

UniBoParty

Relazione per il progetto di
Programmazione ad Oggetti

Caterina Torelli
Nicola Mazzotti
Lorenzo Marchegiani
Alessandro Chierici

Indice

1 Analisi	4
1.1 Descrizione e requisiti	4
1.1.1 Descrizione	4
1.1.2 Requisiti Funzionali	4
1.1.3 Requisiti non funzionali	6
1.2 Modello del Dominio	7
2 Design	9
2.1 Architettura	9
2.2 Design dettagliato	10
2.2.1 Caterina Torelli	10
2.2.2 Alessandro Chierici	13
2.2.3 Lorenzo Marchegiani	16
2.2.4 Nicola Mazzotti	20
3 Sviluppo	24
3.1 Testing automatizzato	24
3.1.1 Caterina Torelli	24
3.1.2 Lorenzo Marchegiani	24
3.1.3 Alessandro Chierici	24
3.1.4 Nicola Mazzotti	25
3.2 Note di sviluppo	25
3.2.1 Caterina Torelli	25
3.2.2 Alessandro Chierici	25
3.2.3 Lorenzo Marchegiani	26
3.2.4 Nicola Mazzotti	26
4 Commenti finali	27
A Guida utente	28
B Esercitazioni di laboratorio	33

1 Analisi

1.1 Descrizione e requisiti

1.1.1 Descrizione

Il software mira alla realizzazione di una versione modificata del gioco da tavolo **Mario Party**, denominata **UniBoParty**. Il gioco supporta un numero di giocatori compreso tra due e quattro. I giocatori si muovono all'interno di un tabellone tramite il lancio di dadi. Il tabellone è composto da caselle che possono innescare **mini-sfide** (minigiochi) che, a seconda dell'esito (vittoria o sconfitta), determinano l'avanzamento o la retrocessione del pedone sulla plancia.

1.1.2 Requisiti Funzionali

Configurazione della Partita

- Il sistema deve permettere l'inserimento del nickname per ciascun giocatore prima dell'inizio della partita.
- Il numero di giocatori deve essere limitato tra un minimo di 2 e un massimo di 4.

Gestione dei Turni

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

Meccaniche di Gioco

- Durante il proprio turno, il giocatore deve lanciare il dado (simulato tramite un pulsante) e il valore risultante definisce il numero di celle di avanzamento sul tabellone.
- A seconda della cella in cui il giocatore termina il movimento, si possono verificare i seguenti eventi:
 - **Stazionamento:** Nessun effetto sul pedone.
 - **Swap:** Il giocatore viene scambiato di posizione con uno degli sfidanti.
 - **-2:** Il giocatore viene spostato indietro di 2 caselle.
 - **Sfida Single Player:** Il giocatore affronta un minigame. In caso di **vittoria** alla sfida, il giocatore avanza di posizione, in caso di **sconfitta**, il giocatore retrocede.

Minigiochi (Sfide)

- **MazeGame** (Responsabile: Torelli)
 - Il gioco è un labirinto classico: il giocatore parte da una cella d'ingresso e deve raggiungere una cella finale.
 - La vittoria si ottiene raggiungendo l'uscita senza superare una soglia massima di mosse o un tempo limite stabilito.
 - Il gioco utilizza tre griglie di gioco predefinite, selezionate in modo casuale.
- **Tetris** (Responsabile: Torelli)
 - Il gioco è una variante di Tetris su una griglia fissa 8x8.
 - Il giocatore riceve un set di tre tasselli alla volta, che deve posizionare sulla griglia.
 - L'atto di posizionare ciascun tassello sulla griglia conferisce punti al giocatore.
 - Il completamento di una riga o di una colonna provoca l'immediato svuotamento della stessa.
 - Il gioco è vinto se il punteggio accumulato è maggiore di 100 ed è perso (Game Over) quando non c'è spazio disponibile per collocare alcuno dei tasselli correnti.

- **DinosaurRun** (Responsabile: Chierici)

- Il gioco è un clone del minigioco offline del browser Google Chrome, in cui il giocatore controlla un dinosauro che deve evitare gli ostacoli saltando.
- Il dinosauro corre automaticamente verso destra, mentre il giocatore può farlo saltare tramite input da tastiera (SPACE).
- Gli ostacoli appaiono a intervalli regolari e avanzano verso il dinosauro; la collisione comporta la fine della partita.
- Il gioco prosegue finché il giocatore non collide con un ostacolo oppure passano 30 secondi.
- La vittoria è determinata dal contatto o meno con un ostacolo.

- **TypeRacer** (Responsabile: Chierici)

- Il gioco è un clone generico dello stile "Typing Game", nei quali l'obiettivo è digitare correttamente una parola mostrata sullo schermo il più velocemente possibile.
- Il giocatore deve inserire la parola nel campo di testo prima che il tempo si esaurisca.
- Ogni parola digitata correttamente viene immediatamente sostituita da una nuova.
- Ogni parola digitata correttamente conferisce 1 punto.
- La partita termina quando il tempo raggiunge 0 oppure il giocatore digita correttamente 10 parole entro 20 secondi..
- La vittoria è determinata dall'ottenimento di 10 punti.

- **Memory** (Responsabile: Marchegiani)

- Il gioco è un classico Memory con carte coperte disposte su una griglia.
- Il giocatore deve scoprire coppie di carte con lo stesso simbolo cliccandole una alla volta.
- Ad ogni turno vengono girate due carte: se corrispondono rimangono scoperte, altrimenti vengono nuovamente coperte dopo un breve intervallo.
- Durante la visualizzazione di una coppia non corrispondente, i click vengono bloccati per evitare interferenze.
- La partita termina quando tutte le coppie sono state trovate. Si vince se si trovano tutte le coppie entro il limite di mosse.
- Il punteggio è determinato dal numero di mosse effettuate (meno mosse = punteggio migliore).

- **Whac-A-Mole** (Responsabile: Marchegiani)

- Il gioco è una caccia alla talpa in cui il giocatore deve colpire le talpe che compaiono casualmente in diverse buche.
- Le talpe appaiono per un tempo limitato in buche casuali e scompaiono automaticamente se non colpiti.
- Oltre alle talpe normali, possono apparire bombe che penalizzano il punteggio se colpite.
- Il giocatore guadagna punti colpendo le talpe e perde punti (con minimo a 0) colpendo le bombe.
- La probabilità di apparizione delle bombe è controllata da una costante configurabile.
- La partita termina quando il tempo si esaurisce. Si vince se si totalizzano 10 punti entro il limite di tempo.
- Il punteggio finale è determinato dalla differenza tra talpe colpite e penalità per bombe.

- **Sudoku** (Responsabile: Nicola Mazzotti)

- Il gioco è una trasposizione digitale del classico rompicapo logico basato su una griglia 9x9 parzialmente riempita.
- Il giocatore deve riempire le celle vuote inserendo numeri da 1 a 9, selezionandoli da un tastierino laterale e cliccando sulla cella desiderata.

- Ogni inserimento deve rispettare i vincoli del Sudoku: ogni numero deve essere unico per riga, colonna e riquadro 3x3 di appartenenza.
- Il sistema valida immediatamente ogni mossa: se il numero è corretto viene fissato nella cella, altrimenti viene conteggiato un errore.
- La partita termina quando la griglia è stata interamente completata oppure quando il giocatore commette il terzo errore.
- La vittoria è determinata dal completamento corretto del tabellone entro il limite massimo di 3 errori.

- **Impiccato** (Responsabile: Nicola Mazzotti)

- Il gioco è una riproduzione del classico gioco di parole "L'Impiccato", nel quale bisogna indovinare un termine segreto scelto casualmente dal sistema.
- Il giocatore può tentare di indovinare singole lettere tramite una tastiera virtuale oppure provare a scrivere l'intera parola in un unico tentativo.
- Ogni lettera errata incrementa il contatore degli errori e aggiorna progressivamente il disegno stilizzato dell'impiccato; le lettere corrette vengono invece rivelate nella posizione appropriata.
- La partita termina quando la parola viene interamente svelata o quando il giocatore raggiunge il limite massimo di 6 errori.
- La vittoria è determinata dall'aver indovinato la parola segreta prima del completamento della figura grafica dell'impiccato.

Fine Partita

- L'obiettivo primario del gioco è raggiungere per primi la cella finale del tabellone.

1.1.3 Requisiti non funzionali

- Il gioco funziona su Windows, Linux e macOS
- Le finestre di gioco sono ridimensionabili

