

# Goose Game

Relazione per il progetto di  
Programmazione ad Oggetti

Anno Accademico 2024/2025



**Componenti del gruppo:**

Giuseppe Fusco  
Lucia Pola  
Giada Tedaldi  
Samuele D'Ambrosio  
Matteo Crepaldi

10 Giugno 2025

# Indice

<b>1</b>	<b>Analisi</b>	<b>3</b>
1.1	Requisiti	4
1.2	Modello del dominio	12
<b>2</b>	<b>Design</b>	<b>14</b>
2.1	Architettura	14
2.2	Design dettagliato	16
2.2.1	Giuseppe Fusco	16
2.2.2	Lucia Pola	19
2.2.3	Giada Tedaldi	23
2.2.4	Samuele D'Ambrosio	27
2.2.5	Matteo Crepaldi	32
<b>3</b>	<b>Sviluppo</b>	<b>36</b>
3.1	Testing automatizzato	36
3.1.1	Giuseppe Fusco	36
3.1.2	Lucia Pola	36
3.1.3	Giada Tedaldi	37
3.1.4	Samuele D'Ambrosio	37
3.1.5	Matteo Crepaldi	37
3.2	Note di sviluppo	37
3.2.1	Giuseppe Fusco	37
3.2.2	Lucia Pola	38
3.2.3	Giada Tedaldi	38
3.2.4	Samuele D'Ambrosio	39
3.2.5	Matteo Crepaldi	39
<b>4</b>	<b>Commenti finali</b>	<b>40</b>
4.1	Autovalutazione e lavori futuri	40
4.1.1	Giuseppe Fusco	40
4.1.2	Lucia Pola	40
4.1.3	Giada Tedaldi	41

4.1.4	Samuele D'Ambrosio	41
4.1.5	Matteo Crepaldi	42
<b>A</b>	<b>Guida utente</b>	<b>43</b>
<b>B</b>	<b>Esercitazioni di laboratorio</b>	<b>56</b>

# Capitolo 1

## Analisi

Il gruppo si pone come obiettivo quello di realizzare una versione modificata del *Gioco dell’Oca*. Il genere è un tipico gioco da tavola: lungo il percorso i giocatori, che possono essere da un minimo di 2 a un massimo di 4, si possono imbattere in sfide, quali possono essere i minigiochi, e penalità. L’obiettivo è quello di raggiungere la cella finale prima degli altri concorrenti tramite il lancio di due dadi.



Figura 1.1: Tipico gioco da tavola “Gioco dell’Oca”

## 1.1 Requisiti

### Requisiti Funzionali

- **Schermata di Menù**

- All'avvio del gioco, viene visualizzata una schermata di menù.
- Il menù deve consentire ai giocatori di consultare il regolamento, iscriversi alla partita e avviare una nuova.

- **Configurazione della Partita**

- Prima di iniziare una partita, i giocatori devono inserire il proprio nickname per registrarsi alla partita.
- A ciascun giocatore viene assegnato un colore in modo casuale, utilizzato per identificarlo nel tabellone di gioco.

- **Gestione dei Turni**

- All'inizio della partita, l'ordine di gioco è determinato dalla sequenza di inserimento dei giocatori nel menù iniziale.
- I turni vengono gestiti in modo sequenziale.

- **Meccaniche di Gioco**

- Durante il proprio turno, il giocatore deve premere un pulsante per lanciare due dadi: il valore ottenuto determina il numero di celle di avanzamento all'interno del tabellone.
- A seconda della cella in cui il giocatore si ferma, si possono verificare i seguenti eventi:
  - \* *Stazionamento*: Nessun effetto.
  - \* *Minigioco Single-Player*: Il giocatore dovrà affrontare un minigioco. In caso di vittoria il giocatore ottiene una carta bonus, reperibile nella sua saccoccia, che gli permette di avanzare di tot celle; in caso di sconfitta, ottiene una penalità, immediatamente applicata, che lo fa retrocedere di tot celle.

### Fine Partita

- L'obiettivo del gioco è raggiungere la cella finale prima degli altri giocatori.

- Per concludere la partita, il giocatore deve ottenere con il lancio dei dadi un valore esatto per raggiungere la cella finale; in caso contrario, i punti eccedenti verranno percorsi all'indietro.
- La partita termina quando almeno un giocatore raggiunge la cella finale.
- Al termine della partita, viene mostrata la classifica finale, determinata in base alla distanza dei giocatori dalla cella finale: più un giocatore è vicino al punto di arrivo, più alta sarà la sua posizione in classifica.

## Minigiochi

### Tris (Tic-Tac-Toe) - Pola

- Il gioco si basa su una griglia 3x3 in cui utente e CPU si sfidano alternando le proprie mosse: il giocatore umano utilizza il simbolo “X”, mentre il computer gioca con il simbolo “O”.
- A ogni turno, il giocatore seleziona una casella vuota della griglia per posizionare il proprio simbolo: inizia sempre per primo l'utente, seguito dal computer.
- Ogni round termina quando:
  - Un giocatore ha completato una linea di tre simboli uguali (orizzontale, verticale o diagonale).
  - Tutte le caselle sono piene e nessuno ha vinto (pareggio).
- La partita segue la modalità “al meglio dei 3”, quindi termina finiti i 3 round o quando uno dei due giocatori raggiunge le 2 vittorie.

### Puzzle - Pola

- Il gioco si basa su una griglia 5x5 composta da tessere numerate che rappresentano un'immagine ordinata.
- Al primo avvio, l'immagine viene mostrata nella sua configurazione corretta per permettere al giocatore di visualizzarla.
- Premuto il pulsante start, le tessere vengono mescolate casualmente e parte un timer di 2 minuti e 30 secondi.
- Il giocatore può selezionare due caselle per scambiarne il contenuto, con l'obiettivo di ricomporre l'immagine originale entro il tempo concesso.

- La partita termina quando l'immagine viene ricostruita correttamente prima dello scadere del tempo (vittoria) o quando il tempo termina senza che l'immagine sia stata completata (sconfitta).

### **Memory - Fusco**

- Si tratta di un gioco di memoria strutturato su una griglia 4x4, composta da 16 carte totali, che formano 8 coppie identiche.
- Tutte le carte inizialmente sono coperte, nascondendo l'immagine sottostante.
- Durante il proprio turno, il giocatore può scoprire due carte alla volta, cercando di individuare una coppia uguale.
- Se le due carte selezionate mostrano la stessa immagine, rimangono scoperte sul tavolo di gioco.
- In caso contrario, le carte tornano coperte al momento della selezione di una terza carta.
- L'obiettivo è quello di ricordare la posizione delle immagini e riuscire a trovare tutte le coppie.

### **Snake - Fusco**

- È una rivisitazione del celebre gioco arcade Snake, dove il giocatore controlla un serpente virtuale che si muove all'interno di una griglia.
- Il serpente avanza in automatico in una direzione, ma può essere guidato dal giocatore nei quattro versi cardinali: su, giù, sinistra e destra.
- All'interno della griglia compare del cibo (rappresentato da un quadratino rosso, simile a una mela), che il serpente deve raggiungere e “mangiare”.
- Ogni volta che il serpente consuma un pezzo di cibo, aumenta di lunghezza, rendendo i movimenti sempre più complessi.
- Il gioco termina quando il serpente:
  - Urta contro i bordi della griglia;

- Oppure si scontra con il proprio corpo, provocando un'autocolli-sione.
- Lo scopo finale è raggiungere uno score pari a 15, evitando ostacoli e gestendo lo spazio a disposizione con attenzione.

### **RockPaperScissors - Tedaldi**

- Ispirato al classico gioco della Morra Cinese, RockPaperScissors consente al giocatore di sfidare un avversario digitale in una serie di round.
- Ad ogni turno, il giocatore può selezionare una delle tre mosse disponibili: Rock, Paper o Scissors. L'avversario digitale, invece, effettua la propria scelta in modo casuale
- La logica su cui si basa una vittoria è la seguente:
  - Rock batte Scissors.
  - Paper batte Rock.
  - Scissors batte Paper.
- Sistema di punteggio:
  - Se il giocatore vince, ottiene 1 punto.
  - Se il bot vince, ottiene 1 punto.
  - Se pareggio (stesso simbolo), nessuno ottiene punti.

- Dopo ogni round viene mostrato lo score aggiornato.
- Lo scopo finale è raggiungere per primo i 3 punti. uno dei due giocatori arriva a 3 punti termina la partita e viene mostrato un banner finale con l'annuncio del vincitore.

### **Hangman - Tedaldi**

- Ispirato al gioco dell'impiccato, Hangman mette alla prova la capacità del giocatore di indovinare una parola nascosta, scelta casualmente da un file di parole predefinito.

- All'avvio della partita, una parola viene selezionata in modo automatico.
- La parola viene visualizzata in formato mascherato, dove:
  - Le vocali sono rappresentate con il simbolo “+”
  - Le consonanti sono rappresentate con il simbolo “\_”
- Il giocatore ha a disposizione un massimo di 5 tentativi per indovinare la parola completa.
- A ogni tentativo, il giocatore può proporre una singola lettera. Se la lettera è presente nella parola, viene rivelata nella posizione corretta; in caso contrario, il numero di tentativi rimanenti viene decrementato.
- Ad ogni errore, viene progressivamente disegnata un'immagine stilizzata dell'impiccato, aumentando la tensione visiva e rappresentando graficamente il rischio di fallimento.
- Una volta selezionata una lettera il bottone si disattiva e non potrà più essere scelta.
- Il gioco termina quando:
  - Il giocatore indovina tutte le lettere della parola → Vittoria
  - Il numero di tentativi errati raggiunge 5 → Sconfitta
- Al termine della partita viene mostrato un messaggio finale con l'esito e la parola corretta.

### **HerdingHound - D'Ambrosio**

- Il gioco prende ispirazione dal genere stealth in cui il giocatore deve evitare di essere visto da nemici con un campo visivo definito. In particolare, il concetto di visibilità limitata, la gestione degli stati di allerta del cane e la presenza di zone d'ombra che permettono di nascondersi, rappresentano elementi chiave mutuati dal genere stealth, adattati in un contesto puzzle su griglia a turni.
- L'obiettivo del giocatore è guidare l'oca dalla posizione iniziale in alto a sinistra fino alla posizione finale in alto a destra, percorrendo esclusivamente il bordo della griglia, senza essere visti dal cane e senza esaurire il tempo a disposizione.

- All'avvio del gioco viene mostrato un countdown di 3 secondi, durante il quale la griglia è visibile.
  - Serve per analizzare il campo di gioco e capire se l'oca è già inizialmente in una zona a rischio.
- Dopo il countdown iniziale, parte un timer di 60 secondi.
  - Se il tempo scade prima che l'oca raggiunga l'arrivo, il gioco termina con una sconfitta.
- Il cane ha tre stati:
  - Dormiente: Il cane non rappresenta un pericolo: l'oca può muoversi liberamente.
  - Allerta: Uno stato intermedio che anticipa il risveglio: serve da avviso per prepararsi.
  - Sveglio: Il cane guarda in una direzione determinata dal movimento dell'oca.
- Zone d'Ombra:
  - Le ombre si trovano dietro le scatole rispetto al punto di vista del cane. Se l'oca si trova in un'ombra quando il cane è sveglio, non viene individuata.
- Movimento dell'Oca:
  - L'oca si muove una casella alla volta premendo la barra spaziatrice. Il percorso è vincolato ai bordi della griglia: parte da in alto a sinistra → va in basso → a destra → in alto → fino all'angolo in alto a destra.
- Condizione di Vittoria: Raggiungere la casella finale in alto a destra entro 60 secondi, senza mai essere vista dal cane.
- Condizioni di sconfitta: Essere visti dal cane durante lo stato di veglia al di fuori delle zone d'ombra. Superare il limite di un minuto senza aver completato il percorso.

