



1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA - SCIENZA E TECNOLOGIA

UNIVERSITÀ DEGLI STUDI DI
URBINO CARLO BO
CORSI DI LAUREA IN
INFORMATICA APPLICATA

RELAZIONE PER IL PROGETTO DEL CORSO DI

PROGRAMMAZIONE E MODELLAZIONE A OGGETTI

Docente: Sara Montagna

Anno accademico ————— **2023/2024**

Autori

Diego Cecchini matricola: 323058 & Palumbo Giovanni matricola: 320180

SOMMARIO

2

- **1.ANALISI**
 - 1.1.Requisiti **pag.3**
 - 1.2.Dettaglio carte **pag.5**
 - 1.3.Dettaglio carte speciali **pag.7**
 - 1.4.Diagramma casi d'uso **pag.9**
 - 1.5.Modello del dominio **pag.10**
- **2.DESING**
 - 2.1.Architettura **pag.19**
 - 2.2.Design dettagliato **pag.21**
- **3.SVILUPPO**
 - 3.1.Metodologia di lavoro **pag.30**
 - 3.2.Testing automatizzato **pag.31**
 - 3.3.Note di sviluppo **pag.32**

1. ANALISI

3

1.1. Requisiti

In questa simulazione, del famosissimo gioco UNO, ci si potrà immergere nel gioco scegliendo tra due modalità:

- **1 contro 3.**
- **1 contro 1.**

La partita inizia dopo aver scelto la modalità desiderata, successivamente si distribuiscono le carte. Ogni giocatore riceverà **8 carte**.

Il mazzo rimanente viene posizionato coperto al centro del tavolo, formando il "pozzo" e, si posiziona scoperta al centro, una carta dal mazzo iniziando così la partita.

Svolgimento del Gioco

Il turno di ogni giocatore consiste nel giocare una carta dalla propria mano che corrisponda, per colore o per numero, alla carta in cima alla pila di carte scoperte. Se non si possiede una carta giocabile, è possibile pescare una carta dal mazzo, perdendo però il turno.

Oltre alle carte standard, questo gioco include anche carte speciali:

- **Salta un turno:** Il giocatore successivo perde il turno.
- **Cambio giro:** L'ordine di gioco viene invertito.
- **Pesca due:** Il giocatore successivo pesca due carte dal mazzo e perde il turno.
- **Cambia colore:** Può essere giocata al posto di qualsiasi altra carta, assumendone colore e numero.
- **Cambia colore +4:** Funziona come un Jolly, oltre a cambiare il colore costringe il giocatore successivo a pescare quattro carte e a perdere il turno.

Modalità 1 contro 3

In questa modalità, il giocatore si scontrerà contro **3 players-bot**. Il computer gestirà i turni dei tre giocatori virtuali, seguendo le stesse regole e strategie di un *giocatore umano*.

Modalità 1 contro 1

In questa modalità, l'obiettivo principale è quello di vincere un vero e proprio *duello* con il *players-bot*.

Regole Aggiuntive

- **Clicca "UNO!"**: Quando un giocatore rimane con **due carte**, deve Cliccare *il pulsante "UNO!"* prima di giocare la **penultima carta**. Se ciò non avviene si pescheranno automaticamente 2 carte dal mazzo centrale al turno successivo.
- **Cambio di Colore**: Quando viene giocata una carta Cambia colore, il giocatore può scegliere un nuovo colore per continuare il gioco.

Conclusione della Partita

Il primo giocatore che si libera di tutte le sue carte vince la partita.

1.2. Dettaglio carte

La grafica delle carte utilizzate è stata creata da noi tramite **Adobe Illustrator** e **Adobe Photoshop**.



Pesca due descritta in precedenza, essa può comparire in **4** colorazioni (Verde, Blu, Giallo e Rosso).



Cambia colore +4 descritta in precedenza, essa può comparire esclusivamente in questo colore a differenze delle altre carte speciali citate.



Salta un turno descritta in precedenza, essa può comparire in **4** colorazioni (Verde, Blu, Giallo e Rosso).



Cambia colore descritta in precedenza, essa può esclusivamente in questo colore come per la carta *Cambia colore +4*.



Cambia giro descritta in precedenza, come per la carta *Pesca due* e per la carta *Salta un turno* può comparire in **4** colorazioni differenti.

6

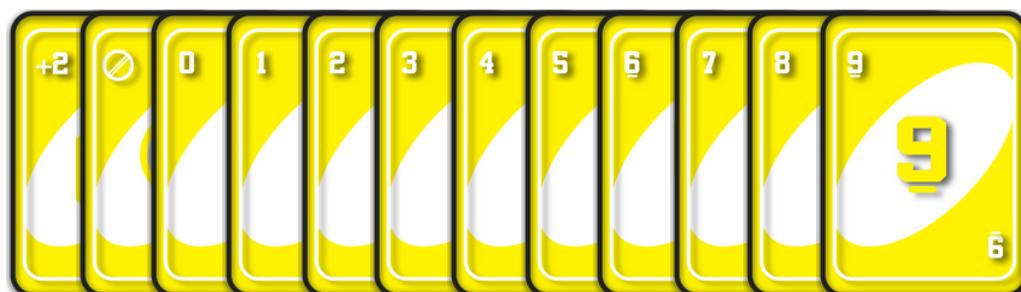
Set carte Rosse



Set carte Blu



Set carte Gialle



Set carte Verdi



1.3. Dettaglio carte speciali

Anche la grafica delle carte speciali è stata creata da noi tramite **Adobe Illustrator** e **Adobe Photoshop**.

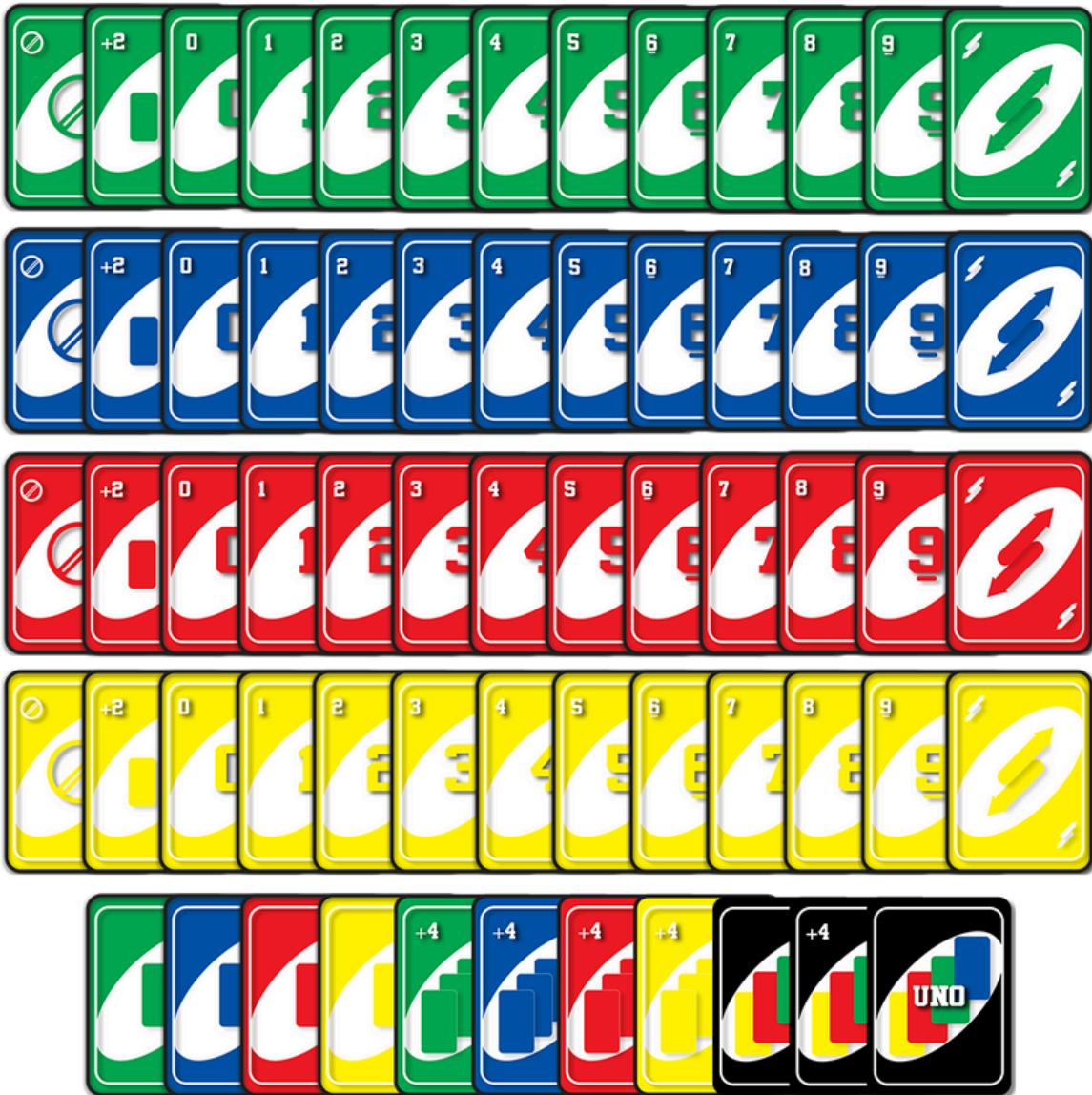


Carta speciale, essa può comparire in **4** colorazioni (Verde, Blu, Giallo e Rosso). **Il suo scopo è a scopo illustrativo**, serve per comunicare a tutti i giocatori il colore che il *Players o Bot* ha scelto a seguito di aver giocato un **Cambia Colore**.

Carta speciale, essa può comparire in **4** colorazioni (Verde, Blu, Giallo e Rosso). **Il suo scopo è a scopo illustrativo**, serve per comunicare a tutti i giocatori che i *Players o Bot* ha scelto giocato un **Cambia colore +4**.