1.2 Modello del Dominio

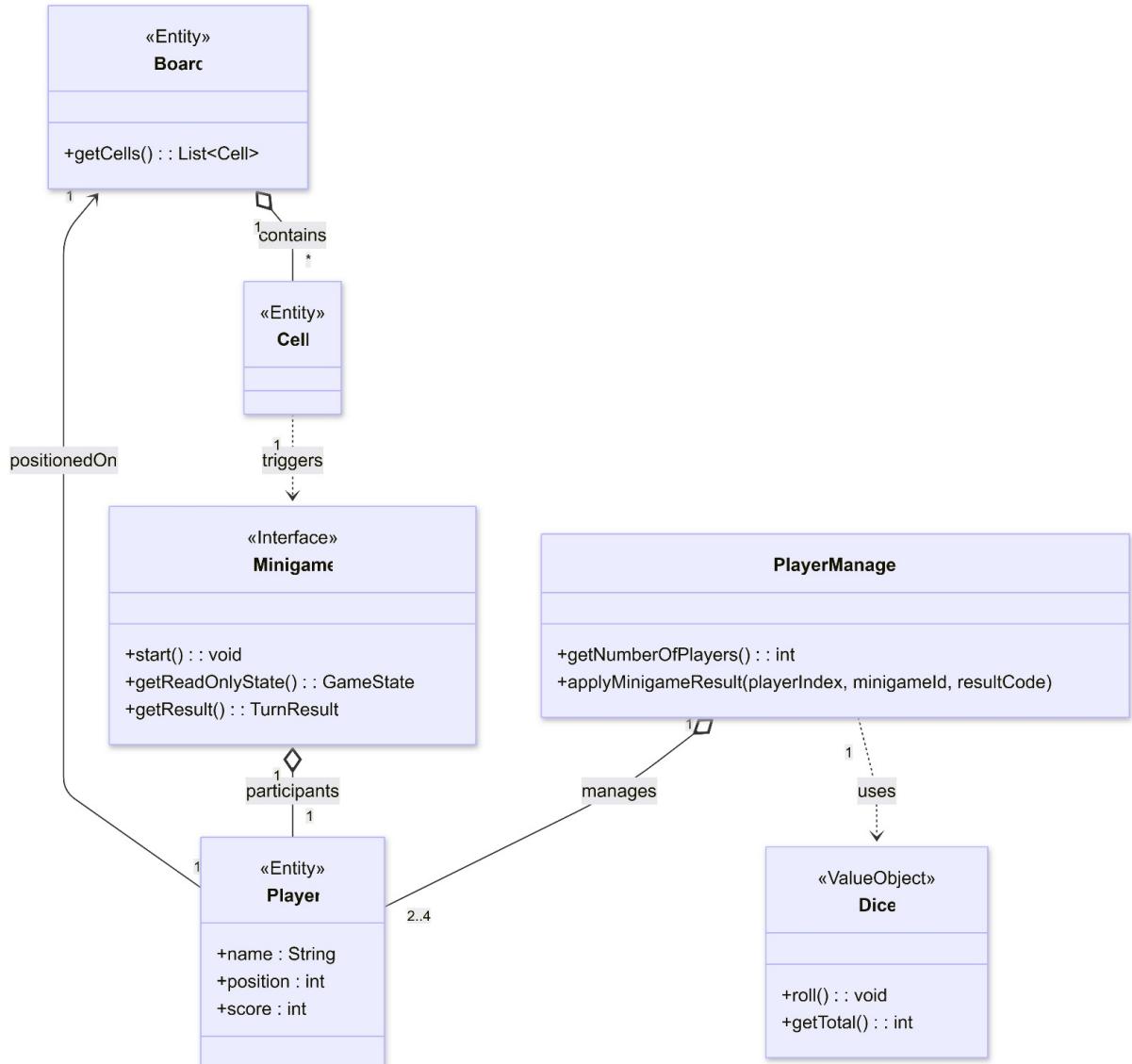


Figura 1: Schema UML del dominio applicativo di UniBoParty, con rappresentate le entità principali e i rapporti tra loro.

Il sistema dovrà gestire una partita di gioco da tavolo digitale alla quale partecipano da due a quattro giocatori. La partita si svolge su una tavola da gioco (Board) che rappresenta il percorso lungo il quale i giocatori avanzano nel corso dei turni.

La Board è composta da un insieme ordinato di celle (Cell), ciascuna delle quali rappresenta una posizione del percorso. Alcune celle possono essere associate all'attivazione di un minigioco (Minigame).

quando un giocatore vi giunge, influenzando l'andamento della partita.

Ogni Player rappresenta un partecipante al gioco ed è caratterizzato da un'identità, da una posizione corrente sulla board e da un punteggio che varia nel tempo. In ogni momento della partita, un giocatore occupa una posizione ben definita sulla tavola.

L'avanzamento dei giocatori è determinato dal lancio dei dadi (Dice), che producono un valore numerico utilizzato per lo spostamento sulla board. Il dice non possiede un'identità propria, ma rappresenta esclusivamente il risultato di un lancio. La gestione complessiva dei giocatori è affidata al PlayerManager, che coordina l'insieme dei player partecipanti e applica gli effetti derivanti dagli eventi di gioco, in particolare i risultati dei minigiochi.

I Minigame rappresentano giochi secondari che possono coinvolgere un giocatore e producono risultati che influenzano la partita principale. Nonostante la varietà delle regole possibili, dal punto di vista del dominio i minigiochi sono trattati come istanze di uno stesso concetto astratto. Il modello del dominio di UniBoParty è basato su 4 entità primarie, che sono gli elementi fondamentali del gioco:

- **Player:** rappresenta un giocatore all'interno della partita. Ogni Player è caratterizzato da un nome univoco e da una posizione sul tabellone, che viene aggiornata durante il gioco in base al lancio del dado, agli effetti delle celle speciali e ai risultati dei minigiochi. La gestione dei Player è centralizzata nel PlayerManager, che traccia lo stato di tutti i giocatori, coordina i turni in sequenza ciclica e applica le modifiche di posizione consultando il BoardController per verificare il tipo di cella su cui il giocatore atterra.
- **GameBoard:** è una sequenza lineare di celle con diverse tipologie. Le celle possono essere di tipo NORMAL (neutre), MINIGAME (avviano un minigame), BACK_2 (arretrano di 2 posizioni) o SWAP (scambiano posizione con un altro giocatore). Il layout del tabellone è predefinito e fisso, con posizionamento strategico delle celle speciali. Alcuni minigiochi vengono ripetuti in più posizioni per bilanciare l'esperienza di gioco.
- **Cell:** rappresentano gli spazi che costituiscono il tabellone e sono la fonte principale dell'interazione di gioco. Ogni cella può innescare un effetto diverso: alcune attivano i minigiochi, altre modificano la posizione del giocatore sulla mappa, e altre ancora sono semplicemente neutre. Ogni cella di tipo MINIGAME è associata a uno specifico Min gameId che identifica quale minigame avviare.
- **MiniGame:** è una breve sfida attivata quando un giocatore capita su una cella di tipo "Minigame". L'esito della sfida ha un impatto diretto sulla posizione del giocatore sul tabellone: la vittoria garantisce l'avanzamento, mentre la sconfitta comporta una retrocessione.

2 Design

2.1 Architettura

L'architettura del sistema segue il pattern Model–View–Controller (MVC), al fine di separare chiaramente la logica del gioco, la presentazione grafica e il coordinamento dei vari eventi. Il Model rappresenta la logica del gioco ed è costituito dalle entità individuate in fase di analisi, tra cui Board, Cell, Player, Minigame, Dice e PlayerManager. Il punto di ingresso principale del model è il PlayerManager, che coordina i giocatori e applica gli effetti derivanti dagli eventi di gioco. Il model è completamente indipendente da view e controller.

La View è responsabile esclusivamente della visualizzazione dello stato della partita e della raccolta degli input dell'utente. Essa comunica con il controller tramite eventi, senza contenere logica di dominio.

Il Controller riceve gli eventi dalla view, governa il flusso dell'applicazione e invoca le operazioni appropriate sul model. Il controller non mantiene lo stato del dominio, ma si limita a orchestrare le interazioni tra le componenti.

Grazie a questa architettura, la sostituzione completa della view non comporta modifiche né al controller né al model, mentre il dominio rimane riusabile anche in contesti applicativi differenti.

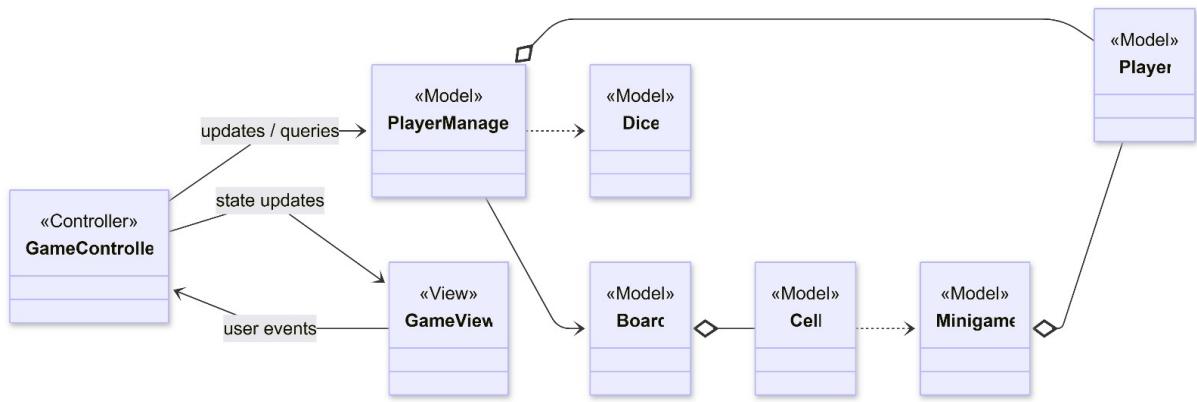


Figura 2: Schema UML architetturale di UniBoParty.

2.2 Design dettagliato

2.2.1 Caterina Torelli

MazeGame

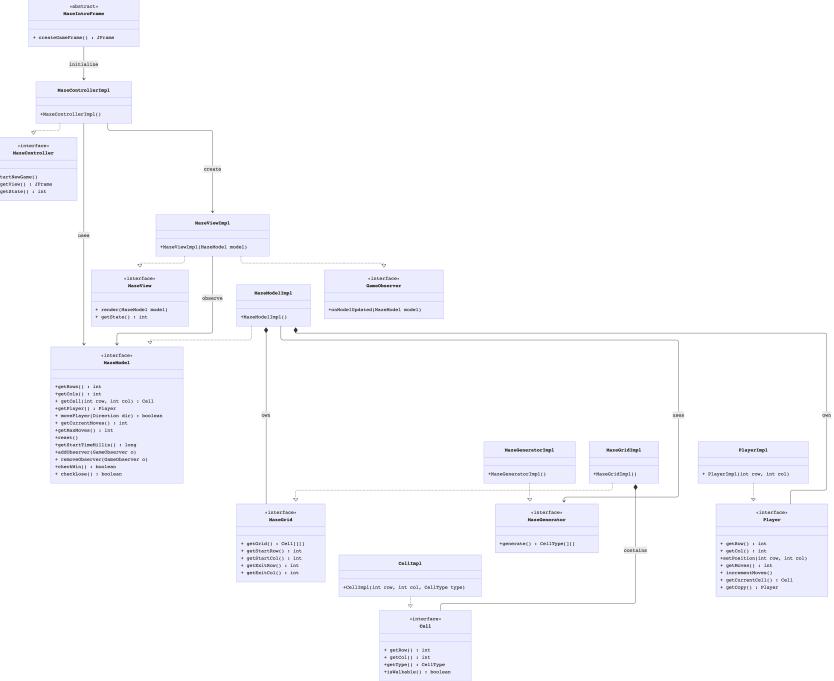


Figura 3: Rappresentazione UML del design del minigioco MazeGame.

Problema Gestione del movimento del giocatore (player) all'interno del labirinto e gestione dei tipi di mappe .

Soluzione La soluzione adottata per la gestione del movimento del giocatore all'interno del labirinto si basa su una separazione netta delle responsabilità grazie al pattern MVC. Per gestire il movimento del player è stato fondamentale l'uso di un modello a coordinate per tracciare la posizione. Il giocatore ha il solo compito di mantenere il conteggio delle mosse e la sua posizone all'interno del tabellone, mentre la logica di movimento è delegata al model che si occupa di ricevere il comando di direzione, calcolare le coordinate target e verificare la legalità della mossa, assicurandosi che la cella verso quale il player vuole muoversi sia all'interno dei limiti e che non sia un muro, consultando la griglia. Per quanto riguarda la gestione dei tipi di mappe la soluzione adottata è stata quella di scegliere 3 modelli preimpostati e poi sceglierne una casualmente tramite un random.