## **HonkMand - D'Ambrosio**

- Il gioco si ispira al celebre gioco di memoria Simon Game.
- Il gioco allena attenzione e memoria visiva.
- L'obiettivo è ripetere correttamente le sequenze mostrate dal sistema per 5 round consecutivi, con sequenze sempre nuove e progressivamente più lunghe.
- Il gioco inizia quando si preme il pulsante "Play".
  - Dopo l'avvio, i pulsanti colorati vengono disabilitati per evitare input accidentali durante la fase di osservazione.
- Fasi del Gioco:
  - Fase del Sistema (memorizzazione): Viene generata una nuova sequenza casuale (non cumulativa) composta da un numero di colori pari al numero del round attuale (es. 1 colore al round 1, 2 colori al round 2, ecc.). La sequenza viene mostrata tramite un cerchio evidenziato intorno ai pulsanti.
  - Fase del Giocatore (ripetizione): Al termine della sequenza, i pulsanti si riattivano. Il giocatore deve ripetere esattamente la sequenza appena vista. Se la sequenza è corretta, si passa al round successivo. Se si sbaglia, il gioco termina con una sconfitta.
- Il gioco è composto da 5 round in totale:
  - A ogni round:
    - \* La sequenza è completamente nuova (non cumulativa come nel Simon Game originale).
    - \* La lunghezza della sequenza aumenta di una unità (da 1 a 5).
  - Condizione di Vittoria: Ripetere correttamente tutte e 5 le sequenze, una per ciascun round.
  - Condizione di Sconfitta: Sbagliare anche solo un colore durante la ripetizione della sequenza in qualsiasi round.

### **Click the color - Crepaldi**

- Ispirato ai giochi di test dei riflessi, testando le capacità di reazione del giocatore.
- All'inizio della partita viene scelto uno dei quattro bottoni visibili dall'interfaccia, che dovrà essere clickato dal giocatore
- Successivamente, ogni tick (0.1 secondi) il gioco "decide" casualmente se attivare il bottone o meno. Per motivi di convenienza l'attesa massima del giocatore è stata limitata a 5 secondi, dopo i quali viene forzata l'attivazione del bottone.
- Quando il bottone viene attivato, si entra nella fase di attesa del giocatore, dalla durata massima di 7 tick (0.7 secondi) entro i quali l'utente deve puntare e clickare il bottone che è stato attivato.
- Una volta che l'utente clicka il bottone, a seconda dello stato attuale del gioco, possono verificarsi quattro situazioni:
  - Un bottone è stato attivato e il giocatore clicka quello giusto, il punteggio viene aumentato di 10 punti
  - Un bottone è stato attivato e il giocatore clicka quello sbagliato, il punteggio viene diminuito di 5 punti
  - Un bottone è stato attivato, ma il giocatore non riesce a clickarlo entro il tempo limite, il punteggio viene diminuito di 10 punti
  - Nessun bottone è attivo, il punteggio viene diminuito di 5 punti
- Una volta che il giocatore clicka il bottone giusto, o il tempo a sua disposizione si esaurisce, viene automaticamente iniziato un nuovo round.
- Il processo si ripete per dieci round, durante i quali il giocatore deve accumulare un totale di almeno 25 punti per vincere il minigioco

### **Three Cups Game - Crepaldi**

- Ispirato al classico gioco delle tre coppette, il giocatore deve quindi indovinare sotto quale coppetta si trova la biglia
- All'inizio della partita viene decisa una coppetta sotto la quale si trova la biglia.

- Il giocatore deve quindi scegliere una delle tre coppette.
- Una volta clickata la coppetta scelta viene verificato se la scelta del giocatore coincide con la scelta del programma.
- In caso di scelta corretta il punteggio del giocatore viene incrementato di uno, altrimenti viene lasciato invariato
- Una volta che il giocatore sceglie la coppetta, viene mostrata la coppetta sotto la quale era presente la biglia. Dopo averla mostrata le coppette vengono mescolate e si attende la scelta del giocatore
- Terminato il decimo round, se il giocatore ha preso almeno tre scelte corrette, il minigioco viene vinto.

### Requisiti Non Funzionali

- Il gioco deve funzionare sui tre principali sistemi operativi: Windows, Linux, macOS.
- La finestra di gioco deve essere ridimensionabile.
- Il gameplay deve essere scorrevole, garantendo una transizione fluida tra i turni e il passaggio dal tabellone ai minigiochi.

## 1.2 Modello del dominio

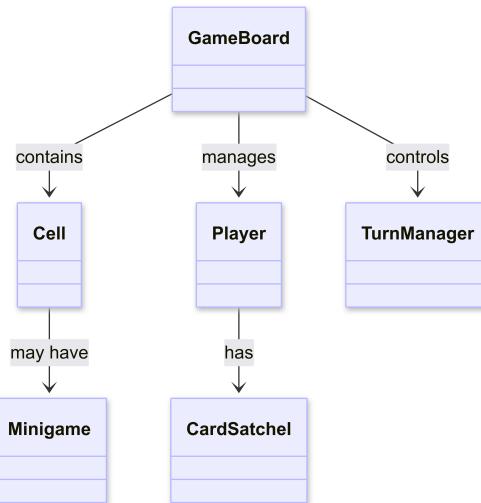


Figura 1.2: Schema UML del dominio

Il modello del dominio del gioco "Gioco dell'Oca" si basa su sei entità principali che rappresentano gli elementi fondamentali della partita:

- **Player**: rappresenta un giocatore, che ha un nome, un colore per l'identificazione e una posizione sul tabellone. Ogni giocatore può anche possedere delle carte bonus, custodite nella sua saccoccia.
- **CardSatchel**: è la "saccoccia" di ogni giocatore, nella quale sono contenute le carte bonus che possono fornire vantaggi durante la partita. Non contiene carte malus, poiché queste vengono applicate immediatamente al giocatore.
- **GameBoard**: è il tabellone del gioco e funge da vero e proprio coordinatore. Contiene il percorso di celle e tiene traccia delle posizioni dei giocatori e del loro avanzamento.
- **Cell**: rappresenta una singola cella del percorso. Ogni cella può avere effetti diversi: alcune attivano un minigioco per il giocatore che vi si ferma, altre invece non hanno alcun effetto.
- **Minigame**: è una sfida che può essere attivata quando un giocatore si ferma su una cella speciale. A seconda dell'esito del minigioco (vittoria o sconfitta), il giocatore può ottenere carte bonus o malus.
- **TurnManager**: gestisce l'ordine dei turni di gioco, determinando chi è il prossimo giocatore a lanciare i dadi.

# Capitolo 2

## Design

### 2.1 Architettura

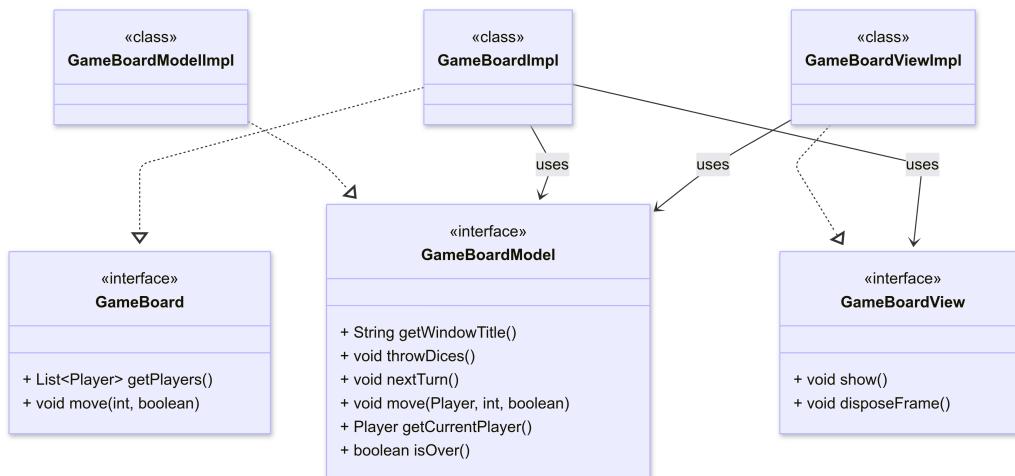


Figura 2.1: Schema UML dell’implementazione del pattern MVC

Per mantenere una chiara separazione tra la logica di gioco, la presentazione grafica e il coordinamento dei vari eventi, abbiamo deciso di adottare il pattern architettonale MVC (Model-View-Controller). Nel nostro progetto , ogni componente svolge un ruolo ben definito:

- **GameBoardModel**: racchiude lo stato della partita e le regole fondamentali.
- **GameBoardView**: si occupa esclusivamente di mostrare all’utente l’evolversi del gioco.

- **GameBoard**: fa da punto di incontro tra gli input dell’utente, la logica di gioco e l’aggiornamento dell’interfaccia.

All’interno del Model abbiamo una classe dedicata che gestisce tutte le operazioni di gioco: dall’aggiornamento delle posizioni dei pedoni al controllo delle condizioni di vittoria, passando per l’avvio e la verifica dei minigiochi e la gestione delle carte bonus o malus. In sostanza, qui si concentra ogni regola e ogni modifica di stato, senza preoccuparsi di come questi cambiamenti verranno visualizzati. La View, invece, si limita a costruire l’interfaccia: un pannello che riproduce il tabellone e una serie di pulsanti per le azioni di base. Non contiene alcuna logica di gioco, ma si occupa soltanto di mostrare i componenti grafici e di intercettare i click dell’utente. Quando un giocatore arriva su una casella che prevede un minigame, la View si incarica di aprire la finestra corrispondente e, al termine dell’attività, segnala al Controller il risultato ottenuto. Il Controller, da parte sua, è l’unico elemento che conosce sia il Model sia la View. Al momento dell’avvio, provvede a creare l’intero tabellone, assegnare a ogni giocatore la propria “saccoccia” di carte e collegare tutti i componenti. Durante la partita, tutti gli input raccolti dalla View (per esempio la richiesta di muovere un pedone o di iniziare un mini-gioco) vengono elaborati dal Controller, che delega al Model le operazioni necessarie. Ogni volta che il Model cambia stato—ad esempio perché un giocatore si è spostato, ha vinto o ha perso un minigame, oppure ha pescato una carta—il Controller ordina alla View di aggiornare la grafica. In questo modo, il Model rimane responsabile di tutte le decisioni e di ogni regola, la View si occupa soltanto di mostrare quanto richiesto, e il Controller fa da tramite, assicurando che gli input dell’utente vengano trasformati in azioni sul Model e che ogni modifica di stato venga riflesso correttamente nella View. In definitiva, l’architettura MVC assicura una netta distinzione tra “cosa fa il gioco” e “come il gioco viene presentato”.

## 2.2 Design dettagliato

### 2.2.1 Giuseppe Fusco

#### Snake

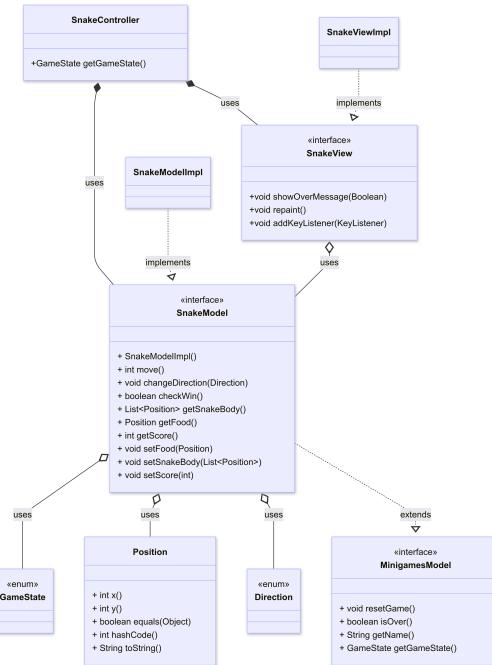


Figura 2.2: Schema UML del mini game Snake

**Problema:** gestione continua del movimento del serpente, del cibo e del punteggio, mantenendo il gioco reattivo e aggiornato in tempo reale.

**Soluzione:** La soluzione adotta una separazione netta tra logica e presentazione. Il controller (`SnakeController`) coordina il flusso tra il modello (`SnakeModelImpl`) e la vista (`SnakeView`). Il modello implementa l'interfaccia `SnakeModel` (che estende `MinigamesModel`), fornendo metodi per gestire direzione, cibo, corpo del serpente e stato del gioco. Il controller si occupa di elaborare l'input dell'utente (es. cambio direzione) e aggiornare la view sulla base dello stato restituito dal modello. La view si limita a visualizzare i dati ricevuti. Il design utilizza interfacce, enum (`Direction`, `GameState`) e oggetti di supporto (`Position`) per mantenere il codice modulare e chiaro. Questa architettura permette di sostituire facilmente la

logica interna senza impattare su controller o view.