Illustrazioni di tutte le carte presenti in gioco

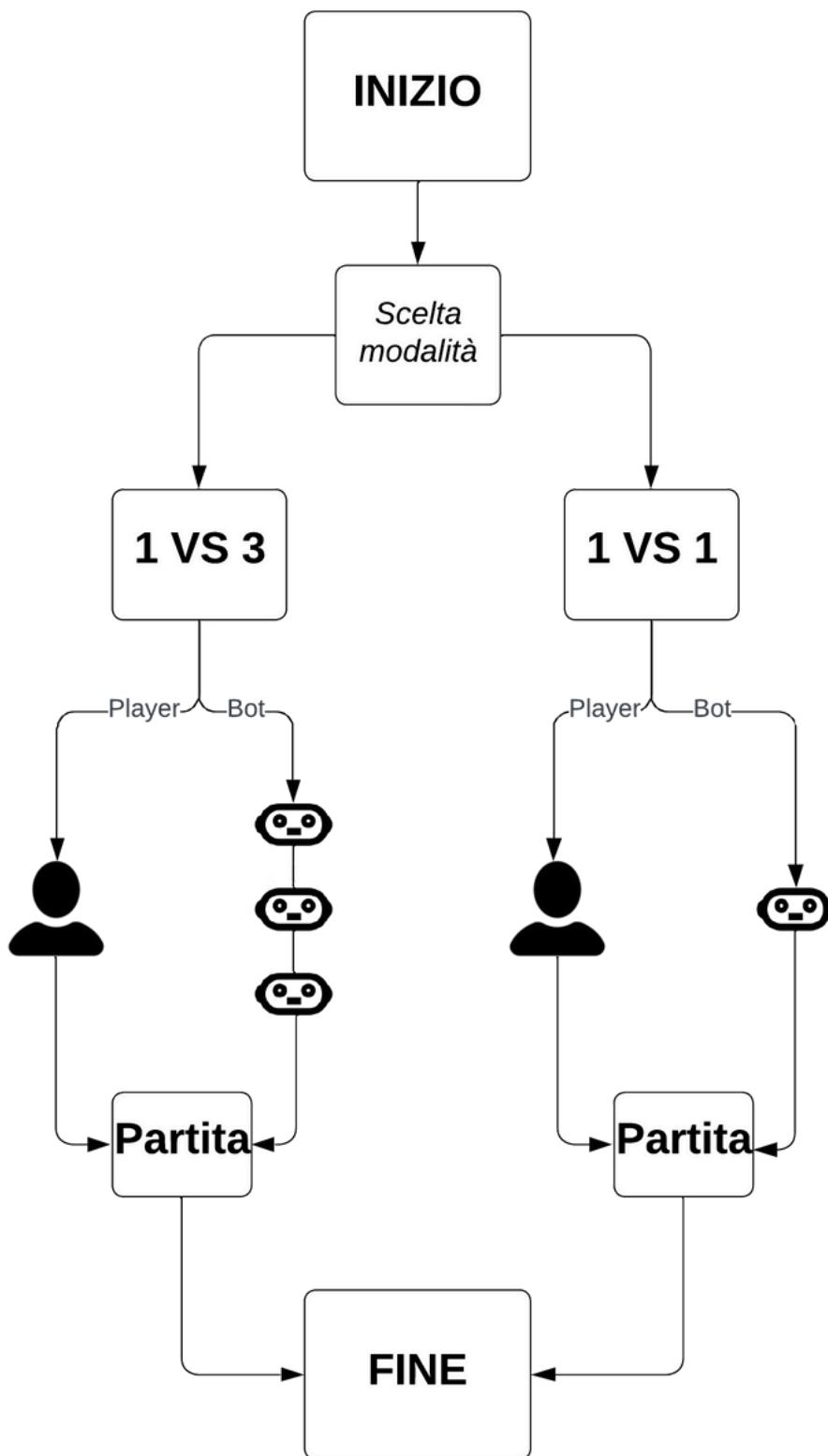


Complessivamente nel mazzo sono presenti **108** carte:

- 76 carte numerate
- 8 salta turno
- 8 cambio giro
- 8 pesca due
- 4 cambio colore
- 4 cambio colore +4

1.4. Diagramma casi d'uso

9



1.5. Modello del dominio

Di seguito vengono raffigurati tutti gli **UML** realizzati per strutturare il *modello del dominio*. Son state create due classi per le carte del gioco, **NormalCard** e **ActionCard** entrambe implementano l'Interface Card.

Implementazione Palumbo Giovanni:

La **NormalCard** rappresenta tutte le carte base del gioco contiene anche due *classi Enum*:

- **Number:** per rappresentare il numero.
- **Colour:** per rappresentare il colore delle carte.

Metodi della classe **NormalCard**:

- **NormalCard(Number num, Colour c):** costruttore della classe.
- **getNumber():** restituisce il numero della Card.
- **getColor():** restituisce il colore della Card.
- **getName():** restituisce il nome della Card.
- **setColor(Colour c):** setter per il colore.
- **setNumber(Number num):** setter per il numero.

Implementazione Diego Cecchini:

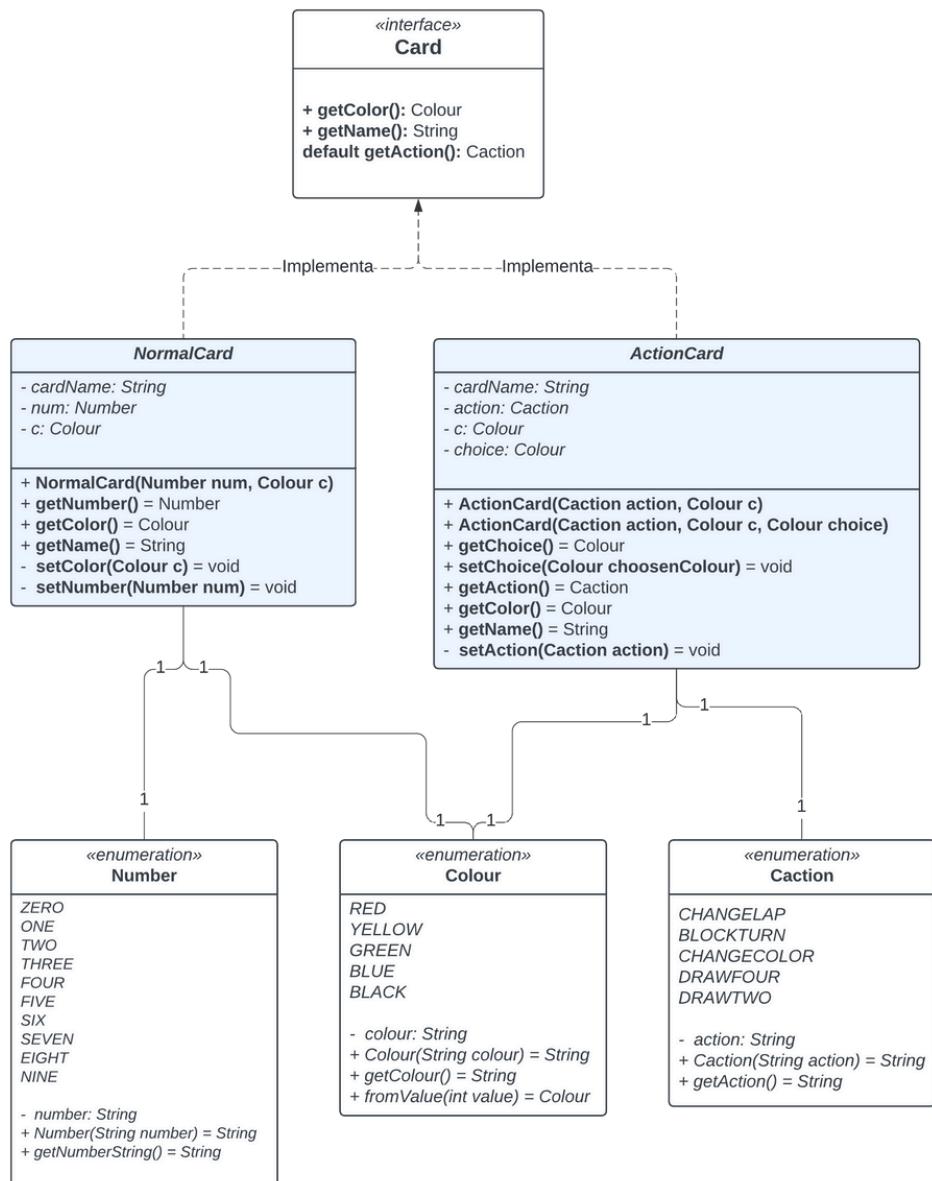
La classe **ActionCard** raffigura le carte che hanno abilità speciali; anche questa classe racchiude al suo interno due classi *Enum*:

- **Number:** per rappresentare il numero.
- **Caction:** per differenziare le diverse abilità.

Metodi della classe **ActionCard** sono presenti i medesimi metodi della classe descritta in precedenza con l'aggiunta dei metodi di seguito riportati:

- **ActionCard(Caction action, Colour c):** costruttore della classe.
- **ActionCard(Caction action, Colour c, Colour choice):** secondo costruttore della classe.
- **getChoice():** restituisce il colore scelto.
- **setChoice(Colour chosenColour):** setter colore.
- **getAction():** restituisce l'abilità della Card.
- **setAction(Caction action):** setter dell'abilità della Card.