Tetris

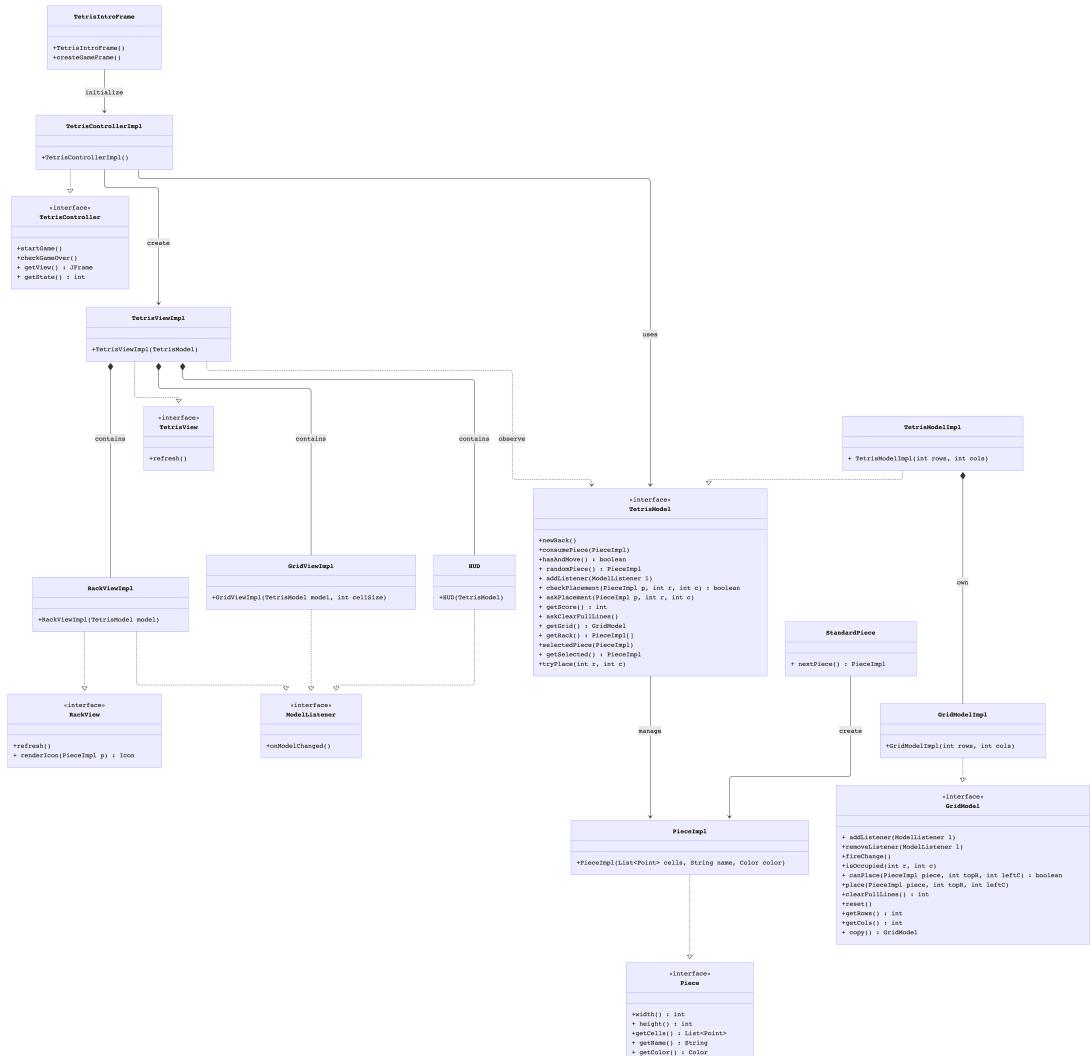


Figura 4: Rappresentazione UML del design del minigioco Tetris.

Problema Gestione fluida dell'inserimento dei tasselli e gestione della cancellazione dei tasselli quando si completa una riga verticale o orizzontale.

Soluzione La soluzione adottata per gestire il posizionamento dei tasselli e la conseguente rimozione delle righe/colonne complete nel minigioco Tetris è stata implementata concentrando la logica principale all'interno del Model. Il posizionamento fluido dei tasselli sul campo di gioco è delegato inizialmente al metodo `tryPlaceAt` in `TetrisModelImpl` e al metodo `place` in `GridModelImpl`. Il processo inizia nella View che calcola le coordinate target, input che viene poi passato al Model, il quale esegue tre passaggi sequenziali: Validazione tramite `canPlace` sulla griglia, che verifica i limiti e l'assenza di collisioni con tasselli preesistenti; Inserimento, dove il metodo `place` posiziona i blocchi del pezzo nella matrice booleana della griglia; e infine Pulizia (Clearing). immediatamente dopo l'inserimento, il Model invoca `grid.clearFullLines()` per eseguire il processo di cancellazione, che scorre l'intera griglia, e identifica le

righe e le collone complete, azzerandone le celle corrispondenti. Il processo si conclude con l'aggiornamento della View tramite la notifica (notifyAllListeners).

StartGame Menu

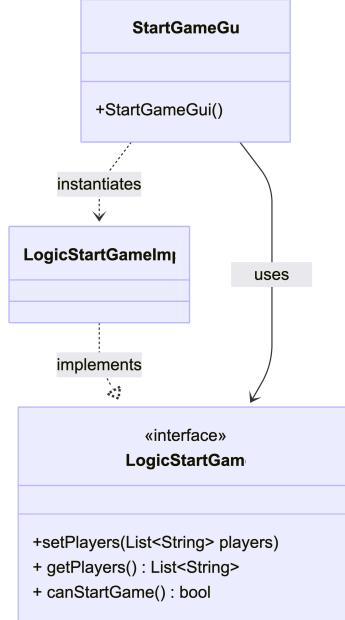


Figura 5: Rappresentazione UML della schermata di avvio del gioco.

Problema Salvataggio dei nomi dei giocatori e controllo che il numero di essi sia valido

Soluzione L'interfaccia utente, rappresentata dalla classe `StartGameGui`, si occupa di raccogliere i nomi digitati nei quattro campi di testo e di interfauciarsi con la logica. Quando l'utente preme "Start Game", la GUI estrae i nomi che non risultano vuoti o composti solo da spazi, li invia a `LogicStartGameImpl` e gli chiede di verificarne la validità. Il metodo `canStartGame` controlla che il numero di giocatori sia valido (compreso tra 2 e 4). Grazie a questa separazione netta, la GUI è mantenuta semplice e reattiva, offrendo un messaggio di avviso tramite `JOptionPane` solo nel caso in cui la logica segnali il fallimento della validazione, garantendo un'esperienza utente fluida e rendendo facile passare la lista utenti alle classi che lo richiedono.

2.2.2 Alessandro Chierici

Dinosaur Run

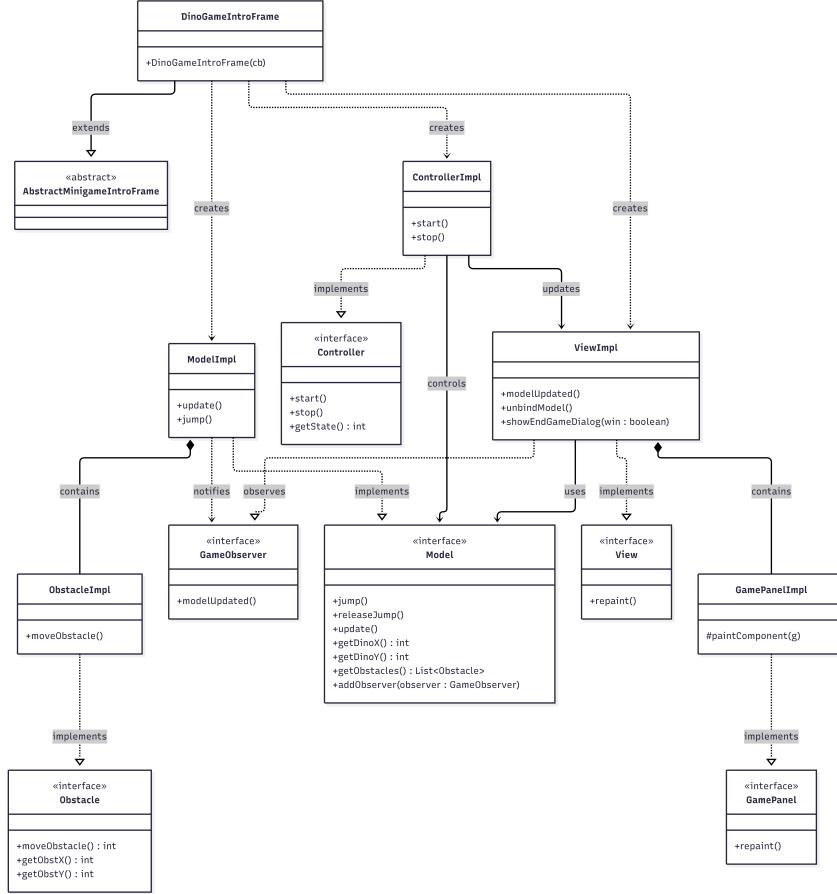


Figura 6: Rappresentazione UML del design del minigioco DinosaurRun.

Problema Gestione della fisica del salto con gravità variabile e rilevamento preciso delle collisioni tra dinosauro e ostacoli.

Soluzione La soluzione adottata per la fisica del salto si basa su un sistema di velocità verticale modificabile. Ad ogni update() viene applicata la gravità sommando GRAVITY a velY, che a sua volta viene sommata a dinoY per aggiornare la posizione. La pressione del pulsante "Spazio" imposta velY negativa, mentre la gravità applicata varia in base a isHoldingJump: se il tasto è tenuto premuto viene moltiplicata per JUMP_GRAVITY (più bassa), che quindi permette di rimanere in aria per più tempo; altrimenti viene applicata normale, permettendo salti di altezza variabile. Per la rilevazione delle collisioni, il metodo checkCollision() confronta le coordinate del rettangolo del dinosauro (DINO_X, dinoY, DINO_WIDTH, DINO_HEIGHT) con quelle di ogni ostacolo verificando sovrapposizione su entrambi gli assi. Gli ostacoli vengono rigenerati quando escono dallo schermo, calcolando la nuova posizione come nearestX + distanza minima + variazione casuale, e la velocità viene incrementata dividendo difficulty per DIFFICULTY_INCREMENT_INTERVAL.

