Relazione Progetto Programmazione III e Laboratorio di Programmazione III

Antonio Iannone 0124002543

Francesco Pio Gravino 0124002703

GitHub Repository

Contents

1 Traccia del progetto **3**

2 Intelligenza Artificiale **5**

3 Implementazione dell'Algoritmo di Dijkstra **9**

3.1 Dichiarazione e Costruttore 11

3.2 Metodo `calculateShortestPath` 11

3.3 Metodo `getShortestPath` 12

3.4 Struttura UML 13

4 Ruolo e Scopo della Classe `Graph` **14**

5 Ruolo e Scopo della Classe `Interactors` **15**

6 Design pattern **16**

7 UML **32**

8 Analisi Principi SOLID **33**

9 Interfaccia Grafica **36**

1 Traccia del progetto

Si vuole sviluppare un algoritmo per il cammino di un robot in un labirinto. La stanza è pavimentata a tasselli quadrati (caselle) ed è dotata di pareti esterne e interne (vedi Figure 1 per un esempio). Il robot si può muovere nelle 8 direzioni adiacenti. Devono essere previsti almeno tre diversi scenari (livelli di difficoltà). Nella stanza sono presenti degli oggetti di diverso colore (red, green, yellow, cyan) che compaiono e scompaiono casualmente sul percorso. Il robot può assumere quattro stati pursuit, evade, flee, seek. Se il robot si trova in prossimità di un oggetto cambia stato secondo lo schema della Macchina a Stati Finiti di 2.

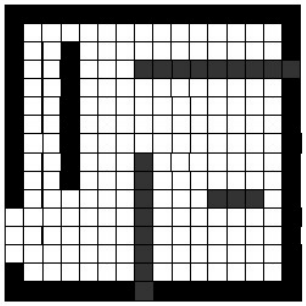


Figure 1: Esempio di percorso

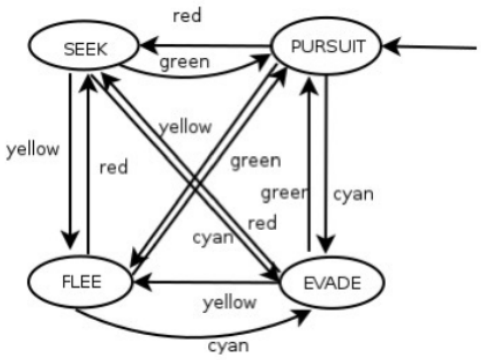


Figure 2: Macchina a stati finiti

Per ogni stato le strategie sono:

- **Pursuit e Seek**:
  - Il robot si muove nella direzione dell'uscita di una singola cella alla volta.
  - Per determinare il percorso, utilizzare l'algoritmo di Dijkstra (vedi 2).
- **Flee**:
  - Il robot si muove nella direzione dell'uscita di due celle alla volta.
  - Per determinare il percorso, utilizzare l'algoritmo di Dijkstra (vedi 2).
- **Evade**:
  - Il robot avanza in maniera casuale di una singola cella.

Il programma per la gestione del labirinto deve rispettare le seguenti caratteristiche:

1. Permettere la registrazione del robot inserendo:
   - Nome
   - Cognome
2. Visualizzare, all'inizio e alla fine di ogni partita, la classifica dei risultati migliori ottenuti, considerando:
   - Il minor numero di passi necessari per raggiungere l'uscita.
   - I dati di tutti i robot registrati in tutte le partite.
3. Mostrare il percorso del robot dopo ogni passo:
   - Rappresentando graficamente la stanza.
   - Indicando la posizione del robot e quella degli oggetti presenti.

2 Intelligenza Artificiale

Per il progetto in questione è stato richiesto, per simulare il cammino dei robot, di utilizzare l'algoritmo di ***Dijkstra***. *The Dijkstra's Shortest Path Algorithm* è uno dei metodi più celebri e utilizzati per la risoluzione del problema del cammino minimo in un grafo pesato con pesi non negativi. Ideato nel 1956 dal celebre informatico olandese Edsger W. Dijkstra, fu successivamente formalizzato e pubblicato nel 1959 nel suo articolo "A Note on Two Problems in Connexion with Graphs". Grazie alla sua semplicità ed efficienza, l'algoritmo si è rapidamente affermato come un pilastro della teoria dei grafi e dell'informatica, trovando applicazione in numerosi ambiti, dall'ingegneria delle reti all'intelligenza artificiale.

Funzionamento e Applicazioni:

L'algoritmo permette di determinare il cammino minimo tra un nodo sorgente e tutti gli altri nodi di un grafo. Il risultato del processo è un albero dei cammini minimi, che identifica il percorso più breve per ciascun nodo rispetto alla sorgente. Per questo motivo, Dijkstra è particolarmente adatto a problemi che richiedono l'ottimizzazione di percorsi, come la pianificazione di rotte in sistemi di navigazione (GPS), il routing nei protocolli di rete (ad esempio, OSPF), e l'analisi dei flussi di traffico su reti stradali o infrastrutture complesse.

Un aspetto fondamentale che contraddistingue l'algoritmo è la sua restrizione a grafi con pesi non negativi sugli archi. Questa limitazione assicura che i risultati prodotti siano corretti, poiché la logica del metodo si basa sull'assunzione che aggiungere un peso non negativo non possa mai ridurre la lunghezza totale di un cammino.

Origine e Contesto storico:

L'idea dell'algoritmo nacque in maniera sorprendentemente informale: come raccontato dallo stesso Dijkstra in un'intervista, egli concepì il metodo in circa venti minuti, durante una pausa in un bar ad Amsterdam. La semplicità e l'efficacia dell'approccio derivano dalla sua progettazione mentale, priva dell'ausilio di carta e penna, che costrinse il ricercatore a evitare ogni complessità superflua.

Nonostante la sua genesi apparentemente casuale, l'algoritmo di Dijkstra ha avuto un impatto significativo, diventando uno degli strumenti più studiati e implementati nella teoria dei grafi. La sua rilevanza è ulteriormente amplificata dalla crescente importanza delle reti e

della modellazione di sistemi complessi, dove la determinazione del percorso più breve è un problema centrale.

Concetti di Base dell'Algoritmo di Dijkstra

L'algoritmo di Dijkstra è un metodo iterativo e greedy per risolvere il problema del cammino minimo in un grafo pesato con pesi non negativi. Il suo funzionamento si basa su una sequenza di aggiornamenti successivi, finalizzati a determinare il percorso più breve dal nodo sorgente a tutti gli altri nodi del grafo.

Struttura Generale del Funzionamento

L'algoritmo opera su alcuni concetti chiave, che possono essere descritti come segue:

- **Nodi e Distanze**: Ogni nodo del grafo è associato a una distanza inizialmente posta a ∞, eccetto il nodo sorgente, la cui distanza è 0. Durante l'esecuzione dell'algoritmo, queste distanze rappresentano i migliori cammini scoperti fino a quel momento.
- **Insiemi di Nodi**: Si definiscono due insiemi distinti:
  - S, che contiene i nodi visitati, ovvero quelli per cui è stato determinato il cammino minimo definitivo.
  - T, che contiene i nodi non ancora visitati.
- **Aggiornamento Greedy**: Ad ogni iterazione, viene scelto il nodo in T con la minima distanza temporanea dal nodo sorgente. Questo nodo viene visitato e le distanze verso i suoi vicini vengono aggiornate, se si trova un percorso più breve.
- **Etichette dei Nodi**: Ogni nodo è caratterizzato da:
  - $f(i)$, che rappresenta la distanza minima calcolata dal nodo sorgente al nodo i.
  - $J(i)$, che identifica il predecessore del nodo i nel cammino minimo.

Dettagli dell'Algoritmo

L'algoritmo di Dijkstra può essere descritto formalmente nei seguenti passi:

1. Inizializzazione.

- Si pongono $f(1) = 0$ e $J(1) = 0$, indicando che il nodo sorgente è raggiunto a costo nullo.
- Per ogni nodo i adiacente al nodo sorgente, si inizializzano:

$$f(i) = p(1, i), \quad J(i) = 1,$$

  dove $p(1, i)$ è il peso dell'arco tra il nodo 1 e il nodo i.
- Per tutti gli altri nodi i, si assegna $f(i) = \infty$, indicando che il cammino verso di essi non è ancora noto.

2. Iterazione.

1. Selezionare il nodo $j \in T$ con la minima distanza temporanea:

$$f(j) = \min\{f(i) \mid i \in T\}.$$

2. Aggiungere j all'insieme S e rimuoverlo da T:

$$S = S \cup \{j\}, \quad T = T \setminus \{j\}.$$

3. Se $T = \emptyset$, l'algoritmo termina.

3. Aggiornamento delle Etichette Provvisorie.

- Per ogni nodo $i \in T$ adiacente a j, si verifica se esiste un percorso più breve passando per j:

$$f(i) > f(j) + p(j, i).$$

- Se la condizione è soddisfatta, si aggiornano:

$$f(i) = f(j) + p(j, i), \quad J(i) = j.$$

4. Condizione di Termine. L'algoritmo termina quando:

- $T = \emptyset$, ovvero tutti i nodi sono stati visitati, oppure
- $f(i) = \infty$ per ogni $i \in T$, indicando che non esistono percorsi dal nodo sorgente a quei nodi.

Considerazioni Aggiuntive

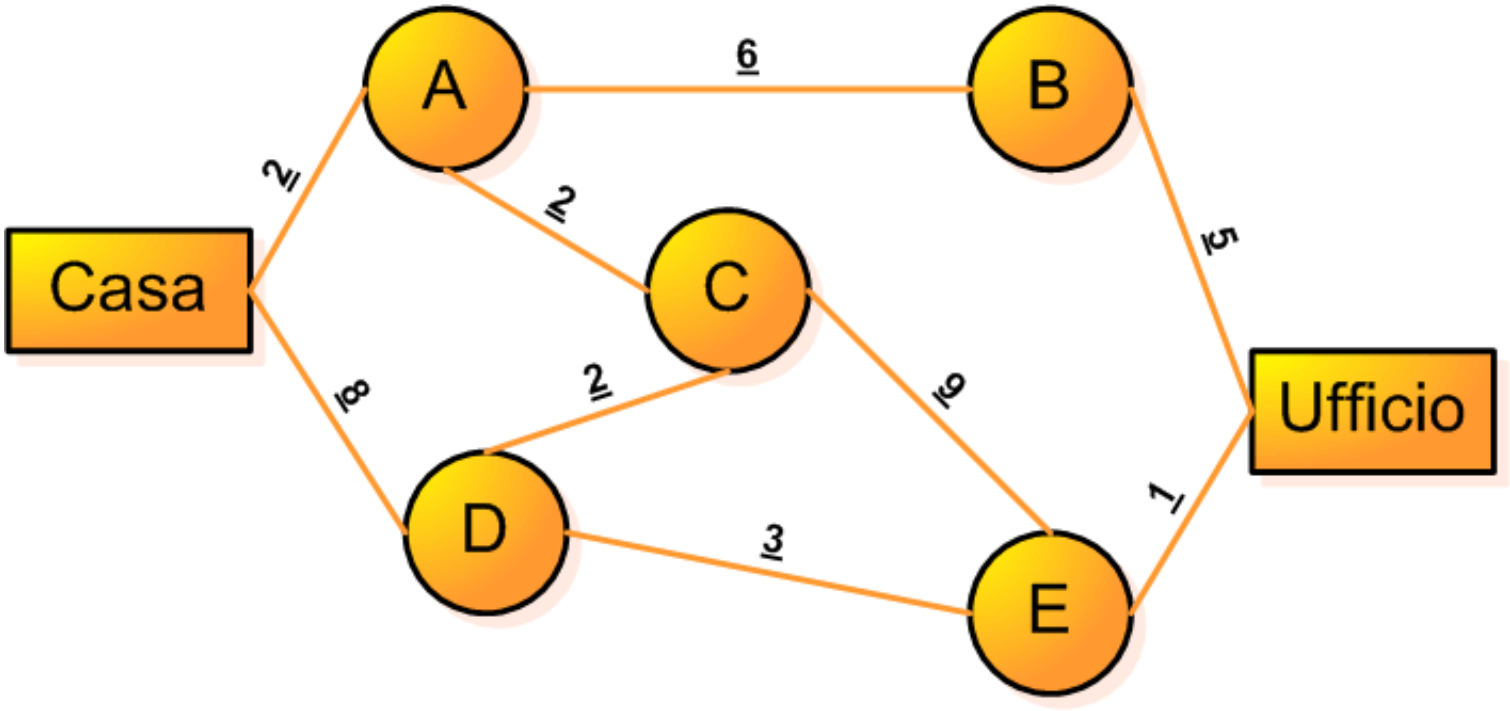
- **Proprietà di Ottimalità**: L'algoritmo si basa sull'assunzione che, in un grafo con pesi non negativi, una volta determinato il cammino minimo per un nodo, esso non può essere migliorato.
- **Efficienza Computazionale**: Utilizzando una coda di priorità per l'aggiornamento delle distanze, l'algoritmo può essere implementato con complessità $O((|V|+|E|) \log |V|)$, dove $|V|$ è il numero di nodi e $|E|$ il numero di archi del grafo.
- **Limiti**: L'algoritmo non è applicabile a grafi con pesi negativi. In questi casi, è necessario utilizzare metodi alternativi come l'algoritmo di Bellman-Ford.

