UNO!

Luca Pasini, Leonardo Perretta, Luca Fagioli

12 febbraio 2026

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software ha come obiettivo la realizzazione di una trasposizione digitale del celebre gioco di carte UNO, espandendone le funzionalità per includere diverse varianti ufficiali e modalità di gioco personalizzabili.
Nella sua versione classica, UNO è un gioco in cui i giocatori, a turno, devono scartare una carta dalla propria mano che corrisponda per colore o per numero a quella presente in cima al mazzo di scarto. L'obiettivo principale è essere il primo giocatore a rimanere senza carte in mano. Durante la partita, l'uso di carte speciali come carte "Azione" (*Skip*, *Reverse*, *Draw Two*, etc.) oppure carte "Jolly" (*Wild*, *Draw Four*, etc.) permette di influenzare l'andamento del gioco, penalizzando gli avversari o alterando l'ordine dei turni.
Oltre alla modalità "Classica", il sistema integra logiche di gioco avanzate per supportare varianti complesse come *UNO Flip* e *UNO All Wild*. Il primo introduce un mazzo a due facce (lato chiaro e lato scuro) e una carta speciale (*Flip*) che capovolge l'intero mazzo e le mani dei giocatori, cambiando drasticamente le regole e le penalità di alcune carte in corso d'opera; il secondo è una variante in cui ogni carta è una carta "Jolly" (*Wild*), eliminando così la necessità di abbinare i colori e focalizzando la strategia esclusivamente sulle azioni speciali.
Il software permette all'utente di configurare la partita attraverso un menu iniziale, selezionando la modalità di gioco desiderata e varie regole personalizzabili. Le partite saranno contro avversari controllati dal computer, i quali adotteranno strategie differenti in base alla variante di gioco scelta. Il sistema gestisce autonomamente il flusso dei turni, la validazione delle mosse, l'applicazione delle penalità e il calcolo del punteggio finale.

1.1.1 Requisiti funzionali

- Il sistema dovrà garantire la selezione della modalità di gioco prima dell'inizio della partita, permettendo di scegliere tra la variante "Classica", la variante "Flip" e la variante "All Wild".

- Il giocatore, durante il proprio turno, dovrà scartare una carta dalla propria mano che sia compatibile per colore, numero o simbolo con la carta in cima sul mazzo degli scarti. Se non possiede mosse valide, sarà costretto a pescare una carta dal mazzo.

- Il sistema dovrà gestire l'uso delle carte azione speciali (come *Skip*, *Reverse*, *Draw Two*, etc.) applicando immediatamente l'effetto corrispondente sul flusso dei turni o sulle carte in mano agli avversari.

- Nella modalità "UNO Flip", il gioco dovrà gestire un mazzo a doppia faccia (lato chiaro e lato scuro). Giocando la carta speciale *Flip*, il sistema dovrà capovolgere istantaneamente il mazzo di pesca, gli scarti e le mani di tutti i giocatori, attivando delle nuove carte con regole e penalità più severe (es. *Draw Five*, *Skip Everyone*).

- Nella modalità "All Wild", il sistema eliminerà il vincolo dei colori, rendendo ogni carta giocabile come un Jolly, e dovrà gestire azioni esclusive come lo *Skip Two* o il *Forced Swap* che scambia le mani tra due giocatori.

- Il software dovrà includere giocatori controllati dal computer in grado di prendere decisioni autonome. L'IA dovrà adattare la propria strategia in base alla variante giocata.

- Il sistema dovrà rilevare automaticamente la condizione di vittoria (quando un giocatore rimane senza carte), calcolare il punteggio basato sulle carte rimaste in mano agli avversari e gestire la fine della partita o del round.

- L'interfaccia dovrà visualizzare chiaramente lo stato attuale del gioco: il colore attivo, la direzione del turno (oraria o antioraria), l'ultima carta giocata e i punteggi dei giocatori.

- Il sistema dovrà supportare la personalizzazione di alcune regole della partita. Il motore di gioco dovrà essere in grado di recepire queste configurazioni all'avvio e adattare dinamicamente la logica in base alle regole scelte.

1.1.2 Requisiti non funzionali

- L'applicazione dovrà essere progettata in modo modulare per permettere l'aggiunta futura di nuove varianti di gioco o nuove regole senza dover riscrivere il nucleo logico esistente.
- Il gioco dovrà essere accessibile attraverso un'interfaccia grafica (GUI) intuitiva, capace di adattarsi alle dimensioni dello schermo e di fornire un feedback visivo immediato alle azioni dell'utente (es. cambio colore, rotazione del mazzo).
- L'applicazione dovrà garantire la portabilità ed essere in grado di essere eseguita sui principali sistemi operativi.

1.2 Modello del dominio

Nella versione fisica del gioco di carte, la dinamica si basa su una rotazione ciclica dei turni durante la quale i giocatori gestiscono la propria mano di carte, tentando di esaurirla prima degli avversari. I partecipanti devono coordinarsi per rispettare l'ordine di gioco, che può subire inversioni o interruzioni, e per verificare la validità di ogni scarto rispetto alla carta visibile sul tavolo. Per modellare questo aspetto, è stata concepita un'entità logica centrale di coordinamento, che chiameremo Partita (*Game*), a cui sono delegate tutte le operazioni di gestione del regolamento e dei partecipanti. Queste operazioni sono:

- Gestire le regole (*GameRules*) selezionate dall'utente prima della selezione della modalità di gioco.
- Coordinare il flusso dei turni (*TurnManager*), determinando quale Giocatore ha il diritto di agire e in quale direzione (oraria o antioraria) procede il gioco.
- Validare le mosse (*MoveValidator*), verificando che la carta giocata sia compatibile per colore, numero o simbolo con quella in cima alla pila degli scarti.
- Amministrare le penalità e gli effetti speciali, imponendo ai giocatori di pescare carte o di saltare il turno in risposta alle azioni degli avversari.
- Decretare gli stati della partita (*GameState*), come la fine del round o della partita quando un giocatore esaurisce la propria mano, calcolando i punteggi (*ScoreManager*) in base alle carte rimaste in possesso degli sconfitti.

Il Giocatore (*Player*) rappresenta l'attore protagonista della competizione, l'entità dotata di potere decisionale. Ogni partecipante custodisce una mano di carte, una collezione privata e nascosta agli avversari, e il suo obiettivo è esaurirla. Nel modello del dominio, il giocatore è concepito come un ruolo astratto che può essere ricoperto indifferentemente da un utente umano o da un'intelligenza artificiale. A prescindere dalla sua natura, il Giocatore interagisce con la Partita secondo un ciclo definito: quando è il proprio turno analizza le carte nella sua mano confrontandole con i vincoli imposti dalla pila degli scarti (colore, numero) ed esercita una scelta, che può consistere nel giocare una carta valida, nel pescare dal mazzo o nel dichiarare un nuovo colore tramite un Jolly e chiamare Uno quando rimane con solo una carta in mano.
Come detto precedentemente, a ogni giocatore viene periodicamente concesso il turno d'azione. L'unità fondamentale con cui il giocatore interagisce è la Carta (*Card*). A differenza di altri giochi dove le carte sono valori statici, in questo dominio la Carta è un'entità che definisce le possibili interazioni. Esse si dividono in carte numeriche, che rappresentano le risorse base, in carte azione (*Skip*, *Reverse*, *Draw Two*, etc.) e carte jolly (*Wild*, *Draw Four*, etc.), che agiscono come eventi in grado di alterare lo stato della partita. Le carte sono organizzate in due raggruppamenti fisici fondamentali: il Mazzo di Pesca, da cui i giocatori attingono risorse quando non hanno mosse valide, e la Pila degli Scarti (*DiscardPile*), la cui carta superiore rappresenta lo "stato corrente" del tavolo e detta i vincoli per la mossa successiva. La gestione di queste due pile è critica: quando il mazzo di pesca si esaurisce, la pila degli scarti (eccetto l'ultima carta) deve essere rigenerata e mescolata per permettere al gioco di continuare all'infinito.
Una complessità specifica di questo dominio risiede nella gestione delle Varianti (*Flip* e *All Wild*), che trasformano radicalmente il comportamento degli oggetti di gioco. Nella variante Flip, le carte assumono una natura duale: possiedono un "Lato Chiaro" e un "Lato Scuro", ognuno con propri colori e valori. L'entità di coordinamento deve quindi tenere traccia non solo di chi tocca, ma anche di quale lato del mazzo è attualmente attivo, gestendo transizioni che capovolgono istantaneamente tutte le carte in gioco (sia quelle nei mazzi che quelle nelle mani dei giocatori). Nella variante "All Wild", invece, viene meno il vincolo del colore, spostando il focus strategico puramente sugli effetti delle carte jolly.
Parte della difficoltà progettuale consisterà nel rappresentare in modo flessibile queste regole mutevoli. Un'azione come "scartare una carta" non è un evento isolato, ma comporta una catena di conseguenze gestite da più entità logiche: la carta lascia la mano del giocatore, viene validata in base alla pila degli scarti, il suo effetto viene eseguito (magari invertendo il giro o

costringendo il prossimo a pescare), e infine il turno passa. Sarà quindi necessario progettare un'architettura che permetta a queste entità (Giocatore, Mazzi, Regole) di comunicare in maniera efficace, mantenendo lo stato del gioco coerente anche a fronte di effetti che ne stravolgono le meccaniche di base.