Schema UML fig.1

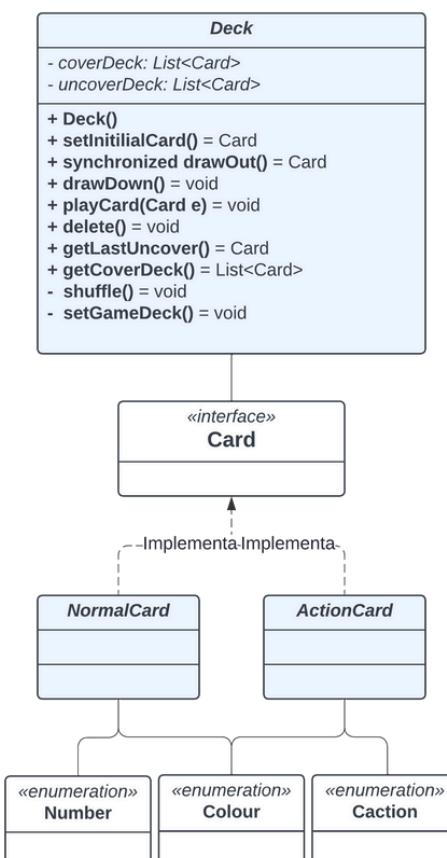


Implementazione Palumbo Giovanni e Diego Cecchini:

In questo **UML** viene rappresentata la classe **Deck**. Al suo interno ha come attributi due *liste* che sono associate all'Interface **Card**. Sono stati realizzati inoltre i seguenti metodi per fornire funzionalità al **Deck**:

- **Deck()**: costruttore della classe.
- **setInitialCard()**: setta la carta iniziale.
- **drawOut()**: pesca la prima carta dal mazzo.
- **drawDown()**: ripristina il mazzo mescolandolo e pesca una carta.
- **playCard(Card e)**: gioca una carta sul tavolo da gioco.
- **delete()**: “pulisce” sia il mazzo scoperto che quello coperto ovver il “Pozzo”.
- **getLastUncover()**: restituisce l’ultima carta visibile a centro del tavolo.
- **getCoverDeck()**: restituisce una *List<Card>* con tutte le carte che compongono il “Pozzo”.
- **shuffle()**: mescola il mazzo.
- **setGameDeck()**: genera il mazzo di carte iniziale.

Schema UML fig.2



*Implementazione **Diego Cecchini**:*

Nell'UML della pagina seguente viene raffigurata la classe **BotPlayer**, questa classe ha lo scopo di raffigurare i “bot” che si trovano nelle due modalità di gioco. Implementa un'interface **Player** con i seguenti metodi default:

- **isCardValid(Card cardToPlay)**: valida la carta giocata.
- **isActionCardValidAgainstNormalCard(ActionCard cardToplay, NormalCard currentCard)**: restituisce un boolean a seguito di un confronto fatto tra l'ActionCard giocata e la NormalCard corrente.
- **isNormalCardValidAgainstActionCard(NormalCard cardToplay, ActionCard currentCard)**: restituisce un boolean a seguito di un confronto fatto tra NormalCard giocata e la NormalCard corrente.
- **isActionCardValidAgainstActionCard(ActionCard cardToplay, ActionCard currentCard)**: restituisce un boolean a seguito di un confronto fatto tra l'ActionCard giocata e la ActionCard corrente.
- **isNormalCardValidAgainstNormalCard(NormalCard cardToplay, NormalCard currentCard)**: restituisce un boolean a seguito di un confronto fatto tra la NormalCard giocata e la NormalCard corrente.

Son state create due classi per differenziare i le tipologie di giocatori, in **BotPlayer** sono presenti i seguenti metodi:

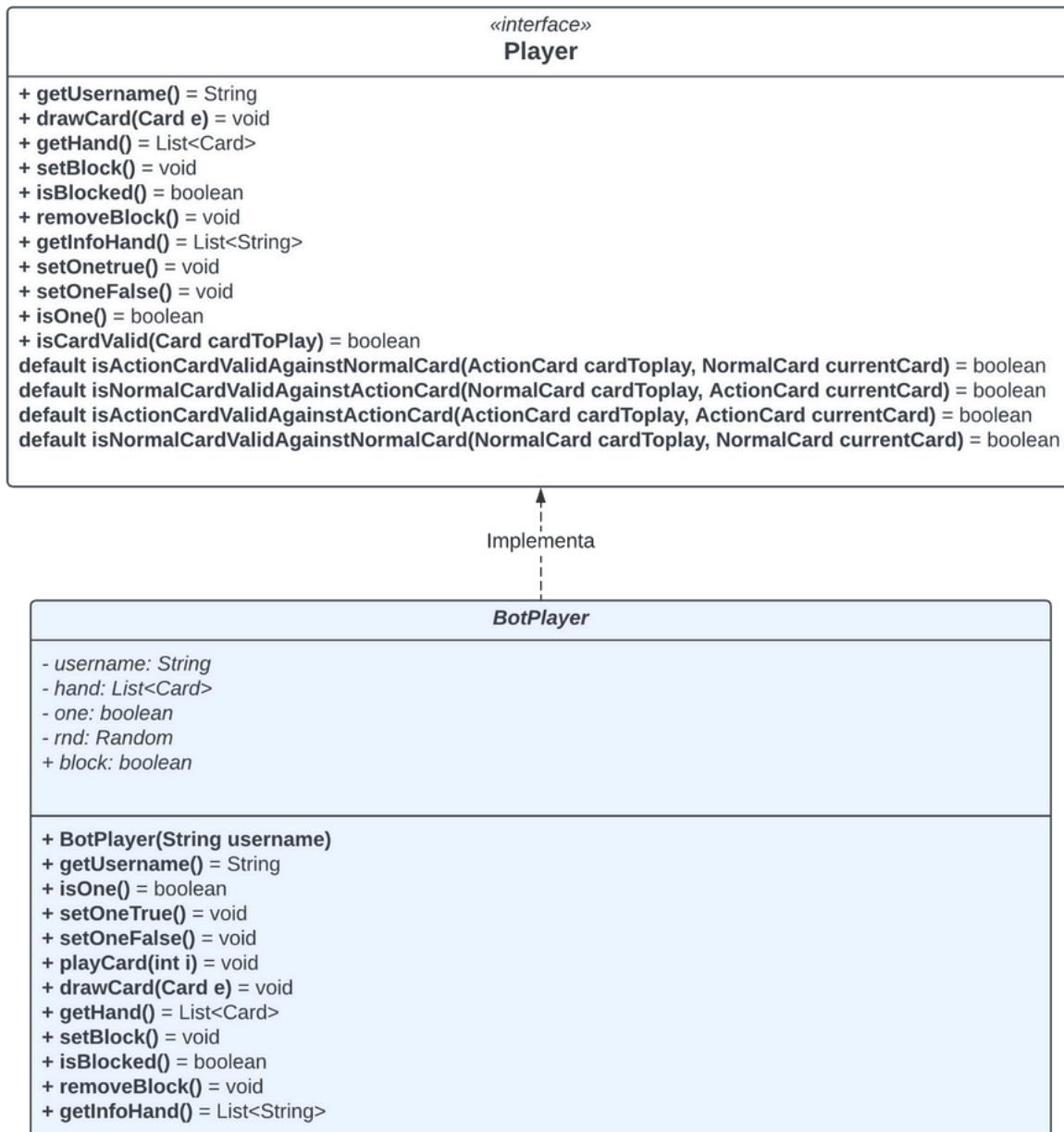
- **BotPlayer(String username)**: costruttore della classe.
- **getUsername()**: restituisce l'Username.
- **isOne()**: restituisce l'attributo one.
- **setOneTrue()**: setta l'attributo one a true.
- **setOneFalse()**: setta l'attributo one a false.
- **playCard(int i)**: gioca una carta.
- **drawCard(Card e)**: aggiunge una Card alla mano del giocatore.
- **getHand()**: restituisce una List<Card> con le carte in mano al giocatore.
- **setBlock()**: blocca un giocatore.
- **isBlocked()**: verifica se il giocatore è bloccato.
- **removeBlock()**: sblocca il giocatore.
- **getInfoHand()**: restituisce una List<String> contenente le carte in mano al giocatore.

*Implementazione **Palumbo Giovanni**:*

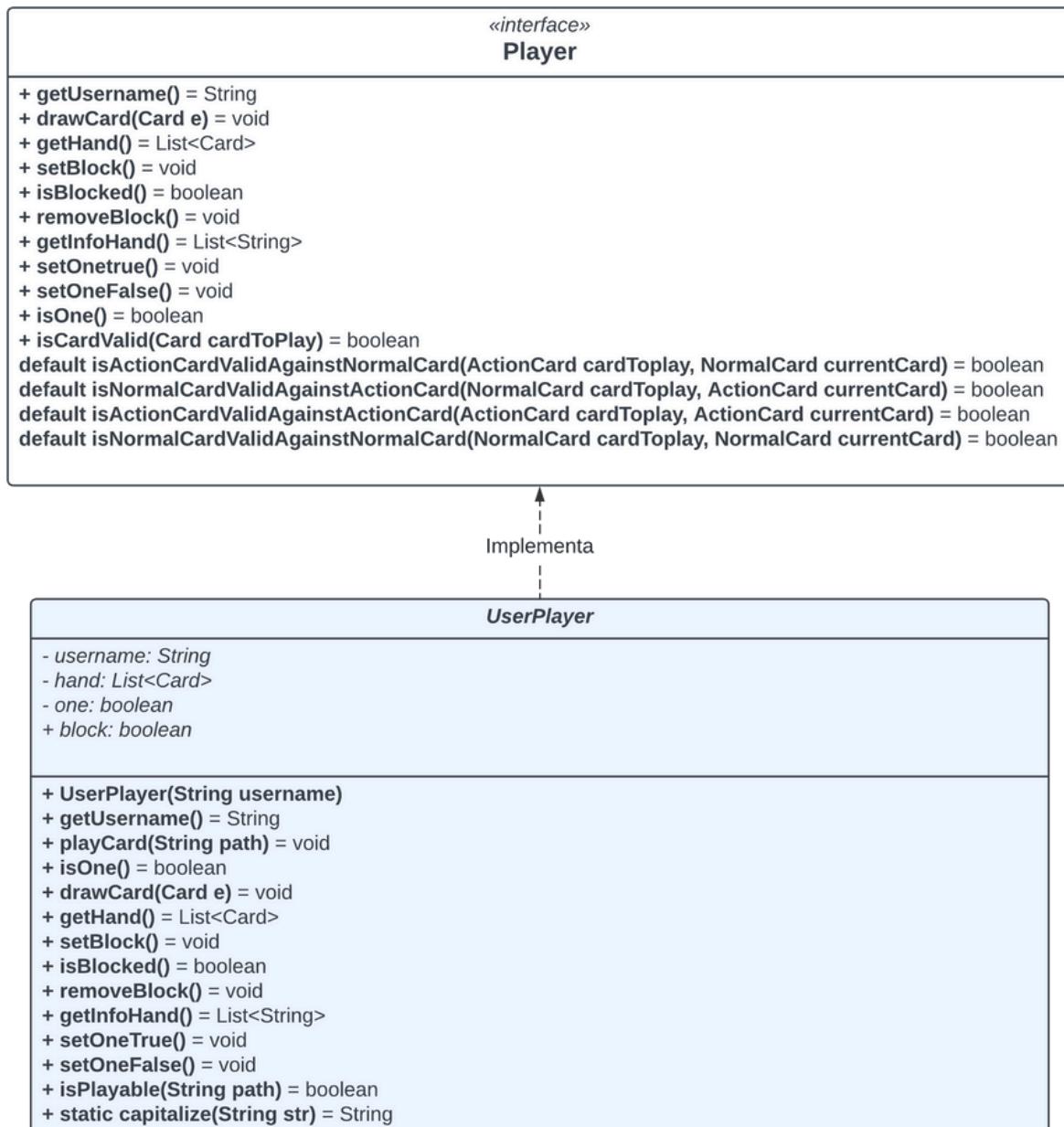
Nella classe **Userplayer** sono presenti i metodi di sopra elencati con l'aggiunta di diversi metodi:

- **UserPlayer(String username)**: costruttore della classe.
- **playCard(String path)**: gioca una carta.
- **isPlayable(String path)**: verifica se la Card giocata risulta valida.
- **capitalize(String str)**: modifica la stringa passata per parametro.