## Memory

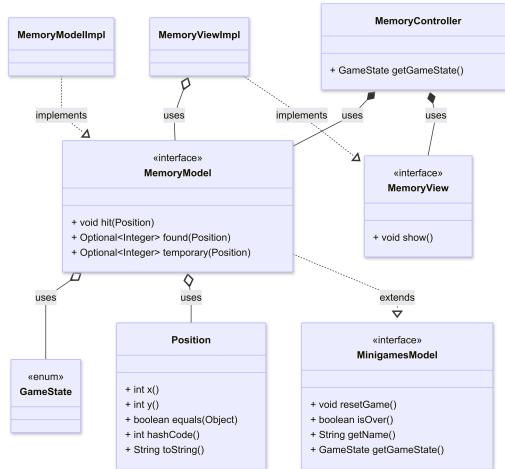


Figura 2.3: Schema UML del mini game Memory

**Problema:** implementazione del gioco “Memory” e utilizzo del pattern architetturale MVC

**Soluzione:** La soluzione implementata si articola attraverso un chiaro modello MVC (Model-View-Controller), che separa la logica di gioco (**MemoryModel**), la visualizzazione (**MemoryView**) e il coordinamento generale (**MemoryController**). Il modello si occupa di assegnare i valori alle celle, gestire le selezioni e verificare le corrispondenze. Il controller inizializza e avvia il gioco coordinando model e view, senza interferire con la loro logica interna. Grazie alla struttura modulare, la logica del gioco può essere testata o estesa indipendentemente dalla GUI, facilitando la manutenzione del software. La gestione dell’interazione avviene in modo coerente: ogni click su una cella viene trasmesso al modello, che aggiorna lo stato, e la vista si occupa del ridisegno.

## FinalBoard

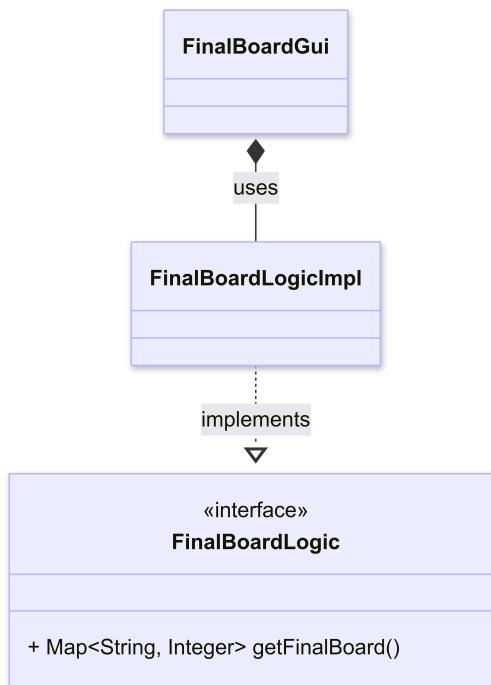


Figura 2.4: Schema UML del tabellone finale

**Problema:** fornire una classifica finale in base alle posizioni nel tabellone. Gestione del codice per favorire estensibilità e riusabilità.

**Soluzione:** Per la schermata finale è stato adottato un approccio che separa chiaramente logica e interfaccia. La classe **FinalBoardGui** si occupa solo della visualizzazione, mentre la logica del punteggio è gestita dall'interfaccia **FinalBoardLogic** e dalla sua implementazione **FinalBoardLogicImpl**. La GUI interroga l'interfaccia per ottenere i dati da mostrare, senza accedere direttamente all'implementazione concreta. Questa scelta, più modulare rispetto all'unificazione in una sola classe, permette maggiore riuso e una futura estensibilità.

Dal punto di vista progettuale, si applica il pattern *Strategy*: la GUI utilizza l'interfaccia come strategia per accedere ai dati, permettendo di sostituire facilmente la logica senza modificare la presentazione. Anche se l'approccio introduce un po' di complessità in più, garantisce chiarezza, testabilità e un buon livello di disaccoppiamento.

## 2.2.2 Lucia Pola

### Tris(Tic-Tac-toe)

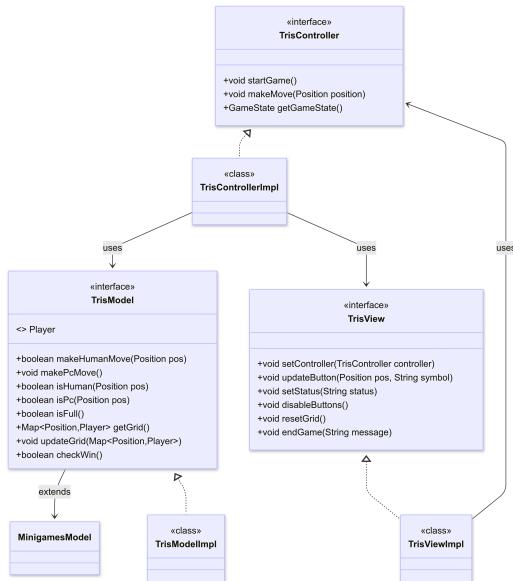


Figura 2.5: Schema UML del mini game Tris(Tic-Tac-Toe)

**Problema:** scelta e gestione del pattern da utilizzare per implementare il mini game Tris.

**Soluzione:** per l'implementazione del mini game Tris(Tic-Tac-Toe) alla fine ho scelto di utilizzare il pattern MVC, perché più diretto e immediato. `TrisModelImpl` è il cuore della logica di gioco: mantiene lo stato della griglia, sa riconoscere vittorie, pareggi e quando la griglia è piena, e offre un metodo per resettare il gioco a inizio partita o di ogni nuovo round. `TrisController` fa da “collante” tra modello e interfaccia utente: riceve la mossa umana, la passa al model, aggiorna la view e quindi scatena la risposta del computer. Tiene anche il conto delle vittorie dell’umano, del PC e gestisce la logica del best-of-3, disabilitando i pulsanti quando serve e pianificando il reset tra un round e l’altro attraverso l’utilizzo di un timer `Swing`. `TrisView`, realizzata con `Swing`, è semplicemente una finestra con una griglia di  $3 \times 3$  pulsanti e un’etichetta di stato. Invia al controller i

click dell’utente e si aggiorna su comando, ad esempio per mostrare “X” o “O”, disabilitare la griglia, resettarla o chiudere la finestra con il punteggio finale(solo una volta terminati i 3 round).

## Puzzle

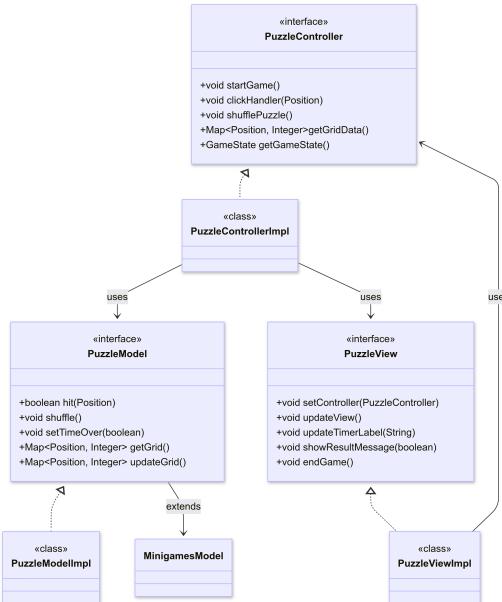


Figura 2.6: Schema UML del mini game Puzzle

**Problema:** scelta e gestione del criterio di vittoria nel mini game, soprattutto considerando che si tratta di un puzzle e non di un gioco a turni classico.

**Soluzione:** inizialmente non sapevo come gestire il criterio di vittoria, dato che nel puzzle non c’è una vittoria “istantanea” come nei giochi a turni, ma una condizione finale da riconoscere solo quando il puzzle è completamente risolto. Dopo aver valutato diverse opzioni, ho deciso di far rilevare la vittoria direttamente dal model **PuzzleModelImpl**, che controlla se la configurazione attuale corrisponde a quella finale. Tuttavia, per gestire in modo fluido il passaggio dalla vittoria alla fine del gioco, ho scelto di utilizzare un timer gestito dal controller **PuzzleControllerImpl**. In questo modo ho mantenuto ben separati i compiti secondo l’architettura MVC: il model si occupa della logica di verifica della vittoria, il controller gestisce la temporizzazione e la coordinazione, mentre la view si limita alla rappresentazione visiva e all’interazione con l’utente.

## TurnManager

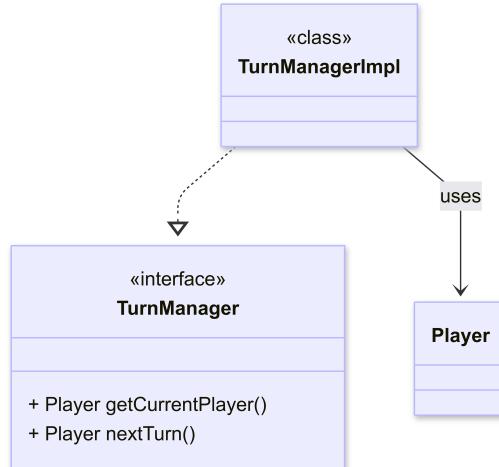


Figura 2.7: Schema UML della gestione dei turni

**Problema:** gestione dell'ordine dei turni tra più giocatori in modo generico, sicuro e facilmente estendibile.

**Soluzione:** per affrontare questo problema ho introdotto l'interfaccia **TurnManager**, che definisce i comportamenti essenziali per la gestione dei turni. L'implementazione concreta, **TurnManagerImpl**, incapsula completamente la logica di rotazione dei turni. Il costruttore accetta una lista di oggetti **Player** e garantisce che essa sia valida: in questo modo si assicura l'integrità del ciclo dei turni fin dalla creazione dell'oggetto. Il metodo **nextTurn** gestisce il passaggio al turno successivo mentre il metodo **getCurrentPlayer** restituisce il giocatore corrente. L'interfaccia **TurnManager** è pensata per essere indipendente dal contesto specifico di gioco. Questo permette di riutilizzarne l'implementazione in diversi giochi o sistemi che richiedono la gestione di turni ciclici tra più entità. Grazie alla chiara separazione delle responsabilità e alla robustezza dell'implementazione, è possibile estendere facilmente il comportamento del gestore dei turni (ad esempio per introdurre un'inversione dell'ordine o la rimozione temporanea di un giocatore), mantenendo la coerenza e la sicurezza dell'intero sistema.

## DoubleDice

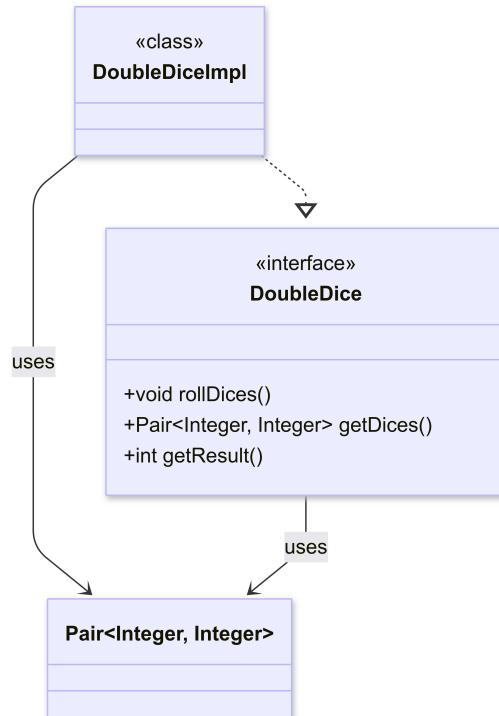


Figura 2.8: Schema UML della gestione del lancio dei dadi

**Problema:** dovevo progettare una componente che rappresentasse una coppia di dadi per il gioco, con la possibilità di lanciarli, ottenere i valori singoli e la loro somma, mantenendo il tutto semplice e coerente

**Soluzione:** per implementare la coppia di dadi `DoubleDice`, ho deciso di definire un'interfaccia chiara con tre metodi principali: uno per effettuare il lancio dei dadi `rollDices()`, uno per ottenere i valori singoli `getDices()` e uno per ottenere la somma dei due dadi `getResult()`. Nell'implementazione concreta `DoubleDiceImpl`, ho scelto di usare una classe `Pair<Integer, Integer>` per rappresentare lo stato dei due dadi in modo semplice e compatto. Per generare i valori casuali, ho utilizzato un'istanza di `Random`, da cui ottengo due numeri compresi tra 1 e 6, che rappresentano i risultati classici di un dado. Ho deciso di effettuare un lancio iniziale già nel costruttore, così da garantire che lo stato dei dadi sia sempre valido fin da subito. Ho mantenuto lo stato interno incapsulato, esponendo solo i metodi necessari per ottenere i valori e per lanciare i dadi, evitando modifiche esterne indesiderate. Questa scelta mi ha permesso di ottenere una classe

semplice, affidabile e facile da integrare nel resto del gioco, mantenendo ben separata la logica del lancio dei dadi dal resto dell'applicazione.

