Progettazione e sviluppo di

Percorso Evolutivo

Gioele Spanò, Silvia Farina, Sonia Culotta

12 Gennaio 2026

Indice

1. Introduzione

Percorso Evolutivo è un gioco single-player, basato su una **griglia dinamica** in continua evoluzione. Il giocatore deve raggiungere l'obiettivo tenendo d'occhio risorse, trappole e ostacoli.

Il gioco si svolge su una griglia 20x20 generata automaticamente a ogni partita. Il giocatore parte da una posizione iniziale, e deve trovare il miglior percorso per arrivare all'obiettivo, raccogliendo risorse, ed evitando trappole, aiutandosi con delle abilità speciali che avrà a sua disposizione all'inizio del gioco.

2. Analisi del problema

Prima della fase implementativa, è stata condotta un'analisi per identificare le criticità legate allo sviluppo dell'applicazione. Durante quest'analisi sono state identificati le seguenti problematiche logiche:

2.1. Complessità e accoppiamento

Lo sviluppo di un'applicazione grafica con logiche algoritmiche profonde pone una sfida: il rischio di creare codice "monolitico" o "spaghetti code". Senza una struttura ben definita, si possono riscontrare i seguenti problemi:

2.1.1. **Dipendenza:** Se la logica del gioco è mescolata al codice della GUI, ogni minima modifica estetica rischia di influenzare, e addirittura rompere il funzionamento del gioco

2.1.2. **Difficoltà di Debugging:** Diventa difficile identificare bug o errori nel codice se non è chiaro da dove provengono

2.1.3. **Gestione degli Stati:** Con l'aumentare delle funzionalità, gestire tutto il codice in un unico blocco risulterebbe inevitabilmente in residui grafici e conflitti logici

2.2. Generazione procedurale

Il problema principale di una griglia generata casualmente è l'imprevedibilità. Posizionare muri, trappole e risorse in modo randomico rischia di creare mappe impossibili da risolvere, dove il giocatore è fisicamente bloccato o privo delle risorse necessarie per avanzare. Una cattiva implementazione della generazione della griglia comporta i seguenti rischi:

2.2.1. **Inaccessibilità dell'obiettivo:** il giocatore potrebbe trovarsi impossibilitato ad avanzare a causa di muri o trappole che eliminano la possibilità di percorrere una strada fino all'obiettivo.

2.2.2. **Sbilanciamento delle risorse:** Anche se c'è un percorso che porta fino all'obiettivo, la mappa potrebbe non avere le risorse necessarie a raggiungerlo entro il limite previsto.

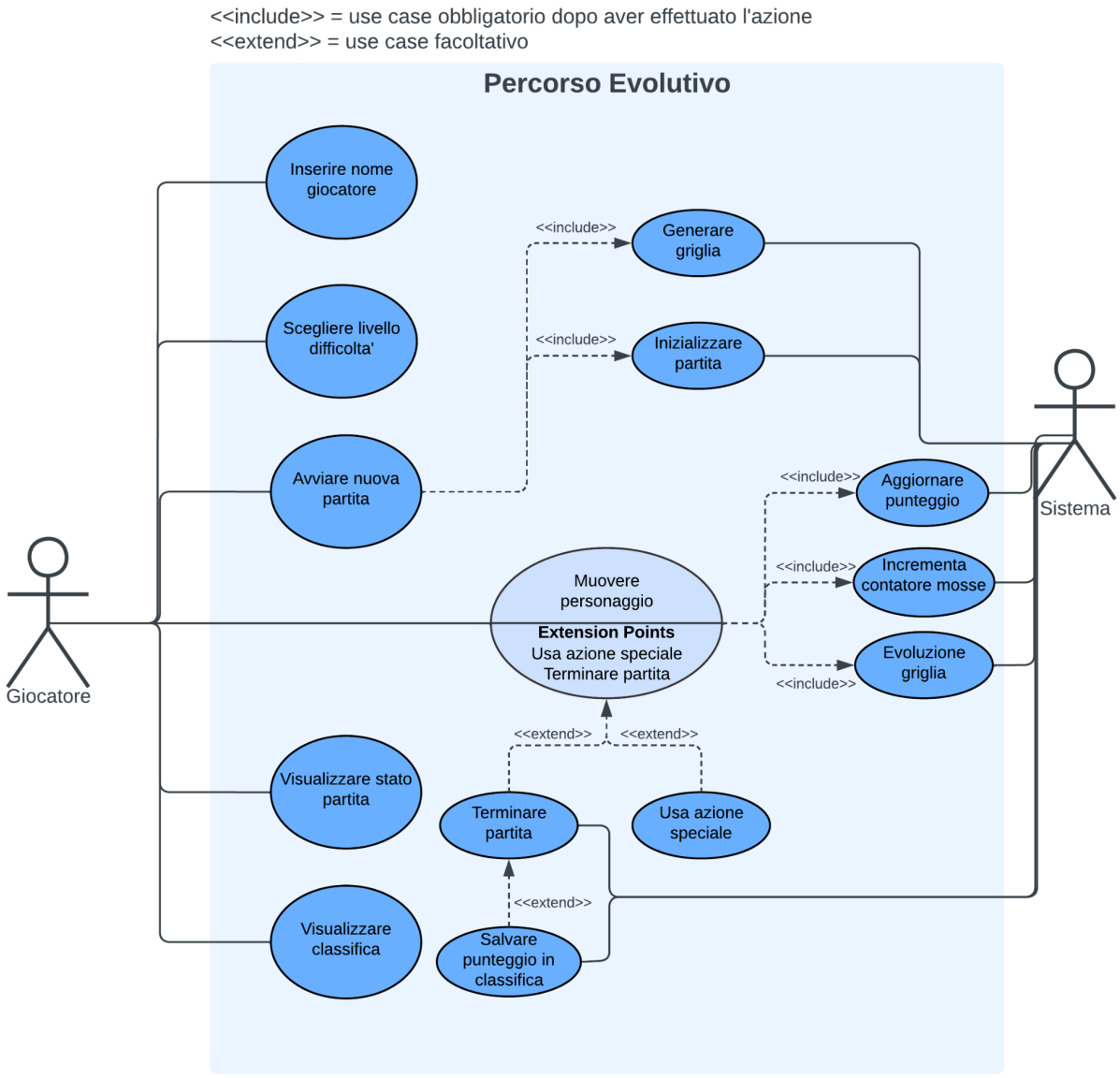
2.2.3. **Distanza troppo grande dall'obiettivo:** Il gioco prevede che il giocatore possa raggiungere l'obiettivo entro un limite di 30 mosse. L'obiettivo potrebbe trovarsi in una posizione che non è possibile raggiungere entro il limite massimo.

