



## “Mario Party”

Relazione per il progetto di Programmazione ad oggetti  
A.A. 2023/2024

Ariyo Daniel  
Ferretti Filippo  
Fronzoni Gabriele  
Ricci Nicholas  
Yan Elisa

13 luglio 2024

# Indice

|   |           |
|---|-----------|
| <b>1 Analisi</b>                              | <b>3</b>  |
| 1.1 Requisiti . . . . .                       | 3         |
| 1.2 Analisi e modello del dominio . . . . .   | 8         |
| <b>2 Design</b>                               | <b>9</b>  |
| 2.1 Architettura . . . . .                    | 9         |
| 2.2 Design dettagliato . . . . .              | 11        |
| 2.2.1 Ariyo Daniel . . . . .                  | 11        |
| 2.2.2 Ferretti Filippo . . . . .              | 16        |
| 2.2.3 Fronzoni Gabriele . . . . .             | 20        |
| 2.2.4 Ricci Nicholas . . . . .                | 26        |
| 2.2.5 Yan Elisa . . . . .                     | 29        |
| <b>3 Sviluppo</b>                             | <b>42</b> |
| 3.1 Testing automatizzato . . . . .           | 42        |
| 3.2 Note di sviluppo . . . . .                | 44        |
| 3.2.1 Ariyo Daniel . . . . .                  | 44        |
| 3.2.2 Ferretti Filippo . . . . .              | 44        |
| 3.2.3 Fronzoni Gabriele . . . . .             | 45        |
| 3.2.4 Ricci Nicholas . . . . .                | 45        |
| 3.2.5 Yan Elisa . . . . .                     | 46        |
| <b>4 Commenti finali</b>                      | <b>48</b> |
| 4.1 Autovalutazione e lavori futuri . . . . . | 48        |
| 4.1.1 Ariyo Daniel . . . . .                  | 48        |
| 4.1.2 Ferretti Filippo . . . . .              | 48        |
| 4.1.3 Fronzoni Gabriele . . . . .             | 49        |
| 4.1.4 Ricci Nicholas . . . . .                | 49        |
| 4.1.5 Yan Elisa . . . . .                     | 50        |
| <b>A Guida utente</b>                         | <b>51</b> |

|   |           |
|---|-----------|
| <b>B Esercitazioni di laboratorio</b>             | <b>61</b> |
| B.0.1 elisa.yan@studio.unibo.it . . . . .         | 61        |
| B.0.2 gabriele.fronzoni@studio.unibo.it . . . . . | 61        |

# Capitolo 1

## Analisi

### 1.1 Requisiti

Il software mira alla costruzione di un videogioco ispirato a “Mario Party”. Il gioco si svolge in un tabellone simil “gioco dell’oca”, all’interno del quale giocano dai 2 ai 4 giocatori a partita. Ogni giocatore ha lo scopo di accumulare quante più monete possibili e, alla fine di un determinato numero di round, il giocatore con più stelle, acquisibili grazie alle monete, vince la partita.

#### Requisiti funzionali

- L’applicazione si apre con una schermata iniziale nella quale viene data la possibilità di cominciare una nuova partita e consultare il regolamento di gioco.
- Quando si inizia una partita viene chiesto di inserire i *nickname* e il *personaggio* per ogni giocatore.
- Ogni giocatore a turno tira due dadi e si muove all’interno del tabellone. Ogni casella ha un comportamento diverso e può essere:
  - **Bonus:** il giocatore guadagna un certo numero di monete;
  - **Malus:** il giocatore perde un certo numero di monete;
  - **Minigioco single-player:** inizia un minigioco al quale prende parte il solo giocatore terminato sulla casella. Nel caso vinca in questo, il player guadagna un certo numero di monete mentre se perde non guadagna nulla;

- **Minigioco multi-player:** inizia un minigioco al quale prendono parte il giocatore capitato sulla casella e un avversario casuale. Il vincitore guadagna un certo numero di monete mentre l’altro non guadagna niente;
- **Stella:** nel momento in cui il giocatore capita o attraversa questa casella se ha abbastanza monete acquista una stella. Nel momento in cui un giocatore acquisisce una stella, la casella “*Stella*” si sposta in un altro punto del tabellone;
- **Negozio:** all’interno del negozio il giocatore può acquistare con le monete accumulate alcuni *items* che può utilizzare per ottenere dei vantaggi nei round successivi (es. tira un dado in più);
- **Sentiero:** una casella di passaggio per i giocatori che non ha nessun effetto ma completa semplicemente la struttura del tabellone;
- Al termine della partita si visualizza una schermata con i dettagli della partita e i punteggi.

### **Minigiochi:**

#### **Memory card (Fronzoni - Single-player)**

- L’idea di base del gioco è quella di un memory con le carte;
- All’inizio del gioco vengono mostrate tutte le carte scoperte. Ogni carta rappresenta un frutto. Per ogni frutto ci sono due carte;
- Nel momento in cui il giocatore è pronto, preme un pulsante e le carte vengono coperte;
- Il giocatore prova ad indovinare le coppie di carte uguali. Durante la partita vengono consentiti al giocatore un massimo di 3 errori;
- Alla fine della partita, in base al numero di coppie che sono state indovinate, viene calcolato un punteggio che corrisponde alle monete guadagnate dal giocatore.

#### **Codice segreto (Ricci - Multi-player)**

- I due giocatori, a turno, dovranno inserire una sequenza ordinata di colori per cercare di indovinare appunto il *codice segreto*
- Il *codice segreto* è una sequenza di colori non nota ai giocatori che non cambia all’interno di una partita

- I giocatori per intuire la soluzione per ogni tentativo fatto avranno degli indizi che li aiuteranno ad arrivare alla soluzione
- L'indizio, per ogni colore inserito nel tentativo, potrebbe essere che tale colore:
  - non è nella soluzione
  - è nella soluzione ma non nella posizione corretta
  - è nella soluzione ed è nella posizione corretta
- Vince il primo giocatore ad indovinare il *codice segreto*;
- Se alla fine dei turni prestabiliti nessun giocatore è arrivato alla soluzione, per determinare il vincitore si considerano in ordine i seguenti parametri:
  - maggior numero di colori che appartengono alla soluzione e sono nella posizione corretta
  - maggior numero di colori che appartengono alla soluzione e non sono nella posizione corretta;
  - in caso di una ulteriore parità, vince il giocatore che ha attivato il minigioco finendo sulla casella

### Nanogram (Yan - Single-player)

- All'inizio della partita, viene mostrata una griglia vuota
- Per ogni riga e colonna vengono forniti suggerimenti numerici che indicano le sequenze di celle adiacenti, l'obiettivo è quello di completare la griglia seguendo tali indicazioni
- Ogni partita inizia con 3 vite. Ogni errore nel riempire una cella comporta la perdita di una vita
- La partita si conclude:
  - con una vittoria, quando il giocatore completa correttamente l'intera griglia
  - con una sconfitta, se il giocatore esaurisce tutte le vite prima di completare la griglia

### **Domino (Yan - Multi-player)**

- Il gioco si basa su una partita dove i giocatori posizionano le proprie tessere sul tavolo di gioco formando una catena di domino
- All'inizio della partita, vengono distribuite un numero di tessere ai giocatori
- Durante il turno, i giocatori posizionano una delle loro tessere in modo che corrisponda a uno dei lati della tessera già posizionata nella catena
- Se un giocatore non può giocare una tessera, deve pescarne una dal mazzo
- La partita finisce quando un giocatore finisce le sue tessere o nessuno può più giocare. Se nessuno può giocare, vince chi ha meno punti nelle tessere rimanenti.

### **Connect4 (Ferretti - Multi-player)**

- L'obiettivo del gioco è mettere in fila (orizzontale, verticale o diagonale) quattro proprie pedine
- Il primo giocatore ad iniziare avrà le pedine di colore rosso mentre il secondo, ovvero quello sfidato, avrà le pedine di colore giallo
- Durante il gioco, ad ogni turno, il giocatore di turno dovrà posizionare una sua pedina in una delle colonne disponibili
- La partita si conclude quando uno dei due giocatori posiziona 4 pedine in fila
- Se a riempimento della griglia nessuno dei due giocatori ha posizionato 4 pedine consecutive la vittoria verrà assegnata al giocatore che è stato sfidato.

### **PerilousPath (Ariyo - Single-player)**

- L'obiettivo del gioco "Perilous Path" è connettere le due palline situate alle estremità opposte della griglia
- All'interno della griglia sono posizionate un certo numero di bombe che devono essere evitate
- Per vincere devi creare una linea continua tra le due palline

- Se si clicca sulla cella contenente una bomba si perde
- La sfida consiste nel trovare un percorso sicuro attraverso la griglia, evitando tutte le bombe.

### **MemorySweep (Ariyo - Multi-Player)**

- Il gioco consiste in una griglia di bottoni
- Ad ogni turno, una serie casuale di bottoni verrà colorata
- La sequenza di bottoni colorati aumenta ad ogni turno, rendendo il gioco sempre più difficile
- Il primo giocatore che commette un errore e non riesce a ricreare correttamente la sequenza perde la partita.

### **Requisiti non funzionali**

- Il gioco deve funzionare sui tre principali sistemi operativi: Windows, Linux, MacOs;
- L'interfaccia di gioco punta ad essere intuitiva;
- L'esperienza di gioco deve essere fluida permettendo il susseguirsi di turni e il passaggio dal tabellone di gioco agli eventuali minigiochi.

## 1.2 Analisi e modello del dominio

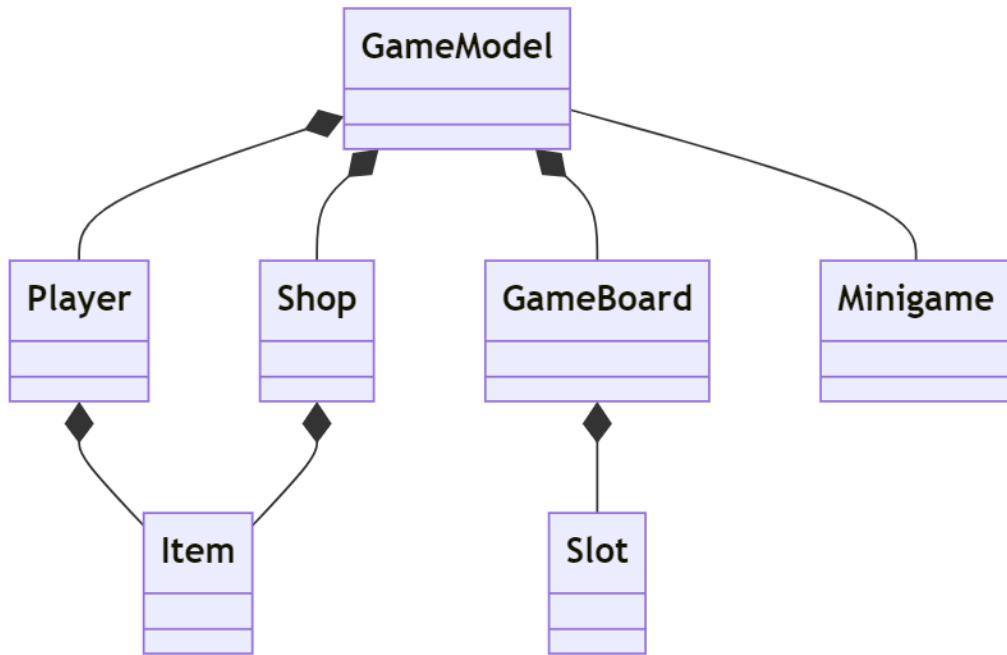


Figura 1.1: Diagramma UML del dominio

Il gioco ha un insieme di **Player** che giocano la partita. I player possono acquisire ed utilizzare degli **Item** nel corso della partita. Gli item possono essere acquistati nell **Shop**. Altro elemento fondamentale del gioco è la **GameBoard** che rappresenta il tabellone di gioco nel quale si muovono i giocatori durante la partita. Ogni **Slot** ha un proprio effetto. Vi sono poi i **Minigame**, ognuno dei quali ha proprie regole e può essere single-player oppure multi-player. Il **GameModel** coordina l'attività di tutte le singole componenti per poter svolgere correttamente l'intero gioco.

# Capitolo 2

## Design

### 2.1 Architettura

Si è deciso di utilizzare il pattern architetturale MVC all'interno del progetto.

Per implementare il pattern MVC si è deciso che il `GameModel` fosse l'entità alla quale sono poi collegate tutte le componenti del gioco (come mostrato in Figura 1.1) e rappresenta quindi l'entry point a livello di Model della nostra applicazione. Il `GameController` ha al suo interno un'istanza di `GameModel` e un'istanza di `GameView`, in modo da poter ricevere gli input dalla View, comandare le modifiche al model e aggiornare la View di conseguenza.

La `GameView` ha al suo interno un riferimento al controller, in modo da poter chiedere di modificare il model in base a quelle che sono le richieste dell'utente. Questa ha inoltre opportuni metodi per cambiare e settare le diverse scene del gioco.

Il controller, quando si è nell'opportuna fase di gioco, controlla se la casella in cui si trova il giocatore comporta un cambio di scena e comanda la View di mostrare quanto necessario. Nel caso i cui sia necessario fare partire un mini-gioco, il `GameController` chiama il metodo della View passandogli il nome del minigioco e i giocatori che partecipano e la `GameView` fa partire a schermo opportunamente il mini-gioco selezionato. Al termine del mini-gioco, quest'ultimo notifica al `GameController` l'esito mediante il metodo `saveMinigameResult`. Il controller poi aggiornerà propriamente il `GameModel`.

Vi sono poi interfacce comuni per Model-Controller-View anche per quanto riguarda i vari mini-giochi.

Lo schema riassuntivo dell'implementazione del pattern MVC è quello esposto in Figura 2.1.

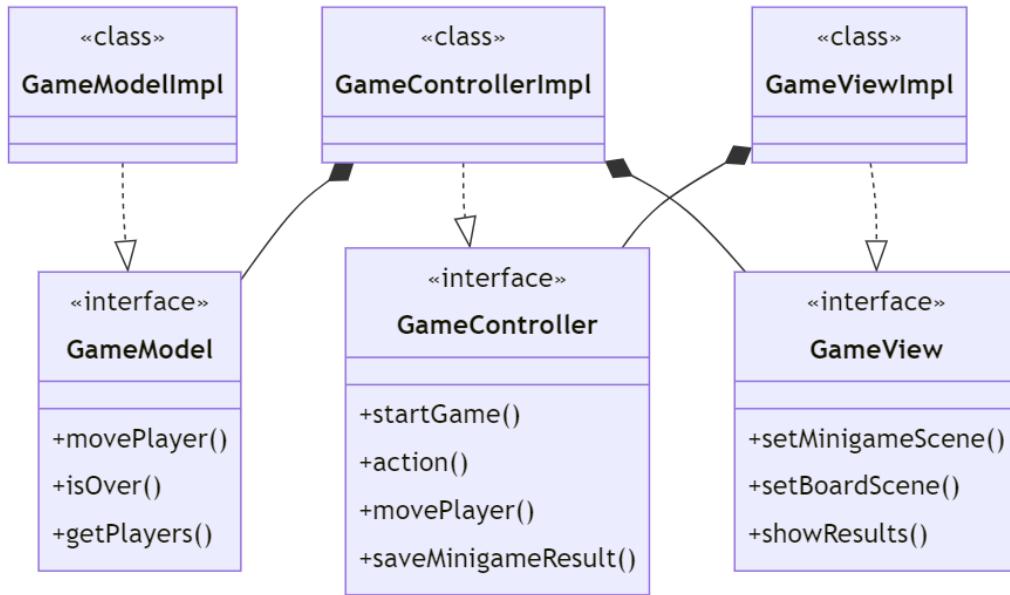


Figura 2.1: Diagramma UML dell'implementazione del pattern MVC

Per la gestione del pattern MVC nel passaggio alle diverse scene del nostro gioco (inizio gioco, tabellone di gioco, mini-giochi vari), abbiamo deciso di definire un interfaccia comune per tutte le scene che è **SceneView** e di fornire un implementazione astratta di questa che sarebbe **AbstractSceneView**. Mediante questa interfaccia si definiscono metodi per poter settare e ottenere in qualsiasi momento opportuni riferimenti alla View e al Controller centrali dell'applicazione per permettere alle View delle diverse scene di potersi opportunamente interfacciare con queste. Tutte le interfacce delle diverse scene estendono **SceneView** e le implementazioni di queste estendono **AbstractSceneView**, la quale ha già implementati i metodi per il setting e l'accesso a View e Controller principali. Lo schema UML di quanto descritto è quello in figura Figura 2.2.

