1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

Relazione per il Progetto del Corso di Programmazione e Modellazione ad Oggetti

Diego Chiodi, Andrii Ursu, Edoardo Guardabascio

12/06/2025

1 ANALISI

1.1 Requisiti

UNO è un classico gioco di carte, qui realizzato in una versione digitale che consente a un singolo giocatore umano di sfidare un numero selezionabile di avversari controllati dal computer. L'obiettivo è terminare le proprie carte prima degli altri giocatori, rispettando le regole originali del gioco da tavolo. Il progetto si propone di implementare un sistema di gioco fedele a tali regole, gestito attraverso un'interfaccia grafica interattiva che accompagna l'utente durante tutta la partita.

All'avvio dell'applicazione l'utente viene accolto da una schermata iniziale dove può:

- Inserire il proprio nome.
- Selezionare il numero di bot avversari (da 1 a un massimo predefinito).
- Scegliere la modalità di gioco: partita amichevole (senza punteggio) o partita a punti.

Una volta avviata la partita, l'utente viene reindirizzato nella schermata principale di gioco, dove si svolge il turno a rotazione tra i giocatori, con la gestione delle carte, dei mazzi (coperto e scarti), del turno e delle regole speciali (carte azione). L'interazione con il gioco avviene tramite pulsanti, click sulle carte e notifiche automatiche del sistema (es. obbligo di chiamare UNO).
L'applicazione include la gestione della logica del turno, delle carte speciali (Salta, Inverti, Pesca 2, Jolly, Jolly Pesca 4, Mischiatutto), e delle penalità (es. dimenticanza di premere il bottone UNO).

1.2 Lista dei Requisiti

Requisiti Funzionali

- L'utente deve poter inserire il proprio nome nella schermata iniziale.
- L'utente deve poter scegliere il numero di bot contro cui giocare.
- Il sistema deve permettere la selezione tra due modalità di gioco: amichevole o a punti.
- Il sistema deve distribuire le carte iniziali a ogni giocatore (umano e bot).
- Il sistema deve gestire il mazzo coperto e quello degli scarti.
- Deve essere possibile pescare una carta se il giocatore non può giocare.

- Le carte speciali devono avere gli effetti previsti:
    - Salta: salta il turno del giocatore successivo.
    - Cambia giro: inverte il senso del gioco.
    - Pesca 2: costringe il prossimo giocatore a pescare due carte.
    - Jolly: permette di cambiare il colore di gioco.
    - Jolly pesca 4: cambia colore e costringe il prossimo giocatore a pescare 4 carte.
    - Mischiatutto: rimescola tutte le carte dei giocatori (funzionalità opzionale avanzata).

- Il bottone UNO deve essere premuto prima di giocare l'ultima carta: in caso contrario, il giocatore viene penalizzato (pesca due carte).

- Il sistema deve gestire il flusso dei turni tra giocatori, tenendo conto dei cambi di direzione e delle penalità.

- I bot devono agire automaticamente durante il loro turno.

- Il gioco termina quando un giocatore finisce le carte in mano (modalità amichevole) o quando si raggiunge un punteggio prefissato (modalità a punti).

Requisiti Non Funzionali

- Il sistema deve fornire un'interfaccia grafica intuitiva e responsiva.

- I bot devono simulare un comportamento pseudo-intelligente (scelta della carta più vantaggiosa).

- Il gioco deve fornire feedback visivi e messaggi di stato chiari all'utente.

1.3 Modello del Dominio

Il modello del dominio è stato progettato per rappresentare in modo astratto e modulare i principali concetti di gioco. Le entità sono modellate in classi e interfacce che rappresentano concetti fondamentali del dominio, con responsabilità ben definite all'interno dell'architettura del sistema.

Le principali entità coinvolte nel modello sono:

- `Game`: rappresenta la logica principale della partita, gestisce i giocatori, i mazzi, lo stato attuale del gioco, e le regole.
- `Player` (astratta): entità base per ogni giocatore. Ha un nome, una mano di carte e metodi per giocare o pescare.
- `Card`: rappresenta una carta generica.
    - `SpecialCard`: estende `Card`, rappresenta le carte con effetti speciali.
- `IDeck`: interfaccia per gestire il mazzo principale.
- `ITurnManager`: interfaccia per gestire l'ordine dei turni, i cambi di direzione e il giocatore corrente.
- `IGameListener`: interfaccia per notificare la view di eventi rilevanti (giocata, cambio turno, fine partita).

