Cavina Mattia

Grandini Matteo

Mariani Matilde

Piras Ilaria

14 Febbraio 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

L'obiettivo del progetto consiste nel realizzare una versione semplificata del gioco di carte Poker Texas Hold'em. L'utente si sfida con tre giocatori virtuali al fine di rimanere l'ultimo ad avere dei gettoni in suo possesso. La partita si articola in più mani, all'inizio delle quali vengono assegnate due carte ad ogni giocatore e, a rotazione, i ruoli Small blind e Big blind, che determinano un obbligo di puntata iniziale.

Ogni mano si sviluppa in quattro fasi (Preflop, Flop, Turn e River) nelle quali viene aggiunto un determinato numero di carte scoperte al tavolo e i giocatori, partendo dallo Small blind e procedendo in senso orario, scelgono se e quanto puntare in base alla migliore combinazione che sono riusciti a formare, utilizzando le proprie carte e quelle sul tavolo. Le combinazioni possibili, in ordine crescente di valore, sono:

1. High Card: la carta di valore più alto
2. Pair: due carte dello stesso valore
3. Two Pair: due Pair
4. Three of a Kind: tre carte dello stesso valore
5. Straight: cinque carte di valore consecutivo
6. Full-House: un Tris e un Pair
7. Flush: cinque carte dello stesso seme
8. Four of a Kind: quattro carte dello stesso valore
9. Straight Flush: una Straight con tutte le carte dello stesso seme
10. Royal Flush: una Straight Flush che va dall'asso al dieci (A, K, Q, J, 10)

Vince la mano il giocatore con la combinazione migliore.

Requisiti funzionali

- Il giocatore deve poter scegliere se puntare (e quanto) oppure se abbandonare la mano.
- Raccogliere e mantenere statistiche di gioco, come il numero partite vinte, vincita massima di gettoni, migliore combinazione, ecc…
- Possibilità di scegliere la difficoltà e la quantità di gettoni da distribuire a inizio partita.

- Possibilità di mettere in pausa la partita e riprendere dal punto in cui la si era interrotta.
- I giocatori avversari devono simulare un comportamento di gioco umano.
- Distribuzione casuale delle carte.

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.

1.2 Modello del dominio

Una partita è composta da una sequenza di mani e dispone di una lista di giocatori e di un mazziere. Una mano è, a sua volta, composta da una serie di fasi durante le quali viene richiesta ai giocatori, che possono essere sia di tipo IA che utente, la loro azione e, eventualmente, la loro puntata. La decisione di ogni giocatore si basa sulla miglior combinazione di carte a loro disposizione tra quelle in suo possesso e quelle comuni a tutti. Le carte vengono gestite dal mazziere che, avendo a disposizione un mazzo, deve essere in grado di mescolarle e distribuirle ai giocatori e alla partita. Al termine di ogni mano viene decretato il vincitore al quale verranno assegnate le vincite, ottenute dall'insieme di puntate effettuate dai giocatori durante la mano stessa. Infine, la partita, data la lista di giocatori, è in grado di stabilire quando è conclusa e di determinare se il giocatore umano ha vinto o perso.

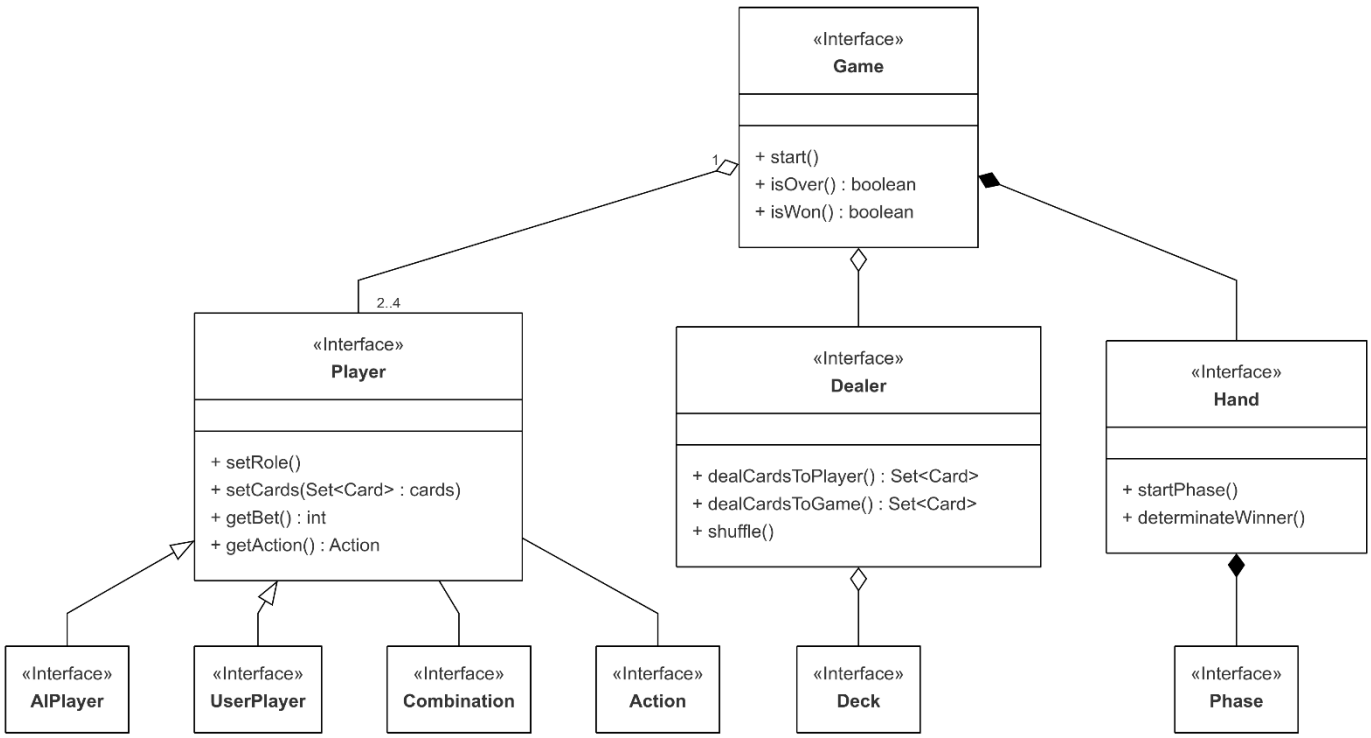

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2

Design

2.1 Architettura

L'architettura dell'applicazione segue il pattern architetturale MVC nella sua versione standard. Il *Model* offre come proprio entry point l'interfaccia ***Game***; infatti, al suo interno sono gestiti tutti gli elementi per la progressione di una partita, incluso il susseguirsi delle mani, l'interazione fra i giocatori e la distribuzione delle carte tramite il mazziere. Il *Controller*, invece, ha come proprio punto d'ingresso l'interfaccia ***GameController*** che funge da intermediario tra la View e il Model per gestire le azioni ricevute dall'utente e visualizzare lo stato aggiornato del gioco. Infine, la *View* offre come proprio entry point l'interfaccia ***View***, la quale, assieme ad un ***SceneController*** e ad un'interfaccia ***Scene***, mette a disposizione sistemi per spostarsi tra le schermate dell'applicazione. La View è inoltre responsabile di ricevere gli input dell' utente e di informare il Controller, in quale, a sua volta, comunicherà i cambiamenti al Model che si aggiornerà di conseguenza. Nella realizzazione di questa architettura ci si è assicurati che qualunque cambiamento effettuato nella View, anche la sua completa sostituzione, non impatti in alcun modo sul Model. Inoltre, se dopo che sono state effettuate le modifiche nella View, le classi continuano a rispettare le interfacce fornite, neanche il Controller sarà impattato e non sarà necessario modificarlo, se non in maniera minimale.

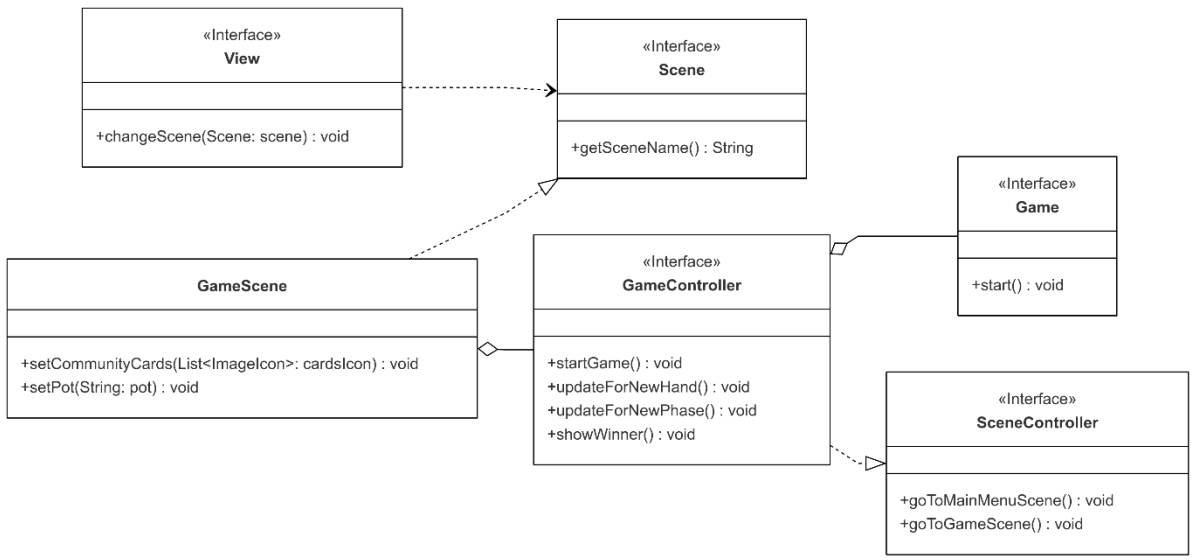

Figura 2.1: Rappresentazione UML dell'architettura MVC realizzata

2.2 Design dettagliato

2.2.1 Cavina Mattia

Implementazione Deck

Problema: Rendere la classe ***Deck*** riutilizzabile per gestire diverse tipologie di gioco con carte diverse.