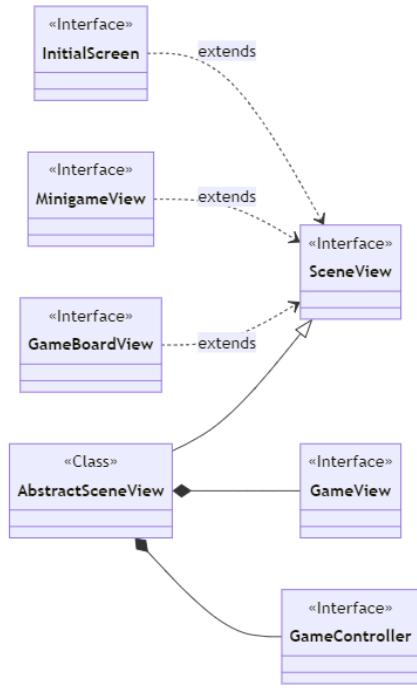


Figura 2.2: Schema UML riassuntivo della gestione delle view delle scene di gioco

## 2.2 Design dettagliato

### 2.2.1 Ariyo Daniel

Mini-gioco Perilous Path

**Problema:** gestione del pattern MVC per il mini-gioco Perilous Path

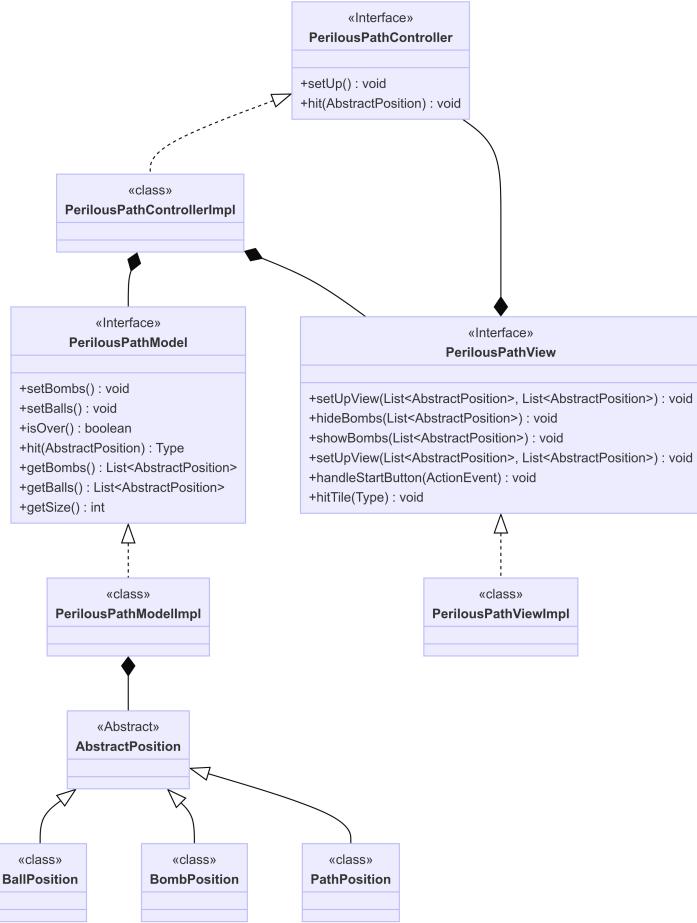


Figura 2.3: schema UML del mini-gioco Perilous Path

**Soluzione:** per il mini-gioco Perilous Path è stato utilizzato il pattern MVC in quanto risulta il più adatto per garantire una chiara separazione della responsabilità. Esso è suddiviso in 3 componenti principali. **PerilousPath** è responsabile della logica del mini-gioco, gestendo le posizioni delle palline e delle bombe oltre a implementare la logica per determinare la validità delle connessioni tra palline che sarà approfondito nel paragrafo successivo. **PerilousPathView** gestisce invece la visione della schermata del mini-gioco, utilizzando JAVAFX , ho creato un’interfaccia utente che viene aggiornata dinamicamente in base a come l’utente interagisce con essa. **PerilousPathController** gestisce l’interazione tra il model e la view e coordinando l’aggiornamento fra essi.

**Problema:** gestione delle posizioni degli elementi all’interno della griglia

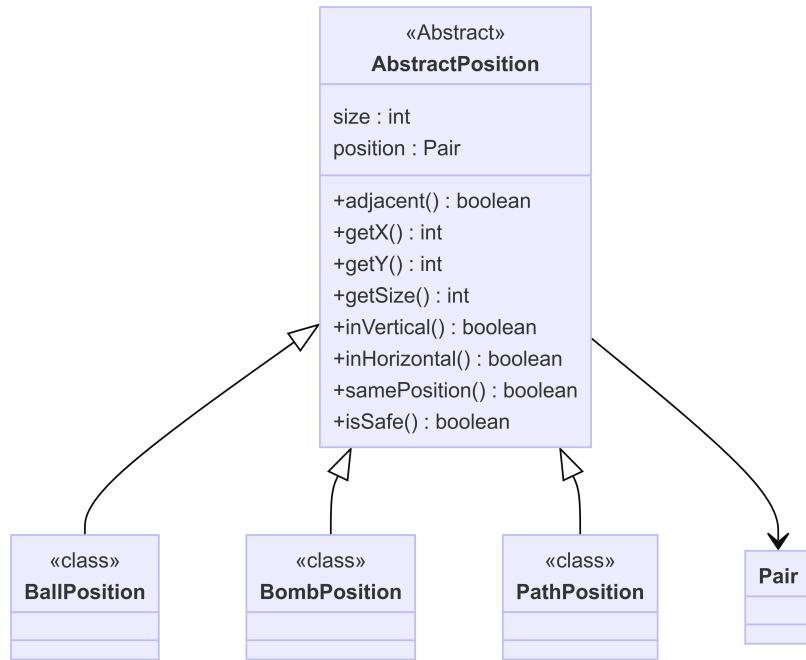


Figura 2.4: schema Abstract Position

**Soluzione:** Per gestire la posizione di palle, bombe e sentieri ho utilizzato il pattern *Template Method*, in quanto gli elementi all'interno della griglia hanno caratteristiche comuni che possono essere astratte in una sola classe andando ad implementare per le singole classi che la estendono il metodo astratto `isSafe()`. Questo approccio semplifica la gestione delle regole di posizionamento degli elementi all'interno del gioco, fornendo un implementazione flessibile. Ogni classe quindi implementa il template method in base alle proprie logiche specifiche.

### Mini-gioco Memory Sweep

**problema:** gestione del pattern MVC per il mini-gioco Memory Sweep

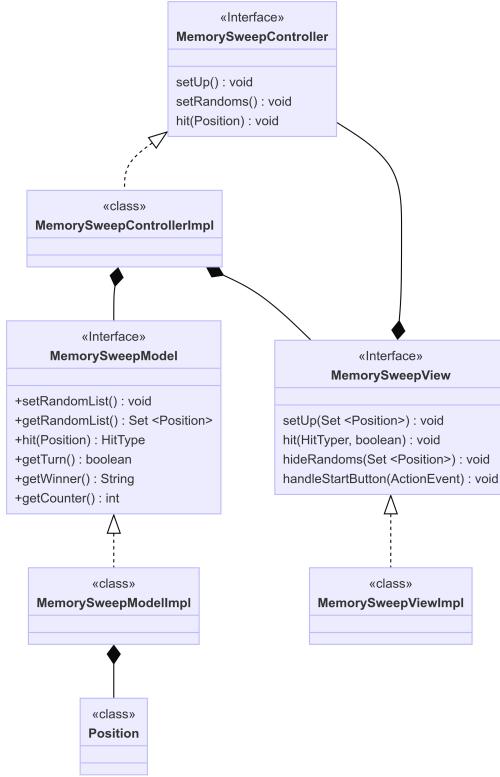


Figura 2.5: schema UML del mini-gioco Memory Sweep

**Soluzione:** per il mini-gioco Memory Sweep ho adottato il pattern MVC, poiché si è rivelato il più appropriato per gestire le complessità del gioco, analogamente a quanto fatto nel mini-gioco precedente. `MemorySweepImpl` gestisce il model del del minigioco, ciò include la generazione e l'organizzazione delle sequenze di bottoni colorati, la gestione dello stato del gioco e della progressiva complessità delle sequenze. `MemorySweepController` agisce come intermediario tra model e view gestendone anche l'aggiornamento. `memorySweepView`, implementata con JAVAFX rappresenta la griglia di bottoni e risponde dinamicamente agli aggiornamenti del model.

**Problema:** gestione del tipo di clic

**Soluzione:** quando un giocatore preme un pulsante possono accadere tre cose:

- il pulsante cliccato è giusto ma la sequenza non è terminata

- il pulsante cliccato è giusto e la sequenza è terminata con conseguente cambio di turno all'avversario
- il pulsante cliccato è sbagliato e quindi il gioco termina con la vittoria dell'avversario

Utilizzo quindi un Set <Position> per la sequenza che dovrà essere ricreata, ad ogni clic di un pulsante viene controllato se la sua posizione appartiene all'interno del Set, in caso affermativo viene aggiunto al Set <Position> che ogni giocatore possiede e si controlla se la grandezza dei due Set equivalga che sta a significare che il turno è terminato con successo, in tal caso il turno passerà all'avversario. Nel caso in cui il pulsante cliccato sia sbagliato la partita terminerà dando la vittoria all'avversario.

## Schermata Iniziale

**Problema:** gestione dell'avvio della partita e dell'istanza gioco

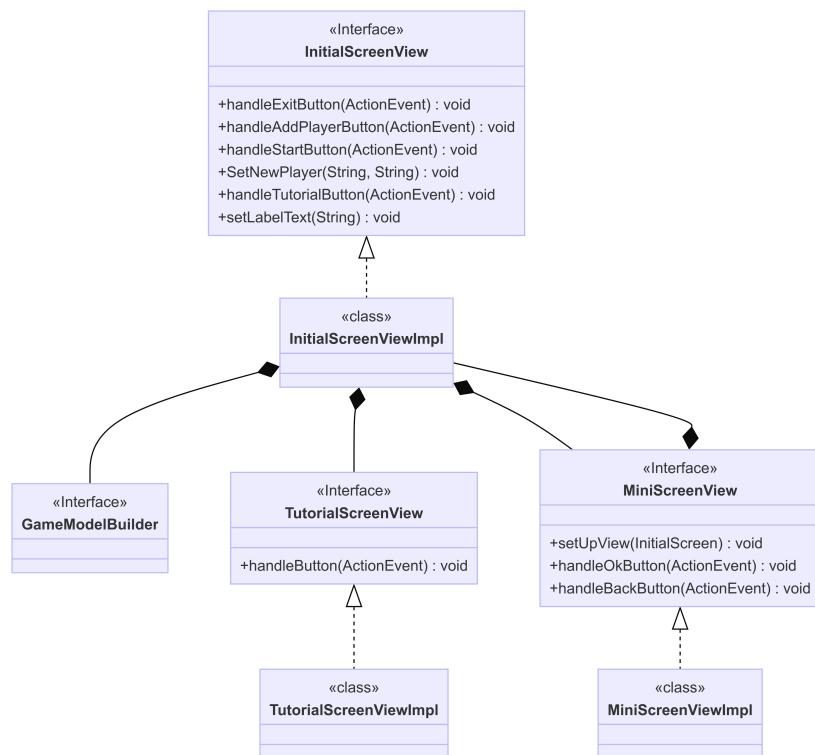


Figura 2.6: schema UML della schermata iniziale

**Soluzione:** per la gestione della partita ho utilizzato Scene Builder per progettare la schermata iniziale, essa include vari elementi dell’interfaccia utente, come un pulsante per aggiungere giocatori, selezionare la difficoltà, avviare il gioco e altra funzioni. Per selezionare la difficoltà, utilizzo l’enum `BoardType` sviluppata, la quale viene poi fornita al `GameModelBuilder` per gestire l’istanza di una nuova partita e operazioni correlate come la creazione di nuovi giocatori, il tutto nascosto dietro l’implementazione curata da Gabriele Fronzoni. Vengono utilizzate le classi `TutorialScreenView`, `MiniScreenView` per gestire le schermate pop up che si aprono quando si cliccano il pulsante per aggiungere un nuovo giocatore e il pulsante per le istruzioni.

Nota di sviluppo: nella classe `MiniScreenViewImpl` il plugin SpotBugs segnalava il seguente avviso: `"may expose internal representation by storing an externally mutable variable"`. Ho deciso di sopprimere il warning poichè, per come la classe era stata progettata la classe, era necessario aggiornare la variabile segnalata.

## 2.2.2 Ferretti Filippo

### Items

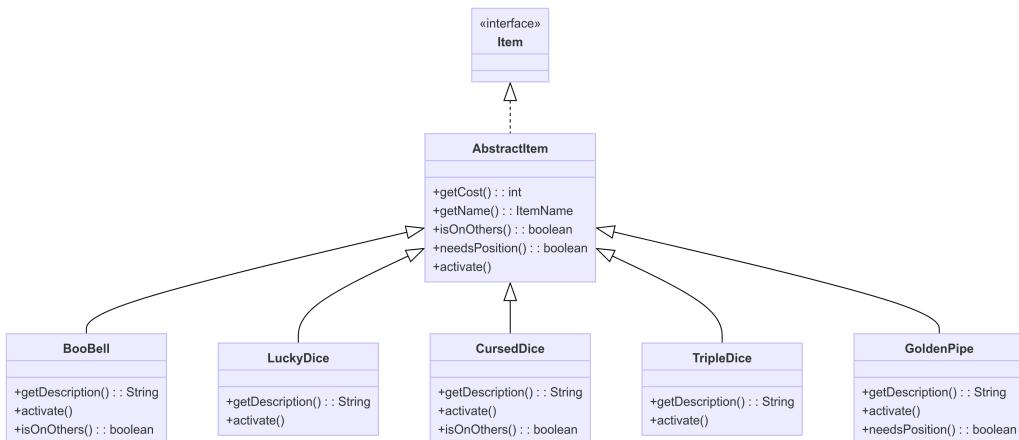


Figura 2.7: Implementazione degli Items

**Problema:** Implementazione degli item. Implementare gli item in modo che il loro utilizzo da parte dei `player` e il loro acquisto tramite lo `shop` venga gestito in modo uguale indifferentemente dal tipo di item.

**Soluzione:** Per risolvere il problema descritto precedentemente ho scelto di utilizzare il pattern **Template Method** per massimizzare il riuso, come da figura Figura 2.7. I metodi template descritti nella classe **AbstractItem** sono `getCost`, `getName`, `isOnOthers` e `needsPosition` che definiscono il comportamento di base per tutti gli item. A seconda degli item i metodi `needsPosition` e `isOnOthers` possono subire un override per definire un comportamento diverso dell'item. Per quanto riguarda invece `activate`, che definisce il vero e proprio comportamento dell'item, e `getDescription` questi vengono gestiti dalle classi specializzate di item. Questo approccio permette aggiunta di nuovi item facilmente avendo già a disposizione la classe astratta che fa da scheletro ad ogni item.