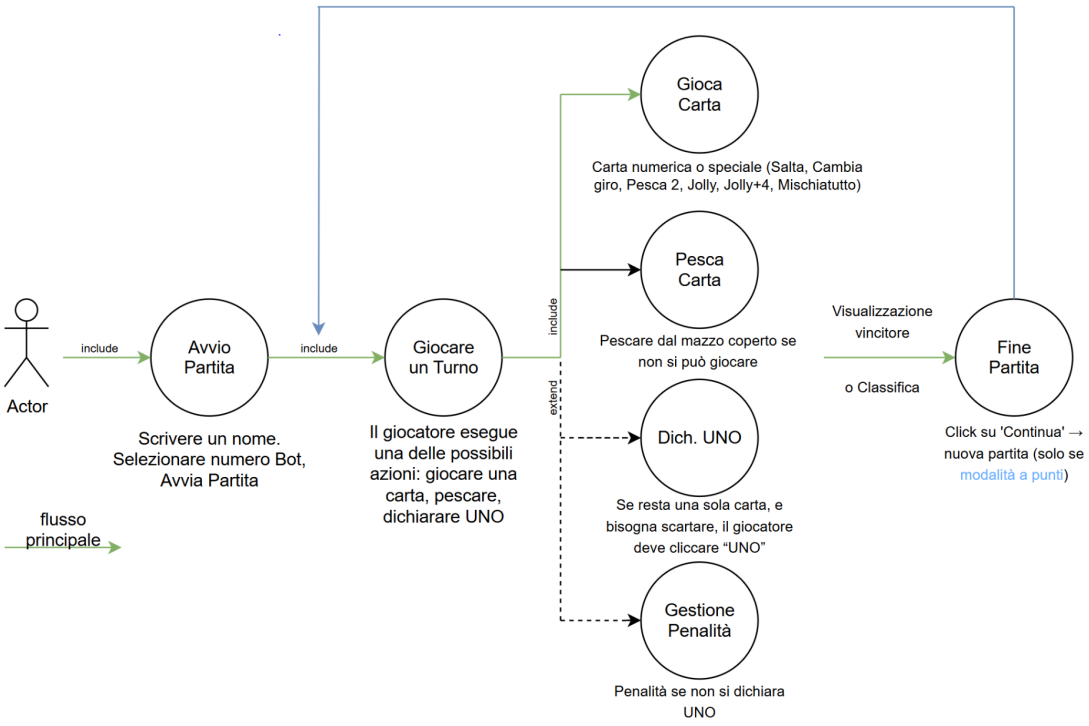

Figura 1: Diagramma d'uso

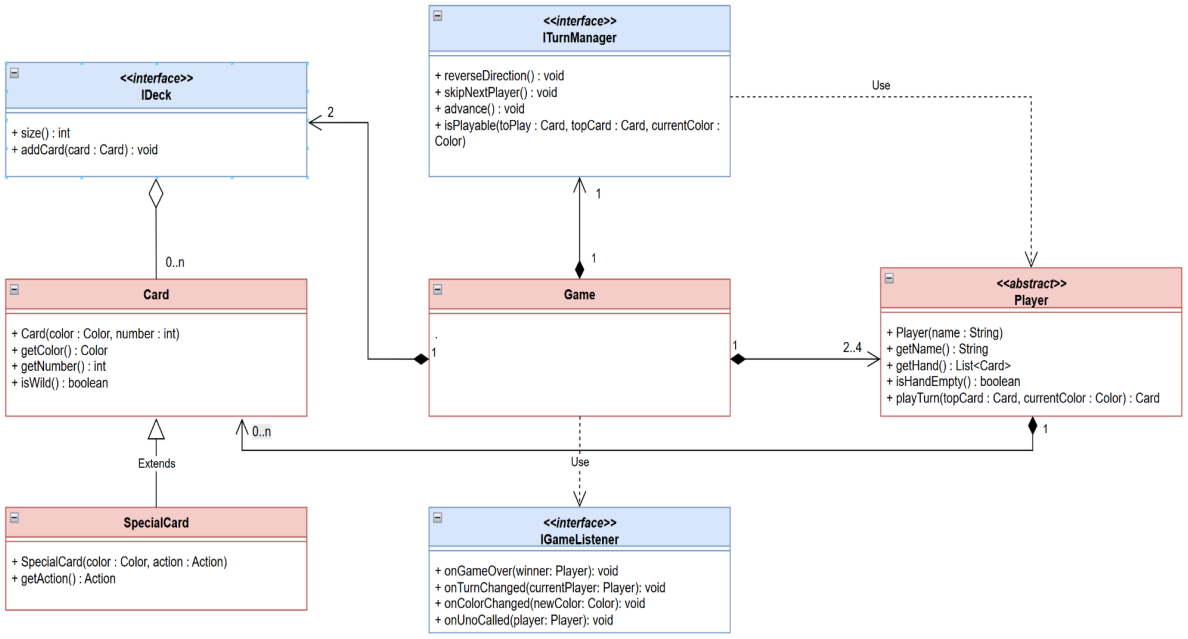

Figura 2: Modello del dominio

Figura 1: La partita ha inizio quando il giocatore inserisce il proprio nome, seleziona il numero di avversari controllati dal computer (bot) e sceglie la modalità di gioco: partita amichevole o partita a punti.

Una volta avviata la partita, il giocatore prende parte al gioco seguendo il normale flusso dei turni. Durante il proprio turno, il giocatore può scartare una carta, pescare dal mazzo, dichiarare “UNO” oppure penalizzato.

Al termine della partita, il comportamento del sistema dipende dalla modalità scelta:

- Se la partita è a punti, il punteggio viene aggiornato e salvato in classifica. Dopo la schermata finale, l’utente può scegliere di continuare con una nuova partita mantenendo i punteggi.

- Se la partita è amichevole, il gioco si conclude con la schermata di fine partita, senza aggiornare la classifica né proseguire automaticamente.

Figura 2: mostra il modello del dominio del sistema, realizzato utilizzando le principali interfacce concettuali come : `IDeck`, `ITurnManager`, `IGameListener` e una macro entità centrale `Game`, che rappresenta il contenitore logico del funzionamento del gioco.

Il diagramma evidenzia le relazioni fondamentali tra le entità:

- `Game` è il fulcro del sistema e coordina gli altri componenti principali.

- `ITurnManager` è in aggregazione con `Player`, poiché ne gestisce l’ordine di gioco senza possederli direttamente.