Soluzione: Per ottenere una riusabilità del ***Deck*** per eventualmente sfruttarlo per ulteriori varianti ho utilizzato il pattern *Strategy* tramite l'interfaccia funzionale ***DeckBuild*** .
L'interfaccia ***Deck*** e la sua concretizzazione ***DeckImpl*** sono generiche, al costruttore di quest'ultima viene passata l'interfaccia funzionale ***DeckBuild*** che racchiudendo il metodo per la creazione del ***Deck*** permettendo di definire l'oggetto generico in questo caso ***Card***. La costruzione è delegata al ***DeckFactory*** sfruttando *Factory Method* per unire l'implementazione di ***Deck*** alla concretizzazione di ***DeckBuild***.

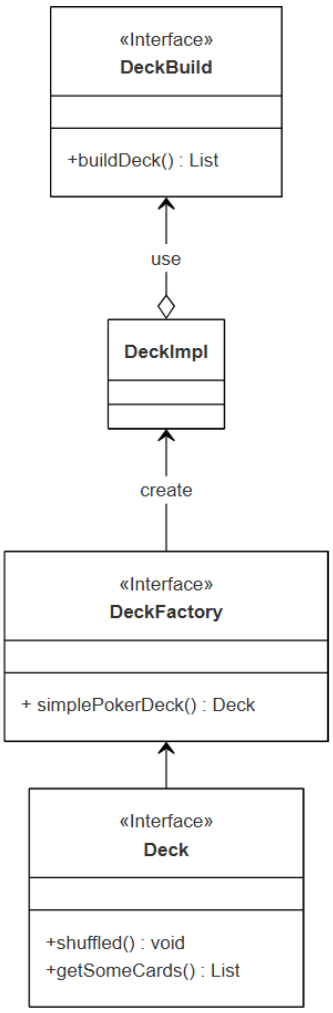

Figura 2.2: Rappresentazione UML delle classi coinvolte

Gestione delle Combinazioni

Problema: Avere una classe di utilità comune per più classi con scopi differenti, ma simile implementazione dei metodi.

Soluzione: Nella creazione delle classi ***CombinationCardGetters*** e ***CombinationRules*** mi si è posto il problema di avere dei metodi utili allo scopo di entrambe le classi, sia per recuperare le carte che compongono una combinazione sia per l'individuazione delle stesse. Nel UML si possono notare i metodi **getPair()** e **isPair()** che all'interno usano dei metodi comuni per arrivare a risultati differenti Boolean e Set. Per risolvere questo problema ho creato una interfaccia ***CombinationUtilities*** che passandola come argomento ad entrambe le utilizzatrici restituisce i metodi necessari ad entrambe come **getHighterStraight()** per controllare l'esistenza di uno Straight. Ho scelto di passarle come argomento per dare spazio ad implementazioni delle Utilità diverse da quella Creata nell'applicazione dato che nel corso della progettazione mi è capitato di cambiarla più volte e quindi di averne di diversa natura dato lo stesso risultato.

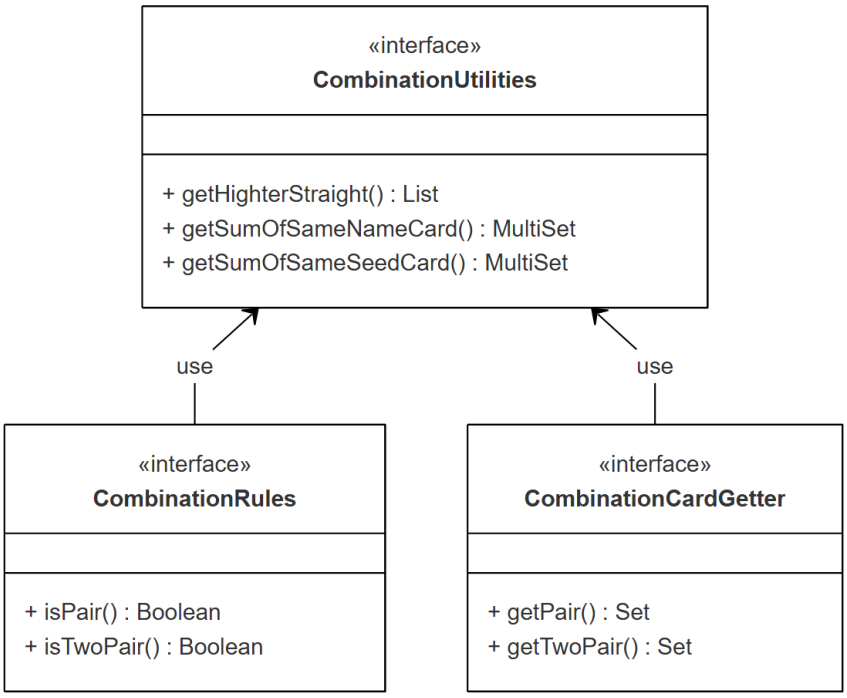

Figura 2.3: Rappresentazione UML delle classi coinvolte.

Problema: Creare la classe ***Combination*** adattata per ogni tipologia di Combinazione in modo da passarla con tutti i dati necessari.

Soluzione: Il primo problema riscontrato è stato l'individuazione della combinazione dato un *Set* di carte, l'ho risolto mediante la classe ***CombinationHandler*** che interroga il mio set tramite il metodo **getBestCombination()** per ogni combinazione in ordine decrescente di importanza fermandosi qual ora si abbia individuato la combinazione contenuta. Una volta individuata ho delegato alla classe **CombinationFactory**, utilizzando il pattern *Factory Method* , la creazione delle varie combinazioni avendo esse dei valori fissi come la tipologia. Viene quindi richiamato il metodo corrispondente come **getPair()** per creare e restituire la combinazione individuata. Questa soluzione riduce al minimo le dipendenze con altre parti del programma.

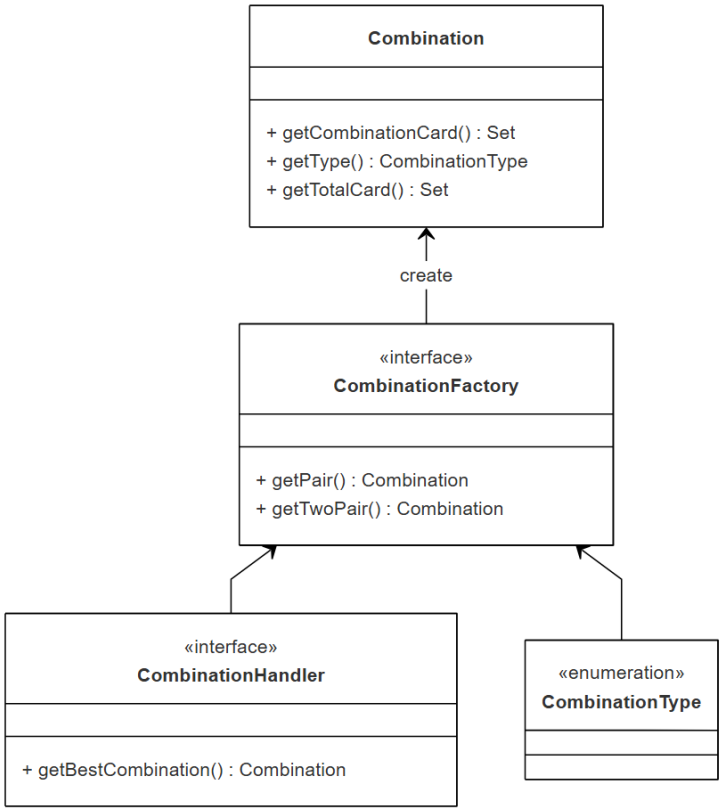

Figura 2.4: Rappresentazione UML della creazione della classe Combination mediante la propria CombinationFactory.

Conversione della Classe Card in ImageIcon

Problema: Convertire per la parte grafica le classi ***Card*** contenute nei Set dei giocatori e del tavolo in immagini utili alla GUI.