2.3. Gestione classifica

Un'ultima criticità analizzata è quella relativa al salvataggio dello score ottenuto dai giocatori. È importante che i dati siano salvati correttamente, utilizzando un algoritmo adatto, che non rallenti il sistema.

2.4. Diagramma Use-Case

Per definire chiaramente le interazioni tra l'utente e il Sistema, è stato redatto un Diagramma dei Casi D'Uso (Use-Case Diagram). Questo permette di rappresentare le funzionalità principali, come l'avvio della partita, o la generazione della griglia.



3. Progettazione della Soluzione

In questo capitolo vengono illustrate le strategie che verranno adottate per risolvere le criticità individuate nella fase di analisi

3.1. Implementazione del Pattern MVC

Per risolvere il problema della complessità e dell'accoppiamento (punto 2.1 dell'analisi), verrà adottata un'architettura **Model-View-Controller**.
Un'architettura che **divide le responsabilità in tre elementi interconnessi**:

3.1.1. Il Modello (Model)

Il modello gestirà i dati e le regole del gioco, senza alcun riferimento a breezypythongui o agli elementi della View. Ciò permetterà di testare la logica di gioco indipendentemente dall'interfaccia grafica.
Classi principali che saranno presenti nel Model: Game, Grid, Cell, Leaderboard, Player, Timer

3.1.2. La Vista (View)

L'interfaccia verrà gestita in modo gerarchico: verrà implementata una classe astratta ***BaseView*** che viene ereditata da ogni scena. La ***MainView*** svolge la funzione di orchestratore delle viste, gestendo la transizione tra le scene e la pulizia della griglia Tkinter per evitare artefatti grafici. Le classi che comporranno la View sono:
Classi principali che saranno presenti nella View: MainView, BaseView, StartScreen, GameView, LeaderboardView, GameInstructions

3.1.3. Il Controller

Il Controller agisce come intermediario. Riceve gli input dalla View (es. click sulle celle, click sui pulsanti direzionali) e interroga il Model. Successivamente, istruisce la View su cosa mostrare

L’architettura dettagliata delle classi e le loro relazioni è consultabile nel diagramma UML riportato in Appendice A

3.2. Generazione e validazione griglia

Per ovviare all’imprevedibilità della generazione della griglia (punto 2.2 dell’analisi) verranno implementati due algoritmi che si basano, il primo sul DFS, il secondo sul BFS. Gli algoritmi saranno modificati adeguatamente su misura delle criticità presenti nel progetto. Il processo di generazione della griglia sarà il seguente:

3.2.1. La griglia viene inizializzata con solo celle ‘**Muro (X)’**.

3.2.2. Il DFS rimpiazza le celle Muro pregenerate con celle **‘Cella Vuota (.)’** al fine di creare un labirinto senza spazi isolati (ogni cella vuota è raggiungibile da qualsiasi posizione).

3.2.3. Le celle Muro rimaste vengono scelte randomicamente e rimpiazzate fino al numero previsto dalle celle ‘**Risorsa (R)'** e ‘**Trappola (T)’.** Le celle Muro restanti o mancanti vengono aggiustate fino al numero di muri previsto. Infine vengono scelte due celle vuote e sostituite con una cella **Punto di Partenza (P)** e **Obiettivo (O)**.

3.2.4. Il Breadth First-Search (BFS) svolge la funzione di validazione: controlla se esiste una strada percorribile dalla cella Punto di Partenza alla cella Obiettivo, tenendo conto delle seguenti variabili:

- Movimento: il movimento del giocatore, qualsiasi sia la cella su cui viene effettuato, sottrae 1 al suo punteggio.
- Muri (X): celle non attraversabili.
- Risorse (R): celle che aggiungono 10 al punteggio del giocatore. Spariscono una volta attraversate.

- Trappole (T): celle che sottraggono 5 al punteggio del giocatore. Rimangono sulla stessa cella anche dopo essere state attraversate.

3.3. Salvataggio del punteggio in classifica

Per risolvere la criticità relativa all'ordinamento dei dati (punto 2,3 dell'analisi), verrà implementato un sistema di ordinamento basato su Insertion Sort.

Logica dell'algoritmo:
Invece di caricare tutti gli score dalla lista e ordinarla da zero ogni volta, il sistema sfrutta il fatto che la classifica esistente è già ordinata.
Quando il giocatore termina la partita:

- Il software scorre la lista dei punteggi salvati
- Il nuovo punteggio viene confrontato con quelli esistenti. Grazie alla logica dell'Insertion Sort, il record si posiziona esattamente dopo il punteggio superiore e prima di quello inferiore
- Una volta trovata la posizione corretta, la classifica viene aggiornata e riscritta sul file

4. Implementazione

L'implementazione del progetto è stata realizzata in Python, utilizzando la libreria *breezypythongui* per la gestione dell'interfaccia grafica.