Schema UML fig.3



Schema UML fig.4



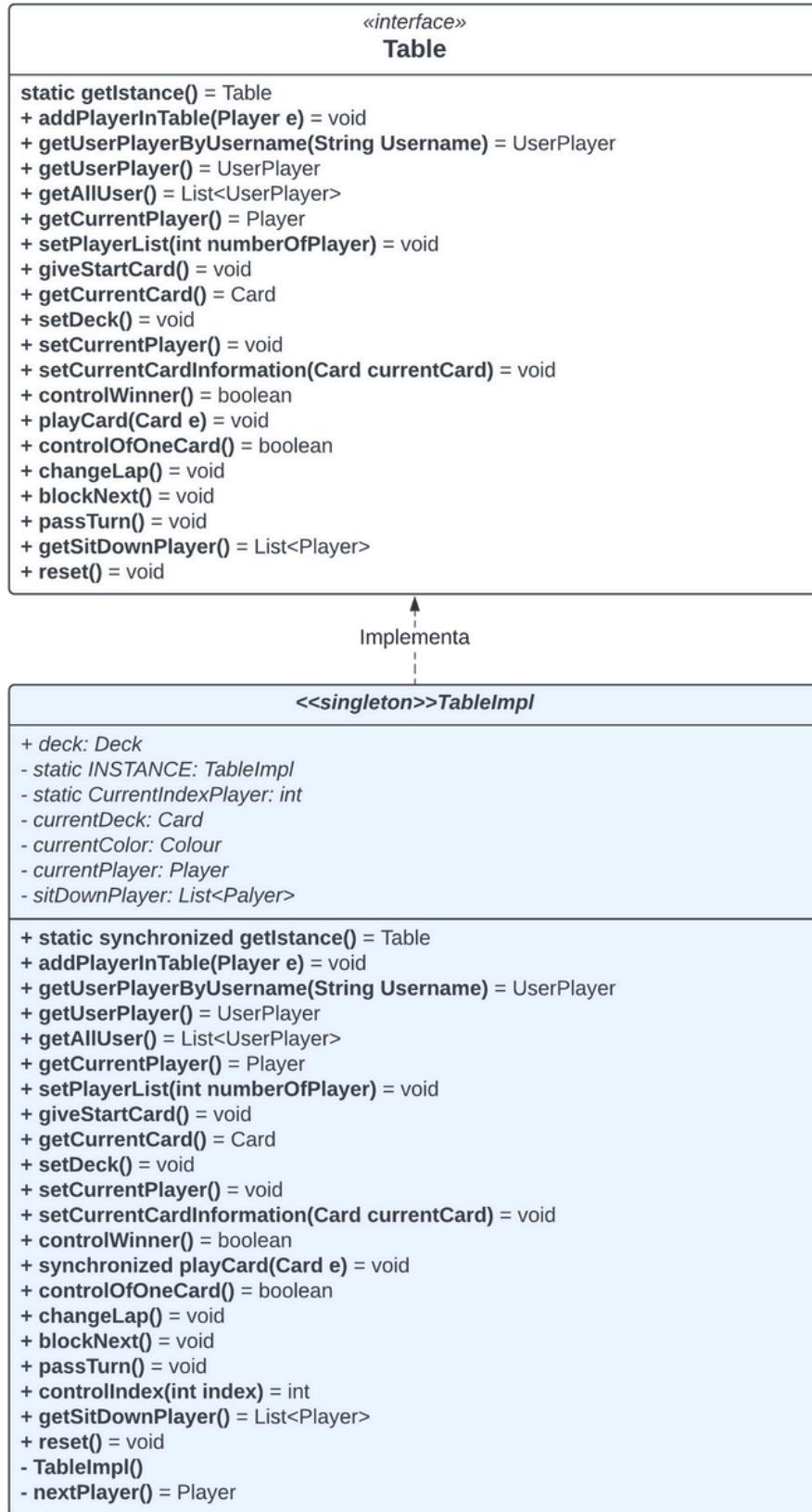
Implementazione **Palumbo Giovanni e Diego Cecchini:**

La classe **TableImpl** rappresenta il tavolo da gioco, che implementa l'interface **Table**; inoltre, in questa classe è stato implementato il **Pattern Singleton**, usato per forzare la creazione di una sola istanza per questa classe.

I metodi presenti nella classe **TableImpl** sono i seguenti:

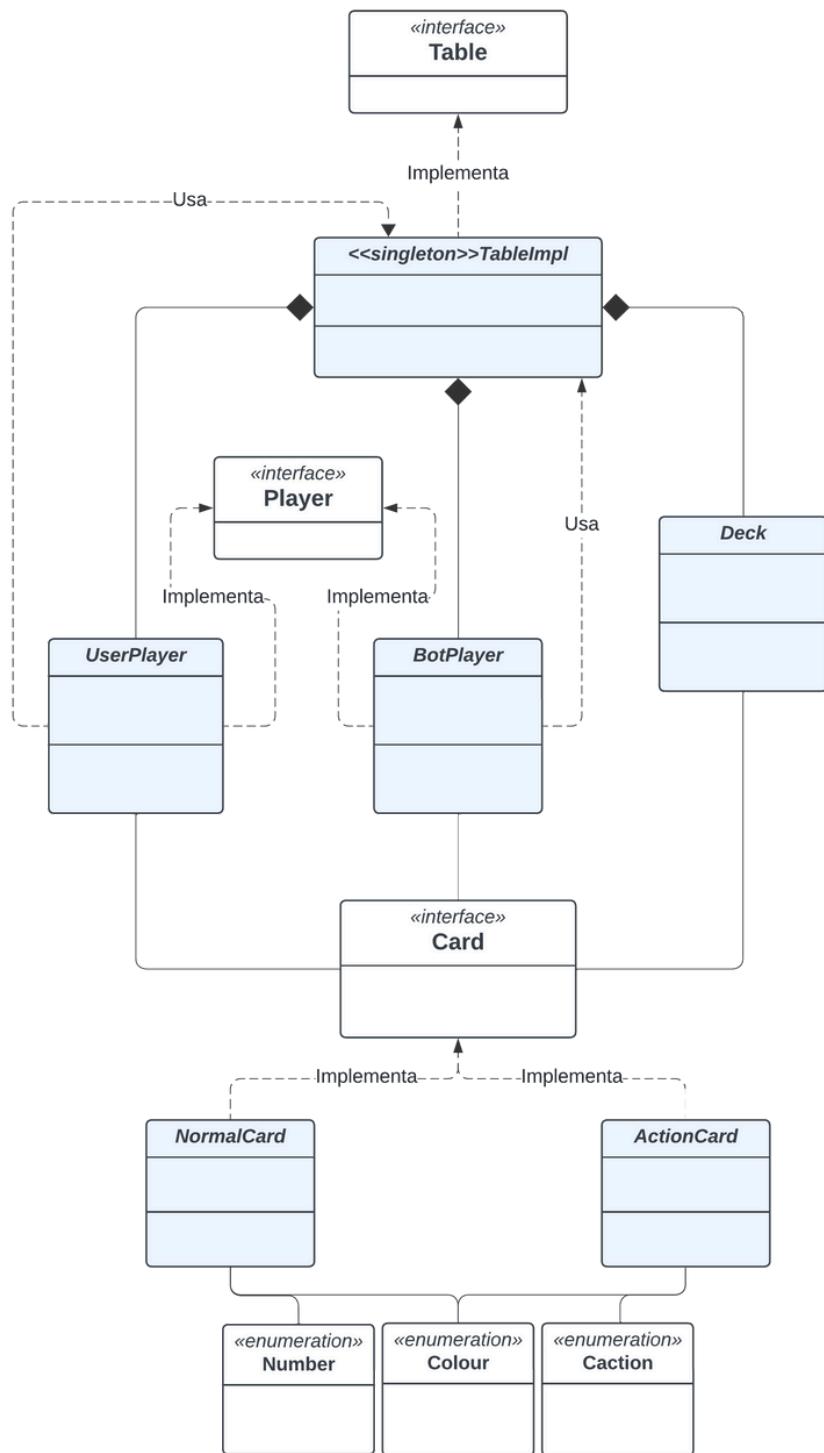
- **TableImpl():** costruttore.
- **getInstance():** restituisce l'istanza della classe **TableImpl**.
- **addPlayerInTable(Player e):** aggiunge giocatori al tavolo da gioco.
- **getUserPlayerByUsername(String Username):** restituisce il giocatore con il nome passato come parametro.
- **getUserPlayer():** restituisce un giocatore della partita.
- **getAllUser():** restituisce una lista con tutti i giocatori della partita.
- **getCurrentPlayer():** restituisce il giocatore che sta giocando.
- **setPlayerList(int numberOfPlayer):** setta la lista dei giocatori.
- **giveStartCard():** distribuisce le carte per iniziare la partita.
- **getCurrentCard():** restituisce la carta sul mazzo scoperto del tavolo.
- **setDeck():** setta il deck del tavolo da gioco.
- **setCurrentPlayer():** setta il giocatore per dargli la possibilità di giocare.
- **setCurrentCardInformation(Card currentCard):** setta le informazioni utili per tener traccia dell'ultima carta giocata.
- **controlWinner():** effettua un controllo sui giocatori per vedere se qualcuno ha vinto.
- **playCard(Card e):** permette di giocare una carta.
- **controlOfOneCard():** verifica che il giocatore ha esclusivamente una carta.
- **changeLap():** cambia il giro in cui i giocatori giocano le proprie carte.
- **blockNext():** blocca il giocatore successivo nel giro di gioco.
- **passTurn():** permette di passare il proprio turno.
- **controlIndex(int index):** verifica che l'indice rientra fra il numero massimo e minimo di giocatori seduti al tavolo.
- **getSitDownPlayer():** restituisce una lista con i giocatori seduti al tavolo
- **reset():** resetta tutti tutti gli attributi presenti nella classe **TableImpl**.
- **nextPlayer():** restituisce l'istanza del giocatore successivo.