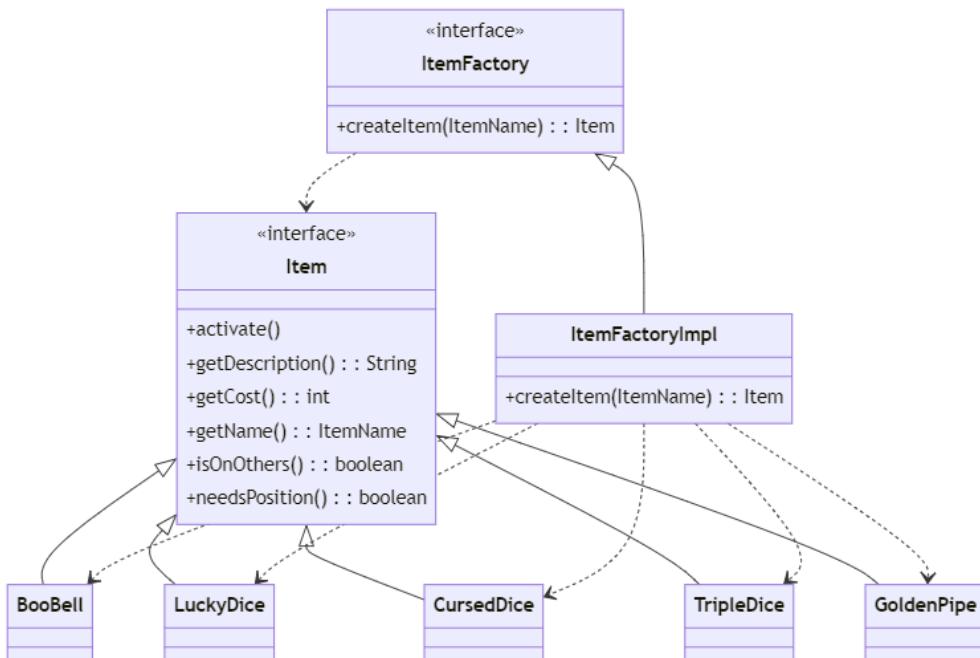


Figura 2.8: Creazione degli Items

**Problema:** Creazione degli **Items** dentro lo shop. Creare tutti i tipi di item all'interno dello shop rendendoli acquistabili al player.

**Soluzione:** Per creare gli Item ho deciso di utilizzare il pattern **Factory Method**, come da figura Figura 2.8. In questo modo la factory si occupa di creare e restituire un oggetto item diverso in base al parametro in ingresso. Questo rende molto facile creare un nuovo oggetto di tipo item permettendo così di aggiungere nuovi item allo shop facilmente.

## Gestione Negozio

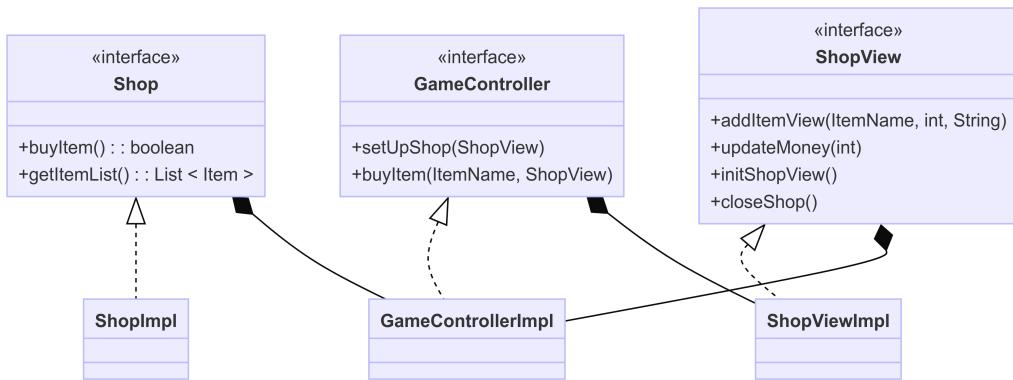


Figura 2.9: Gestione del Negozio

**Problema:** Gestione del Negozio all'interno del gioco. Implementazione del negozio integrandolo con il pattern MVC principale del gioco.

**Soluzione:** Per gestire il negozio ho utilizzato un pattern MVC che si integra all'interno del gioco principale, come da figura Figura 2.9. A differenza dei minigiochi ho scelto di non implementare un controller proprio del negozio ma di utilizzare il **GameController** del gioco principale, implementando lì i metodi necessari a coordinare **ShopView** e **ShopModel** del negozio e gestendo quindi la possibilità per i player di comprare nuovi items. Questo perchè volevo restituire sempre la stessa implementazione dello shop e non creare uno shop nuovo per ogni volta che il player finisce sulla relativa casella. A questo scopo avrei potuto implementare il pattern **Singleton**, ma ho preferito non farlo per avere maggiore elasticità e possibilità di aggiornamenti futuri. Infatti il negozio non deve essere necessariamente solo uno, infatti ci possono essere ad esempio negozi personalizzati per ogni player.

## Minigioco Forza 4

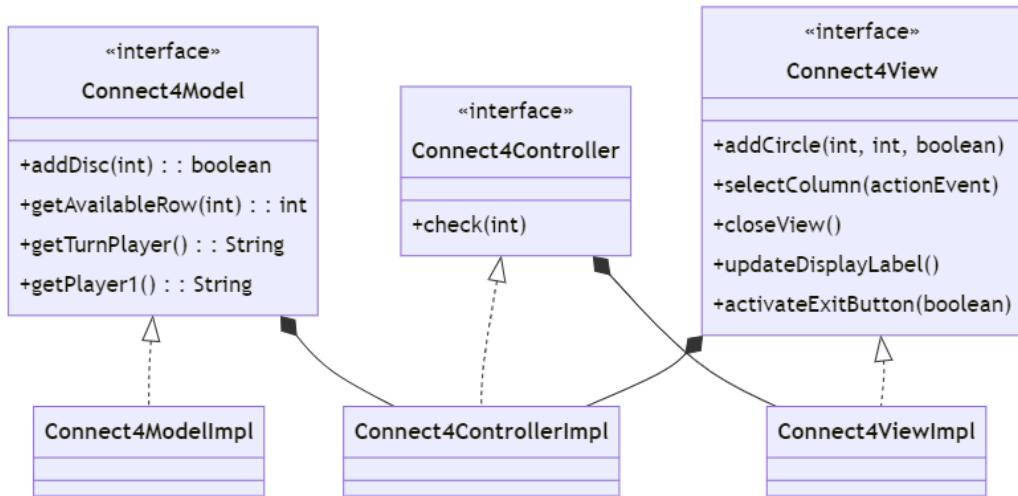


Figura 2.10: Gestione Minigioco Connect4

**Problema:** Implementazione Minigioco Connect4. Creazione di un minigioco multigiocatore uno contro uno che può essere giocato capitando nelle caselle minigioco.

**Soluzione:** Per realizzare questo minigioco ho utilizzato un pattern MVC come rappresentato nella Figura 2.10. Una volta caricata la view i due giocatori si sfideranno al gioco. Il **Connect4Controller** gestisce gli eventi della **Connect4View**, ad esempio la richiesta di aggiunta di una nuova pedina, notificando il **Connect4Model** e riaggiornando la view di conseguenza. All'interno del gioco è stata anche inserita la possibilità di leggere un tutorial con le indicazioni per i giocatori su come giocare. Il **Connect4Model** si occupa di gestire gli aspetti fondamentali della logica del gioco, come ad esempio il termine della partita o l'aggiunta di pedine. Al termine della partita viene mostrato il risultato a schermo e cliccando su esci viene notificato il **GameController** principale che gestirà i risultati e tornerà alla schermata del tabellone di gioco principale.

## 2.2.3 Fronzoni Gabriele

### Player

**Problema :** Creazione del Player

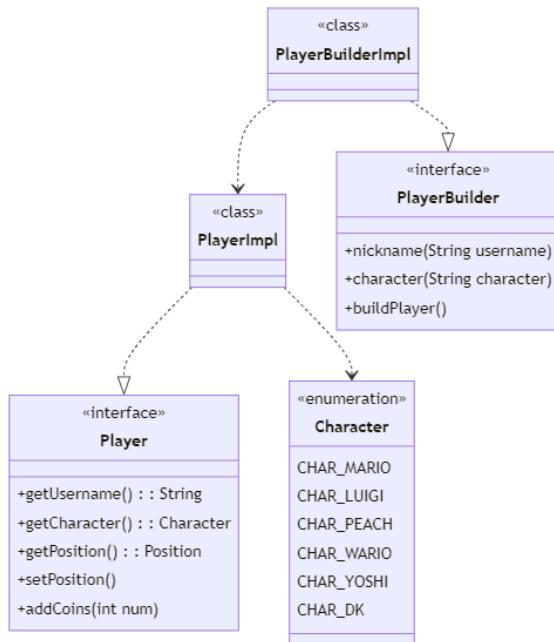


Figura 2.11: Rappresentazione dell'implementazione di Player e PlayerBuilder

**Soluzione:** Il **Player** viene identificato da username scelto in fase di inizializzazione della partita, oltre che da un personaggio ( **Character** ) tra quelli disponibili.

Ho pensato di utilizzare il pattern **Builder** per la costruzione di un oggetto player fatta passo per passo. Nel momento in cui viene chiamato il metodo `buildPlayer`, viene restituito un oggetto **Player** avente i parametri scelti. Ho deciso di usare questo approccio per fare sì che, nel momento in cui una certa classe ha bisogno di creare un **Player**, questa può farlo comodamente mediante l'utilizzo di un Builder e a questo vengono lasciati i controlli relativi alla creazione.

Inoltre, il **Player** avrà opportuni metodi per settare alcuni suoi parametri (es. posizione) e altri per ottenere informazioni utili nel corso della partita.

Per quanto riguarda le *stelle* e le *monete* che il player possiede, questi rappresentano degli attributi che possono essere modificati mediante opportuni

metodi nel corso della partita. Lo schema UML che rappresenta il concetto di Player e PlayerBuilder è rappresentato nell'immagine Figura 2.11

**Problema :** Gestione degli Items posseduti dal Player

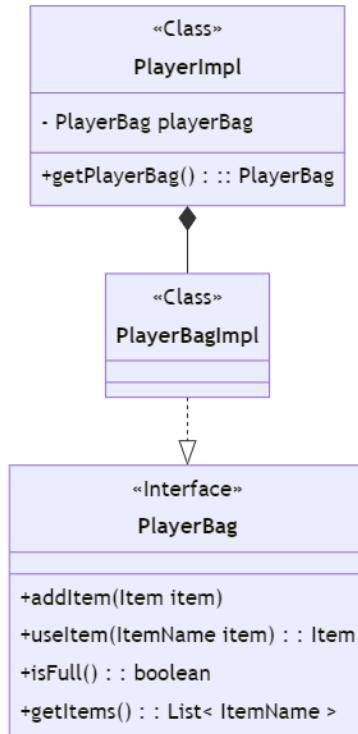


Figura 2.12: Gestione degli Items di un Player mediante PlayerBag

**Soluzione:** Per la gestione degli item che ha un certo player in un dato momento ho deciso di creare un oggetto PlayerBag che rappresenta l'inventario degli oggetti che ha il Player. La PlayerBag nel momento in cui viene creata, ha bisogno che gli venga indicato il massimo numero di Item che può contenere.

Nel momento in cui si decide di utilizzare un certo Item contenuto all'interno della PlayerBag, questo viene rimosso. Possono essere aggiunti anche più Item dello stesso tipo. In qualsiasi momento è possibile poi sapere quali sono gli items presenti all'interno della PlayerBag.

Ho deciso di separare il concetto per non andare ad aggiungere ulteriori campi e metodi al Player e rendere più comprensibile il tutto. La soluzione è illustrata nella figura Figura 2.12

**Problema :** Dado di ogni giocatore

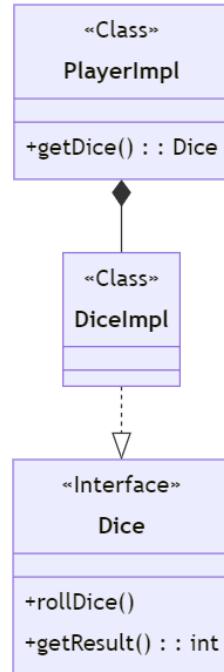


Figura 2.13: Implementazione di Dice e composizione di Player con quest'ultimo

**Soluzione:** Durante lo sviluppo mi sono reso conto, confrontandomi anche con chi implementava gli items, che gli item andavano a modificare il comportamento del dado per ogni singolo giocatore Ho quindi deciso di modellare l'oggetto **Dice** e di fare in modo che ogni giocatore ne possieda uno proprio al suo interno uno.

Questo viene fatto per far sì che nel momento in cui viene attivato un certo oggetto che modifica i dadi, questo vada a impattare il giocatore voluto e non si debba fare operazioni su di un unico dado comune, rendendo più complicato e contorto il tutto.

Il dado ha opportuni metodi per settare i propri parametri ( minimo, massimo, numero di tentativi ). Il dado ha un metodo per essere tirato. Una volta tirato memorizza il risultato, il quale può essere ottenuto mediante un metodo getter. Nel momento in cui il dado verrà modificato, al primo lancio dopo essere stato modificato verrà poi ripristinato con i valori di default. La soluzione al problema è illustrata in figura Figura 2.13

## Minigioco Memory Card

**Problema:** Gestione del pattern MVC all'interno del minigioco

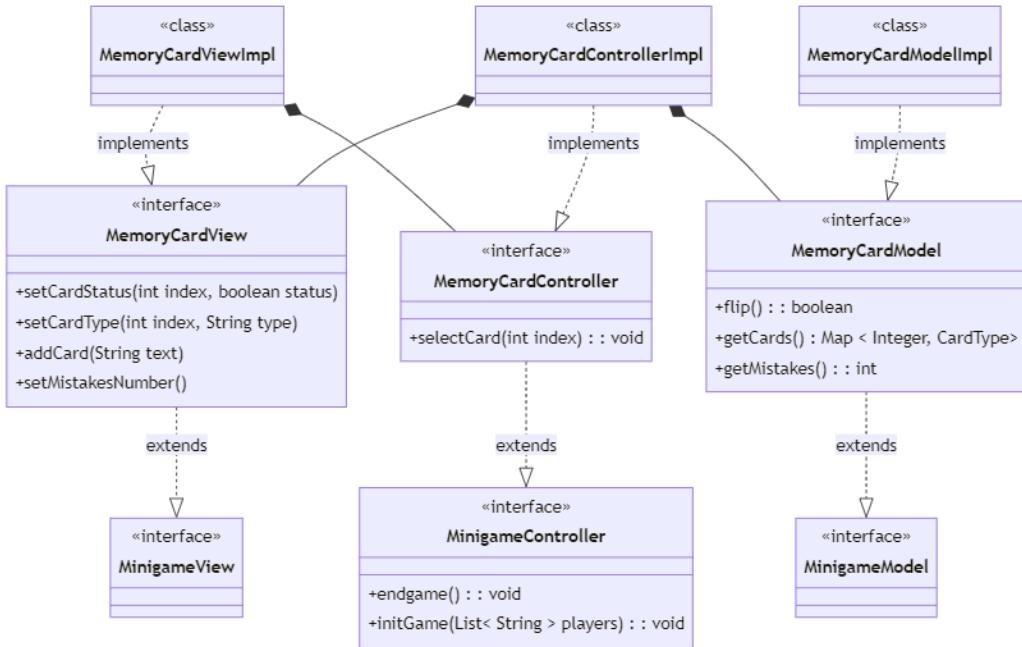


Figura 2.14: Implementazione pattern MVC per il minigioco Memory Card

**Soluzione:** Per quanto riguarda la view del minigioco Memory Card, quando parte il gioco permette all'utente di visualizzare come sono disposte le coppie all'interno del tabellone, dopodiché, quando viene deciso dall'utente, queste vengono coperte e parte il gioco.

Un giocatore prova ad indovinare la coppia e il `MemoryCardController` gestisce l'evento mandando prima il segnale al `MemoryCardModel` per aggiornarlo e in base al valore di ritorno di quest'ultimo, modifica la GUI chiamando i metodi di `MemoryCardView`.

Nel momento in cui termina la partita, che può accadere quando l'utente raggiunge il massimo numero di errori oppure quando tutte le coppie sono state trovate, il controller chiama il metodo del model per ottenere i risultati e poi modifica la view per mostrarli a schermo e notifica il controller principale del gioco il risultato del minigioco.

Il model nel corso del gioco, controlla se la coppia selezionata è giusta e tiene conto di tutti quelli che sono i tipi che sono stati indovinati.

Tutti le interfacce relative a Model-Control-View estendono quelle che sono le interfacce comuni a tutti i mini-giochi, andando ad ereditare i relativi metodi.