Il software segue l'architettura Model-View-Controller (MVC), descritta nel capitolo di progettazione, garantendo la separazione della logica di business, interfaccia grafica e gestione degli input.

L’obiettivo principale dell’implementazione, seguita dalla progettazione, è quello di rendere il software modulare, leggibile, facilmente manutenibile, e soprattutto scalabile.

4.1. Implementazione del Model

Il Model è il cuore logico dell’applicazione e non contiene alcun riferimento agli elementi grafici.
Esso gestisce:

- La generazione ed evoluzione della griglia
- Il movimento del giocatore
- Le mosse speciali
- Il punteggio corrente
- Le regole di interazione con le diverse tipologie di celle
- Il salvataggio nella classifica

4.1.1. Panoramica delle classi

Per garantire una chiara separazione delle responsabilità, il Model è stato suddiviso in più classi, ciascuna con un ruolo ben definito.

Il model è costituito dalle seguenti classi:

Classe **Game**

La classe Game funge da coordinatore della logica di gioco.
Essa inizializza lo stato iniziale della partita e gestisce il flusso generale del gioco, determinando quando una partita inizia, termina o viene conclusa con successo o fallimento.
Metodi rilevanti: setup_game, move_player, use_special_action, can_reach

Classe **Grid**

La classe Grid rappresenta la griglia di gioco.
È responsabile della generazione procedurale della mappa e dell'aggiornamento delle celle a seguito delle azioni del giocatore.
Metodi rilevanti: generate_grid, generative_dfs, step,

Classe **Cell**

La classe Cell rappresenta la singola cella di Grid.
Ogni cella possiede un tipo che ne definisce il comportamento (risorsa, trappola, muro, cella vuota, obiettivo, punto di partenza) e fornisce le informazioni necessarie per sapere interagire con la cella

Classe **Player**

La classe Player rappresenta lo stato del giocatore.
Gestisce la posizione corrente, il punteggio, il numero di mosse effettuate e verifica se le abilità sono ancora disponibili. Non contiene la logica di gioco e non modifica direttamente la griglia.

Classe **Timer**

La classe Timer si occupa della gestione del tempo di gioco.
Utilizza il modulo time per calcolare i secondi dall'avvio della partita, incrementando il tempo trascorso durante il gioco e fermandolo quando è terminata.

Classe **Leaderboard**

La classe Leaderboard gestisce la classifica e l'organizzazione dei punteggi.
Salva i dati in un file di testo, li ordina seguendo il formato *punteggio:mosse:livello:nome tramite un algoritmo di Insertion-Sort,* e restituisce i primi 10 classificati.
Metodi rilevanti: save, load, sorting

Nei paragrafi seguenti verrà analizzata ogni classe nel dettaglio, soffermandosi sui metodi più rilevanti dal punto di vista di logica del gioco.

Metodi principali

(Game) setup_game: inizializza una nuova sessione di gioco e grazie alle funzioni di creazione generate_grid di Grid e is_reachable di Pathfinder garantisce una mappa risolvibile. La sua funzione è quella di creare una mappa valida entro 10 tentativi, per esserlo deve generare una griglia 20x20, piazzare tutte le celle correttamente e creare il punto di partenza e l'obiettivo rispettando i criteri imposti.

(Game) move_player: gestisce il movimento del giocatore in una delle quattro direzioni (N, S, E, W). Il metodo verifica preventivamente se lo spostamento è valido (ovvero se la cella di destinazione è nei limiti e non è un muro); in caso positivo, aggiorna la posizione e decrementa il punteggio per il movimento. Ogni 5 mosse, innesca l'evoluzione della griglia e ricalcola la raggiungibilità dell'obiettivo.

(Game) use_special_action: permette l'utilizzo delle mosse speciali sulle celle adiacenti alla posizione del giocatore: remove_wall per abbattere un muro e convert_trap per convertire una trappola in risorsa. Il metodo verifica che la cella sia adiacente, che le mosse speciali non siano già state utilizzate e che il tipo di cella sia compatibile con l'abilità scelta.

(Grid) generate_grid: richiama la funzione generative_dfs che utilizza l'algoritmo DFS descritto in seguito per la creazione della mappa con almeno un cammino percorribile. Una volta definita questa struttura, calcola il numero di elementi da inserire (muri, risorse e trappole) a seconda del livello di difficoltà. Infine, richiama il metodo _adjust_cells per aggiungere o rimuovere elementi posizionati in modo casuale fino al numero corretto, e _place_special_cells che si occupa di posizionare il punto di partenza e l'obiettivo su due celle vuote.

(Grid) generative_dfs: questo metodo genera un percorso, partendo da una cella iniziale e visitando le altre celle in modo iterativo tramite stack. Questo algoritmo viene spiegato più nel dettaglio nella sezione dedicata agli algoritmi principali del progetto.

(Grid) step: questo metodo effettua l'evoluzione della griglia secondo le seguenti regole:

- Due risorse non raccolte vengono trasformate in celle vuote,
- Una cella vuota casuale viene trasformata in risorsa,
- Due celle vuote casuali vengono trasformate in trappole,
- Una trappola casuale viene trasformata in cella vuota,
- L'evoluzione non può modificare la posizione iniziale, l'obiettivo, i muri e la Safe Zone (posizione del giocatore e celle adiacenti a questa).

Presenta un metodo annidato pick_cells che seleziona casualmente delle coordinate sulla griglia fino a quando nessuna corrisponde alla Safe Zone. Questo metodo viene utilizzato dalla funzione esterna per cambiare il tipo delle celle secondo le regole. Viene richiamata nel metodo move_player della classe Player ogni 5 passi.