Schema UML fig.5



Schema riassuntivo sul modello del dominio

Schema UML fig.6



2.1.Architettura

L'implementazione del pattern architettonico **MVC** (**Model-View-Controller**) implica l'adozione di una metodologia di progettazione che separi tra loro tutte le funzioni e le competenze. Possiamo dividere l'applicazione in tre componenti principali:

- **Model:**

- *si occupa di tutti i dati contenuti nell'applicazione e ne definisce il comportamento. La sua funzione agisce dalla definizione dei dati ad ogni azione che può interagire con essi.*

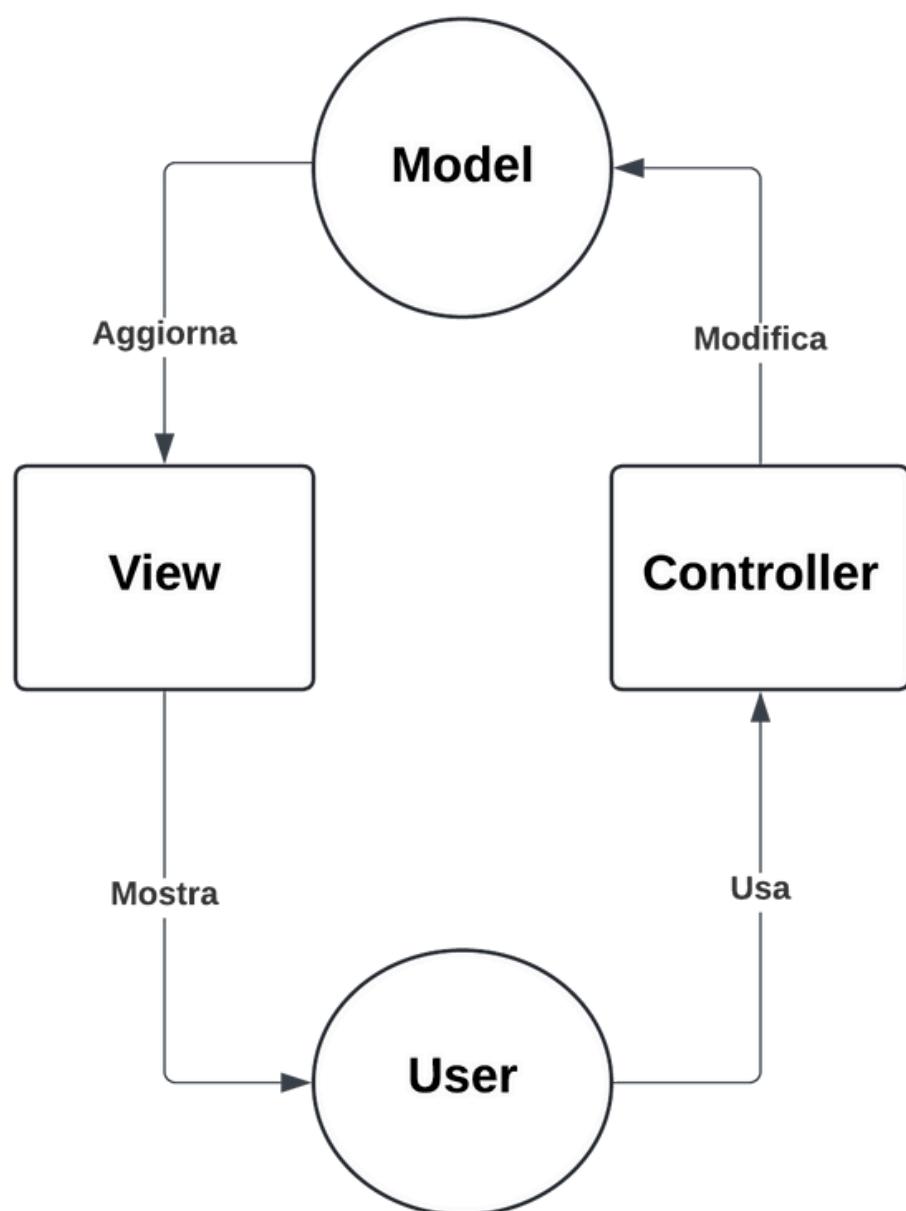
- **View:**

- *si occupa esclusivamente della presentazione dei dati, ovvero come si vede in output. Gestisce, di fatto, la grafica e coordina la presentazione dell'output all'utente.*

- **Controller:**

- *agisce da ponte fra il Model e la View ed è il principale componente dell'interazione, preoccupandosi di gestire le interazioni con l'utente ed aggiornando di conseguenza Model e View.*

Il primo impiego di Model-View-Controller e dei design patterns in generale è quello di migliorare l'implementazione del codice.

Diagramma Architettura MVC generica

2.2. Design dettagliato

Implementazione **Palumbo Giovanni**:

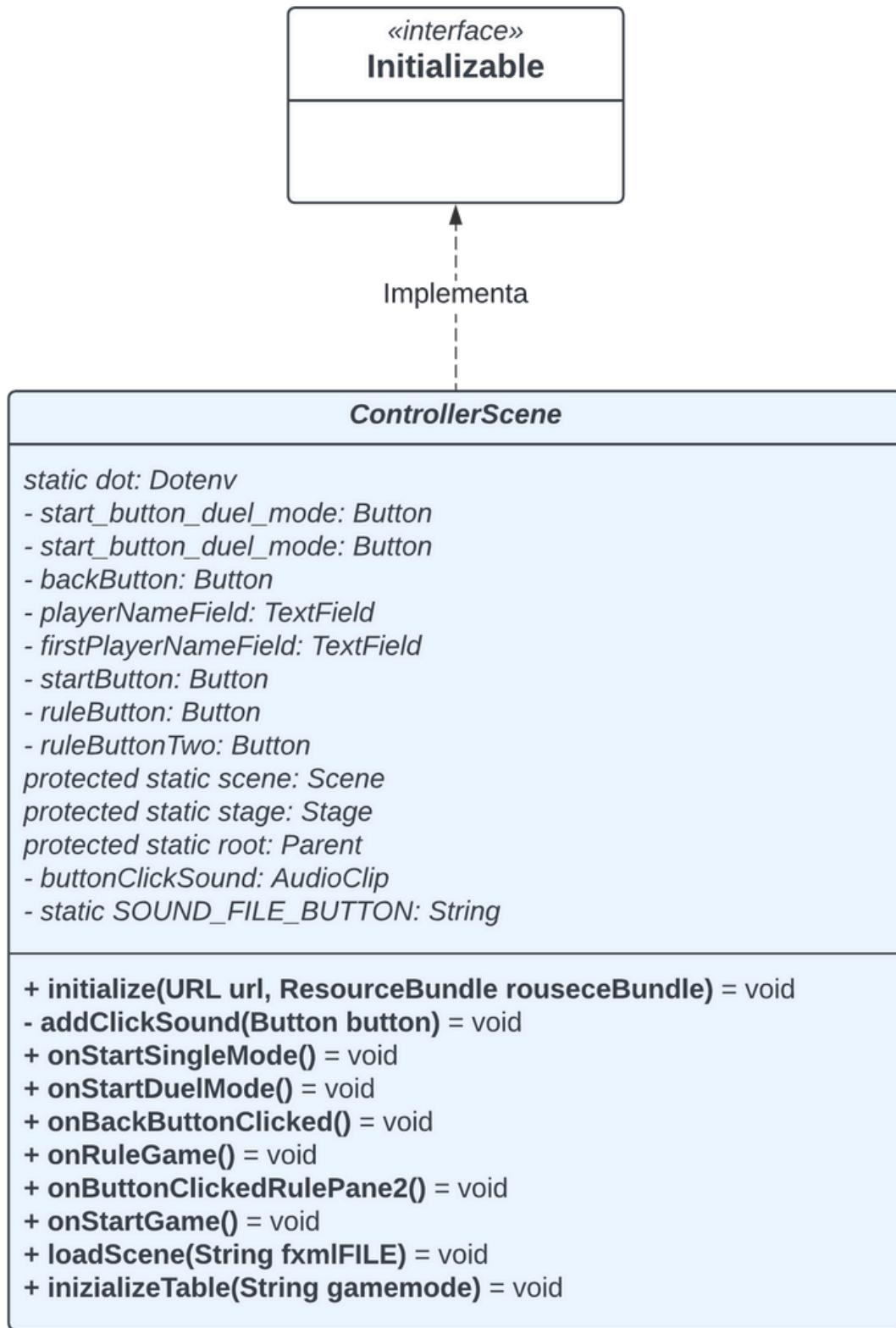
La classe **ControllerScene** fa parte dell'interfaccia grafica associata al modello così come descritto in precedenza.

Implementa ***l'interface Initializable***. La funzione di questa classe è quella di gestire il cambio di scene fra i vari pane realizzati, uno per ogni schermata di gioco; inoltre, per creare un'esperienza di gioco più appagante, è stata implementata una soundTrack di sottofondo e sono stati inseriti dei suoni per dare dei feedback all'utente quando viene giocata una carta e quando viene premuto un bottone.