Lo schema UML che mostra l'organizzazione del minigame è mostrato in figura Figura 2.14.

## GameModel

**Problema:** Gestione dei minigiochi

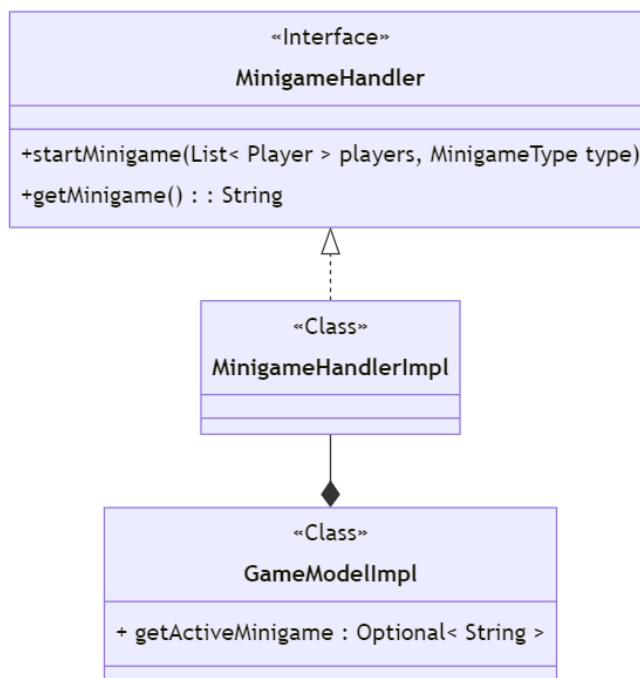


Figura 2.15: Gestione minigames nel GameModel mediante MinigameHandler

**Soluzione:** Dopo una attenta analisi, ho deciso di scorporare la gestione dei minigiochi in una classe separata chiamata `MinigameHandler`. Ho preso questa decisione per far sì che non venga appesantito il `GameModel` con la gestione dei minigiochi.

Il `MinigameHandler` ha il compito di far partire un minigame casuale in base al tipo richiesto ( single-player o multi-player ) e poi, mediante opportuni getter, restituire di quale minigame si tratta. Nel momento in cui viene fatto

partire il minigioco, viene anche passata una lista contenente tutti i player che partecipano al minigioco.

Il `MinigameHandler` per decidere quale mini-gioco è quello da far partire utilizza il meccanismo della `Reflection`: ispeziona quelle che sono le classi che implementano l'interfaccia `MinigameModel` (in quanto tutti i mini-giochi hanno un'interfaccia comune di riferimento). Poi controlla quali di queste sono mini-giochi del tipo necessario (single o multi-player) e poi sceglie in maniera casuale uno di quelli trovati.

Inoltre il `GameModel` ha un metodo che permette di sapere quale è il mini-gioco attualmente in esecuzione, andando ad interfacciarsi direttamente con il `MinigameHandler`. Il diagramma UML relativo alla gestione del problema è quello in figura Figura 2.15.

**Problema:** Creazione di una nuova partita

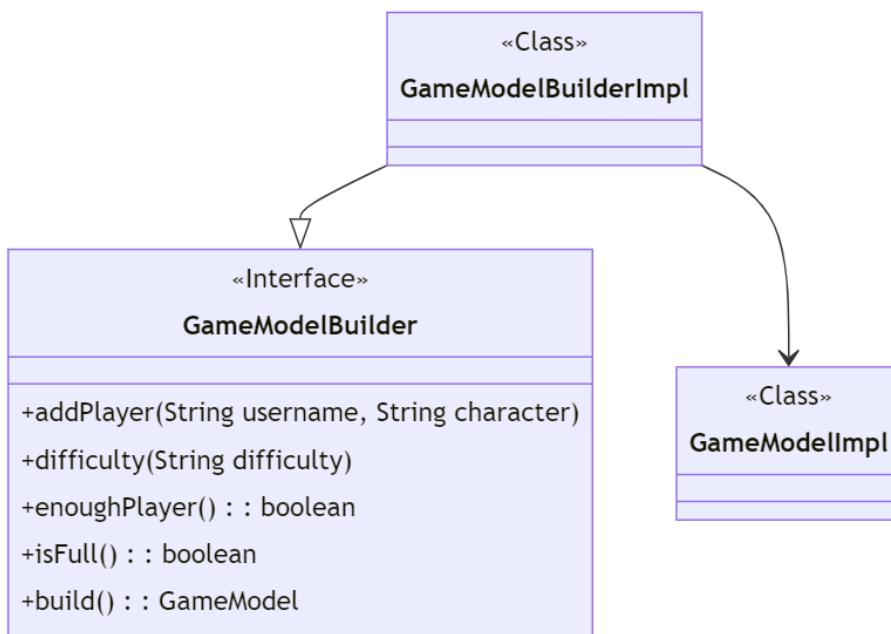


Figura 2.16: Creazione di una nuova partita mediante `GameModelBuilder`

**Soluzione:** Ho pensato che potesse essere utile creare una classe `GameModelBuilder` al fine di semplificare il processo di creazione di un nuova partita di gioco e per incapsulare il processo di creazione di una nuova partita.

La classe fornisce metodi per aggiungere player alla partita e per settare quella che è la difficoltà di gioco. Inoltre fornisce metodi per controllare se

ci sono abbastanza giocatori nella partita oppure se questa è piena e non possono essere aggiunti ulteriori giocatori.

Mediante il metodo build si ottiene una nuova istanza di `GameModelImpl` inizializzata con i valori forniti. Lo schema UML riassuntivo di quanto detto è quello in figura Figura 2.16

## 2.2.4 Ricci Nicholas

### Game Board

**Problema :** Creazione game board specifica

**Breve descrizione del problema :** Essendo che nel gioco originale vi sono più mappe disponibili, il problema era come gestire la creazione dell'eventuale tabellone selezionato.

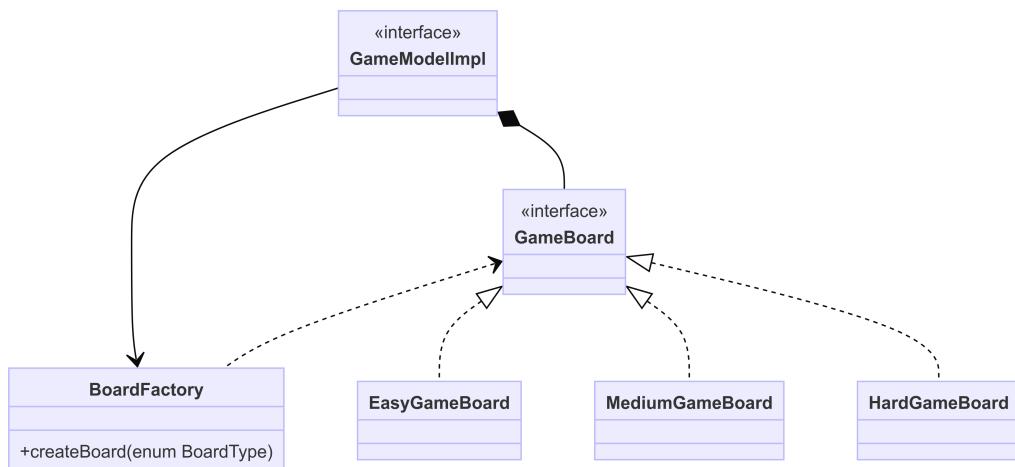


Figura 2.17: Creazione Game board selezionata

**Soluzione:** A primo impatto avevo considerato l'idea di utilizzare il factory method pattern, ma alla fine ho deciso di basare la mia implementazione su lo static factory pattern che sfrutta la classe `BoardFactory` ed il suo unico metodo statico `createBoard()` che restituisce una determinata `GameBoard` basandosi sull'enum `BoardType` preso in input.

Questo perchè il factory method pattern è utilizzato per modellare concetti complessi e classi che differiscono sostanzialmente nella loro logica di creazione.

Mentre nel caso della **GameBoard** è un concetto "semplice" che nella mia implementazione non prevede sostanziali differenze dal punto di vista di logica di creazione, ma nel comportamento dei diversi oggetti creati, argomento trattato nella prossima sezione. Infatti la mia situazione è agevolata dall'uso dello static factory method.

Anche se, quest'ultimo, è sconsigliato per gestire la creazione e gestione di molte tipologie di oggetti, perché la classe "factory" risulterebbe alla lunga difficile da mantenere e poco leggibile, nonostante questo pattern prediliga la semplicità attraverso l'uso di nomi descrittivi, in questo caso degli enum.

Questo problema però non sussiste nel momento in si sta implementando un gioco che non deve la propria fama alla grande quantità di tabelloni che prevede.

**Problema :** Differenziare il comportamento delle diverse game board

**Breve descrizione del problema :** Nel gioco ogni tabellone è caratterizzato da delle meccaniche uniche che potrebbero non avere gli altri tabelloni.

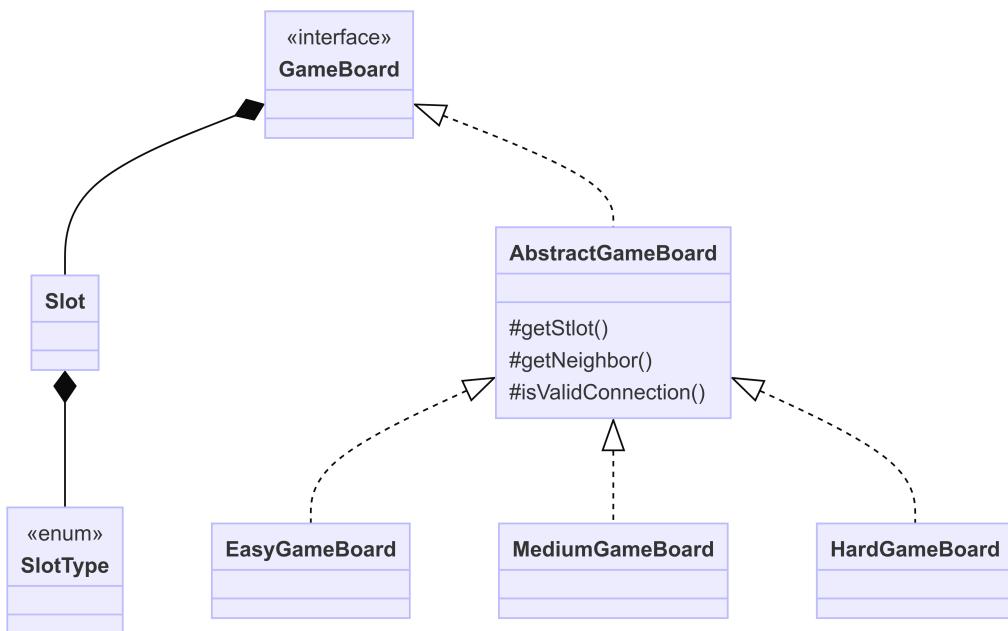


Figura 2.18: Differenziare il comportamento delle diverse game board

**Soluzione:** L'obiettivo iniziale era creare tabelloni con meccaniche diverse sfruttando il Template Method Pattern con dei metodi astratti nella classe

`AbstractGameBoard`. Per motivi di probabile mal gestione del tempo da parte mia l'unica differenza delle mappe `EasyGameBoard`, `MediumGameBoard` e `HardGameBoard` sono i valori passati al costruttore di `AbstractGameBoard`. Ma ho voluto lasciare il codice in modo che sia estendibile lasciando come `protected` i metodi che a mio avviso sono i più addatti per sfruttare questo pattern.

Per esempio, basandosi sulla mia idea iniziale, si possono sovrascrivere i metodi `getNeighbor()` e `isValidConnection` per implementare il movimento in diagonale all'interno del tabellone o rendere possibile dei "salti" tra caselle più lontane per quanto riguardava la `MediumGameBoard`.

Sempre basandomi sulla mia idea iniziale per quanto riguarda `HardGameBoard` ho lasciato il metodo `getSlot()`, che oltre ad essere utile per eventuali meccaniche di gioco da implementare, potrebbe permettere di creare un tabellone dinamico che varia durante la partita.

### **Minigioco Secret Code**

**Problema :** Minigioco Secret Code - Pattern MVC

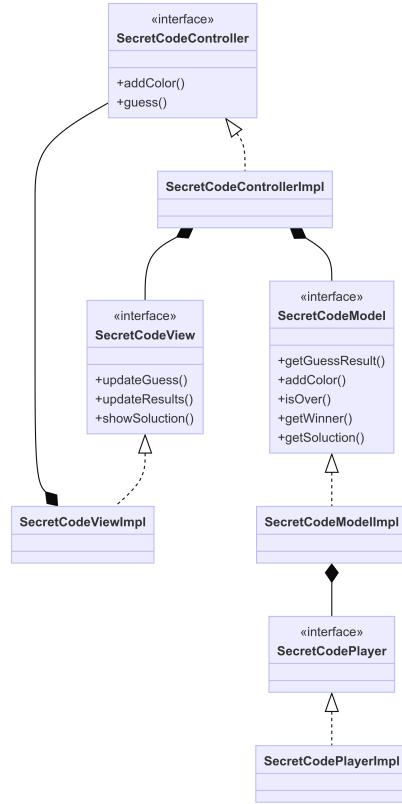


Figura 2.19: Implementazione Pattern MVC di Secret Code

**Soluzione:** Secret Cod è stato implementato seguendo il pattern architettonale MVC 2.19.

Di conseguenza, **SecretCodeModel** si occupa di gestire la logica del gioco e degli eventuali player. Perchè di fatto **SecretCodeModel** non ha un limite massimo di **SecretCodePlayer**, quindi potenzialmente riesce a gestire partite sia con giocatore singolo, sia con più di due giocatori.

**SecretCodeView** si occupa della rappresentazione grafica in base a gli aggiornamenti che deve eseguire impartiti dal **SecretCodeController**.

Quest'ultimo congiunge i primi due elementi, intercettando i comandi dell'utente nella view e comandandoli al model ed in base alla risposta aggiorna la view.

## 2.2.5 Yan Elisa

### Minigioco Nanogram

**Problema:** Gestione del pattern MVC del mini-gioco Nanogram

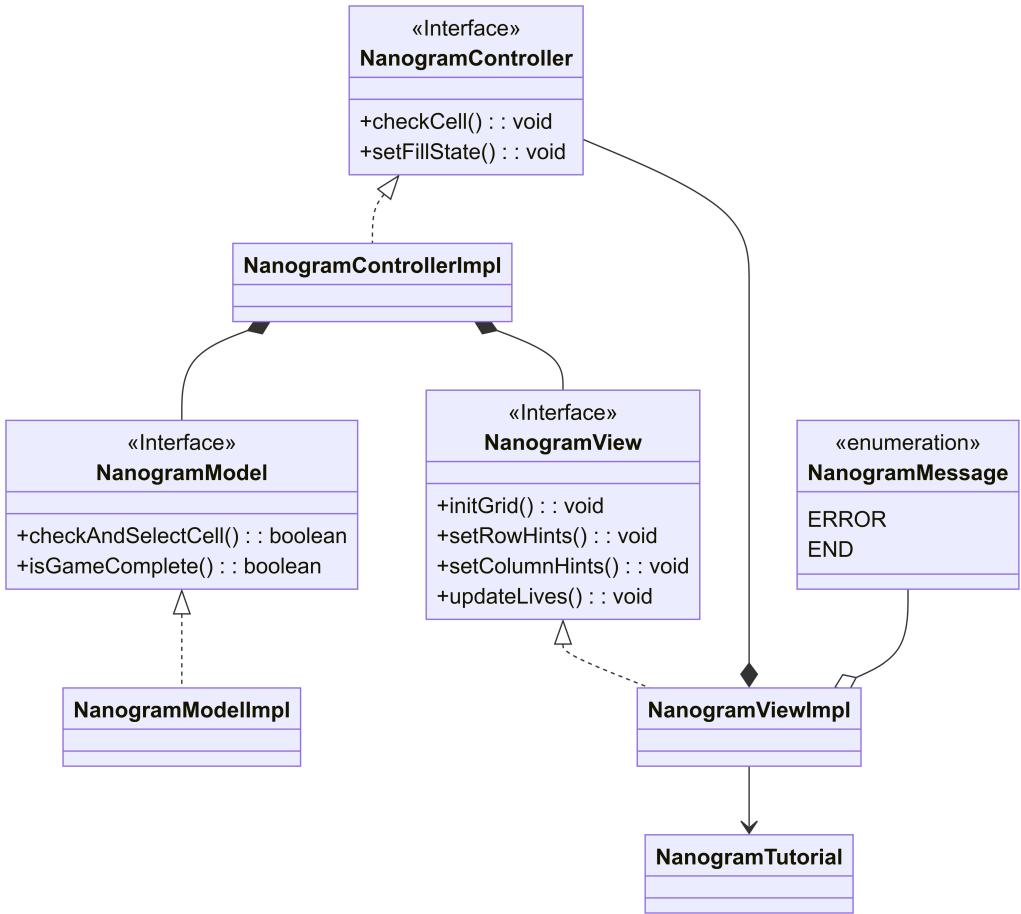


Figura 2.20: Rappresentazione del pattern MVC del mini-gioco Nanogram

**Soluzione:** Il mini-gioco Nanogram è stato progettato seguendo il pattern architettonale *MVC*, che suddivide l'applicazione in tre componenti principali: *NanogramModel*, *NanogramController* e *NanogramView*. Inoltre, implementa anche il pattern *Strategy* per gestire diverse logiche di controllo, rendendo il codice più leggibile e manutenibile. Questo design aderisce al principio SOLID della singola responsabilità. La suddivisione in componenti separati e l'uso di strategie per le logiche di controllo consentono di mantenere ogni classe focalizzata su un singolo compito, migliorando la modularità e la facilità di manutenzione del codice, come illustrato nella Figura 2.20.