La descrizione fornita in questo paragrafo rappresenta una panoramica generale e accademica dell'algoritmo di Dijkstra, con l'obiettivo di introdurre i concetti fondamentali e il funzionamento di questo metodo. Nel paragrafo successivo, analizzeremo come l'algoritmo è stato applicato nel contesto specifico del nostro progetto, illustrando gli adattamenti e le implementazioni utilizzate per affrontare il problema in esame.

Le informazioni presentate in questa sezione sono state elaborate a partire da fonti autorevoli, tra cui:

- *Algoritmo di Dijkstra* disponibile su Wikipedia all'indirizzo: `https://it.wikipedia.org/wiki/Algoritmo_di_Dijkstra`.

- Il libro di testo *Introduzione agli algoritmi e strutture dati* di Thomas H. Cormen, un riferimento fondamentale per lo studio degli algoritmi.



3 Implementazione dell'Algoritmo di Dijkstra

L'algoritmo di Dijkstra, nel contesto del nostro progetto, è stato implementato interamente in *Java*, il linguaggio di programmazione utilizzato per tutto il codice del progetto. L'implementazione è stata adattata per rispondere alle esigenze specifiche del nostro scenario, che prevede la navigazione di un robot all'interno di un labirinto e l'esecuzione di diverse strategie di movimento.

Ruolo dell'Algoritmo nel Progetto

L'algoritmo di Dijkstra viene impiegato per calcolare il cammino minimo necessario al robot per muoversi attraverso il labirinto. Questo processo è strettamente legato a tre strategie di movimento implementate nel progetto:

- **Strategia *Pursuit***: In questa strategia, il robot utilizza l'algoritmo di Dijkstra per identificare l'uscita più vicina dalla cella attuale e spostarsi verso di essa, un passo alla

volta.

- **Strategia *Seek***: Analogamente alla strategia *Pursuit*, anche in questo caso l'algoritmo viene utilizzato per guidare il robot verso l'uscita di una singola cella del labirinto per ogni iterazione.

- **Strategia *Flee***: Questa strategia prevede che il robot utilizzi Dijkstra per calcolare lo spostamento verso un'uscita e spostarsi verso di essa, due passi alla volta.

3.1 Dichiarazione e Costruttore

```
private Graph<Box> graph;
private ArrayList<Integer> previousNodes;
private ArrayList<Integer> distances;
```

- Viene dichiarata la struttura dati per il grafo (`graph`) e per la gestione delle distanze e dei predecessori.

Il costruttore:

```
public Dijkstra(Graph<Box> graph) {
    this.graph = graph;
}
```

- Accetta un grafo orientato e ponderato come input.

3.2 Metodo `calculateShortestPath`

Il metodo calcola il percorso più breve dal nodo sorgente al nodo di destinazione:

```
public ArrayList<Integer> calculateShortestPath(Integer source, Integer dest) {
    int n = graph.getEdge().size();
    distances = new ArrayList<>(Collections.nCopies(n, Integer.MAX_VALUE));
    previousNodes = new ArrayList<>(Collections.nCopies(n, -1));
    distances.set(source, 0);

    PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.comparingInt
    (distances::get));
    pq.add(source);

    while (!pq.isEmpty()) {
```

```
            int current = pq.poll();
            if (current == dest) break;

            for (Edge edge : graph.getEdge().get(current)) {
                int neighbor = edge.getDest();
                int newDist = distances.get(current) + edge.getWeight();
                if (newDist < distances.get(neighbor)) {
                    distances.set(neighbor, newDist);
                    previousNodes.set(neighbor, current);
                    pq.add(neighbor);
                }
            }
        }

        return getShortestPath(dest);
    }
```

- Le distanze vengono inizializzate con `Integer.MAX_VALUE`, a indicare che inizialmente non esiste un percorso noto.
- La coda di priorità (`PriorityQueue`) gestisce i nodi da esplorare in ordine crescente di distanza.
- L'algoritmo termina quando:
  - La coda di priorità è vuota.
  - Il nodo di destinazione è stato raggiunto (`if (current == dest) break;`).

3.3 Metodo `getShortestPath`

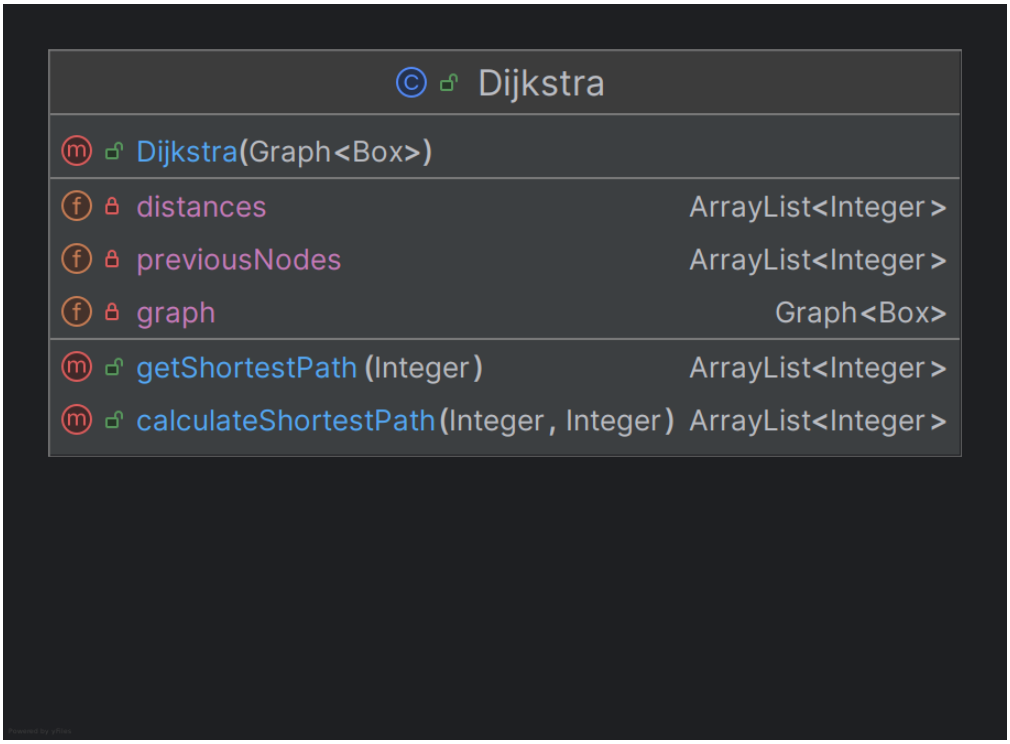
Il metodo ricostruisce il percorso più breve dal nodo sorgente al nodo di destinazione:

```
public ArrayList<Integer> getShortestPath(Integer destination) {
```

```
    ArrayList<Integer> path = new ArrayList<>();
    for (Integer at = destination; at != -1; at = previousNodes.get(at)) {
        path.add(at);
    }
    Collections.reverse(path);
    return path;
}
```

- Ricostruisce il percorso iterando all'indietro attraverso i predecessori.
- Inverte l'ordine del percorso per restituire la sequenza corretta.
- Restituisce un percorso vuoto se il nodo di destinazione non è raggiungibile.

3.4 Struttura UML



1

[1]UML Legend: (f)Fields, (m)constructors/methods , (i)Inner Classes

4 Ruolo e Scopo della Classe `Graph`

La classe `Graph` e le sue derivazioni, `GraphMaze` e `Edge` , svolgono un ruolo fondamentale nell'implementazione dell'algoritmo di Dijkstra. Esse forniscono un modello astratto e generico per rappresentare un grafo orientato e ponderato, necessario per rappresentare la struttura del labirinto in cui si muove il robot.

Scopo Generale

- La classe `Graph` definisce le basi per la gestione di un grafo:
  - **Nodo**: rappresentato da un identificatore (`Integer`) e da un tipo generico (`T`), che in questo caso è la classe `Box`.
  - **Arco**: rappresentato dalla classe `Edge`, che descrive la relazione tra due nodi con un peso associato.
- Il grafo è rappresentato come una `Map`, dove ogni nodo è associato a una lista di archi (`ArrayList<Edge>`), rendendo efficiente l'accesso ai nodi adiacenti.
- La classe `GraphMaze` estende `Graph` e include funzionalità specifiche per generare il grafo a partire dalla struttura del labirinto. Ogni nodo del grafo corrisponde a una cella valida del labirinto.

Utilizzo nella Logica del Progetto `GraphMaze` fornisce una rappresentazione naturale del labirinto come grafo:

- Le celle non murate del labirinto diventano nodi del grafo.
- Gli archi tra i nodi rappresentano possibili movimenti, con un peso uniforme o personalizzabile (in questo caso 1, per movimenti equidistanti).

Questa struttura consente di applicare direttamente l'algoritmo di Dijkstra per determinare il percorso più breve tra due celle del labirinto.

- La classe astratta `Graph` garantisce flessibilità, mentre `GraphMaze` aggiunge la logica specifica per trasformare il labirinto in un grafo.

5 Ruolo e Scopo della Classe `Interactors`

La classe `Interactors` e le sue componenti forniscono gli elementi fondamentali per rappresentare e gestire il labirinto e le sue caratteristiche. Queste classi definiscono le entità di base utilizzate nel progetto, tra cui caselle, posizioni, e valori associati, consentendo di modellare la struttura del labirinto e i suoi attributi in modo chiaro e organizzato.