Soluzione: Per ottenere una conversione il più snella possibile ho chiamato le carte con nomi che richiamassero le variabili contenute nel record ***Card*** (in particolare ***seedName*** e ***valueOfCard*** ovvero Seme e Valore). Tramite uno stream ho convertito così agilmente l'oggetto Card in ImageIcon, mappando i dati della prima in uno String che richiamasse il nome corretto della carte e tramite il Classloader ho recuperato dalla risorse il .jpg. Ho infine restituito la nuova lista a chi dovrà sfruttarla per la GUI.

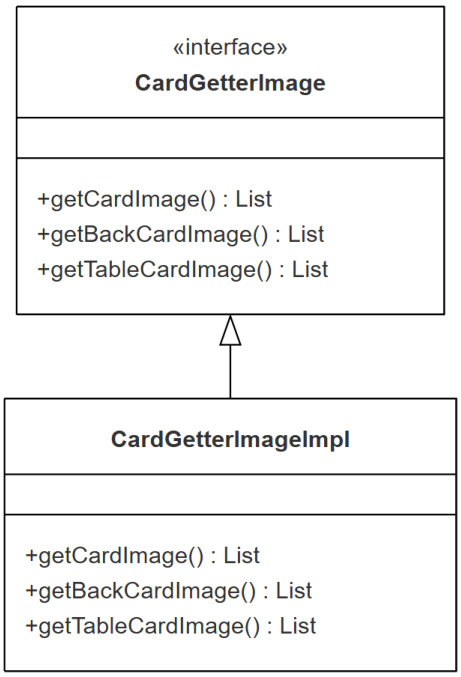

Figura 2.5: Rappresentazione UML dell' implementazione di CardGetterImage .

2.2.2 Grandini Matteo

Creazione di giocatori IA di difficoltà diverse

Problema: Deve essere possibile creare facilmente diverse difficoltà di giocatori IA e il loro utilizzo deve essere il medesimo.

Soluzione: Il sistema per la creazione dei giocatori IA utilizza il pattern *Factory Method*, come da Figura 2.6: l'interfaccia ***AIPlayerFactory*** definisce i metodi per la creazione trasparente di diversi oggetti ***AIPlayer***. Per creare giocatori con difficoltà diverse, la classe ***AIPlayerFactoryImpl*** concretizza la classe ***AbstractAIPlayer*** definendo logiche distinte per quando ciascun giocatore dovrebbe effettuare una *Call* o una *Raise*.

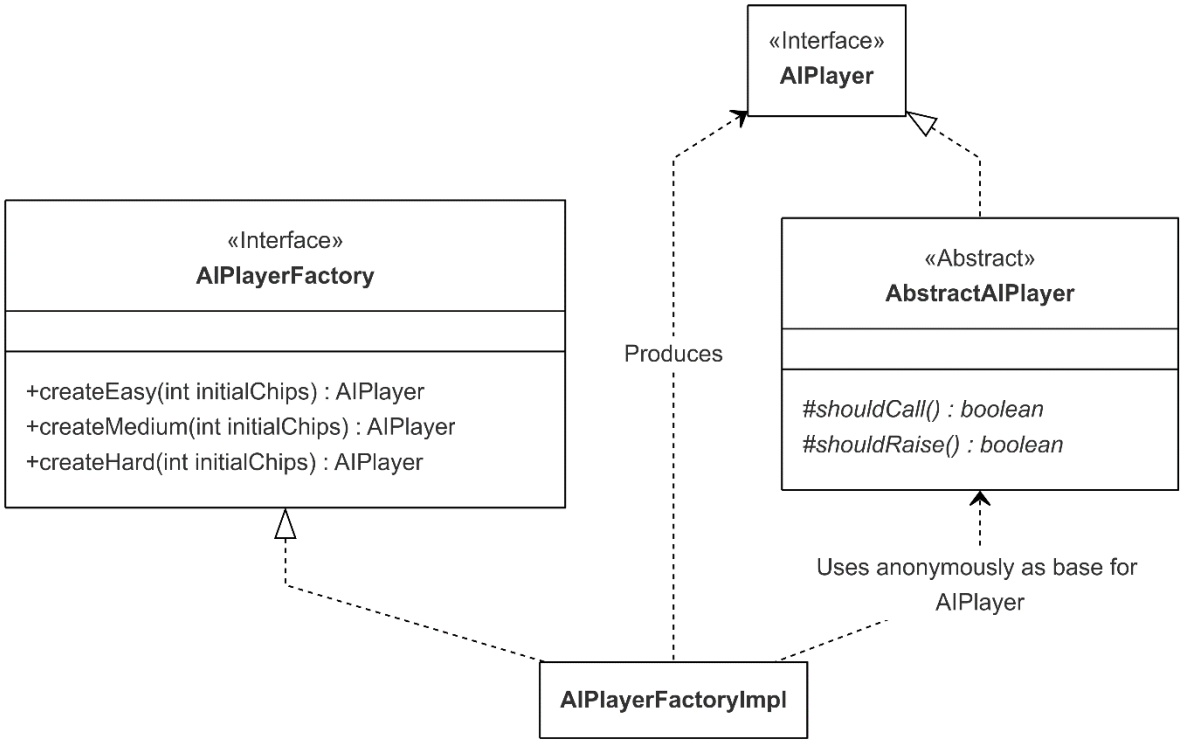

Figura 2.6: Rappresentazione UML del pattern *Factory Method* per la creazione di oggetti AIPlayer di difficoltà diversa.

Gestione giocatori IA di difficoltà diverse

Problema: Il gioco deve poter utilizzare giocatori umani e giocatori IA in maniera trasparente; inoltre, ci si è accorti che i giocatori IA delle varie difficoltà condividevano grande parte del codice, portando a classi molto simili con codice ripetuto.

Soluzione: Siccome i giocatori di difficoltà diverse differiscono solo nel comportamento da effettuarsi quando viene richiesta l'azione da compiere, si è deciso di utilizzare il pattern *Template Method*, come da Figura 2.7. Il metodo template in questione è **getAction()** della classe ***AbstractAIPlayer***, che richiama i metodi astratti **shouldCall()** e **shouldRaise()**.
Inoltre, per garantire il medesimo utilizzo dei giocatori IA e di quelli umani mantenendo le ripetizioni al minimo, ***AbstractAIPlayer*** estende ***AbstractPlayer*** e implementa l'interfaccia ***AIPlayer***, la quale a sua volta estende l'interfaccia ***Player***.

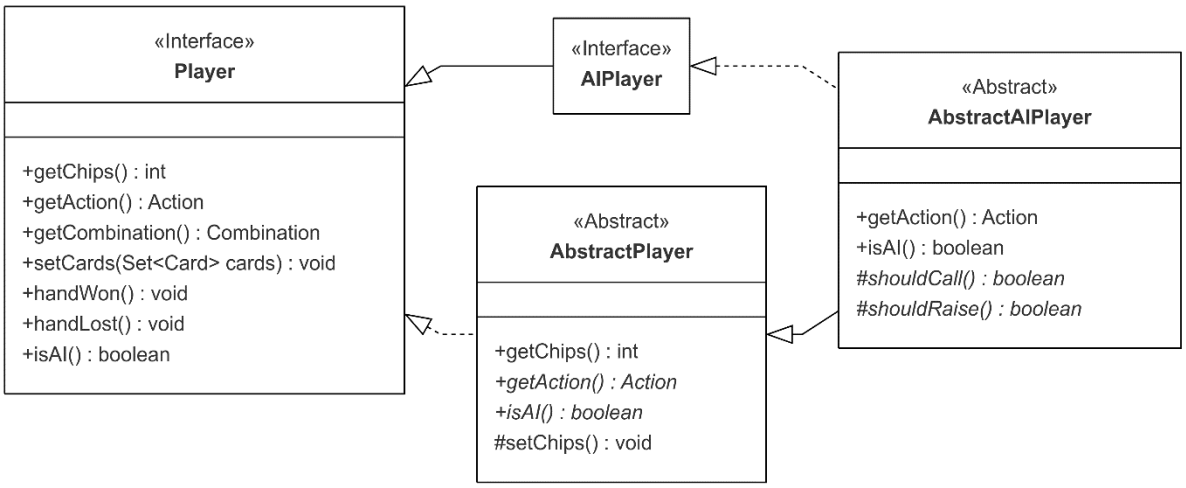

Figura 2.7: Rappresentazione UML dell'utilizzo del pattern *Template Method* in AbstractAIPlayer e della compatibilità degli AIPlayer con Player.

Raccolta statistiche

Problema: L'applicazione deve poter raccogliere delle statistiche di vario genere sul gioco. Queste statistiche potrebbero provenire da classi diverse ed è necessario che vengano salvate su file per renderle persistenti.

Soluzione: Utilizzo di un ***StatisticsManager*** generico (in caso si vogliano gestire categorie di statistiche diverse), in grado di raccogliere le statistiche complessive provenienti da tutte le classi che implementano l'interfaccia ***StatisticsContributor*** e che sono state aggiunte al manager stesso, come da Figura 2.8. Le statistiche utilizzate nell'applicazione sono definite dall'interfaccia ***BasicStatistics***, che permette di memorizzare delle statistiche di base. Il ***StatiaticsManager***, inoltre, è in grado di salvare e caricare le statistiche totali su/da file grazie alla serializzazione.