*NanogramModel* gestisce la logica del gioco, compresa la gestione dello stato della griglia e delle vite del giocatore. Quando il giocatore seleziona una cella, il modello verifica la correttezza della scelta e aggiorna lo stato di conseguenza.

`NanogramView` è responsabile della rappresentazione grafica delle informazioni. Gestisce la visualizzazione della griglia di gioco, i suggerimenti per le righe e le colonne, i messaggi di stato del gioco e il numero di vite rimanenti. Quando il gioco inizia, viene inizializzata una griglia di bottoni che permette all’utente di colorare le celle seguendo i suggerimenti disposti sopra e lateralmente. La view aggiorna l’interfaccia utente in base alle notifiche ricevute dal `Nanogramcontroller`. La view include un pulsante che, quando cliccato, apre una nuova finestra che illustra il tutorial del gioco, gestito dalla classe `NanogramTutorial`. Questa finestra fornisce le istruzioni su come giocare.

Il `NanogramController` funge da intermediario tra la view (`NanogramView`) e il modello (`NanogramModel`). La view riceve gli input dall’utente, come la selezione di una cella, e li trasmette al controller. Il controller, a sua volta, li passa al model per la verifica. In base alla risposta del `NanogramModel`, il `NanogramController` aggiorna la `NanogramView` per riflettere i cambiamenti. Questo processo garantisce che l’interfaccia utente sia sempre sincronizzata con lo stato attuale del gioco, offrendo un’esperienza utente coerente e reattiva.

La partita termina se l’utente esaurisce le vite a disposizione o se riesce a colorare correttamente tutte le celle richieste.

**Gestione del NanogramModel** La logica del modello di gioco per Nanogram è suddivisa in diverse componenti, ognuna delle quali svolge un ruolo specifico, ottimizzando così il funzionamento del modello complessivo. Questa struttura modulare facilita la manutenzione e l’espansione del gioco. Nella Figura 2.21 viene presentata una rappresentazione UML di questa suddivisione, che evidenzia le principali componenti e le loro interazioni.

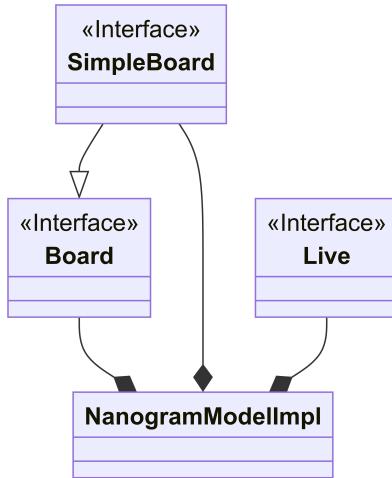


Figura 2.21: Rappresentazione della gestione del Model del mini-gioco Nanogram

**Problema:** Gestione del board del mini-gioco Nanogram

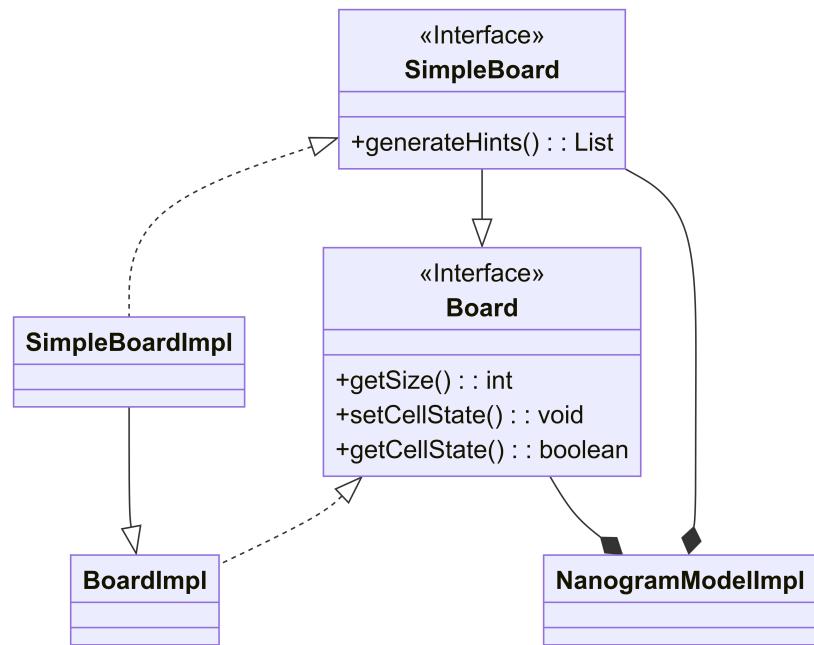


Figura 2.22: Rappresentazione del board del mini-gioco Nanogram

**Soluzione:** Per affrontare questo problema, è stato utilizzato il design pattern *Decorator*. Questo pattern permette di estendere le funzionalità di una classe in modo dinamico, senza dover modificare il codice originale.

L'entità **Board** è responsabile della memorizzazione dello stato di ogni cella nella griglia di gioco e fornisce metodi per impostare e recuperare lo stato delle celle.

L'entità **SimpleBoard** decora la **Board**, aggiungendo funzionalità come l'inizializzazione casuale dello stato delle celle e la generazione di suggerimenti basati su tale inizializzazione. Questa classe utilizza la composizione per estendere le capacità della **Board** senza modificarla direttamente.

In futuro, questo approccio consentirà di implementare ulteriori decorazioni per introdurre nuove modalità di difficoltà o altre funzionalità avanzate senza dover ristrutturare il codice esistente, mantenendo così il sistema manutenibile e facilmente estensibile.

**Problema:** Gestione delle vite nel mini-gioco Nanogram

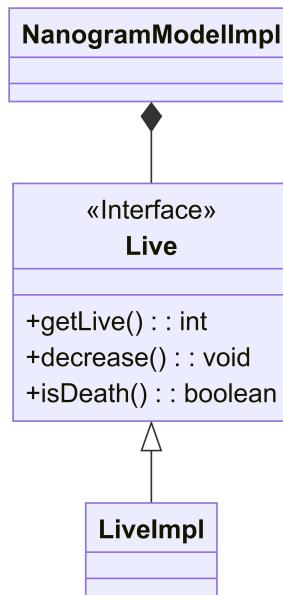


Figura 2.23: Rappresentazione delle vite del mini-gioco Nanogram

**Soluzione:** Per risolvere il problema è stata progettata una struttura che decomponga la logica di gioco, rappresentata da **NanogramModel**, dalla logica della gestione delle vite, implementata dall'entità **Live**.

L'entità `Live` gestisce lo stato delle vite del giocatore. Include metodi per ottenere il numero di vite rimanenti, diminuirle e verificare se tutte le vite sono state esaurite.

Questa separazione consente una maggiore manutenibilità e flessibilità, permettendo modifiche e miglioramenti a ciascun componente senza influenzare gli altri.

## Minigioco Domino

**Problema:** Gestione del pattern MVC del mini-gioco Domino

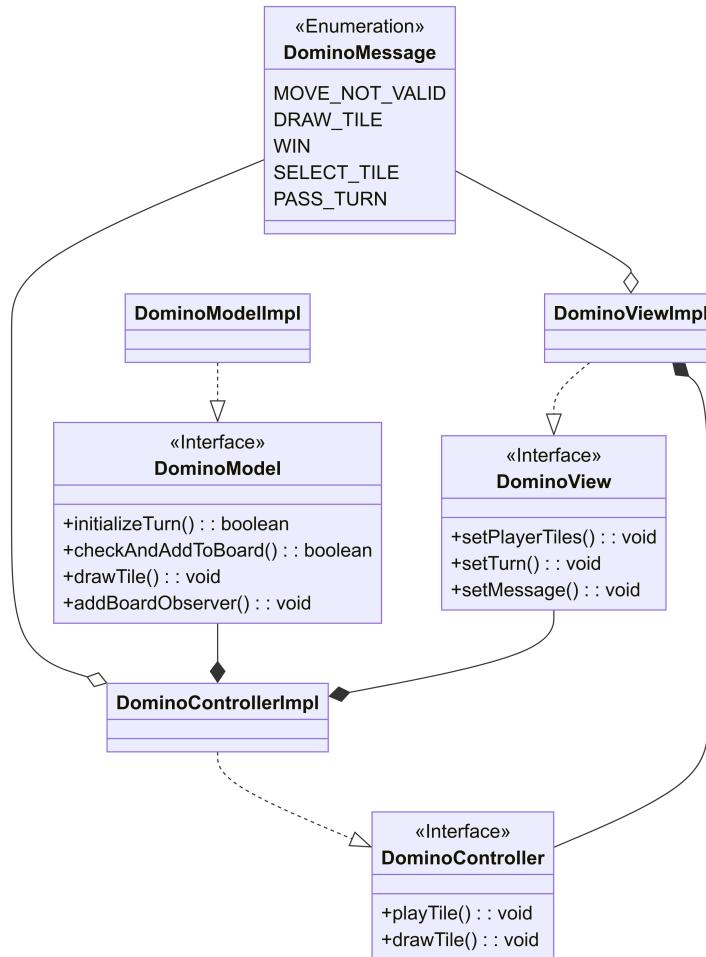


Figura 2.24: Rappresentazione del pattern MVC del mini-gioco Domino

**Soluzione:** Per la realizzazione del mini-gioco Domino, è stato utilizzato il pattern *MVC* perchè consente l'indipendenza tra il modello e la visualizza-

zione, rendendo più semplici eventuali modifiche future. Il gioco presenta un suo model, `DominoModel`, un suo controller, `DominoController` e una sua view, `DominoView`. Oltre al pattern *MVC*, è stato anche utilizzato il pattern *Strategy* per gestire diverse strategie di gioco e il pattern *Observer* per osservare e aggiornare gli eventi, come illustrato nella Figura 2.24.

Per l'implementazione del gioco, si è iniziato definendo la struttura del `DominoModel`. Questo componente gestisce la logica del gioco, inclusa la gestione delle tessere, la verifica delle mosse e il controllo dello stato del gioco. Quando un giocatore seleziona una tessera, il modello verifica la correttezza della mossa e aggiorna lo stato del gioco di conseguenza.

`DominoController` funziona da intermediario tra il model e la view: quando il giocatore gioca una tessera, questa viene notificata al model tramite il metodo `playTile`. In base alla risposta del model, il controller aggiorna la view per riflettere i cambiamenti. Il controller gestisce anche la logica di pesca delle tessere tramite il metodo `drawTile`.

`DominoView` rappresenta la view del gioco e si occupa di interagire con l'utente. Gestisce la visualizzazione del board di gioco, i messaggi di stato del gioco e le tessere rimanenti. A ciascun pulsante presente sulla view è associato un'azione che, alla pressione del pulsante, interagisce con il controller per verificare la mossa. La view viene aggiornata su ordine del controller per riflettere i cambiamenti nel gioco.

**Problema:** Gestione delle tessere sulla board, il problema si concentra sulla necessità di aggiornare la view ogni volta che una tessera viene aggiunta alla board.

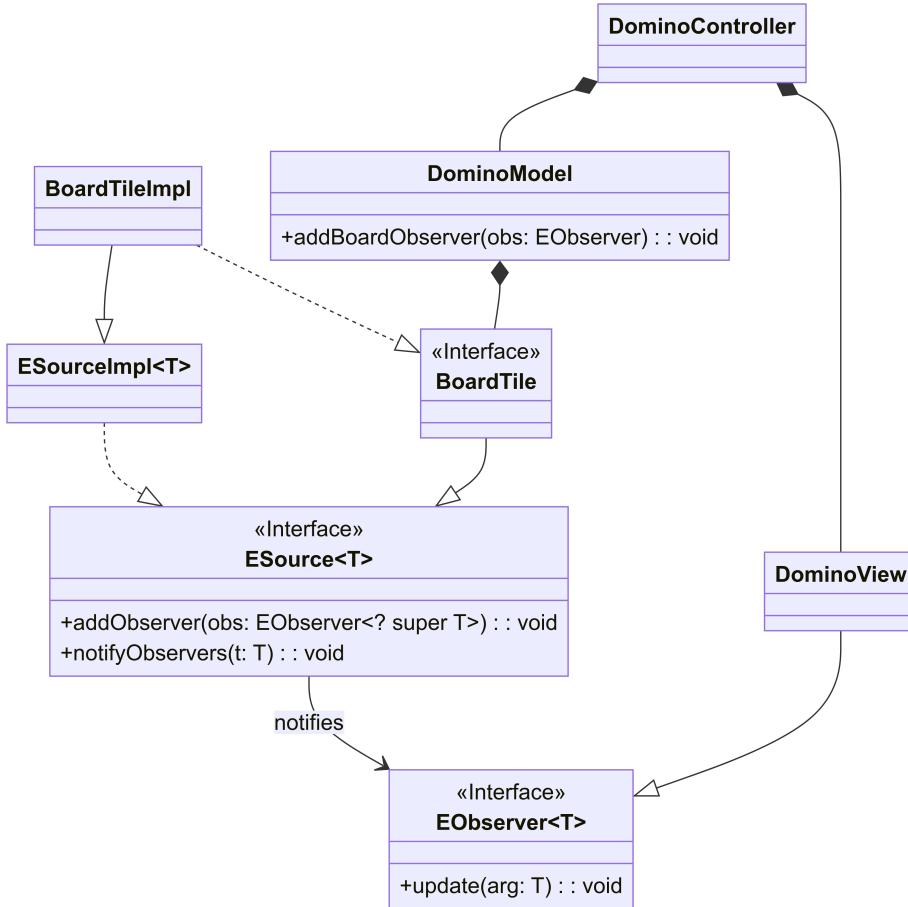


Figura 2.25: Rappresentazione del pattern Observer del mini-gioco Domino

**Soluzione:** Per affrontare questo aspetto, viene utilizzato il pattern *Observer*, che permette a un oggetto di notificare automaticamente un insieme di oggetti dipendenti ogni volta che il suo stato cambia. In questo contesto specifico, il soggetto è la `Board` che contiene le tessere, e l'osservatore è il `DominoView`, che deve essere aggiornato ogni volta che una tessera viene aggiunta alla board.

Quando una tessera viene aggiunta alla board, viene chiamato il metodo `notifyObservers` di `ESourceImpl`. Questo metodo invoca il metodo `update` di tutti gli osservatori, passando le informazioni necessarie per aggiornarli tra cui il `DominoView`. `DominoView` registra tutti i cambiamenti delle tessere aggiunte alla board, assicurando che lo stato della view sia sempre sincronizzato con quello della board.