`Box`: Rappresenta una singola casella del labirinto, caratterizzata da un valore (`ValueBox`), una posizione (`Position`), e un identificatore unico (`id`) utilizzato per l'integrazione con il grafo. `Position`: Definisce una posizione nel labirinto tramite coordinate `x` e `y`, fornendo i metodi necessari per gestire e accedere a queste coordinate. `ValueBox`: Enumerazione che rappresenta i tipi di caselle nel labirinto, inclusi `WALL` (muro), `EMPTY` (vuoto), e i colori (`RED`, `GREEN`, `YELLOW`, `CYAN`) che influenzano il comportamento del microrobot. `Hardships`: Enumerazione che definisce i livelli di difficoltà del labirinto (`EASY`, `MEDIUM`, `HARD`).

6 Design pattern

In questa sezione, analizzeremo i design pattern utilizzati nel progetto. Per ogni design pattern, presenteremo in maniera generica il suo scopo, la motivazione dietro la sua implementazione e l'applicabilità nel contesto del progetto oltre alla struttura UML. Le informazioni saranno estrapolate dal testo "Design Patterns: Elements of Reusable Object-Oriented Software" [1].

I design pattern utilizzati nel nostro progetto includono:

- Factory Method
- Observer
- State
- Strategy
- Proxy

FACTORY METHOD:

Scopo:

Lo scopo del pattern Factory Method è quello di definire un'interfaccia per la creazione di oggetti in una classe madre, consentendo alle sottoclassi di decidere quale tipo di oggetto creare. Quando un oggetto deve essere istanziato, la responsabilità di creare l'istanza viene delegata alle sottoclassi, che forniscono implementazioni specifiche del metodo factory.

Motivazione:

Il pattern Factory Method è utile quando si desidera delegare la creazione di oggetti a sottoclassi, consentendo una maggiore flessibilità e estensibilità del codice. Spesso, nelle applicazioni, si ha la necessità di creare diverse varianti di oggetti senza conoscere in anticipo il tipo specifico di oggetto necessario. Utilizzando il Factory Method, è possibile separare il

codice che crea gli oggetti dal codice che li utilizza, promuovendo una migliore modularità e facilitando l'aggiunta di nuovi tipi di oggetti senza modificare il codice esistente.

Applicabilità nel progetto:

Il Factory Method Pattern è stato adattato nel progetto per gestire la creazione di labirinti con diverse configurazioni di difficoltà (*facile*, *media*, e *difficile*), separando la logica di creazione delle istanze dal loro utilizzo. Questo approccio è stato scelto per aumentare la flessibilità del sistema e garantire un basso accoppiamento tra le classi.

Nel dettaglio:

La classe astratta `MazeCreator` fornisce un'interfaccia comune per la creazione dei labirinti tramite il metodo `createMaze()`, che rappresenta il Factory Method. La sottoclasse concreta `MazeDifficulty` implementa il Factory Method per restituire istanze specifiche di labirinti (`EasyMaze`, `MediumMaze`, `HardMaze`), che variano in base alla difficoltà richiesta. Questo design consente di centralizzare la logica di creazione, mantenendo il codice dei client (classi che utilizzano i labirinti) indipendente dai dettagli delle implementazioni specifiche.

Grazie a questo pattern, il sistema è altamente estensibile. Per aggiungere nuove difficoltà di labirinto, è sufficiente creare una nuova classe che estenda `Maze` e modificarne il Factory Method, senza necessità di alterare il codice esistente. Questo approccio promuove la riusabilità del codice e semplifica la manutenzione.

Analisi del codice:

Nel dettaglio troviamo :
`IMaze` (Interfaccia):

Definisce due metodi principali: `generateMaze()` per generare la struttura del labirinto e `createGraph()` per costruire il grafo associato al labirinto.

`Maze` (Classe astratta):

La classe astratta `Maze` rappresenta il concetto generale di un labirinto e implementa l'interfaccia `IMaze`, fornendo funzionalità comuni per tutte le sue sottoclassi.
Utilizza una matrice di oggetti `Box` per rappresentare le celle del labirinto. Ogni cella può essere configurata come vuota (`EMPTY`) o personalizzata in base ai requisiti delle sottoclassi.

Il costruttore della classe accetta la dimensione del labirinto come parametro e inizializza: La matrice con celle vuote (`EMPTY`). La dimensione (`dim`) del labirinto. La posizione casuale dell'uscita (`exitMaze`), generata lungo un bordo predefinito. Un grafo (`emptyBox`) che rappresenta la struttura del labirinto senza muri, utile per algoritmi di navigazione.

Ci sono poi i metodi per ottenere informazioni sulle celle, come:
`getBox(int x, int y)`: restituisce la cella in una posizione specifica. `getBoxById(int id)`: cerca una cella tramite il suo identificatore univoco nel grafo. `getGraphMaze()`: converte la matrice del labirinto in un grafo di `Box`, eliminando i muri. `createGraph()`: costruisce il grafo associato al labirinto, permettendo l'integrazione con algoritmi di navigazione come Dijkstra. `getMaze(), getDim()` : che restituiscono rispettivamente il labirinto e la sua dimensione. `generateExit()`: che genera l'uscita del labirinto, con `getExitMaze()` che la restituisce.

`MazeCreator` (Classe astratta):

Fornisce l'interfaccia per il Factory Method tramite il metodo astratto `createMaze()`. Permette di delegare alle sottoclassi la creazione dei diversi tipi di labirinto.

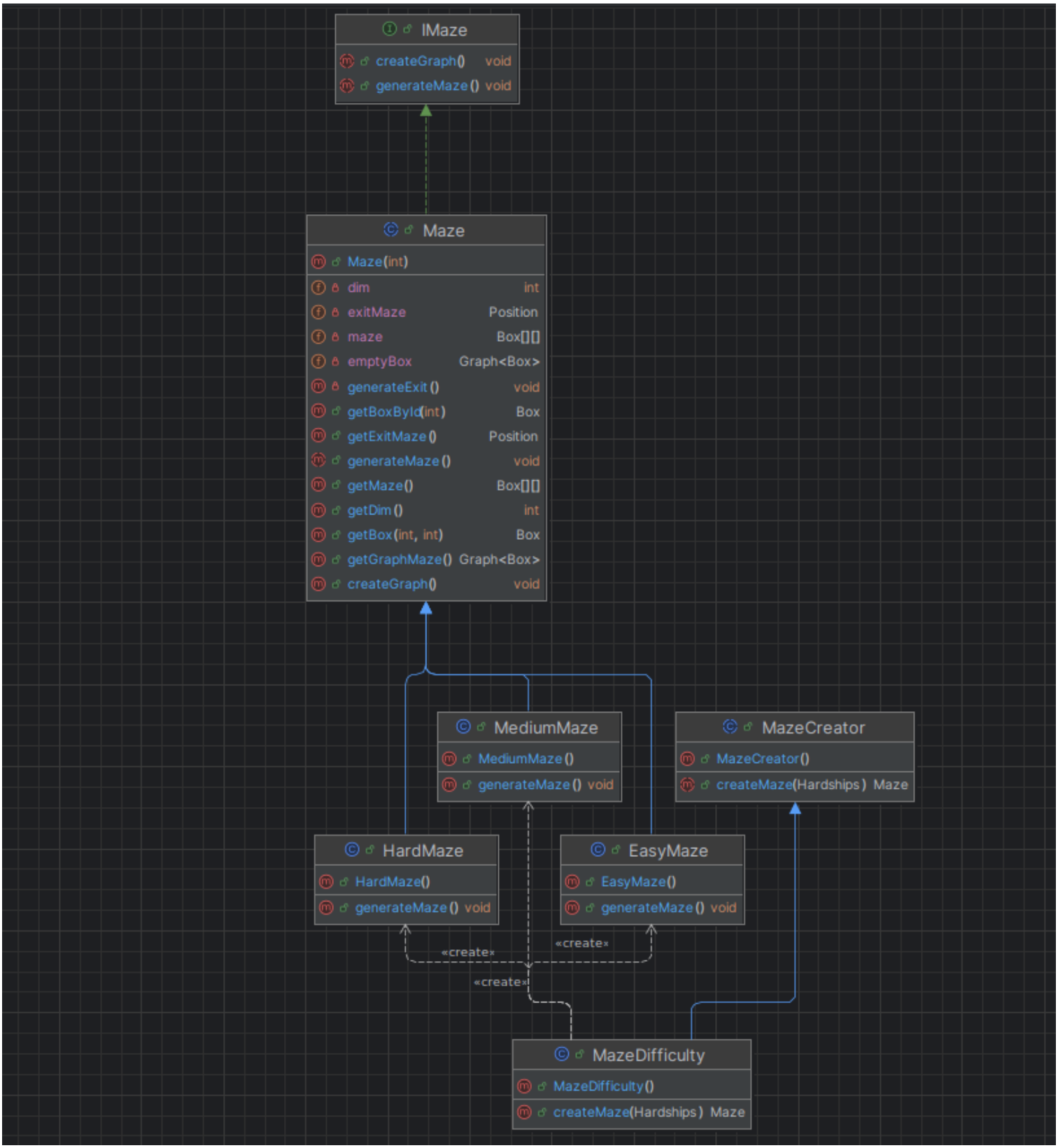
`MazeDifficulty` (Classe concreta):

Implementa il Factory Method `createMaze()` per restituire istanze di `EasyMaze`, `MediumMaze`, o `HardMaze`, in base alla difficoltà richiesta. Ogni istanza viene generata con il metodo `generateMaze()` per configurare la struttura interna e `createGraph()` per costruire il grafo associato.

`EasyMaze, MediumMaze, HardMaze` (Sottoclassi di `Maze`):

Ogni classe fornisce un'implementazione specifica del metodo `generateMaze()` per creare labirinti con diverse configurazioni: `EasyMaze`: Labirinto di dimensione 8×8. `MediumMaze`: Labirinto di dimensione 10×10. `HardMaze`: Labirinto di dimensione 12×12. Utilizzano la matrice del labirinto per definire la posizione di muri e adattando le difficoltà.

Struttura UML:



OBSERVER:

Scopo:

Lo scopo del pattern Observer è quello di definire una dipendenza uno-a-molti tra oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti dipendenti da esso vengono notificati e aggiornati automaticamente

Motivazione:

Il pattern Observer è utile quando si desidera garantire la coerenza tra oggetti dipendenti all'interno di un sistema senza accoppiamento eccessivo tra di essi. Spesso, nelle applicazioni, ci sono oggetti che devono essere aggiornati quando un altro oggetto cambia il suo stato. Senza l'Observer pattern, potremmo dover implementare un sistema di notifiche manuali, il che può portare a codice duplicato e complicato. Utilizzando l'Observer pattern, gli oggetti osservatori possono registrarsi per ricevere notifiche da un oggetto osservato, eliminando la necessità di un'interfaccia diretta tra gli oggetti.

In sintesi, il pattern Observer consente di stabilire un meccanismo di comunicazione tra oggetti in modo flessibile e disaccoppiato, permettendo agli osservatori di essere informati e aggiornati sui cambiamenti degli oggetti osservati in modo efficiente e coerente.

Applicabilità nel progetto:

Il Pattern Observer è stato utilizzato nel progetto per implementare un sistema di notifica che consente a diversi componenti di essere aggiornati automaticamente quando cambia lo stato del gioco. Questo approccio è stato adottato per gestire in modo modulare l'interazione tra il *Game* (il soggetto osservabile) e i suoi osservatori (*PositionSub*). Nel contesto del progetto:

La classe `Game` funge da *Observable* e si occupa di notificare gli osservatori quando avvengono modifiche rilevanti, come il movimento del microrobot o l'aggiornamento del labirinto. Gli osservatori, implementati tramite l'interfaccia `PositionSub`, ricevono aggiornamenti sullo stato del labirinto, la posizione del microrobot e il suo stato attuale (*Pursuit, Seek, Flee, Evade*). La classe `UpdateGame`, che rappresenta un *ConcreteSubscriber*, riceve queste notifiche e memorizza informazioni aggiornate sul gioco, rendendole disponibili per altre parti del sistema.