- `IDeck` è associato a `Card`, indicando che i mazzi contengono carte, ma le carte possono esistere anche indipendentemente dal mazzo.

- `GameListener` consente l’osservazione degli eventi di gioco, permettendo di disaccoppiare la logica del gioco dall’interfaccia utente o da altri componenti reattivi.

- `Player` è in composizione con `Card`.

- Le relazioni sono annotate con **cardinalità**, dove necessario, per indicare il numero di elementi coinvolti (es. un `Deck` contiene molte `Card`).

2 DESIGN

2.1 Architettura

Il progetto adotta una struttura basata sul pattern **Model-View-Controller (MVC)**, con una chiara separazione tra:

- **Model**: contiene la logica di gioco.
- **View**: è l'interfaccia grafica costruita in JavaFX con supporto FXML.
- **Controller**: gestisce gli eventi della GUI e li traduce in azioni nel Model.

Model

Rappresenta il cuore del gioco e comprende:

- Entità di gioco: `Card`, `SpecialCard`, `Color`, `Action`.
- Giocatori: `Player`(abstract), `HumanPlayer`, `BotPlayer`.
- Gestione partita: `Game`, `TurnManager`, `CoveredDeck`, `PlayedDeck`.
- Notifiche verso la view: interfaccia `IGameListener`.

Il `Game` centralizza la logica e si occupa di:

- Validare le mosse.
- Applicare effetti delle carte.
- Avanzare i turni.
- Notificare gli eventi ai listener registrati.

View

La view è realizzata con FXML e JavaFX. Comprende:

- Schermata iniziale (***`start.fxml`***).
- Schermata di gioco (***`game.fxml`***).
- Immagini delle carte, bottoni, messaggi, popup.

La View non contiene logica di gioco: riceve notifiche dal **Controller** e si limita a presentare lo stato.

Controller

I controller collegano **View** e **Model**:

- `StartController`: raccoglie nome utente e configura la partita iniziale.
- `GameController`: gestisce la partita vera e propria.

Il `GameController`:

- Riceve gli input dell'utente (giocata, pesca, UNO).
- Chiama i metodi nel Game.
- Si registra come listener del Game per ricevere notifiche sugli eventi.
- Aggiorna la GUI di conseguenza.

Comunicazione tra componenti

Da → A	**Modalità**	**Esempio**
View → Controller	Eventi utente	Click su una carta → `playTurn(...)`
Controller → Model	Chiamate dirette	`game.drawCardFor(player)`
Model → Controller	Eventi / listener	Cambio turno → `onTurnChanged(...)`
Controller → View	Aggiornamento GUI	Mostra messaggio, aggiorna mani, popup

Grazie a questa struttura, la GUI può essere sostituita o rifatta senza cambiare la logica del gioco o il controller.

2.2 Design Dettagliato

2.2.1 Design del Model

Gestione dei turni e notifiche eventi

1. Descrizione del problema

Nel gioco UNO, ogni turno coinvolge l'alternanza ordinata dei giocatori secondo una direzione di gioco (oraria o antioraria). Alcune carte speciali, come "Salta", "Inverti" o "Pesca 2", modificano questa rotazione. Inoltre, il sistema deve essere in grado di notificare eventi importanti alla GUI, come il cambio di turno o la fine della partita, senza che la logica di gioco sia accoppiata all'interfaccia grafica.

2. Soluzione proposta

Per separare le responsabilità tra gestione dei turni e logica del gioco, è stato introdotto un componente dedicato: la classe `TurnManager`. Questa classe implementa l'interfaccia `ITurnManager`, usata dal `Game` per accedere alle operazioni di turno in modo astratto. In questo modo, `Game` può concentrarsi sulla logica delle regole (carte speciali, vincita, punteggio) e delegare la rotazione dei giocatori.

Parallelamente, è stata progettata l'interfaccia `IGameListener`, implementata dai controller (GUI), che permette al `Game` di inviare notifiche di evento (turno cambiato, colore cambiato, vittoria, ecc.) senza dipendere direttamente dall'interfaccia utente. In questo modo, è possibile sostituire o modificare la GUI senza intaccare il core logico del gioco.

Alternative valutate:

- Gestione dei turni all'interno della classe `Game`: scartata perché avrebbe accoppiato troppo la logica del turno alla logica delle regole.
- Usare una classe astratta anziché un'interfaccia per `ITurnManager`: scartata per ridotta flessibilità.

