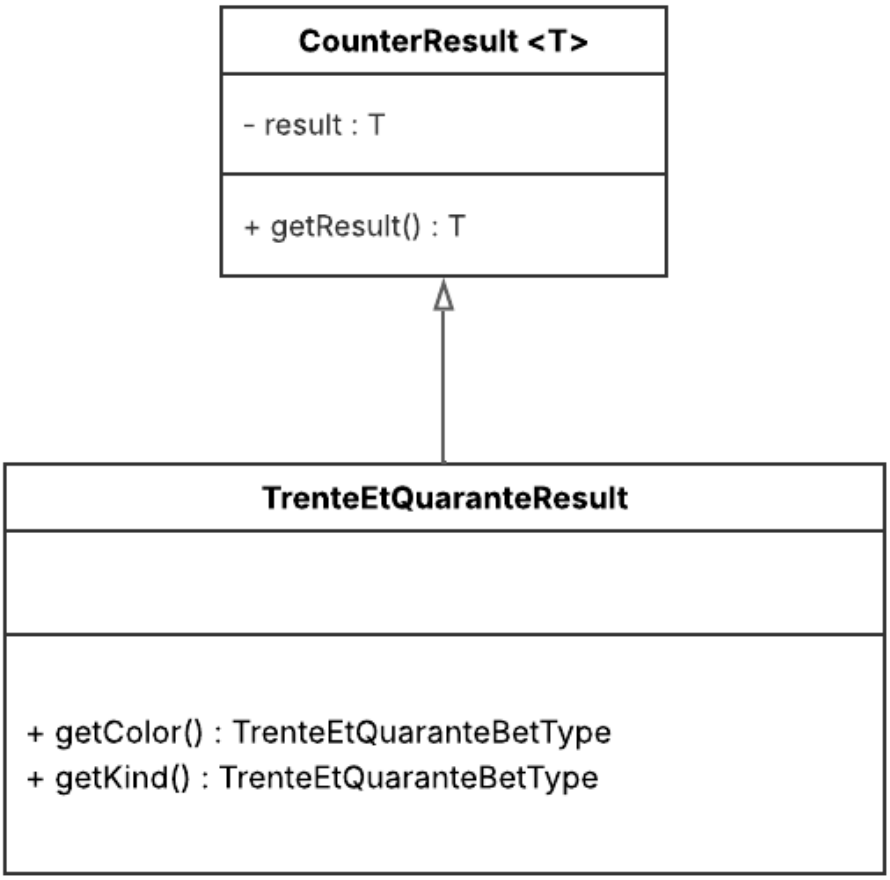
TrenteEtQuaranteGame(final TrenteEtQuaranteDealer dealer,
final List<TrenteEtQuarantePlayer> players, final int deckNumber)

Problema La classe TrenteEtQuaranteGame riceve un TrenteEtQuaranteDealer nel suo costruttore. Se il dealer venisse modificato esternamente dopo la creazione del gioco, lo stato del gioco potrebbe diventare inconsistente.

Soluzione Nel costruttore di TrenteEtQuaranteGame, viene creata una copia del dealer passato come argomento tramite il metodo dealer.copy().

- Il metodo copy() in TrenteEtQuaranteDealer crea una nuova istanza di TrenteEtQuaranteDealer e copia lo stato rilevante (scommesse del round, carte nelle mani, valori delle mani).
- Questo garantisce che la TrenteEtQuaranteGame lavori su una propria istanza del mazziere, isolando il suo stato e prevenendo modifiche indesiderate dall'esterno. Sebbene non sia un'implementazione formale del pattern Prototype con un'interfaccia Cloneable, il comportamento è simile: un oggetto crea una copia di sé stesso.

Astrazione dei risultati di gioco



Problema Rappresentare un risultato generico di un gioco (o di una "mano" del banco) che può assumere diverse forme a seconda del gioco specifico. Ad esempio, il risultato potrebbe essere un'enumerazione per un gioco, un valore numerico per un altro, o un oggetto complesso per un terzo. Senza una classe generica, si dovrebbe creare una classe risultato separata per ogni tipo di gioco, portando a duplicazione di codice e mancanza di coerenza nell'interfaccia.

Soluzione La classe CounterResult<T>agisce come un wrapper generico per qualsiasi tipo di risultato (T): incapsula un'istanza del tipo di risultato

specifico (private final T result) e fornisce un metodo generico getResult() per accedere a questo risultato. Questo approccio permette di:

- Riutilizzare il codice: La classe CounterResult può essere riutilizzata in diversi giochi di casinò per incapsulare i loro specifici tipi di risultato.
- Mantenere la flessibilità: Non vincola la natura del risultato a un tipo specifico finché non viene istanziata. Questo rispetta il principio Open/Closed, in quanto la classe CounterResult è "aperta" all'estensione tramite la parametrizzazione del tipo T ma "chiusa" alle modifiche interne per gestire nuovi tipi di risultati.

2.2.3 Carlo Michele Nicastro

Struttura modulare e ben organizzata

Il codice è suddiviso in classi con responsabilità ben distinte, secondo i principi Single Responsibility e Open/Closed. Ogni classe incapsula una parte logica ben definita del gioco (dealer, giocatore, gestione del mazzo, logica di vittoria, etc.).

Leggibilità e manutenzione

Il codice è leggibile, ben commentato, e organizzato in metodi brevi e coerenti. Questo lo rende facilmente estendibile e manutenibile, soprattutto per futuri sviluppi come UI o nuove regole di gioco.