La classe **ControllerScene** al suo interno prevede i seguenti metodi:

- **initialize(URL url, ResourceBundle rouseceBundle):**
 - inizializza l'AudioClip con il suono e aggiunge ad ogni bottone il suono.
- **addClickSound(Button button):**
 - aggiunge il suono al click del bottone.
- **onStartSingleMode():**
 - carica il pane “Insert_name_single_players.fxml”.
- **onStartDuelMode():**
 - carica il pane “Insert_name_single_players.fxml”.
- **onBackButtonClicked():**
 - ricarica il pane “Select_game_mode.fxml”.
- **onRuleGame():**
 - carica il pane “Rule_pane_one”.
- **onButtonClickedRulePane2():**
 - carica il pane “Rule_pane_two”.
- **onStartGame()**
 - crea il campo da gioco e inizia la partita.
- **inizializeTable(String gamemode);**
 - inizializza il table.
- **loadScene(String fxmFILE):**
 - metodo con vari controlli che permette di caricare la scena in base alla schermata in cui ci troviamo.

Schema UML fig.7

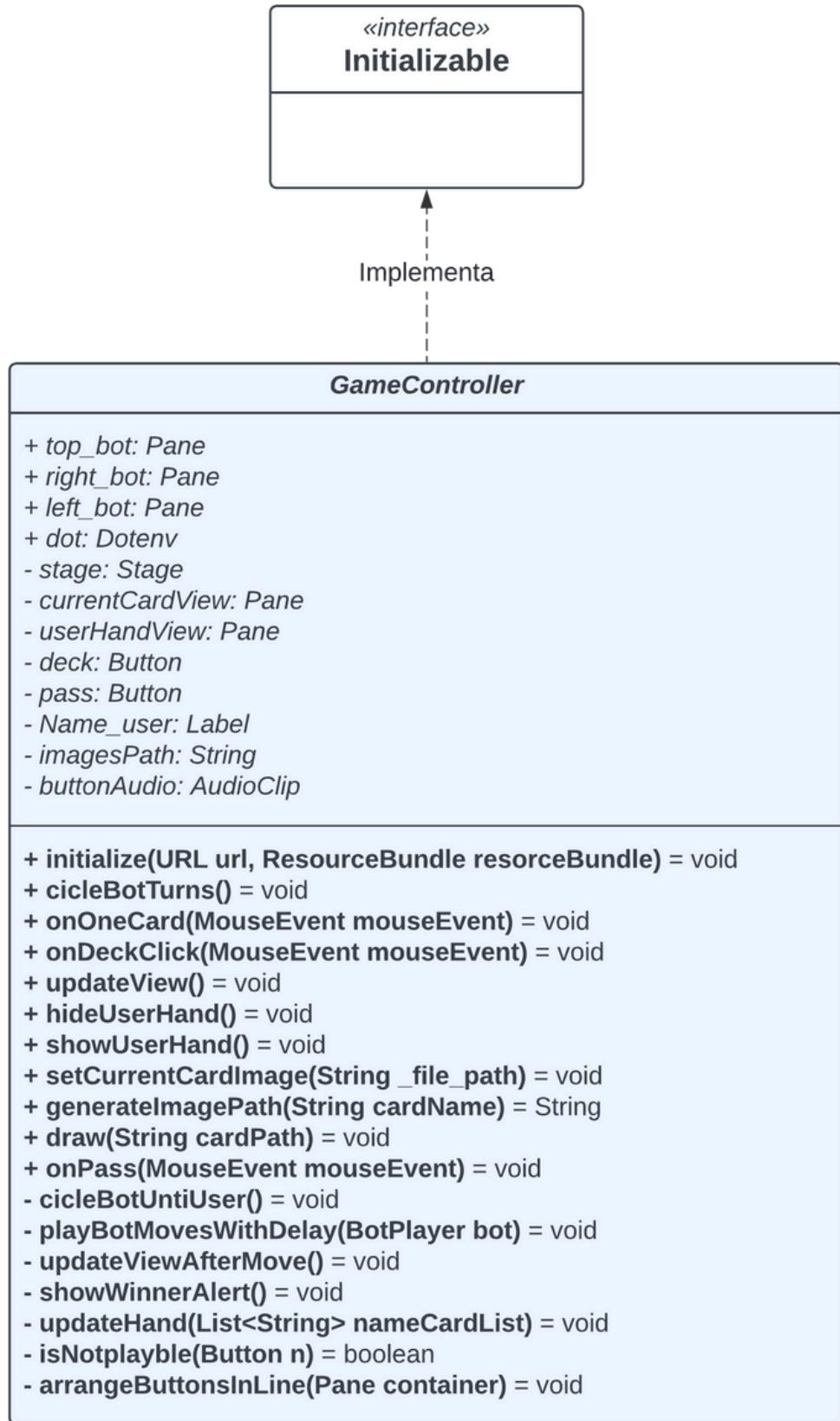


Implementazione **Diego Cecchini**:

La classe **GameController** rappresenta la classe Controller delle due modalità di gioco, implementa l'interface **Initializable**. Al suo interno sono presenti i seguenti metodi:

- **initialize(URL url, ResourceBundle resorceBundle)**: *inizializza la partita.*
- **cicleBotTurn()**: *permette il passaggio dei turni fra i bot.*
- **onOneCard(MouseEvent mouseEvent)**: *metodo utilizzato per l'esecuzione del tasto UNO.*
- **onDeckClick(MouseEvent mouseEvent)**: *metodo per pescare dal deck centrale.*
- **updateView()**: *aggiorna la View.*
- **hideUserHand()**: *disabilita la possibilità di utilizzare dei button.*
- **showUserHand()**: *abilita la possibilità di utilizzare dei button.*
- **setCurrentCardImage(String _file_path)**: *setta la carta giocata.*
- **generateImagePath(String cardName)**: *crea l'imagePath.*
- **draw(String cardPath)**: *pesca una carta.*
- **onPass(MouseEvent mouseEvent)**: *passa il turno.*
- **isNotplayble(Button n)**: *verifica se l'User può giocare una carta.*
- **cicleBotUntiUser()**: *controlla se il giocatore corrente è un Bot e se la partita non è ancora conclusa.*
- **playBotMovesWithDelay(BotPlayer bot)**: *attribuisce un delay ai movimenti del bot.*
- **updateViewAfterMove()**: *aggiorna la View dopo aver giocato.*
- **showWinnerAlert()**: *mostra il vincitore.*
- **updateHand(List<String> nameCardList)**: *aggiorna la vista delle carte che ha in "mano" il giocatore.*
- **arrangeButtonsInLine(PaneContainer)**: *adatta la View in base alle carte in mano al giocatore.*

Schema UML fig.8



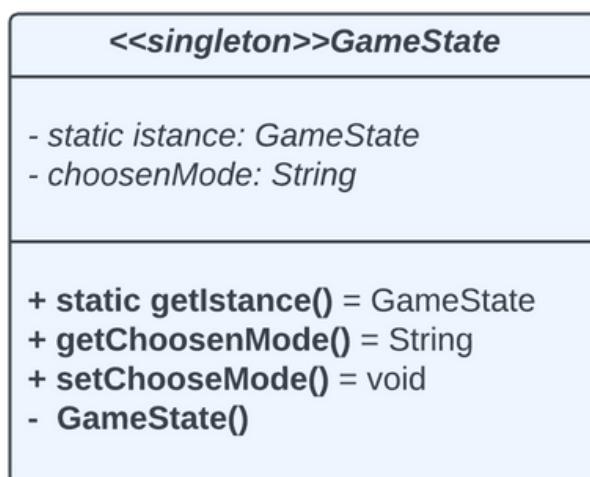
Implementazione Diego Cecchini:

La classe **GameState** è stata creata per memorizzare la modalità di gioco scelta dall'utente, in questo modo si possono gestire più schermate di gioco con un solo *ControllerScene* e un unico *GameController*.

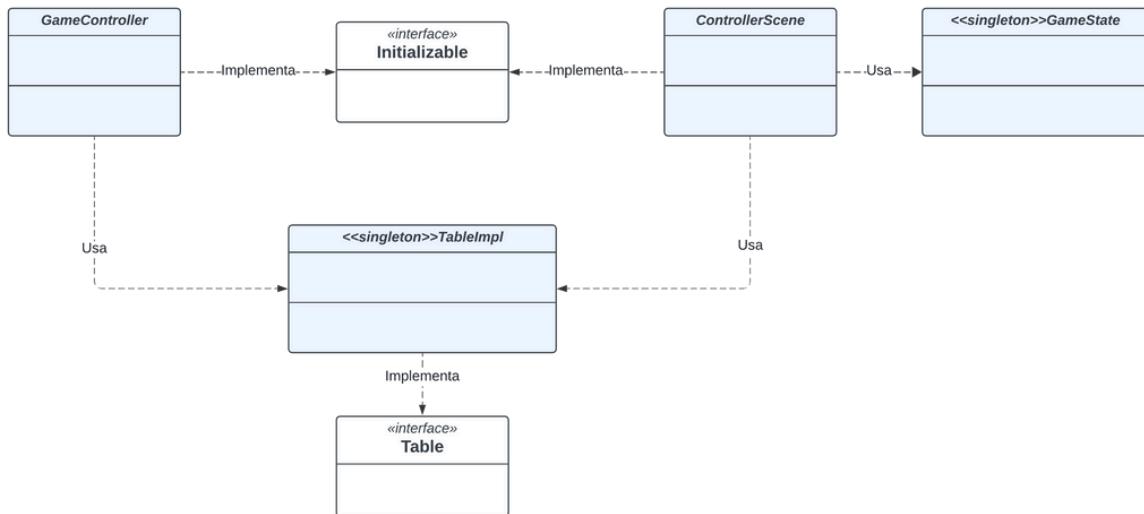
Al suo interno la classe **GameState** contiene i seguenti metodi:

- **GameState()**: costruttore privato della classe.
- **getInstance()**: restituisce l'istanza.
- **getChoosenMode()**: restituisce l'attributo *chooseMode* che racchiude la modalità di gioco scelta.
- **setChooseMode()**: setta l'attributo *chooseMode*.