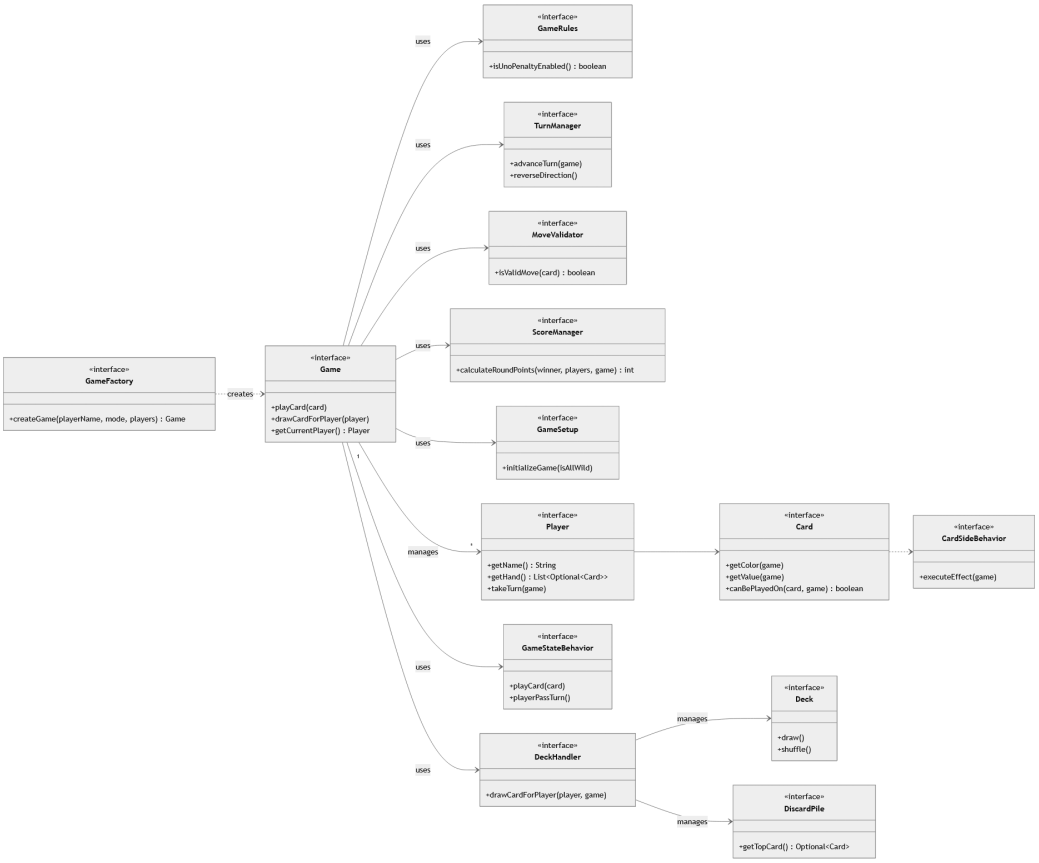

Figura 1.1: Schema UML dell'analisi del problema, con rappresentazione delle entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettura del sistema è stata progettata seguendo il pattern Model-View-Controller (MVC). L'impiego di questo pattern risponde alla necessità di gestire la logica a eventi tipica di un match di carte: ogni transizione di stato (cambio turno, carta giocata, penalità applicata) è scatenata da un input discreto dell'utente o da una decisione dell'intelligenza artificiale, rendendo necessaria una chiara separazione tra la logica di gestione (Controller), i dati della partita (Model) e la loro rappresentazione grafica (View).
A livello macroscopico, il sistema è orchestrato da un punto di ingresso iniziale, il MenuController, che si occupa della configurazione della partita (scelta delle regole e della modalità). Una volta terminata la fase di setup, il controllo viene ceduto al GameController, che diventa il cuore dell'applicazione durante lo svolgimento del gioco.
Al fine di garantire un rigoroso disaccoppiamento tra le componenti, sono stati definiti precisi punti d'ingresso basati su interfacce che evitano dipendenze dirette tra le implementazioni. Il ruolo di mediatore è affidato al Controller (GameController). Esso agisce attraverso il pattern Observer in una duplice funzione: da un lato intercetta gli eventi generati dall'interazione utente sulla View, dall'altro reagisce prontamente alle notifiche di cambiamento di stato provenienti dal Model.
Il cuore della logica di dominio è incapsulato nel Modello, accessibile tramite l'interfaccia Game. Per evitare la degenerazione in una "God Class", questa interfaccia viene divisa in un sottosistema articolato di gestori specializzati: il GameSetup per inizializzare il gioco (distribuire le carte ai giocatori e girare la prima carta nella pila degli scarti), il GameStateBehavior per decidere cosa il giocatore può fare in un determinato momento, il TurnManager per

la sequenzialità dei turni, il DeckHandler per la gestione fisica delle carte e i moduli MoveValidator e ScoreManager per la verifica delle regole e il calcolo dei punteggi. Un aspetto cruciale di questa progettazione è la protezione dei dati: il Modello non espone mai direttamente le proprie strutture interne alla vista, ma comunica lo stato corrente esclusivamente attraverso oggetti di trasferimento dati (DTO) in sola lettura, definiti dall'interfaccia GameViewData.
A chiudere vi è la View (GameScene), dedicata esclusivamente alla presentazione visiva. Mantenendo un profilo totalmente passivo e privo di capacità decisionali, la View non agisce mai direttamente sul modello in risposta agli input dell'utente, ma si limita a notificare le intenzioni di gioco all'osservatore registrato (il Controller) sfruttando gli Observer. La comunicazione tra i componenti del sistema si articola attraverso un flusso circolare continuo, innescato dall'interazione diretta dell'utente. La View, responsabile di catturare gli input, non elabora direttamente le richieste ma delega la gestione dell'evento al Controller. Quest'ultimo agisce come decisore, interpretando l'intenzione del giocatore e traducendola in comandi rivolti al Model. A seguito di tali direttive, il Model evolve il proprio stato interno applicando le regole di dominio e, una volta consolidata la modifica, notifica il cambiamento senza preoccuparsi di come verrà rappresentato. Il ciclo si conclude con il Controller che, ricevendo la notifica di aggiornamento, estrae le nuove informazioni di stato e le trasmette alla View, la quale provvede infine a sincronizzare l'interfaccia grafica con la nuova situazione di gioco.

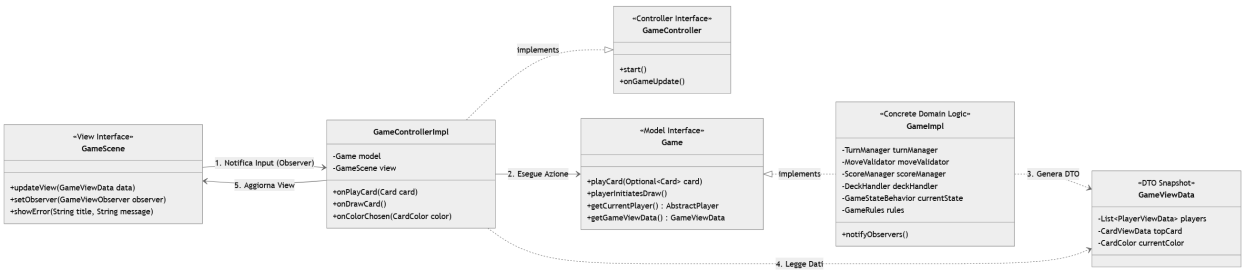

Figura 2.1: Semplice UML del dominio dell'applicazione.

2.2 Design Dettagliato

2.2.1 Luca Fagioli

Gestione dei comportamenti delle carte

Problema In UNO, le carte non sono semplice entità contenitori di dati (colore e numero), ma possiedono effetti attivi che modificano il flusso di gioco (inversioni, salti, pesca carte). Una soluzione banale avrebbe comportato l'uso di lunghe catene di if-else o switch per verificare il tipo di carta ed eseguirne l'effetto. Con questo approccio l'introduzione di una nuova carta (es. "Salto Due" nella variante All Wild) avrebbe richiesto la modifica del codice esistente in molteplici punti.

Soluzione Ho scelto di utilizzare lo Strategy Pattern per incapsulare l'algoritmo di esecuzione dell'effetto. La carta non "sa" cosa fa, ma delega l'esecuzione a un oggetto comportamento. Ho definito l'interfaccia CardSideBehavior, che rappresenta la strategia astratta. Le implementazioni concrete

(es. ActionBehavior, NumericBehavior, WildBehavior, FlipBehavior) definiscono le specifiche logiche di gioco. In questo modo, la classe Card mantiene un riferimento di tipo CardSideBehavior. Quando una carta viene giocata, il sistema invoca il metodo performAction() sulla strategia corrente, senza preoccuparsi del tipo specifico di carta.

Gestione delle carte

Problema Nella versione classica di UNO, l'entità carta è strutturalmente semplice: possiede una faccia anteriore con attributi specifici (colore, valore) e una faccia posteriore ("back") puramente estetica e senza funzione. Tuttavia con l'introduzione della variante "UNO Flip!" ogni carta fisica possiede due logiche distinte (Lato Chiaro e Lato Scuro), entrambe giocabili e dotate di valori, colori ed effetti differenti. Cercare di modellare questa dualità estendendo semplicemente una classe base o scambiando gli oggetti in memoria al momento del "Flip" avrebbe comportato gravi problemi di consistenza dei dati (riferimenti invalidati nelle mani dei giocatori) e costringendo la carta a gestire logicamente la propria mutazione.

Soluzione La soluzione che ho adottato si basa sulla classe DoubleSidedCard che agisce come un contenitore. Questa classe non definisce direttamente colore o valore come campi primitivi, ma aggrega al suo interno due istanze distinte di CardSideBehavior: `lightSide` che incapsula il comportamento e i dati del Lato Chiaro, e `darkSide` che incapsula il comportamento e i dati del Lato Scuro. Quando viene invocato un metodo sulla carta (es. `getColor()` o `executeEffect()`), la classe non esegue la logica direttamente, ma reindirizza la chiamata al behavior corrispondente al lato attualmente attivo. Questo design garantisce che la transizione tra le due modalità di gioco sia istantanea e trasparente per il resto dell'applicazione: l'oggetto carta rimane lo stesso in memoria, ma il suo comportamento esterno cambia radicalmente in risposta allo stato del gioco.

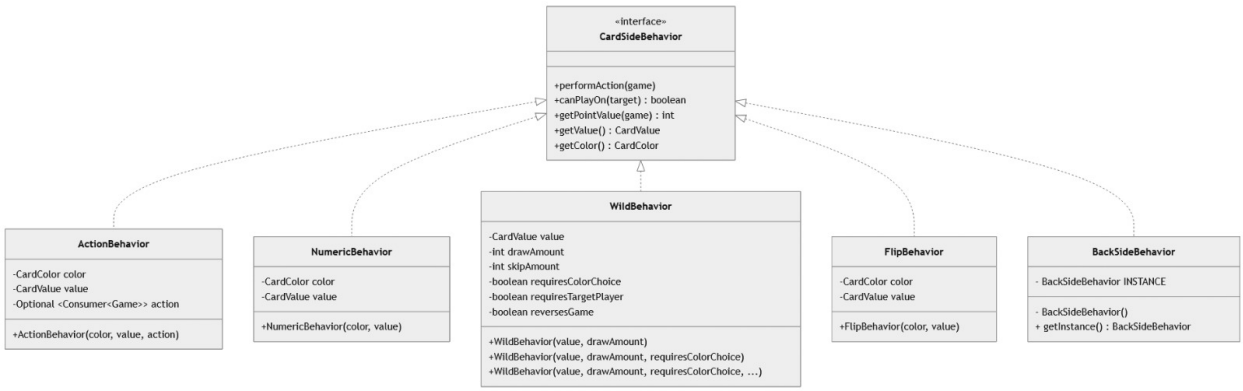

Figura 2.2: Schema UML della struttura dei Behavior.

Ottimizzazione e integrità del lato inattivo

Problema L'architettura della classe DoubleSidedCard impone strutturalmente che ogni carta sia composta da due comportamenti distinti (fronte e retro) per garantire l'uniformità del trattamento. Tuttavia, nelle carte standard, il retro è funzionalmente inerte e privo di dati specifici. Assegnare il valore null a questo componente esporrebbe il sistema a rischi di NullPointerException e obbligando a continui controlli difensivi. D'altro canto, istanziare un nuovo oggetto "vuoto" per il retro di ogni singola carta del mazzo comporterebbe un inutile spreco di memoria, dato che tale comportamento è concettualmente identico per tutte le carte.

Soluzione Ho implementato il Singleton Pattern per la classe BackSideBehavior. Poiché il comportamento di un lato inattivo è stateless e immutabile, una singola istanza è sufficiente per rappresentare il retro di qualsiasi carta nel gioco. Questa classe implementa l'interfaccia comune CardSideBehavior, ma i suoi metodi sono progettati per lanciare eccezioni runtime qualora invocati. Questo approccio garantisce l'integrità del flusso di gioco: qualsiasi tentativo di interagire con il lato sbagliato della carta viene intercettato istantaneamente come errore logico critico.

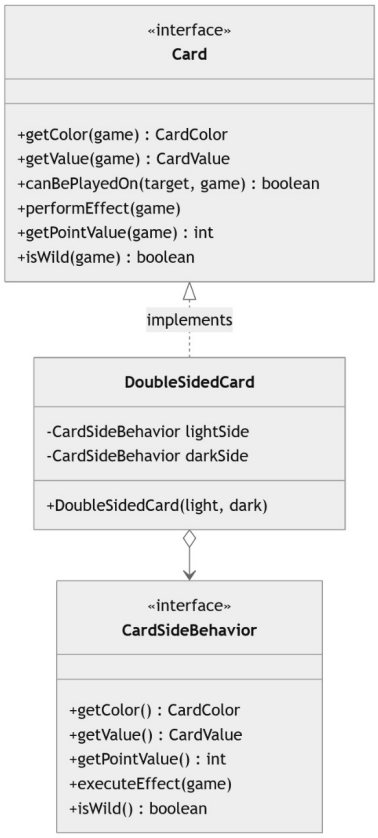

Figura 2.3: Schema UML della struttura delle Card.

Validazione delle mosse e Calcolo punteggi

Problema Nelle fasi iniziali di sviluppo, la classe principale GameImpl tendeva ad accentrare su di sé non solo la gestione dello stato, ma anche l'applicazione delle regole di gioco, evolvendo pericolosamente verso una "God Class". Mantenere la logica di verifica delle mosse e calcolo punteggio all'in-

terno del gestore della partita rendeva il codice rigido e difficile da testare singolarmente.

Soluzione La responsabilità del controllo delle regole è stata estratta e isolata nel componente MoveValidator. Ho progettato questa classe che non modifica lo stato del gioco, ma si limita a ricevere in input la carta candidata e la carta attuale in cima agli scarti (insieme al contesto necessario, come il colore attivo), restituendo un verdetto booleano sulla validità dell'azione. La responsabilità del calcolo è stata delegata al componente specializzato ScoreManager, che ricevendo in input il vincitore calcola il punteggio delle mani degli avversari e lo aggiunge al suo score.

