PROGRAMMAZIONE III E LABORATORIO DI PROGRAMMAZIONE III

# RELAZIONE PROGETTO

## GIOCO CON INTELLIGENZA ARTIFICIALE

**Fulvio Serao - 0124/002423**
**Luca Pezzolla - 0124/002411**
**Simone Micillo - 0124/002439**

## ***Sommario***

## ***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. 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 assumerne 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 Figura 2.

Per ogni stato le strategie sono:

- *pursuit* e *seek* - va nella direzione dell'uscita di una singola cella alla volta (usare l'algoritmo *Artificial Ant Colony Optimization*, vedi sotto)
- *flee* - va nella direzione dell'uscita di due celle alla volta (usare l'algoritmo *Artificial Ant Colony Optimization*, vedi sotto)
- *evade* - avanza in maniera casuale di una singola cella

Scrivere un programma per la gestione del labirinto. Il programma deve permettere di registrare il robot con il suo *nome* e *cognome* e visualizzare, ad ogni inizio e fine partita, la classifica dei risultati migliori ottenuti, da tutti i robot, in tutte le partite (minore numero di passi per raggiungere l'uscita). Visualizzare il percorso del robot dopo ogni passo, mostrando la stanza, la sua posizione e quella degli oggetti.

## ***Intelligenza Artificiale***

Per il nostro progetto è stato richiesto, per la gestione del movimento del robot, l'utilizzo dell'algoritmo di intelligenza artificiale ***Ant Colony Optimization***, un algoritmo del tipo *Swarm Intelligence.* Esso viene utilizzato per la risoluzione di problemi computazionali complessi. L'idea di base di questo algoritmo prende spunto dall'organizzazione di una colonia di formiche, si usa infatti un meccanismo di feedback positivo come una sorta di feromone virtuale, per rafforzare quelle parti di soluzione che contribuiscono alla risoluzione del problema. Per evitare invece la convergenza verso opzioni non idonee, viene utilizzato un meccanismo di feedback negativo (ad esempio l'evaporazione del feromone virtuale) che introduce una componente temporale nell'algoritmo. Il punto di forza dell'algoritmo consiste quindi nella creazione di un sistema intelligente distribuito.

## ***Implementazione dell'ACO***

L'algoritmo e' stato implementato adattandolo alla risoluzione di un problema diverso rispetto ai campi di utilizzo soliti: deve infatti trovare, in un grafo non pesato, il cammino minimo da un nodo sorgente (la cella di partenza del robot) ad un nodo destinazione (la cella d'arrivo). Per fare cio', c'era bisogno che l'implementazione tenesse conto dell'insieme degli archi del grafo (e quindi di tutte le possibili celle con i loro vicini): a tal proposito, e' tornata utile la struttura dati ***MultiMap***, non presente nei package di Java, e che implementa una Map gestita con la logica dei bag, piuttosto che dei set (in sostanza, una chiave -nodo sorgente- puo' avere piu' valori -nodi destinazione-).

Regina portante ***dell'ACO*** e' la classe ***AntSystem***, che contiene al suo interno uno "screenshot" degli archi del labirinto, ***edges***, una ***Multimap<Integer,Integer>.*** Dopo aver inizializzato la classe, passandole il numero di formiche richiesto e il numero di iterazioni volute (oltre che l'insieme degli archi), il metodo pubblico ***pathCalculator*** si occupa di calcolare un possibile cammino minimo tra il nodo sorgente e il nodo destinazione passati come input. Tale cammino verra' poi restituito sotto forma di ***ArrayList<Integer>***, ove ogni intero al suo interno sara' un l'id di una cella da attraversare per giungere all'uscita.

```java
public ArrayList<Integer> pathCalculator(int src, int dest)
{
    Map<ArrayList<Integer>, Integer> pathCount = new HashMap<>();
    double[][] edgePheroQnt = createTopology();

    int i = 0;
    while(i++ < iterations)
    {
        int ant = 0;
        while(ant++ < ants)
        {
            ArrayList<Integer> tour = antColonyVisit(src, dest, edgePheroQnt);
            if(tour.size() > 1)
                if(pathCount.containsKey(tour))
                {
                    Integer currentValue = pathCount.get(tour);
                    pathCount.replace(tour, currentValue, newValue: currentValue + 1);
                }
                else
                    pathCount.put(tour, 1);
        }
        updatePheroQnt(pathCount, edgePheroQnt);
    }

    return convergedPath(pathCount);
}
```

Le funzioni utilizzate da ***pathCalculator***, dichiarate private, attuano la logica dell'algoritmo:

***createTopology*** -> rappresenta la topologia del labirinto, tramite la creazione di una matrice di size numero_nodi*numero_nodi e contenente sugli archi esistenti la quantita' iniziale di feromone.

***antColonyVisit*** -> fa partire l'iesima formica da un nodo sorgente, sperando arrivi alla destinazione.

***updatePheroQnt*** -> aggiorna i livelli di feromone sugli archi.

***convergedPath*** -> restituisce il cammino con la piu' alta occorrenza tra tutti i cammini computati.

Ad ogni iterazione, quindi, se l'iesima formica ha trovato un cammino fino alla destinazione e quest'ultimo non e' stato trovato, lo si inserisce all'interno di una ***Map<ArrayList<Integer>,Integer>***, che assegna ad ogni cammino una occorrenza; altrimenti, se quest'ultimo gia' esisteva, si procede ad aumentarne di uno la sua occorrenza.