Analisi del codice:

`Observable` (Interfaccia):

Definisce i metodi `subscribe(PositionSub observer)` e `notifyObservers()` per gestire l'aggiunta degli osservatori e la loro notifica.

`PositionSub` (Interfaccia):

Rappresenta il contratto per gli osservatori. Fornisce il metodo `update()`, che viene chiamato dal soggetto per notificare cambiamenti nello stato del gioco, tra cui: Posizione del microrobot. Stato attuale del labirinto. Dimensione del labirinto. Stato del microrobot (*Pursuit, Seek, Flee, Evade*).

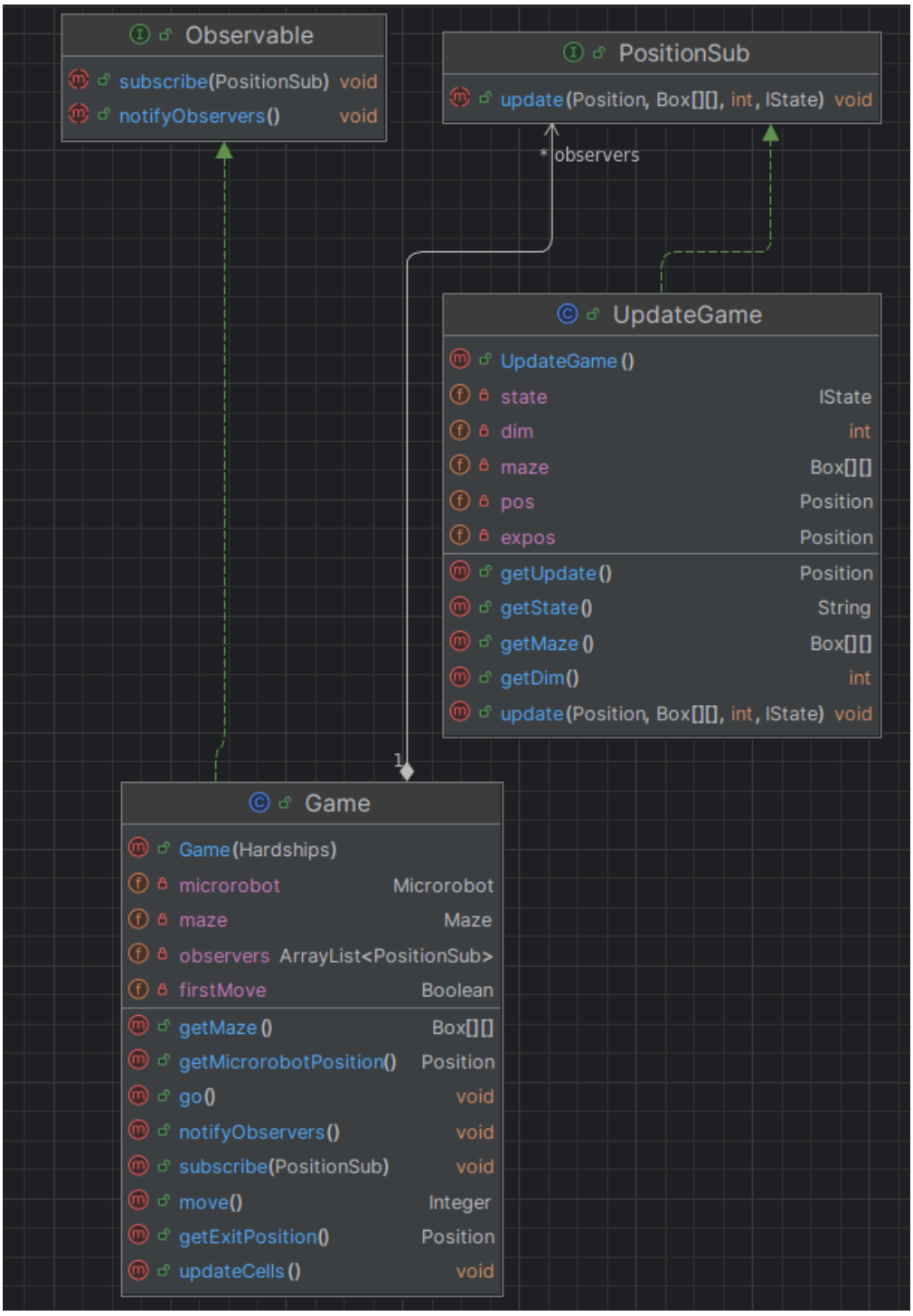
`Game` (Classe concreta):

Funziona da *Observable* e mantiene una lista di osservatori (`ArrayList<PositionSub>`). Gestisce gli aggiornamenti del microrobot e del labirinto, notificando gli osservatori tramite `notifyObservers()`. Permette di iscrivere nuovi osservatori tramite il metodo `subscribe()`. Esegue le logiche di gioco, come il movimento del microrobot e l'aggiornamento delle celle, integrandosi con il sistema di notifica.

`UpdateGame` (ConcreteSubscriber):

Implementa l'interfaccia `PositionSub` e memorizza lo stato attuale del gioco (posizione e stato del microrobot, configurazione del labirinto). Fornisce metodi per calcolare lo spostamento del microrobot (`getUpdate()`) e restituire lo stato del labirinto (`getMaze()`, `getDim()`). Determina lo stato corrente del microrobot, distinguendo tra le diverse strategie.

Struttura UML:



STATE:

Scopo:

Il pattern State ha lo scopo di consentire a un oggetto di modificare il suo comportamento quando il suo stato interno cambia. Questo consente all'oggetto di apparire come se stesse cambiando la sua classe.

Motivazione:

Il pattern State è utile quando un oggetto deve modificare il suo comportamento in base al suo stato interno e quando ci sono diverse combinazioni di stati e azioni che devono essere gestite in modo pulito e flessibile. Piuttosto che utilizzare condizioni complesse per controllare il comportamento dell'oggetto in base allo stato, il pattern State separa il comportamento in classi separate per ciascuno stato. Questo semplifica il codice e rende più chiaro il design dell'applicazione. Inoltre, il pattern State favorisce il principio di singola responsabilità, poiché ogni stato è responsabile del proprio comportamento in risposta agli eventi.

Applicabilità nel progetto:

Il State Pattern è stato utilizzato nel progetto per gestire i diversi stati del microrobot e il comportamento dinamico che essi determinano. Il microrobot può assumere quattro stati principali: *Pursuit*, *Evade*, *Flee*, e *Seek*, ciascuno dei quali è implementato come una classe separata. Questo design consente al microrobot di cambiare il proprio comportamento in base al contesto del labirinto, seguendo lo schema di una Macchina a Stati Finiti (FSM), come richiesto esplicitamente dalla traccia.

Analisi del codice:

`IState` (Interfaccia):

Definisce il metodo `action(Box box)` che calcola la prossima mossa del microrobot sulla base dello stato corrente. Fornisce un'interfaccia comune per tutte le implementazioni concrete degli stati.

`Microrobot` (Classe):

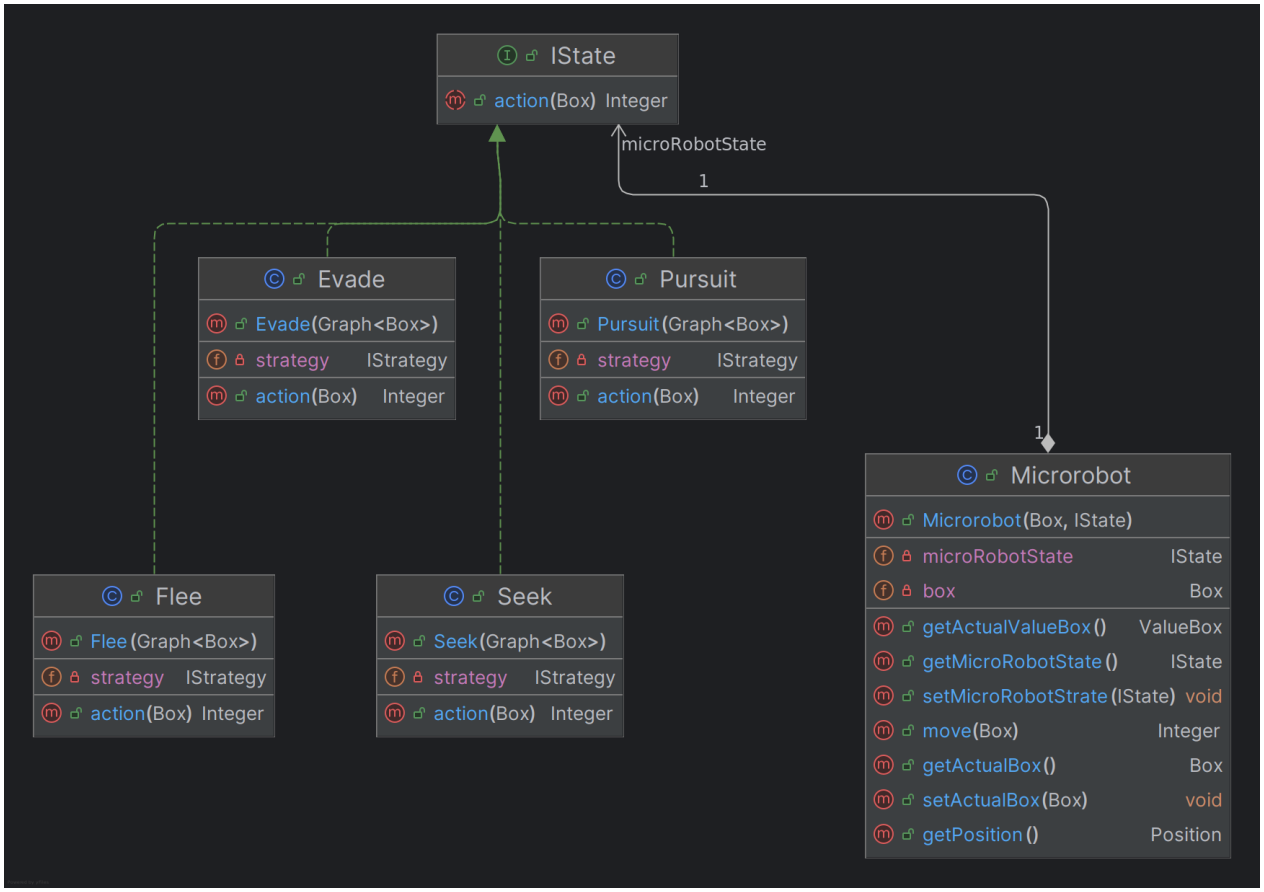
Rappresenta il microrobot all'interno del labirinto. Mantiene una variabile di stato (`IState microRobotState`) che rappresenta lo stato attuale del microrobot. Implementa il metodo `move(Box box)`, che delega allo stato corrente il calcolo della prossima mossa tramite il metodo `action()`. Consente di cambiare dinamicamente lo stato del microrobot

tramite il metodo `setMicroRobotStrate(IState newState)`.