Gestione accurata delle regole del BlackJack

Le regole base del Blackjack sono correttamente implementate:

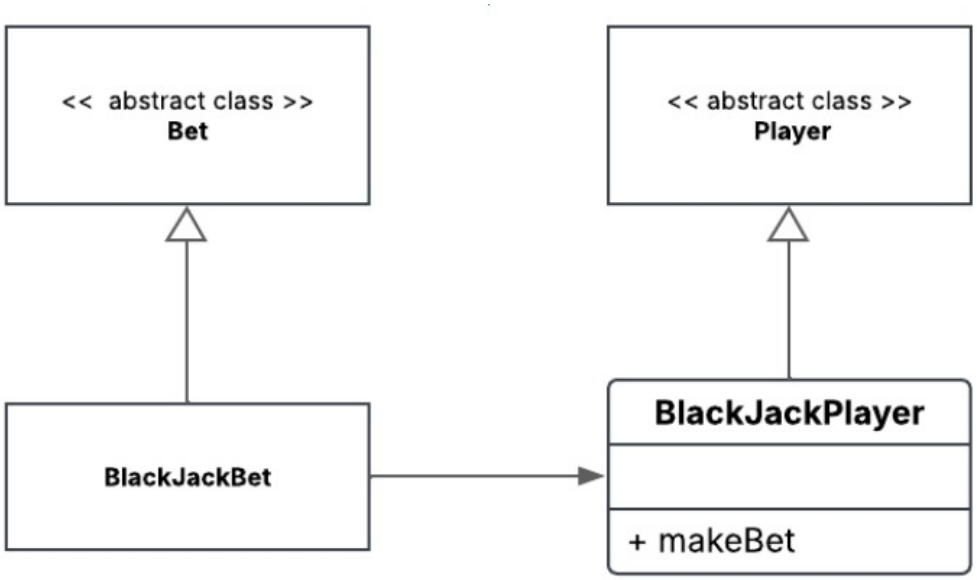
- Il dealer si ferma a 17 o più.
- Gestione dell'asso come 1 o 11.
- Valutazione corretta di vincita, pareggio o sconfitta.
- Gestione del BlackJack (21 con due carte).

Uso efficace delle strutture dai di Java

L'uso di Map, List, UUID e Enum rende il codice robusto e orientato agli oggetti. I tipi generici permettono flessibilità nella gestione di collezioni.

Astrazione e riuso tramite interfacce

L'interfaccia Game e l'uso di una classe base CardDealer permettono l'estensione del motore di gioco ad altri giochi di carte in futuro, favorendo il riutilizzo del codice.

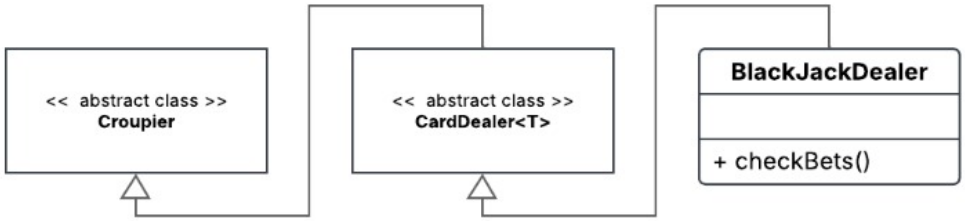


Problema Nel gioco del Blackjack, anche se il sistema di scommessa è più semplice rispetto a giochi come la Roulette, è comunque importante separare la logica relativa alle scommesse dalla logica di gioco principale. In prospettiva, si potrebbero introdurre varianti di scommessa come assicurazione (insurance), raddoppio (double down) o scommesse secondarie (side bets). Gestire tutte queste logiche in un'unica classe renderebbe il codice rigido e difficile da estendere.

Soluzione Viene definita un'interfaccia BetType con i metodi getTypeName() e getPayout() che rappresentano, rispettivamente, il nome del tipo di scommessa e il moltiplicatore della vincita. Un'enumerazione BlackJackBetType implementa BetType e definisce i tipi di scommessa standard del Blackjack (BASE, BLACKJACK, PUSH, LOSE), ciascuno con la propria logica di payout. Una classe astratta Bet gestisce l'importo della scommessa e il tipo di scommessa generico (BetType). La classe BlackJackBet estende Bet e implementa un metodo evaluate(), che calcola il guadagno usando il payout associato al tipo di scommessa selezionato (BlackJackBetType.getPayout()). Questo approccio permette di estendere facilmente il sistema introducendo

nuove tipologie di scommessa (es. Side Bet), senza modificare il codice esistente, in conformità con il principio Open/Closed (è possibile estendere i comportamenti dei manager senza modificarne l'implementazione).

Gestione del Mazziere

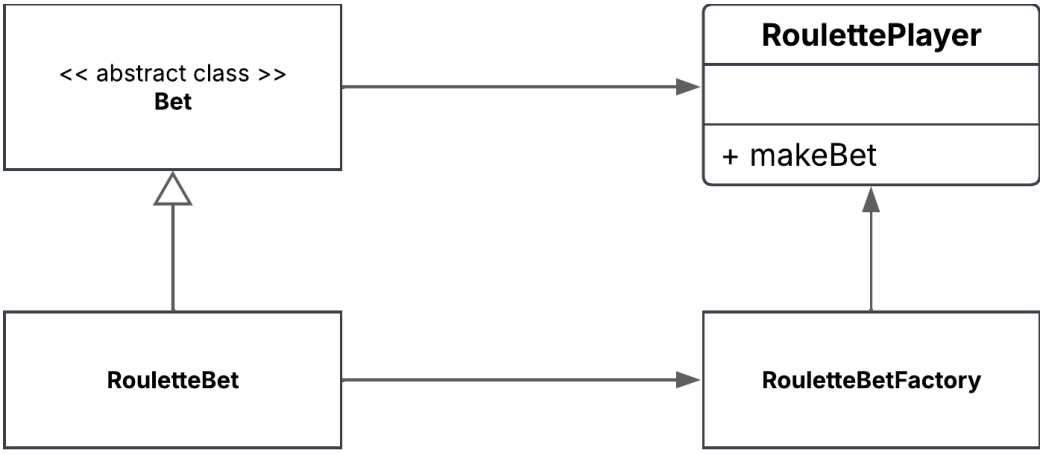


Problema In un gioco come il Blackjack, la gestione del mazziere (dealer) include sia logiche comuni a tutti i giochi di carte (es. mescolare il mazzo, pescare carte) sia regole specifiche del Blackjack (es. il dealer si ferma a 17, conteggio dell'asso come 1 o 11, ecc.). Fondere tutto in un'unica classe rende difficile il riuso del codice per altri giochi.

Soluzione Una classe astratta CardDealer<T>estende Croupier e fornisce funzionalità comuni: gestione del mazzo tramite una DeckFactory, inizializzazione e rimescolamento (initDeck(), shuffleAll()), pescaggio di carte (drawCard()). La classe BlackJackDealer implementa CardDealer<BlackJackOutcomeResult>, e contiene la logica specifica del Blackjack: mantiene lo stato delle mani del giocatore e del dealer, gestisce il punteggio (con gestione intelligente degli assi), implementa la regola che il dealer si ferma con 17 o più, valuta le mani e determina il risultato (checkBets()). Questa separazione favorisce riuso, specializzazione, e rende il codice più semplice da estendere o adattare ad altri giochi di carte.