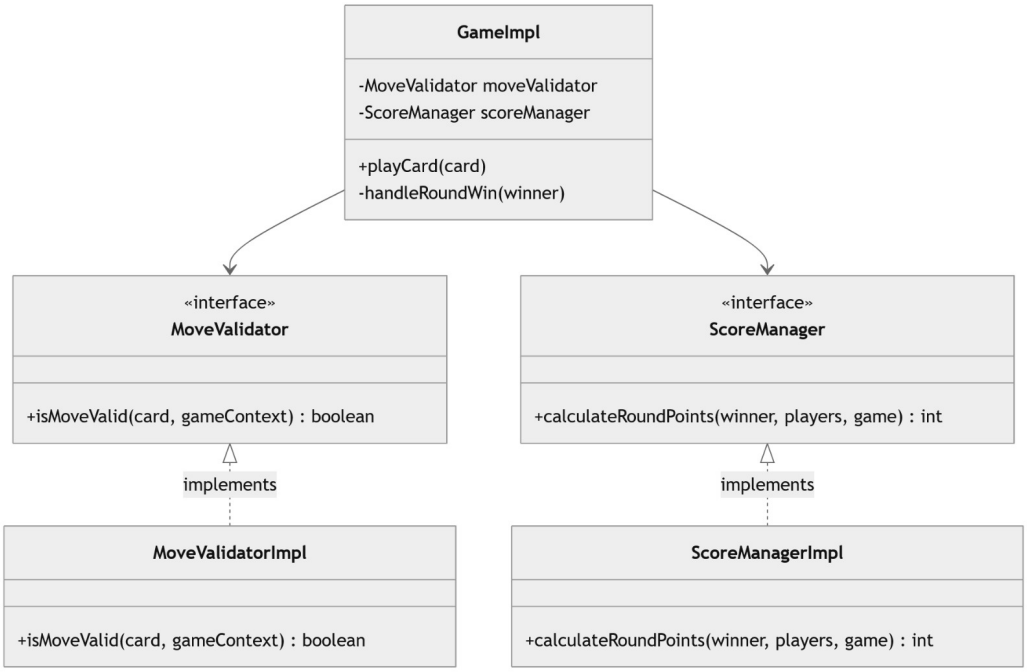

Figura 2.4: Schema UML della struttura del MoveValidator e dello ScoreManager.

2.2.2 Luca Pasini

Gestione degli stati del gioco

Problema Il gioco UNO non è un processo lineare uniforme; attraversa fasi distinte in cui le regole di interazione cambiano drasticamente. Ad esempio, durante il normale svolgimento (Running), un giocatore può giocare carte o pescare. Tuttavia, se viene giocato un "Jolly", il gioco entra in una fase di sospensione (WaitingForColor) in cui nessun altro input è accettato se non la scelta del nuovo colore. Gestire questa logica con una serie di flag booleani all'interno della classe principale GameImpl avrebbe portato a un codice fragile, pieno di istruzioni condizionali annidate e difficile da mantenere.

Soluzione Ho applicato lo State Pattern. La logica dipendente dallo stato è stata estratta dalla classe GameImpl e incapsulata in classi dedicate che implementano l'interfaccia comune GameState. GameImpl mantiene un riferimento allo stato corrente (currentState). Lo stato può essere:

- *Running*: Durante il normale flusso di gioco.
- *WaitingForColor*: Accetta solo l'input di cambio colore e blocca altre azioni.
- *WaitingForPlayer*: Accetta solo l'input di cambio colore e blocca altre azioni.
- *RoundOver*: Gestisce la fine del round.
- *GameOver*: Gestisce la fine della partita.

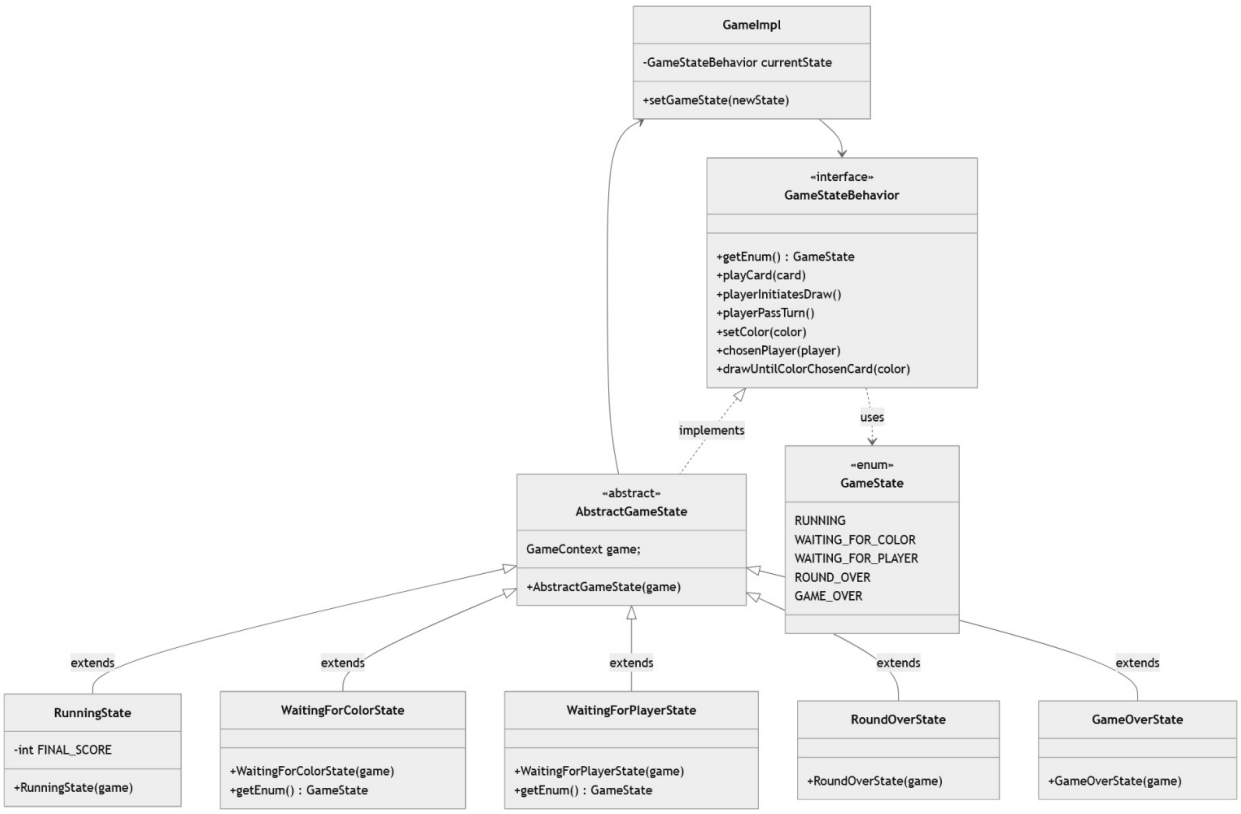

Figura 2.5: Schema UML degli Stati del gioco.

Creazione dei mazzi

Problema Il sistema deve supportare diverse tipologie di mazzi (StandardDeck, FlipDeck, AllWildDeck). Sebbene la composizione delle carte vari, le operazioni fondamentali sono identiche: mescolare, pescare, gestire l'esaurimento delle carte e ricreare il mazzo dagli scarti. Duplicare questa logica in ogni classe di mazzo avrebbe violato il principio DRY.

Soluzione Ho definito una classe base astratta AbstractDeckImpl che implementa l'interfaccia Deck. Questa classe definisce lo scheletro dell'algoritmo per la gestione delle carte (es. metodi shuffle, draw, refill), lasciando

alle sottoclassi il compito specifico di definire quali carte inserire nel mazzo. Le classi concrete come StandardDeck estendono questa base e si limitano a popolare la struttura dati nel loro costruttore, ereditando tutta la logica di manipolazione.

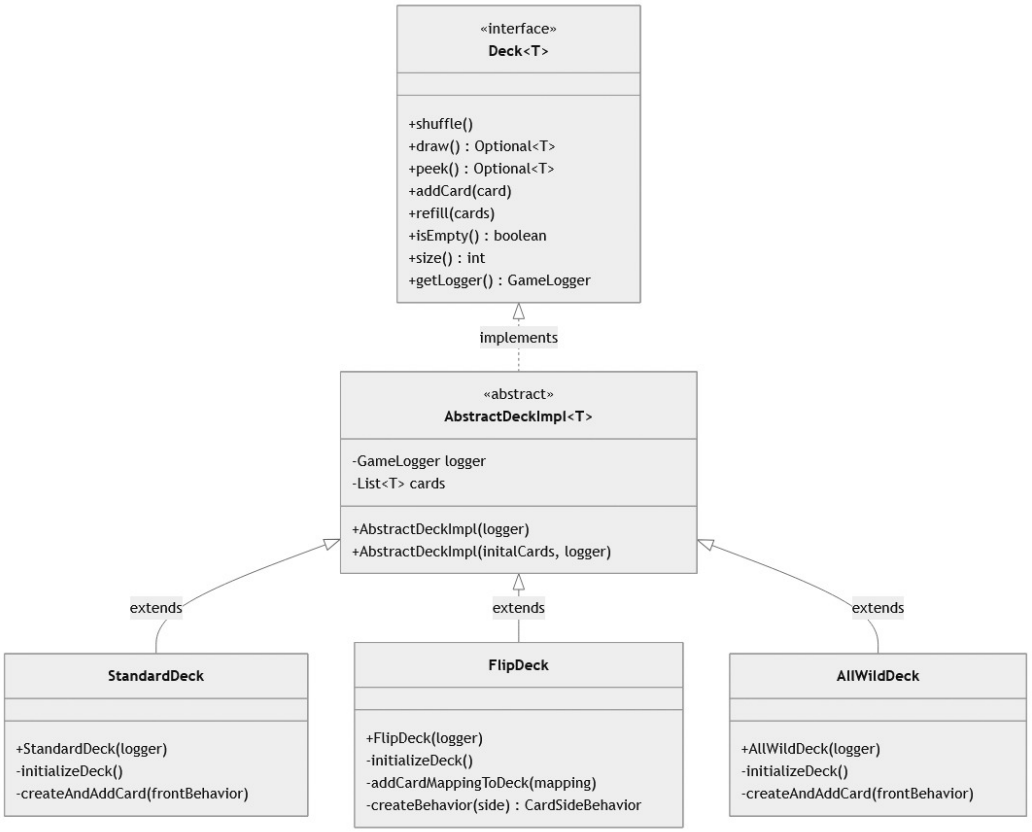

Figura 2.6: Schema UML della struttura del Deck.

Gestione del flusso e della turnazione

Problema La gestione dei turni in UNO è complessa: non è una semplice rotazione circolare. La direzione può invertirsi (senso orario/antiorario) dinamicamente e i giocatori possono essere "saltati". Implementare questa logica direttamente nel Game renderebbe la classe troppo grande e "tuttofare", violando l'SRP.

Soluzione Ho deciso si affidare questa logica al componente TurnManager. Questa classe incapsula la lista dei giocatori, mantiene un puntatore all'indice corrente e gestisce le azioni come il salto turno o il cambio direzione.