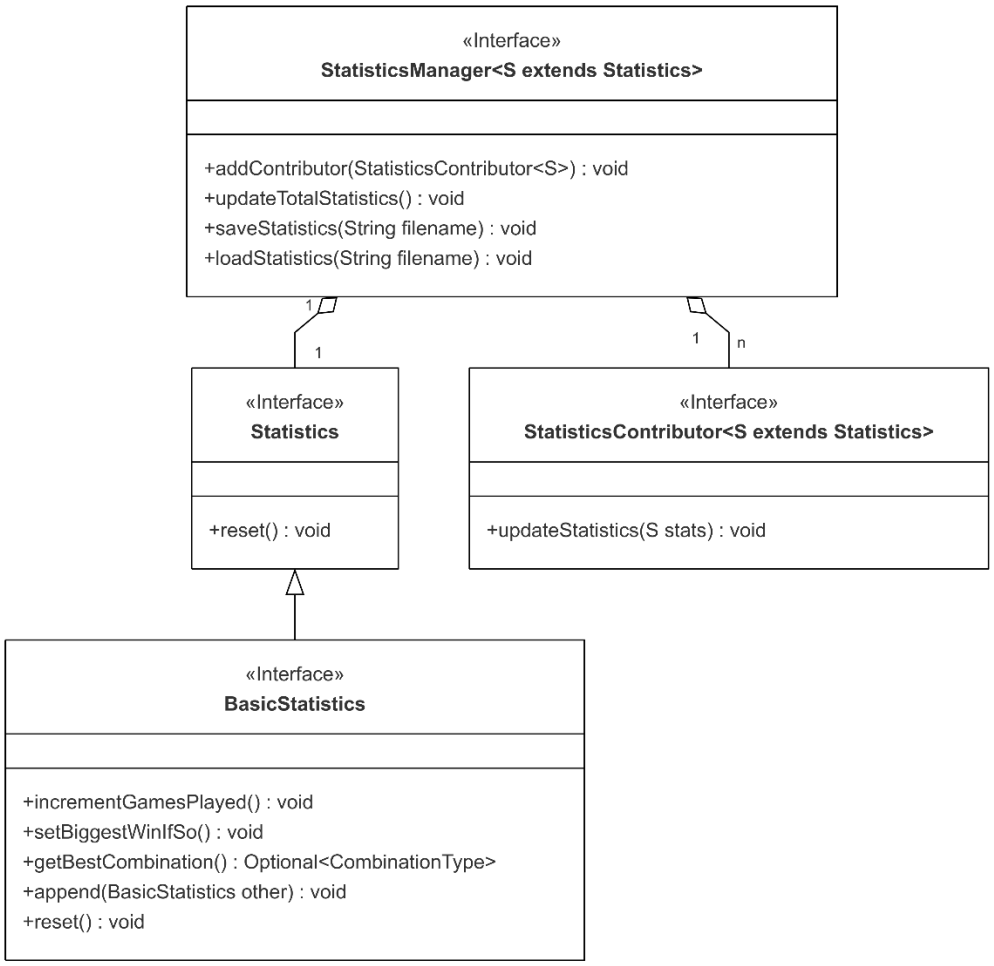

Figura 2.8: Rappresentazione UML del sistema di raccolta, salvataggio e caricamento delle statistiche di gioco.

Visualizzazione delle statistiche

Problema: Deve essere possibile accedere ad una schermata in cui sono visualizzate le statistiche del gioco. Queste statistiche devono essere reperibili all'avvio dell'applicazione, prima ancora di aver avviato una partita, e deve essere possibile resettare queste statistiche tramite un'interfaccia grafica.

Soluzione: Per fare sì che la classe ***StatisticsScene***, rappresentante la schermata di visualizzazione delle statistiche, possa essere mostrata e scambiata con le altre schermate dell'applicazione, si è pensato di farla aderire all'interfaccia ***Scene***. Quest'ultima è implementata da tutte le schermate del sistema e ha lo scopo di rendere più agevole il passaggio fra scene diverse. Inoltre, per creare una separazione fra la *View* e il *Model*, si è utilizzato un ***StatisticsController*** in grado di accedere alle statistiche, caricate da un

StatisticsManager, e fornirne una rappresentazione comprensibile alla *View*, oltreché alla possibilità di resettarle, come da Figura 2.9.

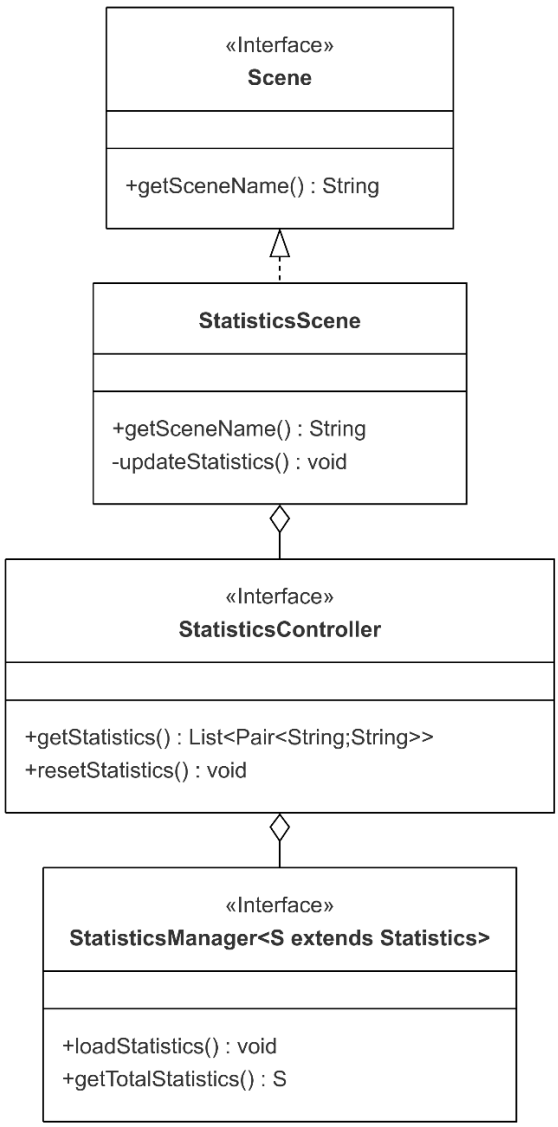

Figura 2.9: Rappresentazione UML delle classi e interfacce coinvolte nella visualizzazione delle statistiche di gioco.

2.2.3 Mariani Matilde

Gestione dell'interazione utente

Problema: La principale difficoltà consiste nel gestire l'interazione dell'utente con il gioco, separando le preoccupazioni tra logica di gioco e interfaccia utente.

Soluzione: L'adozione del modello MVC ha permesso di risolvere questi problemi. La View è gestita dalla classe ***UserPanel,*** che permette al giocatore di interagire con l'interfaccia grafica. Il model è rappresentato da ***UserPlayerImpl***, che si occupa della logica di gioco, gestendo le azioni del giocatore tramite il metodo **getAction()**, calcolando l'importo delle chips da scommettere con **calculateChipsToBet()** e aggiornando le caratteristiche del giocatore. Infine, il controller, rappresentato da ***UserPlayerController***, riceve le azioni dell'utente dalla View tramite il metodo **receiveUserAction()**, le elabora e aggiorna il model con il metodo **getUserAction()**. La comunicazione tra i componenti è disaccoppiata, garantendo così flessibilità.

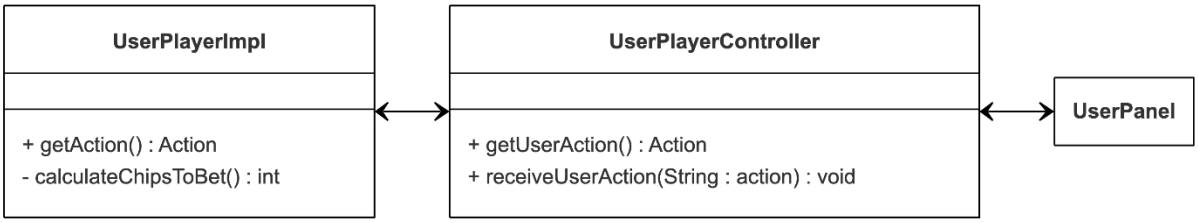

Figura 2.10: Rappresentazione UML delle classi coinvolte nella gestione dell'interazione utente.

Gestione delle azioni dell'utente

Problema: Il giocatore per poter agire durante la partita deve premere i tasti a sua disposizione (Check, Call, Raise, Fold, All-in) o inserire la sua puntata in caso di Raise. Gli input vengono catturati dal ***UserPlayerController***.

Soluzione: Quando è il turno dell'utente, vengono abilitati i pulsanti per le azioni che può compiere utilizzando **updateButtonStates()**. La gestione degli input avviene attraverso un listener comune per tutti i pulsanti, che viene associato a ciascun pulsante durante la creazione del pannello nel metodo **createUserPanel()**. Questo listener è un oggetto della classe ***InputActionListener***, una classe privata che implementa ***ActionListener.*** Al momento della pressione di un pulsante, viene chiamato il metodo **receiveUserAction()**, che aggiorna il valore di action all'interno di ***UserPlayerController***. Se l'azione scelta è "Raise", viene verificato l'importo della puntata tramite il metodo **isAmountOK()** del ***UserPlayerController*** per assicurarsi

che sia valido. Infine, una volta ricevuta l'azione, tutti i pulsanti vengono disabilitati chiamando **disableAllButtons()**.