2.2.4 Salvatore Zammataro

Astrazione delle Scomesse e Tipi di Scommessa



Problema nel gioco della Roulette è possibile piazzare diversi tipi di scommesse sul tavolo di gioco. Il piazzamento di tali scommesse e l'ammontare del loro valore sono a discrezione del giocatore. Il gioco qui implementato considera 11 tipi di scommesse possibili, la cui differenza principale risiede nella valutazione del loro successo. È possibile individuare due macrogruppi in cui suddividerle:

- scommesse su uno o più numeri
- scommesse su un insieme ben definito e non modificabile di valori

Del primo gruppo fanno parte i seguenti tipi di scommesse:

- scommesse su uno o più numeri
- scommesse su un insieme ben definito e non modificabile di valori
- plein
- cheval
- carre
- colonne
- douzaine.

Nel secondo gruppo troviamo:

- Rouge
- Noir
- Pair
- Impair
- Passe
- Manque.

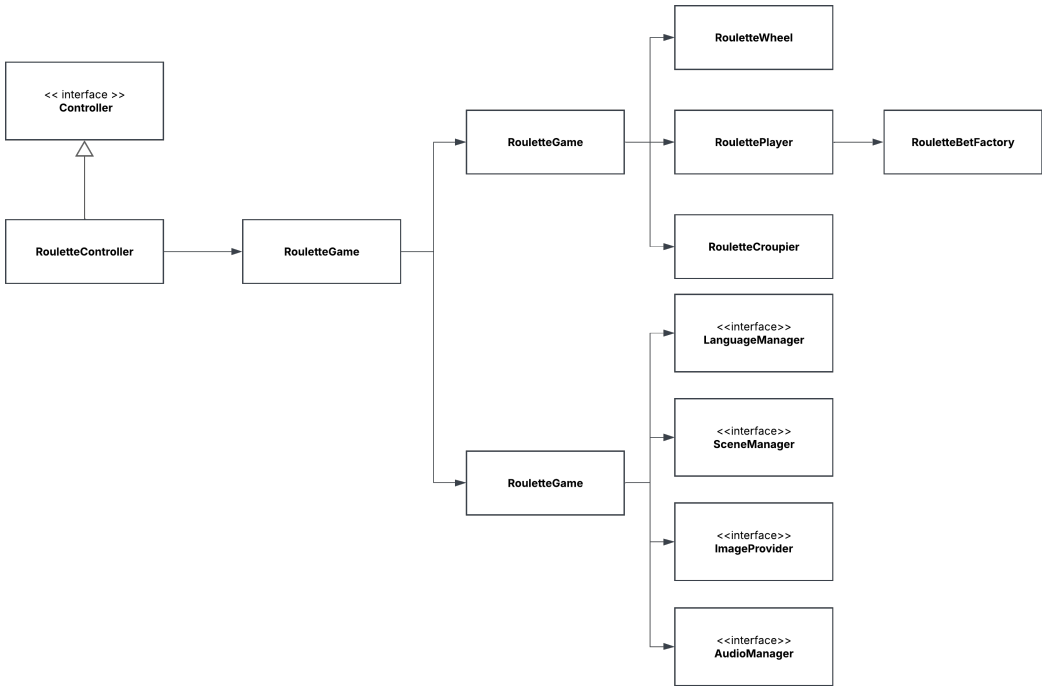
Appare evidente, già da questa breve analisi, che modellare una classe per ciascuna di esse comporterebbe un'eccessiva ridondanza del codice con conseguente mancanza di robustezza dello stesso e difficile manutenibilità.

Soluzione Per ovviare ai problemi sopra esposti si è deciso di seguire un processo di sviluppo basato sul design pattern Factory Method. Tale approccio facilita il riutilizzo del codice e, se correttamente implementato, consente anche una futura espansione dello stesso.

Di seguito gli attori del pattern:

- Creator: RoulettePlayer.La classe RoulettePlayer implementa la classe astratta Player, la quale rappresenta l'interfaccia per il Creator in tutti e tre i giochi dell'applicazione, attraverso la dichiarazione del metodo makeBet.
- ConcreteCreator: RouletteBetFactory.Si occupa di creare gli oggetti Bet richiesti.
- Product: RouletteBet.Rappresenta l'oggetto della factory. Implementa a sua volta la classe astratta Bet, comune a tutti i giochi dell'applicazione.

Gestione delle logica di gioco e comunicazione con il Controller



Un altro problema affrontato durante lo sviluppo del progetto è la gestione della logica del gioco e la comunicazione tra il Model e il Controller. Come anticipato, per lo sviluppo è stato adottato il design pattern MVC, che permette la separazione degli aspetti implementativi di un applicativo in tre componenti, ciascuno con il suo ruolo.

Problema il punto qui affrontato riguarda l'implementazione del model che deve gestire numerose interazioni con il controller, le quali possono in certi casi richiedere l'esecuzione di operazioni complesse, o comunque lunghe, e potenzialmente riguardanti aspetti diversi dell'applicazione.

Soluzione poiché l'implementazione in una sola di classe di tutte le operazioni avrebbe reso il codice pesante e poco espandibile, si è deciso di applicare il design pattern Facade.

Di seguito i principali attori del pattern:

- Client: RouletteController. Si occupa di gestire le interazioni con la view e richiamare i corrispettivi metodi del model.
- Facade: RouletteGame. Rappresenta l'interfaccia tra il Controller e le classi che contengono l'effettiva implementazione della logica di gioco.

- Classes: RouletteGameManager, RouletteSceneManager. Contengono rispettivamente la logica del game e della gestione dell'interfaccia utente. A loro volta queste richiamano altre classi che implementano aspetti più particolari dell'applicazione

L'utilizzo di tale approccio ha permesso una semplificazione delle classi ed inoltre ne permette la riusabilità e manutenibilità.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In questo progetto abbiamo scelto di testare le principali classi del Model e alcuni dettagli della View attraverso una suite di test automatici JUnit. Riportiamo a seguire una breve descrizione dei test sviluppati:

- WalletImplTest: verifica il comportamento della classe WalletImpl, che gestisce il saldo del portafoglio del giocatore. In particolare: controlla che il saldo iniziale venga impostato correttamente alla creazione, verifica che un deposito aumenti correttamente il saldo, e verifica che un prelievo venga eseguito solo se sufficiente, senza andare in negativo, mantenendo il saldo invariato in caso contrario.