Classi Concrete di Stato (`Pursuit`, `Evade`, `Flee`, `Seek`):

Ogni classe implementa `IState` e rappresenta uno specifico comportamento del microrobot: `Pursuit` e `Seek`: Utilizzano la strategia `OneMove`, basata su Dijkstra, per calcolare la prossima mossa verso la cella adiacente più vicina all'uscita. `Flee`: Utilizza la strategia `TwoMove`, basata su Dijkstra, per spostarsi di due celle alla volta lungo il cammino minimo. `Evade`: Utilizza la strategia `RandomMove` per selezionare casualmente una cella adiacente e garantire un comportamento imprevedibile.

Struttura UML:



STRATEGY:

Scopo:

Il pattern Strategy ha lo scopo di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Questo permette al cliente di selezionare dinamicamente l'algoritmo appropriato senza modificare la struttura degli oggetti coinvolti.

Motivazione:

Il pattern Strategy è utile quando si desidera che un oggetto possa cambiare il suo comportamento in base a determinati criteri o situazioni. Utilizzando il pattern Strategy, è possibile isolare l'implementazione specifica dell'algoritmo dall'oggetto che lo utilizza. Questo promuove la flessibilità del codice, in quanto permette di modificare o estendere il comportamento dell'oggetto senza dover modificare direttamente la sua implementazione.

Applicabilità nel progetto:

Il Pattern Strategy è stato utilizzato nel progetto per implementare diverse strategie di movimento del microrobot all'interno del labirinto, come espressamente richiesto dalla traccia. Questo approccio permette di definire un comportamento flessibile e intercambiabile, delegando la logica di calcolo della prossima mossa a classi specifiche.

Nel contesto del progetto:

L'interfaccia `IStrategy` definisce un metodo comune `nextMove(Box currentBox)` per calcolare la prossima mossa del microrobot. Le implementazioni concrete (`OneMove`, `TwoMove`, e `RandomMove`) rappresentano strategie di movimento specifiche: `OneMove`: Il microrobot si sposta verso la cella successiva lungo il cammino minimo calcolato tramite Dijkstra. `TwoMove`: Il microrobot avanza di due celle alla volta lungo il cammino minimo, sempre utilizzando Dijkstra. `RandomMove`: Il microrobot sceglie casualmente una delle celle adiacenti.

Analisi del codice:

`IStrategy` (Interfaccia):

Definisce il metodo `nextMove(Box currentBox)` per calcolare la prossima mossa del microrobot. Rappresenta un contratto comune per tutte le strategie di movimento.

`OneMove` (Classe concreta):

Implementa una strategia basata sull'algoritmo di Dijkstra: Calcola il percorso più breve dalla posizione corrente del microrobot alla cella di uscita. Se il percorso esiste, restituisce

l'ID della cella adiacente immediata lungo il cammino minimo. Garantisce un movimento preciso e deterministico verso l'uscita.

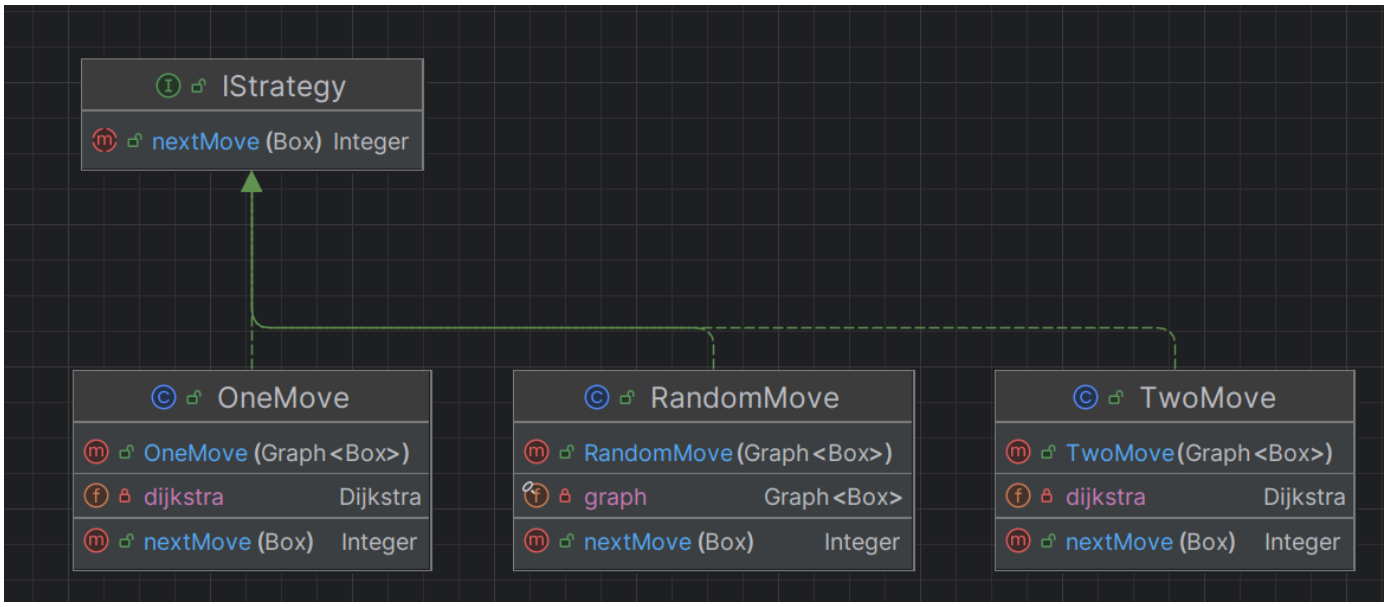
`TwoMove` (Classe concreta):

Simile a `OneMove`, ma consente al microrobot di avanzare di due celle alla volta lungo il cammino minimo. Verifica che il percorso contenga almeno due celle successive, altrimenti si comporta come `OneMove`.

`RandomMove` (Classe concreta):

Implementa una strategia di movimento casuale: Ottiene la lista dei nodi adiacenti alla posizione attuale del microrobot. Sceglie casualmente una cella tra le adiacenti. Aggiunge un elemento di imprevedibilità, rendendo il comportamento del microrobot non deterministico.

Struttura UML:



PROXY :

Scopo:

Il pattern Proxy fornisce un intermediario per controllare l'accesso a un oggetto reale, aggiungendo un livello di astrazione che permette di gestire operazioni come la creazione, l'autenticazione, il caricamento ritardato (lazy loading) o la gestione delle risorse. Questo approccio è particolarmente utile in scenari in cui l'accesso diretto all'oggetto reale potrebbe essere inefficiente o inappropriato, consentendo di implementare logiche aggiuntive senza modificare l'oggetto originale.

Motivazione:

Il pattern Proxy è motivato dall'esigenza di migliorare il controllo e l'efficienza nell'accesso agli oggetti. Questo approccio consente di ottimizzare le risorse, proteggere l'oggetto da accessi inappropriati e rendere più flessibile l'interazione con il sistema.

Applicabilità nel progetto:

Il Proxy Pattern è stato implementato nel progetto per gestire l'accesso al file della classifica, fornendo un livello di controllo e astrazione per le operazioni di lettura e scrittura. La classe `Classification` funge da proxy, implementando l'interfaccia `IFile` e gestendo direttamente le operazioni sui file. Questo approccio garantisce che le operazioni di lettura e scrittura siano eseguite in modo controllato, separando la logica dell'applicazione dal livello di gestione dei file.

Analisi del codice:

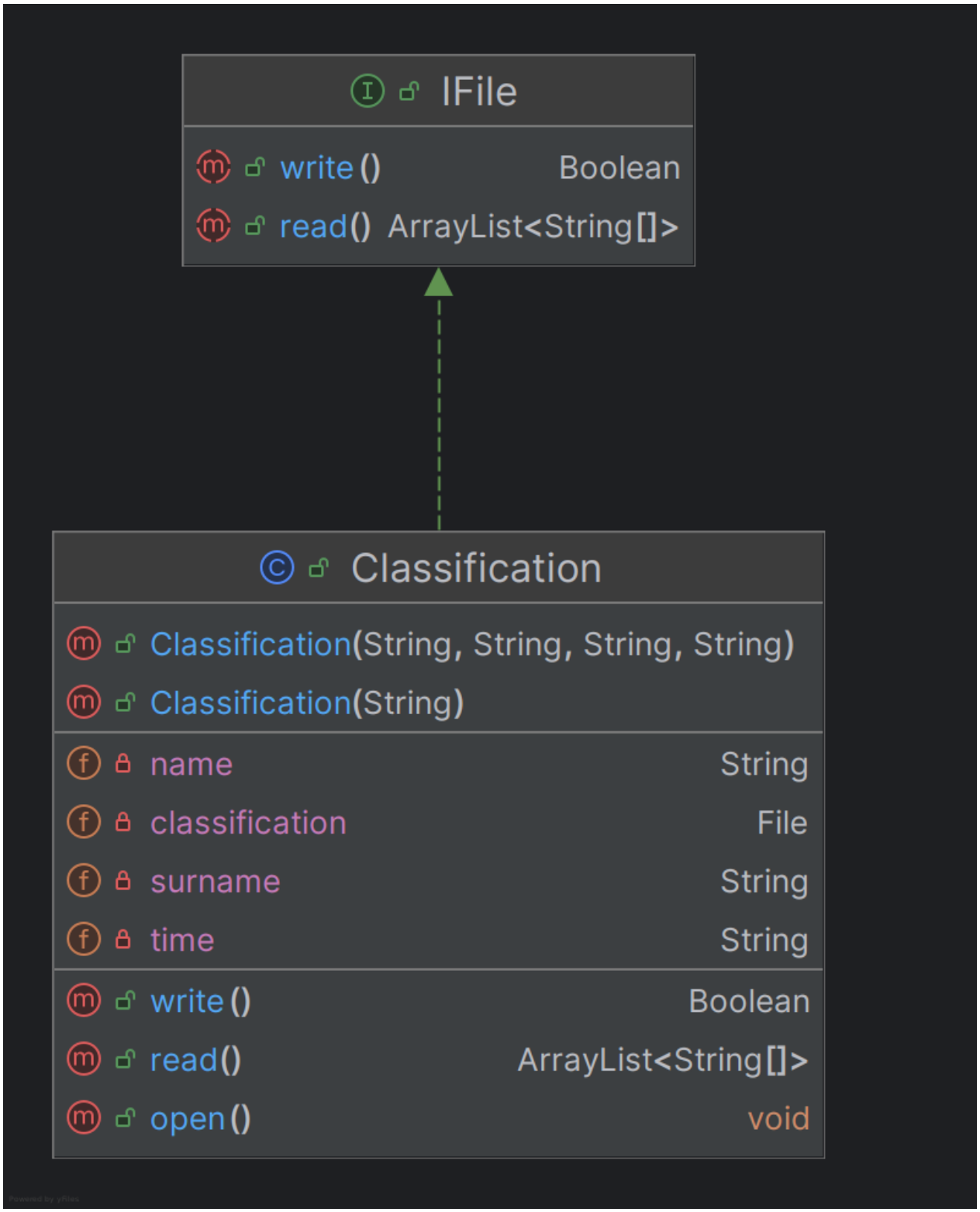
`IFile` (Interfaccia):

Definisce i metodi `write()` e `read()` per le operazioni sui file. Fornisce un contratto comune che può essere implementato da diversi tipi di proxy, garantendo flessibilità ed estensibilità.