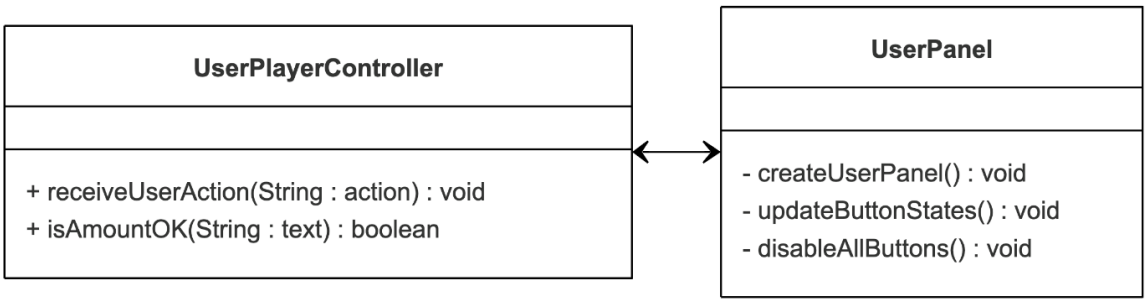

Figura 2.11: Rappresentazione UML delle classi coinvolte nella gestione delle azioni dell'utente.

Sincronizzazione dell'azione utente

Problema: Nel gioco, il metodo **getAction()** richiama a sua volta **getUserAction()**, che restituisce l'azione dell'utente. Tuttavia, se l'utente non ha ancora premuto nulla, il valore di action risulta vuoto, causando problemi nel flusso del gioco.

Soluzione: Per risolvere il problema, il metodo **receiveUserAction()** è stato sincronizzato, garantendo che solo un thread alla volta possa accedere al valore di action. All'interno di questo metodo, è stato utilizzato **notifyAll(),** mentre nel metodo **getUserAction()**, che attende l'input dell'utente, è stato impiegato **wait().** Entrambi i metodi utilizzano un oggetto di blocco (lock) per sincronizzare l'accesso alla variabile action. Solo quando l'azione è stata ricevuta, **getUserAction()** può restituire correttamente il valore di action, permettendo al gioco di proseguire senza interruzioni. L'uso di **notifyAll()** e **wait()** è preferibile rispetto a **Thread.sleep()**, poiché consente una gestione più precisa e reattiva del flusso del programma. Con **wait()**, il thread resta in attesa fino a quando non riceve la notifica che il valore di action è stato aggiornato, evitando pause inutili e migliorando l'efficienza. Al contrario, **Thread.sleep()** introduce una pausa fissa che non dipende dallo stato del programma, risultando meno dinamica e potenzialmente meno efficiente.

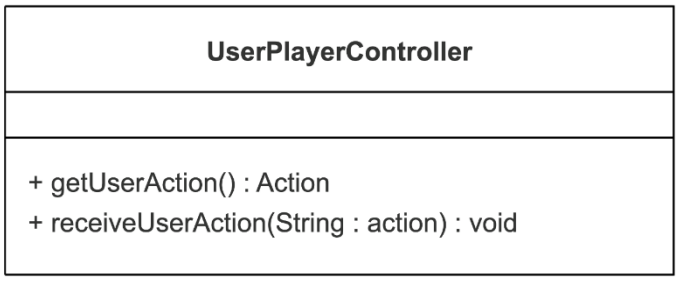

Figura 2.12: Rappresentazione UML della classe coinvolta nella sincronizzazione dell'azione utente.

Garantire Dati Corretti e Sicuri

Problema: L'utente potrebbe inserire valori non validi (come caratteri non numerici o numeri fuori intervallo) nel campo di testo per il numero di chip, causando malfunzionamenti o comportamenti imprevisti nell'applicazione.

Soluzione: La validazione dell'input è stata implementata con un controllo in tempo reale nella ***DifficultySelectionScene*** per garantire che i caratteri siano numerici e che il valore rientri nell'intervallo di chips consentito (tra 1.000 e 1.000.000). Questo processo utilizza un ***KeyListener*** per verificare la validità dei caratteri digitati e un ***ActionListener*** per gestire la logica al momento dell'invio del valore. In caso di errore, viene visualizzato un messaggio di errore, che viene inizializzato durante la creazione del pannello tramite il metodo **initialize()**. Una volta abilitato, premendo "Play", la difficoltà selezionata e il numero iniziale di chips vengono passati al gioco, permettendo di avviare la partita con le impostazioni corrette.

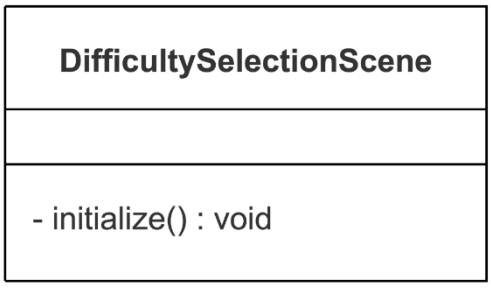

Figura 2.13: Rappresentazione UML della classe coinvolta nella validazione dei dati inseriti dall'utente.

2.2.4 Piras Ilaria

Creazione gioco

Problema: All'avvio di una partita l'utente sceglie la difficoltà e il valore iniziale di chips con il quale iniziare. Il gioco deve essere creato in funzione di queste scelte.

Soluzione: Per la creazione del gioco è stato utilizzato il pattern *Factory Method,* come in figura 2.14: l'interfaccia **GameFactory** definisce i factory methods, uno per difficoltà, mentre la classe **GameFactoryImpl** implementa questi metodi e crea un gioco concretizzando la classe **AbstractGame**.

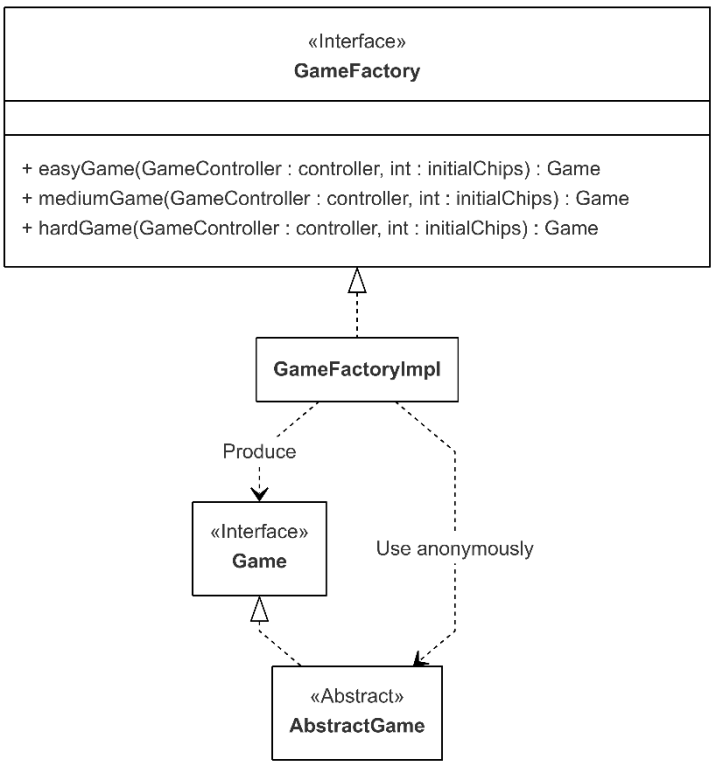

Figura 2.14: Rappresentazione UML del pattern Factory Method per la creazione di un Game di una specifica difficoltà.

Inizializzazione giocatori

Problema: Una partita ha una lista di giocatori. Al momento della creazione essa è formata dal giocatore utente e da tre giocatori IA. Quest'ultimi in particolare possono essere di diverso tipo a seconda della difficoltà scelta.

Soluzione: Si è notato che le diverse tipologie di partita differiscono solo nell'inizializzazione dei giocatori IA, è stato quindi utilizzato il pattern *Template Method* per massimizzare il riuso come da figura 2.15: il metodo template è **setInitialPlayers()** che chiama il metodo astratto e protetto **getAIPlayer()**, implementato direttamente nei factory methods in **GameFactoryImpl**.