- CosmeticThemeTest:verifica il corretto comportamento dell'enum CosmeticTheme, responsabile della creazione dei temi grafici. In particolare, controlla che: ogni tema (EUROPEAN, AMERICAN, NEON) crei le rispettive istanze corrette di card, background e fiche, che il metodo fromId(String) restituisca il tema corretto per ID validi, e ritorni il tema di default (EUROPEAN) per ID non validi o null.

- SceneManagerImplBindingTest: verifica i binding automatici gestiti da SceneManagerImpl controllando che i cambiamenti nelle impostazioni utente (volume, lingua, risoluzione) si riflettano correttamente sui rispettivi componenti (AudioManager, LanguageManager, Stage). In questo modo si assicura che le modifiche siano applicate in tempo reale senza interventi manuali nei controller, favorendo modularità e reattività dell'interfaccia.

- TrenteEtQuaranteBetTest: verifica il corretto comportamento della classe TrenteEtQuaranteBet, controllando la corretta inizializzazione e il corretto valore di pagamento in caso quella bet risulti vincente.

- TrenteEtQuarantePlayerTest: controlla che il player possa piazzare solo bet valide e che possa farlo solo quando il suo credito è sufficente.

- TrenteEtQuaranteDealerTest: si occupa di controllare il corretto comportamento del TrenteEtQuaranteDealer verificando che aggiunga alla lista dei vincitori i player corretti e la vincita corrispettiva. Verifica inoltre la correttezza della valutazione del risultato alla fine di un wound.

- TrenteEtQuaranteGameTest: breve classe di test sviluppata per sistemare un bug, che controlla che dopo la runGame() nelle mani del dealer vengano effettivamente inserite le carte.

- BlackJackBetTest: verifica il comportamento della classe BlackJackBet, che rappresenta una scommessa nel gioco del BlackJack. In particolare: verifica che, dopo la creazione dell'oggetto, valore e tipo della scommessa siano corretti; che il payout associato al tipo di scommessa (BlackJackBetType) sia appropriato; e che il metodo evaluate() calcoli correttamente l'importo restituito in base al valore e all'esito della scommessa.

- BlackJackPlayerTest: verifica il comportamento della classe BlackJackPlayer, che rappresenta un giocatore di BlackJack dotato di un portafoglio (WalletImpl) e la capacità di effettuare scommesse. In particolare: verifica che una scommessa valida con tipo BlackJackBetType venga creata correttamente, con valore corretto e saldo del portafoglio ridotto in modo appropriato; che venga sollevata un'eccezione per tipi non validi o fondi insufficienti, lasciando invariato il saldo in entrambi i casi.

- BlackJackDealerTest: verifica il comportamento della classe BlackJackDealer, che gestisce le puntate, le mani e i risultati del banco nel gioco del BlackJack. In particolare: verifica che il costruttore salvi correttamente le puntate iniziali, che si ripristini lo stato di banco e giocatori, e che si aggiunga correttamente una carta alla mano. Controlla che si aggiornino i totali, che venga rilevato correttamente il limite del banco e che si gestisca correttamente i vari esiti, senza svuotare inavvertitamente la mappa locale.

- BlackJackGameTest: verifica il comportamento della classe BlackJackGame, che gestisce il flusso completo di una partita a BlackJack tra un giocatore e il dealer. In particolare: verifica che la partita venga inizializzata correttamente con un giocatore e un dealer; che, dopo l'inizio del round e la scommessa, distribuisca le carte iniziali; assegni almeno due carte a ciascuno; e che il risultato sia generato correttamente e non sia null.

- RouletteBetTest: verifica il comportamento della classe RouletteBetfactory, che rappresenta una factory per le scommesse nel gioco della Roulettei. Ne viene controllata la corretta creazione e la valutazione del loro successo.

- RoulettePlayerTest: verifica il comportamento della classe RoulettePlayer, che rappresenta un giocatore di Roulette. Viene controllato la corretta gestione del portafogli e dell'ammontare della prossima somma da scommettere.

- RouletteCroupierTest: verifica il comportamento della classe RouletteCroupiet, che mantiene le scommesse e restituisce un insieme delle vincite, associate al corrispettivo giocatore, se ve ne sono.

3.2 Note di sviluppo

3.2.1 Valerii Sargov

Utilizzo di lambda per binding in SceneManagerImpl

Permalink: `https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a`
`src/main/java/ludomania/core/impl/SceneManagerImpl.java#L130`

Utilizzo di lambda per gestione eventi pulstante in MainMenuViewBuilder

Permalink: `https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a`
`src/main/java/ludomania/view/MainMenuViewBuilder.java#L90`

Progettazione con Generici per codice riutilizzabile in ComsmeticMenuViewBuilder

Permalink: `https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a`
`src/main/java/ludomania/view/CosmeticMenuViewBuilder.java#L101`

Utilizzo di reflection in CosmeticMenuViewBuilder

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a
src/main/java/ludomania/view/CosmeticMenuViewBuilder.java#L144

Utilizzo di JavaFX Properties in SettingManagerImpl

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a
src/main/java/ludomania/settings/impl/SettingsManagerImpl.java#L33

Utilizzo di JavaFX Properties in SettingController

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a
src/main/java/ludomania/settings/impl/SettingsController.java#L58

Utilizzo UI dinamica con binding in MainMenuViewBuilder

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a
src/main/java/ludomania/view/MainMenuViewBuilder.java#L154

Utilizzo di Apache Batik per gli svg

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a
src/main/java/ludomania/core/impl/ImageProviderImpl.java#L109

Utilizzo audio di JavaFX Media

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/e10c428b835e6ad1d004a
src/main/java/ludomania/core/impl/AudioManagerImpl.java#L50

3.2.2 Marco Pertegato

Progettazione con generici della classe CounterResult<T>

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/70b0690e11095c1b48296
src/main/java/ludomania/model/game/impl/CounterResult.java#L11

Utilizzo di lambda expression

Utilizzo di lambda in vari punti, ad esempio: https://github.com/Tsargov71/
OOP24-ludomania/blob/879aac9f5ae892cec41dbd0903c64edc1d236179/src/
main/java/ludomania/view/TrenteEtQuaranteViewBuilder.java#L207

Utilizzo di Stream

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/70b0690e11095c1b4829
src/main/java/ludomania/view/TrenteEtQuaranteViewBuilder.java#L192