**Gestione del DominoModel** La logica del gioco è suddivisa in varie componenti, ognuna delle quali svolge un compito specifico, alleggerendo così l'incarico del modello stesso. Di seguito nella Figura 2.26 viene illustrata una semplice suddivisione rappresentata in UML.

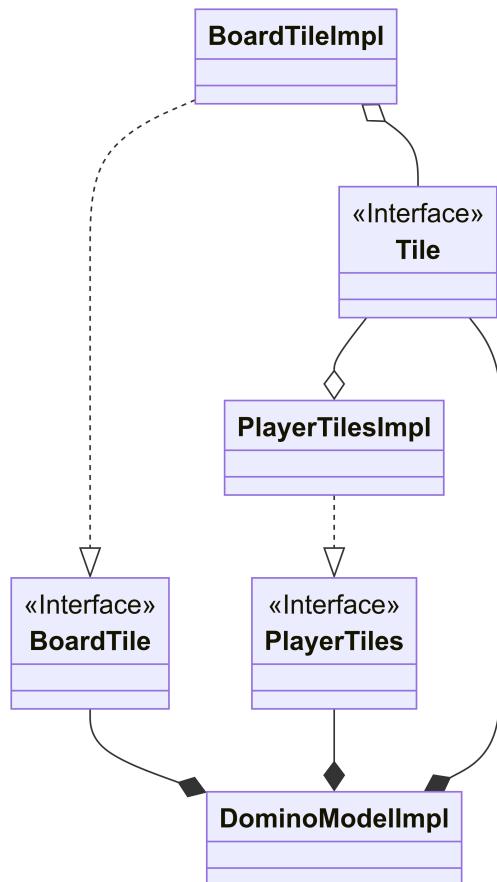


Figura 2.26: Rappresentazione della gestione del Model del mini-gioco Domino

**Problema:** Gestione del board nel mini-gioco Domino

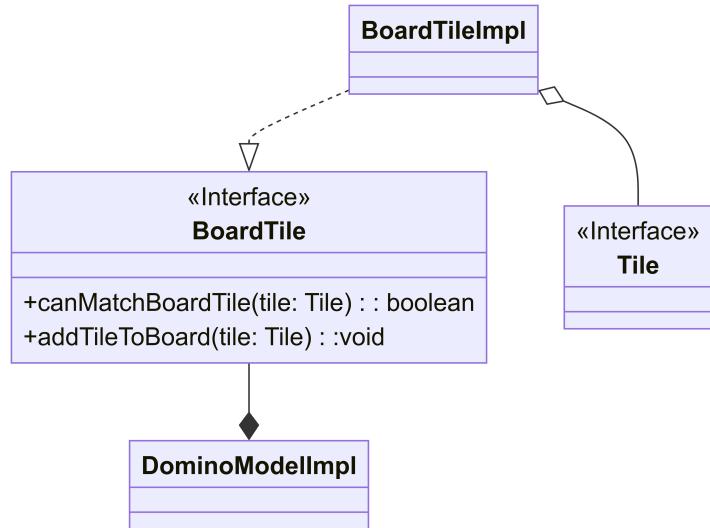


Figura 2.27: Rappresentazione della board del mini-gioco Domino

**Soluzione:** Il problema principale è gestire l'inserimento delle tessere nel gioco del Domino, garantendo che ogni volta che una tessera viene aggiunta al board, si verifichino le combinazioni corrette con le tessere già presenti.

L'entità **BoardTile** definisce i metodi `canMatchBoardTile` e `addTileToBoard` per verificare se una tessera può essere aggiunta al board e per aggiungerla.

**Problema:** Gestione delle tessere nel mini-gioco Domino

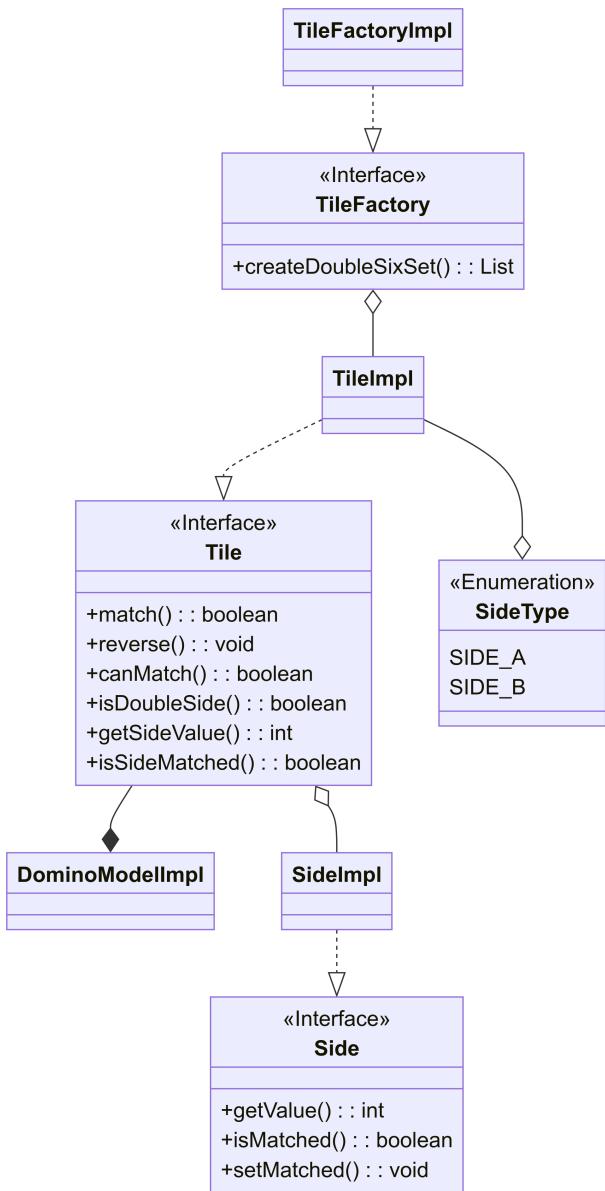


Figura 2.28: Rappresentazione delle tessere nel minigioco Domino

**Soluzione:** La gestione delle tessere è fondamentale nel mini-gioco Domino, poichè l'intero gioco si basa sulle interazioni tra queste. Per risolvere questo problema, è stata adottata un'architettura che utilizza il pattern di design *Factory* per la creazione delle tessere. Questo pattern permette di estendere le funzionalità delle tessere senza modificare le classi esistenti, supportando così diverse modalità di gioco.

`TileFactory` utilizzata per la creazione delle tessere. Il metodo `createDoubleSixSet` genera un set di tessere `Tile` per il gioco del Domino standard, adatto per due giocatori.

`Tile` rappresenta una tessera del Domino. Ogni tessera è composta da due lati `Side`, ciascuno dei quali è di tipo `SideType`. Include vari metodi per gestire e manipolare le tessere come illustrato nella Figura 2.28.

`Side` rappresenta un lato della tessera. Include metodi per ottenere il valore del lato, verificare se è stato abbinato e impostare lo stato di abbinamento.

**Problema:** Gestione delle tessere dei giocatori nel mini-gioco Domino

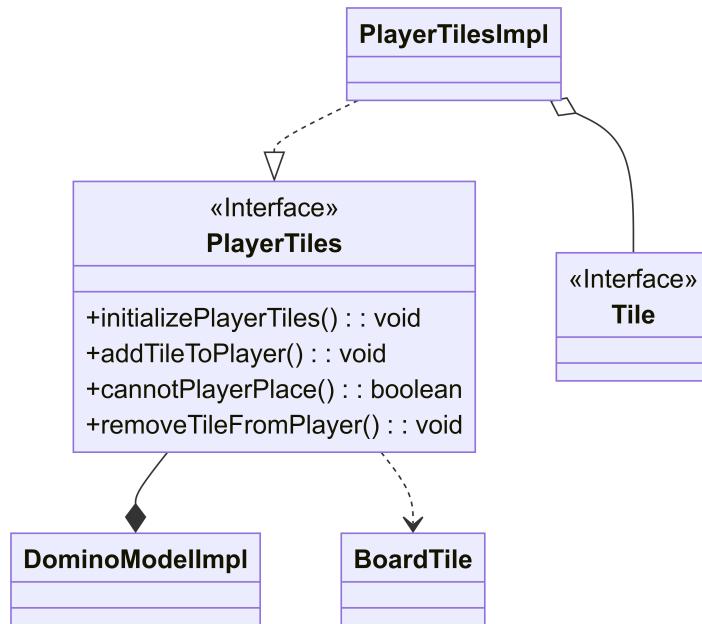


Figura 2.29: Rappresentazione delle tessere dei giocatori nel mini-gioco Domino

**Soluzione:** Per gestire le tessere possedute dai giocatori nel gioco del Domino, si è deciso di memorizzarle in una mappa, dove per ogni giocatore è memorizzato un `Set` di tessere. Questo approccio permette di tracciare facilmente le tessere disponibili per ciascun giocatore.

Uno dei problemi principali è di controllare il blocco del gioco, ovvero la situazione in cui nessun giocatore ha tessere giocabili. Questo problema viene risolto utilizzando il metodo `cannotPlayerPlace`, che verifica se le tessere possedute dal giocatore sono compatibili con quelle presenti sul tavolo.

Se nessun giocatore può piazzare nessuna tessera, la partita è considerata bloccata e terminata.

### Fine gioco

**Problema:** Gestione della fine partita e calcolo dei risultati di ogni giocatore

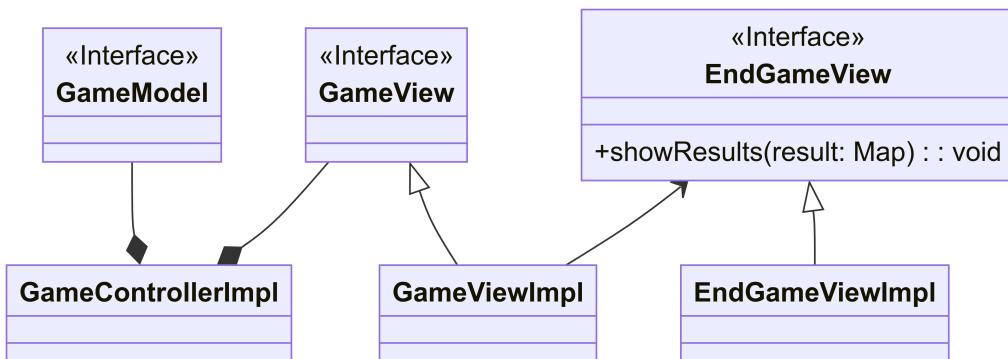


Figura 2.30: Rappresentazione della schermata finale del gioco principale

**Soluzione:** Per risolvere questo problema, è stato utilizzato il pattern *MVC* del gioco principale, come illustrato in Figura 2.30.

*GameModel* è responsabile di determinare quando la partita è finita.

*GameController* riceve i risultati dei giocatori dal *GameModel* e li ordina in base al numero di stelle e, in caso di parità, al numero di monete. Passa i risultati ordinati a *EndGameView* sotto forma di una mappa che contiene i punteggi e le classifiche di ogni giocatore.

*GameView* aggiorna l'interfaccia utente in base alle notifiche ricevute dal controller e mostra i risultati alla fine della partita. *EndGameView* implementa un metodo *showResults* che prende una mappa (*Map*) come argomento e presenta una classifica dei giocatori basata sui risultati finali.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Per la verifica del corretto funzionamento delle diverse componenti dell'applicazione sono stati creati Test automatizzati usando JUnit.

#### Ariyo Daniel

- **TestPerilousPath** : Testa il corretto funzionamento della logica del mini-gioco PerilousPath
- **TestMemorySweepImpl** : Testa il corretto funzionamento della logica del mini-gioco Memory Sweep
- **TestAbstractPositionImpl** : Testa la logica riguardante le implementazioni della classe astratta AbstractPosition

#### Ferretti Filippo

- **TestItemImplementation**: controlla che l'implementazione degli item sia corretta e che i suoi metodi restituiscano i risultati attesi;
- **TestItemFactoryImpl**: controlla che la `ItemFactory` crei correttamente l'item richiesto.
- **ShopImplTest**: controlla il corretto funzionamento dello shop che permetta quindi al player di acquistare un item se possibile.
- **Connect4Test**: Testa il corretto funzionamento della logica del mini-gioco Connect4.

## Fronzoni Gabriele

- **TestPlayerImpl** : controlla che il player venga inizializzato correttamente e che variazioni ai diversi attributi vengano correttamente registrate;
- **PlayerBuilderImplTest**: controlla che il **PlayerBuilder** funzioni correttamente con le opportune informazioni e fallisca invece con le informazioni sbagliate;
- **PlayerBagImplTest**: controlla che la player bag funzioni correttamente quando si vogliono aggiungere oppure usare degli oggetti;
- **DiceImplTest**: controlla che i metodi che modificano il funzionamento del dado funzionino correttamente e che il tiro del dado generi un numero in maniera giusta;
- **TestMemoryCardImpl**: controlla che la logica del minigioco **Memory Card** funzioni correttamente;

## Ricci Nicholas

- **TestGameBoard**: Testa la factory creando diverse tipologie di board per poi testarle sotto tutti i loro aspetti;
- **TestSecretCode**: Testa che la logica di gioco sia corretta e coerente con le aspettative inoltre a testare i **SecretCodePlayer**.

## Yan Elisa

- **LiveImplTest**: Verifica la corretta gestione delle vite nel mini-gioco **Nanogram**.
- **NanogramModelImplTest**: Viene testata il corretto funzionamento della logica del mini-gioco **Nanogram**.
- **SimpleBoardImplTest**: Verifica il corretto riempimento della board e la generazione dei suggerimenti del minigioco **Nanogram**.
- **BoardTileImplTest**: Verifica la corretta gestione delle tessere del domino sul board nel mini-gioco.
- **DominoFactoryImplTest**: Verifica la corretta generazione delle tessere del mini-gioco **Domino**.

- **DominoModelImplTest**: Verifica il corretto funzionamento della logica del minigioco Domino.
- **PlayerTilesImplTest**: Verifica la corretta gestione delle tessere dei giocatori nel minigioco Domino.
- **TileImplTest**: Verifica la corretta gestione delle tessere nel gioco del domino, inclusi i metodi di confronto, inversione e controllo delle tessere corrispondenti.