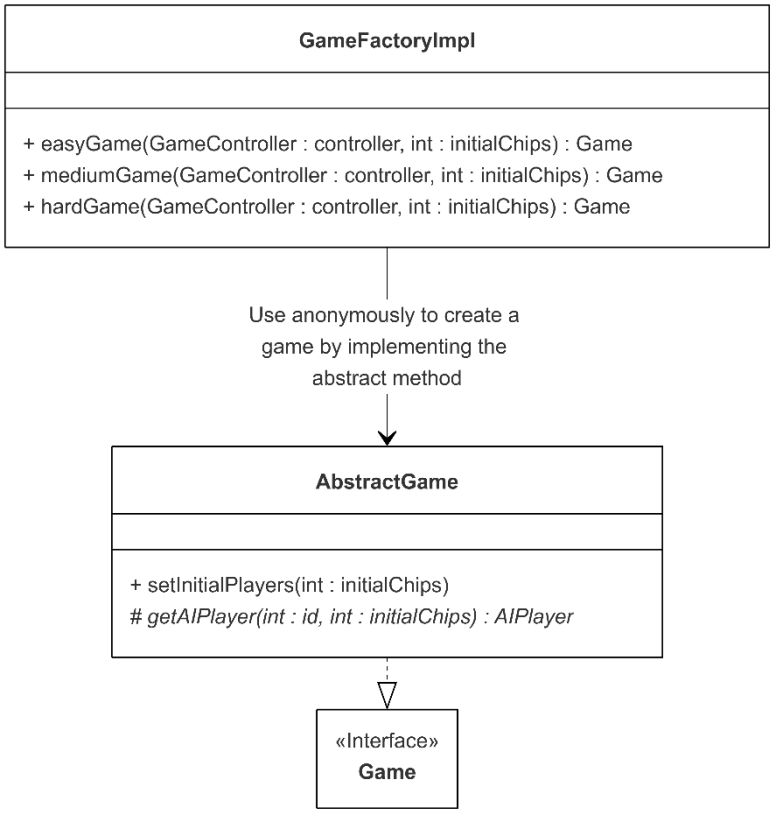

Figura 2.15: Rappresentazione UML del pattern Template Method per l'inizializzazione dei giocatori IA.

Gestione fasi di una mano

Problema: Il gioco si divide in più mani, ognuna delle quali si svolge seguendo sempre la stessa sequenza di passaggi. Il model, inoltre, deve essere costantemente aggiornato in quanto le risposte e le azioni che i componenti della partita possono fare, variano a seconda dello stato corrente.

Soluzione: Per rispettare il principio *SRP* è stato deciso di separare l'implementazione vera e propria delle mani e il mantenimento delle informazioni sullo stato della partita in due interfacce separate: ***Hand***, e ***State*** rispettivamente. Il gioco, implementato nella classe ***AbstractGame*** si compone di una serie di mani (***Hand***), a loro volta composte da una

serie di quattro fasi. Si è inoltre notato che la struttura delle singole fasi è la medesima quindi, per riusare il codice si è deciso di individuarle in un enum ***Phase*** e utilizzare un unico metodo **startPhase()** per la loro implementazione. In particolare questo itera la lista di giocatori ancora in gioco, chiedendo loro l'azione e aggiornando lo stato della partita e la lista stessa di conseguenza fino al termine della fase stessa. La classe ***Hand*** è inoltre responsabile di decretare il vincitore (della singola mano).

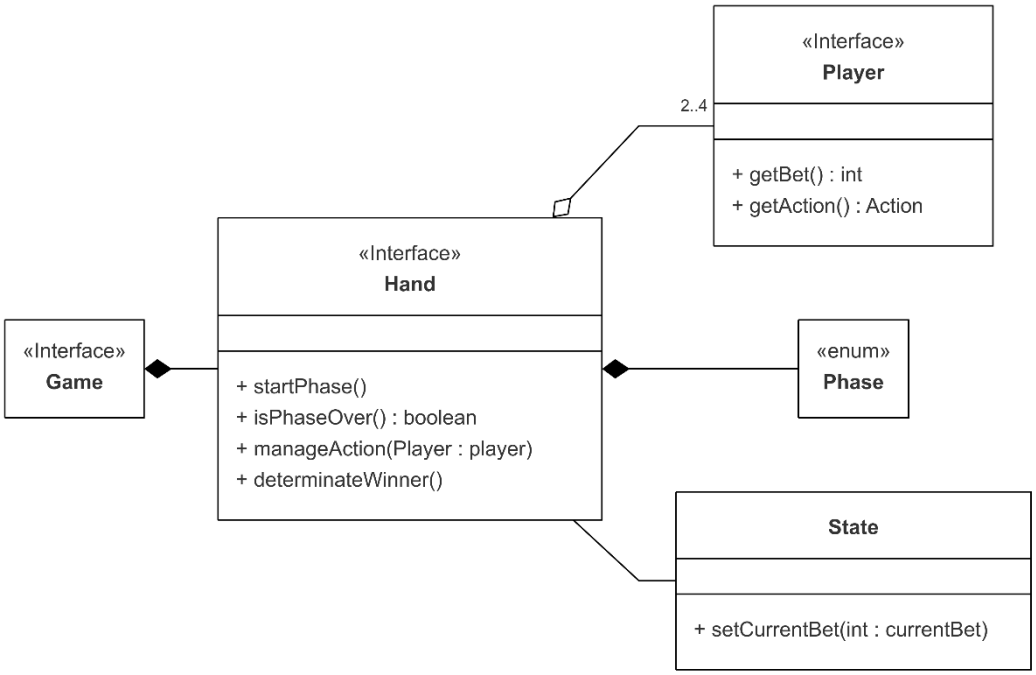

Figura 2.16 : Rappresentazione UML delle classi coinvolte nelle fasi di una mano.

GameLoop

Problema: Gestire l'avvicendamento delle fasi di gioco, controllando lo stato della partita. Quest'ultimo in particolare deve inoltre essere reso visibile all'utente tramite la View.

Soluzione: Si è notato che la partita segue sempre la stessa sequenza di fasi e passaggi fino al suo termine. Si è quindi deciso di gestire il loro avvicendamento nel tempo, e quindi la comunicazione tra i singoli componenti della partita, tramite un ***GameLoop***, ovvero un thread separato istanziato nel metodo **start()** della classe ***AbstractGame***. Al suo interno inoltre, vengono comunicati alla ***GameScene*** (la View), tramite la classe ***GameControllerImpl***, gli aggiornamenti da effettuare. Viene così garantita una chiara separazione tra *Model* e *View* che potranno comunicare unicamente attraverso il *Controller*, esso si occuperà anche di tradurre gli elementi del *Model* in elementi

rappresentabili nella *View*. Infine si è optato di legare la temporizzazione al tempo reale piuttosto che a quello della CPU per garantire la stessa resa su dispositivi diversi.

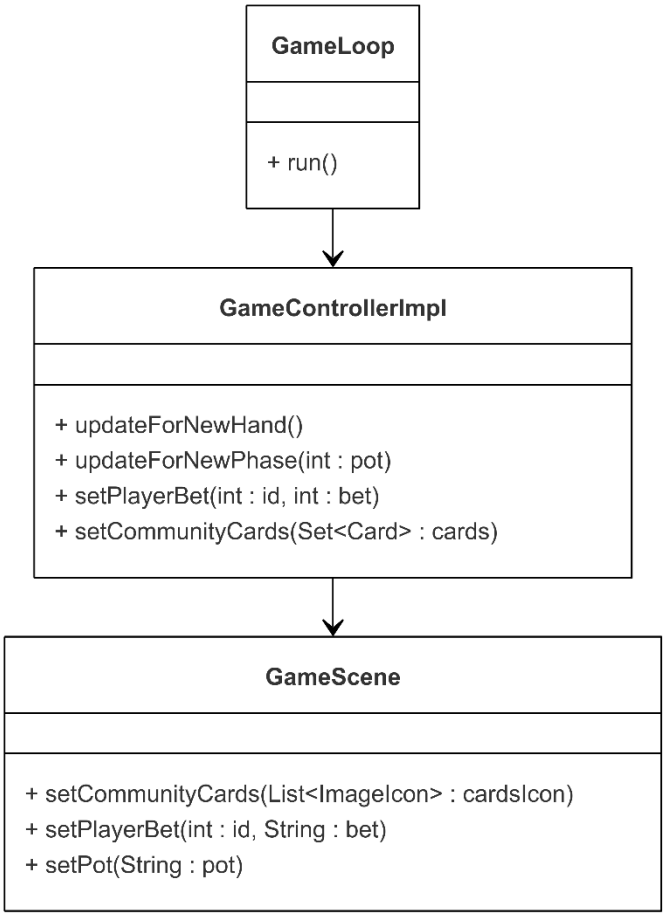

Figura 2.17 : Rappresentazione UML delle classi coinvolte.

Pausa partita

Problema: Nella schermata di gioco è presente un pulsante "Pause" con il quale l'utente può mettere in pausa il gioco. Viene inoltre aperta una schermata di pausa contenente un pulsante "Resume" per riprendere la partita dal punto in cui è stata interrotta.

Soluzione: L'interfaccia ***GameController*** ha tre metodi **pauseGame()**, **resumeGame()** e **waitIfPaused()**. I primi due sono richiamati nell'actionListener dei rispettivi pulsanti e settano il valore di una variabile booleana "isPaused", il terzo invece viene richiamato in punti opportuni del ***GameLoop*** e, in base al valore della variabile, sospende il thread fino a che esso non viene risvegliato. Per far sì che il valore della variabile letto sia quello effettivamente in memoria è stato deciso di impostare la variabile volatile. Inoltre, per garantire la mutua esclusione, sono stati utilizzati dei blocchi synchronized, vincolando l'esecuzione dei metodi ad un solo thread per volta. Al loro interno sono stati utilizzati dei metodi di sincronizzazione esplicita.