Cuore pulsante dell'algoritmo e' rappresentato dalla scelta dei "vicini" di ogni cella, la quale viene gestita dalle restanti funzioni dell'algoritmo:

***pickUpNeighbour***

```
private Integer pickUpNeighbour(int node, double[][] edgePheroQnt)
{
    ArrayList<Integer> neighs = availableNeighbours(node, edgePheroQnt);
    if(neighs.size() == 0)
        return NO_NEIGHBOUR;

    double probs[] = new double[neighs.size()];
    int index = 0;


    for(int neigh : neighs)
        probs[index++] = probCalculator(node, neigh, edgePheroQnt);

    double value = Math.random();


    double sum = 0;
    for(index = 0; index < neighs.size(); ++index)
    {
        sum += probs[index];
        if(value <= sum)
            break;
    }

    return neighs.size() > 0 ? neighs.get(index) : NO_NEIGHBOUR;
}
```

Tale funzione, tramite l'utilizzo di ***availableNeighbours***, sceglie il prossimo vicino da visitare in base alla probabilita' calcolata in ***probCalculator***:

```
private double probCalculator(int start, int end, double[][] edgePheroQnt) throws IllegalArgumentException
{
    double num = Math.pow(edgePheroQnt[start][end], ALPHA) * Math.pow(1, BETA);
    double denum = 0;
    ArrayList<Integer> neighs = availableNeighbours(start, edgePheroQnt);
    if(neighs.size() == 0)
        throw new IllegalArgumentException("No neighbours");

    for(int neigh : neighs)
        denum += Math.pow(edgePheroQnt[start][neigh], ALPHA) * Math.pow(1, BETA);

    return num / denum;
}
```

questo metodo, in base alla scelta dei parametri ***ALPHA*** e ***BETA***, genera una probabilita' di scelta di un particolare nodo; nello specifico, i parametri ***ALPHA*** e ***BETA*** influenzano il risultato perche' mentre uno rendera' piu' influenti i valori di feromone (***ALPHA***), l'altro dara' piu' importanza agli archi rimanenti.

Nella nostra implementazione, i valori ***ALPHA*** e ***BETA*** sono costanti dal valore di 0.5 cadauno, scelta motivata dal fatto che la somma delle costanti e' preferibile sia 1, e inoltre, nel nostro caso, abbiamo deciso di dare uguale importanza alle scelte possibili per rendere l'algoritmo bilanciato e libero di esplorare ogni possibile scelta gli si presenti davanti, senza condizionamenti "esterni".

Chiude il pacchetto la funzione ***tourLength***, che calcola la lunghezza di un particolare cammino.

Ogni labirinto incapsula al suo interno un oggetto di tipo ***Graph<Cell>***, che si occupa di vedere il labirinto come un insieme di archi (e che quindi contiene un oggetto di tipo ***MultiMap<Integer,Integer>***, costruito insieme alla creazione del labirinto.

C'e' pero' da fare una precisazione, perche' non tutti i nodi (o celle) hanno dei vicini: le celle "wall", infatti, non solo non sono inserite tra i vicini delle altre celle, ma a loro volta non hanno vicini; questa scelta e' stata dovuta poiche' ***l'ACO*** ha una complessita' non indifferente, e scatenarlo su un grafo con una notevole quantita' di archi avrebbe reso l'attesa tra le mosse del robot lunga e tediosa.

# ***I pattern utilizzati***

Ecco, di seguito, l'elenco dei pattern utilizzati:

***Factory Method;***

***Command;***

***Observer;***

***State;***

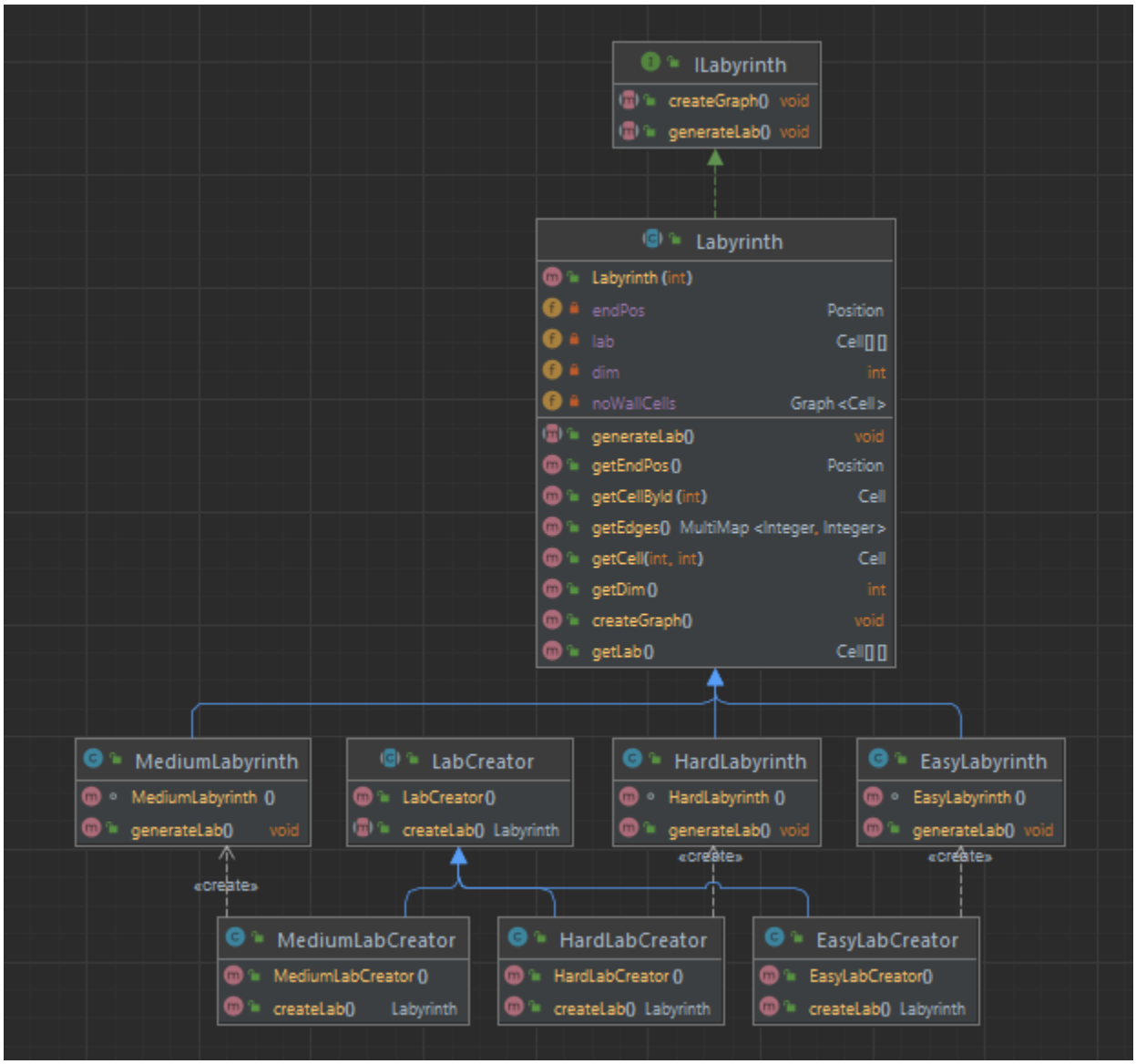
***Strategy;***

***Proxy.***

## **Factory Method:**

Factory Method e' stato implementato utilizzando una classe astratta ***LabCreator***, con al suo interno un unico metodo, astratto, ***createLab***, che viene ridefinito nei ConcreteCreators (***EasyLabCreator/MediumLabCreator/HardLabCreator***), e che in tutti i casi restituisce un labirinto generato (metodo ***generateLab***), insieme con la sua rappresentazione in termini di archi(metodo ***createGraph***).

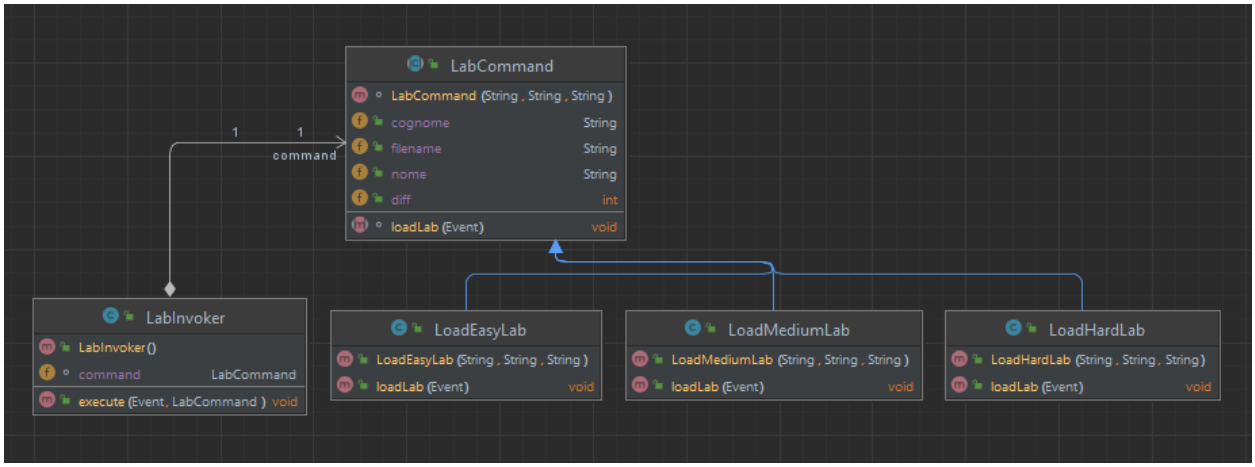
Il prodotto restituito, ovvero il labirinto, e' di tipo Labyrinth: quest'ultima e' una classe astratta, che implementa l'interfaccia ***ILabyrinth***, contenente i metodi precedentemente menzionati. La scelta di definire un prodotto come classe astratta, che implementa questi metodi (generateLab come abstract, in particolare), e' presto detta: si e' infatti voluto raggruppare le caratteristiche comuni di tutti i labirinti in una superclasse, lasciando pero' alle sottoclassi la liberta' di avere una creazione dello scenario differente, a seconda del tipo di labirinto costruito dai ConcreteCreators.


ILabyrinth
createGraph() void
generateLab() void
Labyrinth
Labyrinth (int)
endPos Position
lab Cell[][]
dim int
noWallCells Graph <Cell>
generateLab() void
getEndPos() Position
getCellById (int) Cell
getEdges() MultiMap <Integer, Integer>
getCell(int, int) Cell
getDim() int
createGraph() void
getLab() Cell[][]
MediumLabyrinth
MediumLabyrinth ()
generateLab() void
LabCreator
LabCreator()
createLab() Labyrinth
HardLabyrinth
HardLabyrinth ()
generateLab() void
EasyLabyrinth
EasyLabyrinth ()
generateLab() void
«create»
«create»
«create»
MediumLabCreator
MediumLabCreator ()
createLab() Labyrinth
HardLabCreator
HardLabCreator ()
createLab() Labyrinth
EasyLabCreator
EasyLabCreator()
createLab() Labyrinth

## **Command**:

Command e' stato implementato tramite l'utilizzo di una classe astratta, ***LabCommand***, che contiene al suo interno i parametri che l'invoker dovra' passare al receiver (***LabController***), per mezzo dei Concrete Commands (nome e cognome del player, il nome del file della scoreboard da aprire dopo la partita e la difficolta' del labirinto). L'unico metodo dichiarato all'interno di questa classe e' astratto (***loadLab***), e viene ridefinito nei ConcreteCommands (***LoadEasyLab/LoadMediumLab/LoadHardLab***) al fine di caricare gli scenari fxml realizzati per ogni labirinto.