Utilizzo di libreria esterna io.lyuda.jcards

Utilizzata in svariati punti ad esempio https://github.com/Tsargov71/
OOP24-ludomania/blob/879aac9f5ae892cec41dbd0903c64edc1d236179/src/
main/java/ludomania/model/croupier/impl/TrenteEtQuaranteDealer.java#
L152

Utilizzo audio di JavaFX

Utilizzata per la view in diversi punti ad esempio https://github.com/
Tsargov71/OOP24-ludomania/blob/879aac9f5ae892cec41dbd0903c64edc1d236179/
src/main/java/ludomania/view/TrenteEtQuaranteViewBuilder.java#L188

3.2.3 Carlo Michele Nicastro

UI dinamica con binding in BlackJackViewBuilder

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/879aac9f5ae892cec41d
src/main/java/ludomania/view/blackjack/BlackJackMenuViewBuilder.
java#L381

Utilizzo di java.util.UUID in BlackJackPlayer:

Permalink https://github.com/Tsargov71/OOP24-ludomania/blob/879aac9f5ae892cec41db
src/main/java/ludomania/model/player/impl/BlackJackPlayer.java#L19

Utilizzo della libreria esterna io.lyuda.jcards

Utilizzata in diverse classi come ad esempio: https://github.com/Tsargov71/
OOP24-ludomania/blob/879aac9f5ae892cec41dbd0903c64edc1d236179/src/
main/java/ludomania/model/game/impl/BlackJackGame.java#L241

Utilizzo di Stream

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/879aac9f5ae892cec41d
src/main/java/ludomania/model/croupier/impl/BlackJackDealer.java#
L45

3.2.4 Salvatore Zammataro

Utilizzo di classi anonime

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/879aac9f5ae892cec41d
src/main/java/ludomania/model/bet/RouletteBetType.java#L26

Utilizzo di Stream

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/879aac9f5ae892cec41d
src/main/java/ludomania/model/bet/RouletteBetFactory.java#L26

Utilizzo di JavaFX e FXML

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/6ba64c11c320452497524
src/main/resources/RouletteViewTemplate.fxml
Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/6ba64c11c320452497524
src/main/java/ludomania/model/game/roulette/RouletteSceneManager.
java#L89

Utilizzo di principi di programmazione funzionale

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/6ba64c11c320452497524
src/main/java/ludomania/model/bet/RouletteBetFactory.java#L28

Implementazione di classe Singleton

Permalink: https://github.com/Tsargov71/OOP24-ludomania/blob/6ba64c11c320452497524
src/main/java/ludomania/model/croupier/roulette/RouletteWheel.java

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Valerii Sargov

Sono particolarmente soddisfatto della facilità di creare le scene, che era il mio compito principale, e di come i manager siano stati integrati efficacemente. Ritengo che l'implementazione del pattern MVC (Model-View-Controller) abbia fornito una chiara separazione delle responsabilità. Inoltre, la gestione dei binding in JavaFX si è rivelata molto efficace nel sincronizzare i dati e aggiornare l'interfaccia in tempo reale. Sono anche contento di come ho concepito la logica dei cosmetici e l'uso delle preferences che ha funzionato sin da subito. Aree di Possibile Miglioramento Caricamento delle risorse: Attualmente, musica e immagini sono caricate tramite percorsi codificati manualmente nel codice. Questo approccio risulta rigido e poco manutenibile. Avrei preferito un sistema più flessibile, in grado di gestire dinamicamente le risorse, ad esempio tramite un file di configurazione.

Gestione di carte e fiches: La rappresentazione grafica degli elementi di gioco come carte e fiches poteva essere più modulare. Invece di disegnarle come semplici immagini (o quasi), sarebbe stato più efficace implementarle come componenti autonomi, dotati di logica e aspetto propri (come un JButton), così da facilitarne l'interazione e il riutilizzo.

Creazione e struttura dei manager: I manager sono attualmente del tutto indipendenti tra loro, ma condividono alcune funzionalità comuni (come init(), load(), save()). Queste potevano essere generalizzate tramite un'interfaccia comune, riducendo duplicazioni e migliorando l'organizzazione del codice. esempio di codice duplicato: `https://github.com/Tsargov71/OOP24-ludomania/blob/bf285c61d35926a4fbab6364dbbbd5a08225c429/src/main/java/ludomania/core/LudomaniaLauncher.java#L37`

Gestione dei binding: Ogni manager espone dati osservabili utilizzati per i binding tramite getter dedicati. Una soluzione più ordinata sarebbe stata introdurre una mappa del tipo <chiave, proprietà>per accedere dinamicamente alle proprietà, evitando di scrivere un metodo per ognuna di esse. Esempio di codice duplicato: `https://github.com/Tsargov71/OOP24-ludomania/blob/bf285c61d35926a4fbab6364dbbbd5a08225c429/src/main/java/ludomania/settings/impl/SettingsManagerImpl.java#L83`

Passaggio dei manager alle componenti: Al momento, il passaggio dei manager avviene in modo esplicito e manuale tra SceneManager, controller e viste. Una soluzione più scalabile sarebbe stata l'uso di una mappa <chiave, manager>o, ancora meglio, l'introduzione di un Facade centralizzato per fornire l'accesso ai vari servizi, semplificando la struttura e limitando la necessità di modifiche multiple in caso di aggiunta di nuovi manager.

4.1.2 Marco Pertegato

Sono soddisfatto del lavoro realizzato nonostante non siano state implementate tutte le funzionalità opzionali che avevamo pensato nella proposta del progetto, ritengo comunque che il lavoro sia complessivamente buono e le meccaniche principali dei vari giochi sono state implementate correttamente. Per quanto riguarda la mia parte all'interno del progetto ritengo di aver aiutato notevolmente con la progettazione e la modellazione del dominio e delle classi da utilizzare, sono molto soddisfatto di aver trovato una libreria esterna come jcards (`https://github.com/lyudaio/jcards`) che ci ha permesso di facilmente implementare classi che utilizzavano carte, mani, deck. Se dovessi criticare una parte del lavoro svolto, direi sicuramente che abbiamo avuto problemi per quanto riguarda la comunicazione e la stesura iniziale del dominio, ritengo che certe classi potessero essere modellate meglio, ma via del tempo e della scarsa interazione non è stato possibile. Pensando ai miglioramenti futuri sicuramente possiamo partire dall'implementare le funzionalità opzionali mancanti, come il multiplayer, che le mie classi sono già predisposte a supportare ma che non è stata sviluppata a causa del tempo. Per altri sviluppi futuri probabilmente sarebbero da ripensare meglio le interfacce comuni poichè quelle che avevamo inizialmente tirato giù poi si sono rivelate non ben definite per tutte le situazioni e tutti i casi, ciò a portato a delle classi un po' ripetute e mal progettate. Nonostante ciò sono contento di essere riuscito a collaborare con un gruppo molto diverso di altri studenti e di essere riuscito a sviluppare la mia parte fino ad ottenere un risultato soddisfacente.