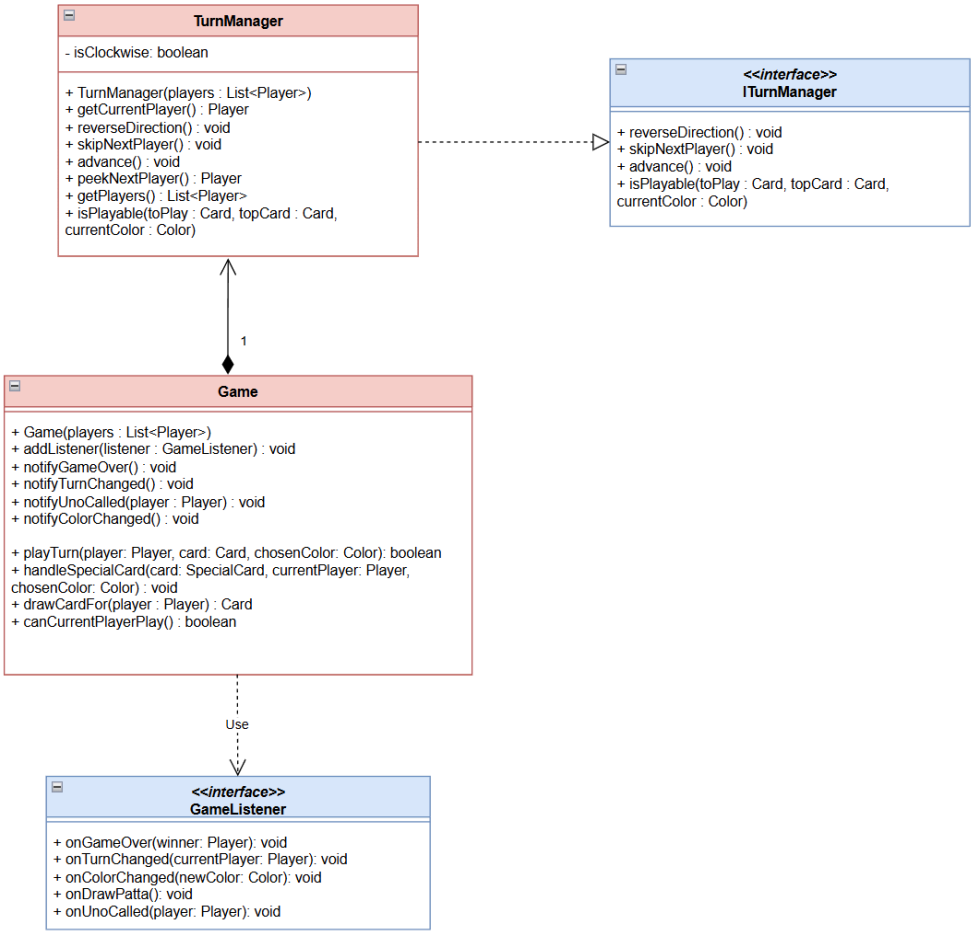

Diagramma UML

3. Pattern utilizzati

- Observer pattern: `Game` notifica gli eventi di gioco alla GUI tramite l'interfaccia `GameListener`. Questo garantisce un collegamento flessibile e disaccoppiato tra Model e View.

- Strategy pattern: `Game` dipende dall'astrazione `ITurnManager` e non dalla classe concreta `TurnManager`, permettendo di sostituire facilmente la logica dei turni, ad esempio con varianti personalizzate (a squadre, random, ecc.).

Gestione delle Regole e Carte Speciali

1. Descrizione del problema

Nel gioco UNO, le carte speciali (es. +2, Inverti, Salta, Jolly, Jolly +4, Mischiatutto) devono produrre effetti diversi sul turno corrente o sui giocatori successivi.
Serve quindi un modulo dedicato alla verifica delle mosse valide e all'esecuzione degli effetti delle carte speciali, mantenendo separata questa logica dal controller e dall'interfaccia.

2. Soluzione proposta

Tutta la logica delle regole è stata centralizzata nella classe `Game`, in particolare nei metodi:

- `handleSpecialCard(SpecialCard, Player)`.
- `playGame()`, dove viene chiamato `TurnManager.isPlayable(...)`.

La validazione di una mossa è implementata in `TurnManager.isPlayable(...)`, che verifica se una determinata carta può essere giocata sullo stato attuale (carta in cima al mazzo + colore corrente).

Alternative considerate:

- Inserire la logica di validazione dentro `Player.playTurn()`: scartata perché viola la separazione delle responsabilità.
- Usare metodi polimorfici su ogni `Card`: ritenuta eccessiva per il numero limitato di regole.

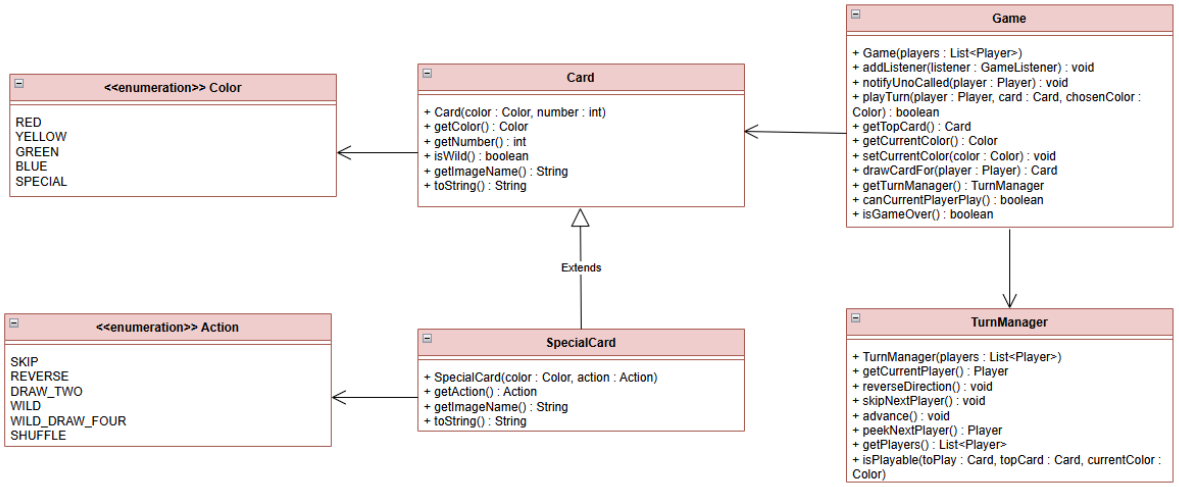

Diagramma UML

Logica dei Bot

1. Descrizione del problema

Nel gioco UNO, ogni giocatore deve scegliere una carta dalla propria mano che sia giocabile rispetto alla carta in cima al mazzo degli scarti, rispettando le regole del gioco (colore, numero, effetto). Per i giocatori umani, questa decisione viene demandata all'interfaccia utente.
Nel caso di un giocatore controllato dal computer (bot), è necessario implementare una logica automatica che:

- Valuti le carte disponibili nella mano.
- Selezioni una mossa legale.
- Notifichi la decisione al sistema di gioco, in modo che il turno possa proseguire.