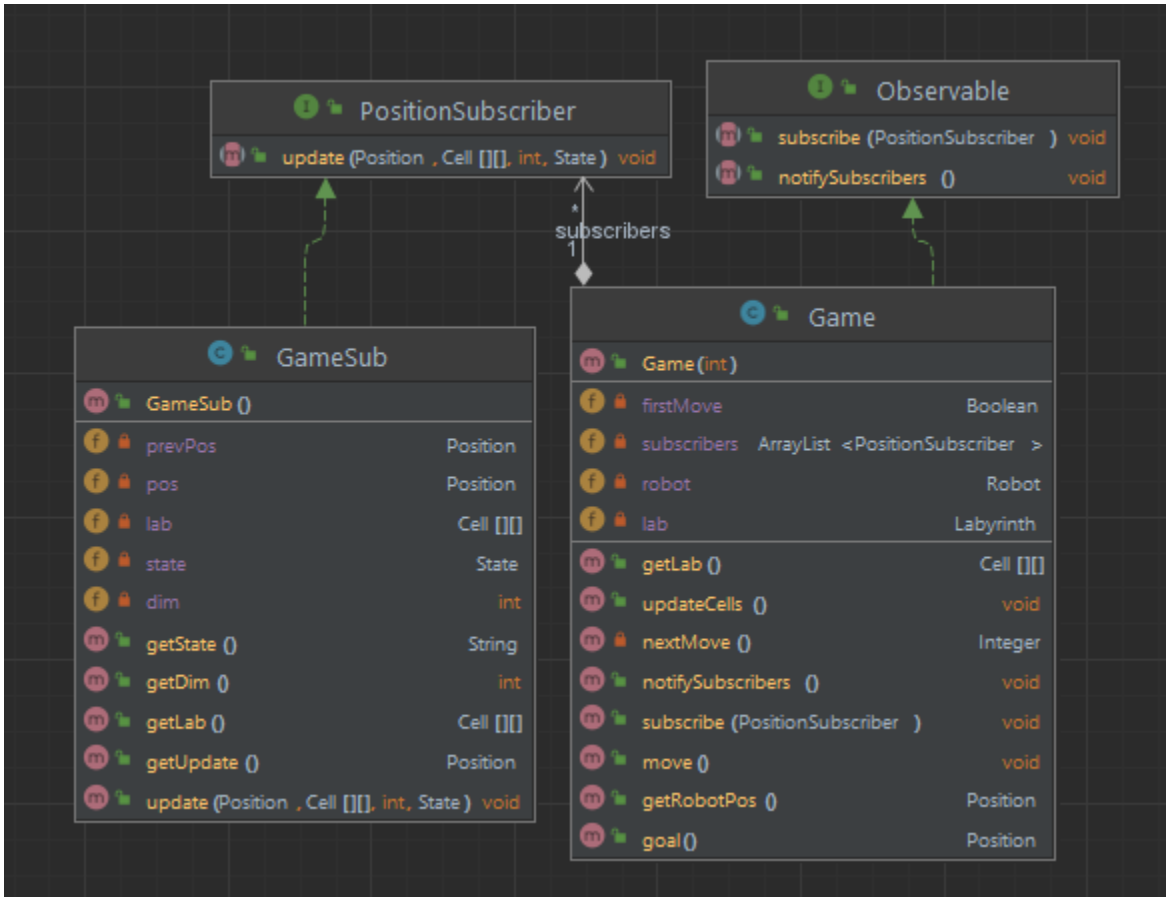
Chiosa, infine, l'invoker (***LabInvoker***), che in base all'input dell'utente (pressione di un bottone piuttosto che di un altro) va a richiamare il metodo loadLab del ConcreteCommand corrispondente, dopo aver memorizzato al suo interno il comando passato come argomento della sua unica funzione, execute.



## **Observer**:

Observer e' stato implementato utilizzando due interfacce, una per i subscriber (***PositionSubscriber***) e una per il publisher (***Observable***): la prima (implementata dalla classe ***GameSub*** e contenente il solo e unico metodo update), permette di aggiornare lo stato corrente del Subscriber in base all'Observable (nel nostro caso, permette alla classe Game di aggiornare GameSub riguardo lo stato del labirinto, la dimensione dello stesso e la posizione/stato del robot).