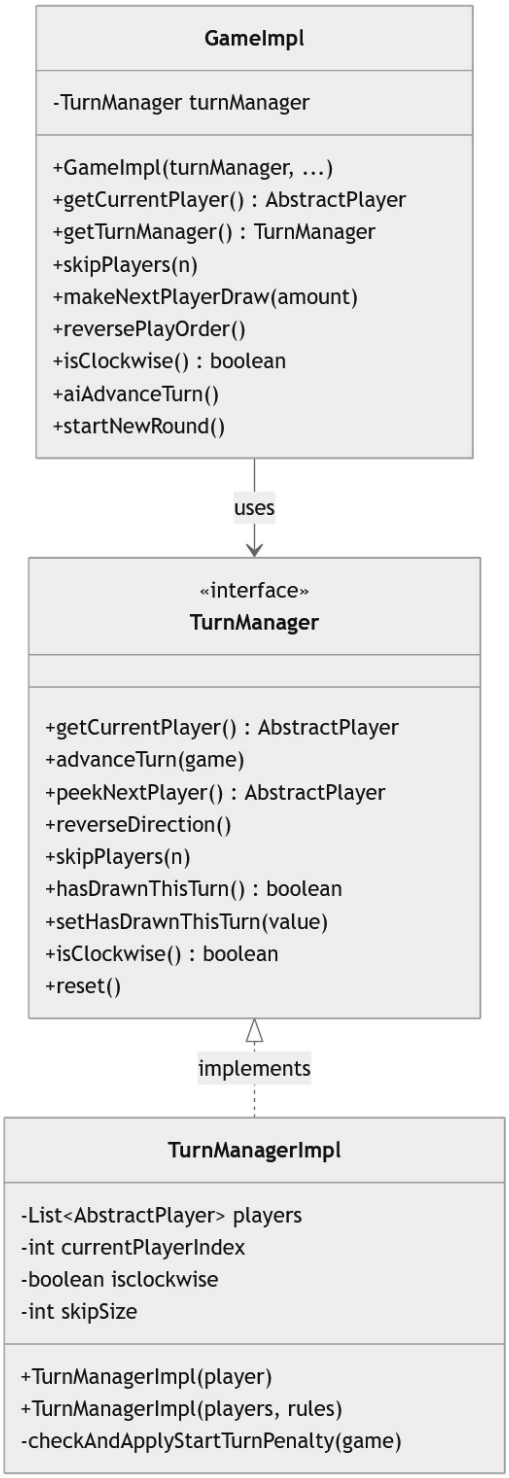

Figura 2.7: Schema UML della struttura del TurnManager.

Configurazione e mapping del mazzo flip

Problema Nella variante "Flip", esiste una corrispondenza biunivoca specifica tra ogni carta del "Lato Chiaro" e la sua controparte nel "Lato Scuro". Implementare queste associazioni direttamente nel codice sorgente della classe FlipDeck avrebbe prodotto centinaia di righe di codice ripetitivo.

Soluzione Ho personalmente mappato le coppie di carte (come nel mazzo fisico di Uno Flip) in un file JSON. Per importare questi dati nell'applicazione, è stato definito uno specifico Data Transfer Object (DTO) chiamato DoubleSidedEntryDTO. Questa classe agisce come un contenitore temporaneo durante la fase di deserializzazione: questa rispecchia la struttura del file JSON (contenente i campi per identificare la carta frontale e quella posteriore). All'avvio, il FlipDeck utilizza una libreria di parsing (GSON) per leggere il file e trasformarlo in una lista di DoubleSidedEntryDTO, che vengono poi processati per istanziare le vere carte di gioco DoubleSidedCard.

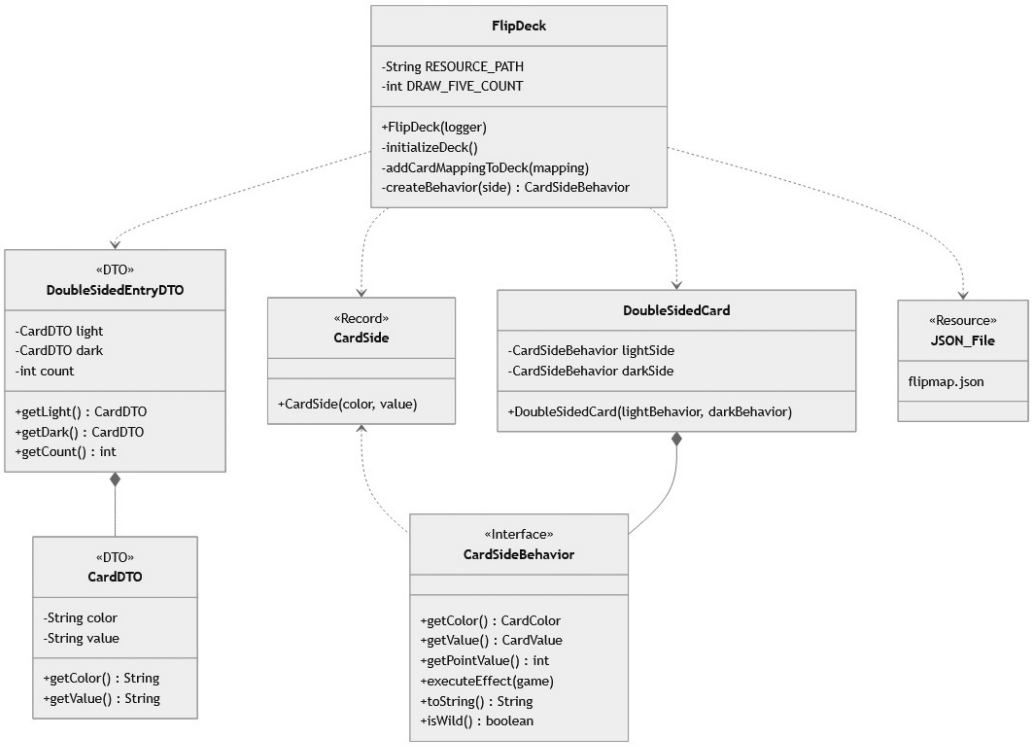

Figura 2.8: Schema UML della struttura per la configurazione del FlipDeck.

Gestione della pila degli scarti

Problema In una fase preliminare di analisi, avevo pensato alla pila degli scarti come una proprietà statica della classe Deck, assumendo che fosse concettualmente solo la "parte esausta" del mazzo di pesca. Tuttavia, questa modellazione rendeva complessa la gestione di logiche avanzate, come il riciclo delle carte quando il mazzo si esaurisce o l'interazione con regole specifiche.

Soluzione Ho deciso di pensare alla pila degli scarti come un componente effettivo del gioco, definendo l'interfaccia dedicata DiscardPile. Nel design attuale, la pila degli scarti è un componente autonomo, istanziato dalla GameFactory separatamente dal mazzo e iniettato nel contesto di gioco (Game). Questo perchè il Deck è responsabile solo delle operazioni di pesca e mescolamento, mentre il DiscardPile mantiene il riferimento alla carta attiva necessaria per la validazione delle mosse. Inoltre, quando il mazzo si esaurisce, l'entità DeckHandler può coordinare esplicitamente l'interazione tra i due componenti distinti, prelevando le carte dagli scarti per rigenerare il mazzo, mantenendo però l'architettura modulare e testabile.

Coordinamento delle risorse di gioco

Problema Durante una partita, l'interazione tra il mazzo di pesca (Deck) e la pila degli scarti (DiscardPile) è costante e complessa. Quando un giocatore pesca e il mazzo è vuoto, è necessario prendere gli scarti, mescolarli e creare un nuovo mazzo di pesca, salvaguardando l'ultima carta giocata. Gestire queste operazioni di coordinamento direttamente all'interno della classe principale GameImpl avrebbe violato il principio di responsabilità singola (SRP).

Soluzione Ho introdotto il componente DeckHandler. Questa entità incapsula interamente la logica di manipolazione delle due pile. Il resto del gioco non interagisce mai direttamente con il mazzo grezzo per pescare o rimescolare. Questo componente si fa carico automaticamente delle operazioni di routine (come il ripopolamento del mazzo quando esaurito), assicurando che il mazzo e gli scarti siano sempre sincronizzati.

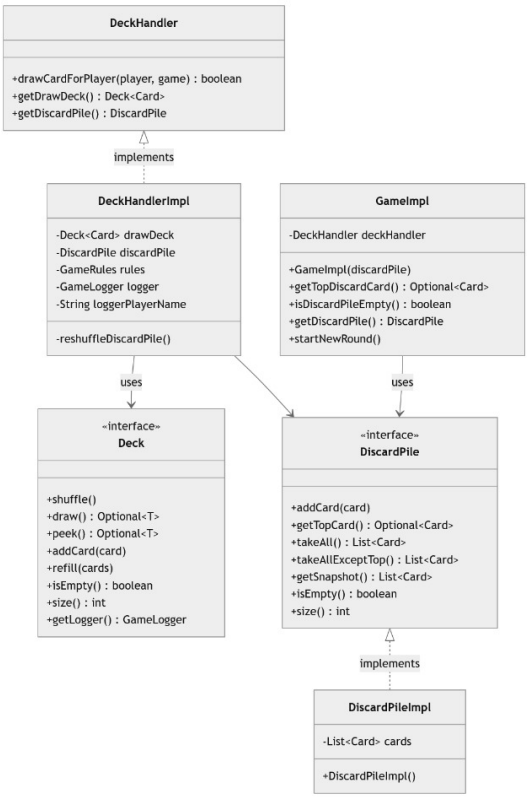

Figura 2.9: Schema UML della struttura della DiscardPile e il suo utilizzo nel DeckHandler.

2.2.3 Leonardo Perretta

Gerarchia dei giocatori e Logica decisionale

Problema Il gioco prevede la presenza di due macrotipologie di attori: giocatori umani (HumanPlayer) e intelligenze artificiali (AIPlayer). Sebbene il modo in cui scelgono la mossa sia radicalmente diverso (input UI per l'umano, calcolo algoritmico per l'AI), condividono molte responsabilità comuni: possedere una mano di carte, avere un nome e gestire le interazioni base con il modello (es. pescare una carta). Duplicare queste funzionalità comuni in due classi distinte avrebbe violato il principio DRY (Don't Repeat Yourself) e reso difficile la manutenzione.

Soluzione Ho utilizzato il Template Method Pattern strutturando una gerarchia di classi. È stata definita una classe astratta AbstractPlayer che

implementa l'interfaccia base Player. Questa classe fornisce l'implementazione concreta per i metodi comuni come la gestione della mano, l'aggiunta/rimozione di carte e il controllo del numero di carte rimanenti. Le sottoclassi concrete (HumanPlayer, AbstractAIPlayer) devono implementare solo i metodi astratti che definiscono la "strategia di input", lasciando alla classe padre la gestione dello stato condiviso.
Inoltre, lo stesso pattern si ripete per le AI: AbstractAIPlayer definisce lo scheletro del turno di un bot, ma delega alle sottoclassi specifiche (AIClassic, AIFlip, AIAllWild) l'implementazione dell'algoritmo di valutazione della mossa migliore, che varia in base alle regole della modalità attiva.