4.1.3 Carlo Michele Nicastro

Lavorare in gruppo ha rappresentato una sfida per me in quanto mi sono reso conto che la comunicazione e il coordinamento sono aspetti fondamentali, spesso sottovalutati all'inizio. Molte delle difficoltà incontrate non sono derivate dalla complessità tecnica, ma piuttosto da interpretazioni diverse degli stessi requisiti o dalla mancanza di allineamento iniziale. Ritengo che il lavoro svolto sia stato complessivamente positivo, pur riconoscendo che ci siano ancora margini di miglioramento, sia nella scrittura del codice sia nelle scelte architetturali. Il progetto non è stato solo un esercizio di programmazione, ma anche un'importante esperienza formativa, che mi ha aiutato a crescere come sviluppatore ed essere membro di un team. L'aspetto più prezioso è stato forse proprio questo: imparare a confrontarsi con gli altri, accettare compromessi, rivedere le proprie idee e collaborare verso un obiettivo comune. Il codice si può sempre migliorare, ma le competenze relazionali e la capacità di adattarsi a un contesto di gruppo sono conquiste che restano nel tempo.

Ci sono delle funzionalità opzionali che abbiamo scritto nella richiesta di progetto che poi non siamo riusciti ad implementare per questioni di tempo, nel mio caso una futura possibile implementazione è quella della possibilità di split della puntata base quando escono due carte uguali come prima mano del giocatore, la possibilità di raddoppio con l'estrazione di una sola carta, le sides bet (come ad esempio colore o scala) e l'assicurazione, che permette al giocatore di non perdere la propria puntata quando il mazziere ha come carta scoperta un asso. Inoltre non vi è la gestione del multiplayer in locale, funzione che non abbiamo implementato ma che è possibile farlo grazie al codice facilmente estensibile ed adattabile.

In conclusione sono molto soddisfatto di questo progetto, mi è piaciuto molto sviluppare un gioco che mi piace e farlo con un gruppo mi ha aperto la possibilità di conoscere nuove persone ed interessi. Molto importante anche per la gestione del tempo ed il coordinamento con il resto del gruppo.

4.1.4 Salvatore Zammataro

Ritengo che il lavoro svolto sia accettabile e per certi aspetti anche buono. Considero, tuttavia che ci sia ancora molto da migliorare sia per quanto riguarda la banale scrittura del codice, sia per le scelte progettuali fatte. Lo svolgimento del progetto all'interno di un gruppo ha messo in luce l'importanza di una buona comunicazione e coordinamento con gli altri membri e soprattutto di una approfondita analisi del problema e delle soluzioni che vi si vogliono apportare, prima della messa in opera delle stesse.

Per quanto riguarda la parte da me svolta ho trovato qualche difficoltà nel riutilizzo di codice sviluppato dai compagni, poiché non sempre rispettava le mie necessità e questo ha probabilmente portato a soluzioni subottimali, di cui mi attribuisco comunque tutta la responsabilità. Altri aspetti dello sviluppo sono invece stati notevolmente facilitati dall'impostazione data dai colleghi, nonché dal loro pronto supporto ogni qualvolta vi fosse bisogno di un chiarimento o consiglio.

In conclusione posso affermare che l'utilità del progetto non si limita al mero sviluppo dello stesso e del conseguente studio effettuato, ma anche nelle capacità di relazione con colleghi, la cui idea non sempre combacia con la propria.

Appendice A

Guida utente

A.1 Menù Principale

L'utente potrà selezionare il gioco che desidera avviare, il quale verrà evidenziato con un bordo rosso. Una volta effettuata la selezione, sarà possibile premere il pulsante "Avvio/Start" per iniziare la partita. Sono inoltre presenti i pulsanti "Settings", che apre il menù delle opzioni, "Exit", che chiude l'applicazione, e un'icona dedicata per accedere al menù dei cosmetici.

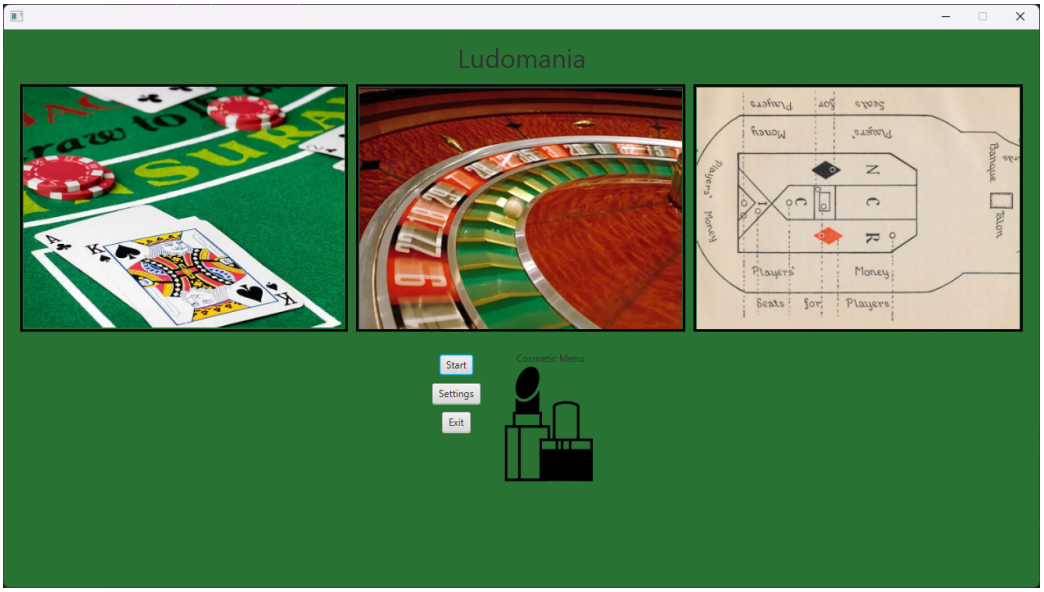


Figura A.1: Shermata del Menù Principale

A.2 Menù Impostazioni

In questa schermata l'utente può selezionare la lingua dell'applicazione e modificare diverse impostazioni: il volume audio, la modalità a schermo intero e la risoluzione dello schermo. Sono presenti tre pulsanti: uno per applicare e salvare le modifiche nelle preferenze, uno per ripristinare i valori di default e uno per tornare al menù principale.