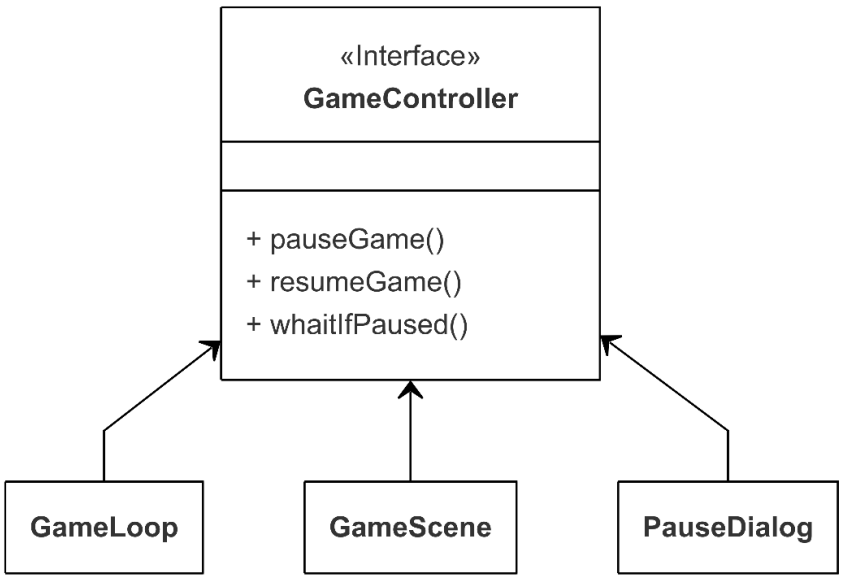

Figura 2.18 : Rappresentazione UML delle classi coinvolte.

Termine Partita

Problema: Nella schermata di pausa sono presenti due pulsanti "New game" e "Menu" con i quali è possibile uscire dalla partita. In entrambi i casi il thread del GameLoop deve terminare la sua attività prima del suo completamento.

Soluzione: L'interfaccia ***GameController*** ha due metodi **endGame()** e **isTerminated()**. Il primo è richiamato nell'actionListener dei due pulsanti nella ***PauseDialog*** (la schermata di pausa) mentre il secondo viene utilizzato nel ***GameLoop***. Entrambi accedono ad una variabile volatile booleana "isTerminated" all'interno di blocchi sincronizzati per garantire la mutua esclusione. È quindi stato utilizzato il tipo di terminazione detto *deferred cancellation* in cui è il target thread stesso, in questo caso il ***GameLoop***, a controllare

periodicamente all'interno del metodo **run()** se si sono verificate le condizioni per le quali deve terminare.

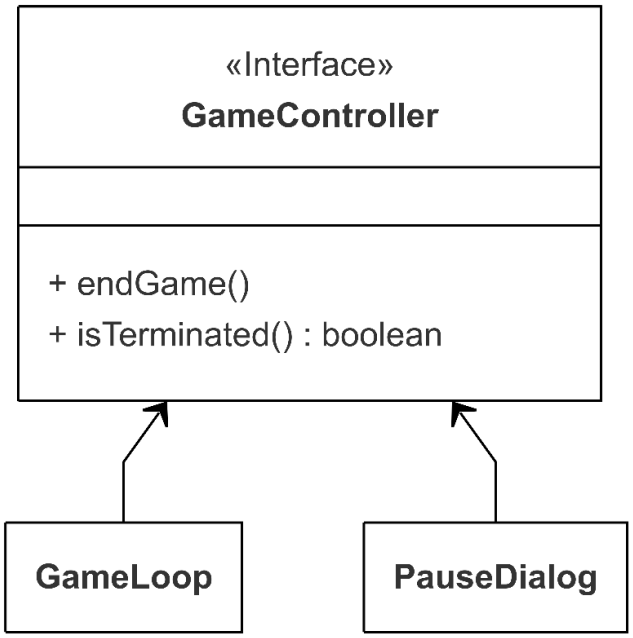

Figura 2.19 : Rappresentazione UML delle classi coinvolte.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per il testing automatizzato è stata utilizzata la libreria JUnit. Sono stati realizzati una serie di test su tutte le classi rilevanti del Model e alcune del Controller per garantire il corretto funzionamento dell'applicazione. Le classi della View, invece, sono state testate solo manualmente durante le fasi di sviluppo e collaudo del software poiché, a causa di mancanza di tempo, non è stato possibile approfondire le metodologie di testing automatizzato riguardanti le componenti grafiche. Gli aspetti su cui si è maggiormente soffermato il testing sono i seguenti:

- Gestione delle combinazioni, verificando se sono presenti in un dato set di carte e in tal caso determinandone la migliore, estrazione delle carte che formano tali combinazioni e il confronto di due combinazioni per determinare quale risulta essere la migliore.
- Generazione e gestione corretta di un mazzo di carte, ovvero la sua creazione, il suo rimescolamento, la distribuzione delle carte e il lancio di eccezioni se il mazzo è vuoto.
- Creazione del gioco, aggiornamento del suo stato e funzionamento delle mani.
- Verifica che i giocatori IA e utente funzionino correttamente, in particolare controllare le loro azioni e verificare le loro decisioni in diverse situazioni di gioco.
- Raccolta e mantenimento delle statistiche, nonché il loro salvataggio e caricamento su/da file.

3.2 Note di sviluppo

3.2.1 Cavina Mattia

Utilizzo di Lists della libreria Guava

https://github.com/Jackmo04/OOP24-poker-tex/blob/800d25f684b6f15a5bd6a94f1705bf03b25be05e/src/main/java/pokertexas/model/combination/CombinationUtilitiesImpl.java#L109

Utilizzo di Lambda Expression per concretizzare una Interfacce Funzionali

https://github.com/Jackmo04/OOP24-poker-tex/blob/800d25f684b6f15a5bd6a94f1705bf03b25be05e/src/main/java/pokertexas/model/deck/DeckFactoryImpl.java#L26

Utilizzo di Stream

https://github.com/Jackmo04/OOP24-poker-tex/blob/800d25f684b6f15a5bd6a94f1705bf03b25be05e/src/main/java/pokertexas/controller/card/CardGetterImageImpl.java#L28

https://github.com/Jackmo04/OOP24-poker-tex/blob/800d25f684b6f15a5bd6a94f1705bf03b25be05e/src/main/java/pokertexas/model/combination/CombinationCardGetterImpl.java#L51

3.2.2 Grandini Matteo

Utilizzo della libreria *SLF4J*

Utilizzata in diversi punti. Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/model/statistics/StatisticsManagerImpl.java#L30

Utilizzo di *Stream* e *Lambda*

Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/model/player/ai/AbstractAIPlayer.java#L44C1-L61C6

Scrittura di classi e interfacce *generiche bounded*

Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/model/statistics/StatisticsManagerImpl.java#L27

Utilizzo di *ImmutablePair* dalla libreria *Apache Commons*

Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/controller/statistics/BasicStatisticsControllerImpl.java#L60C5-L71C6

Utilizzo di *Optional*

Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/model/player/ai/AbstractAIPlayer.java#L103C1-L109C6

3.2.3 Mariani Matilde

Utilizzo della libreria SLF4J
Utilizzata in diversi punti. Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/782b854ba3b6916e7bbfdb34e06c97773549ed9a/src/main/java/pokertexas/controller/player/user/UserPlayerController.java#L18

Utilizzo Optional
Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/782b854ba3b6916e7bbfdb34e06c97773549ed9a/src/main/java/pokertexas/model/player/user/UserPlayerImpl.java#L44

Utilizzo di lock e meccanismi di sincronizzazione
Esempio: https://github.com/Jackmo04/OOP24-poker-tex/blob/782b854ba3b6916e7bbfdb34e06c97773549ed9a/src/main/java/pokertexas/controller/player/user/UserPlayerController.java#L72C5-L84C6

3.2.4 Piras Ilaria

Utilizzo della libreria SLF4J

Utilizzata in vari punti. Il seguente è un singolo esempio. Permalink: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/model/game/AbstractGame.java#L38

Utilizzo di Iterables dalla libreria Google Guava

Permalink: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/model/game/HandImpl.java#L114

Utilizzo di Stream e lambda

Utilizzate molte volte. Il seguente è un singolo esempio. Permalink: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/controller/game/GameControllerImpl.java#L171C5-L177C6

Utilizzo di lock e meccanismi di sincronizzazione

Utilizzata in vari punti. Il seguente è un singolo esempio. Permalink: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/controller/game/GameControllerImpl.java#L268C3-L278C6

Utilizzato codice esterno per la creazione del glassPane

Permalink nel progetto: https://github.com/Jackmo04/OOP24-poker-tex/blob/baca80a5d0d1720f8917d377b0b87a60e1eaf0b7/src/main/java/pokertexas/view/scenes/GameScene.java#L167C7-L182C40

Permalink source: https://stackoverflow.com/questions/45777039/how-to-create-a-game-option-pause-screen-in-swing#:~:text=JPanel%20glassPane%20%3D,setVisible(true)%3B

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Cavina Mattia

Valuto il mio lavoro in modo positivo in generale. Senza dubbio poteva essere fatto meglio infatti nel corso del progetto ho cambiato e migliorato sempre più la mia parte notando potesse essere fatta meglio, ma essendo un lavoratore a tempo pieno e, con il tempo a mia disposizione, sento di aver dato il mio contributo. Il mio ruolo all'interno del gruppo non credo sia stato fondamentale, in quanto per lo stesso motivo prima citato, mi sono dedicato a parti con le minime interferenze con quelle dei miei colleghi. L'esperienza ed il risultato in generale mi è piaciuto, mi sarebbe piaciuto aver più tempo per migliorarlo ancora. Voglio comunque ringraziare i miei compagni per essermi venuti in contro nelle mie esigenze e per la loro disponibilità. Le schermate finali le ho seguite io ed abbiamo espressamente richiesto agli interessati il permesso di usare le foto, l'unico che ancora non mi ha effettivamente risposto è Chuck Norris.