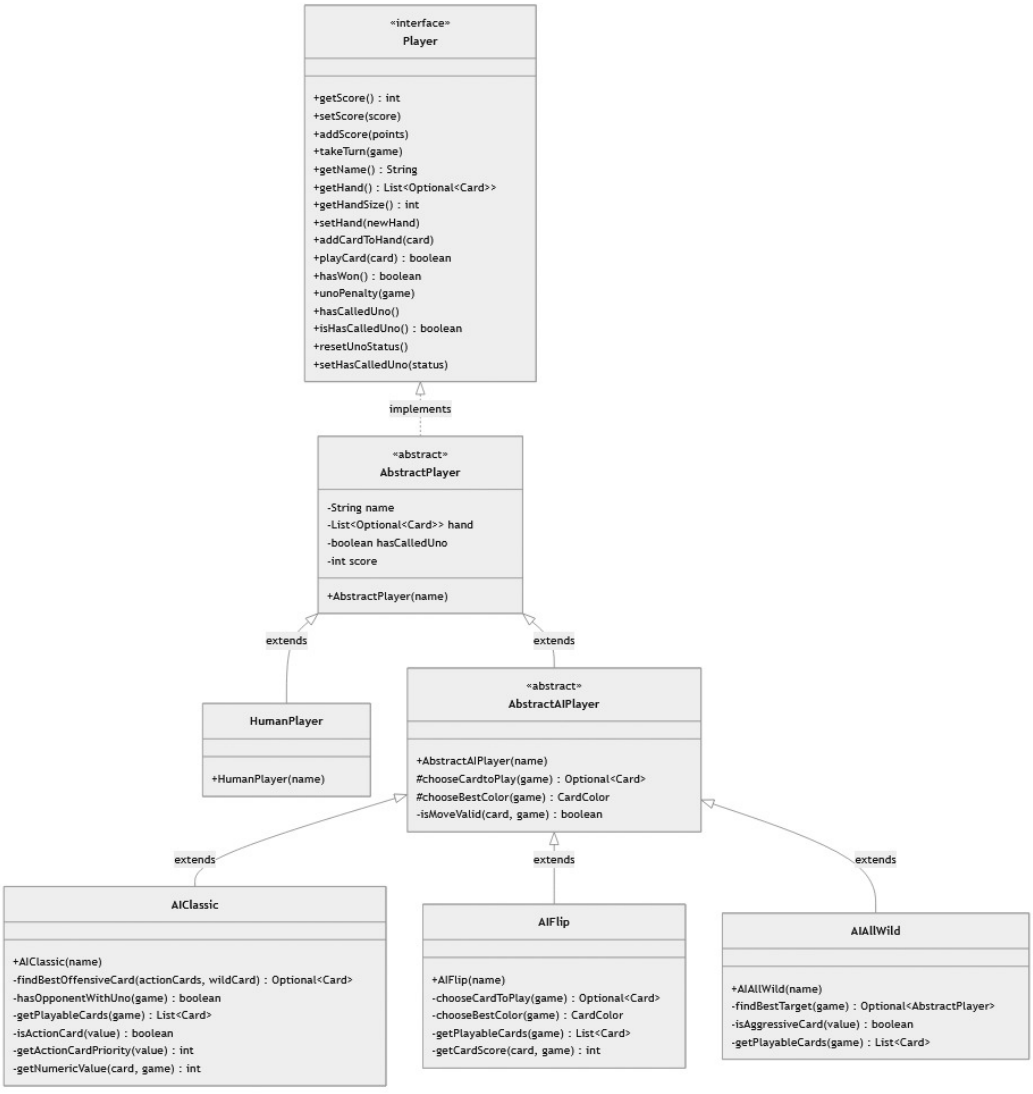

Figura 2.10: Schema UML della struttura dei Player.

Creazione della partita

Problema La costruzione di un'istanza di Game è un'operazione che varia strutturalmente in base alla modalità di gioco selezionata dall'utente. Ad esempio, la modalità "Flip" richiede l'allocazione di un FlipDeck e logiche specifiche, mentre la modalità "All Wild" richiede un AllWildDeck. Lasciare che sia il Controller a decidere quale classe concreta istanziare tramite lunghi blocchi condizionali (switch o if-else) creerebbe un forte accoppiamento tra la logica di controllo e le implementazioni specifiche del modello.

Soluzione Ho implementato il Factory Pattern tramite la classe GameFactoryImpl. Questa classe centralizza l'intera logica di creazione e assemblaggio della partita. Questa classe si occupa di creare deck, discard pile, turn manager, game e game setup in base alla modalità scelta.

Separazione della Logica di Inizializzazione

Problema La creazione di una nuova partita non si esaurisce con la semplice l'inizializzazione degli oggetti in memoria. Esiste una fase di avviamento che prevede una sequenza procedurale ben definita: mescolare il mazzo, distribuire esattamente 7 carte a ciascun giocatore, determinare la prima carta della pila degli scarti (gestendo eventuali casi limite in cui la prima carta è un Jolly o un +4) e impostare il primo giocatore attivo. Includere questa logica procedurale nel costruttore di GameImpl o mescolarla con la logica di creazione strutturale della GameFactory avrebbe violato l'SRP.

Soluzione Ho creato l'entità GameSetup (implementata da GameSetupImpl) per incapsulare esclusivamente la logica di preparazione dello stato iniziale. Questa classe non crea gli oggetti, ma li riceve in input: il metodo di inizializzazione accetta le istanze già costruite di Game, Deck, DiscardPile e la lista di AbstractPlayer. Una volta ricevute le dipendenze, GameSetup agisce come un operatore che manipola questi oggetti per portarli allo stato "pronto per giocare". Questo componente viene invocato direttamente dalla GameFactory subito dopo l'instanziazione delle entità, garantendo che l'oggetto Game restituito al controller sia non solo creato, ma anche correttamente inizializzato e valido.

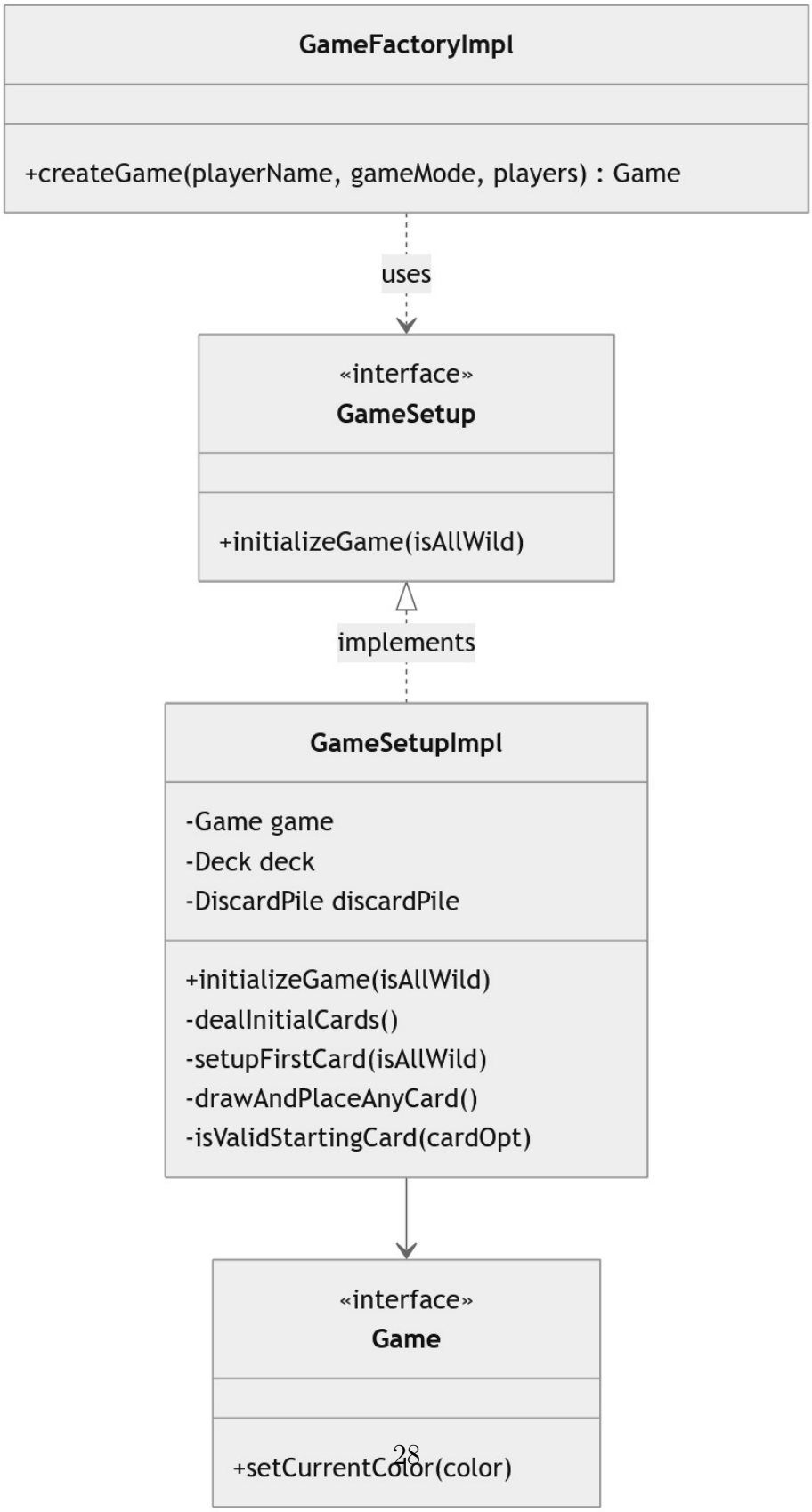

Figura 2.11: Schema UML del Setup del Game.

Personalizzazione dell'esperienza di gioco

Problema Oltre alla modalità di gioco principale (Classic, Flip, ecc.), l'utente deve poter abilitare una serie di "Game Rules" o regole opzionali (ad esempio: la possibilità di disabilitare il pulsante Uno!). Gestire queste micro-variazioni comportamentali passando decine di parametri booleani sparsi alle varie classi avrebbe reso le firme dei metodi ingestibili e il codice difficile da leggere.

Soluzione Ho utilizzato l'entità GameRules, che funge da contenitore centralizzato per tutte le opzioni di personalizzazione attive nella partita corrente. Durante la fase di setup nel menu, le scelte dell'utente vengono salvate in questa struttura sotto forma di flag booleani. Una volta avviata la partita, i componenti logici ricevono l'istanza di GameRules e ne interrogano lo stato per decidere se consentire o meno determinate azioni. Questo approccio disaccoppia la logica di controllo dalla configurazione: il motore di gioco rimane unico, ma il suo comportamento si adatta dinamicamente alle preferenze espresse dall'utente prima dell'inizio della partita.

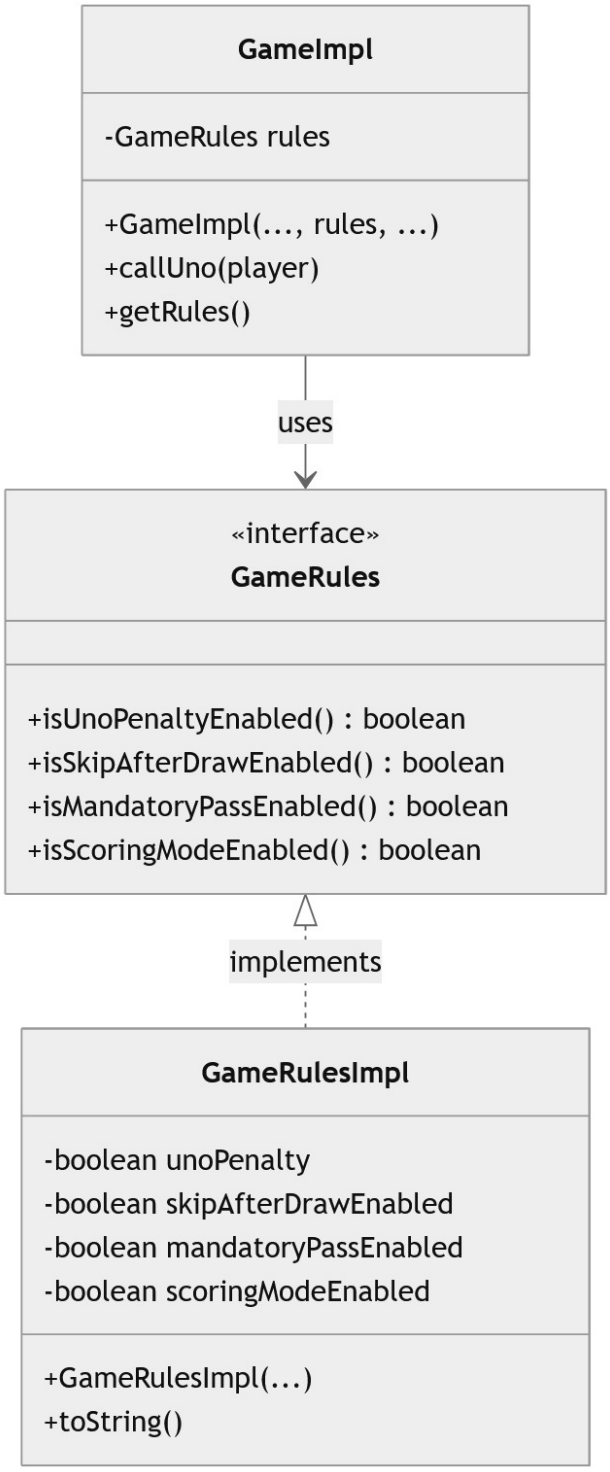

Figura 2.12: Schema UML della gestione delle regole.

Sistema di Logging

Problema Per scopi di debug e per salvare lo storico delle azioni, è necessario tracciare gli eventi di gioco (es. "Giocatore 1 ha pescato 2 carte"). Scrivere direttamente su System.out avrebbe creato un accoppiamento indesiderato tra la logica di gioco e l'output.