Questo comportamento deve essere il più possibile flessibile, comprensibile e testabile.

2. Soluzione proposta

Per gestire il comportamento dei bot, è stata implementata la sottoclasse `BotPlayer`, che estende la classe astratta `Player`.

Il metodo centrale è `playTurn(Card topCard, Color currentColor)`, che rappresenta il turno del bot. La logica seguita è la seguente:

- Il bot scorre tutte le carte nella propria mano.
- Utilizza `TurnManager.isPlayable(card, topCard, currentColor)` per verificare se una carta è giocabile.
- Se trova una carta giocabile, e questa **non è un jolly +4**, la gioca e la rimuove dalla mano.
- In caso contrario, pesca una carta (restituisce `null` al controller, che gestisce la pesca).

Per gestire il comportamento del jolly +4, è stata introdotta una condizione che ne vieta l'uso se il bot ha altre carte compatibili con lo stato attuale del gioco.

Nel caso venga giocato un jolly (normale o +4), il colore da dichiarare viene selezionato tramite il metodo `chooseColor()`.
Questo metodo esegue una scansione della mano del bot e individua il colore più presente tra le sue carte, scegliendolo come colore preferenziale. In questo modo, il bot tende a scegliere il colore che gli permette di aumentare le probabilità di avere carte giocabili nei turni successivi.

- Esempio: Se la carta in cima al mazzo è un “9 Giallo” e il bot ha sia un “9 Rosso” (unica carta rossa) sia tre carte gialle, verrà giocata una carta gialla, privilegiando il colore più abbondante nella mano.

Alternative valutate:

- Una possibile alternativa sarebbe stata utilizzare una struttura di strategia esterna al bot stesso (ad esempio, applicando esplicitamente il *Strategy Pattern* per permettere a diversi bot di avere comportamenti diversi, ad esempio: aggressivo, difensivo, casuale).
  Tuttavia, dato che il progetto prevede un solo tipo di bot e la logica di selezione delle carte non cambia dinamicamente, si è scelto di implementare direttamente questa logica nella sottoclasse `BotPlayer`, mantenendo il codice semplice e più leggibile.

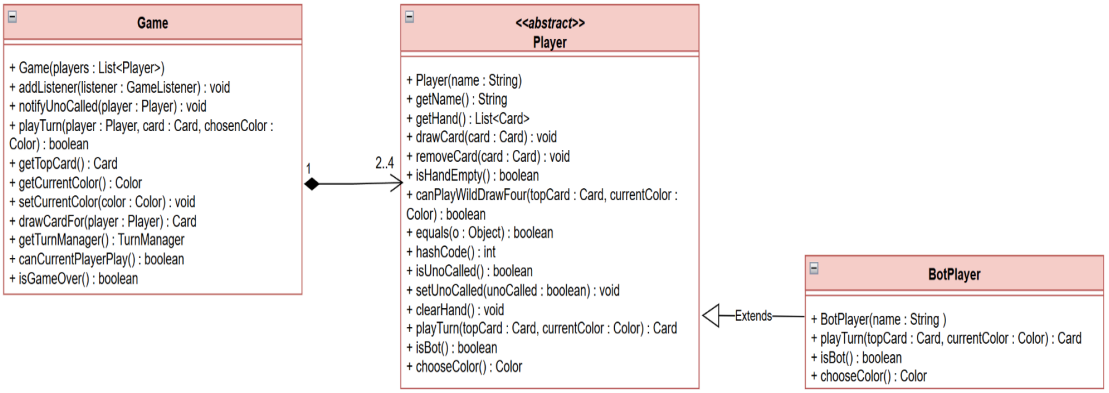

Diagramma UML

3. Pattern utilizzati

- **Template Method Pattern :**

  Il comportamento implementato è riconducibile al. In particolare, la classe astratta `Player` definisce il metodo astratto `playTurn(...)`, che funge da "template" per la gestione del turno di gioco. Le classi `HumanPlayer` e `BotPlayer`, che estendono `Player`, forniscono implementazioni concrete del metodo, ciascuna adatta al tipo specifico di giocatore.

  Questa scelta progettuale consente di avere una struttura comune e facilmente riutilizzabile, pur mantenendo una netta separazione tra la logica associata al giocatore umano e quella del bot. Inoltre, rende possibile estendere il sistema con nuovi tipi di giocatori in futuro senza dover modificare il codice esistente, favorendo così la manutenibilità e l’estensibilità dell’applicazione.