## 3.2 Note di sviluppo

### 3.2.1 Ariyo Daniel

**utilizzo di Stream e lambda functions** : Utilizzato in vari punti. Un esempio è: <https://github.com/fronzofronzo/OOP23-m-party/blob/438f1a94e76cfb43d4bdbe5f6294b6bd1d43a9b6/src/main/java/it/unibo/mparty/model/minigames/perilouspath/impl/BombPosition.java#L45>

**utilizzo della libreria JavaFX:** utilizzo della libreria per la costruzione della view dei minigiochi e la schermata iniziale:<https://github.com/fro nzofronzo/OOP23-m-party/blob/438f1a94e76cfb43d4bdbe5f6294b6bd1d43a9b6/src/main/java/it/unibo/mparty/view/minigames/memorysweep/MemorySweepViewImpl.java#L50>

### 3.2.2 Ferretti Filippo

**Utilizzo di lambda expressions:** Utilizzato in vari punti. Esempio: <https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/controller/GameControllerImpl.java#L111-L112>

**Utilizzo di Stream:** Utilizzato in diverse sezioni di codice. Esempio: <https://github.com/fronzofronzo/OOP23-m-party/blob/2492854735400bcf465fa022d35ab08e29b788c7/src/main/java/it/unibo/mparty/model/shop/impl/ShopImpl.java#L36>

**Utilizzo di Optional:** Usato nel metodo activate degli Item. Esempio: <https://github.com/fronzofronzo/OOP23-m-party/blob/2492854735400bcf465fa022d35ab08e29b788c7/src/main/java/it/unibo/mparty/model/item/impl/GoldenPipe.java#L25-L26>

**Utilizzo della libreria JavaFX:** Per realizzare le view dello shop e del mio minigioco. Esempio: <https://github.com/fronzofronzo/OOP23-m-party/blob/2492854735400bcf465fa022d35ab08e29b788c7/src/main/java/it/unibo/mparty/view/minigames/connect4/impl/Connect4ViewImpl.java#L86-L90>

### 3.2.3 Fronzoni Gabriele

**Utilizzo di Stream e lambda functions:** Utilizzato in vari punti. Un esempio è: <https://github.com/fronzofronzo/OOP23-m-party/blob/15176493d317828ed6eb70035843abed91c91f50/src/main/java/it/unibo/mparty/model/player/impl/PlayerBagImpl.java#L44>

**Utilizzo della libreria grafica JavaFX:** Utilizzo della libreria per la costruzione della view del minigioco e delle parti comuni. Esempio : <https://github.com/fronzofronzo/OOP23-m-party/blob/15176493d317828ed6eb70035843abed91c91f50/src/main/java/it/unibo/mparty/view/minigames/memorycard/MemoryCardViewImpl.java#L63>

**Utilizzo di Optional:** Utilizzati in diversi parte del codice. Un esempio: <https://github.com/fronzofronzo/OOP23-m-party/blob/15176493d317828ed6eb70035843abed91c91f50/src/main/java/it/unibo/mparty/model/GameModelImpl.java#L207>

**Utilizzo della libreria Reflections:** Utilizzato per scelta dei minigiochi : <https://github.com/fronzofronzo/OOP23-m-party/blob/15176493d317828ed6eb70035843abed91c91f50/src/main/java/it/unibo/mparty/model/minigamehandler/MinigameHandlerImplementation.java#L76>

### 3.2.4 Ricci Nicholas

**Progettazione con generici:** <https://github.com/fronzofronzo/OOP23-m-party/blob/0c2b170f9c2723a79a7cd168c7fc904949cdb732/src/main/java/it/unibo/mparty/model/gameboard/util/RandomListGenerator.java#L27>

<https://github.com/fronzofronzo/OOP23-m-party/blob/0c2b170f9c2723a79a7cd168c7fc904949cdb732/src/main/java/it/unibo/mparty/utilities/RandomFromSet.java#L24>

**Utilizzo di Stream e Lambda expressions:** Un esempio: <https://github.com/fronzofronzo/OOP23-m-party/blob/b9714a10af8c8292d253265242516f1fd3d4f8d8/src/main/java/it/unibo/mparty/model/gameboard/impl/AbstractGameBoardImpl.java#L99> Utilizzato in vari punti. Un esempio è:

**Utilizzo di Optional:** <https://github.com/fronzofronzo/OOP23-m-party/blob/c04631d82c13693c7ec1bc14e91ddecff4fd8261/src/main/java/it/unibo/mparty/model/GameModelImpl.java#L81>

**Classe anonima:** <https://github.com/fronzofronzo/OOP23-m-party/blob/9b3540adeb6e2ab2645fa03013fea2b8d7653e3d/src/main/java/it/unibo/mparty/model/minigames/secretcode/impl/SecretCodeModelImpl.java#L192>

**Utilizzo della libreria JavaFX:** Utilizzato per costruire le view del minigioco e la schermata del tabellone: <https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/view/gameboard/GameBoardViewImpl.java#L1>

<https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/view/minigames/secretcode/SecretCodeViewImpl.java#L1>

### 3.2.5 Yan Elisa

**Progettazione con generici:** Utilizzato nell'interfaccia ESource e nelle sue sottoclassi: <https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/utilities/impl/ESourceImpl.java#L14>

**Utilizzo di Lambda expressions:** Utilizzato in vari punti. Un esempio è: <https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/model/minigames/domino/tile/impl/TileFactoryImpl.java#L24-L26>

**Utilizzo di Stream:** Utilizzato in vari punti. Un esempio è: <https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/controller/GameControllerImpl.java#L143-L153>.

**Utilizzo di Optional** Utilizzato in: <https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/model/minigames/domino/tile/impl/TileImpl.java#L142-L149>

**Utilizzo della libreria JavaFX:** Utilizzato per costruire le view dei minigiochi e la schermata finale. Un esempio è: <https://github.com/fronzofronzo/OOP23-m-party/blob/06f54dae38ce98c485fd3f756f4431f10f803c18/src/main/java/it/unibo/mparty/view/minigames/nanogram/impl/NanogramViewImpl.java#L92>

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Ariyo Daniel

Mi ritengo piuttosto soddisfatto del lavoro svolto. Come per tutti i ragazzi del gruppo era il primo progetto di questo tipo, quindi l'inesperienza si è fatta sentire. Le difficoltà incontrate sono state tante ma credo di averle superate positivamente. Lavorare con strumenti come Git, JavaFX e Scene Builder inizialmente mi aveva messo in difficoltà ma con il tempo, e grazie all'aiuto dei miei compagni sono riuscito ad apprendere le conoscenze basilari di questi strumenti. Nonostante sono soddisfatto del progetto in generale, sono consapevole che le soluzioni implementate ai problemi affrontati potrebbero non essere ottimali. Tuttavia, considerando che all'inizio del corso non mi sarei mai aspettato di arrivare fino a creare un gioco, ritengo il risultato positivo. La cosa più importante che questo progetto mi ha lasciato è stata la crescita come programmatore, migliorando le mie abilità tecniche, ma anche la capacità di relazionarmi e collaborare con un gruppo. L'obiettivo dei progetti è proprio quello di far crescere una persona sotto questi punti di vista e anche per questo è stato così.

#### 4.1.2 Ferretti Filippo

Mi ritengo abbastanza soddisfatto del mio lavoro. Non era un progetto facile, nonché il primo così importante con cui mi approcciavo, ma devo dire che dopo alcune difficoltà iniziali sono riuscito a gestire bene la mia parte coadiuvandola con quelle svolte dai miei compagni. Credo che ci sia stato un ottimo lavoro di gruppo dove ognuno è riuscito ad arricchire il progetto con le proprie conoscenze. Avendo avuto più tempo a disposizione forse qualcosa-

na avrei cercato di perfezionarla o realizzarla un po' diversamente, ma tutto sommato sono abbastanza soddisfatto del mio lavoro. Questo progetto mi ha aiutato molto ad ampliare il mio bagaglio da programmatore, cercando sempre di trovare la soluzione migliore ai problemi che abbiamo affrontato. Ho inoltre approfondito e migliorato l'utilizzo di alcuni strumenti come Git o librerie esterne come JavaFX, i quali sono stati inizialmente complicati da capire per utilizzarli nel modo corretto. Quindi questo progetto è stato impegnativo però credo che mi abbia aiutato molto a migliorare e stimolato molto verso l'ampliamento delle mie conoscenze e abilità in futuro.

#### **4.1.3 Fronzoni Gabriele**

In generale, mi ritengo abbastanza soddisfatto del lavoro fatto. Per me era la prima volta che mi interfacciavo con un progetto di tali dimensioni e con questa metodologia di lavoro di gruppo, quindi, soprattutto in prima battuta, devo ammettere che non è stato semplice. A posteriori si sarebbe potuta curare meglio la prima parte di analisi e progettazione fatta di gruppo per evitare alcuni problemi sorti durante lo sviluppo del progetto, che comunque siamo riusciti a risolvere. Mi ritengo anche soddisfatto di quella che è la realizzazione finale dell'idea originale. Per quanto riguarda la mia parte, credo di aver svolto un buon lavoro in ottica object oriented. Anche in questo caso, forse cambierei qualche cosa di quanto fatto per migliorare alcune implementazioni, ma comunque mi ritengo soddisfatto. Quello che però mi rende molto soddisfatto riguardo a questo progetto è la mia crescita come programmatore. La gestione di un progetto mi ha sicuramente aiutato a prendere maggiore confidenza con quelli che sono gli strumenti (Git su tutti) e ha allenato la mia capacità di trovare una soluzione ad un problema. Dopo questo progetto mi sento sicuramente cresciuto sotto tutti gli aspetti e nel prossimo futuro vorrei "mettere le mani in pasta" con altre idee di progetto.

#### **4.1.4 Ricci Nicholas**

Sinceramente non sono troppo soddisfatto del mio lavoro nonostante mi abbia fatto acquisire abbastanza consapevolezza. Essendo il nostro primo progetto di queste dimensioni e di gruppo mi ha fatto capire l'importanza di determinate fasi del percorso di sviluppo di un progetto a cui non ho dato l'importanza necessaria. Non sono troppo soddisfatto perché considero di essere partito impondendomi troppe pretese facendomi rallentare nella fase iniziale arrivando alla fine in modo sbrigativo scendendo a compromessi implementativi che non mi soddisfano. Anche se non mi pento delle mie troppe pretese che hanno causato problemi solo per inconvenienti tempistici e di

approccio da parte mia al progetto. Infatti ho intenzione di finire la mia idea di implementazione iniziale che ho lasciato indietro. Una nota positiva che posso trarre da questo progetto è, grazie a questa nuova consapevolezza, la voglia di iniziare altri progetti e mettermi in gioco per affinare le mie modalità di lavoro e progettazione cercando di trovare la più adatta e redditizia in termini di produttività in base alle mie caratteristiche.

#### 4.1.5 Yan Elisa

Complessivamente, mi ritengo piuttosto soddisfatta del lavoro svolto. Questo progetto rappresenta la mia prima esperienza di lavoro di gruppo e, nonostante le difficoltà incontrate, sono riuscita a superarle con successo. Grazie a questo progetto, ho ampliato significativamente le mie conoscenze di programmazione. In particolare, ho acquisito competenze nell'uso di Git e nella programmazione dell'interfaccia utente utilizzando JavaFX. Ho anche migliorato le mie abilità riguardo alla programmazione orientata agli oggetti. Queste competenze saranno sicuramente molto utili per il mio futuro, anche se inizialmente mi sembravano argomenti difficili. Inoltre, sono consapevole che alcune implementazioni potrebbero non essere ottimali, ma ritengo di aver applicato tutte le conoscenze che possiedo al momento e di aver fatto del mio meglio. Oltre agli aspetti tecnici, da questo progetto ho imparato l'importanza della divisione dei compiti e del coordinamento del gruppo per raggiungere un obiettivo comune. Concludendo, questa esperienza è stata estremamente formativa e mi ha permesso di crescere sia dal punto di vista tecnico che collaborativo.

# Appendice A

## Guida utente



Figura A.1: Immagine della schermata di inizio gioco

Quando si apre il gioco ci si trova davanti la schermata di inizio gioco Figura A.1. In questa schermata mediante il pulsante "Aggiungi giocatore" si può aggiungere un giocatore, scegliendo username e personaggio. I giocatori devono essere minimo 2 e massimo 4. Viene poi chiesto di scegliere la difficoltà. Una volta finita l'inizializzazione mediante il pulsante "Start" viene fatto partire il gioco.



Figura A.2: Schermata con il tabellone di gioco

La schermata principale dello svolgimento del gioco è quella in figura Figura A.2. Ai lati vengono mostrate le statistiche dei giocatori. Al centro si ha il tabellone di gioco. Ogni casella del tabellone di gioco ha un effetto diverso e questo è indicato dal colore di questa. In basso a sinistra si ha la lista degli oggetti del giocatore corrente. Nel momento in cui il giocatore ha uno di questi oggetti, lo può utilizzare all'inizio del proprio turno.

Abbiamo poi dei tasti direzione, i quali devono essere utilizzati dall'utente quando è necessario decidere in quale direzione andare quando si giunge in un qualche bivio. Vi sono poi i tasti mediante i quali si controlla l'andamento del gioco. Mediante il tasto "Tira i dadi", il giocatore tirerà i dadi e visualizzerà il risultato. Con il tasto "Muovi", la pedina del giocatore viene mossa di un numero di caselle corrispondente al risultato del dado. Nel caso in cui ci sia da scegliere una direzione, il gioco si ferma e aspetta la decisione del giocatore mediante gli opportuni comandi. Abbiamo poi il tasto "Avanti" che deve essere usato per attivare l'effetto dello slot in cui si finisce e per passare il turno al giocatore successivo.

Infine, in basso a destra si ha uno spazio in cui viene mostrato un testo contenente le indicazioni all'utente riguardo quello che deve fare in quel momento (es. se è a inizio turno, chiede di tirare i dadi). Viene anche detto in quale tipo di slot il giocatore è capitato una volta che si è mosso.

Una delle caselle nelle quali si può capitare è "Negozio". In questo caso si apre una schermata come quella in figura Figura A.3. Sulla sinistra vi è la lista degli oggetti con la relativa descrizione. Sulla destra per ogni oggetto

c'è un tasto che permette al giocatore di acquistare una copia dell'oggetto corrispondente ( nel caso in cui abbia abbastanza monete). Vi è poi il tasto "Esci" per tornare alla schermata principale del tabellone.

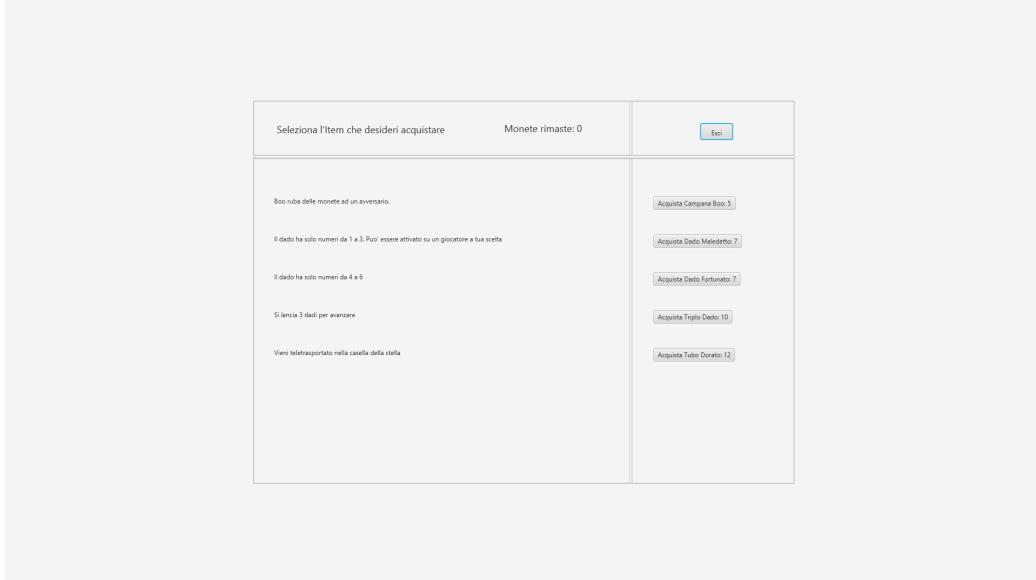


Figura A.3: Schermata dello shop

Nel caso in cui si termini in una casella “Minigioco” si apre un minigioco casuale. Ogni mini-gioco ha un proprio funzionamento che viene propriamente spiegato. Alla fine del minigioco, viene mostrato il numero di monete guadagnate dal vincitore: nel caso sia un gioco single-player, il giocatore guadagna un certo numero di monete se vince, mentre non guadagna niente se perde; nel caso sia un gioco multi-player, il giocatore vincente guadagna un certo numero di monete.

### Ariyo Daniel

**Perilous Path** : All'avvio del mini-gioco verrà visualizzata una schermata con delle istruzioni in alto e un pulsante ”START”. Premendo sul pulsante, appariranno le palle e le bombe come mostrato in figura A.4. Dopo 3 secondi, le bombe scompariranno, e a quel punto il giocatore dovrà cercare di unire le palline evitando le bombe, poiché cliccarle comporterà la fine del gioco con una sconfitta. Le caselle necessarie per congiungere le palle dovranno essere cliccate seguendo le regole fornite prima dell'inizio della partita; in caso contrario, il clic sarà ignorato.