`Classification` (Classe concreta):

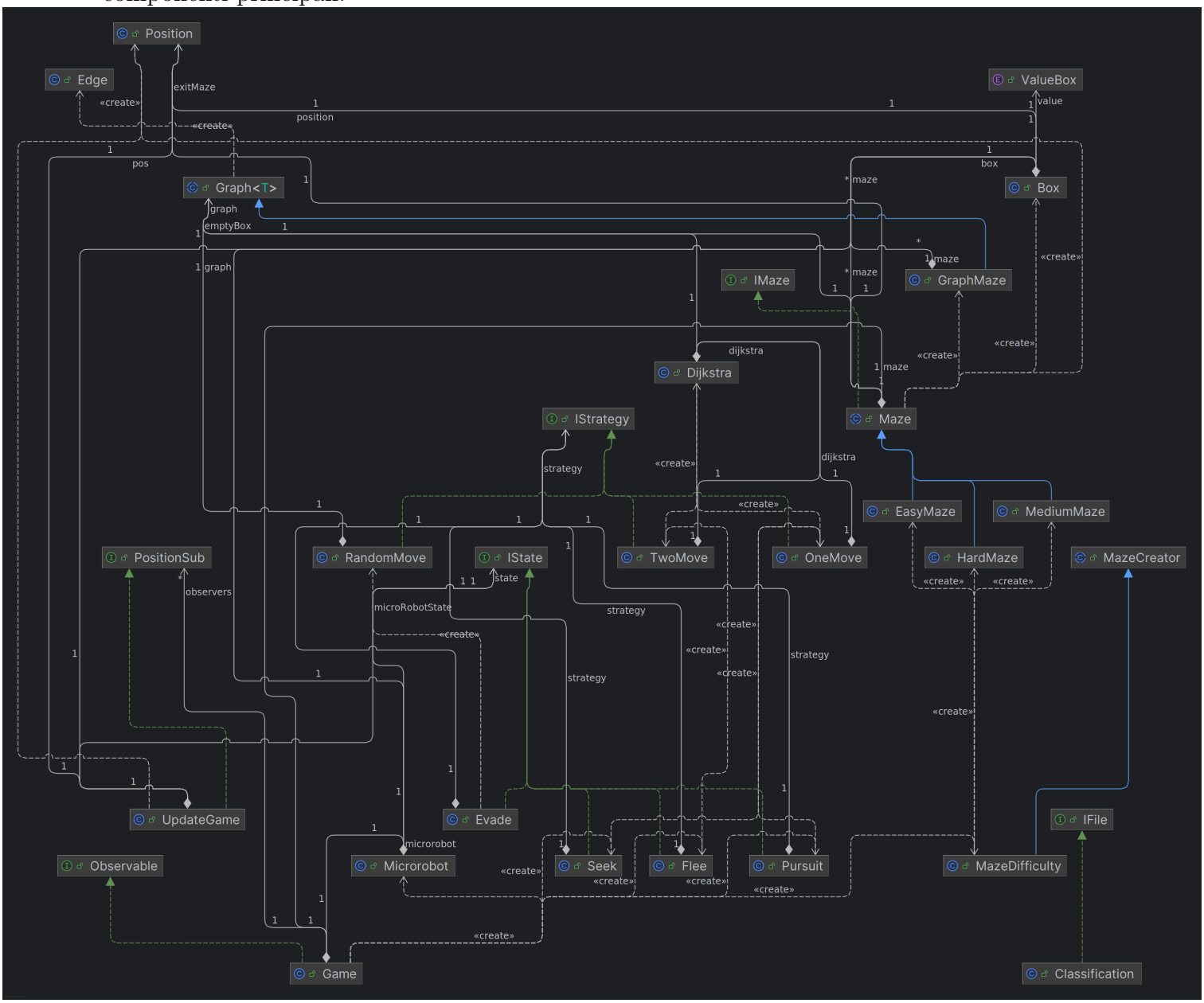
Implementa l'interfaccia `IFile` e funge da proxy per il file della classifica. Funzionalità principali: **Scrittura del file** (`write()`): Scrive il nome, cognome e tempo del giocatore nella classifica, mantenendo l'ordine basato sui tempi migliori. **Lettura del file** (`read()`): Legge il contenuto della classifica e lo restituisce come una lista di stringhe. **Apertura del file** (`open()`): Verifica se il file esiste e lo crea se necessario.

Struttura UML:



7 UML

Dopo aver analizzato tutte le classi che costituiscono il progetto, inclusi i design pattern adottati (Proxy, State, Strategy, Observer), presentiamo il diagramma UML generale. Questo diagramma offre una visione d'insieme della struttura del sistema e delle relazioni tra le sue componenti principali.



2

[2]Per una visione migliore si rimanda al file maze.png

8 Analisi Principi SOLID

I principi SOLID sono un insieme di linee guida e concetti di progettazione orientati agli oggetti, formulati per promuovere la creazione di codice sostenibile, flessibile e manutenibile. Questi principi, introdotti da Robert C. Martin nel suo libro "Design Principles and Design Patterns", sono diventati uno standard per gli sviluppatori di software che mirano a produrre codice di alta qualità e modularità.

Ora procederemo a un'analisi del progetto in questione per valutare se rispetta i principi SOLID, questo può aiutare a valutare la sua architettura e la sua qualità complessiva, garantendo che il software sia ben strutturato, flessibile e manutenibile nel tempo:

1. Principio di Singola Responsabilità (Single Responsibility Principle - SRP):

Il principio SRP afferma che una classe dovrebbe avere una sola ragione per cambiare. In altre parole, una classe dovrebbe essere responsabile di un'unica parte del funzionamento del sistema. Questo implica che la classe dovrebbe gestire un solo aspetto del comportamento del software.

2. Principio di Apertura/Chiusura (Open/Closed Principle - OCP):

Il principio OCP stabilisce che le entità software (classi, moduli, funzioni, ecc.) dovrebbero essere aperte all'estensione ma chiuse alla modifica. Ciò significa che il comportamento di un'entità dovrebbe essere estendibile senza la necessità di modificarne il codice sorgente originale.

3. Principio di Sostituzione di Liskov (Liskov Substitution Principle - LSP):

Il principio LSP afferma che gli oggetti di un sottotipo dovrebbero poter essere sostituiti con oggetti della loro classe base senza interrompere l'applicazione. Questo implica che i sottotipi dovrebbero estendere il comportamento dei loro tipi base senza modificarne il comportamento.

4. Principio di Segregazione delle Interfacce (Interface Segregation Principle - ISP):

Il principio ISP suggerisce che "i client non dovrebbero essere costretti a dipendere da interfacce che non utilizzano". In altre parole, le interfacce dovrebbero essere specifiche per i client che le utilizzano, evitando di costringere i client ad implementare metodi che non necessitano.

5. Principio di Inversione delle Dipendenze (Dependency Inversion Principle - DIP):

Il principio DIP indica che i moduli di alto livello non dovrebbero dipendere dai moduli di basso livello, ma entrambi dovrebbero dipendere da astrazioni. Le astrazioni non dovrebbero dipendere dai dettagli, ma i dettagli dovrebbero dipendere dalle astrazioni.

L'analisi del nostro progetto rivela una solida aderenza ai principi SOLID, resa possibile dall'implementazione attenta di 5 design pattern (Proxy, State, Strategy, Observer, Factory) tra i 23 delineati dalla Gang of Four i quali costituiscono l'ossatura strutturale del sistema.

Principio di Singola Responsabilità (SRP): Ogni classe nel progetto è stata progettata con una responsabilità ben definita. Ad esempio, la classe `Box` gestisce esclusivamente le proprietà delle caselle del labirinto (valore, posizione e identificatore), mentre la classe `Microrobot` è responsabile dello stato e del movimento del robot. Questo approccio assicura che le modifiche richieste in un aspetto specifico del sistema non impattino altre parti del

codice.

Principio di Apertura/Chiusura (OCP): Il progetto è altamente estendibile senza richiedere modifiche al codice esistente. Ad esempio, le strategie di movimento del robot, implementate tramite il pattern Strategy, consentono di aggiungere nuove logiche di movimento semplicemente creando una nuova classe che implementa l'interfaccia `IStrategy`. Analogamente, i diversi stati del microrobot seguono questo principio grazie al pattern State.

Principio di Sostituzione di Liskov (LSP): Le classi che estendono interfacce o classi base rispettano pienamente il comportamento previsto. Ad esempio, tutte le strategie di movimento (`OneMove`, `TwoMove`, `RandomMove`) possono essere utilizzate in modo intercambiabile tramite l'interfaccia `IStrategy`, senza introdurre comportamenti inattesi. Lo stesso vale per gli stati (`Pursuit`, `Seek`, `Flee`, `Evade`), che estendono `IState` garantendo coerenza nell'interazione con il `Microrobot`.

Principio di Segregazione delle Interfacce (ISP): Le interfacce sono state progettate per essere specifiche e granulari. Ad esempio, `IFile` definisce esclusivamente i metodi necessari per la gestione del file della classifica (`write()` e `read()`), senza imporre metodi superflui alle classi concrete come `Classification`. Lo stesso vale per l'interfaccia `IState`, che fornisce un singolo metodo `action()` per gestire il comportamento degli stati del microrobot.

Principio di Inversione delle Dipendenze (DIP): Il progetto è costruito attorno ad astrazioni, riducendo la dipendenza diretta tra moduli di alto e basso livello. Ad esempio, la classe `Microrobot` dipende dall'interfaccia `IState` piuttosto che dalle implementazioni concrete degli stati. Analogamente, la gestione dei file avviene tramite l'interfaccia `IFile`, isolando le dipendenze specifiche come `Classification`. Questo design garantisce flessibilità e semplifica l'integrazione di nuove funzionalità.

In sintesi, l'architettura del progetto, guidata dall'adozione dei design pattern e da una chiara attenzione ai principi SOLID, ha prodotto un codice estensibile e facilmente manutenibile.

9 Interfaccia Grafica

L'interfaccia grafica del nostro progetto è stata realizzata utilizzando JavaFX, un framework software per la creazione di interfacce utente grafiche (GUI) e applicazioni web in Java. JavaFX offre un insieme di librerie e strumenti che semplificano lo sviluppo di applicazioni desktop, consentendo la creazione di interfacce utente moderne e interattive con facilità.

Inoltre, l'ambiente di sviluppo integrato (IDE) scelto per il nostro progetto è IntelliJ IDEA. IntelliJ è un potente IDE sviluppato da JetBrains, progettato per migliorare la produttività degli sviluppatori Java e supportare una vasta gamma di tecnologie, inclusi JavaFX, Maven, Gradle e molti altri. Grazie alle sue funzionalità avanzate di sviluppo, debugging e refactoring, IntelliJ IDEA ha facilitato il processo di sviluppo e ha contribuito alla realizzazione efficiente dell'interfaccia grafica del nostro progetto.