Soluzione È stata definita un'interfaccia GameLogger che astrae l'operazione di scrittura dei log. Il modello comunica con questa interfaccia generica. In fase di esecuzione, viene fornita un'implementazione (GameLoggerImpl) che dirige l'output verso un file.

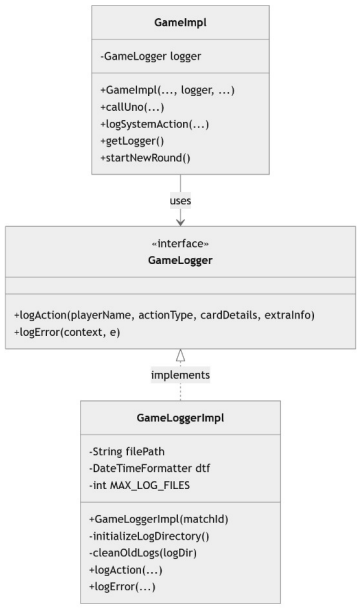

Figura 2.13: Schema UML del sistema di logging.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per la verifica della correttezza del codice e la prevenzione di regressioni, è stata utilizzata la libreria di JUnit che ci ha consentito di testare le parti critiche del progetto in maniera ottimale. Per quanto riguarda i comportamenti (**Behaviors**) delle carte, sono stati effettuati i seguenti test:

- **ActionBehaviorTest**: abbiamo è verificato il corretto funzionamento delle carte azione standard, controllando in particolare l'applicazione dell'effetto "Salto" (Skip) per bloccare il turno successivo e l'effetto "Inverti" (Reverse) per modificare la direzione di gioco.
- **AllWildBehaviorTest**: sono state testate le carte esclusive della modalità "All Wild", verificando la corretta esecuzione di meccaniche complesse come il "Salto Due" (WildSkipTwo), la "Pesca Mirata" (WildTargetedDrawTwo), lo "Scambio Forzato" (WildForcedSwap) e l'inversione Jolly (WildReverse).
- **BackSideBehaviorTest**: si è controllata la robustezza del sistema, verificando che l'invocazione di metodi sul lato passivo (non attivo) di una carta sollevi le eccezioni previste, prevenendo stati inconsistenti.
- **DrawBehaviorTest**: si è testata la logica delle penalità di pesca, verificando nello specifico la corretta applicazione dell'effetto "+2" e l'aggiornamento della mano del giocatore bersaglio.
- **FlipBehaviorTest**: si è verificato il comportamento della carta "Flip", accertandosi che la sua esecuzione inneschi correttamente la logica di inversione dei mazzi e delle mani.

- **NumericBehaviorTest**: si è validato il comportamento delle carte numeriche semplici, assicurandosi che vengano giocate senza scatenare effetti collaterali sul flusso della partita.

- **WildBehaviorTest**: si è verificata la gestione delle carte Jolly, testando sia il cambio colore standard (Wild), sia le varianti con penalità come il "+4" e il Wild Draw Color, controllando che il sistema registri correttamente il nuovo colore scelto.

Per quanto riguarda i tipi di carte:

- **DoubleSidedCardTest**: si è verificata la coerenza degli attributi di colore e valore sia per il lato chiaro (Light Side) che per quello scuro (Dark Side). È stata inoltre validata la corretta esecuzione dell'effetto associato e controllata la logica di compatibilità, assicurando che la carta venga accettata dal sistema solo in presenza di corrispondenze valide.

Per quanto riguarda la gestione dei mazzi:

- **AllWildDeckTest**, **FlipDeckTest**, **StandardDeckTest**: si è verificata la correttezza della dimensione iniziale del mazzo e il suo progressivo decremento a seguito delle pescate. Sono stati effettuati controlli sulla composizione per garantire la presenza delle carte previste dalla variante, oltre a testare l'efficacia delle operazioni di mescolamento (shuffle) e di rigenerazione del mazzo dagli scarti (refill).

Per quanto riguarda i componenti di gioco:

- **DeckHandlerTest** il test valida la corretta gestione del flusso delle carte, verificando in primo luogo il successo dell'operazione di pesca (draw) e la conseguente riduzione del mazzo. In secondo luogo, è stato testato il meccanismo di rigenerazione automatica (refill): simulando l'esaurimento del mazzo principale, si è controllato che il sistema sia in grado di ricostituirlo attingendo dalla pila degli scarti e rimescolandoli, garantendo la continuità della partita senza perdere l'ultima carta giocata (Top Card).

- **DiscardPileTest** si è verificata la coerenza dello stato iniziale della pila e la correttezza delle operazioni di inserimento, accertandosi che l'ultima carta scartata venga correttamente identificata come riferimento per il gioco. Abbiamo fatto test anche per il metodo takeAllExceptTop, fondamentale per la rigenerazione del mazzo: sono stati testati scenari con degli scarti per confermare il recupero completo della cronologia

e il caso in cui la pila contiene una sola carta, garantendo che questa rimanga ancorata al tavolo senza generare errori o svuotare la pila.

- **GameFactoryTest** si è validata la logica di creazione della partita, assicurandosi che il metodo restituisca un'istanza di Game correttamente configurata per ciascuna delle modalità supportate (Standard, Flip, All Wild).

- **GameRulesTest** si è validata la flessibilità del sistema di configurazione, verificando la corretta implementazione delle regole. Nello specifico, sono stati testati: il meccanismo di gestione della dichiarazione "UNO" (attivazione/disattivazione della regola), il riciclo del mazzo dalla pila degli scarti in caso di esaurimento (attivazione/disattivazione della regola), la regola di Force Pass che obbliga il giocatore a cedere il turno subito dopo una pescata (indipendentemente dalla giocabilità della carta estratta), e la selezione della condizione di vittoria, scegliendo tra la modalità a round (basata sull'accumulo di punti) e la partita secca.

- **GameSetupTest** si è validata la procedura di avvio della partita (setup), concentrandosi sulla corretta distribuzione delle mani iniziali ai giocatori e sul conseguente decremento del mazzo di pesca. Inoltre, è stata verificata la logica di selezione della prima carta della pila degli scarti, assicurandosi che il gioco inizi con uno stato valido e coerente per la prima mossa.

- **GameTest** si è verificata la coerenza dello stato globale della partita all'avvio e la corretta transizione tra i round. Sono stati testati i flussi critici: l'accettazione di una mossa valida con conseguente avanzamento del turno e il rigetto di una carta invalida, garantendo che lo stato del gioco rimanga inalterato in caso di errore. Inoltre, si è validata la logica di gestione delle azioni passive, coprendo sia la sequenza di pesca seguita dal passaggio del turno, sia il blocco dei tentativi di passare il turno senza aver prima soddisfatto i requisiti di pescata. Infine, è stata testata l'integrità della meccanica "Flip", controllando l'inversione sincronizzata di tutte le componenti di gioco.

- **MoveValidatorTest** si è verificata la robustezza della logica di controllo delle regole, accertandosi che il sistema approvi le giocate basate sulla corrispondenza di colore e valore. È stata testata l'accettazione incondizionata delle carte Jolly e, specularmente, la corretta applicazione dei vincoli di colore imposti da una Wildcard precedentemente giocata. Infine, si è validato il rifiuto di mosse illegali e l'efficacia del controllo sulla mano di un giocatore per determinare se ha carte giocabili.

- **ScoreManagerTest** si è verificata l'accuratezza del calcolo del punteggio, controllando la corretta attribuzione dei pesi specifici per ogni categoria di carta. Sono stati eseguiti test mirati sul calcolo delle carte numeriche, delle carte azione e delle carte Jolly, oltre a scenari con mani miste.

- **TurnManagerTest** si è verificata la coerenza dell'inizializzazione del ciclo di gioco e la gestione della direzionalità, testando l'avanzamento dei turni sia in senso orario che antiorario. Sono state validate le logiche di interruzione del flusso, controllando il corretto funzionamento del salto turno singolo e multiplo, nonché la combinazione di effetti complessi come inversione più salto. È stato inoltre verificato il ripristino dello stato del giocatore all'inizio del turno e la corretta applicazione della penalità in caso di mancata dichiarazione di "UNO".

Per quanto riguarda le AI:

- **AIAllWildTest** si è analizzata la logica decisionale dell'AI, focalizzandosi sull'utilizzo strategico della carta "Scambio Forzato". È stato verificato che l'algoritmo massimizzi il vantaggio, giocando la carta prioritariamente in situazioni di svantaggio numerico (mano con tante carte, contro avversario con poche carte) ed evitandola tassativamente quando si trova in prossimità della vittoria, prevenendo così scambi controproducenti.

- **AIClassicTest** si è analizzata la strategia base dell'AI, verificando la corretta gerarchia delle priorità di scarto. Nello specifico, è stato controllato che l'agente preferisca giocare carte Azione per ostacolare gli avversari e, in alternativa, carte numeriche di valore elevato per minimizzare il punteggio residuo in caso di sconfitta. Sono stati controllati l'utilizzo delle carte Jolly con conseguente scelta del colore e l'azione obbligata di pesca in assenza di mosse valide. Infine, si è accertata la corretta chiamata di "UNO" per evitare penalità.

- **AIFlipTest** si è analizzata la strategia dell'AI nella gestione dei due lati, verificando che venga assegnata la massima priorità alla carta "Flip" per forzare la transizione tra i lati del mazzo. È stato inoltre controllato l'uso efficace della carta "Wild Draw Color", assicurandosi che l'IA la giochi correttamente.

- **HumanTest**: si è verificata la corretta implementazione della passività del giocatore umano, accertandosi che l'invocazione del turno non inneschi alcuna logica decisionale automatica.

Per quanto riguarda il logger:

- **GameLoggerTest**: si è verificata la corretta gestione del ciclo di vita dei file di Log, validando le operazioni di Input/Output su disco. Nello specifico, è stata accertata l'effettiva creazione del file al momento dell'inizializzazione del logger e la sua corretta eliminazione.

3.2 Note di sviluppo

3.2.1 Luca Fagioli

Uso di Lambda Expressions
Usate in vari punti.
`https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417`
`src/main/java/uno/model/cards/behaviors/impl/ActionBehavior.java#`
`L41`

Uso di Optional
Usati in vari punti.
`https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417`
`src/main/java/uno/model/cards/behaviors/impl/ActionBehavior.java#`
`L33`

3.2.2 Luca Pasini

Uso di Lambda Expressions
Usate in vari punti.
`https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417`
`src/main/java/uno/model/cards/deck/impl/StandardDeck.java#L63`

Uso di Optional
Usati in vari punti.
`https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417`
`src/main/java/uno/model/game/impl/GameImpl.java#L212`

Uso di Generici
`https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417`
`src/main/java/uno/model/cards/deck/impl/AbstractDeckImpl.java#L44`

Uso di Stream
https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417
src/main/java/uno/controller/impl/GameControllerImpl.java#L131

Utilizzo di librerie esterne - Gson
https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417
src/main/java/uno/model/cards/deck/impl/FlipDeck.java#L60

3.2.3 Leonardo Perretta

Uso di Optional
Usati in vari punti.
https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417
src/main/java/uno/model/players/impl/AbstractAIPlayer.java#L35

Uso di Streams
https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417
src/main/java/uno/model/players/impl/AIAllWild.java#L108

Uso di Lambda Expressions Usate in vari punti.
https://github.com/LucaPaso04/OOP24-uno/blob/270f55b891bd13729e94c0781c132bb5417
src/main/java/uno/model/players/impl/AIAllWild.java#L109

Creazione e Scrittura su file https://github.com/LucaPaso04/OOP24-uno/
blob/270f55b891bd13729e94c0781c132bb5417c23b2/src/main/java/uno/
model/utils/impl/GameLoggerImpl.java#L47

Capitolo 4

Commenti finali

4.1 Autovalutazioni e lavori futuri

4.1.1 Luca Fagioli

L'obiettivo principale del mio lavoro è stato dare "vita" alle carte, trasformando dei semplici dati statici in oggetti capaci di agire e modificare il corso della partita. Sono particolarmente fiero di come ho modellato i comportamenti delle carte speciali: riuscire a far convivere meccaniche classiche con varianti più complesse, come quelle del mazzo All Wild, ha richiesto un grande sforzo di astrazione per mantenere il codice pulito e gestibile.
La difficoltà maggiore è risieduta nel garantire che il sistema di validazione delle mosse fosse infallibile. Ho dovuto prevedere ogni possibile interazione tra le carte per evitare che il gioco si trovasse in stati inconsistenti. Ritengo di aver consegnato un motore di regole solido e affidabile, su cui i miei compagni hanno potuto costruire l'interfaccia e la gestione della partita senza preoccuparsi della correttezza logica delle azioni.
Un futuro aggiornamento potrebbe riguardare l'espansione delle carte. Grazie alla modularità con cui ho progettato i comportamenti, sarebbe estremamente semplice introdurre nuove carte con effetti inediti o personalizzati, arricchendo l'esperienza di gioco senza dover intaccare le fondamenta del sistema di validazione.