TypeRacer

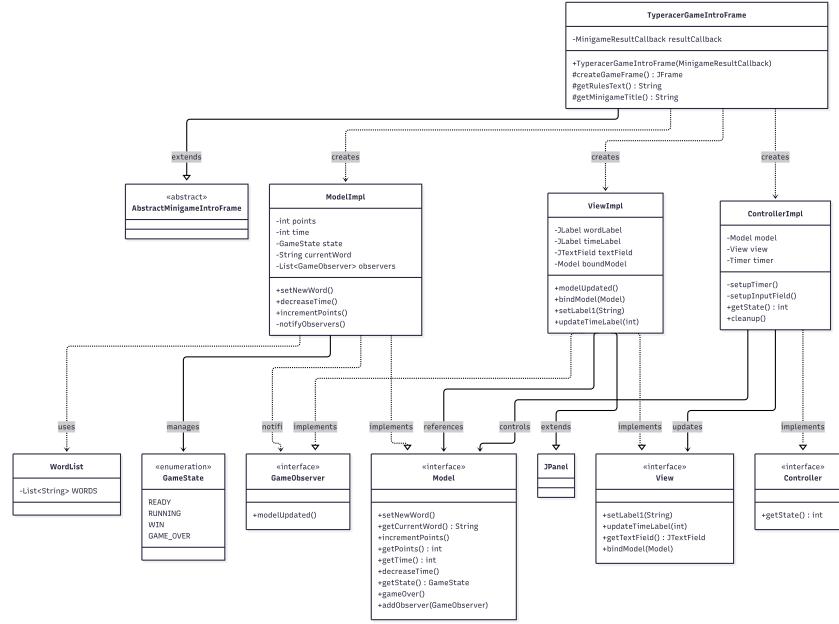


Figura 7: Rappresentazione UML del design del minigioco TypeRacer.

Problema Confronto in tempo reale tra input utente e parola target con gestione simultanea di countdown e condizioni di vittoria.

Soluzione La soluzione implementata per il confronto dell'input utilizza un ActionListener sul campo di testo che esegue equals() tra typed e currentWord. La selezione casuale avviene tramite random.nextInt(WORDS.size()) applicato alla lista WordList.WORDS. Il Timer esegue decreaseTime() ogni secondo, che decremente il contatore e verifica simultaneamente due condizioni: se time == 0 imposta state = GAME_OVER, mentre incrementPoints() verifica se points >= WIN_WORD_COUNT per impostare state = WIN. Il Controller monitora lo stato nel callback del Timer e invoca timer.stop() quando state != RUNNING, chiamando showFinalScore() per sconfitta o showVictoryMessage() per vittoria.

PlayerManager

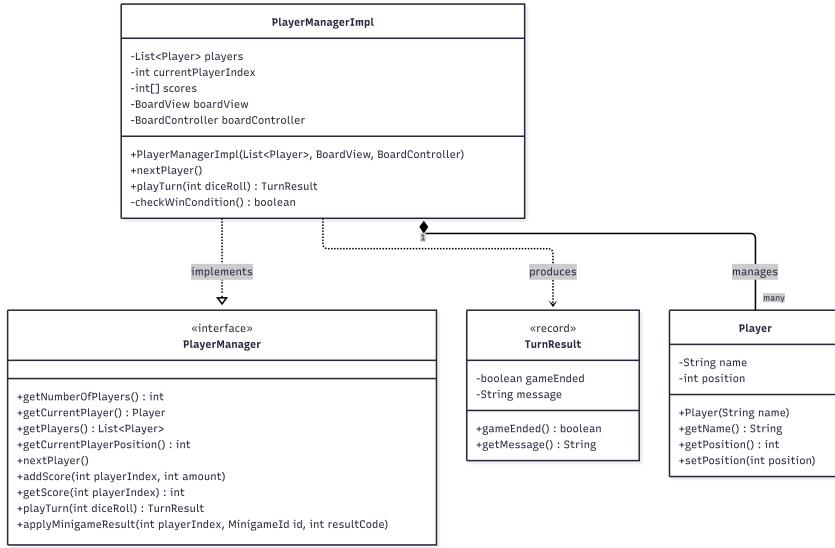


Figura 8: Rappresentazione UML del design del PlayerManager.

Problema Applicazione degli effetti delle celle speciali (BACK_2, SWAP, MINIGAME) e gestione dei limiti del tabellone.

Soluzione

Soluzione La soluzione adottata per la gestione dei turni e degli effetti delle celle speciali si basa sul metodo playTurn(int steps), che dirige l'intero flusso di movimento e applica gli effetti della cella su cui il giocatore atterra. Internamente, playTurn() calcola la nuova posizione sommando steps alla posizione corrente, verifica il tipo di cella tramite boardController.getCellTypeAt(newPos) e applica l'effetto corrispondente. Per le celle BACK_2, la posizione viene ricalcolata con Math.max(0, newPos - 2) controllando di non scendere sotto zero. Per le celle SWAP, viene selezionato casualmente un altro giocatore tramite un ciclo do-while che esegue otherIndex = (int)(Math.random() * numberOfPlayers) finché otherIndex non è diverso da playerIndex, poi le posizioni vengono scambiate. Per le celle MINIGAME, il metodo restituisce il MinigameId tramite boardController.getMinigameAt(newPos) nel TurnResult. I limiti del tabellone sono garantiti verificando newPos >= boardSize e aggiornando prima di ogni setPosition(). L'applicazione dei risultati minigiocchi converte il codice risultato e applica nuovamente playTurn() con il movimento calcolato, applicando gli stessi controlli sui limiti. Il metodo ritorna un TurnResult immutabile contenente la nuova posizione, il minigioco da avviare (se presente) e il flag gameEnded che eventualmente segnala la fine della partita.

GameplayController

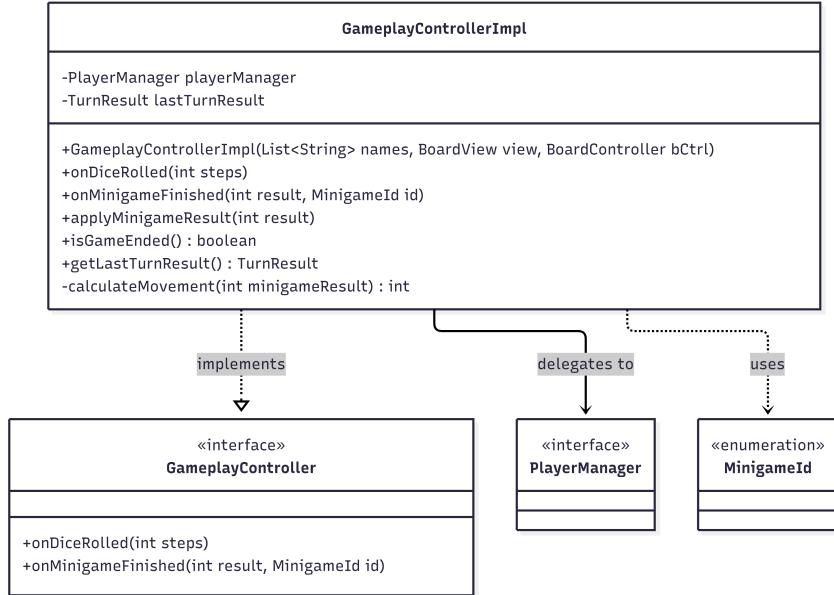


Figura 9: Rappresentazione UML del design del `GameplayController`.

Problema Memorizzazione dello stato del turno per permettere alla View di verificare minigiochi da avviare e condizione di fine partita.

Soluzione

Soluzione La soluzione implementata è un intermediario tra la View e il `PlayerManager`, memorizzando il `TurnResult` in `lastTurnResult` ad ogni invocazione di `playTurn()`. Questo permette alla View di accedere a `minigameToStart` e `gameEnded` tramite `getLastTurnResult()` senza dover mantenere riferimenti diretti al `PlayerManager`. Il metodo `onMinigameFinished` riutilizza la stessa logica convertendo il codice risultato con `calculateMovement()` e delegando a `playTurn()` invece di reimplementare altri controlli sui limiti. Il costruttore trasforma la lista di nomi in oggetti `Player` per gestire meglio posizioni e membri. La lista di giocatori viene quindi passata a `PlayerManagerImpl`, che internamente la rende immutabile tramite `List.copyOf()` per garantire thread-safety e prevenire modifiche accidentali. Il metodo `nextTurn()` resetta il valore null di `lastTurnResult` dopo aver chiamato `nextPlayer()`, invalidando lo stato del turno precedente e preparando il controller per il turno successivo.

2.2.3 Lorenzo Marchegiani

Memory

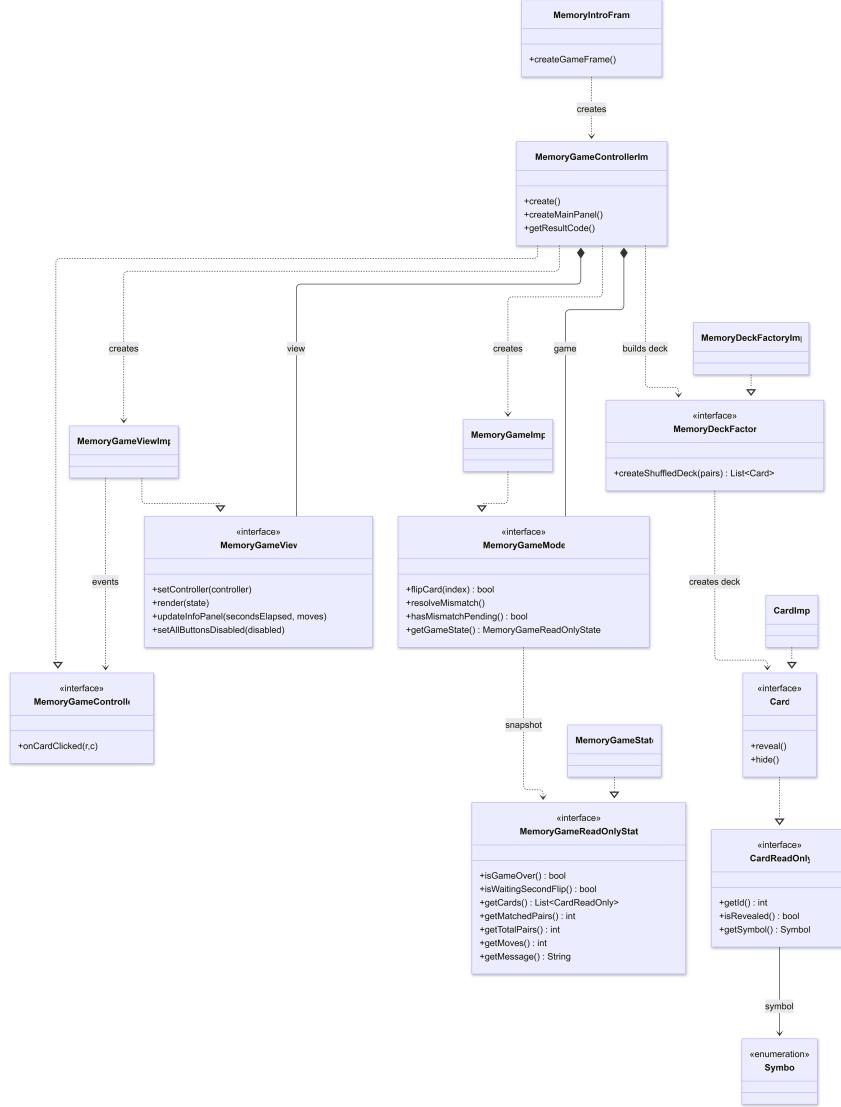


Figura 10: Rappresentazione UML del design del minigioco Memory.