Schema UML fig.9

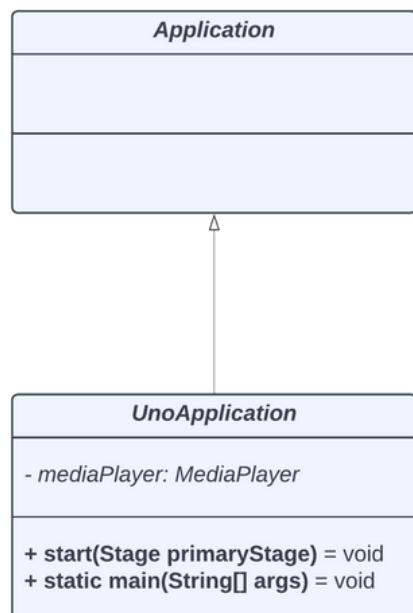


Schema UML fig.10



La classe **UnoApplication**, riportata qui sotto, viene utilizzata solo per il caricamento della prima schermata.

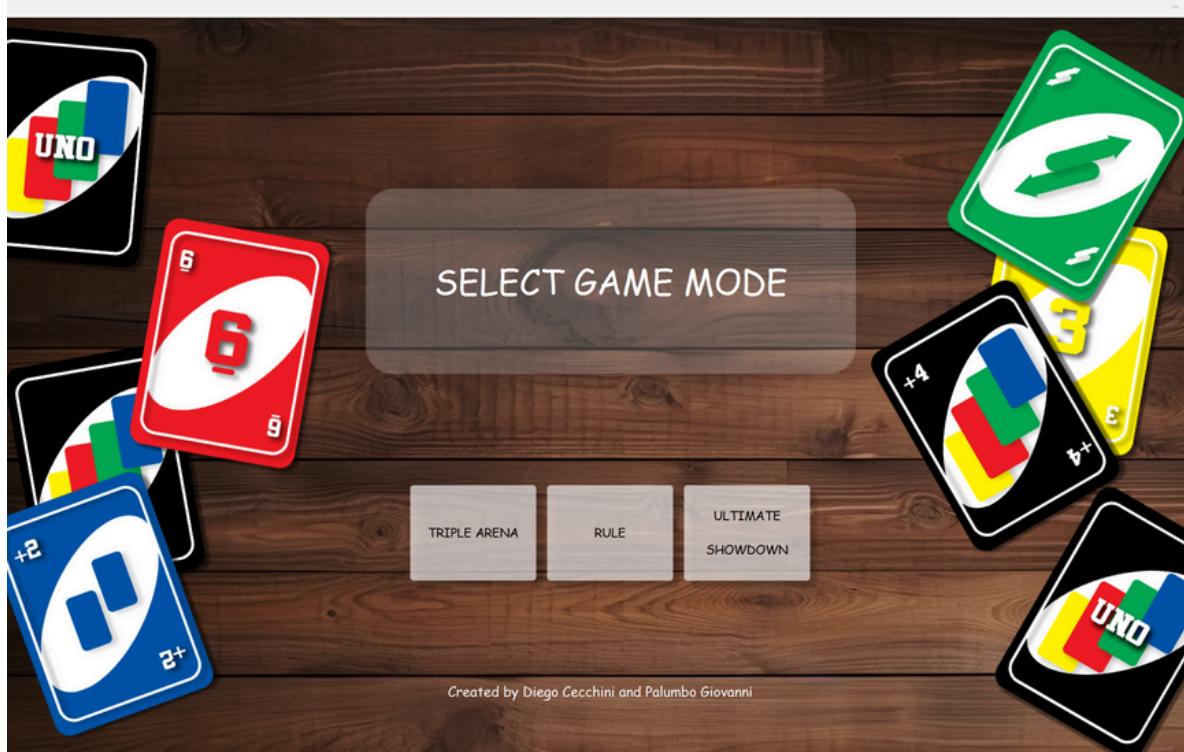
Schema UML fig.11



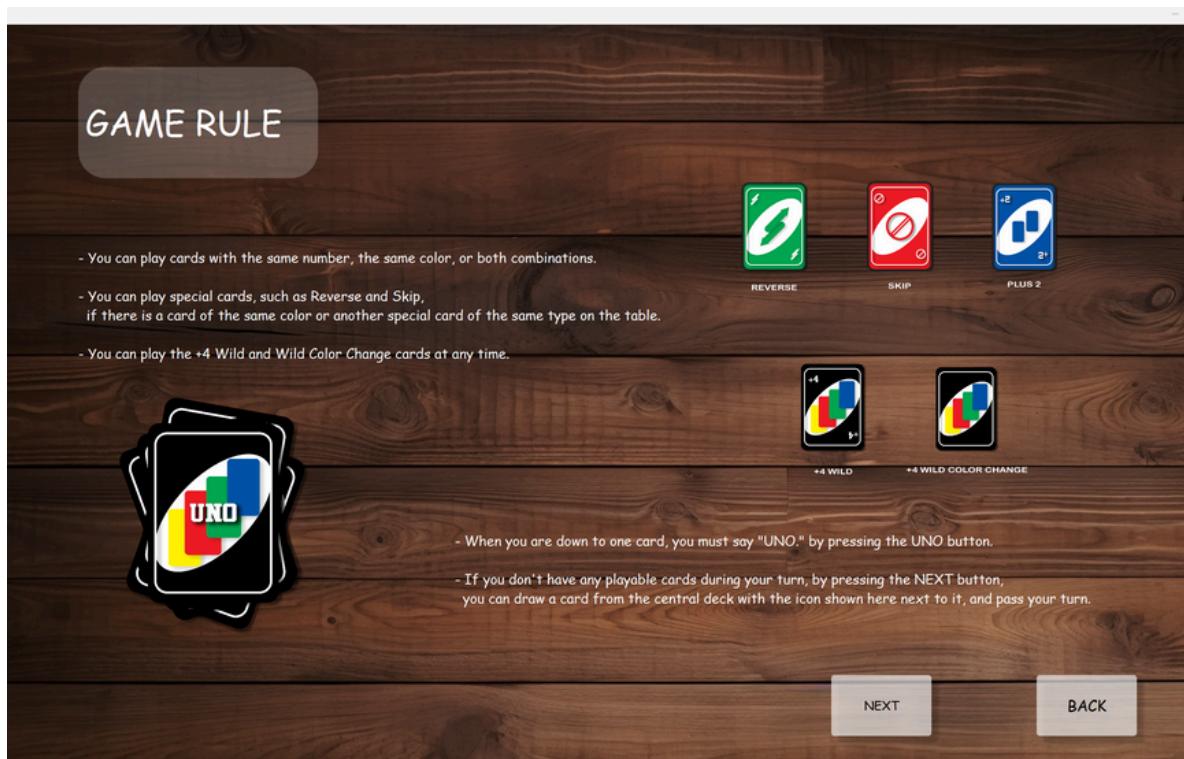
Le foto di seguito riportate non sono render grafici creati prima di sviluppare il gioco, ma rappresentano le effettive schermate di gioco.

27

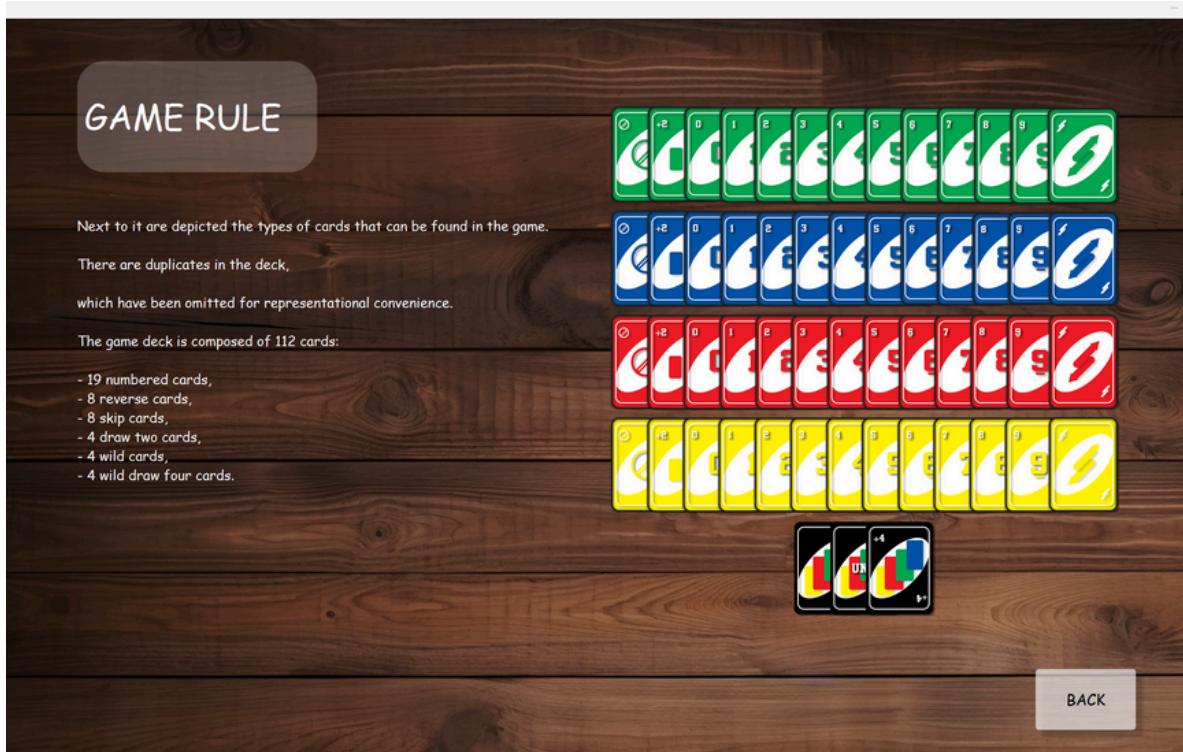
Schermata selezione modalità di gioco fig.12