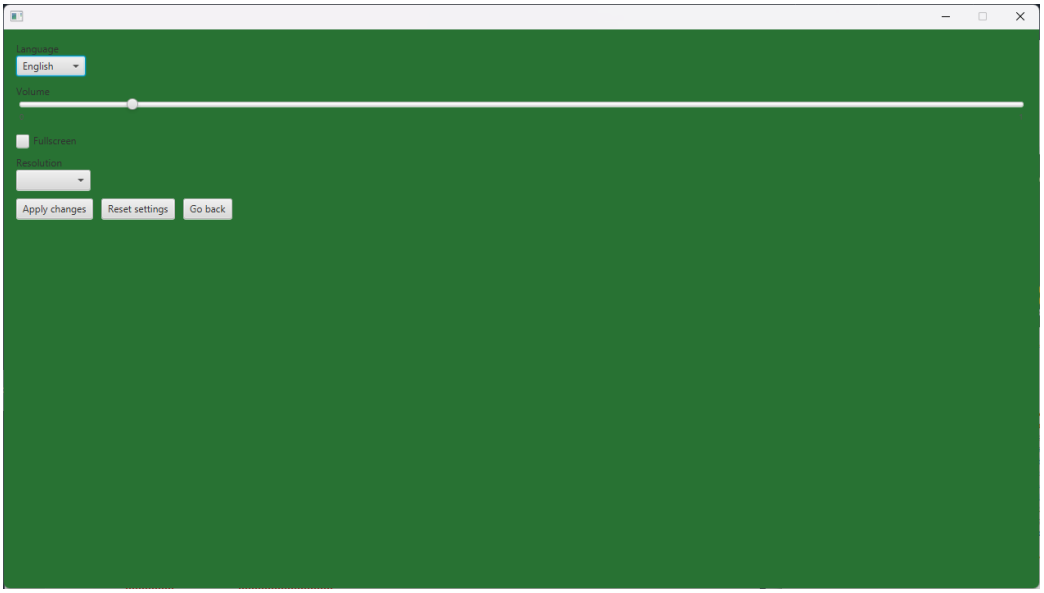


Figura A.2: Shermata delle Impostazioni

A.3 Menù Cosmetici

In questo menù l'utente può visualizzare e selezionare uno dei tre temi grafici disponibili (europeo, americano e neon), che influenzano lo sfondo, le fiches e le carte da gioco. Il tema scelto viene applicato all'intera applicazione e salvato per essere mantenuto anche nei successivi avvii. È inoltre presente un pulsante per tornare al menù principale.

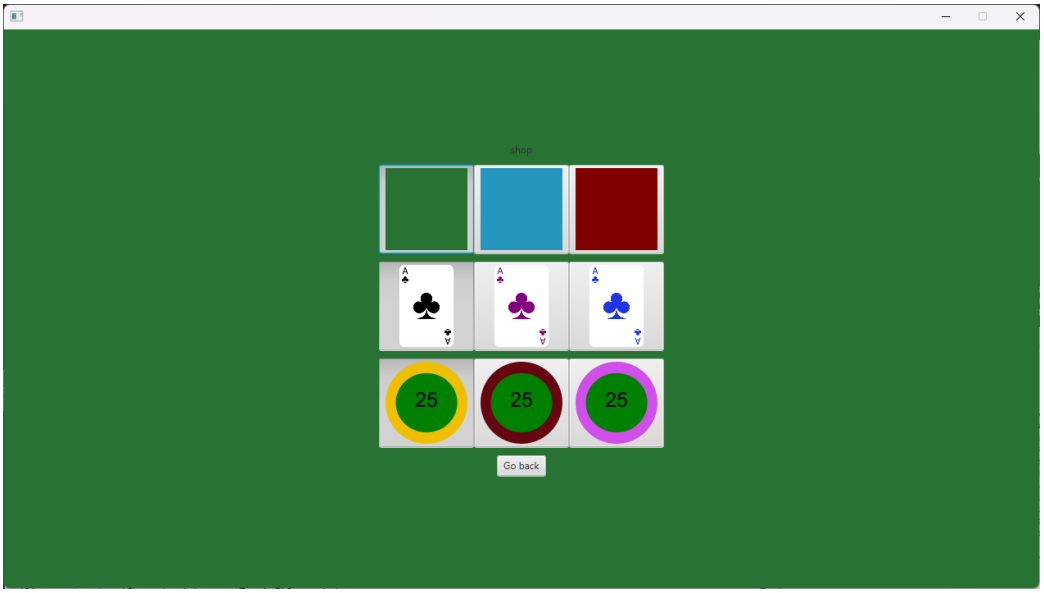


Figura A.3: Shermata del Menù dei cosmetici

A.4 BlackJack

Una volta selezionato il gioco del BlackJack dal menù principale, si aprirà la schermata di gioco, con in alto a destra il pulsante "Regole" che apre una finestra con le regole del gioco del BlackJack, a seguire un pulsante "Esci" per uscire dal gioco e tornare al menù principale. Per iniziare una partita basta selezionare una o più fiches (il valore della bet viene mostrato a destra di esse) e premere sul pulsante "avvia"; se invece si vuole annullare la bet corrente basta premere il pulsante cancella. Una volta iniziata la partita vengono mostrate sul tavolo di gioco le carte del Mazziere e del Giocatore con accanto il numero che indica il totale che essi possiedono. All'utente sarà possibile chiedere carta o stare, successivamente viene fatto il gioco del mazziere e in alto uscirà l'esito della partita appena giocata. Al termine di ogni partita sarà possibile farne un'altra facendo una nuova puntata oppure uscire. Inoltre in basso a destra viene visualizzato il nome del giocatore e il suo saldo, il quale si aggiorna dopo ogni partita.