### 2.2.3 Giada Tedaldi

#### GameMenu

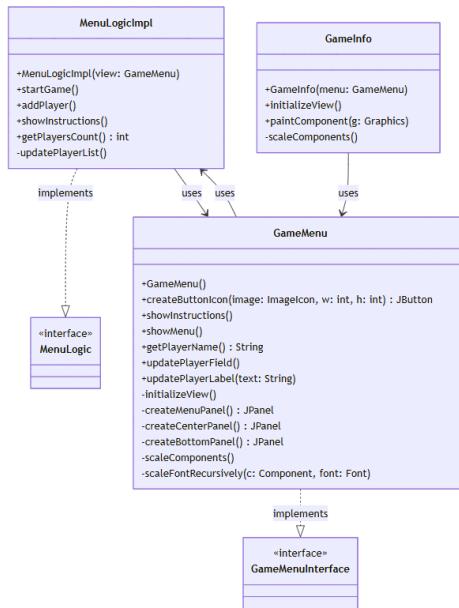


Figura 2.9: Schema UML del menu di gioco

**Problema:** Gestire dinamicamente l'aggiunta di giocatori e l'aggiornamento dell'interfaccia in modo modulare, mantenendo separata la logica di gestione dal codice della vista, per garantire riusabilità e facilità di manutenzione.

**Soluzione:** Ho adottato una separazione tra la logica del menu **MenuLogicImpl**, che gestisce i giocatori, i controlli e i vincoli (massimo giocatori, nomi duplicati), e la vista **GameMenu**, che si occupa solo della rappresentazione grafica e dell'interazione utente. La vista comunica con la logica tramite metodi dedicati, mantenendo il codice pulito e facilmente estendibile. Inoltre, la GUI si adatta dinamicamente alle dimensioni della finestra, migliorando l'esperienza utente.

## RockPaperScissors



Figura 2.10: Schema UML del mini game RockPaperScissors

**Problema:** Necessità di strutturare il programma in modo modulare e manutenibile, evitando di accorpore logica, interfaccia e gestione degli eventi in un'unica classe, garantendo al contempo una gestione asincrona dell'interazione dell'utente, in modo che il sistema fornisca una risposta immediata, mostrando le scelte, l'esito del round e l'aggiornamento dei punti.

**Soluzione:** Per affrontare i problemi ho deciso di adottare il pattern architettonico MVC. Questo mi ha permesso di separare:

- **Logica di gioco:** RockPaperScissorsModelImpl gestisce il punteggio, genera le mosse casuali del computer, determina il vincitore di ogni round e controlla quando un giocatore raggiunge la vittoria.
- **Interfaccia grafica:** RockPaperScissorsViewImpl rappresenta la vista dell'applicazione, realizzata con Swing. Mostra i punteggi aggiornati, le scelte fatte, i tre pulsanti per selezionare la propria mossa, ecc. Inoltre, gli elementi si adattano dinamicamente al ridimensionamento della finestra, garantendo un'interfaccia responsive.
- **Gestione degli eventi e del flusso:** RockPaperScissorsController funziona da collante tra il modello e la view. Riceve le azioni dell'utente, calcola il turno aggiornando il modello, aggiorna l'interfaccia con le scelte grafiche opportune, mostra l'esito del round e gestisce la fine della partita.

## Hangman

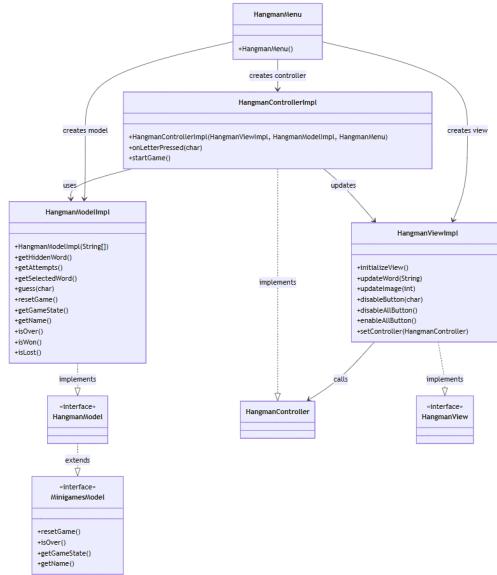


Figura 2.11: Schema UML del mini game Hangman

**Problema:** Scelta del pattern architetturale più adatto per evitare di accoppare logica di gioco, interfaccia grafica e gestione degli eventi in un'unica classe; Inoltre, le interfacce grafiche dinamiche richiedono un continuo aggiornamento e ridimensionamento degli elementi visivi per garantire un'esperienza fluida e coerente su diverse dimensioni dello schermo.

**Soluzione:** Per risolvere i problemi ho adottato il pattern architetturale MVC, che consente una chiara separazione delle responsabilità:

- **Logica di gioco:** HangmanModelImpl gestisce la parola segreta, i tentativi rimasti, verifica le lettere indovinate e determina la vittoria o la sconfitta.
- **Interfaccia grafica:** HangmanViewImpl implementa la vista con Swing, mostrando la parola nascosta, l'immagine dell'impiccato aggiornata in base agli errori e la tastiera virtuale. L'interfaccia si ridimensiona in modo responsivo adattandosi alla finestra (utilizzo di metodi di scaling dinamico per font e componenti).
- **Gestione eventi e flusso:** HangmanControllerImpl coordina modello e vista, riceve gli input dell'utente, aggiorna lo stato del gioco e modifica la vista di conseguenza.

## MinigameMenu

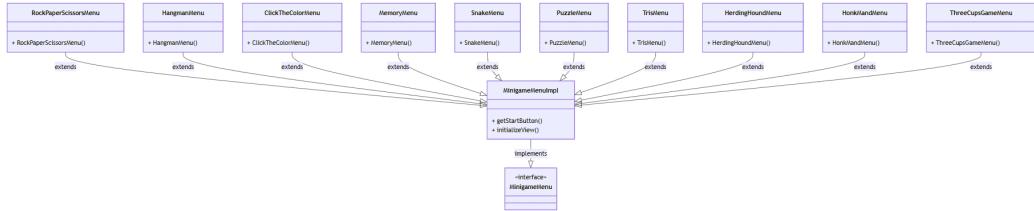


Figura 2.12: Schema UML del menu dei mini games

**Problema:** La necessità di creare un’interfaccia coerente e riutilizzabile per i menu dei mini-giochi, evitando la duplicazione di codice per ogni singola implementazione. Ogni mini-gioco richiedeva un menu con caratteristiche simili: pulsante di avvio, gestione delle informazioni e adattamento dinamico all’interfaccia grafica. Scrivere da zero queste componenti per ogni gioco avrebbe comportato codice ridondante e difficoltà di manutenzione.

**Soluzione:** Per evitare la ripetizione del codice e standardizzare la creazione dei menu, ho progettato una classe astratta `MinigameMenuImpl`, che funge da template comune per tutti i menu dei mini-giochi. Questa classe incapsula la logica di base per la costruzione del menu, inclusi il layout grafico, la gestione degli eventi, la visualizzazione dinamica con immagini di sfondo scalabili, e pulsanti iconici per avvio e informazioni. Ogni specifico gioco può estendere questa classe e concentrarsi solo sulle logiche personalizzate, senza dover ridefinire gli aspetti comuni. Questo approccio promuove il riutilizzo del codice, migliora la coerenza visiva dell’applicazione e semplifica l’evoluzione dell’interfaccia nel tempo.

## 2.2.4 Samuele D'Ambrosio

### HerdingHound

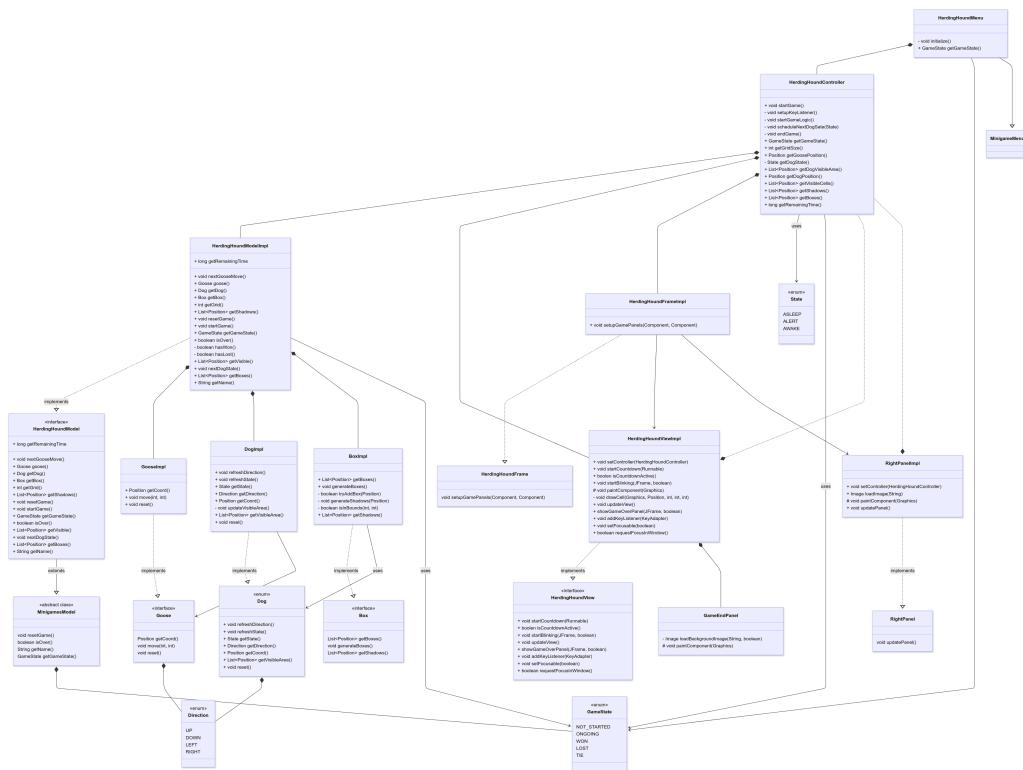


Figura 2.13: Schema UML del mini game HerdingHound

**Problema:** Durante lo sviluppo del gioco Herding Hound ho incontrato diversi problemi, sia strutturali che logici. La presenza di più oggetti con comportamenti diversi (oca, cane, scatole) richiede una gestione separata e ben organizzata delle logiche. In particolare, è stato complesso implementare le ombre generate dalle scatole in base alla posizione del cane, come se fosse una fonte di luce. Inoltre, era fondamentale far interagire correttamente questi elementi tra loro, far muovere l'oca lungo un percorso predefinito senza uscire dai bordi, e controllare se veniva vista dal cane escludendo le zone d'ombra.

**Soluzione:** Per organizzare il gioco in modo chiaro ho seguito il pattern MVC, dividendo il progetto in Model, View e Controller. Ho risolto questi problemi creando un Model diviso in più classi (GooseImpl, DogImpl,

BoxImp) ognuna con la propria responsabilità, e una classe HerdingHound-Model che le unisce tutte. La comunicazione tra queste avviene tramite il Controller che comunica anche con la View. La View si limita a disegnare gli elementi e ricevere gli input, mentre tutte le decisioni vengono prese dal Model. Per le ombre, ho sviluppato un algoritmo che le genera dietro alle scatole in base alla direzione rispetto al cane, tenendo conto di distanza, larghezza e lunghezza, limitandole ai bordi. Questo approccio è risultato il più logico e visivamente coerente, anche se ha richiesto molto tempo per trovare una soluzione efficace. Infine, il controller verifica se l'oca si trova nel campo visivo del cane (quando è sveglio) e non nascosta da un'ombra, determinando la sconfitta. Inoltre il gioco si serve di un GameEndPanel che varia in base a se il gioco è stato vinto o perso.