Problema Gestione della logica di confronto tra carte e prevenzione di click multipli durante la visualizzazione di una coppia non corrispondente.

Soluzione La soluzione adottata per gestire il confronto tra carte si basa su due variabili firstSelectedCard e secondSelectedCard. Al primo click viene assegnata firstSelectedCard e la carta viene rivelata tramite reveal(). Al secondo click viene assegnata secondSelectedCard, incrementato il contatore moves e verificata la corrispondenza tramite checkForMatch() che confronta i simboli con ==. In caso di match viene incrementato matchedPairs e invocato endTurn() che azzerà entrambe le variabili. In caso di mismatch, viene impostato mismatchPending = true che blocca ulteriori click fino alla chiamata di resolveMismatch(). Il metodo resolveMismatch() nasconde le due carte tramite hide(), invoca endTurn() e resetta mismatchPending = false. Il blocco dei click durante il mismatch è implementato controllando mismatchPending all'inizio di flipCard() e restituendo false se true.

Whac-A-Mole

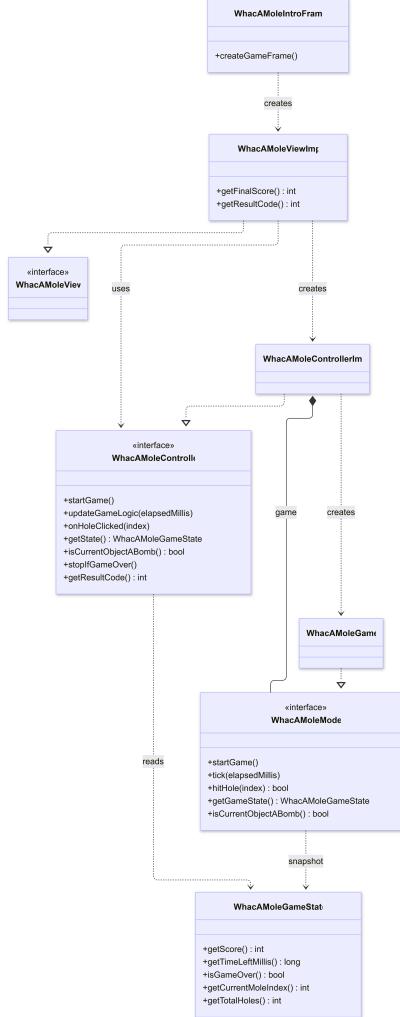


Figura 11: Rappresentazione UML del design del minigioco Whac-A-Mole.

Problema Gestione temporizzata della comparsa e scomparsa di talpe e bombe con penalità per le bombe colpite.

Soluzione La soluzione implementata per la gestione temporale utilizza il metodo tick(elapsedMillis) chiamato periodicamente dal Controller tramite Timer. Il metodo decrementa timeLeftMillis e verifica se il tempo è scaduto chiamando endGame(). Se è visibile un oggetto (currentMoleIndex != -1), viene incrementato moleVisibleMillis e quando supera MOLE_LIFETIME_MILLIS viene invocato removeMole() che azzera currentMoleIndex. Se nessun oggetto è visibile, viene incrementato timeSinceNoMole e quando supera MIN_TIME_BETWEEN_MOLES viene chiamato spawnNewMoleRandom() che genera un indice casuale con random.nextInt(TOTAL_HOLES) e decide se è bomba confrontando random.nextDouble() < BOMB_PROBABILITY. Il metodo hitHole(index) confronta index con currentMoleIndex e se corrispondono verifica currentIsBomb: se true decremente score di BOMB_PENALTY con controllo score < 0, altrimenti incrementa score di 1.

Board

Problema Gestione di un tabellone a configurazione fissa con celle speciali (minigiochi, arretramento e scambio) e supporto al coordinamento del flusso di gioco (lancio dado, turni, avvio minigiochi e fine partita).

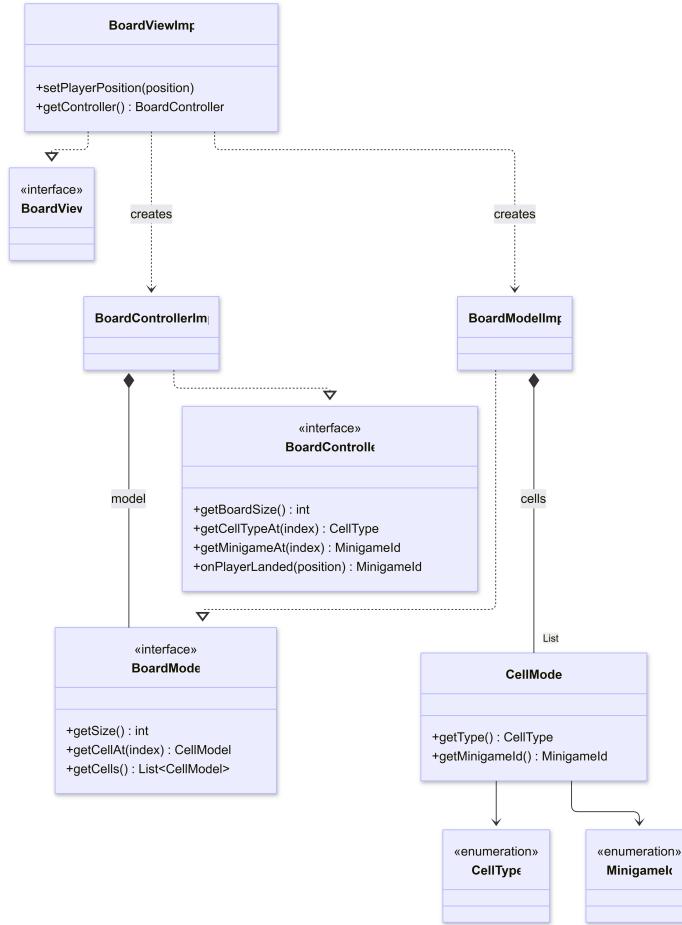


Figura 12: Rappresentazione UML del design del tabellone (componenti core: Model–Controller–View).

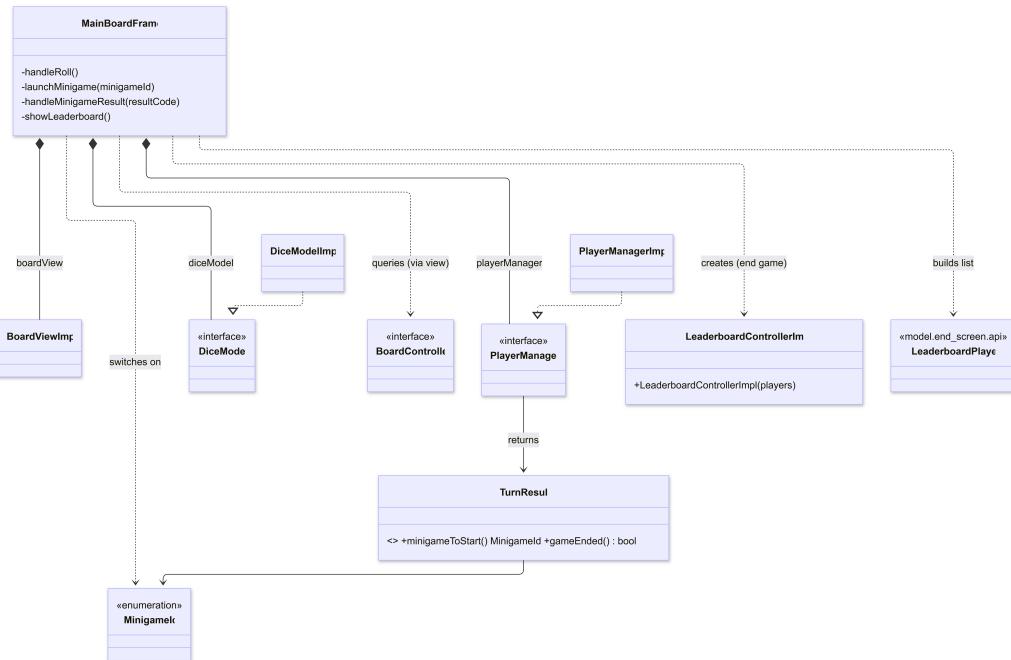


Figura 13: Rappresentazione UML del flusso di gioco: coordinamento del turno tramite **MainBoardFrame**, interazione con dado e **PlayerManager**, avvio minigiochi e schermata finale di classifica.

Soluzione La configurazione del tabellone è definita in modo statico in `BoardModelImpl` tramite una lista ordinata di `CellModel`. Ogni cella è costruita con `new CellModel(CellType, MinigameId)` specificando il tipo (`NORMAL`, `MINIGAME`, `BACK_2`, `SWAP`) e l'eventuale minigioco associato. Il `BoardControllerImpl` fornisce un accesso *read-only* al model: `getCellTypeAt(index)` e `getMinigameAt(index)` leggono i dati della cella, mentre `onPlayerLanded(position)` restituisce un `MinigameId` solo se la cella è di tipo `MINIGAME` (altrimenti `null`), senza avviare logica UI. La `BoardViewImpl` si occupa del rendering del tabellone in layout “a serpente” e dell’aggiornamento della pedina con `setPlayerPosition`. Il flusso partita è orchestrato da `MainBoardFrame`, che integra `DiceModel` e `PlayerManager`: dopo ogni lancio del dado applica il turno, avvia l’intro del minigioco indicato dal risultato e, a fine partita, crea `LeaderboardControllerImpl` per mostrare la classifica.

2.2.4 Nicola Mazzotti

Sudoku

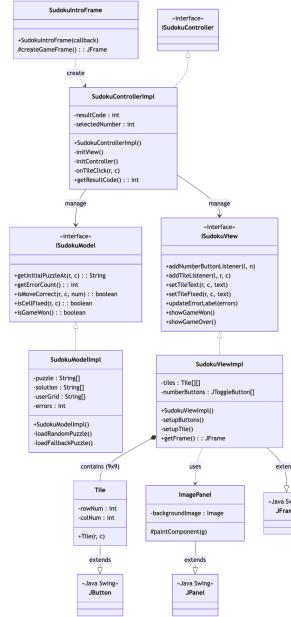


Figura 14: Rappresentazione UML del design del minigioco Sudoku.