(Leaderboard) sorting: trasforma il dizionario dei punteggi in una lista di tuple chiamata current_data, e inizializza una lista vuota, sorted_data, dove inserire gli elementi ordinati. Attraverso il ciclo for principale, per ogni giocatore in current_data, estrae le statistiche. A questo punto entra nel for annidato dove ad ogni iterazione vengono estrapolate le informazioni dell'i-esimo giocatore e confrontate con quello attuale (funzione is_better). Se il giocatore corrente è migliore dell i-esimo viene inserito in quella posizione e vengono fatti scivolare tutti i giocatori successivi di un posto con sorted_data.insert(i, item). Se non è inserito viene appeso in coda alla lista sorted_data. Torna sorted_data come lista di tuple.
L'uso di questo algoritmo è ottimale poiché la classifica viene aggiornata un elemento alla volta. L'algoritmo è stabile e **impiega O(n) nel caso migliore**

cioè quando i dati sono già parzialmente ordinati, come accade nella classifica).

(Leaderboard) save: se il file classifica.txt non esiste viene creato, altrimenti lo apre in modalità scrittura e sovrascrive i dati presenti solo se il nome del giocatore non è presente nel dizionario o il nuovo risultato è un record personale, verificando tramite il metodo sorting.

(Leaderboard) load: apre il file classifica.txt in modalità lettura e carica i dati dal file nel dizionario 'scores' dict[nome: statistiche] solo se il formato è corretto.

4.2. Implementazione della View

La View rappresenta l'interfaccia grafica dell'applicazione e si occupa esclusivamente della visualizzazione dello stato del gioco e la ricezione degli input da parte dell'utente.
Essa non contiene logica di gioco, ma comunica con il Controller, che a sua volta interagisce con il Model, interrogandolo sulle informazioni che servono alla View.

La View gestisce in particolare:

- La visualizzazione delle diverse scene (GameView, StartScreen etc.)
- La ricezione dell'input da parte dell'utente

L'aggiornamento della griglia e delle statistiche in tempo reale

4.2.1. Panoramica delle classi

Classe **BaseView**

La classe BaseView è la classe astratta da cui ereditano tutte le schermate. Fornisce metodi comuni per la gestione della finestra, insieme a metodi obbligatori per la costruzione dell'interfaccia.

Classe **MainView**

La classe MainView rappresenta la finestra principale su cui vengono costruite tutte le altre componenti di View.
Coordina la transizione tra le diverse scene, e fornisce metodi per la pulizia della finestra.

Classe **StartScreen**

La classe StartScreen è la prima finestra con cui l'utente interagisce quando avvia il programma.
Si occupa di costruire il Menù principale, dove l'utente può inserire il proprio nickname e avviare il gioco. Inoltre sono presenti i pulsanti per la visualizzazione della classifica e delle istruzioni.

Classe **GameView**

La classe GameView si occupa di mostrare la schermata di gioco vera e propria.
Disegna la griglia, aggiorna le celle, mostra il punteggio corrente insieme alle abilità speciali.

Classe **LeaderboardView**

La classe LeaderboardView contiene la classifica fino ai primi 10 giocatori. Genera una finestra accessibile dallo StartScreen e mostra i classificati secondo i criteri della leaderboard, mostrando nome, livello, punti e mosse.

Classe **SaveScoreDialog**

La classe SaveScoreDialog mostra un Dialog Box dove l'utente, nel caso in cui abbia vinto, può scegliere se salvare il suo punteggio o meno, dopo aver cliccato il pulsante "Riprova" oppure "Menù Principale".

Classe **GameInstructions**

La classe GameInstructions rappresenta la finestra con le istruzioni di gioco.

Crea una schermata accessibile dal MenuBar della schermata iniziale e di gioco e mostra le regole del gioco, la legenda dei colori e cosa comporta passare sopra ad ognuno, oltre ai muri non attraversabili.

Classe **DifficultyDialog**

La classe DifficultyDialog mostra un overlay per la scelta della difficoltà. Si presenta come una schermata con le difficoltà “Facile”, “Medio” e “Difficile”

Metodi principali

(MainView) switch_to: centralizza la logica di transizione da una scena a un’altra, salvando lo stato della scena precedente e istanziando la nuova. Il metodo accetta come parametro una classe ‘view_class’ e utilizza la seguente logica:

1. **Gestione dello storico:** Il sistema salva una funzione ‘congelata’ nell’attributo came_from, con il riferimento alla scena attuale. Questo permette di implementare facilmente tasti “Indietro”.
2. **Pulizia della memoria:** Viene chiamato un metodo interno ‘*clear_windows’,* che distrugge tutti i widget presenti e ripristina i pesi delle colonne.
3. **Istanziazione dinamica:** Grazie all’uso di *args e **kwargs, la funzione è universale. Può istanziare una classe senza parametri, come StartScreen, o una come Leaderboard, che richiede gli scores.
4. **Preservazione dello stato tramite Lambda:** Nell’attributo current_screen_func viene ‘congelata’ la chiamata alla scena attuale, in modo da poterla ricaricare esattamente nel suo stato attuale in futuro. Questo serve nel caso di pulsanti ‘Indietro’

(MainView) update_game: funge da ponte tra il Controller e la GameView. Riceve come argomento un dizionario di stato. L’uso del dizionario permette di trasferire le informazioni del gioco alla GameView, a cui delega

l'aggiornamento grafico attraverso la funzione 'update_game_view' della stessa GameView.