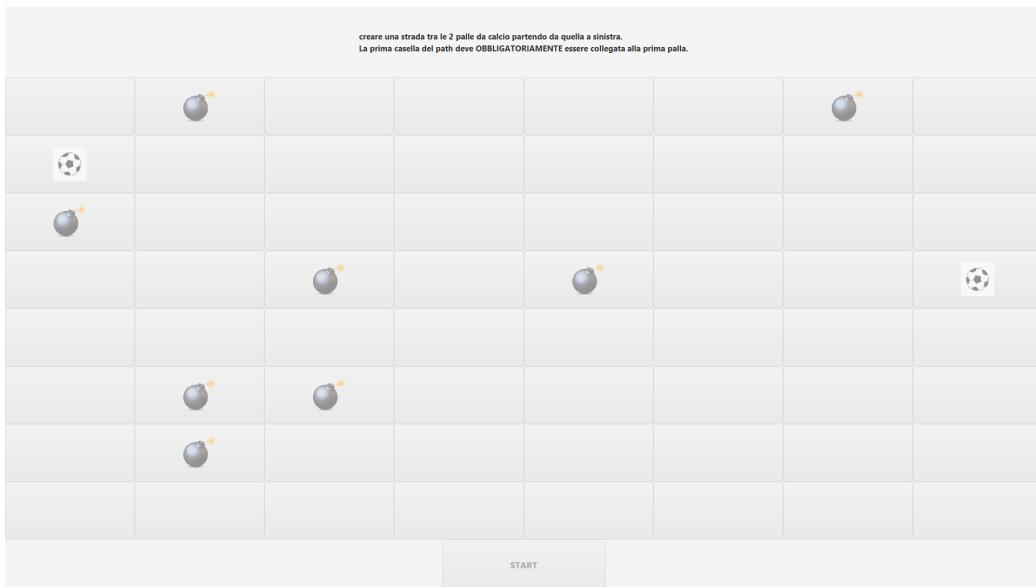


Figura A.4: Schermata del minigioco “Perilous Path”

**Memory Sweep** : All'avvio del mini-gioco, similmente al mini-gioco Perilous Path, comparirà una schermata con delle istruzioni e un pulsante "START". Cliccandolo su "START" la partita inizierà. I due giocatori si alterneranno ad ogni turno. Durante ogni turno verrà colorata una sequenza di bottoni, come mostrato in figura A.5; dopo 3 secondi questa sequenza verrà nascosta e il giocatore di turno dovrà ricrearla. Ad ogni turno, il numero di bottoni colorati aumenta. Il primo che sbaglia perde la partita.



Figura A.5: Schermata del minigioco “Memory Sweep”

### Ferretti Filippo

**Connect4:** All’inzio del gioco viene mostrata la griglia di gioco con sotto i bottoni relativi alle colonne in cui i giocatori potranno inserire le loro pedine cercando di posizionarne 4 in fila per vincere. Durante ogni turno viene visualizzato il giocatore di turno e c’è la possibilità di leggere un tutorial su come si gioca. Dal momento che i giocatori cliccano sui bottoni delle colonne verranno aggiunte le pedine relative al colore del giocatore, come mostrato in figura A.6. Al termine del minigioco verrà visualizzato il vincitore e le monete guadagnate, dandogli poi la possibilità di tornare al tabellone di gioco principale.

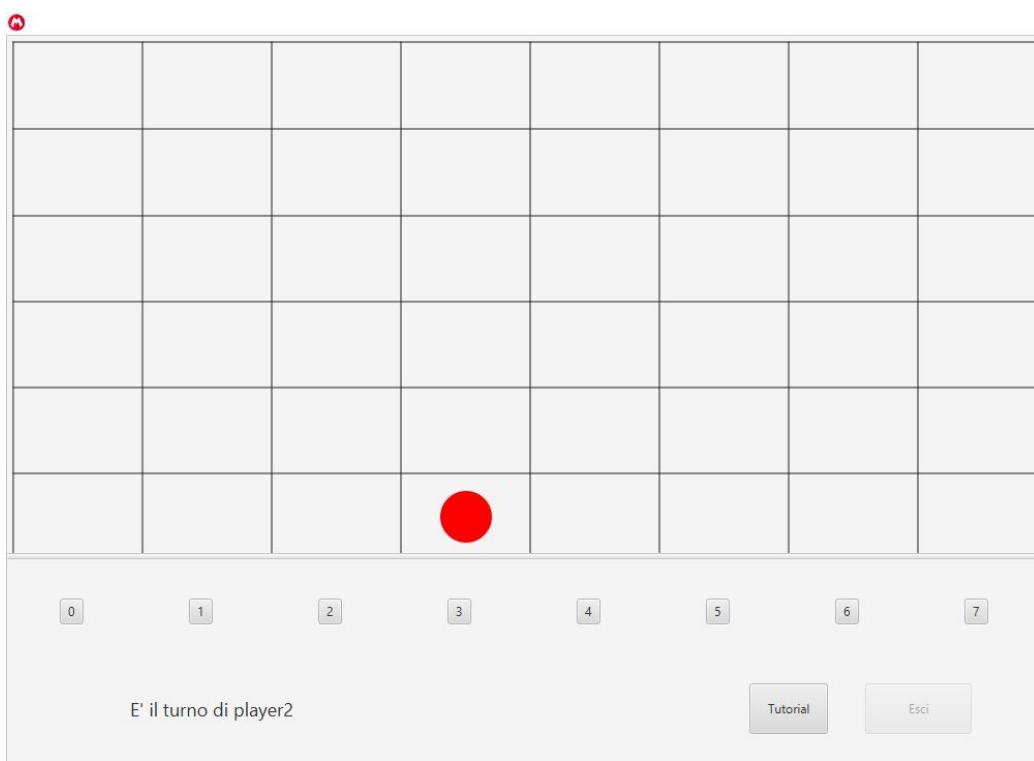


Figura A.6: Schermata del minigioco “Connect4”

### Fronzoni Gabriele

**MemoryCard:** All’inzio all’utente si apre una schermata con la spiegazione del gioco e un pulsante. Nel momento in cui si clicca il pulsante viene mostrata la finestra in figura A.7. Il giocatore visualizza e poi fa partire il gioco premendo il pulsante “Pronto”. Il giocatore poi ha le carte coperte e deve indovinare le coppie. Quando termina il gioco viene mostrato al giocatore il numero di monete guadagnate e gli viene data la possibilità di tornare al gioco principale.

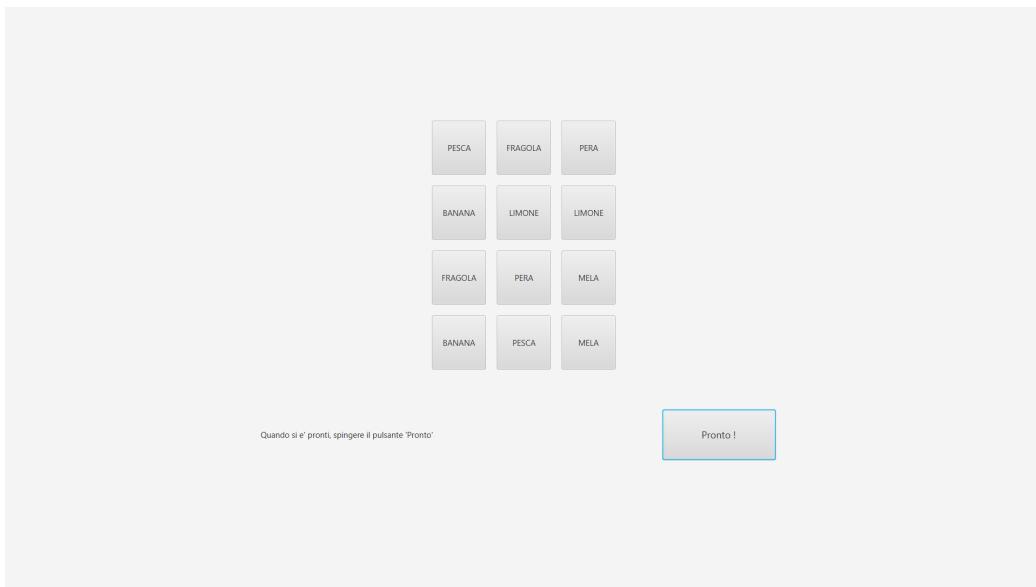


Figura A.7: Schermata del minigioco “MemoryCard”

### Ricci Nicholas

**Secret Code** : I due giocatori, uno per volta, dovranno creare il proprio “guess” premendo i pulsanti in fondo alla schermata per cercare di indovinare la soluzione corretta, unica per entrambi i giocatori. Ad ogni tentativo corrispondono degli indizi che aiuteranno i giocatori ad arrivare alla soluzione. Quindi attenzione perché il tentativo di un giocatore potrebbe aiutare l'avversario. Gli indizi non corrispondono ai tentativi in termini di ordinamento ma si riferiscono all'intero tentativo. La soluzione non presenta doppioni. Se nessuno dei due giocatori arriva alla soluzione, vince il giocatore che ha fatto più punti, essendo che ogni indizio guadagnato corrisponde ad un punteggio. Una volta terminato il gioco verrà visualizzata la soluzione e si potrà tornare al tabellone principale.

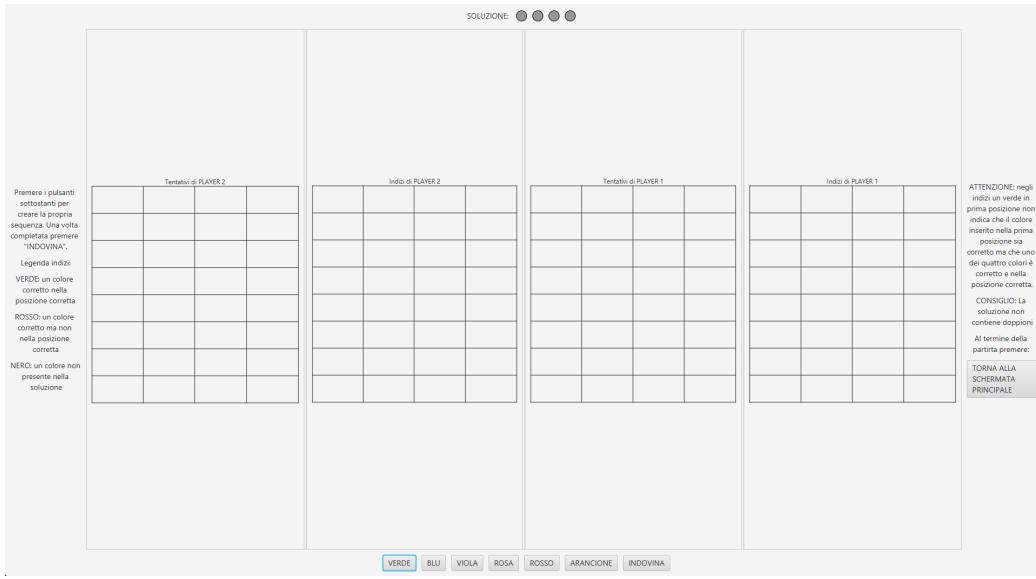


Figura A.8: Schermata del minigioco “Secret Code”

### Yan Elisa

**Nanogram:** All'avvio del mini-gioco “Nanogram”, verrà visualizzata una griglia di bottoni insieme ai suggerimenti su come riempirli correttamente, come viene mostrata nella Figura A.9. Di default, i bottoni sono impostati per la colorazione. Se l'utente commette un errore nel riempire la griglia, verrà visualizzato un messaggio di errore e perderà una vita.

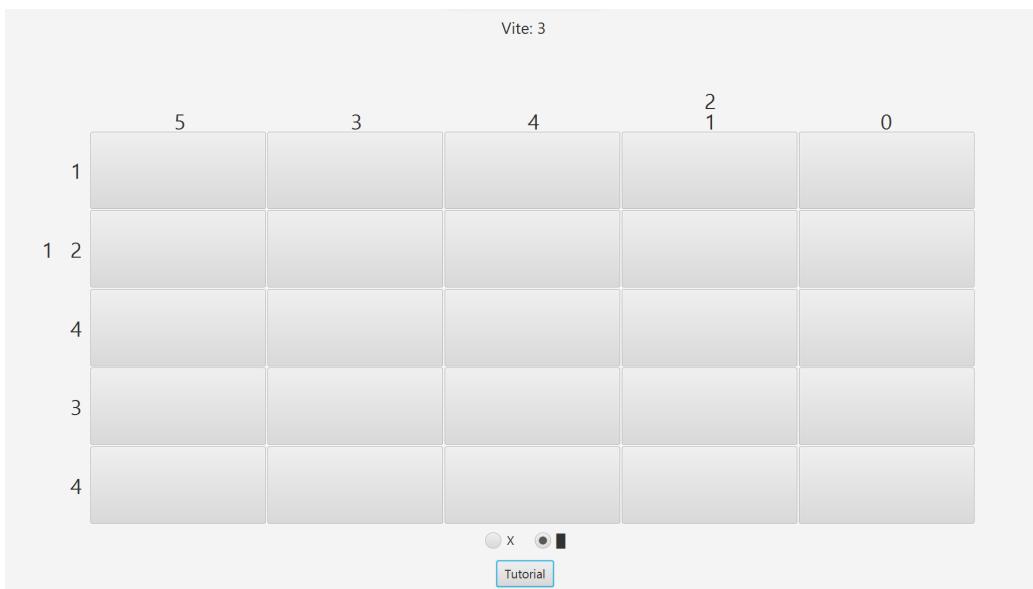


Figura A.9: Schermata del minigioco “Nanogram”

**Domino:** All’avvio del mini-gioco “Domino”, le tessere vengono distribuite ai giocatori, come viene illustrata nella Figura A.10. Per giocare, basta cliccare sul bordo della tessera desiderata e successivamente sul pulsante **Gioca Tessera**. La tessera verrà inserita nella board se è compatibile con una delle due estremità delle tessere già presenti.

Se il giocatore commette un errore e la tessera non è compatibile, verrà mostrato un messaggio di errore chiedendogli di scegliere un’altra tessera. Se il giocatore non ha tessere compatibili, il pulsante **Pesca Tessera** verrà attivato, permettendogli di pescare una nuova tessera dal mazzo.

Il gioco continua in questo modo fino a quando un giocatore finisce tutte le sue tessere o entrambi i giocatori non hanno più mosse disponibili.

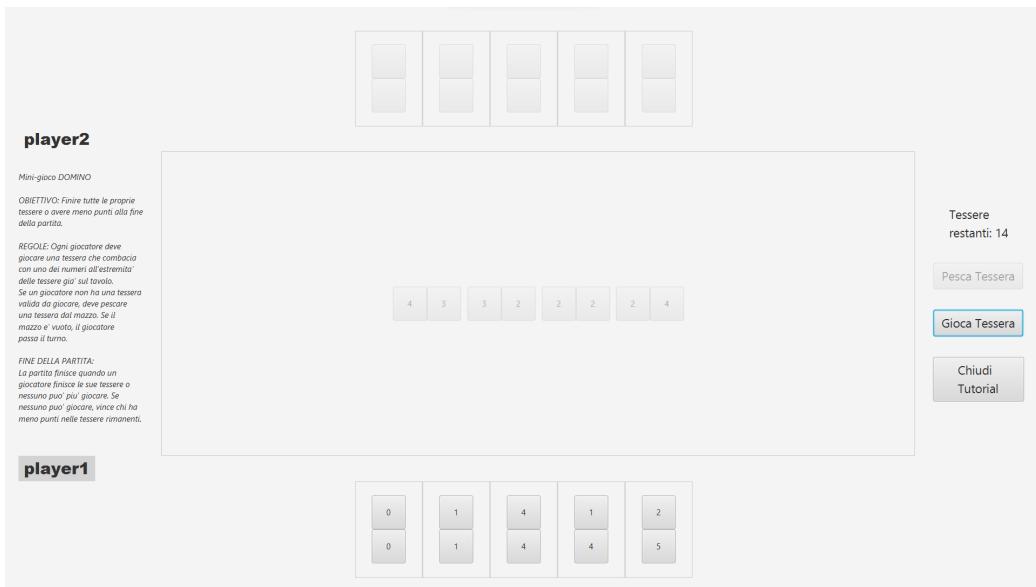


Figura A.10: Schermata del minigioco “Domino”

Alla fine viene mostrata una leaderboard con i risultati della partita. Vince il giocatore che ha guadagnato più monete. In caso di pareggio sul numero delle monete, si guarda a quante monete si ha.



Figura A.11: Schermata della leaderboard

# Appendice B

## Esercitazioni di laboratorio

### B.0.1 elisa.yan@studio.unibo.it

- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210306>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211468>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212522>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=151542#p213925>

### B.0.2 gabriele.fronzoni@studio.unibo.it

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209316>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210103>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211570>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212764>