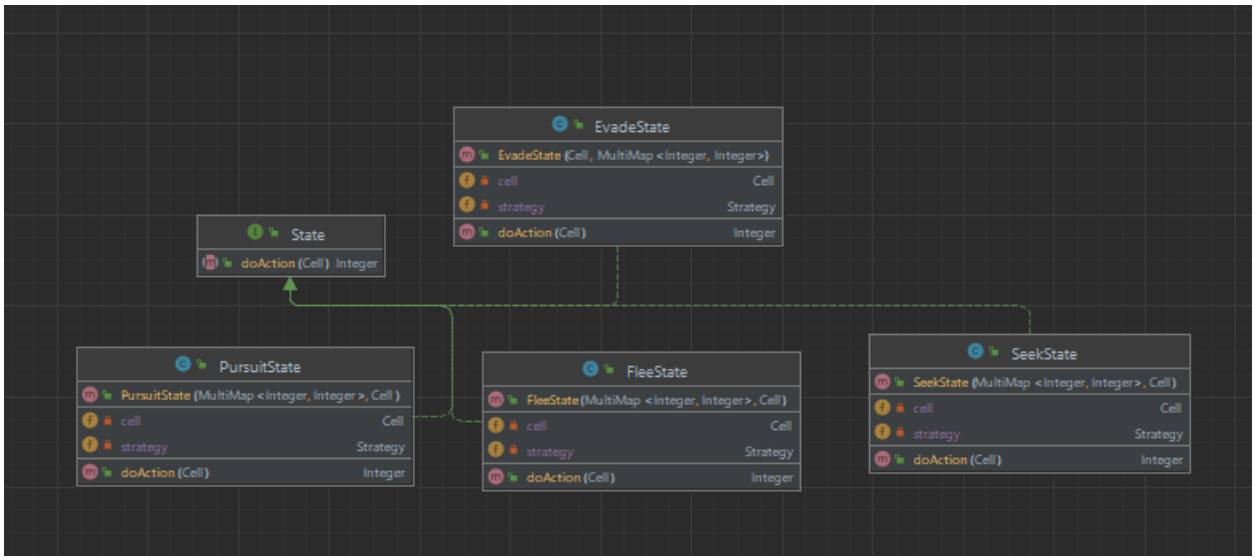
In seconda e ultima analisi, l'interfaccia Observable implementa i metodi ***subscribe*** e ***notifySubscribers***, ridefiniti nella classe game, e che consentono ad un qualsiasi Subscribe di iscriversi alla "newsletter" di Game (venendo aggiunti ***all'ArrayList<PositionSubscriber>*** presente nella classe), e di riceverne gli aggiornamenti (invocando ***l'update***, per ogni singolo Subscriber nell'ArrayList, all'interno della funzione notifySubscribers).



## **State**:

Il pattern State consta di un'interfaccia, ***State***, con un solo metodo: ***doAction***. Quest'ultimo, ridefinito nei ConcreteStates (***EvadeState***, ***PursuitState***, ***FleeState***, ***SeekState***), consente al robot di muoversi in maniera differente in base al suo stato (anche grazie al pattern Strategy, che approfondiremo in seguito.

Il Context, impersonificato dal Robot, si preoccupa semplicemente di immagazzinare al suo interno lo stato corrente in cui si trova e la cella attuale, per poi invocare il metodo move, il quale richiama il doAction dello stato corrente, computando la sua prossima mossa.

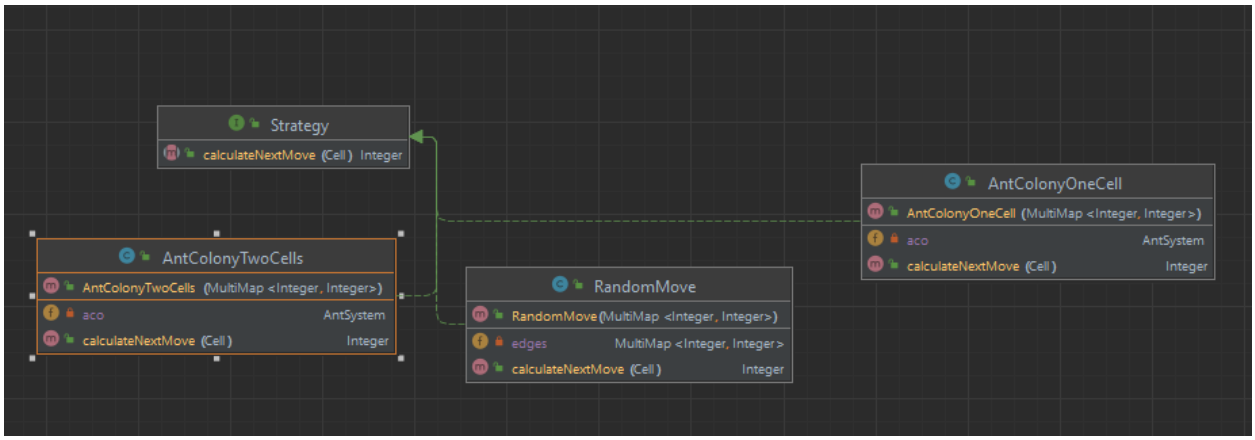


## **Strategy**:

Strategy e' il cuore pulsante del movimento del robot, ed e' stato implementato grazie ad un'interfaccia (Strategy) che contiene un singolo metodo: ***calculateNextMove***. Questo metodo, ridefinito nelle ConcreteStrategies (***AntColonyOneCell***, ***AntColonyTwoCells***, ***RandomMove***) computa la prossima mossa restituendo un intero, che rappresenta l'id della prossima cella nella quale il robot si dovra' spostare.

Tutte le ConcreteStrategies prendono in input l'insieme degli archi del labirinto, e quelle dedicate all'AntColony si preoccupano di ripetere la ricerca del cammino fin quando questo non viene trovato, prima di restituire un risultato (puo' capitare, infatti, che talvolta le formiche non arrivino al nodo destinazione, specialmente nel labirinto di difficolta' hard).

Le ConcreteStrategies sono immagazzinate negli stati del robot, e il metodo doAction non fa altro che richiamare il ***calculateNextMove*** della Strategy contenuta al suo interno.