4.1.2 Grandini Matteo

In generale mi ritengo soddisfatto del lavoro svolto. Non avevo mai lavorato in applicazioni che richiedessero un grado di progettazione come quello richiesto da questo progetto e l'unica esperienza di programmazione in Java precedente era quella ottenuta dalle lezioni. Ho avuto perciò alcune difficoltà nel decidere come risolvere certi problemi, come la raccolta delle statistiche o il passaggio fra le diverse schermate dell'applicazione. Ho dovuto spesso modificare (a volte anche estensivamente) le soluzioni che avevo trovato inizialmente con altre, poiché o non funzionavano a dovere o non ne ero soddisfatto. Sono convinto che, se dovessi ritornare a questo progetto anche solo fra qualche mese, penserò "Che codice terrificante che ho scritto", ma allo stesso tempo, per il tempo a disposizione e le conoscenze attuali, penso di essermi impegnato al massimo. Per fortuna i miei colleghi sono sempre stati disponibili per chiarimenti e per condividere idee, sono stati incredibilmente gentili e, secondo me, hanno fatto davvero un ottimo lavoro. Se c'è una cosa che cambierei se potessimo ricominciare da capo sarebbe utilizzare JavaFX come libreria grafica invece che Swing.

4.1.3 Mariani Matilde

Mi reputo soddisfatta del lavoro svolto da me e dai miei compagni, essendo questo il mio primo progetto di grandi dimensioni svolto in gruppo. Vorrei ringraziare i miei compagni per il supporto e per avermi chiarito dubbi riguardanti il funzionamento del gioco. Mi sono trovata bene con questo gruppo di lavoro, essendo tutte persone disponibili al confronto, allo scambio di idee e alla cooperazione. Sicuramente potevo svolgere meglio i miei compiti e avrei voluto dedicargli più tempo. Mi rendo conto di non avere ancora molta esperienza con Java e di non sentirmi completamente sicura nel suo utilizzo: infatti inizialmente non mi sentivo preparata e non credevo di poter affrontare questo compito con le mie conoscenze, rendendo così le prime fasi di lavoro complesse per me. Sto comunque cercando di migliorare e di acquisire maggiore competenza. Nonostante ciò, penso di poter dire di essermi impegnata per dare il maggior contributo possibile. Sicuramente grazie a questo progetto ho acquisito maggiore familiarità con questo linguaggio, ho scoperto nuove nozioni e ho avuto modo di provare cosa vuol dire creare un programma in tutte le sue fasi. Devo dire che, a progetto terminato, sono contenta di come ho realizzato la parte relativa al controllo della ricezione degli input dell'utente e del successivo aggiornamento del pannello.

4.1.4 Piras Ilaria

Essendo questo il primo progetto di grandi dimensioni al quale ho partecipato, ero inizialmente molto preoccupata, specialmente in quanto, essendo un lavoro di gruppo, dal mio lavoro dipendeva anche la ben riuscita di quello degli altri. Con il proseguire del progetto, però, queste preoccupazioni sono diminuite. Mi sono trovata molto bene nel gruppo, poiché tutti erano sempre pronti al confronto e al dialogo rendendo l'esperienza molto stimolante. Per quanto riguarda il mio contributo al progetto, posso ritenermi abbastanza soddisfatta. Riguardando il lavoro svolto, infatti, mi rendo conto di quante nuove cose io abbia imparato e approfondito ma anche di aspetti che, con le mie attuali conoscenze, avrei potuto gestire e organizzare in maniera più efficiente. Ci sono state alcune parti, in particolar modo la gestione della pausa, nelle quali ho incontrato delle difficoltà, ma queste mi hanno spinta a fare delle ricerche molto interessanti, seppur impegnative. In conclusione, sono fiera di essere riuscita ad affrontare le difficoltà che si sono presentate e del risultato che siamo riusciti a ottenere. Ma, soprattutto, sono felice di aver provato, anche se solo in parte, cosa significa creare un programma in tutti i suoi aspetti.

Appendice A

Guida Utente

L'applicazione dispone di una schermata dedicata per dare una spiegazione di base su come si gioca a Poker Texas Hold'em, perciò in questa guida non saranno trattate le regole del gioco. È tuttavia fondamentale che l'utente riesca a muoversi all'interno dei menu e possa compiere azioni nel gioco; perciò, nonostante si sia cercato di costruire le interfacce grafiche in modo che fossero il più chiare possibile, verrà fornita una breve guida per le azioni meno banali.

- **Selezione della difficoltà e delle chips di partenza**: una volta selezionata l'opzione "New Game" dal menu principale, verrà visualizzata la schermata di Figura A.1. In questa scena è necessario scegliere una difficoltà selezionando uno dei tre pulsanti a fianco della difficoltà desiderata e inserire un numero intero positivo nella targhetta "Enter chips and press enter". È **fondamentale** la pressione del tasto Invio dopo l'inserimento del valore di chips di partenza. Se entrambe le scelte sono state effettuate correttamente, sarà possibile premere il pulsante "PLAY" e iniziare la partita.

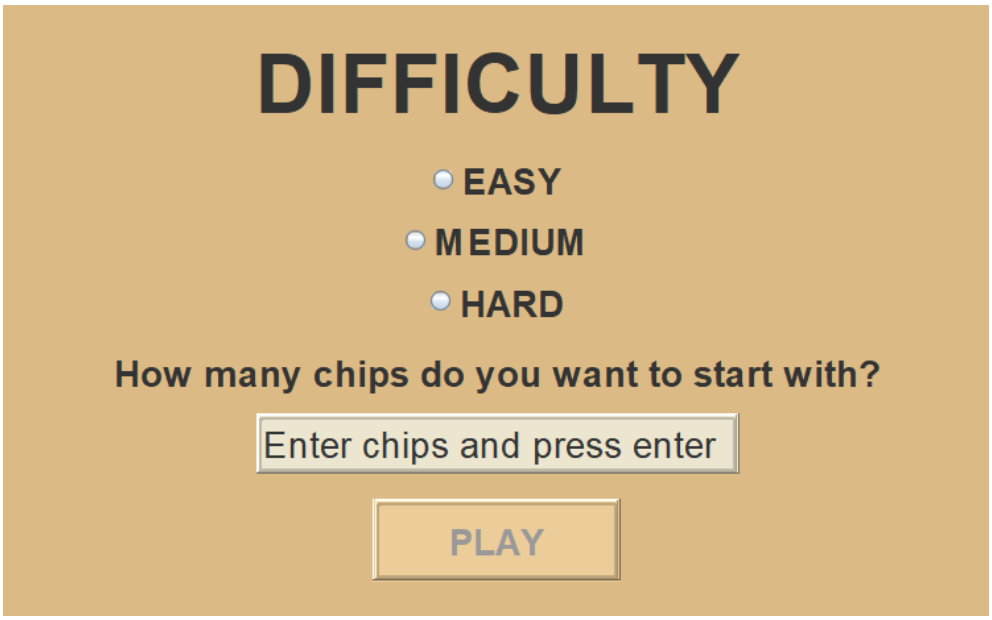

Figura A.1: Screenshot della schermata di selezione della difficoltà di gioco.

- **Scelta delle azioni nel gioco**: durante una partita, l'utente può agire utilizzando i pulsanti ripostati in Figura A.2. Sono abilitati solo i pulsanti delle azioni che sono effettuabili in un dato momento. Nel caso si volesse effettuare una *Raise*, è necessario inserire un importo valido di chips nell'etichetta "Insert your bet

here…", **digitare Invio**, e infine premere il pulsante "Raise". Le altre azioni sono di banale utilizzo.

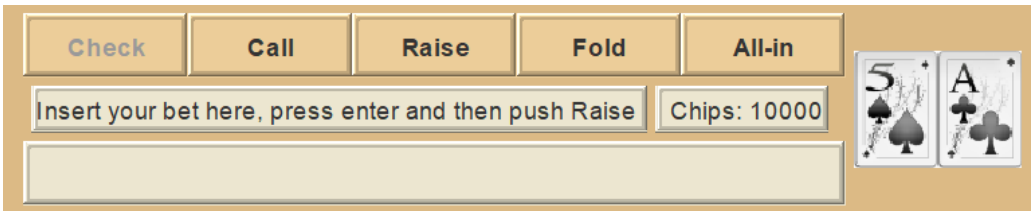

Figura A.2: Screenshot dei pulsanti per agire durante una partita.