(GameView) update_game_view: è il metodo che permette il refresh totale della view dopo ogni evento scatenato dall'utente attraverso i seguenti metodi:

- *'refresh_grid_display'* svuota lo schermo, fa in modo che la griglia sia sempre al centro del canvas e ridisegna i colori di ogni cella,
- *'update_player_position_display'* cancella solo il disegno sulla vecchia posizione del giocatore e lo disegna nella nuova,
- *'update_stats'* aggiorna le Label delle statistiche sullo schermo, escludendo il timer che viene gestito in modo asincrono.

(GameView) handle_double_click gestisce l'attivazione delle abilità speciali tramite l'intercettazione del doppio click del mouse nel canvas della griglia.
Calcola l'Offset: determina lo spazio vuoto ai bordi della griglia per determinare il punto di origine della prima cella.
Sottrae all'offset le coordinate dell'evento per ottenere la posizione relativa alla griglia.
Divide i valori ottenuti per la dimensione della cella, ottenendo le coordinate 'row' e 'column' della matrice.
Una volta individuata la cella cliccata:

- Se è un muro ('X'), attiva l'abilità di rimozione del muro
- Se è una trappola ('T'), attiva l'abilità per convertire la trappola in risorsa.

Infine, invia la richiesta al Controller per verificare se l'azione è possibile, se non lo è viene richiamato il metodo '*screen_flicker*' che fornisce un feedback visivo per comunicare l'errore all'utente.

4.3. Controller

Il Controller rappresenta il collegamento tra la View e il Model.
Il suo compito principale è ricevere gli input dall'utente dalla View, aggiornare il Model sulla base degli input, e aggiornare la View sui cambiamenti del Model

In particolare, il Controller gestisce:

- L'acquisizione degli input dell'utente che arrivano dalla View
- L'invio degli input al Model
- L'aggiornamento della View con i dati restituiti dal Model
- La gestione del ciclo di gioco (inizio, fine partita)

Metodi principali

handle_start_game_request: viene chiamata quando si clicca sul pulsante di inizio gioco. Chiama la funzione per mostrare la scelta della difficoltà.

handle_selected_difficulty: dopo che la difficoltà è stata scelta, avvia il gioco e cambia la finestra

handle_movement_request: gestisce il movimento del giocatore, aggiorna l'interfaccia grafica di GameView e verifica se il gioco è terminato

handle_game_over: gestisce la fine della partita, inviando a GameView la ragione di fine partita

refresh_view: ottiene lo stato dalla classe Game e invoca il metodo 'update_game' di MainView per aggiornare l'interfaccia di gioco

4.4. Pathfinder (Service)

Il Pathfinder rappresenta un servizio di supporto alla logica di gioco, utilizzato dal Model per verificare la raggiungibilità dell'obiettivo durante la

generazione e ogni evoluzione della griglia.
Il servizio implementa un algoritmo di Breadth First Search (BFS) adattato alle regole di gioco, tenendo conto di muri, risorse, trappole e abilità speciali.

4.4.1. Gestione dello Stato

In un labirinto classico, una cella è “visitata” o “non visitata” in base alla sua posizione. Nel nostro progetto, l’accessibilità ad una cella dipende dal suo stato nel momento in cui è attraversata. Lo stato è definito come tale:

$$Stato = (Posizione,\ Muri\ Rimovibili,\ Trappole\ Convertibili,\ Punteggio)$$

Se l’algoritmo raggiunge la cella (2,3) con 10 punti e successivamente la raggiunge con 20 punti, lo stato è considerato diverso. Lo stesso vale con le abilità speciali. Questo permette alla BFS di rivalutare percorsi precedentemente scartati, in modo da garantire la tracciabilità di una soluzione ottimale.

4.4.2. Backtracking

Mentre la BFS classica determina quanti passi sono necessari per raggiungere l’obiettivo, il Pathfinder restituisce anche l’effettiva strada percorsa. Per ottenere questo risultato senza appesantire troppo il programma, è stata implementata una struttura a dizionario di parentela del tipo dict[child:parent]:

- Ogni volta che un figlio (cella adiacente) viene aggiunto alla coda, viene salvato nel dizionario il legame parent[child_state] = parent_state.
- Una volta raggiunto l’obiettivo, l’algoritmo interrompe la ricerca e risale di genitore in genitore fino a raggiungere la posizione iniziale (finché parent[child_state] == None).
- Ritorna una lista ordinata di coordinate che rappresenta il cammino ideale.

4.4.3. Analisi della complessità logica

L'algoritmo utilizzato dal Pathfinder ha una complessità logica maggiore di un BFS standard, a causa della sua natura.
In una BFS classica, la complessità sarebbe O(V + E), dove V sono i nodi (vertices) ed E sono gli archi (edges). Nel nostro caso, il numero di nodi è moltiplicato per le combinazioni possibili delle variabili di stato:

- W: width
- H: height
- S: score
- B: breakable walls
- C: convertible traps

Perciò, la complessità logica, nel peggiore dei casi è O(W*H*S*B*C)

4.5. Algoritmo DFS

L'algoritmo Depth-First Search (DFS) o ricerca in profondità, in versione generativa ha l'obiettivo di creare un labirinto che colleghi tutte le celle, assicurandosi che non ci siano aree isolate. Utilizza lo **stack (Last In, First Out)**. L'algoritmo scava corridoi all'interno di una griglia inizialmente composta solo da muri, partendo da una cella di partenza, inizializzata a (0,0), e segue questo procedimento:

1. L'algoritmo si muove di due unità alla volta. Questo è necessario per mantenere un muro di separazione tra i corridoi, per evitare che vengano create intere aree di spazi vuoti.
2. Dalla cella corrente, l'algoritmo identifica se ci sono dei vicini non visitati, e ne sceglie uno casualmente (random.choice).