Figura A.4: Shermata del gioco BlackJack

A.5 Roulette

Una volta selezionato il gioco dal menù principale e dopo aver cliccato sul bottone "Avvia" si aprirà la schermata di gioco. Di fronte a se si trova la plancia su cui è possibile piazzare le scommesse, attraverso il click del mouse. Le caselle che si stanno per selezionare verranno evidenziate in blu per facilitarne la visione, altrimenti delegata al solo cambiamente della forma del cursore. In basso sono visibili le fiche da selezionare per poter piazzare una scommessa, in verde viene evidenziato l'ammontare della prossima scommessa da piazzare ed in rosso l'ammontare di denaro ancora disponibile nel portafoglio del giocatore. Nell'angolo in basso a sinistra sono presenti due bottoni, uno apre un riepilogo di tutte le scommesse piazzate sino a quel momento, l'altro le regole del gioco. Una volta piazzate tutte le scommesse volute è possibile cliccare sull'immagine della roulette a sinistra per estrarre un numero, il quale verrà visualizzato al di sotto di essa. Per avviare poi il calcolo delle vincite si deve cliccare sul bottone "Ok".

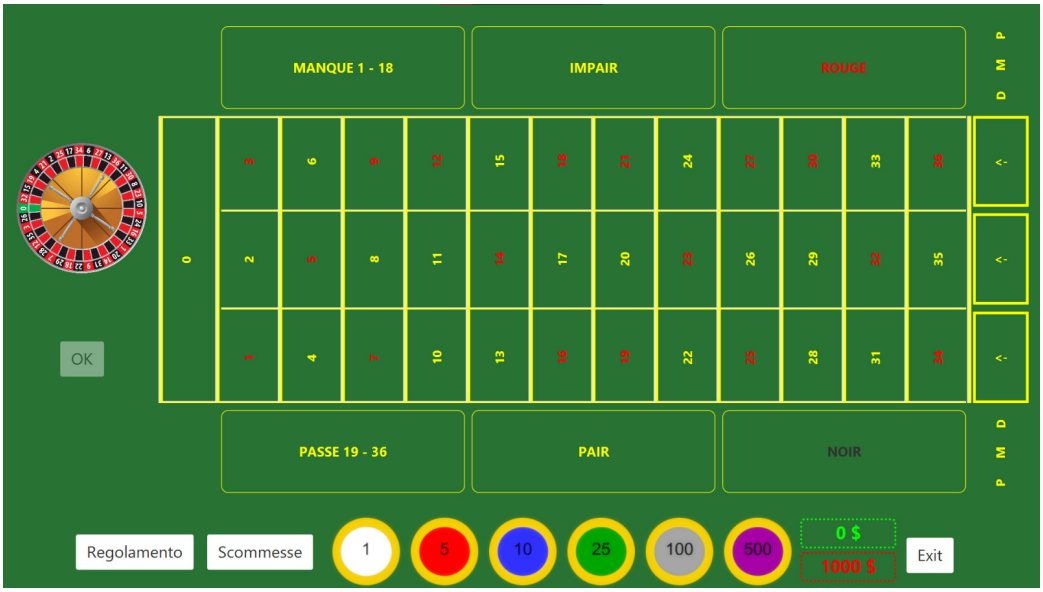


Figura A.5: Shermata del gioco Roulette

A.6 TrenteEtQuarante

Una volta avviato il gioco si aprirà la finestra del TrenteEtQuarante in cui l'utente vedrà in alto a partire da sinistra, il suo nome, il titolo, e a destra due pulsanti uno per visualizzare le regole del gioco e uno per tornare al menù principale. In basso partendo da sinistra si avranno le varie fiches che si potranno selezionare solo una alla volta, in caso si clicchi su una fiche già selezionata essa verrà deselezionata, poi a destra viene mostrato il bilancio del giocatore. Al centro invece abbiamo:

- Un log che mostra il turno, le bet piazzate dal giocatore e a fine round l'esito
- Due file una per Noir e una per Rouge che vengono riempite di carte di cui a lato viene indicato il valore totale
- Quattro riquadri selezionabili per piazzare le bet, essi verranno disabilitati se non è selezionata alcuna fiche o le viene selezionata una fiche dal valore più alto di quello presente nel portafoglio del player
- un pulsante per confermare le bet piazzate ed avviare il round

Una volta terminati i soldi non si potrà più giocare, l'unica opzione per giocare di nuovo è tornare alla schermata iniziale e avviare una nuova sessione.

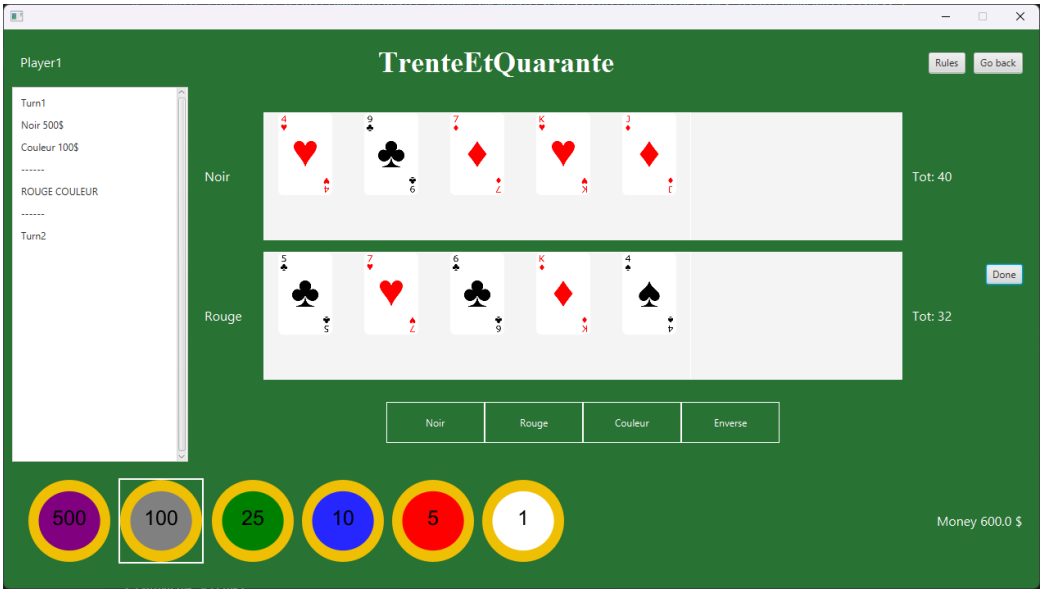


Figura A.6: Shermata del gioco TrenteEtQuarante