2.2.2 Design del Controller

Controller principale della partita

1. Descrizione del problema

L'interfaccia grafica del gioco UNO deve interagire in tempo reale con la logica di gioco, consentendo al giocatore umano di:

- Pescare carte
- Giocare carte valide
- Chiamare UNO
- Ricevere aggiornamenti visivi in base allo stato corrente del gioco

Al tempo stesso, l'applicazione deve gestire autonomamente i turni dei bot, applicare le regole e notificare all'utente eventuali eventi (vittoria, patta, cambio colore, ecc.), senza bloccare l'interfaccia utente o generare comportamenti inconsistenti.

2. Soluzione proposta

Per risolvere queste esigenze, è stato introdotto il componente `GameController`, che rappresenta il controller principale durante lo svolgimento della partita.

Questa classe collega direttamente:

- la GUI (**View**) definita in FXML → con la logica di gioco (**Model**).

`GameController` implementa l'interfaccia `IGameListener`, ricevendo notifiche automatiche dalla classe `Game`.

In questo modo, reagisce agli eventi di gioco senza richiedere polling o controlli ciclici inefficaci, applicando il pattern **Observer**.

L'interfaccia utente viene aggiornata con metodi dedicati (`updateUI()`, `updateHand()...`), innescati da eventi asincroni provenienti dal **Model**, mantenendo così la GUI reattiva e sincronizzata con lo stato del gioco.

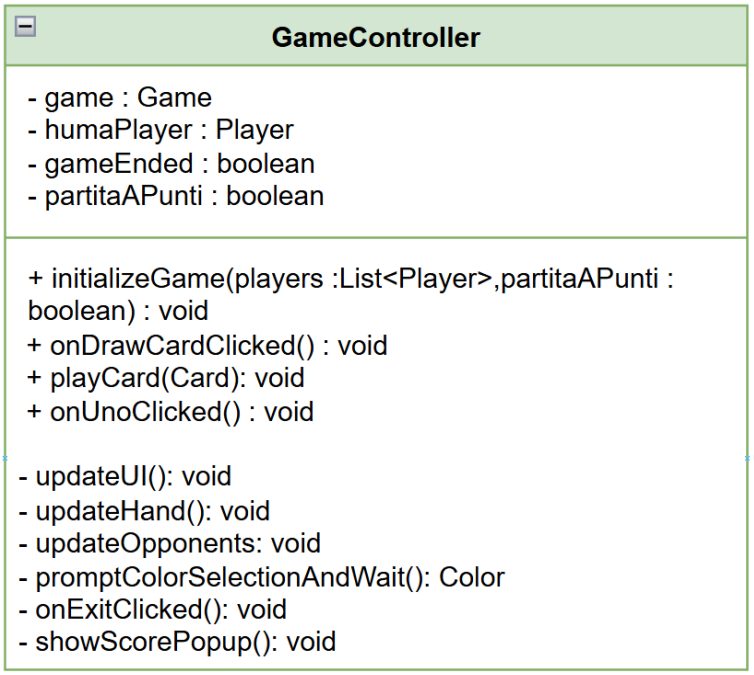

Diagramma UML

Controller di inizializzazione della partita

1. Descrizione del problema

All'avvio del gioco, l'utente deve poter configurare la partita selezionando:

- Il proprio nome.
- Il numero di avversari (bot).
- La modalità di gioco (partita a punti o amichevole).

Il sistema deve quindi inizializzare correttamente tutti i componenti della partita (giocatori, controller, scena di gioco) e avviare il `GameController` passando i dati raccolti. È importante che questa logica non sia contenuta nella **View** (FXML) né nel `GameController`, ma sia gestita in modo separato.

2. Soluzione proposta

È stata introdotta la classe `StartController`, collegata alla schermata FXML di avvio. Questa classe rappresenta il controller associato al punto di ingresso dell'applicazione.

Le sue responsabilità principali sono:

- Ricevere l'input dell'utente dalla GUI (nome, numero di bot, modalità).
- Creare dinamicamente la lista di giocatori (`HumanPlayer`, `BotPlayer`).
- Caricare il file FXML della schermata di gioco (***game.fxml***).
- Inizializzare il `GameController` passando i dati raccolti (tramite `initializeGame(...)`).

Grazie a questa separazione, la classe `GameController` non deve occuparsi di configurazioni iniziali, ma solo della gestione della partita.

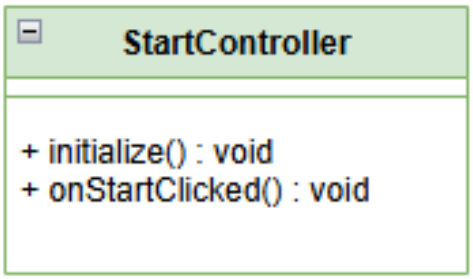

Diagramma UML

2.2.3 Design della View

La View si articola in due interfacce principali, realizzate in FXML con supporto JavaFX, e gestite rispettivamente da `StartController` e `GameController`.

start.fxml :

> Visualizzata all'avvio dell'applicazione, consente all'utente di configurare la partita, scegliendo il proprio nome, il numero di bot contro cui giocare e la modalità ("Amichevole" o "A punti"). Questi parametri vengono raccolti mediante componenti interattivi (es. `TextField`, `Spinner`, `RadioButton`) e trasmessi al **Controller** attraverso eventi onAction, rispettando il pattern MVC. In questa fase, la **View** si comporta come semplice contenitore di input, delegando la validazione e l'inizializzazione della partita alla logica del **Controller**.

start.fxml

game_view.fxml:

Caricata al momento dell'inizio della partita, mostra dinamicamente lo stato di gioco: la carta in cima al mazzo degli scarti, le carte del giocatore, i bot avversari, i messaggi testuali legati agli eventi di gioco (es. cambio colore, turno corrente) e due pulsanti principali: "UNO!" e "Pesca una carta". La disposizione è realizzata con un `BorderPane`, separando chiaramente le aree visive (top, center, bottom).I pulsanti e le altre componenti interattive sono collegati a metodi del **GameController** tramite eventi definiti in FXML (onAction). La logica che abilita o disabilita componenti come il bottone UNO è gestita nel **Controller**, ma segue uno stile di progettazione che mantiene separata la logica di visualizzazione da quella di controllo. Per esempio, la visualizzazione di un messaggio di penalità, o la comparsa di una schermata di vittoria (eventualmente accompagnata da una classifica in modalità "A punti"), viene innescata da callback su eventi che arrivano dal **Model**.

game_view.fxml

3 SVILUPPO

3.1 Testing automatizzato

Il progetto include una solida suite di test automatizzati realizzati con **JUnit**, focalizzati principalmente sulla logica del **Model**. I test hanno l'obiettivo di garantire il corretto funzionamento delle principali funzionalità del gioco e prevenire regressioni in caso di modifiche future.

Classi testate e copertura:

`BotPlayerTest`

- Verifica che il bot sia correttamente identificato come tale (`isBot()`), e che giochi una carta valida o peschi se non può giocare (`playTurn(...)`).

`CardTest` e `SpecialCardTest`

- Verificano il comportamento delle carte numeriche e speciali.
- Controllo su metodi come `getColor()`, `getNumber()`, `isWild()`, `getAction()`.

`CoveredDeckTest`

- Verifica la corretta riduzione del mazzo coperto quando si pesca, e il corretto comportamento del riciclo da `PlayedDeck` quando il mazzo è esaurito.
- Test del comportamento di riciclo da `PlayedDeck` quando il mazzo coperto finisce.

`PlayedDeckTest`

- Verifica l'aggiunta e il recupero della carta più recente.
  Testa il metodo `resetAndReturnAllExceptTop()` per il riciclo corretto delle carte scartate.
- Controllo dell'eccezione lanciata se si accede a una pila vuota.

`GameTest`

- Verifica che all'avvio della partita ogni giocatore riceva esattamente 7 carte.
- Controlla l'inizializzazione dei giocatori e delle mani (7 carte a testa), e la logica del metodo `canCurrentPlayerPlay()` in base alla carta in cima agli scarti.

`HumanPlayerTest`

- Verifica che un `HumanPlayer` non sia un bot (`isBot()`).
- Controlla l'eccezione generata da metodi non gestiti dal model (`playTurn()`).
  Testa aggiunta e rimozione di carte dalla mano.
- Controlla la gestione del flag `unoCalled`.

`TurnManagerTest`

- Controlla il corretto scorrimento dei turni (`advance()`), la rotazione (`reverseDirection()`) e il giocatore successivo (`peekNextPlayer()`).

3.1.1 Approccio ai test

- Isolamento: Ogni test è indipendente, testando una singola responsabilità.
- Comportamento previsto: I test controllano output attesi o effetti sullo stato interno degli oggetti.
- Eccezioni: I test includono casi in cui devono essere lanciate eccezioni (`@Test(expected = ...)`) per garantire la robustezza del codice.
- Stati limite: Alcuni test verificano condizioni limite, come la mano vuota, mazzo esaurito o carte speciali senza effetto.

3.2 Metodologia di Lavoro

Strumenti Utilizzati:

- Git: utilizzato per il versionamento e lo sviluppo cooperativo.
- Visual Studio Code, Eclipse: utilizzati come IDE.
- GitHub: utilizzato come repository principale.
- draw.io: utilizzato per la creazione e la modellazione dell'UML.
- Maven: utilizzato per la gestione delle dipendenze e la compilazione

3.2.1 Andrii Ursu

Attività svolte in autonomia

Mi sono occupato della progettazione e implementazione della **View** e dei **Controller**, seguendo l'architettura MVC con JavaFX.
Ho:

- Progettato e realizzato le schermate ***`start.fxml`*** e ***`game.fxml`***.
- Scritto le classi `StartController` e `GameController`, che si interfacciano con il modello tramite `IGameListener`.
- Gestito le azioni utente: selezione delle carte da giocare, bottone UNO, pesca, scelta colore.
- Aggiornato dinamicamente l'interfaccia in base allo stato del gioco.

Classi/FXML principali:

- `StartController`, `ITurnManager`
- `start.fxml, game.fxml`
- Eventi di input utente e aggiornamento grafico (pulsanti, animazioni, messaggi)

Attività svolte in collaborazione

Ho collaborato con gli altri membri del gruppo allo sviluppo del componente `TurnManager`, confrontandoci a partire dall'interfaccia `ITurnManager` per definire insieme le funzionalità fondamentali relative alla gestione dei turni (avanzamento, inversione, salto del giocatore). Inoltre, ci siamo coordinati per assicurare una corretta integrazione tra **Controller e Model**, lavorando insieme per far sì che gli eventi generati dall'interfaccia utente venissero tradotti in modo coerente in azioni sul modello di gioco, e viceversa.