Problema Gestione delle regole di validazione della griglia (rispetto dei vincoli di riga, colonna e riquadro) e rappresentazione grafica intuitiva della struttura 9x9.

Soluzione Adottando il pattern MVC, la logica è encapsulata in `SudokuModelImpl`. La robustezza del sistema è garantita da `loadRandomPuzzle()` che, in caso di fallimento nella lettura del file esterno, attiva automaticamente `loadFallbackPuzzle()` per iniettare una configurazione di default, prevenendo crash. Il metodo `isMoveCorrect()` confronta l’input utente direttamente con la soluzione in memoria, aggiornando la griglia o il contatore errori, mentre `isGameWon()` verifica la corrispondenza completa degli array. Lato interfaccia (`SudokuViewImpl`), la suddivisione visiva è ottenuta nel metodo `setupTiles()`: iterando sulla griglia, viene applicato condizionalmente `BorderFactory.createMatteBorder()` con spessori maggiorati sugli indici di confine dei quadranti, disegnando programmaticamente la struttura 3x3 senza l’uso di immagini di sfondo aggiuntive.

Impiccato



Figura 15: Rappresentazione UML del design del minigioco dell’Impiccato.

Problema Gestione della logica di mascheramento dinamico della parola segreta e sincronizzazione tra il conteggio degli errori e la rappresentazione grafica progressiva dell’impiccato.

Soluzione Logica del gioco separata in **HangmanModelImpl** e l’interfaccia in **HangmanViewImpl**. Il Model gestisce lo stato della partita: una parola viene estratta casualmente da una lista precaricata (con meccanismo di fallback simile al Sudoku). Il metodo chiave `guessLetter()` verifica la presenza del carattere: se corretto, aggiorna l’insieme delle lettere indovinate, altrimenti incrementa il contatore errori. La costruzione della stringa da visualizzare è delegata a `getMaskedWord()`, che rigenera dinamicamente la sequenza di lettere e underscore ad ogni turno. Lato View, la progressione visiva della sconfitta è gestita aggiornando l’icona principale: un array di immagini pre-caricate viene indicizzato tramite il numero di errori correnti (`getErrorCount()`), garantendo un feedback visivo immediato fino al completamento della figura dell’impiccato (Game Over).

End Screen

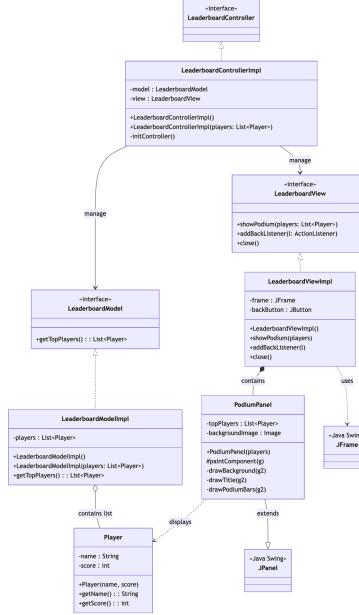


Figura 16: Rappresentazione UML del design della schermata finale.

Problema Aggregazione e normalizzazione punteggi dei vari giocatori per la generazione di una classifica.

Soluzione La logica è centralizzata nella classe LeaderboardModelImpl. L'elaborazione dei dati avviene direttamente nel costruttore, il quale riceve la lista dei partecipanti, ne crea una copia difensiva e la ordina immediatamente in modo decrescente tramite Comparator.comparingInt(Player::getScore).reversed(). La lista risultante viene memorizzata come collezione immutabile (List.copyOf) per prevenire modifiche accidentali. Il recupero dei dati per la View è demandato al metodo getTopPlayers(), che applica una logica di filtro per estrarre solamente i primi 3 classificati (o un numero inferiore se i giocatori sono meno di 3), fornendo i dati necessari al riempimento del podio grafico.

Dice

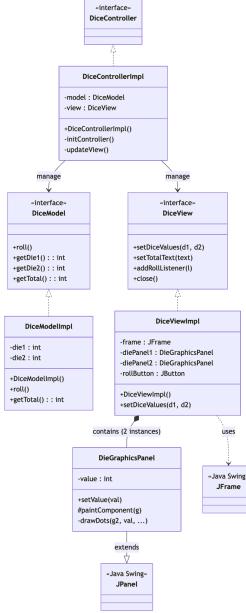


Figura 17: Rappresentazione UML del design del lancio dei dadi.

Problema Implementazione di un sistema di generazione casuale affidabile per il lancio dei dadi e realizzazione di una componente grafica realistica che permetta di visualizzare dinamicamente i risultati senza dipendere da un set rigido di immagini pre-renderizzate per ogni faccia.

Soluzione La logica è isolata nella classe DiceModelImpl. La generazione dei valori è gestita tramite un'istanza di `java.util.Random` all'interno del metodo `roll()`, che aggiorna lo stato dei due dadi garantendo matematicamente un range di valori valido (1-6) e calcolando istantaneamente il punteggio complessivo accessibile via `getTotal()`. Per la rappresentazione visiva, la classe DiceViewImpl delega il rendering al componente personalizzato DieGraphicsPanel. All'interno del metodo sovrascritto `paintComponent()`, viene utilizzata la libreria Java 2D per un disegno a livelli: inizialmente viene renderizzata un'immagine di sfondo (texture del tavolo) caricata dalle risorse; successivamente, il corpo del dado e i relativi pallini vengono disegnati vettorialmente sopra di essa. Questo approccio programmatico permette di aggiornare la faccia del dado (`setValue`) ridisegnando solo le forme geometriche necessarie, garantendo una scalabilità grafica superiore rispetto all'uso di sprite statici.

3 Sviluppo

3.1 Testing automatizzato

Tutti i test riportati in seguito sono stati creati usando JUnit 5.

3.1.1 Caterina Torelli

- **MazeGame:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “MazeGame”:
 - **CellTest** verifica il corretto funzionamento delle celle del Maze.
 - **MazeGridTest** verifica la corretta creazione e funzionamento della griglia del Maze.
 - **MazeModelTest** verifica il corretto funzionamento generale.
 - **PlayerTest** verifica la corretta creazione e il corretto funzionamento dei player del Maze.
- **Tetris:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “Tetris”:
 - **GridModelTest** verifica il corretto funzionamento della griglia di Tetris.
 - **PieceTest** verifica la corretta creazione dei pezzi.
 - **TetrisModelTest** verifica il corretto funzionamento generale.

3.1.2 Lorenzo Marchegiani

- **Memory:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “Memory”:
 - **MemoryGameImplTest** verifica lo stato iniziale del modello, la gestione dei flip, il riconoscimento di match/mismatch, il blocco dei click durante un mismatch, la corretta esecuzione di **resolveMismatch()** e la terminazione della partita al completamento di tutte le coppie.
- **Whac-A-Mole:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “Whac-A-Mole”:
 - **WhacAMoleGameTest** verifica l'inizializzazione del modello con **startGame()**, la terminazione a timeout, l'assenza di aggiornamenti dopo **gameOver**, la corretta gestione di spawn e hit (mole/bomba), l'assenza di effetti su hit errate e il comportamento di **isCurrentObjectABomb()** quando nessun oggetto è visibile.
- **Board:** di seguito elenco i test eseguiti per il corretto funzionamento del tabellone:
 - **BoardModelImplTest** verifica la configurazione statica del tabellone: dimensione totale, presenza delle celle, corretta assegnazione del **CellType** e dell'eventuale **MinigameId** nelle posizioni previste (celle **MINIGAME**, **BACK_2**, **SWAP** e **NORMAL**).

3.1.3 Alessandro Chierici

- **TypeRacer:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “TypeRacer”:
 - **ModelImplTest** verifica il corretto funzionamento generale del modello, come lo stato iniziale del gioco, la generazione di nuove parole valide, l'incremento dei punti, la gestione del conto alla rovescia con il passaggio allo stato **GAME_OVER** e l'aggiornamento dello stato del gioco.
- **DinoRun:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “DinoRun”:
 - **ModelImplTest** verifica il corretto funzionamento generale del modello, incluso il controllo della posizione iniziale del dinosauro, le dimensioni del dinosauro, la generazione degli ostacoli, il movimento degli ostacoli, il cambiamento di posizione durante il salto e lo stato iniziale del gioco.
- **Player Manager:** di seguito elenco i test eseguiti per il corretto funzionamento della gestione dei giocatori:

- `PlayerManagerImplTest` verifica il corretto funzionamento della gestione dei turni, degli spostamenti dei giocatori sulla board, l’interazione con i tipi di caselle (`NORMAL`, `MINIGAME`, `BACK_2`), il completamento della partita al raggiungimento della fine della board e l’aggiornamento dei punteggi in base ai risultati dei minigiochi.

3.1.4 Nicola Mazzotti

- **Sudoku:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “Sudoku”:
 - `SudokuModelTest` verifica lo stato iniziale del gioco (zero errori e griglia popolata), la coerenza dei dati nelle celle e la logica di validazione delle mosse, accertandosi che il sistema rilevi correttamente le mosse errate incrementando il contatore e accetti quelle valide senza penalità.
- **Impiccato:** di seguito elenco i test eseguiti per il corretto funzionamento del minigioco “Impiccato”:
 - `HangmanModelTest` verifica l’inizializzazione corretta della parola segreta e della maschera, la gestione dei tentativi su singola lettera (aggiornamento maschera vs incremento errori), la condizione di vittoria immediata indovinando l’intera parola e la condizione di sconfitta al raggiungimento del limite massimo di errori.
- **End Screen:** di seguito elenco i test eseguiti per il corretto funzionamento della schermata finale:
 - `LeaderboardModelTest` verifica che la lista dei giocatori non sia mai nulla o vuota, che venga rispettato il limite di visualizzazione (solo i primi 3 per il podio) e che l’algoritmo di ordinamento disponga correttamente i giocatori in ordine decrescente di punteggio.
- **Dadi:** di seguito elenco i test eseguiti per il corretto funzionamento del lancio dei dadi:
 - `DiceModelTest` verifica l’inizializzazione del modello e, tramite un ciclo di iterazioni multiple, garantisce che la generazione casuale dei valori rispetti sempre il range valido [1-6] e che la somma totale calcolata sia coerente con i singoli dadi.