## HonkMand

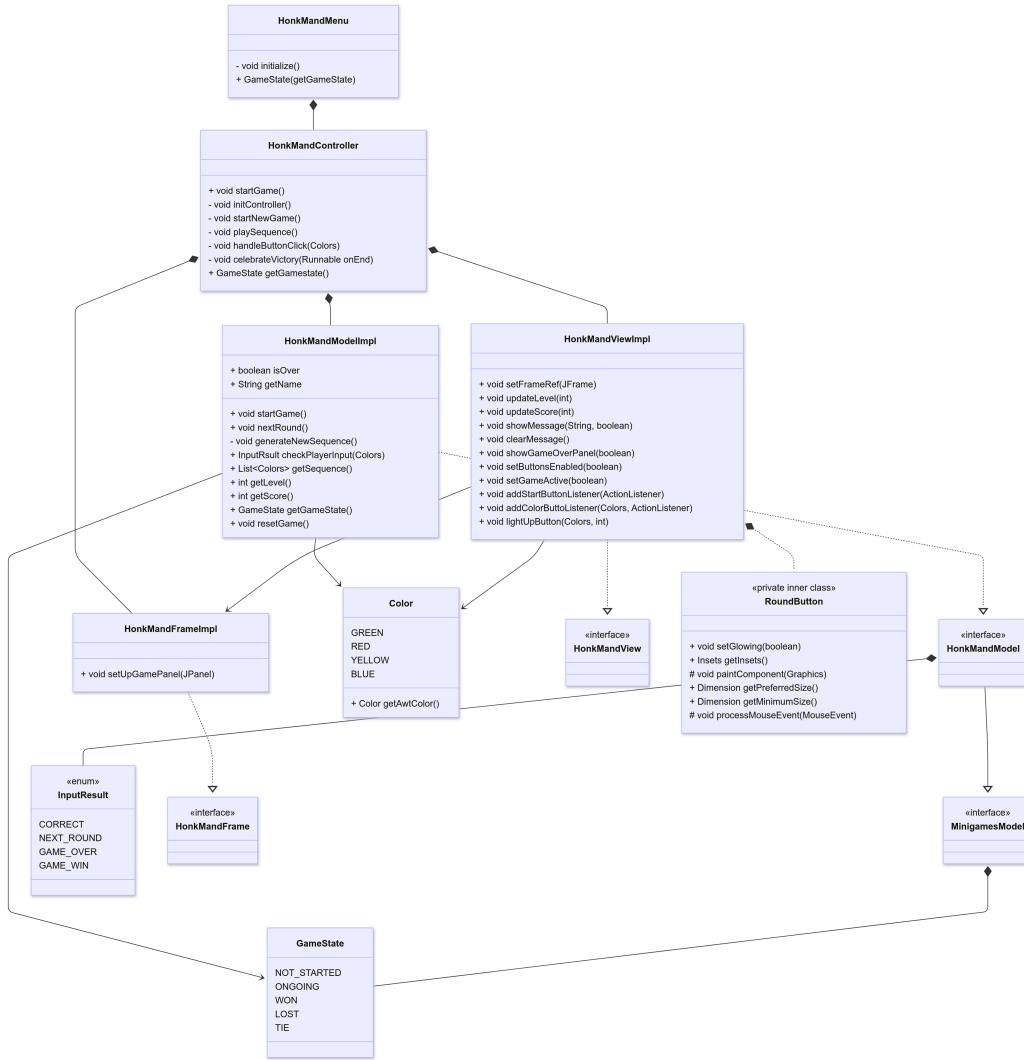


Figura 2.14: Schema UML del mini game HonkMand

**Problema:** Durante la realizzazione del gioco HonkMand ho incontrato diverse difficoltà legate alla gestione dei turni e della sequenza di colori da mostrare. Era importante distinguere chiaramente quando doveva agire il gioco e quando l'utente, mantenendo un comportamento coerente tra logica e interfaccia. Inoltre, serviva dare un buon feedback visivo per ogni pulsante e far sì che la sequenza venisse mostrata in modo ordinato, senza confondere il giocatore.

**Soluzione:** Per organizzare al meglio il progetto ho seguito l'architettura MVC, separando la logica del gioco dalla grafica. Ogni componente (Model, View e Controller) ha il suo ruolo ben definito, facilitando la gestione dei turni e dell'interazione. La sequenza viene mostrata con un breve effetto visivo e il giocatore può poi ripeterla. Inoltre il gioco si serve di un GameEndPanel che varia in base a se il gioco è stato vinto o perso. Questo approccio ha permesso di mantenere il gioco semplice da usare e visivamente chiaro, risolvendo i problemi iniziali di confusione tra fasi e feedback.

## CardSatchel

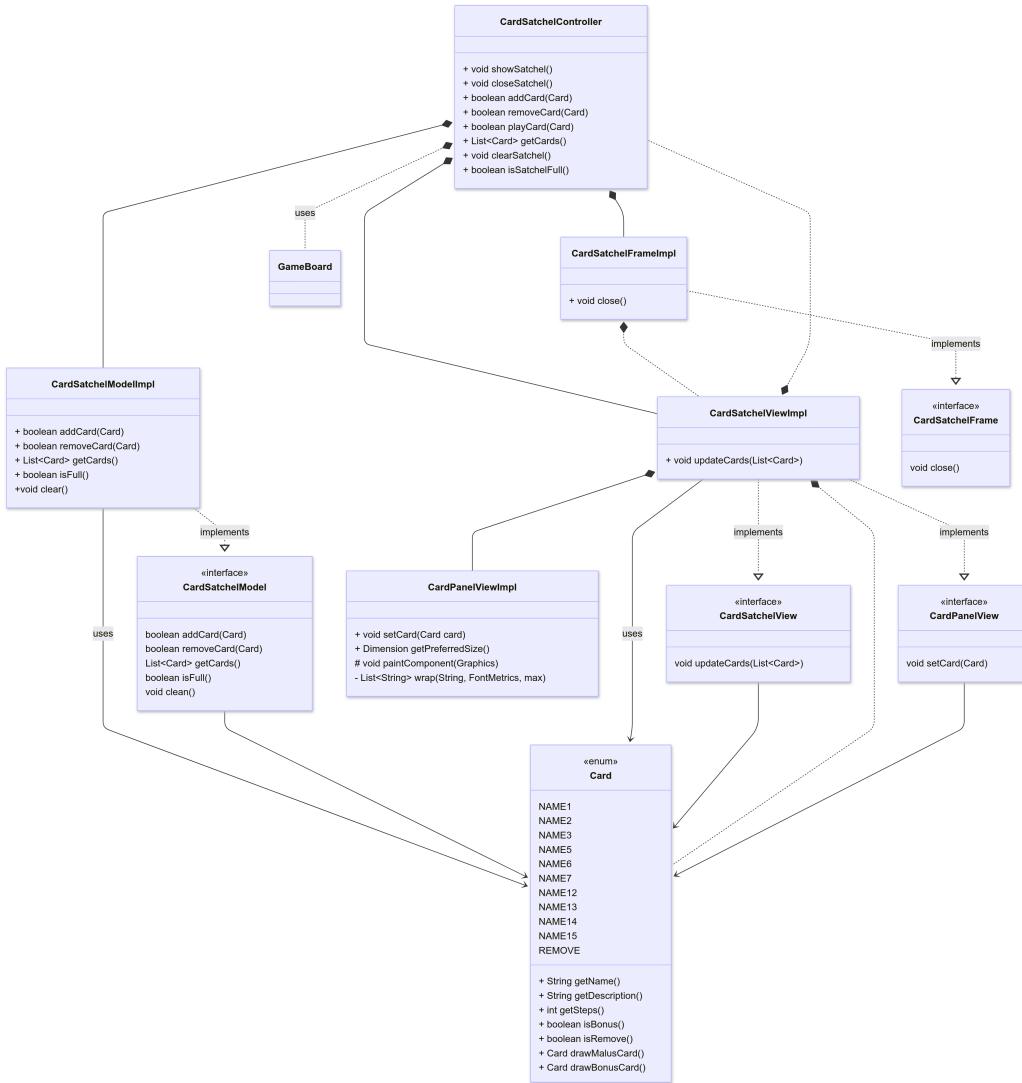


Figura 2.15: Schema UML del CardSatchel

**Problema:** CardSatchel è uno zaino virtuale pensato per contenere un insieme limitato di carte bonus da utilizzare all'interno di un gioco. Il problema principale era sviluppare un sistema che permettesse di gestire in modo efficiente e coerente l'aggiunta, la rimozione e l'uso delle carte, tenendo conto delle regole specifiche definite dall'enum Card. Era necessario gestire

limiti di capienza, e assicurarsi che le interazioni con lo zaino fossero semplici sia a livello logico (per il controller) che visivo (per l'utente finale).

**Soluzione:** Per risolvere questi problemi la classe è stata progettata con responsabilità chiare: gestire una collezione limitata di oggetti Card, offrendo metodi per aggiungere, rimuovere, visualizzare e selezionare le carte. La logica è completamente separata dalla vista e dall'interazione con l'utente: Card Satchel rappresenta solo il modello dati, rendendolo facilmente riutilizzabile e testabile. I controlli interni impediscono l'aggiunta di carte oltre il limite massimo o non compatibili con lo stato corrente. Il sistema prevede anche metodi per gestire effetti specifici (come la rimozione e l'aggiunta di tutte le carte), in modo che il controller o altri componenti del gioco possano usarli facilmente. La view del satchel fa uso di un CardPanelView per poter gestire singolarmente l'estetica e alcune funzionalità delle singole carte. Questa struttura modulare e pulita permette allo zaino di essere integrato in diversi contesti di gioco, senza dipendere direttamente dal funzionamento interno del gioco stesso. In questo modo si ottiene un alto grado di riusabilità e robustezza.

## 2.2.5 Matteo Crepaldi

### Click the color

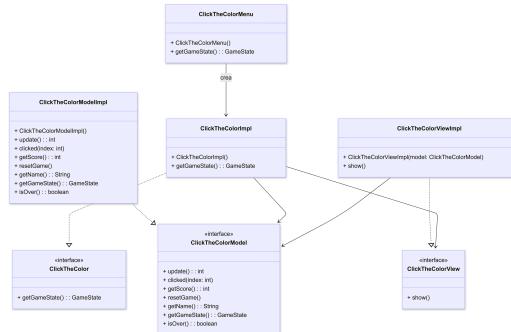


Figura 2.16: Diagramma UML del minigioco Click The Color

**Problema:** la natura stessa del gioco richiede che le tempistiche di comparsa dei bottoni non siano tutte uguali, per cui è necessario che il tempo di attesa da parte del giocatore sia diverso per ogni turno, quidni casuale.

**Soluzione:** inizialmente avevo deciso di strutturare la classe in modo da riuscire a comunicare tra le classi il momento in cui il bottone si sarebbe dovuto accendere tramite un oggetto LocalTime, dopo aver concluso che

un'architettura di questo tipo si dimostra troppo complessa e convoluta, ho optato per un sistema di "decisione" da parte di `ClickTheColorModelImpl`. Inizialmente optando per il metodo `Random.nextBoolean()`, dimostratosi però non efficace, in quanto la probabilità che il bottone impiegasse più di 4 tick (0.4 secondi) per attivarsi era quasi nulla. Appoggiandomi sul codice già scritto per implementare la soluzione che sfruttava `Random.nextBoolean()` ho deciso di utilizzare invece `Math.random()`, che restituisce un valore pseudorandom compreso tra 0 e 1.0, che verrà successivamente confrontato con un parametro deciso arbitrariamente, dopo aver effettuato alcuni test sulle tempistiche, impostato a 0.01. Impostato il parametro ho anche introdotto il tempo di attesa massimo, impostato a 50 tick (5 secondi), in modo da impedire attese troppo lunghe che potrebbero compromettere l'esperienza generale di gioco.

## Three Cups Game

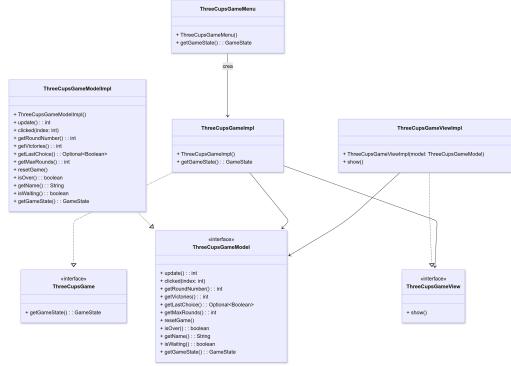


Figura 2.17: Diagramma UML del minigioco Three Cups Game

**Problema:** era necessario che la parte grafica del minigioco fosse in grado di cambiare l'immagine mostrata per rivelare la posizione della biglia all'interno delle tre coppe.

**Soluzione:** per risolvere questo problema ho implementato un Timer prendendo ispirazione dal lavoro già realizzato con click the color. Il gioco è quindi aggiornato ogni 0.1 secondi, e questo lasso di tempo viene chiamato tick, per comodità. Ogni tick di gioco viene controllato lo stato attuale del gioco utilizzando una metoda di `ThreeCupsGameModelImpl` chiamato `update()` che aggiorna i valori dei campi. Questa funziona ritorna quindi i valori aggiornati della cup che deve essere rimpiazzata dall'immagine della biglia, per mostrare al giocatore l'opzione giusta.