3. Quando si sposta da una cella (x, y) a una (nx, ny), l'algoritmo rimuove il muro sulla cella in cui si è spostato, e quello nella cella intermedia (mid_x, mid_y).
4. Utilizzando una pila (stack), l'algoritmo può ritornare sui suoi passi quando trova un vicolo cieco, e riprendere da una cella che ha dei vicini non visitati.

La **complessità logica è O(H*W)**, dove H=height e W=width, quindi è lineare rispetto al numero totale di celle della griglia.

Divisione dei ruoli

Tutti i componenti del team hanno lavorato in modo equo ed equilibrato, contribuendo alla progettazione e alla creazione di tutte le classi dell'applicazione. In particolare, il lavoro è stato svolto nel seguente modo:

Silvia Farina: Si è occupata della progettazione e implementazione dell'interfaccia grafica, gestendo la rappresentazione della griglia di gioco, la transizione tra le scene e l'aggiornamento grafico dello stato della partita.
Ha inoltre collaborato all'implementazione delle interazioni utente, sia specifiche del gioco che dei pulsanti al di fuori di esso.

Sonia Culotta: Ha implementato la logica relativa alla classifica, gestendo il salvataggio, caricamento e ordinamento dei dati. Insieme alla classifica, ha gestito la logica del Timer. Si è inoltre occupata dell'interazione tra View e Controller, assicurando il corretto flusso di informazioni e la separazione delle logiche.

Gioele Spanò: Ha svolto il ruolo di coordinatore del team, organizzando il lavoro di gruppo e facilitando l'integrazione delle diverse componenti del progetto. Si è concentrato sugli algoritmi di generazione e validazione della griglia basati su BFS e DFS, sulla centralizzazione della logica di gioco nel Model e sulla gestione della connessione tra Model, View e Controller.

Conclusioni

Il progetto “Percorso Evolutivo” ha raggiunto tutti gli obiettivi prefissati in fase di analisi. L’adozione del pattern MVC si è rivelato fondamentale per mantenere il codice leggibile e facilmente manutenibile, consentendo di separare la logica di business dall’interfaccia grafica. Questo approccio ha reso possibile testare le singole componenti del codice indipendentemente.

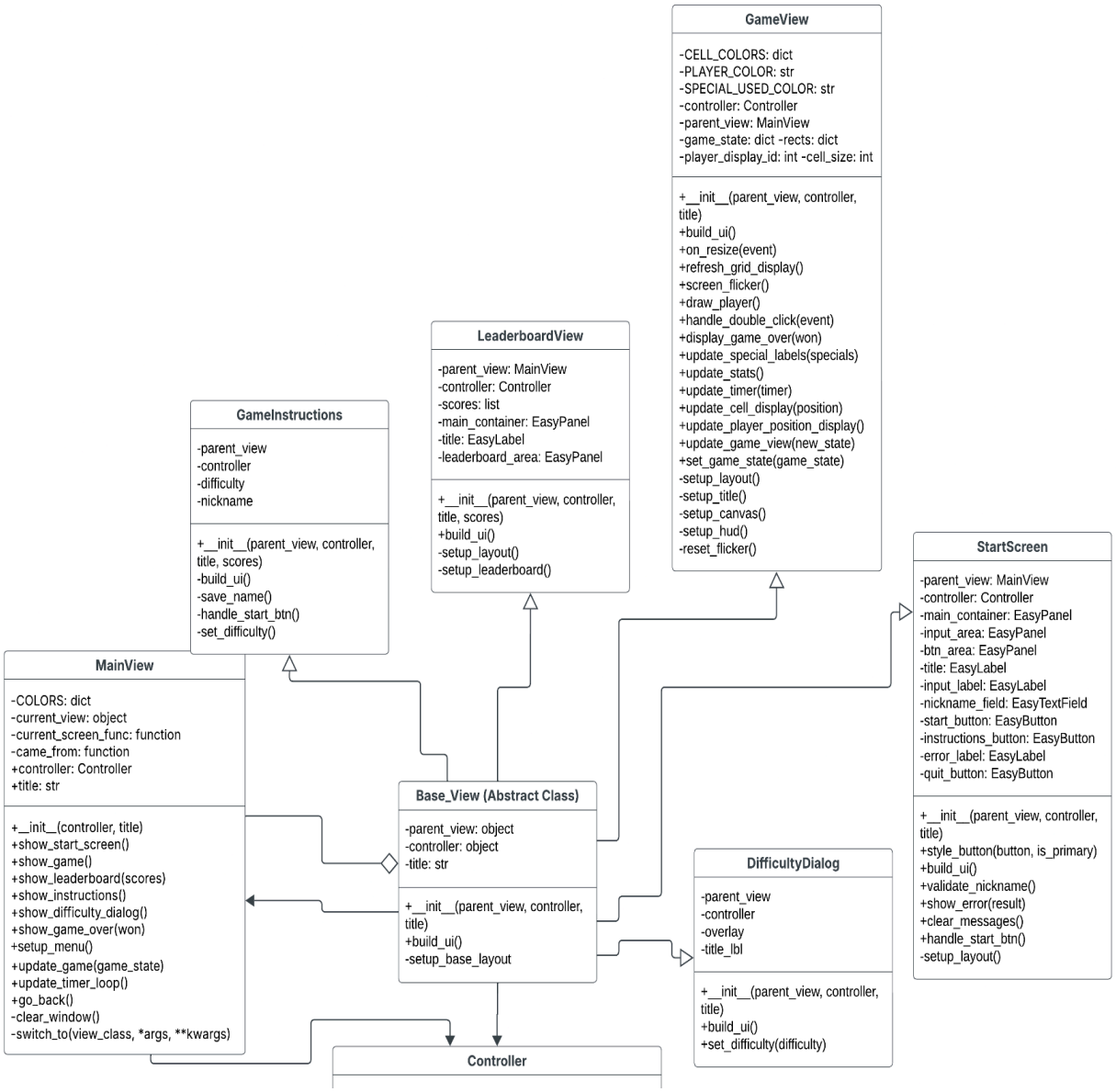
L’integrazione del Pathfinder, insieme all’implementazione del DFS generativo, ha garantito la giocabilità dell’applicazione, eliminando il rischio di generare mappe irrisolvibili, mentre l’uso dell’Insertion Sort per la classifica ha permesso di gestire il salvataggio su file senza rallentare il software.