3.2 Note di sviluppo

3.2.1 Caterina Torelli

- **Utilizzo di Lambda e Stream** Le Stream API e le espressioni Lambda sono state impiegate in diverse sezioni del codice per migliorarne la leggibilità. Un esempio significativo di questo approccio si trova nella classe `GridModelImpl`. <https://github.com/Al3ckss/UniBoParty/blob/master/src/main/java/it/unibo/uniboparty/model/minigames/tetris/impl/GridModelImpl.java#L120>
- **Utilizzo di Optional** Utilizzato in `TetrisModelImpl` <https://github.com/Al3ckss/UniBoParty/blob/master/src/main/java/it/unibo/uniboparty/model/minigames/tetris/impl/TetrisModelImpl.java#L180>

3.2.2 Alessandro Chierici

- **Utilizzo di List.copyOf per collezioni immutabili** Il metodo `List.copyOf()` è stato utilizzato in `PlayerManagerImpl` per creare una copia difensiva immutabile della lista di giocatori, garantendo che la collezione interna non possa essere modificata dall’esterno dopo la costruzione. <https://github.com/Al3ckss/UniBoParty/blob/master/src/main/java/it/unibo/uniboparty/model/player/impl/PlayerManagerImpl.java#L64>
- **Utilizzo di Java Record** I Java Record sono stati impiegati per definire `TurnResult`, eliminando il boilerplate code di getter, equals, hashCode e `toString`, creando un oggetto immutabile value-based in modo conciso. <https://github.com/Al3ckss/UniBoParty/blob/master/.../model/player/api/TurnResult.java>
- **Utilizzo di Stream API e Method Reference** Le Stream API con method reference sono state utilizzate in `GameplayControllerImpl` per trasformare una lista di nomi in oggetti Player tramite costruttore, sfruttando il paradigma funzionale. [https://github.com/Al3ckss/UniBoParty/blob/master/src/main/java/it/unibo/uniboparty/controller/player/impl/GameplayControllerImpl.java#\[LINEA\]](https://github.com/Al3ckss/UniBoParty/blob/master/src/main/java/it/unibo/uniboparty/controller/player/impl/GameplayControllerImpl.java#[LINEA])

3.2.3 Lorenzo Marchegiani

- **Uso di lambda expressions per event handling** Le lambda expressions sono state utilizzate per gestire eventi GUI in modo conciso, ad esempio per associare le azioni ai pulsanti della griglia e ai pulsanti di start. `MemoryGameViewImpl.java#L110–L114` `WhacAMoleViewImpl.java#L162–L165` `WhacAMoleViewImpl.java#L189–L192`
- **Uso di lambda expressions con javax.swing.Timer** Le lambda sono state impiegate anche come task periodici e delayed task nei Timer Swing (tick di gioco e gestione di ritardi), migliorando la leggibilità rispetto ad `ActionListener` espliciti. `MemoryGameControllerImpl.java#L108–L112` `MemoryGameControllerImpl.java#L203–L217` `WhacAMoleControllerImpl.java#L72–L75`
- **Uso di method reference** È stato utilizzato un method reference per passare in modo pulito una callback tipizzata nella gestione del risultato dei minigiochi (`this::handleMinigameResult`). `MainBoardFrame.java#L195`

3.2.4 Nicola Mazzotti

- **Utilizzo di Comparator e Method Reference** Per implementare la logica di ordinamento della classifica in modo conciso e leggibile, è stata utilizzata l'interfaccia `Comparator` combinata con i `Method Reference`. Questo ha permesso di ordinare la lista dei giocatori in base al punteggio decrescente in una singola riga di codice dichiarativo. https://github.com/A13ckss/UniBoParty/blob/2418d724dfbe39bb1b6e796fdb69a9db02239824/src/main/java/it/unibo/uniboparty/model/end_screen/impl/LeaderboardModelImpl.java
- **Utilizzo di Try-with-resources e I/O Streams** Per garantire la gestione sicura delle risorse durante il caricamento dei file di configurazione (es. `puzzle.txt` per il Sudoku), è stato adottato il costrutto `try-with-resources`. Questo assicura che gli `InputStream` e i `BufferedReader` vengano chiusi automaticamente anche in caso di eccezioni per prevenire memory leak. <https://github.com/A13ckss/UniBoParty/blob/2418d724dfbe39bb1b6e796fdb69a9db02239824/src/main/java/it/unibo/uniboparty/model/minigames/sudoku/impl/SudokuModelImpl.java>
- **Utilizzo di List.copyOf per collezioni immutabili** Similmente a quanto fatto nel `PlayerManager`, anche nell'implementazione del model della leaderboard è stato utilizzato il metodo statico `List.copyOf` per restituire alla View una versione non modificabile della lista del podio, preservando l'integrità del Model. https://github.com/A13ckss/UniBoParty/blob/2418d724dfbe39bb1b6e796fdb69a9db02239824/src/main/java/it/unibo/uniboparty/model/end_screen/impl/LeaderboardModelImpl.java

4 Commenti finali

- **Torelli** Sebbene riconosca margini di miglioramento nel mio lavoro, sono complessivamente soddisfatta del lavoro svolto. Ho imparato a sviluppare una gestione del codice più strutturata, che si è dimostrata efficace anche durante l'integrazione dei moduli: la flessibilità dell'architettura mi ha permesso di implementare nuove funzionalità e apportare modifiche correttive con estrema agilità. Lavorare in gruppo non è mai stato facile per me ma durante questo progetto ho imparato come mai prima quanto sia importante la coordinazione con i colleghi.
- **Chierici** Sebbene ci siano ancora punti di possibile miglioramento nel mio modo di lavorare, sono complessivamente soddisfatto del percorso svolto in questo progetto. Ho imparato a gestire il codice in maniera più strutturata e modulare, approcciandomene con una visione più professionale, cosa che ha reso più semplice l'integrazione dei vari componenti e l'implementazione di funzionalità. Il progetto nel suo insieme è stata un'ottima occasione per confrontarmi con sfide reali, imparare a organizzare il lavoro in team e capire quanto sia importante la pianificazione e la comunicazione tra i membri. Collaborare con gli altri non è sempre stato immediato, e ammetto che avrei potuto contribuire in maniera migliore, ma nel complesso questa esperienza mi ha permesso di crescere sia tecnicamente sia nella capacità di lavorare insieme ad altri, lasciandomi insegnamenti per il futuro.
- **Marchegiani** L'esperienza di lavoro di gruppo che questo progetto mi ha permesso di sperimentare è stata senza dubbio impegnativa. La necessità di produrre codice compatibile ed integrabile con il lavoro dei colleghi, come fosse un puzzle, è stata la sfida più difficile da vincere. A questa si sono aggiunte difficoltà più "pratiche", come la gestione dei merge e la risoluzione dei conflitti, il rispetto dei vincoli di qualità imposti dagli strumenti di analisi statica (Checkstyle e SpotBugs) e la cura della coerenza architettonica tra componenti sviluppati in parallelo. In diversi momenti ho dovuto rivedere scelte che "in locale" funzionavano, ma che nel contesto del progetto risultavano poco modulari o poco riusabili. Nel complesso, pur con le difficoltà incontrate, ritengo che il progetto abbia avuto un impatto formativo concreto: mi ha fatto toccare con mano cosa significhi "fare software" in un contesto collaborativo, dove la qualità non è solo far funzionare il programma, ma renderlo comprensibile, mantenibile e integrabile.
- **Mazzotti** Credo che uno dei valori che ho meglio appreso da questa esperienza di gruppo sia l'importanza di scrivere un codice non solo funzionale ma anche di qualità, immergendosi in quello che può essere considerata una "demo" di un ambiente lavorativo di gruppo. Ho appreso quanto può semplificare il processo la scrittura di un codice pulito e comprensibile, sia per la facilità di modifica che per la comprensione da parte del resto del gruppo. Riconosco di aver ancora molto da imparare, sia dal lato di programmazione e sia dalla parte di comunicazione con il team, per quest'ultimo ammetto che avrei potuto essere stato più disponibile rispetto a quanto ho effettivamente concesso.

A Guida utente

Menù

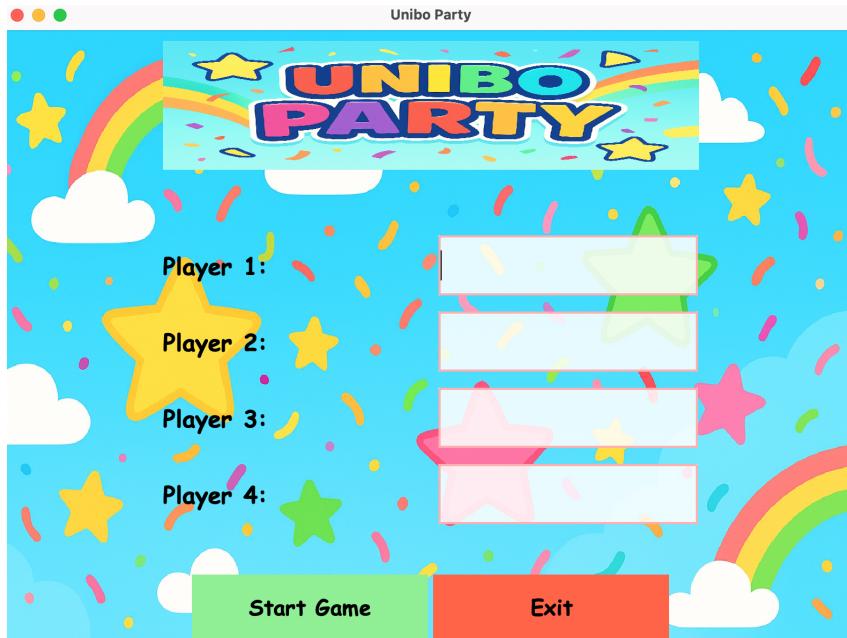


Figura 18: Immagine della schermata iniziale del gioco UniBo Party

All'avvio del gioco, compare la schermata iniziale (Figura 18). In questa schermata è possibile inserire i nomi dei giocatori (da 2 a 4). Il pulsante "Exit" chiude l'applicazione, mentre il pulsante "Start Game" avvia il gioco, a condizione che il numero dei giocatori sia valido.