3.2.2 Edoardo Guardabascio

Attività svolte in autonomia

Mi sono occupato della progettazione e implementazione del **Model**, che costituisce il nucleo logico dell'applicazione. In particolare, ho:

- Creato le classi principali del modello: `Player`, `HumanPlayer`, `Card`, `CoveredDeck`, `PlayedDeck`.
- Implementato la gestione delle carte speciali (`handleSpecialCard(...)`) e la validazione delle mosse (`TurnManager.isPlayable(...)`).
- Curato la logica di flusso: gestione dei mazzi, verifica condizioni di vittoria (modalità amichevole), pesca e gestione delle penalità.

Classi principali:

- `Game`, `Player`, `HumanPlayerCard`, `SpecialCard`, `Color`, `Action`, `CoveredDeck`, `PlayedDeck`

Attività svolte in collaborazione

Abbiamo collaborato attivamente per garantire una corretta integrazione tra **Model** e **View**. In particolare, ci siamo confrontati sulla progettazione e modellazione di classi fondamentali del dominio, come `Player`, `Card`, `SpecialCard` e `Game`, con l'obiettivo di rendere il modello facilmente collegabile alla GUI tramite meccanismi di osservazione e una chiara separazione delle responsabilità.
All'interno di questa fase, ho implementato le classi `PlayedDeck` e `CoveredDeck`, a partire dall'interfaccia `IDeck`, definita insieme al gruppo, per garantire una struttura coerente e flessibile nella gestione dei mazzi.
Questa collaborazione è stata fondamentale per mantenere una netta distinzione tra logica e interfaccia, in linea con il pattern architetturale MVC.

3.2.3 Diego Chiodi

Attività svolte in autonomia:

Mi sono occupato principalmente della fase iniziale di analisi e documentazione del progetto, contribuendo alla struttura della relazione e allo sviluppo della logica di gioco.

In particolare:

- Redazione delle sezioni 1.1 (Requisiti) e 1.2 (Lista dei Requisiti), con attenzione agli aspetti funzionali e non funzionali.
- Prima stesura del modello dei casi d'uso e supporto alla creazione degli schemi UML.
- Sviluppo della classe `BotPlayer`, con implementazione del metodo `playTurn()` e `chooseColor()` per permettere al bot di scegliere e giocare autonomamente una carta valida in base alla situazione corrente.
- Implementazione della modalità di gioco "a punti", comprensiva della verifica della condizione di vittoria (raggiungimento di 500 punti) e della gestione delle transizioni tra le partite.

Attività svolte in collaborazione:

Abbiamo collaborato all'integrazione del `StartController`, curando in particolare la logica per la selezione della modalità di gioco (partita amichevole o a punti) e la trasmissione corretta di questa informazione al `GameController`.
Inoltre, abbiamo lavorato congiuntamente alla definizione e implementazione della logica di calcolo del punteggio all'interno della classe `Game`, garantendo la corretta gestione della modalità a punti.
Infine, abbiamo progettato e realizzato il popup di fine round nel `GameController`, che mostra un riepilogo dei punteggi ottenuti e gestisce in modo dinamico la transizione verso il round successivo o la conclusione della partita.

3.3 Note di sviluppo

3.3.1 Andrii Ursu

Feature avanzate del linguaggio Java utilizzate:

- Uso combinato di FXML e controller binding per separare logicamente **View** e **Controller** in architettura MVC.

Algoritmi sviluppati (non forniti da libreria):

- Algoritmo per la rappresentazione dinamica dello stato del gioco, con aggiornamenti in tempo reale di (colore attuale, giocatore di turno, chiamata UNO, messaggi contestuali di gioco)

- Meccanismi di gestione interattiva: click sulle carte, selezione colore tramite popup modale, disabilitazione dinamica dei pulsanti in base allo stato.

Competenze acquisite:

- Padronanza nell'uso di JavaFX per la creazione di GUI reattive.

- Capacità di progettare Controller scalabili secondo il pattern MVC.

- Esperienza concreta nell'integrazione di GUI con modelli complessi tramite interfacce (`IGameListener`).

- Collaborazione con Git utilizzando GitHub Desktop per versionamento e integrazione continua del codice.

3.3.2 Edoardo Guardabascio

Algoritmi sviluppati (non forniti da libreria)

- Sviluppo dell'algoritmo di riciclo del mazzo (`resetAndReturnAllExceptTop()`), che consente di svuotare il mazzo degli scarti mantenendo solo la carta in cima, garantendo così la continuità della partita secondo le regole di UNO.

- Implementazione manuale della logica di fine partita e della penalità per la mancata chiamata di "UNO", non supportata da alcuna libreria esterna.

Competenze acquisite

- Approfondimento della programmazione funzionale in Java (stream, lambda).

- Esperienza concreta nella progettazione e gestione della logica di gioco in un contesto reale.

- Capacità di gestire lo stato e il flusso di gioco in maniera modulare e testabile.

- Utilizzo di Git e GitHub Desktop per il controllo di versione e la collaborazione in team.

3.3.3 Diego Chiodi

Algoritmi sviluppati (non forniti da libreria):

- Logica decisionale del bot per selezionare e giocare la carta più adeguata in base alla situazione di gioco
- Algoritmo per il calcolo dei punti a fine round e verifica della soglia di vittoria

Competenze acquisite:

- Consolidamento dell'uso di Git e GitHub per la collaborazione in team
- Approfondimento della progettazione software a partire dai requisiti funzionali
- Miglioramento nella scrittura chiara e precisa della documentazione tecnica
- Sviluppo incrementale e debugging collaborativo per la risoluzione di problemi logici complessi
- Consapevolezza dell'importanza dell'allineamento tra i membri del team per la gestione delle regole di gioco