4.1.2 Luca Pasini

Questa esperienza è stata molto formativa, mi ha costretto a confrontarmi con la necessità di mantenere un'architettura pulita nonostante la crescente complessità delle regole di gioco. Sono complessivamente soddisfatto del lavoro che ho svolto, soprattutto sulla gestione degli stati.

La difficoltà più grande è stata quella di coordinare il flusso continuo della partita, assicurando che l'avanzamento dei turni avvenisse senza errori. Ritengo di aver costruito una struttura logica solida, capace di mantenere sempre coerente lo stato del gioco a beneficio delle altre componenti.
Un futuro aggiornamento potrebbe includere l'aggiunta di nuove regole, grazie alla flessibilità con cui è stato progettato il modulo di gestione delle regole, sarebbe possibile integrare logiche addizionali (come penalità cumulative o condizioni di vittoria alternative).

4.1.3 Leonardo Perretta

Il mio contributo al progetto si è concentrato sulla definizione dei giocatori. È stata un'esperienza stimolante che mi ha portato a riflettere su come astrarre correttamente il concetto di "Player" per permettere al sistema di trattare in modo uniforme sia gli utenti umani che i bot, nascondendo al motore di gioco la complessità delle loro differenze interne.
La parte di cui vado più fiero è indubbiamente lo sviluppo delle Intelligenze Artificiali. Non volevo che i bot si limitassero a giocare carte casuali, ma che avessero una "personalità" strategica adatta alla modalità scelta. La sfida più grande è stata proprio quella di tradurre le intuizioni umane in algoritmi come programmare l'AI "All Wild" per essere aggressiva verso chi ha poche carte.
Ritengo di aver creato un sistema estensibile e divertente, dove i bot offrono un livello di sfida adeguato senza risultare frustranti. Un futuro aggiornamento potrebbe concentrarsi sull'introduzione di livelli di difficoltà selezionabili o addirittura di un sistema di apprendimento basilare, che permetta all'AI di adattare la propria strategia in base allo stile di gioco dell'avversario umano.

Appendice A

Guida utente

A.1 Uno! Guida utente

A.1.1 Introduzione e obiettivi di gioco

Benvenuto nel gioco di UNO!
L'obiettivo è semplice: essere il primo giocatore a scartare tutte le carte presenti nella propria mano. Per farlo, i giocatori devono scartare a turno una carta che corrisponda a quella in cima al mazzo di scarto per colore o valore, utilizzando strategicamente le carte speciali per ostacolare gli avversari.
Sono disponibili tre modalità:

- Classic Mode: il gioco Uno! con le sue carte classiche.
- Flip Mode: variante dinamica che introduce carte a doppia faccia (lato chiaro e lato scuro), stravolgendo le regole a ogni cambio di lato.
- All Wild Mode: versione frenetica dove il mazzo è composto interamente da carte "Wild" (Jolly), rendendo ogni mossa imprevedibile.

A.1.2 Avvio del gioco

Una volta avviato il gioco, si presenterà a schermo un menù iniziale con varie opzioni, in cui è possibile:

- Avviare una partita Classica.
- Avviare una partita Flip.
- Avviare una partita All Wild.

- Accedere alla schermata delle Regole

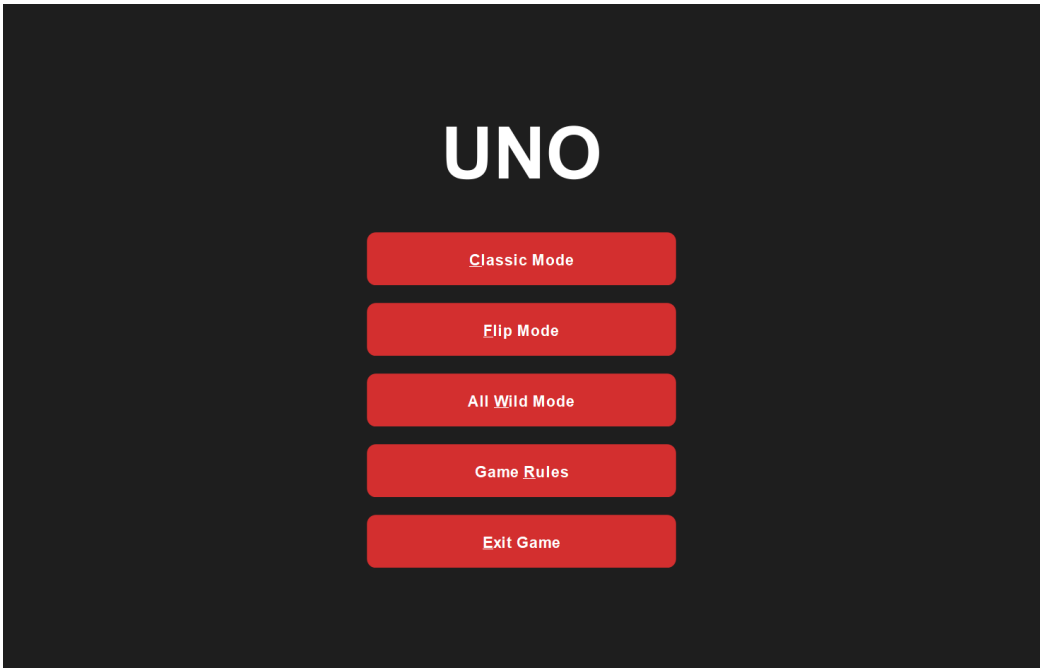

Figura A.1: Il menu iniziale di gioco.

A.1.3 Games rules

In questa schermata è possibile personalizzare le regole della prossima partita.

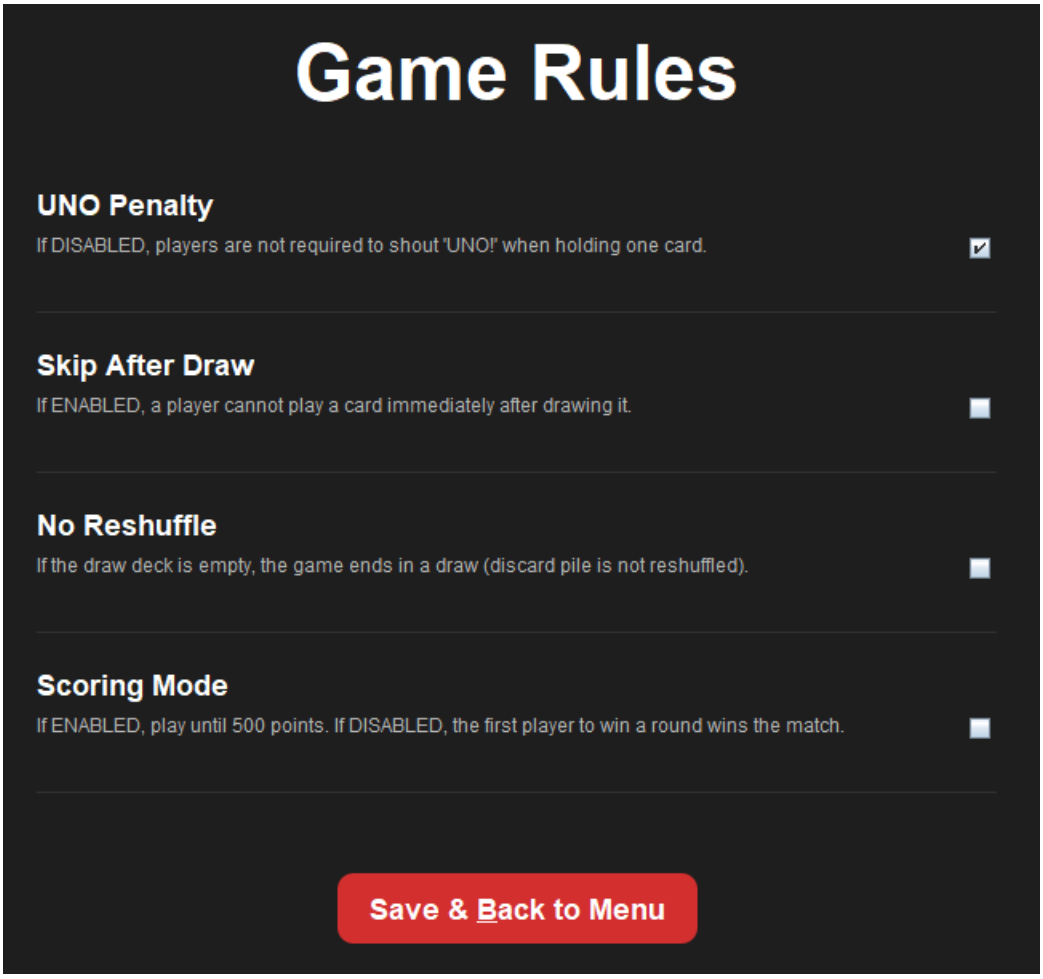

Figura A.2: La schermata delle regole.

A.1.4 Interfaccia grafica e schermate principali della partita

La partita, una volta avviata e dopo qualche mano, si presenterà in questo modo:

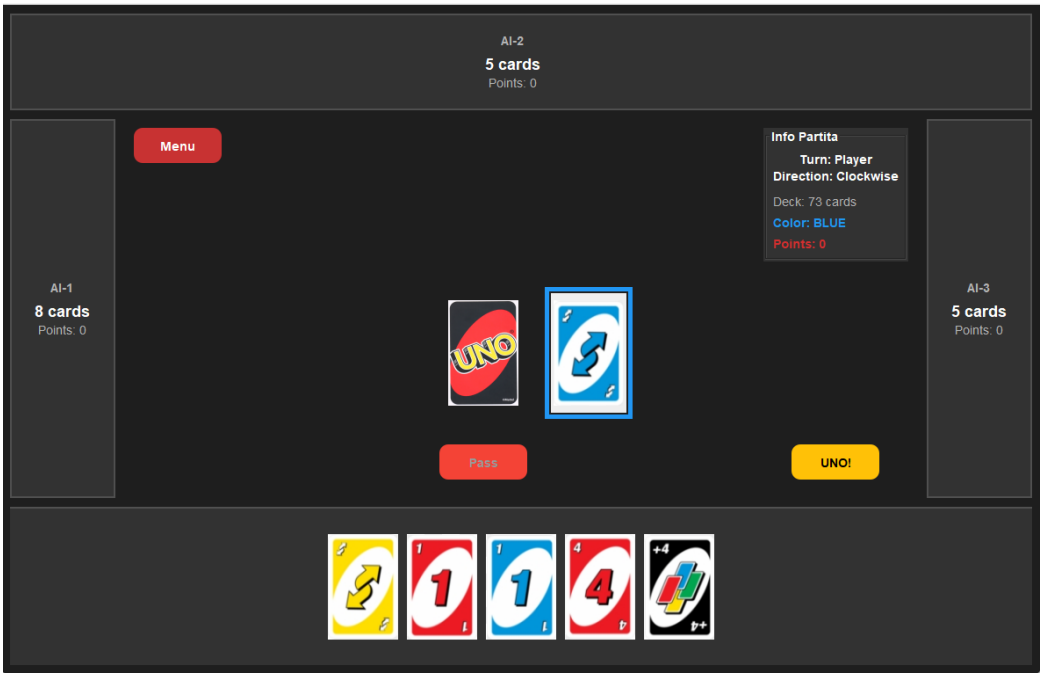

Figura A.3: L'interfaccia.