Tabellone di gioco

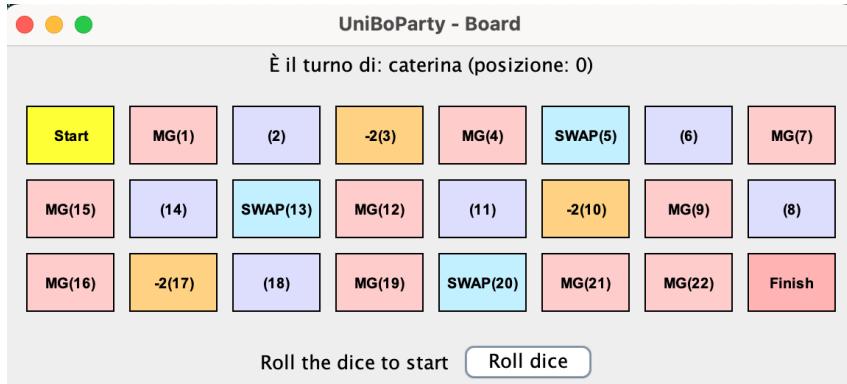


Figura 19: Immagine del tabellone del gioco UniBo Party

Una volta iniziato il gioco appare il tabellone principale (Figura 19). Il percorso di gioco si snoda con un andamento a serpente, partendo dalla casella Start in alto a sinistra fino alla casella Finish in basso a destra (ogni casella è numerata). Nella parte superiore dell'interfaccia sono indicati il nome del giocatore di turno e la sua posizione attuale prima del lancio. Cliccando il pulsante Roll dice in basso viene generato randomicamente un numero da 1 a 6 e vicino al pulsante compare immediatamente il punteggio ottenuto dal lancio dei dadi. La posizione dell'ultimo giocatore che ha lanciato il dado è indicato sulla plancia dal colore giallo. Le celle rosa con la scritta "MG" aprono un minigioco, quelle arancioni con la scritta "-2" mandano l'utente indietro di due caselle mentre quelle azzurre con la scritta "SWAP" scambiano la posizione del giocatore con quella di uno sfidante casuale.

Minigiochi



Figura 20: Immagine di lancio dei minigiochi del gioco UniBo Party

Nel caso in cui il giocatore capiti su una cella minigioco si apre la schermata di lancio dei giochi (Figura 20). Il giocatore può cliccare sul pulsante "how to play" per scoprire le regole del gioco oppure "play" per iniziare il gioco.

Caterina Torelli
Tetris

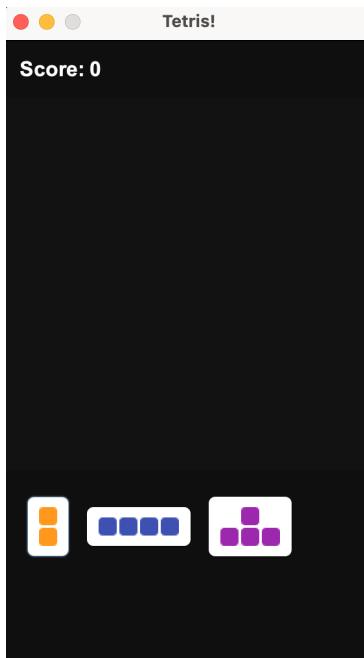


Figura 21: Schermata iniziale del minigioco Tetris

All'avvio del minigioco l'utente si trova davanti una griglia 8x8 e tre blocchi, di forme diverse, da posizionare (Figura 21). I pezzi vanno trascinati sulla griglia tramite il cursore. In alto il valore "score" aumenta in proporzione alla dimensione del blocco piazzato. Se una riga o una colonna viene riempita completamente allora essa viene cancellata per intero liberando spazio per i successivi blocchi. La partita si conclude con una sconfitta se il giocatore esaurisce le mosse possibili prima di aver raggiunto la soglia dei 100 punti.

Maze Game

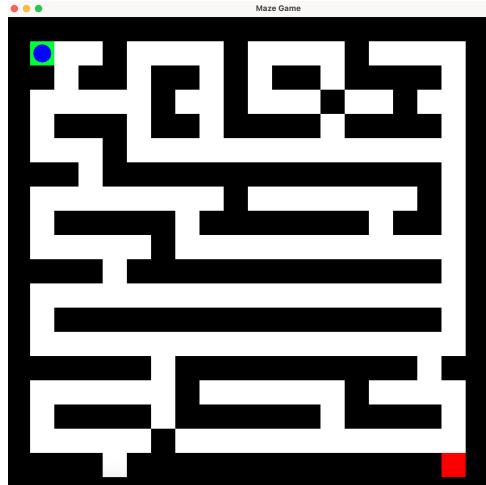


Figura 22: Schermata iniziale del minigioco Maze Game

All'avvio del minigioco l'utente si trova all' ingresso di un labirinto. Il giocatore, rappresentato dalla pallina blu nella figura 22, deve trovare il percorso per raggiungere l'uscita del labirinto (casella rossa) senza superare il limite di tempo o di mosse. La pallina blu puo muoversi a sinistra, a destra, in alto e in basso tramite le frecce direzionali della tastiera.

Lorenzo Marchegiani
Memory



Figura 23: Schermata iniziale del minigioco Memory

All'avvio del minigioco l'utente si trova davanti 16 carte (Figura 23). L'utente deve scoprire una carta (cliccandoci sopra tramite il cursore) e poi scoprirla un'altra. Se il simbolo sulle carte coincide, si è trovata una coppia, altrimenti, le carte vengono rigirate. L'obiettivo è trovare tutte le coppie prima di superare il limite di mosse.

Whac A Mole

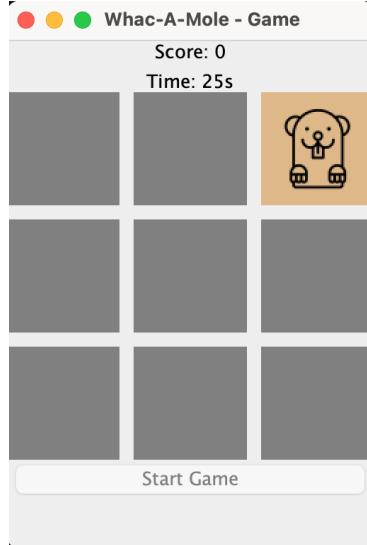


Figura 24: Schermata del gioco Whac A Mole

All'avvio del minigioco l'utente si trova davanti una griglia 3x3 (figura 24) sulla quale possono comparire talpe o bombe. L'utente deve cliccare (tramite il cursore) più talpe possibili evitando le bombe. Ogni talpa presa conferisce un punto al giocatore mentre ogni bomba cliccata ne toglie 2. L'obiettivo è totalizzare almeno 10 punti in 30 secondi.

Alessandro Chierici
Dinosaur Run



Figura 25: Schermata iniziale del minigioco Dinosaur Run

All'avvio del minigioco l'utente si trova davanti un rettangolo nero, il dinosauro (Figura 25). Da lì a breve inizieranno ad arrivare i vari ostacoli, di larghezze e altezze differenti. L'utente dovrà saltare premendo SPAZIO per evitare gli ostacoli. Tenendo premuto spazio si rimane in aria più a lungo. L'obiettivo è sopravvivere per 30 secondi senza toccare ostacoli.

TypeRacer

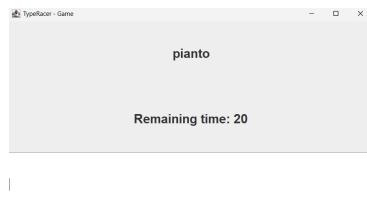


Figura 26: Schermata iniziale del minigioco TypeRacer

All'avvio del minigioco l'utente si trova davanti una parola presa casualmente da una lista, dal tempo rimanente e da un box di testo vuoto. Da lì il tempo inizierà a scorrere. L'utente dovrà scrivere il più velocemente possibile la parola mostrata nel box di testo, e premere invio. Se la parola digitata è corretta,

ovvero uguale a quella mostrata, si guadagna 1 punto, la box si svuota e la parola corrente si aggiorna. L'obiettivo è guadagnare 10 punti entro 30 secondi.

Nicola Mazzotti Sudoku

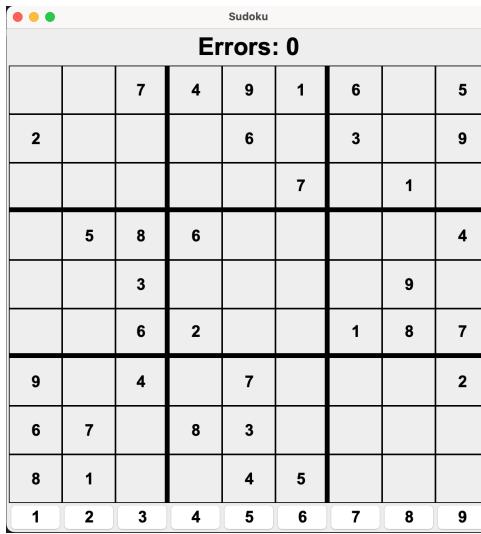


Figura 27: Schermata iniziale del minigioco Sudoku

All'avvio del minigioco, verrà presentata una griglia 9x9 suddivisa a sua volta in 9 riquadri 3x3, con alcuni numeri già inseriti. L'obiettivo è riempire le caselle vuote assicurandosi che ogni numero non si ripeta mai all'interno della stessa riga, colonna o riquadro. Per inserire i numeri, si deve cliccare sulle cifre mostrate nella parte inferiore dello schermo tramite il cursore. All'eventuale raggiungimento di 3 errori (conteggiati nella parte alta della schermata) la partita terminerà con una sconfitta.

Hangman

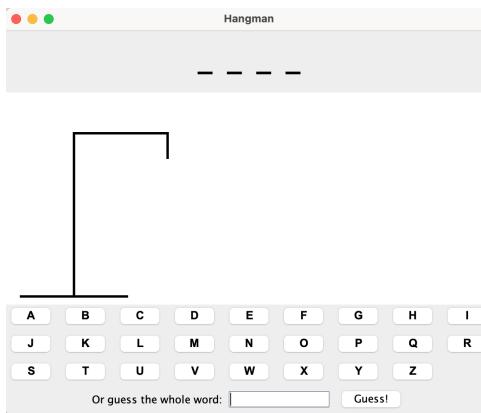


Figura 28: Schermata iniziale del minigioco Hangman

All'avvio del minigioco, l'interfaccia mostra una sequenza di trattini corrispondenti alle lettere di una parola nascosta. L'obiettivo del giocatore è indovinare il termine selezionando le singole lettere tramite i pulsanti presenti sullo schermo (da cliccare tramite il cursore); tuttavia, è necessario prestare attenzione poiché ogni errore comporta il progressivo completamento di una sagoma che, se ultimata, determina la sconfitta. La vittoria si ottiene rivelando tutte le lettere corrette oppure inserendo l'intera parola nell'apposita sezione e premendo il pulsante "Guess!" prima di esaurire i tentativi a disposizione.

Schermata finale quando uno dei giocatori raggiunge l'ultima cella si apre la schermata finale (29) che mostra la classifica dei giocatori. In basso è presente un pulsante "Go back to Menu" che riporta alla schermata iniziale (18).



Figura 29: Immagine della schermata finale

B Esercitazioni di laboratorio