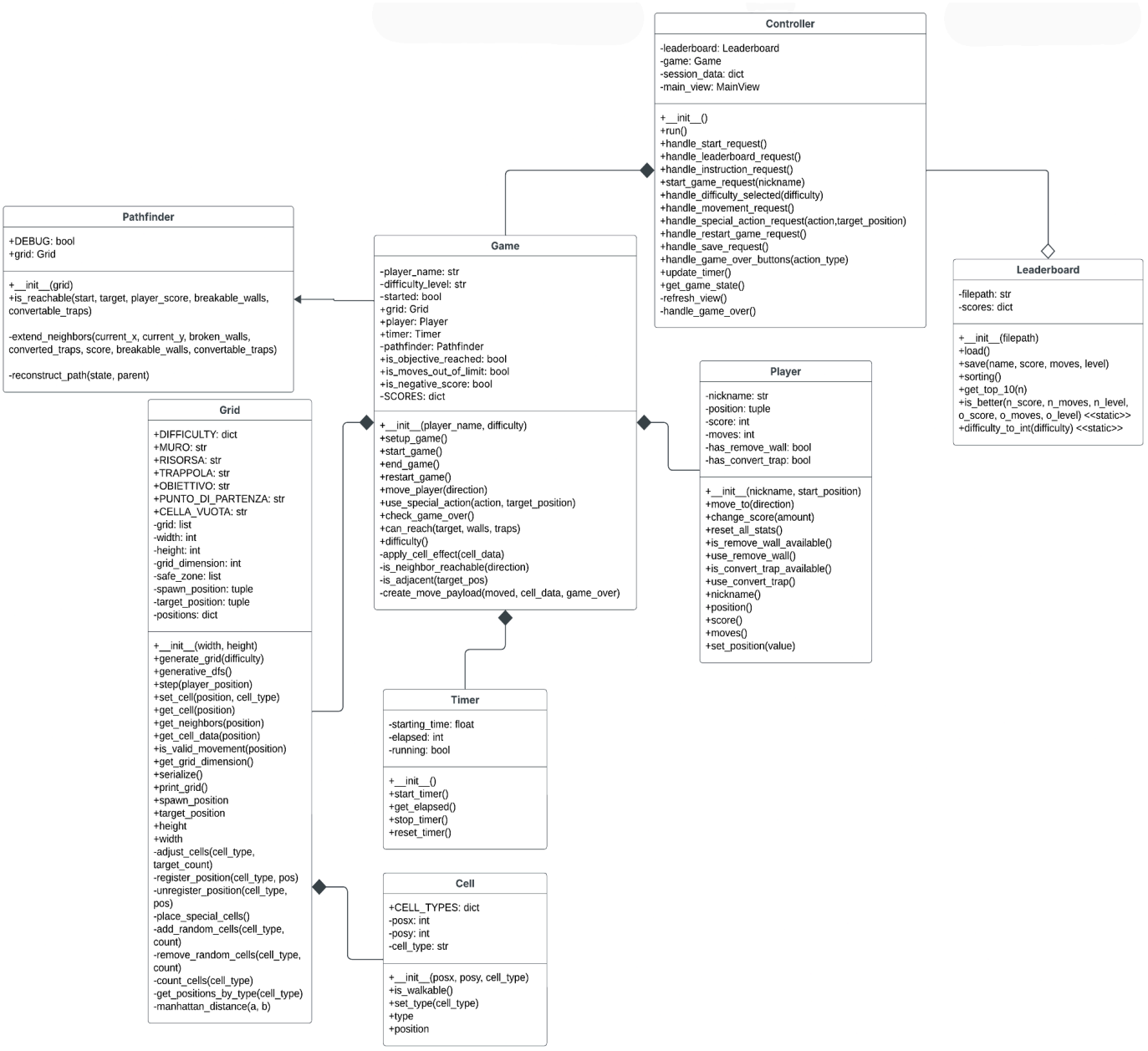
Lo sviluppo di questa applicazione ha permesso di consolidare e approfondire una serie di competenze fondamentali nell’ambito informatico:

- Applicazione pratica di design pattern, in questo caso MVC, per facilitare la manutenzione e il testing del codice.
- Creazione di GUI intuitive, con la gestione di layout responsivi, gestione degli eventi e input dell’utente, e feedback visivi.
- Attività di problem solving, come ad esempio la risoluzione di criticità legate alla sincronizzazione tra evoluzione della griglia e posizione del giocatore.

Un’altra caratteristica del progetto che è importante evidenziare è che è stato concepito con un’ottica modulare che lo rende facilmente scalabile. La struttura attuale permette di espandere il gioco e risolvere eventuali criticità con sforzi minimi.