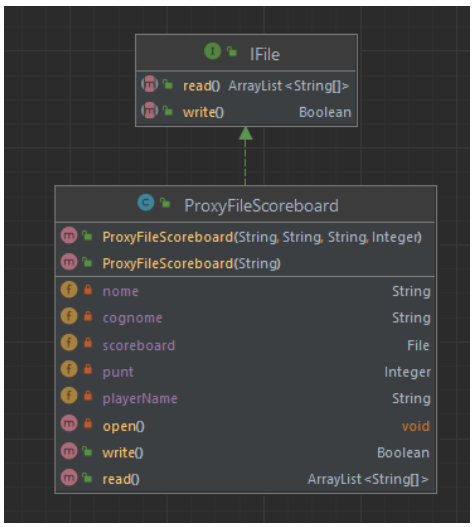


## **Proxy**:

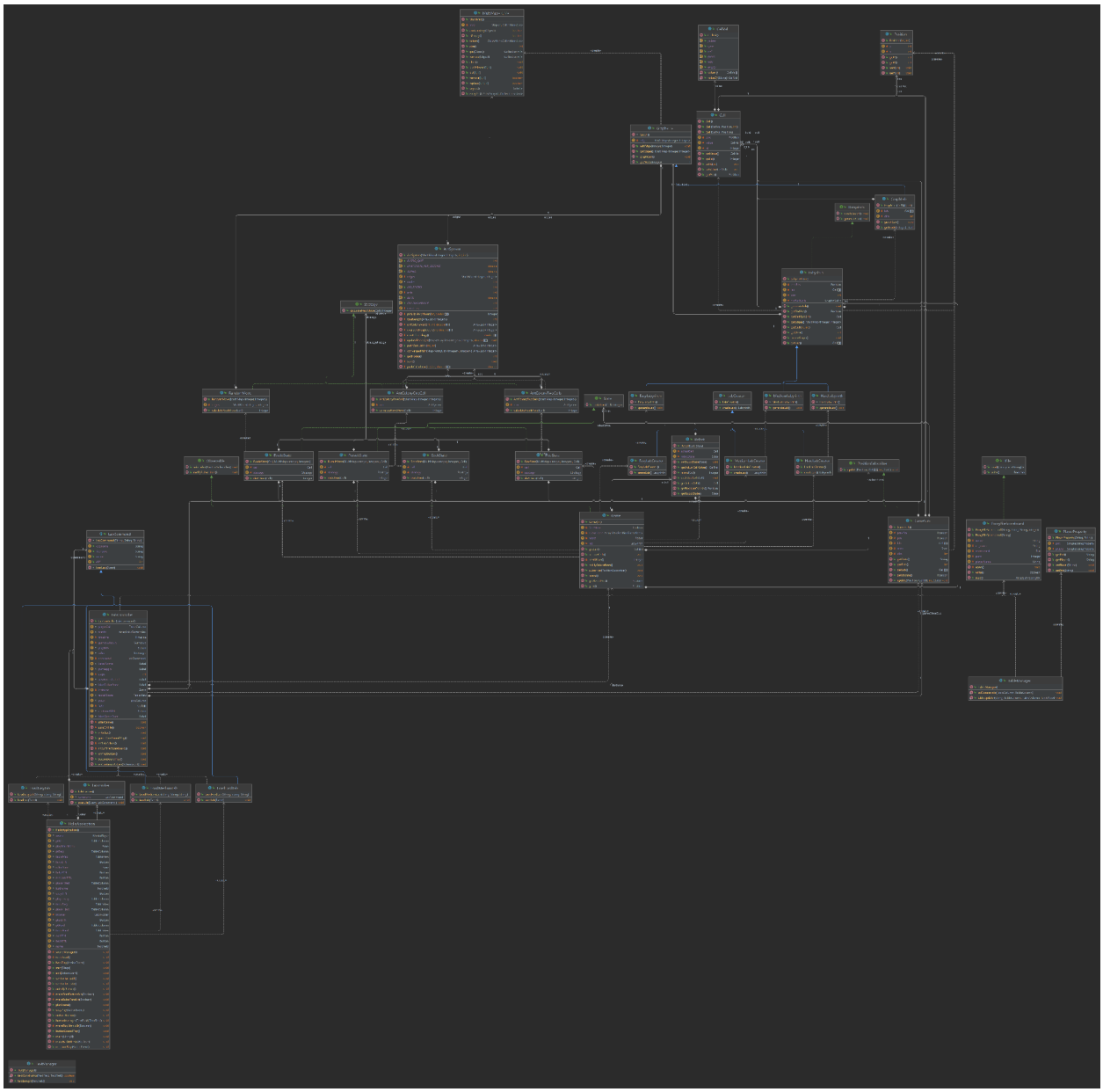
Proxy, infine, fornisce un layer tra il gioco e l'interazione coi files delle scoreboards. E' stato implementato utilizzando un'interfaccia, ***IFile***, che contiene due metodi: ***read*** e ***write***. Questi due, ridefiniti nel ***ProxyFileScoreboard***, permettono di semplificare all'inverosimile l'interazione coi files, poiche' al loro interno e' incapsulata sia la logica dell'interazione che la responsabilita' della stessa.

In particolare, nella classe ***ProxyFileScoreboard*** si crea dapprima nei costruttori la cartella RoboMazeScbrds (nella home del sistema), e successivamente il file scoreboard coerentemente con la difficolta' (si noti che queste operazioni sono eseguite unicamente una volta, poiche' dopo la creazione di cartella/scoreboards le funzioni addette alla creazione daranno esito negativo, facendo semplicemente da "checker" dell'esistenza delle cartelle/files).

Le funzioni read e write, quindi, permettono di leggere i files delle scoreboard (implementati come .dat) e di scriverci sopra i punteggi in maniera ordinata, facendo attenzione a controllare che il player non abbia gia' giocato una partita in precedenza (in caso positivo, salvera' il suo score se e soltanto se avra' migliorato il suo punteggio).