L'interfaccia grafica è suddivisa in aree:

- Pannello giocatore: mostra tutte le carte che il giocatore possiede nella sua mano.

Figura A.4: La mano del giocatore.

- Pannelli giocatori AI: sono 3 e mostrano il nome del bot, il numero delle carte presenti nella loro mano e infine gli eventuali punti nel caso la "scoring mode" fosse attiva. Il bordo dei pannelli diventerà giallo quando è il loro turno e rosso quando avranno rimasto una sola carta.

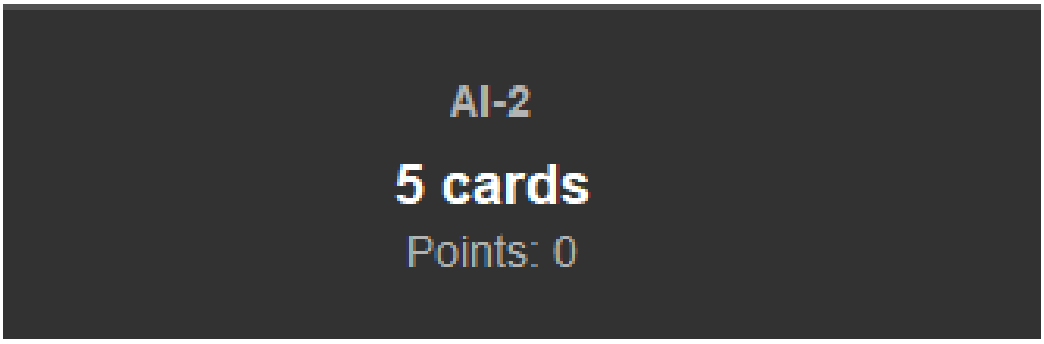

Figura A.5: Il panel dell'avversario.

- Deck e Pila degli scarti: Il deck (ovvero la carta girata di schiena) dovrà essere premuto per pescare una carta nel caso non si avesse nulla da giocare.

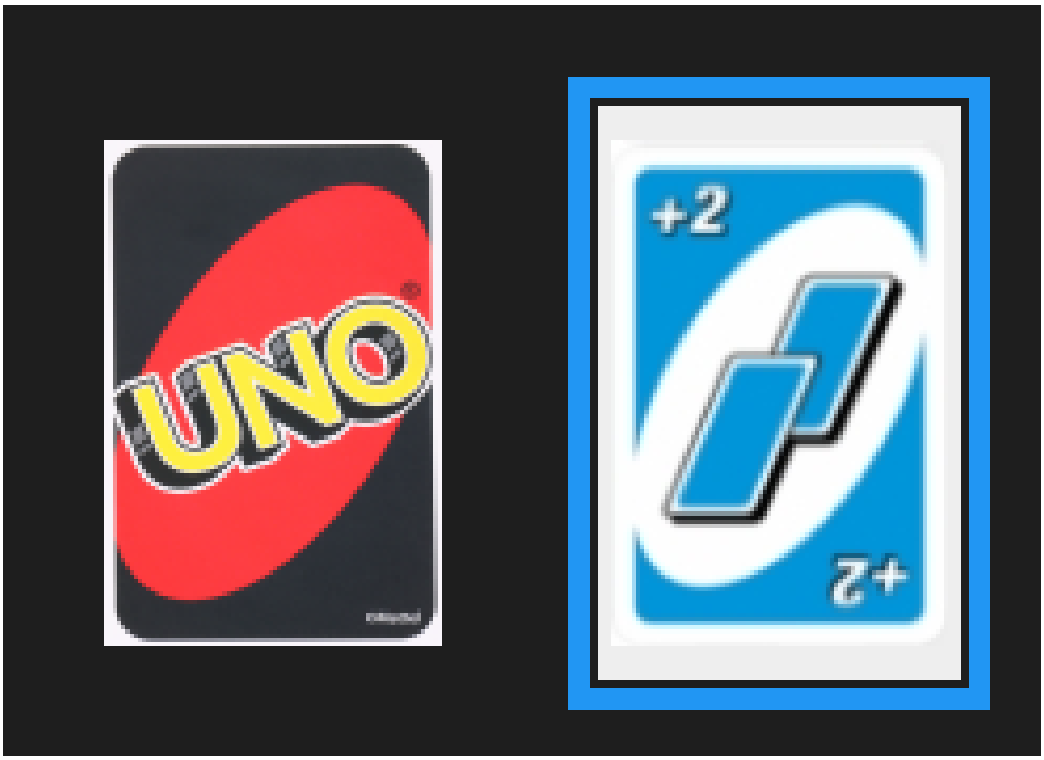

Figura A.6: Deck e Pila degli scarti.

- Pulsante "Pass": Se dopo aver pescato una carta non si ha in mano una carta giocabile, bisogna passare il turno.

- Pulsante "Uno!": Va premuto quando si rimane con solo una carta in mano durante il turno degli avversari, nel caso si arrivi al proprio turno successivo senza aver premuto Uno! allora verrà attivata una penalità.
- Pulsante "Menu": Pulsante per chiudere la partita e tornare al menu iniziale.
- Informazioni partita: indicano di chi è il turno, la direzione di gioco (orario o antiorario), le carte rimaste nel mazzo (Deck), il colore della carta da giocare e infine gli eventuali punti del giocatore umano nel caso la "scoring mode" fosse attiva.

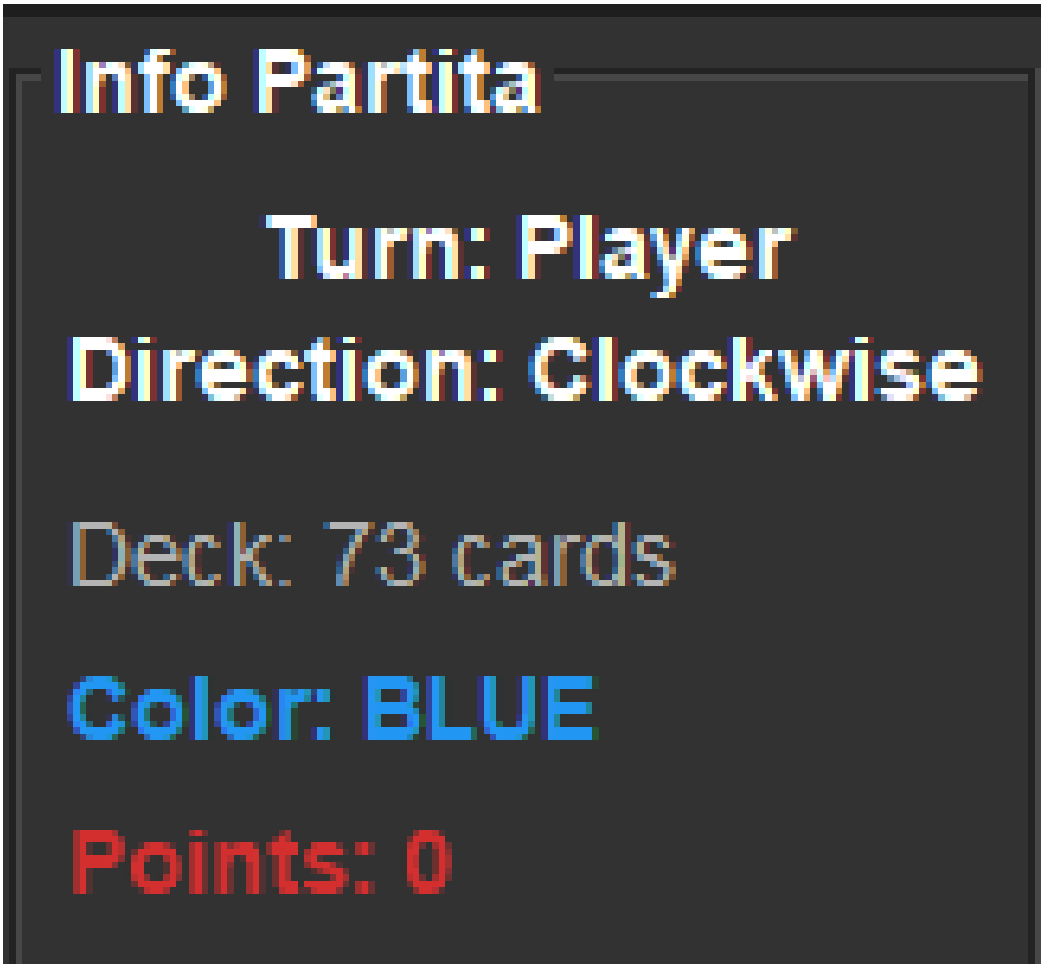

Figura A.7: Label con info partita.

- Scelta colore: nel caso dovessimo giocare una carta "+4" oppure un "cambia colore" si aprirà un pop-up che ti chiederà quale colore vuoi scegliere

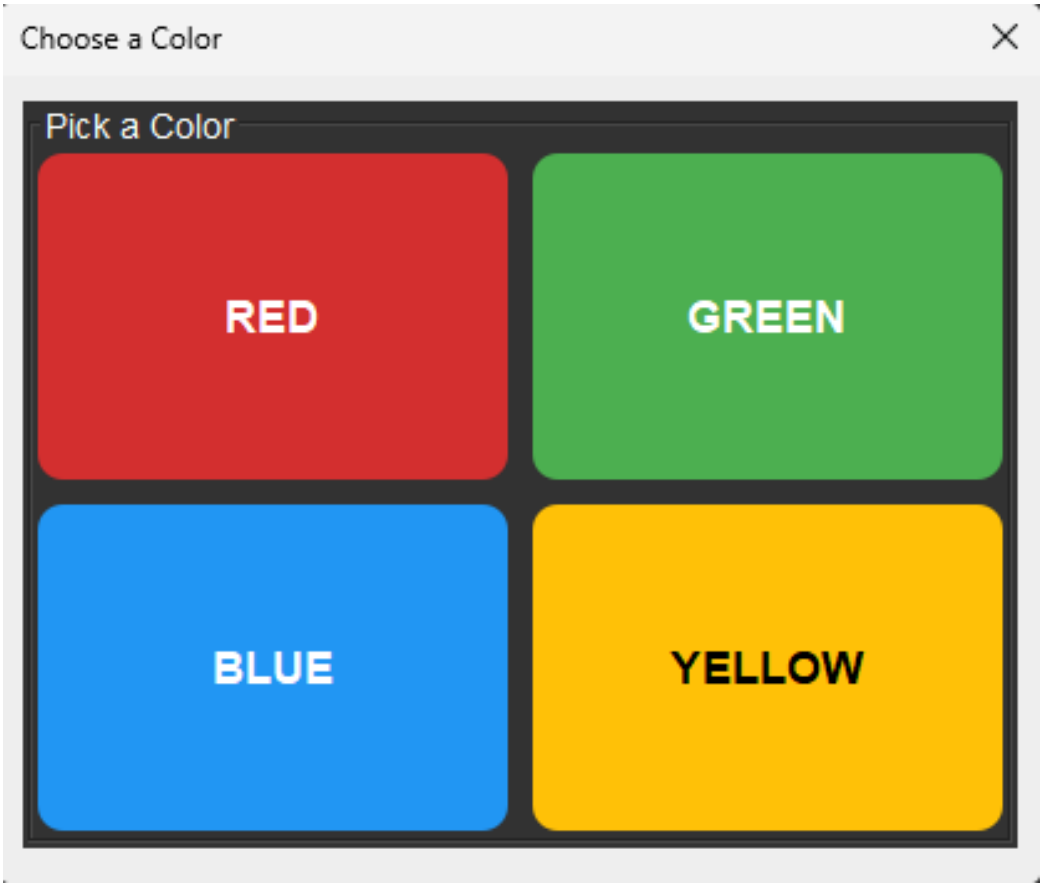

Figura A.8: Popup di scelta colore.

- Scelta giocatore: nel caso dovessimo usare una carta come "scambio mani" nella modalità "All wild", si aprirà un pop-up che ti chiederà di scegliere con quale bot vuoi effettuare lo scambio

Figura A.9: Popup di scelta giocatore.

Appendice B

Esercitazioni di laboratorio

...