### Game board

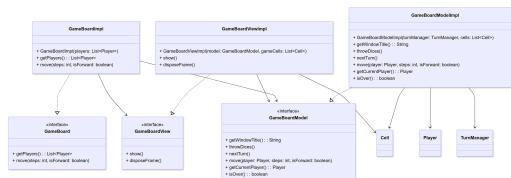


Figura 2.18: Diagramma UML del tabellone

**Problema:** mostrare la posizione attuale della pedina del giocatore, in modo che sia graficamente visibile e che sia possibile risalire in modo semplice alla posizione del giocatore. Oltre a questo, rendere semplice la comunicazione con le altre classi necessarie allora svolgimento del gioco (`DoubleDice`, `TurnManager`, `FinalBoard`, ecc...)

**Soluzione:** essendo una serie di problematiche abbastanza complesse sono state adottate diverse soluzioni a seconda del contesto:

- Per la gestione del player era stata inizialmente pensata una struttura centralizzata sulla classe `Player` stessa, la quale avrebbe gestito il movimento del player e tutte le sue proprietà. Durante lo svolgimento del progetto però questo metodo si è dimostrato inefficiente e incompatibile con la struttura del progetto stesso. È stato quindi deciso di centralizzare le responsabilità della gestione dei player alla classe `Cell`, la quale rappresentava la singola cella del tabellone. All'interno della classe viene quindi tenuto conto dei giocatori attualmente presenti in quella cella. La lista delle celle del tabellone viene memorizzata nella classe `GameBoardImpl`, all'interno della quale vengono gestiti lo spostamento del giocatore, in modo da renderlo facilmente integrabile con le classi `DoubleDice` e la meccanica delle carte.
- Per mantenere la compatibilità con `FinalBoard` il campo `Position` all'interno della classe `Player` è stato mantenuto, in modo che la classifica finale sia comunque gestibile passando esclusivamente la lista dei giocatori in partita
- Per rendere funzionante la meccanica delle carte (postiva se vinco il minigioco, negativa se perdo il minigioco) è stato necessario riuscire a riportare lo stato del minigioco (vittoria, sconfitta, pareggio) all'interno della classe `GameBoardModelImpl`. Per ottenere questo risultato, il minigioco viene inizialmente avviato all'interno della classe `CellImpl`, dove un Timer controlla ogni 0.1 secondi che il gioco sia terminato. Una volta che il minigioco termina, il timer viene fermato in modo da preservare lo stato del minigioco. All'interno di `GameBoardModelImpl` è presente un altro timer che continua a verificare che il minigioco sia finito. Una volta terminato il timer viene fermato e richiamato il metodo che gestisce l'aggiunta delle carte.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Tutti i test di seguito riportati sono stati creati usando JUnit.

#### 3.1.1 Giuseppe Fusco

- **TestMemoryModel:** verifica il corretto funzionamento della logica del mini game *Memory*.
- **TestSnakeModel:** verifica il corretto funzionamento della logica del mini game *Snake*.

#### 3.1.2 Lucia Pola

- **TestTrisModelImpl:** verifica il corretto funzionamento della logica del mini game *Tris(Tic-Tac-Toe)*.
- **TestPuzzleModelImpl:** verifica il corretto funzionamento della logica del mini game *Puzzle*.
- **TestDoubleDiceImpl:** verifica il corretto funzionamento della logica della coppia dei dadi.
- **TestTurnManagerImpl:** verifica il corretto funzionamento della logica della gestione dei turni.

### 3.1.3 Giada Tedaldi

- **TestRockPaperScissorsModelImpl:** La classe TestRockPaperScissorsModelImpl consiste in un insieme di test progettati per assicurare il corretto funzionamento della logica del mini game.
- **TestHangmanModelImpl:** La classe TestHangmanModelImpl consiste in un insieme di test progettati per assicurare il corretto funzionamento della logica del mini game.

### 3.1.4 Samuele D'Ambrosio

- **CardSatchelTest:** Verifica l'aggiunta e rimozione di carte bonus, il limite di sei carte, lo svuotamento con clear(), lo stato con isFull() e l'immutabilità della lista restituita da getCards().
- **HerdingHoundModelTest:** Controlla l'avvio (ONGOING), la vittoria (WON) e il corretto ripristino dello stato iniziale con resetGame().
- **HonkMandModelTest:** Testa l'avvio (ONGOING), il passaggio al turno successivo con input corretto (NEXTROUND) e la fine del gioco con errore (GAMEOVER).

### 3.1.5 Matteo Crepaldi

A causa della natura casuale dei minigiochi sviluppati, ho trovato estremamente complesso riuscire ad integrare dei test per i miei minigiochi. Pur facendo ricerche continue durante la realizzazione del progetto non sono stato in grado di trovare un modo che fosse pratico e utile per integrare dei test.

## 3.2 Note di sviluppo

### 3.2.1 Giuseppe Fusco

- **Utilizzo di Stream:** utilizzati in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: <https://github.com/giuseppezfusco9/OOP24-goosegame/blob/master/src/main/java/it/unibo/goosegame/model/finalboard/impl/FinalBoardLogicImpl.java#L35>

- **Utilizzo di Lambda Function:** utilizzati in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/master/src/main/java/it/unibo/goosegame/view/minigames/memory/impl/MemoryViewImpl.java#L96>
- **Riuso del codice in MemoryModelImpl:** alcune porzioni di codice presenti in questa classe sono state adattate da esempi didattici opportunamente modificati per integrarsi nel progetto. <https://github.com/giuseppefusco9/OOP24-goosegame/blob/master/src/main/java/it/unibo/goosegame/model/minigames/memory/impl/MemoryModelImpl.java#L49>

### 3.2.2 Lucia Pola

- **Utilizzo di Optional:** utilizzato nel minigame *Puzzle*. Un esempio: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/b05ceada14c539a6a220fea6b95953377069c772/src/main/java/it/unibo/goosegame/model/minigames/puzzle/impl/PuzzleModelImpl.java#L38>
- **Utilizzo di Lambda Function:** utilizzati in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/b05ceada14c539a6a220fea6b95953377069c772/src/main/java/it/unibo/goosegame/controller/minigames/tris/impl/TrisControllerImpl.java#L113>
- **Utilizzo di Stream:** utilizzati in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/b05ceada14c539a6a220fea6b95953377069c772/src/main/java/it/unibo/goosegame/model/minigames/tris/impl/TrisModelImpl.java#L205>

### 3.2.3 Giada Tedaldi

- **Utilizzo di Lambda Function:** utilizzati in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/b05ceada14c539a6a220fea6b95953377069c772/src/main/java/it/unibo/goosegame/controller/minigames/tris/impl/TrisControllerImpl.java#L113>

```
//github.com/giuseppefusco9/OOP24-goosegame/blob/  
ed1e796b44278bb3f7c73aa7f3d92c03b7f81112/src/main/java/it/  
unibo/goosegame/model/gamemode/impl/MenuLogicImpl.java#L46
```

- **Utilizzo di Stream:** utilizzati in vari punti del progetto per semplificare e rendere più leggibile il codice. Un esempio significativo è visibile qui: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/ed1e796b44278bb3f7c73aa7f3d92c03b7f81112/src/main/java/it/unibo/goosegame/model/gamemode/impl/MenuLogicImpl.java#L67>

### 3.2.4 Samuele D'Ambrosio

- **Utilizzo di Stream:** utilizzati nel codice per rendere più semplice e fluido il funzionamento di aluni metodi. Un esempio è visibile qui: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/master/src/main/java/it/unibo/goosegame/utilities/Card.java#L147>
- **Utilizzo di Lambda Function:** utilizzate per semplificare il codice, evitando la creazione esplicita di classi anonime per interfacce funzionali, anche per Runnable. Questo ha reso il codice più leggibile, compatto e coerente con lo stile moderno di Java. Un esempio specifico è visibile qui: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/master/src/main/java/it/unibo/goosegame/controller/minigames/herdinghound/HerdHoundController.java#L67>

### 3.2.5 Matteo Crepaldi

**Utilizzo di Optional:** utilizzato nel controller della Cella. Un esempio: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/master/src/main/java/it/unibo/goosegame/controller/cell/impl/CellImpl.java#L34>

**Utilizzo di Supplier:** utilizzato nel controller della Cella. Un esempio: <https://github.com/giuseppefusco9/OOP24-goosegame/blob/master/src/main/java/it/unibo/goosegame/controller/cell/impl/CellImpl.java#L39>

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Giuseppe Fusco

Il mio compito in questo progetto è stato quello di sviluppare i due minigiochi "Snake" e "Memory" e della schermata di gioco finale. Sono cosciente del fatto che il mio lavoro non è impeccabile, ci saranno sicuramente dettagli da limare e aspetti da chiarire meglio ma sono complessivamente soddisfatto del mio operato in questi ultimi mesi. Questo progetto ha portato tante difficoltà quanti insegnamenti: da questa esperienza ho imparato come gestire un lavoro a gruppi, quanto sia importante la coordinazione del lavoro e quanto siano altrettanto importanti molti aspetti della programmazione che prima non conoscevo (i pattern, ad esempio). Ho cercato di ampliare ancor di più la mia conoscenza nella programmazione ad oggetti e più specificatamente le mie conoscenze in Java.

#### 4.1.2 Lucia Pola

Complessivamente, mi ritengo abbastanza soddisfatta del lavoro svolto. È stato il mio primo progetto di questo tipo e calibro, e infatti, soprattutto all'inizio, ho riscontrato alcune difficoltà sia nella gestione del lavoro individuale, sia nella gestione e comunicazione con il gruppo. Queste difficoltà, però, mi hanno fatta crescere sia come persona sia come programmatrice: ho acquisito e ampliato le mie conoscenze in Java e ho compreso quanto sia importante e costruttivo il confronto con i colleghi, anche se non sempre facile. Durante lo sviluppo del progetto, mi sono occupata principalmente dello sviluppo dei minigiochi Tris e Puzzle, e della realizzazione delle classi per la gestione dei turni e del lancio del dado. Questi compiti mi hanno permesso

di toccare diversi aspetti della programmazione a oggetti, dalla parte di view fino alla logica vera e propria del model, e mi hanno insegnato quanto spesso io sia insoddisfatta del lavoro svolto, pretendendo sempre di più e puntando al meglio, non solo per avere un codice funzionante, ma anche pulito e ben strutturato.

#### 4.1.3 Giada Tedaldi

Durante lo sviluppo del progetto mi sono occupato principalmente di ampliare il menu principale del gioco e di progettare un'interfaccia comune per i menu dei minigiochi. Questo mi ha permesso di lavorare sull'aspetto visivo e sull'organizzazione delle schermate, cercando di rendere la navigazione il più chiara e intuitiva possibile per l'utente. Oltre a questo, ho realizzato due minigiochi: "Hangman" e "RockPaperScissors". Per entrambi ho sviluppato sia la logica di gioco che la relativa interfaccia grafica. È stato interessante affrontare la sfida di creare giochi semplici ma funzionanti, curando l'interazione tra l'utente e il programma, e gestendo i vari stati di gioco in modo efficace. Questa esperienza è stata un'opportunità stimolante per mettermi alla prova, sia dal punto di vista tecnico che collaborativo; essa mi ha aiutato a migliorare le mie capacità di progettazione e sviluppo in Java, in particolare nell'ambito delle interfacce grafiche con Swing. Inoltre, ho apprezzato molto il lavoro di squadra e la possibilità di confrontarmi con i miei compagni per risolvere problemi e prendere decisioni condivise.

#### 4.1.4 Samuele D'Ambrosio

La realizzazione di questo progetto è stata per me un'occasione preziosa per mettere in pratica le conoscenze teoriche, ma anche per crescere nella gestione concreta di un software complesso. Lavorare su componenti come Satchel, HerdingHound e HonkMand mi ha permesso di affrontare sfide diverse e complementari, che hanno affinato il mio approccio alla progettazione. In particolare, lavorare su HerdingHound mi ha aiutato a migliorare molto nella strutturazione del codice e nell'organizzazione dei moduli. L'articolazione delle sue logiche — dalla gestione della griglia al comportamento dinamico del cane — mi ha spinto a rivedere e ottimizzare anche le altre componenti su cui stavo lavorando. Grazie a questa esperienza, ho imparato a riflettere in modo più critico su come scrivo e organizzo il codice, cercando soluzioni più modulabili, pulite e coerenti. Un esempio concreto di questa evoluzione è la gestione del Satchel, dove ho deciso di creare una view dedicata per la carta, trattandola come un'entità indipendente e rendendola più gestibile e

riutilizzabile all'interno dello zaino. In generale, questo progetto mi ha fatto capire l'importanza della continua raffinazione del proprio modo di lavorare: non solo per migliorare la qualità del codice, ma anche per rendere più efficace e piacevole la collaborazione all'interno del team. Sono soddisfatto del percorso fatto e del risultato raggiunto.