# UML

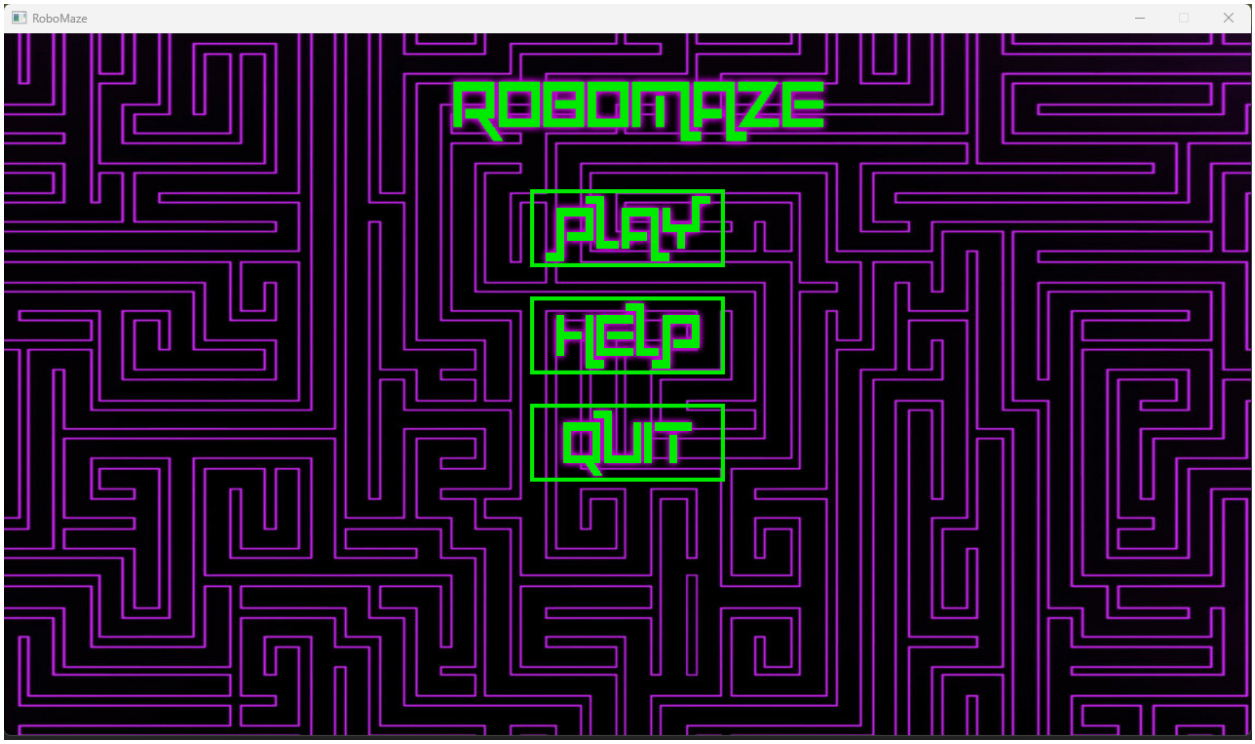


Per consentirne una corretta visualizzazione, ***l'UML*** completo e' disponibile anche all'interno dell'archivio come png.