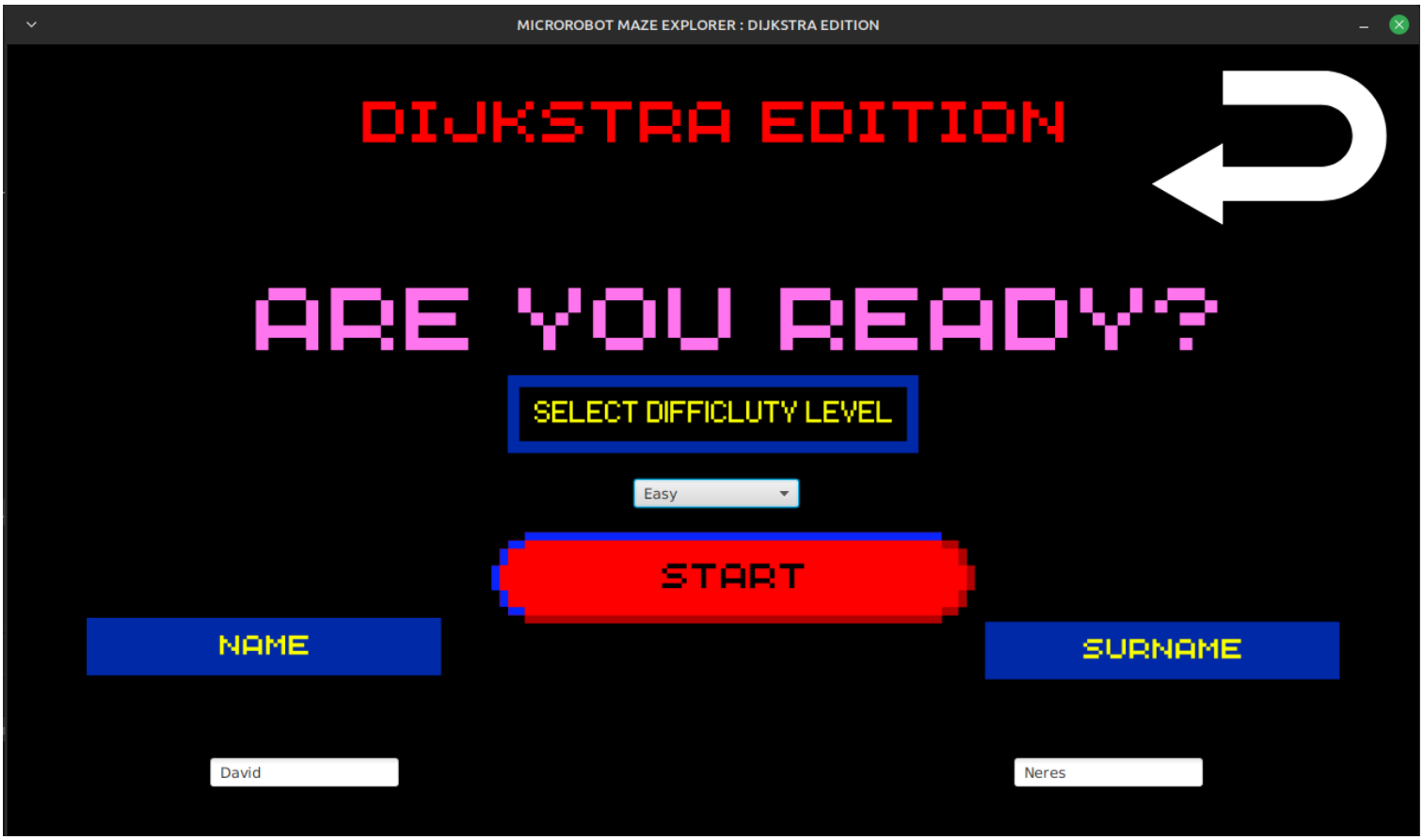
HomePage

La pagina di accesso al gioco presenta una disposizione simile alle schermate home dei più celebri giochi, con un'enfasi particolare sui due bottoni principali: "Play".

In alto a sinistra è posizionato il pulsante di aiuto, che reindirizza alla pagina GitHub del progetto. In basso a destra, si trova il pulsante "Quit" che consente di chiudere direttamente la schermata. Per quanto riguarda le scelte di design, il tema principale si ispira ai giochi retro tipici del Gameboy, con uno sfondo scuro che accentua l'importanza delle scritte e dei bottoni.

Il nome scelto, "Microrobot Maze Explorer: Dijkstra Edition", è accattivante e riflette le caratteristiche fondamentali del progetto. Esso suggerisce che il gioco fa parte di una saga immaginaria dedicata all'esplorazione di labirinti da parte di microrobot, con l'aggiunta dell'edizione "Dijkstra" per indicare la particolare enfasi sull'intelligenza artificiale utilizzata, differenziandosi così dagli altri algoritmi.

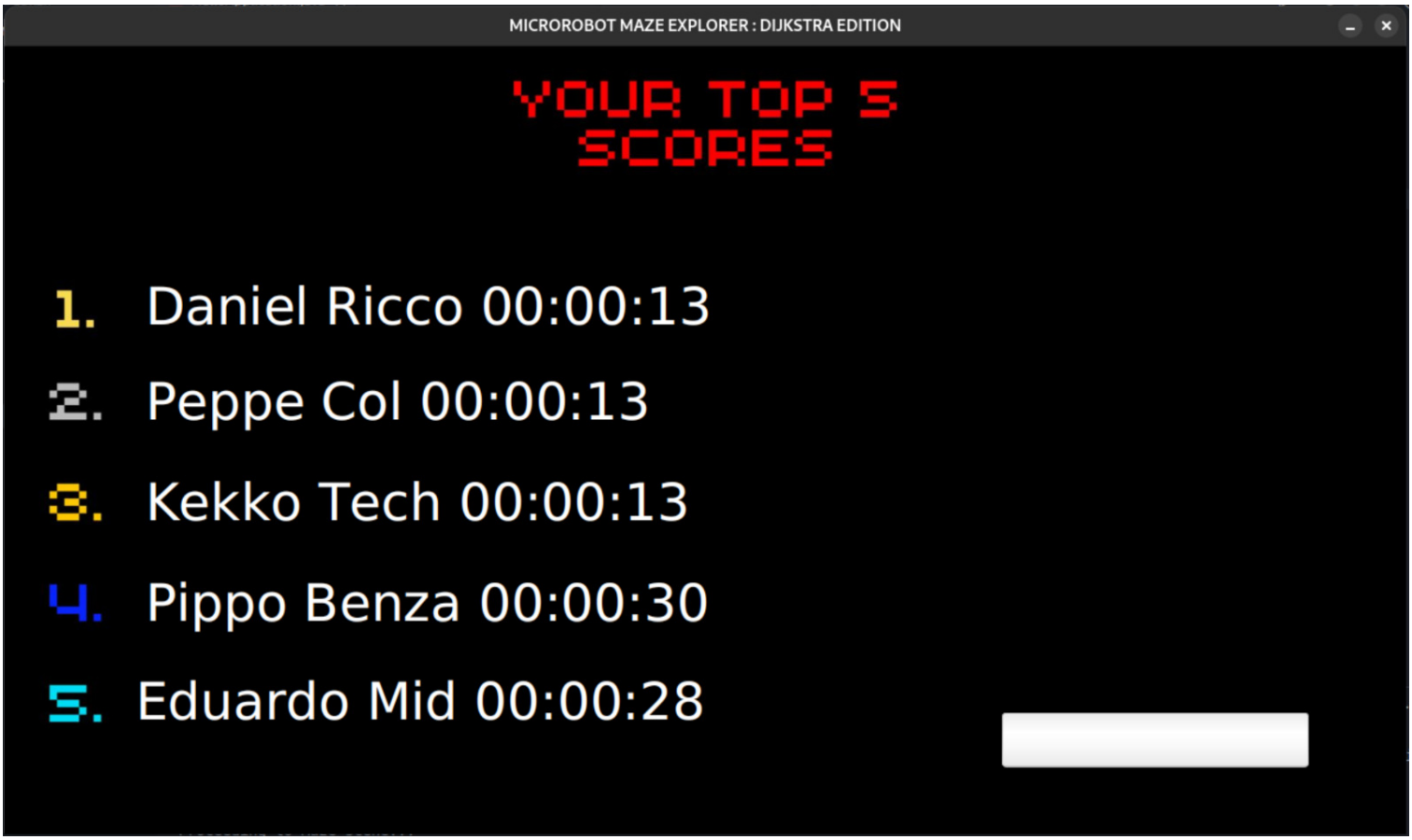
UserAndPlayPage



Dopo aver premuto il pulsante "Play" nella pagina precedente, si accede alla schermata pre-gioco dove l'utente può personalizzare le impostazioni e inserire i propri dati. Innanzitutto, l'utente deve selezionare la difficoltà di gioco tra tre livelli: easy, medium e hard, che influenzano la dimensione del labirinto e il numero di muri e la loro disposizione.

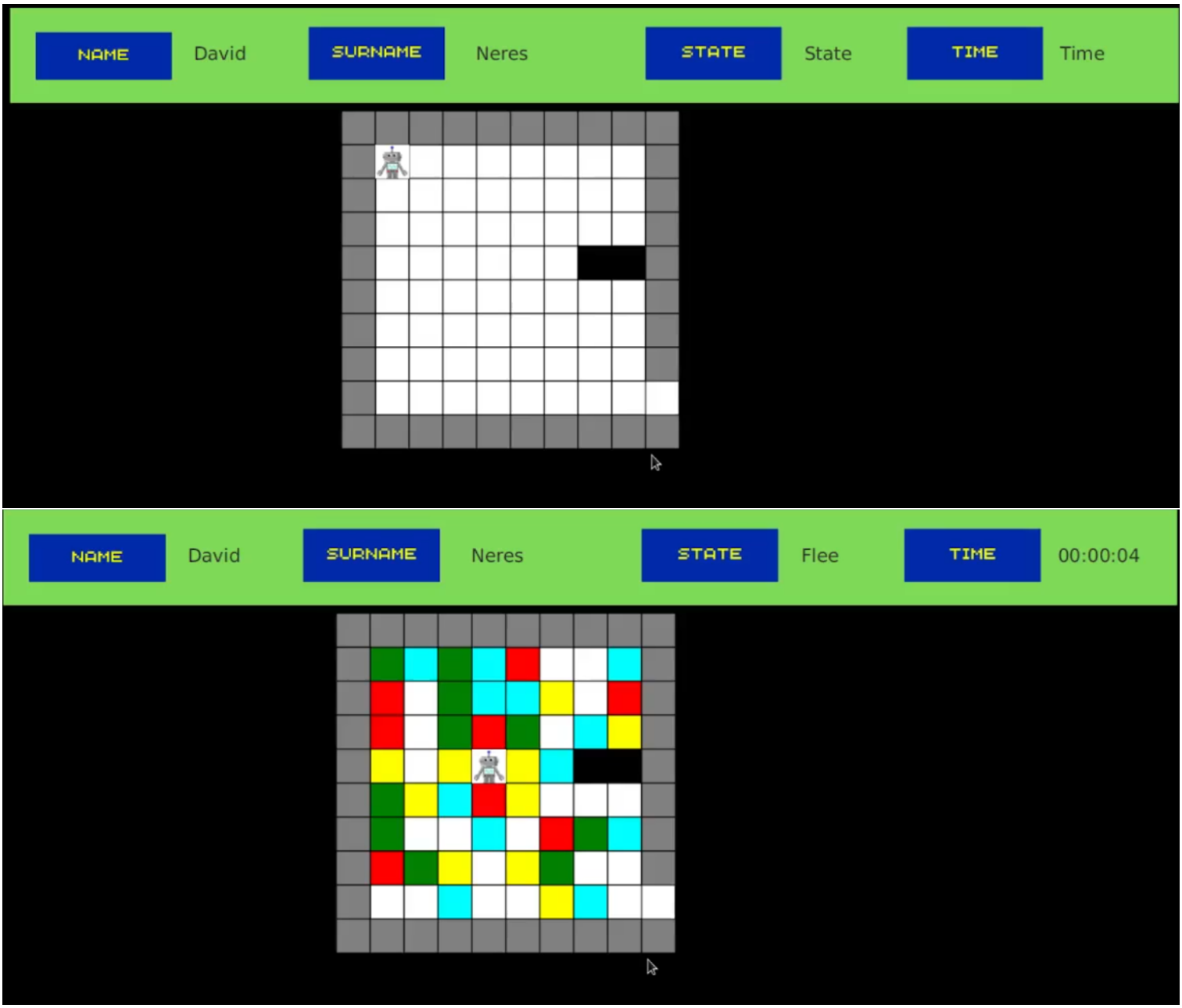
In seguito, l'utente è tenuto ad inserire i suoi dati personali, tra cui nome e cognome, i quali sono obbligatori affinché il pulsante "Play" diventi cliccabile. Una volta completati questi passaggi, il giocatore sarà pronto per iniziare la partita.