Appendice A


GameView
-CELL_COLORS: dict
-PLAYER_COLOR: str
-SPECIAL_USED_COLOR: str
-controller: Controller
-parent_view: MainView
-game_state: dict -rects: dict
-player_display_id: int -cell_size: int
+__init__(parent_view, controller, title)
+build_ui()
+on_resize(event)
+refresh_grid_display()
+screen_flicker()
+draw_player()
+handle_double_click(event)
+display_game_over(won)
+update_special_labels(specials)
+update_stats()
+update_timer(timer)
+update_cell_display(position)
+update_player_position_display()
+update_game_view(new_state)
+set_game_state(game_state)
-setup_layout()
-setup_title()
-setup_canvas()
-setup_hud()
-reset_flicker()
LeaderboardView
-parent_view: MainView
-controller: Controller
-scores: list
-main_container: EasyPanel
-title: EasyLabel
-leaderboard_area: EasyPanel
+__init__(parent_view, controller, title, scores)
+build_ui()
-setup_layout()
-setup_leaderboard()
GameInstructions
-parent_view
-controller
-difficulty
-nickname
+__init__(parent_view, controller, title, scores)
-build_ui()
-save_name()
-handle_start_btn()
-set_difficulty()
StartScreen
-parent_view: MainView
-controller: Controller
-main_container: EasyPanel
-input_area: EasyPanel
-btn_area: EasyPanel
-title: EasyLabel
-input_label: EasyLabel
-nickname_field: EasyTextField
-start_button: EasyButton
-instructions_button: EasyButton
-error_label: EasyLabel
-quit_button: EasyButton
+__init__(parent_view, controller, title)
+style_button(button, is_primary)
+build_ui()
+validate_nickname()
+show_error(result)
+clear_messages()
+handle_start_btn()
-setup_layout()
MainView
-COLORS: dict
-current_view: object
-current_screen_func: function
-came_from: function
+controller: Controller
+title: str
+__init__(controller, title)
+show_start_screen()
+show_game()
+show_leaderboard(scores)
+show_instructions()
+show_difficulty_dialog()
+show_game_over(won)
+setup_menu()
+update_game(game_state)
+update_timer_loop()
+go_back()
-clear_window()
-switch_to(view_class, *args, **kwargs)
Base_View (Abstract Class)
-parent_view: object
-controller: object
-title: str
+__init__(parent_view, controller, title)
+build_ui()
-setup_base_layout
DifficultyDialog
-parent_view
-controller
-overlay
-title_lbl
+__init__(parent_view, controller, title)
+build_ui()
+set_difficulty(difficulty)
Controller


Controller
-leaderboard: Leaderboard
-game: Game
-session_data: dict
-main_view: MainView
+__init__()
+run()
+handle_start_request()
+handle_leaderboard_request()
+handle_instruction_request()
+start_game_request(nickname)
+handle_difficulty_selected(difficulty)
+handle_movement_request()
+handle_special_action_request(action,target_position)
+handle_restart_game_request()
+handle_save_request()
+handle_game_over_buttons(action_type)
+update_timer()
+get_game_state()
-refresh_view()
-handle_game_over()
Pathfinder
+DEBUG: bool
+grid: Grid
+__init__(grid)
+is_reachable(start, target, player_score, breakable_walls, convertable_traps)
-extend_neighbors(current_x, current_y, broken_walls, converted_traps, score, breakable_walls, convertable_traps)
-reconstruct_path(state, parent)
Game
-player_name: str
-difficulty_level: str
-started: bool
+grid: Grid
+player: Player
+timer: Timer
-pathfinder: Pathfinder
+is_objective_reached: bool
+is_moves_out_of_limit: bool
+is_negative_score: bool
-SCORES: dict
+__init__(player_name, difficulty)
+setup_game()
+start_game()
+end_game()
+restart_game()
+move_player(direction)
+use_special_action(action, target_position)
+check_game_over()
+can_reach(target, walls, traps)
+difficulty()
-apply_cell_effect(cell_data)
-is_neighbor_reachable(direction)
-is_adjacent(target_pos)
-create_move_payload(moved, cell_data, game_over)
Leaderboard
-filepath: str
-scores: dict
+__init__(filepath)
+load()
+save(name, score, moves, level)
+sorting()
+get_top_10(n)
+is_better(n_score, n_moves, n_level, o_score, o_moves, o_level) <<static>>
+difficulty_to_int(difficulty) <<static>>
Player
-nickname: str
-position: tuple
-score: int
-moves: int
-has_remove_wall: bool
-has_convert_trap: bool
+__init__(nickname, start_position)
+move_to(direction)
+change_score(amount)
+reset_all_stats()
+is_remove_wall_available()
+use_remove_wall()
+is_convert_trap_available()
+use_convert_trap()
+nickname()
+position()
+score()
+moves()
+set_position(value)
Grid
+DIFFICULTY: dict
+MURO: str
+RISORSA: str
+TRAPPOLA: str
+OBIETTIVO: str
+PUNTO_DI_PARTENZA: str
+CELLA_VUOTA: str
-grid: list
-width: int
-height: int
-grid_dimension: int
-safe_zone: list
-spawn_position: tuple
-target_position: tuple
-positions: dict
+__init__(width, height)
+generate_grid(difficulty)
+generative_dfs()
+step(player_position)
+set_cell(position, cell_type)
+get_cell(position)
+get_neighbors(position)
+get_cell_data(position)
+is_valid_movement(position)
+get_grid_dimension()
+serialize()
+print_grid()
+spawn_position
+target_position
+height
+width
-adjust_cells(cell_type, target_count)
-register_position(cell_type, pos)
-unregister_position(cell_type, pos)
-place_special_cells()
-add_random_cells(cell_type, count)
-remove_random_cells(cell_type, count)
-count_cells(cell_type)
-get_positions_by_type(cell_type)
-manhattan_distance(a, b)
Timer
-starting_time: float
-elapsed: int
-running: bool
+__init__()
+start_timer()
+get_elapsed()
+stop_timer()
+reset_timer()
Cell
+CELL_TYPES: dict
-posx: int
-posy: int
-cell_type: str
+__init__(posx, posy, cell_type)
+is_walkable()
+set_type(cell_type)
+type
+position