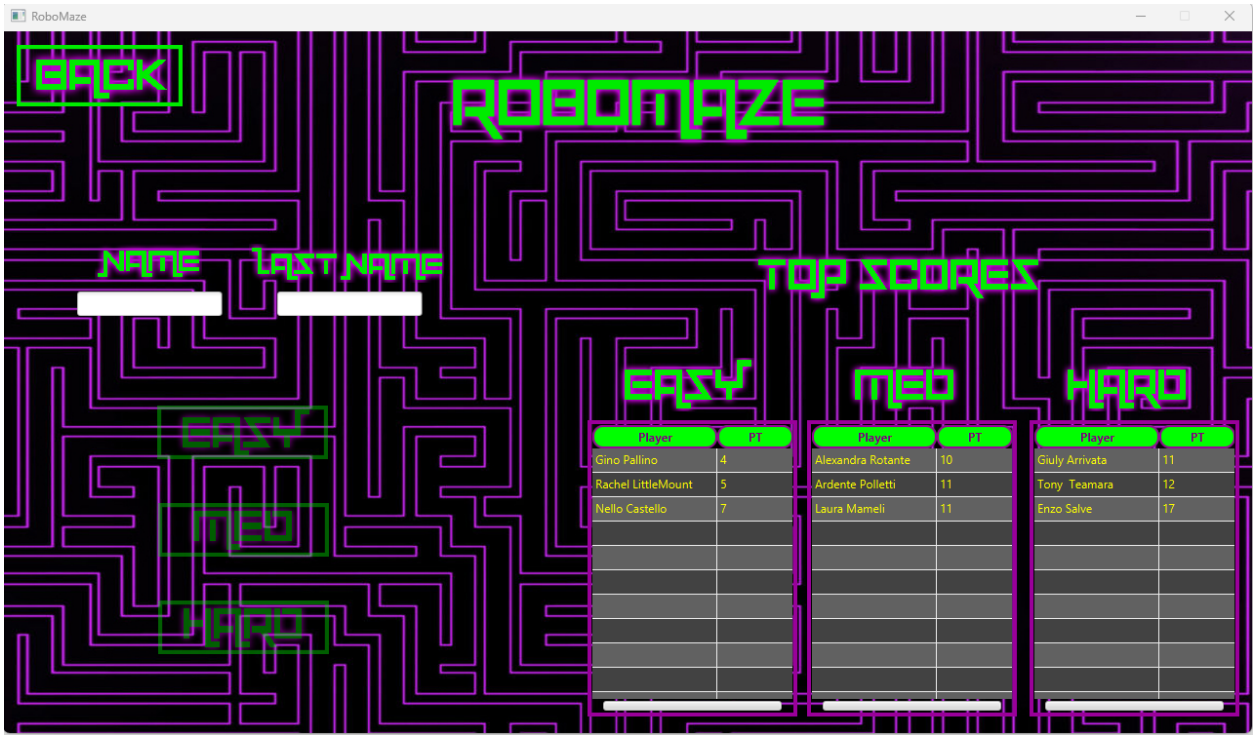
## ***L'interfaccia grafica***

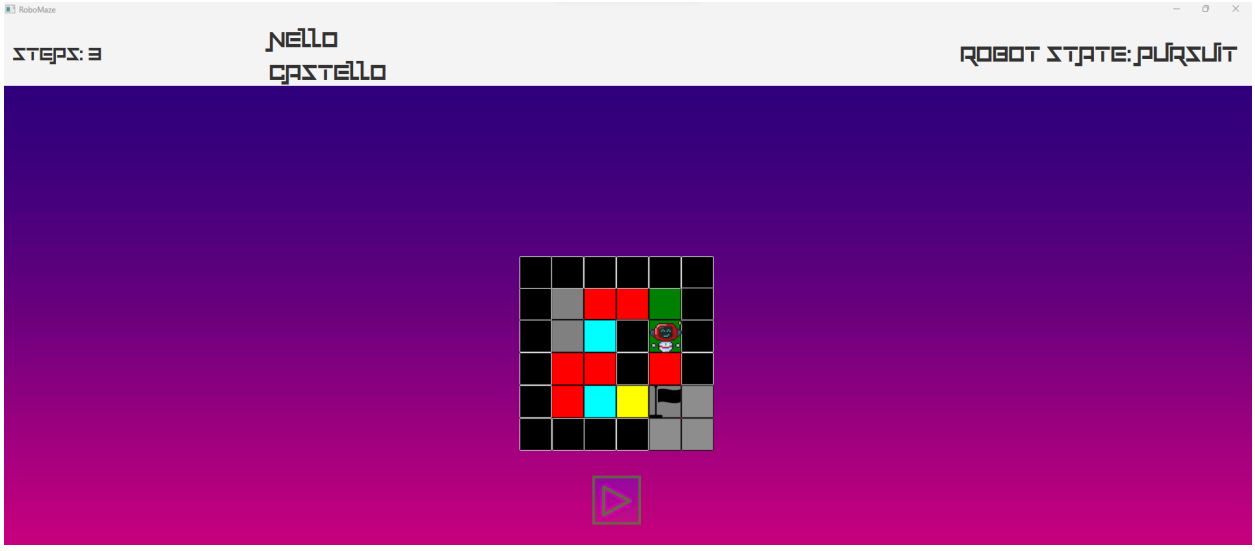
L'interfaccia grafica è stata creata tramite l'utilizzo di ***JavaFX,*** una piattaforma software altamente flessibile e versatile, basata su Java, che consente la creazione di interfacce utente ricche e dinamiche che supportano la multimedialità in maniera rapida e agevole, abbiamo inoltre utilizzato software come ***Scene Builder*** ed ***Adobe Photoshop***, infine abbiamo definito lo stile degli elementi della nostra interfaccia grazie all'ausilio del linguaggio **CSS**.

Abbiamo puntato su una GUI dal design accattivante anche grazie all'utilizzo di una accurata selezione di una palette di colori vivaci ed originali.



Il progetto si struttura in 4 file *.fxml* che rappresentano il nostro menu principale *"**main-menu**"* e i 3 labirinti divisi per livello di difficoltà "***easyLab***", "***mediumLab***" e "***hardLab***", generalmente il controller del file fxml viene assegnato all'interno del file stesso; nel nostro caso, però, abbiamo scelto di assegnare il controller tramite codice.



Per quanto riguarda la gestione delle caselle nei vari scenari, ad ogni casella viene attribuito un **'fx:id'** permettendo così di inserire ognuna di esse all'interno di un ArrayList, ogni elemento di quest'ultimo corrisponde ad ogni singolo elemento della matrice che regola il colore dell'oggetto presente nelle caselle a livello logico nel nostro codice, in maniera tale da rendere possibile l'aggiornamento grafico dei colori all'interno del labirinto.

```
<fx:define>
   <ArrayList fx:id="matrix">
      <fx:reference source="Rectangle1" />
      <fx:reference source="Rectangle2" />
      <fx:reference source="Rectangle3" />
      <fx:reference source="Rectangle4" />
      <fx:reference source="Rectangle5" />
      <fx:reference source="Rectangle6" />
      <fx:reference source="Rectangle7" />
      <fx:reference source="Rectangle8" />
      <fx:reference source="Rectangle9" />
      <fx:reference source="Rectangle10" />
      <fx:reference source="Rectangle11" />
      <fx:reference source="Rectangle12" />
      <fx:reference source="Rectangle13" />
      <fx:reference source="Rectangle14" />
      <fx:reference source="Rectangle15" />
      <fx:reference source="Rectangle16" />
   </ArrayList>
</fx:define>
```

Inoltre, al fine di rendere più godibile l'esperienza dell'utente, abbiamo deciso di inserire degli effetti sonori e una colonna sonora, per farlo abbiamo utilizzato le classi "***Media***" e "***MediaPlayer***" contenuti all'interno del package '***javafx.scene.media.Media'***.

All'interno del codice la gestione degli effetti sonori viene affidata ai metodi ***buttonSoundPlay*** e ***soundManager*** che si occupano di riprodurre i file multimediali nel momento in cui vengono richiamati gli eventi interessati.