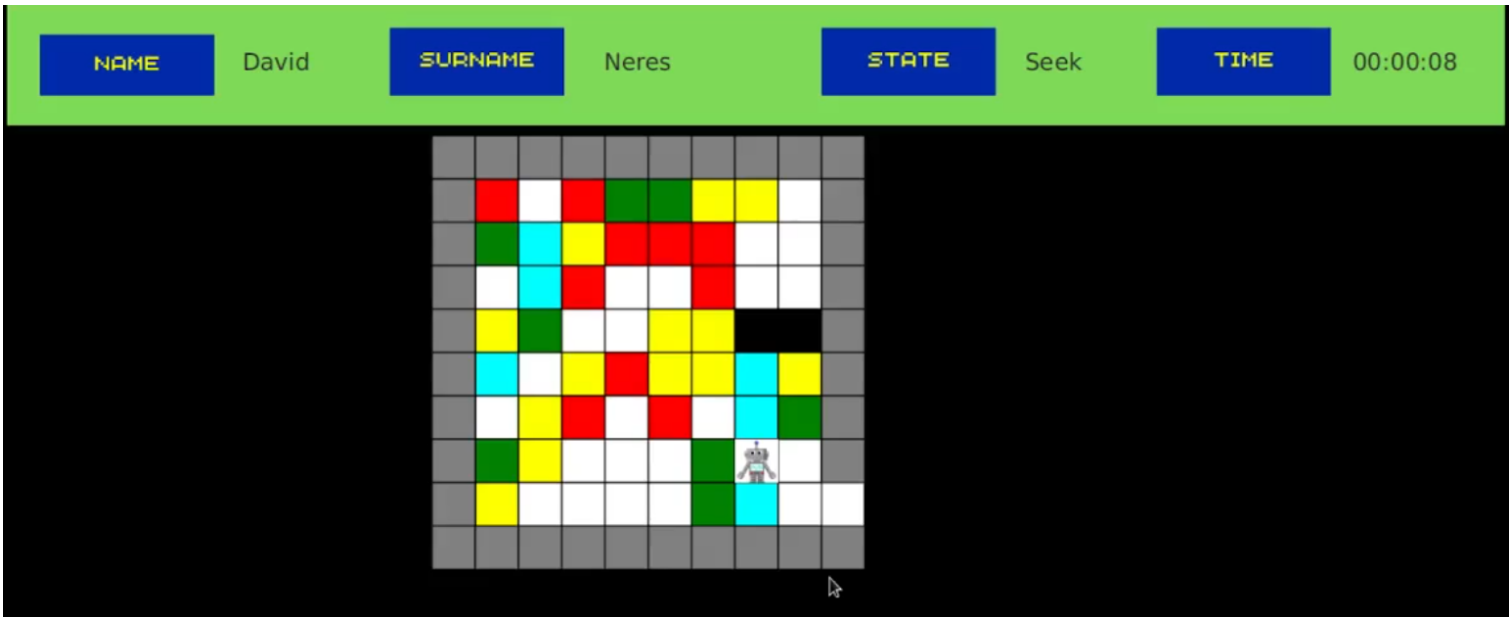
RankingLoadPage



La pagina `RankingAndLoad` offre una panoramica dei 5 migliori tempi attuali per il livello di difficoltà selezionato. Contemporaneamente, viene avviata una barra di caricamento che segnala il progresso nella preparazione al gioco, creando un'anticipazione per l'inizio della partita.

Let's Start


NAME
David
SURNAME
Neres
STATE
State
TIME
Time
NAME
David
SURNAME
Neres
STATE
Flee
TIME
00:00:04



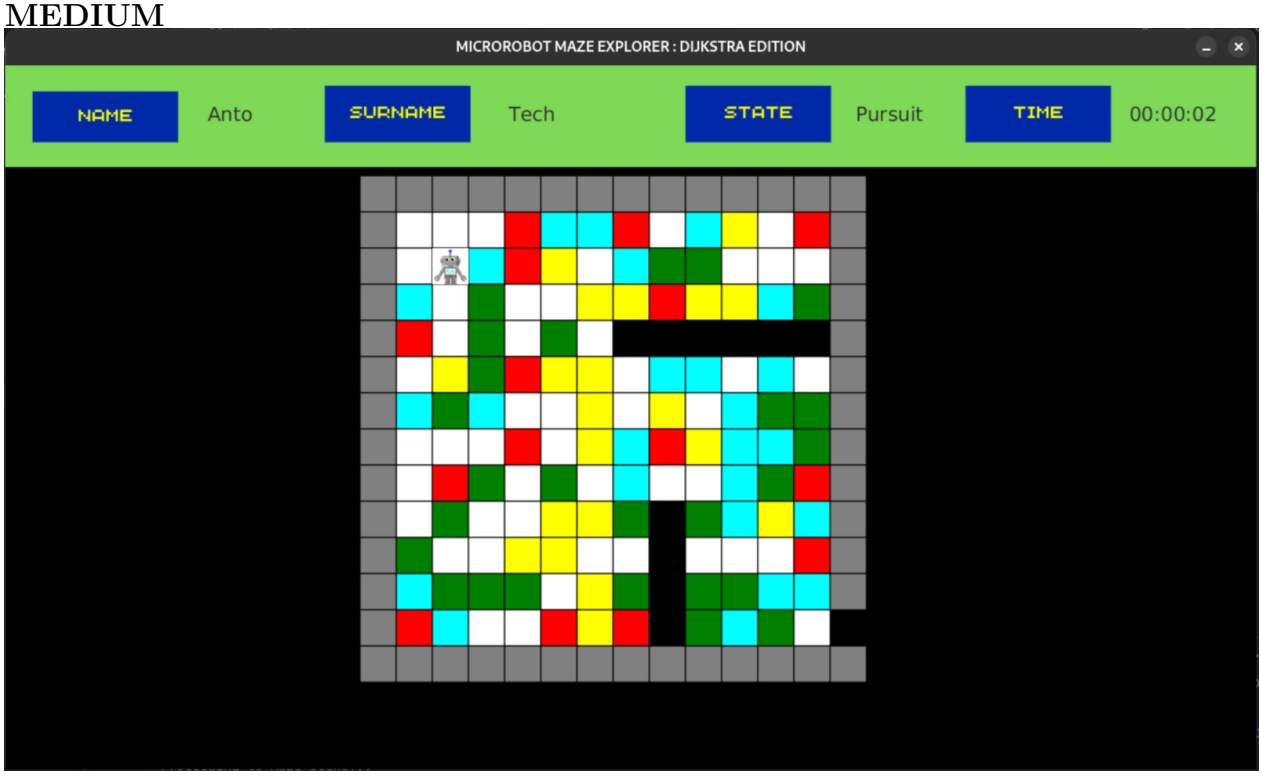
Il nostro utente, *David Neres*, è pronto a mettersi alla prova per scalare le classifiche! Il labirinto, all'apparenza semplice, si trasforma rapidamente: bastano pochi secondi per veder comparire numerosi ostacoli e caselle colorate che obbligano il microrobot a rivedere la sua strategia.

Grazie al supporto dell'amico Dijkstra, il robot non si arrende e continua a muoversi verso l'uscita, adattandosi ai cambiamenti in tempo reale. Lo stato attuale del robot è sempre visibile nella barra superiore, offrendo una panoramica costante della partita.

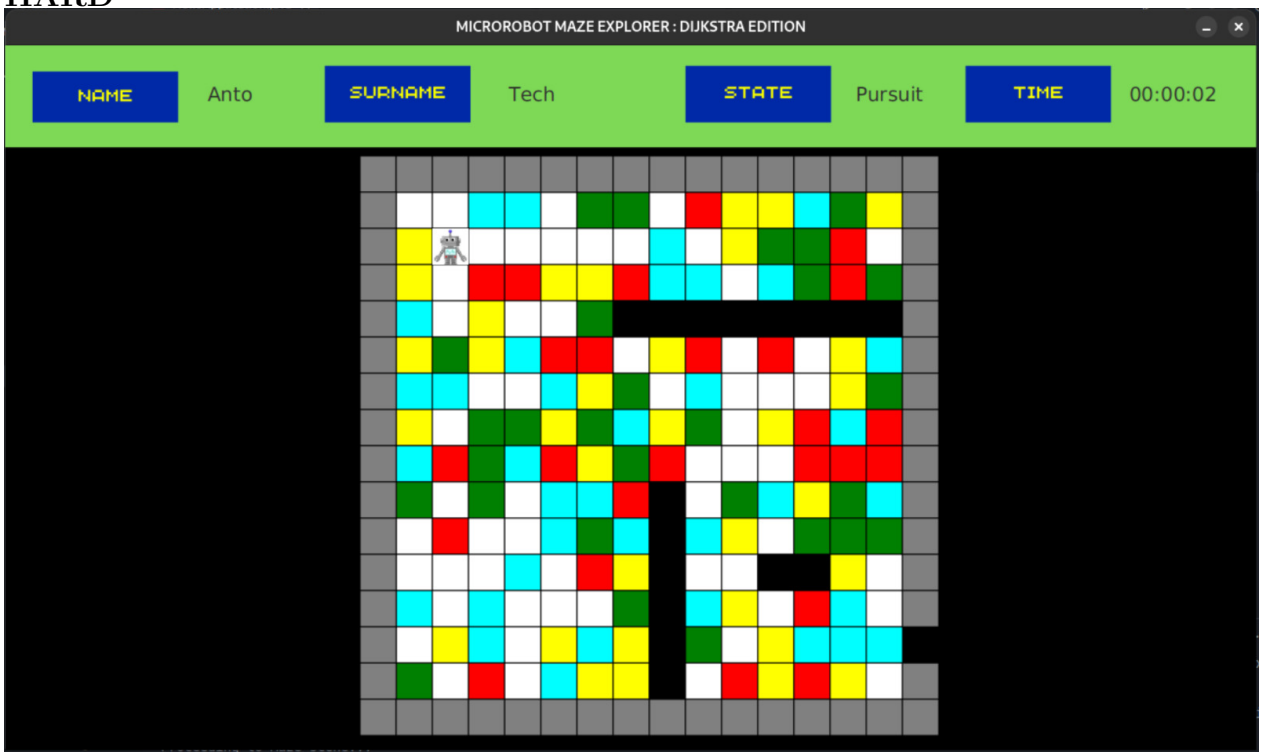
Dopo soli 9 secondi, il microrobot raggiunge l'uscita. Ma sarà riuscito a guadagnare un posto tra i migliori in classifica?

Cosa cambia nelle modalità Medium e Hard ?

MEDIUM



HARD



References

[1] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 978-0201633610.

Definizioni

- UML (Unified Modeling Language) è un linguaggio di modellazione grafica utilizzato principalmente nel campo dell'ingegneria del software per visualizzare, specificare, costruire e documentare artefatti del sistema software.

- I DESIGN PATTERN sono soluzioni progettuali generali a problemi ricorrenti nello sviluppo software. Essi forniscono un approccio standardizzato e testato per risolvere determinati tipi di problemi di progettazione.