#### **4.1.5 Matteo Crepaldi**

La realizzazione di questo progetto mi ha messo particolarmente alla prova, soprattutto per una mia mancanza di abitudine per il lavoro di gruppo. Ho avuto principalmente difficoltà ad organizzarmi con le tempistiche comuni del progetto e soprattutto a dipendere da codice scritto da altre persone. Nonostante la difficoltà iniziale, sono riuscito in poco tempo ad adattarmi al workflow del gruppo, facendo progressi in maniera veloce e riuscendo infine a far combaciare il mio lavoro con quello degli altri. Mi ha soprattutto aiutato ad acquisire l'utilizzo di concetti teorici come i pattern, che fino a questo momento risultavano principalmente astratti. Sono tutto sommato soddisfatto del lavoro eseguito.

## Appendice A

## Guida utente

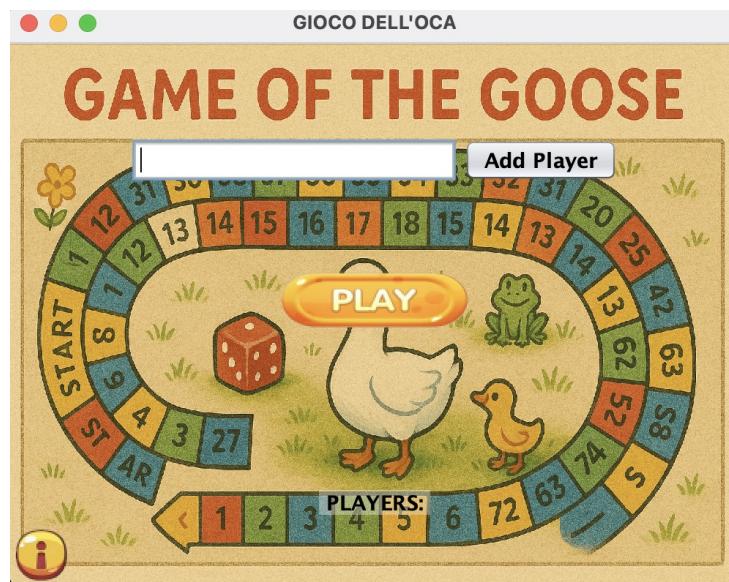


Figura A.1: Immagine della schermata iniziale

**Menù** All'avvio del gioco, compare la schermata iniziale (Figura A.1). In questa schermata è possibile aggiungere da 2 a 4 giocatori con nomi univoci, utilizzando la barra bianca e il pulsante "Add Player". Il pulsante con la "i" in basso a sinistra consente di aprire la schermata delle istruzioni di gioco, mentre il pulsante centrale "Play" avvia la partita.

## Tabellone di gioco e "saccoccia"

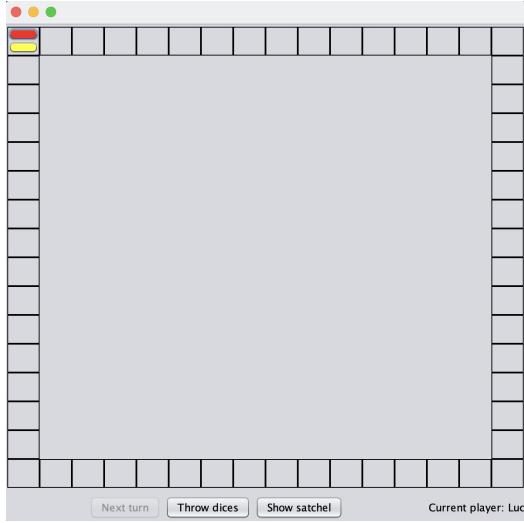


Figura A.2: Immagine del tabellone di gioco

La Figura A.2 mostra la schermata principale del gioco. Nella parte inferiore si trovano tre pulsanti: "Throw Dices" consente al giocatore corrente, indicato in basso a destra, di lanciare i dadi; "Show Satchel" permette di visualizzare la "saccoccia" del giocatore corrente, mostrata in Figura A.3, che contiene le carte bonus ottenute durante la partita e che possono essere utilizzate a proprio vantaggio nel corso dei propri turni; "Next Turn" consente di terminare il turno del giocatore corrente e passare il turno al giocatore successivo. Al centro dello schermo si trova il tabellone di gioco, che mostra le pedine colorate dei giocatori e le 60 celle, suddivise tra celle stazionarie e celle minigioco.



Figura A.3: Immagine della "saccoccia"

## Minigiochi



Figura A.4: Immagine d'esempio dei menù dei minigiochi

Nel caso in cui si capiti su una cella minigame, questo si apre con un menu iniziale, come mostrato nella Figura A.4, caratterizzato da un'immagine di sfondo che rappresenta il minigame stesso. In basso a sinistra è presente il pulsante "i" per accedere alle istruzioni, mentre al centro si trova il pulsante "Play" per avviare il minigame.

**Giuseppe Fusco**  
**Snake**

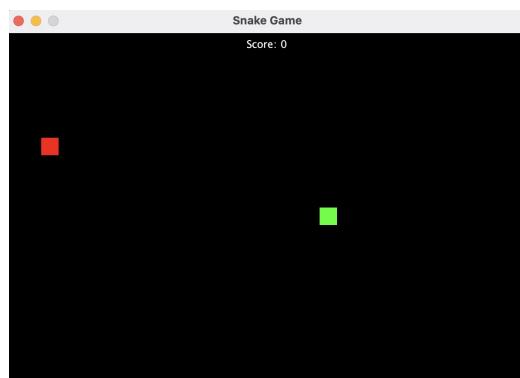


Figura A.5: Schermata del miniogame Snake

All'avvio del minigioco compare una schermata nera con un quadrato verde, che rappresenta il serpente, e un quadrato rosso, che rappresenta la mela (Figura A.5). Il serpente si muove automaticamente e può essere controllato tramite le frecce direzionali della tastiera. In alto al centro è presente un testo con la scritta “Score”, che tiene il conto delle mele mangiate. Ogni volta che il serpente mangia una mela, si allunga. L'obiettivo è raggiungere uno score pari a 15, senza scontrarsi con i ”muri”, cioè i bordi della schermata, o contro il serpente stesso. Alla fine verrà mostrato al giocatore l'esito del minigioco.

### Memory

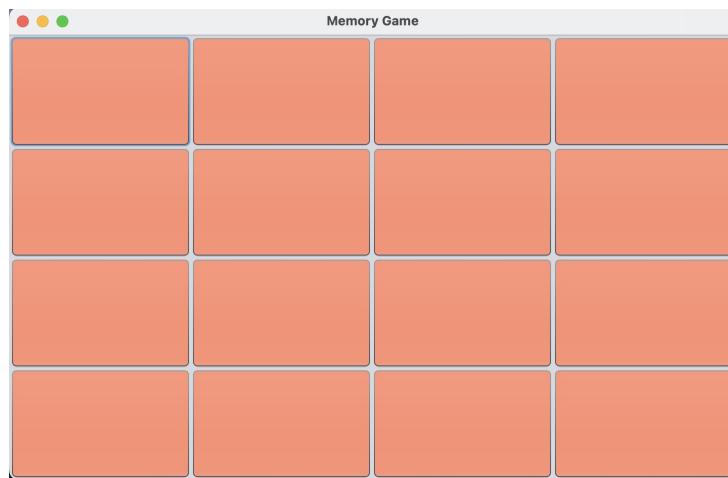


Figura A.6: Schermata del miniogioco Memory

All'avvio del minigioco compare una griglia 4x4 composta da bottoni rossi come mostrato in Figura A.6. Il giocatore può selezionare due bottoni alla volta per rivelare il contenuto nascosto. Se le due carte selezionate corrispondono, resteranno scoperte; in caso contrario, torneranno a essere coperte dopo aver cliccato su un'altra carta coperta. Alla fine, verrà mostrato al giocatore l'esito del minigioco.

**Lucia Pola**  
**Tris(Tic-Tac-Toe)**

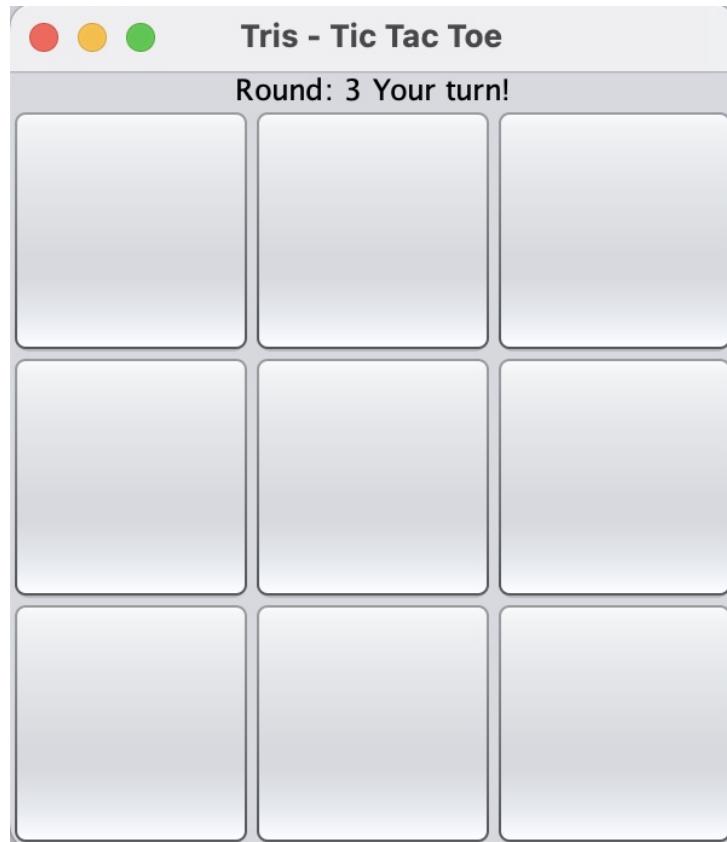


Figura A.7: Schermata del minigioco Tris(Tic-Tac-Toe)

All'avvio del minigioco compare una schermata con una griglia 3x3 (Figura A.7). Il giocatore e il computer si alternano nel selezionare una delle caselle vuote per inserire il proprio simbolo, “X” o “O”. La modalità è “best of three”: si giocheranno fino a tre partite e chi vince due round si aggiudica il minigioco. In alto al centro è presente un testo che indica il numero del round corrente. Alla fine, verrà mostrato al giocatore l'esito del minigioco.

## Puzzle

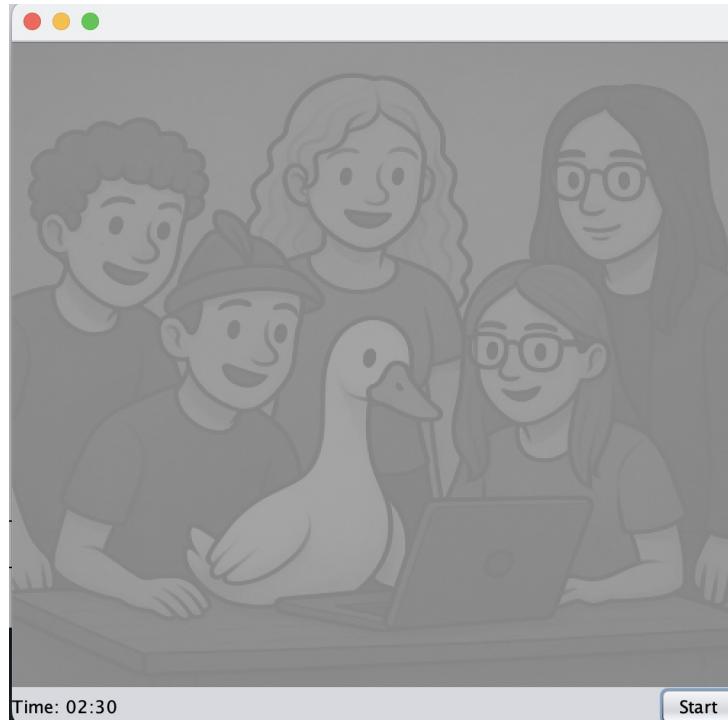


Figura A.8: Schermata del miniogioco Puzzle

All'avvio del minigioco compare una schermata con una griglia 5x5, inizialmente con un'immagine ordinata (Figura A.8). In basso è presente il pulsante “Start” che, una volta premuto, avvia il timer visibile in basso a sinistra e mescola la griglia, dando inizio al gioco. Per scambiare due caselle, il giocatore deve cliccare prima su una e poi sull'altra. Alla fine, verrà mostrato al giocatore l'esito del minigioco.