Schermata regole di gioco pag.1 fig.13



Schermata regole di gioco pag.2 fig.14

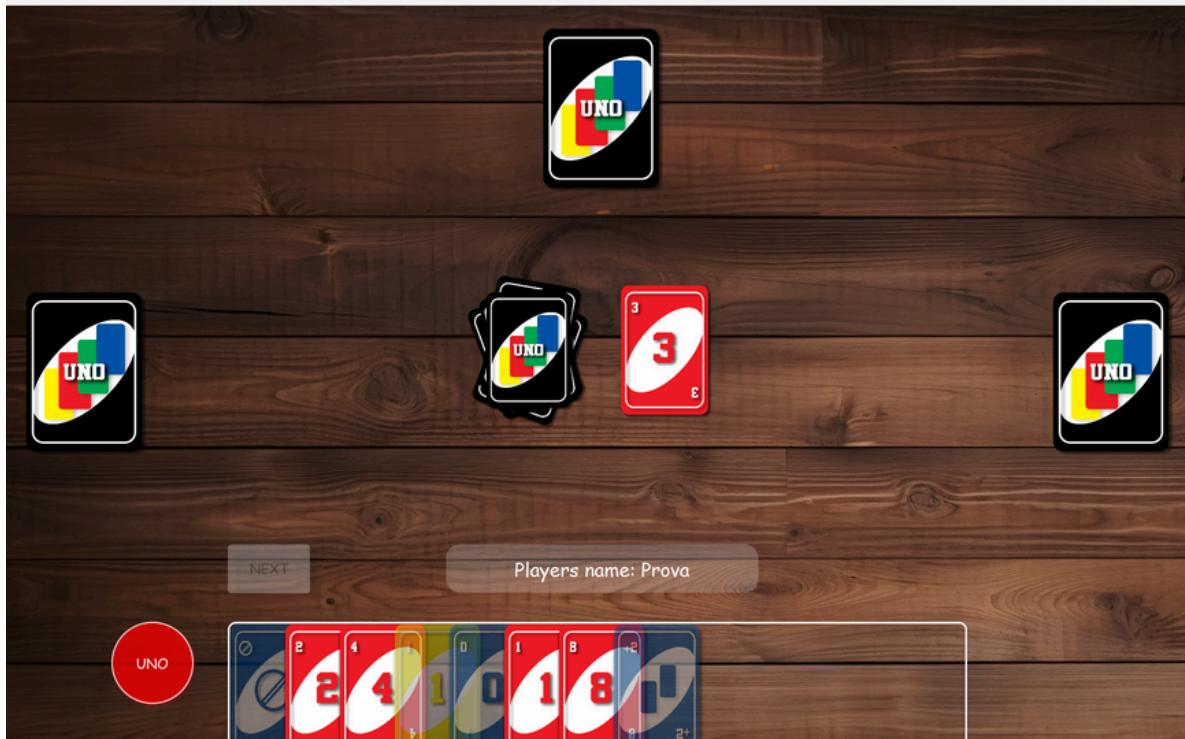


Schermata inserimento nome fig.15

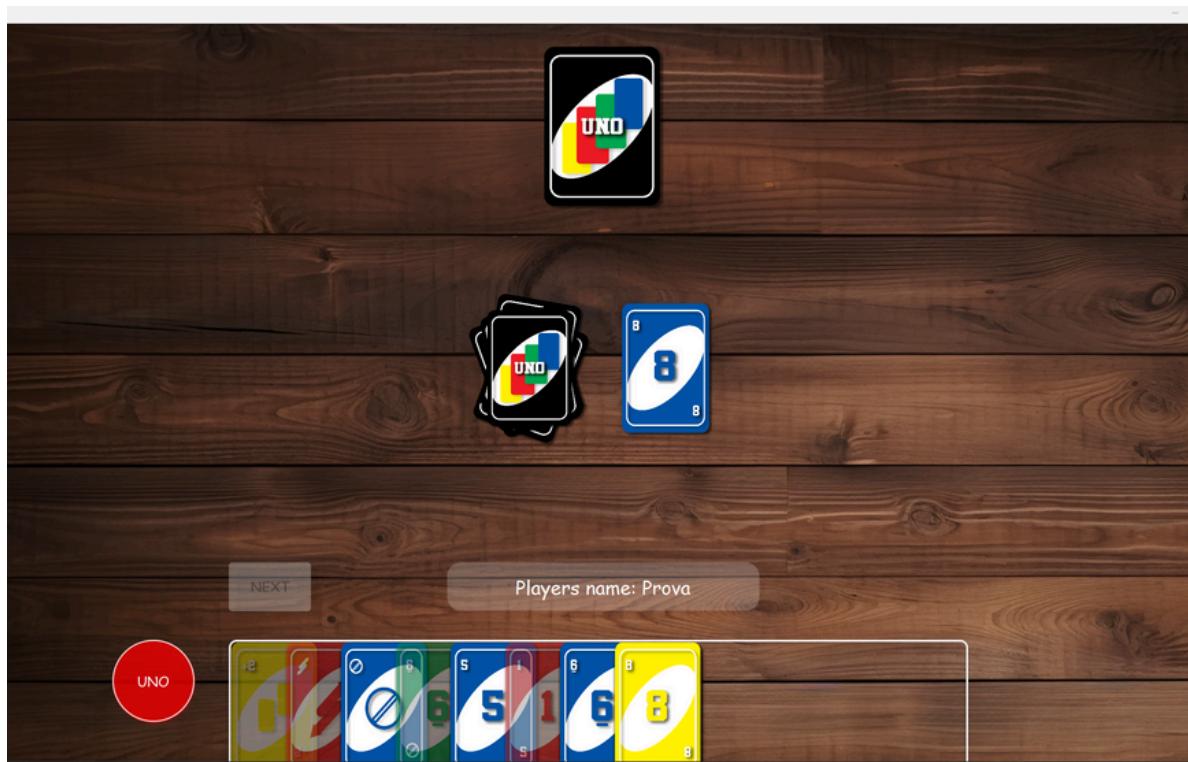


29

Schermata di gioco modalità 1 vs 3 pag.2 fig.16



Schermata di gioco modalità 1 vs 1 pag.2 fig.17



3.SVILUPPO

30

3.1.Metodologia di lavoro

In questa sezione si affronta la metodologia di lavoro sulla cui base è stato sviluppato l'intero progetto.

Il **primo passo** è stato quello di cercare ed ideare una specifica originale; nella fase iniziale, ci siamo confrontati molto su quale gioco realizzare. Inizialmente le idee erano molto vaghe, ma con il passare dei giorni siamo riusciti ad elaborare una specifica originale.

Una volta decisa la specifica ci siamo concordati sullo stilare una proposta corrispondente alle linee guida.

Il **secondo passo** è stato quello di creare e sviluppare un Modello, il più ottimizzato possibile, tramite l'utilizzo di *Unified Modeling Language (UML)* introdotto durante il corso. Questo passaggio è stato fondamentale anche per creare un piano di lavoro.

Il **terzo passo** è stato iniziare l'implementazione del modello sotto forma di codice. Per ottimizzare il lavoro abbiamo deciso fin da subito di creare un progetto su **GitHub** e utilizzare come ambiente di lavoro **IntelliJ IDEA**; procedendo in questo modo siamo riusciti a lavorare al meglio anche a distanza e a tener traccia di tutte le modifiche che effettuavamo tramite i vari *commit*. Questa modalità operativa non è stata l'unico strumento di comunicazione, infatti oltre al vederci spesso di persona per lavorare insieme, abbiamo fatto ampio utilizzo anche di **chat online** e strumenti per videochiamate come **Discord**.

3.2. Testing automatizzato

Nel **quarto passo** ci siamo concentrati sul rendere il modello completamente funzionante per poterlo testare.

Per il *testing automatizzato* abbiamo utilizzato il framework di *unit testing* (**JUnit5**), già visto anche durante alcune simulazioni in classe. Questo ci ha permesso di procedere con dei test periodici durante lo sviluppo. Abbiamo implementato un *modello di sviluppo* chiamato **TDD** (*Test-Driven-Development*) che viene spesso implementato perché di facile comprensione e molto intuitivo. Tutto ciò ha permesso di effettuare test in modo da verificare se le classi da noi create fossero valide ed utilizzabili.

All'interno del package **com.unofx.model** è presente una cartella che comprende le seguenti classi di test:

- **BotPlayerTest**
- **UserPlayerTest**
- **DeckTest**
- **TableTest**

Il **quinto passo** è stato sviluppare un modello in grado di funzionare regolarmente, permettendo quindi una partita completa anche senza l'implementazione della grafica.

Il **sesto passo** è stato studiare e implementare l'interfaccia grafica. Per creare un gioco il più realistico possibile, abbiamo scelto di implementare la **JavaFx**; inizialmente questo ci ha rallentati nella nostra tabella di marcia, ma, dopo aver approfondito le nostre conoscenze, siamo riusciti ad implementare l'interfaccia grafica al modello che avevamo creato in precedenza.

3.3. Note di sviluppo

In questo paragrafo vengono mostrate la distribuzione delle **funzionalità di Java** implementate da ogni componente del gruppo.

Palumbo Giovanni:

- Collections:
 - **UserPlayer**
- Interface:
 - **Card**
 - **Table**
- Enum:
 - **Colour**
 - **Number**
 - **Caction**
- Stream:
 - **Deck**
 - **UserPlayer**
 - **TableImpl**
- Lamda Expressions:
 - **Deck**
 - **TableImpl**
- Pattern Singleton:
 - **TableImpl**

A livello algoritmico mi sono concentrato sulla realizzazione e implementazione delle seguenti classi:

- **Enum Colour**
- **Enum Number**
- **Enum Caction**
- **NormalCard**
- **UserPlayer**
- **Deck**
- **TableImpl**
- **ControllerScene**

A livello grafico ho curato l'implementazione della schermata principale, della schermata delle regole del gioco e del tavolo da gioco, compresa la creazione delle carte da gioco tramite Adobe Illustrator e Adobe Photoshop.

Cecchini Diego:

- Collections:
 - **BotPlayer**
- Interface
 - **Player**
 - **Table**
- Enum:
 - **Colour**
 - **Number**
 - **Caction**
- Stream:
 - **BotPlayer**
 - **Deck**
 - **TableImpl**
- Lambda Expressions:
 - **Deck**
 - **TableImpl**
 - **Botplayer**
 - **UserPlayer**
- Pattern Singleton:
 - **TableImpl**

A livello algoritmico mi sono concentrato sulla realizzazione delle seguenti classi:

- **Enum Colour**
- **Enum Number**
- **Enum Caction**
- **ActionCard**
- **BotPlayer**
- **Deck**
- **TableImpl**
- **GameController**

A livello grafico ho curato l'implementazione della schermata principale e del tavolo da gioco, le animazioni, la disposizione delle carte e i suoni del gioco.