**Giada Tedaldi**  
**RockPaperScissors**

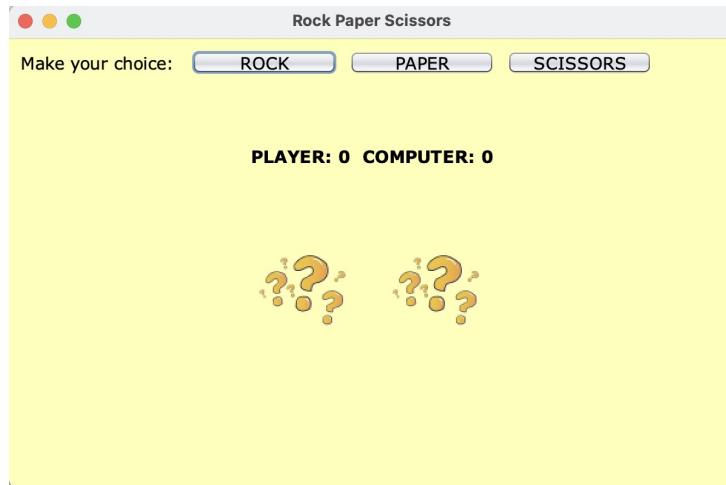


Figura A.9: Schermata del miniogioco RockPaperScissors

All'avvio del minigioco compare una schermata (Figura A.9) con tre bottoni che rappresentano le scelte del giocatore: sasso, carta e forbice. Il giocatore sfida il computer. Sotto questi bottoni è presente un testo che mostra i punteggi di entrambi i giocatori. Ogni volta che il giocatore fa una scelta, al centro dello schermo compare l'immagine corrispondente sia alla propria scelta sia a quella del computer, così da determinare il vincitore del round. Alla fine, verrà mostrato al giocatore l'esito del minigioco.

## Hangman

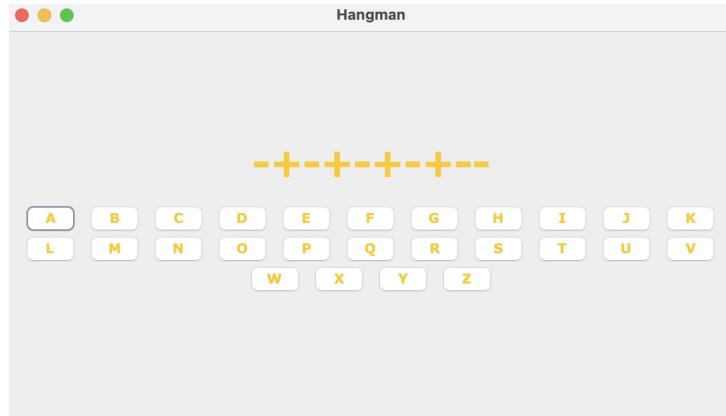


Figura A.10: Schermata del miniogioco Hangman

All'avvio del minigioco compare una schermata con una parola nascosta da indovinare (Figura A.10). Il giocatore sfida il computer e deve cercare di indovinare la parola selezionando le lettere dell'alfabeto, disposte come bottoni sulla schermata. In alto al centro viene mostrata la parola con le lettere già indovinate e i trattini/più (consonanti/vocali) al posto di quelle mancanti. Ogni lettera sbagliata fa comparire un pezzo del disegno dell'impiccato, che appare sulla destra della schermata. Alla fine, verrà mostrato al giocatore l'esito del minigioco.

**Samuele D'Ambrosio**  
**HerdingHound**

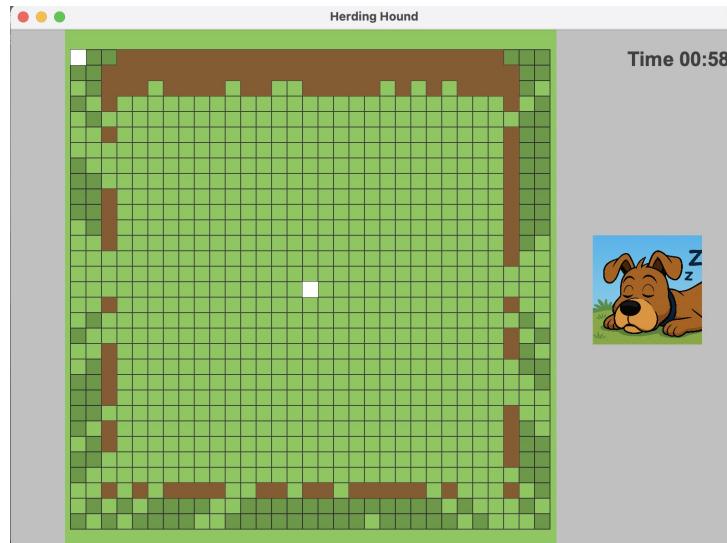


Figura A.11: Schermata del minigioco Herdinghound

All'avvio del minigioco compare la schermata con il campo di gioco al centro, come mostrato in Figura A.11. Al centro del campo c'è un quadrato che rappresenta un cane da guardia: ogni tanto il cane si sveglia e illumina un lato del campo. Il giocatore è rappresentato da un quadratino in alto a sinistra e, premendo la barra spaziatrice, deve avanzare e cercare di raggiungere la fine del percorso senza farsi vedere dal cane. A destra dello schermo è visibile un'immagine del cane che cambia a seconda che il cane stia dormendo, si stia svegliando o sia completamente sveglio. In alto a destra è presente un timer che indica il tempo massimo entro il quale il giocatore deve completare il percorso. Alla fine, verrà mostrato l'esito del minigioco.

## HonkMand

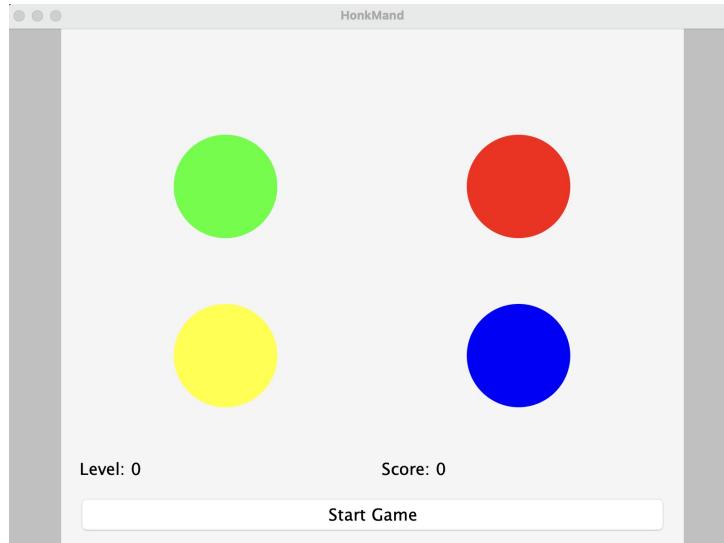


Figura A.12: Schermata del minigioco HonkMand

All'avvio del minigioco compare una schermata con quattro bottoni colorati disposti a formare un quadrato (Figura A.12). I bottoni si illuminano in sequenza per mostrare la combinazione che il giocatore dovrà ripetere. In basso a sinistra è visibile il numero del livello corrente, mentre in basso a destra viene mostrato il punteggio raggiunto. Al centro in basso si trova un bottone che permette di iniziare il minigioco. Alla fine, verrà mostrato l'esito del minigioco.

Matteo Crepaldi

## Click The Color

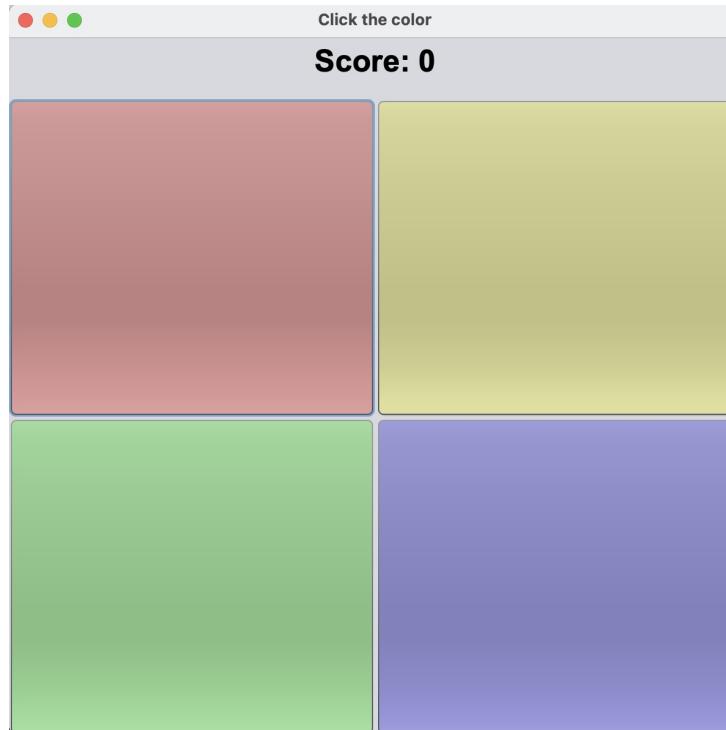


Figura A.13: Schermata del minigioco Click The Color

All'avvio del minigioco compare una schermata con una griglia 2x2 composta da bottoni colorati (Figura A.13). Uno alla volta, i bottoni si illuminano e il giocatore deve cliccare rapidamente sul bottone colorato illuminato entro meno di un secondo. In alto al centro dello schermo viene mostrato il punteggio corrente, che aumenta man mano che il giocatore riesce a cliccare correttamente e velocemente, altrimenti decremente. Alla fine, verrà mostrato l'esito del minigioco.

## Three Cups Game



Figura A.14: Schermata del minigioco Three Cups Game

All'avvio del minigioco compaiono tre bicchieri capovolti posizionati al centro della schermata, come mostrato in Figura A.14. Il giocatore deve cliccare sui bicchieri per capovolgerli e cercare di indovinare sotto quale bicchiere si nasconde la biglia. In alto al centro dello schermo è presente un testo che mostra il round attuale e il numero di vittorie raggiunte. Alla fine, verrà mostrato l'esito del minigioco.

## Schermata finale

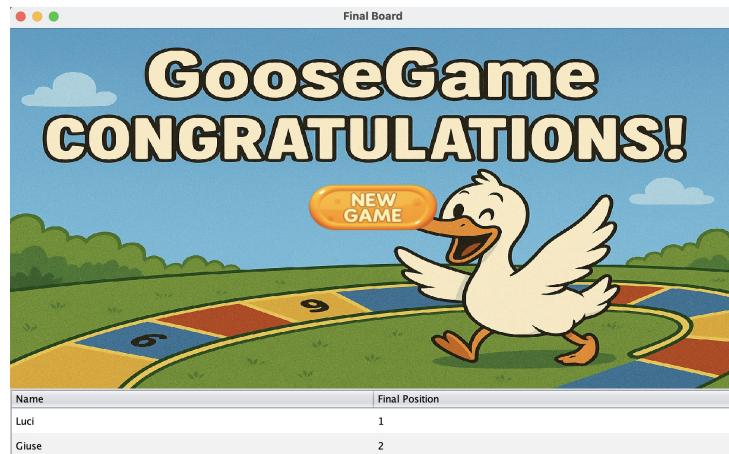


Figura A.15: Immagine della schermata finale

Alla fine del gioco compare una schermata finale (Figura A.15) che mostra in basso la classifica con i nomi dei giocatori sulla sinistra e la posizione in classifica sulla destra. Al centro della schermata si trova un pulsante “New Game” che permette di tornare al menu iniziale (Figura A.1) e di avviare una nuova partita.

# **Appendice B**

## **Esercitazioni di laboratorio**

### **Appendice B Esercitazioni laboratorio**

#### **B.1.1 giuseppe.fusco9@studio.unibo.it**

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p245934>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247292>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247786>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249042>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250325>

#### **B.1.2 lucia.pola@studio.unibo.it**

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p245907>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247368>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247787>

- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249014>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250527>

### B.1.3 giada.tedaldi@studio.unibo.it

- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p248331>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249197>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250953>