**UNIVERSITA’ DEGLI STUDI DI NAPOLI “PARTHENOPE”**



Dipartimento di Scienze e Tecnologie

Corso di Laurea Triennale in

*Informatica*

Relazione di progetto per l’esame

“Programmazione 3 e Laboratorio di Programmazione 3”

Traccia - Labirinto 2

ow987yljysb

|  |  |
| --- | --- |
|  |  |

**Sommario**

[Definizione del problema 3](#_Toc172199402)

[Descrizione sviluppo 3](#_Toc172199403)

[Diagramma delle classi 4](#_Toc172199404)

[Singleton 4](#_Toc172199405)

[DbConnectionSingleton 5](#_Toc172199406)

[Factory 6](#_Toc172199407)

[ScreenFactory 7](#_Toc172199408)

[HomeScreen 7](#_Toc172199409)

[NameInputScreen 7](#_Toc172199410)

[MazeScreen 7](#_Toc172199411)

[HighScoreScreen 8](#_Toc172199412)

[Builder 9](#_Toc172199413)

[MazeBuilder 9](#_Toc172199414)

[Maze 10](#_Toc172199415)

[Player 11](#_Toc172199416)

[Position 11](#_Toc172199417)

[Strategy 11](#_Toc172199418)

[Interfaccia MovementStrategy 12](#_Toc172199419)

[AStarMovementStrategy 12](#_Toc172199420)

[PlayerMovementStrategy 12](#_Toc172199421)

[RandomMovementStrategy 13](#_Toc172199422)

[Bibliografia e sitografia 14](#_Toc172199423)

# **Definizione del problema**

Il progetto mira allo sviluppo di un algoritmo per il cammino di un robot all’interno di un labirinto. La stanza è composta da tasselli quadrati, dotata di pareti esterne e all’interno contiene *N* ostacoli, selezionati dall’utente su tre livelli di difficoltà.  
Il robot può muoversi solo di una casella alla volta tra le otto caselle vicine, secondo la seguente logica:

* nel 30% dei casi il robot si muove a caso in una delle quattro caselle vicine possibili (parete permettendo)
* nel 30% dei casi il giocatore decide la direzione
* nel 40% dei casi la direzione del robot viene calcolata usando l’algoritmo A∗

Ad ogni movimento del robot la disposizione degli ostacoli cambia. La simulazione termina quando il robot esce dalla porta della stanza. Il programma deve permettere di registrare un giocatore con il suo nome e cognome e visualizzare, ad ogni inizio e fine partita, la classifica dei risultati migliori ottenuti, da tutti i giocatori, in tutte le partite (minore numero di passi per raggiungere l’uscita).

# **Descrizione sviluppo**

Il sistema è sviluppato con **Java 17**, implementando il toolkit **Swing** ed in particolare le proprietà della classe **JFrame** per la gestione dell’interfaccia grafica, che è stata pensata per essere user-friendly per la selezione della difficoltà edl’inserimento dei dati utente e comprensibile nel funzionamento per la parte di gioco.

È presente un interfacciamento con il database **MySQL** (XAMPP) tramite l’API **JDBC** per la gestione dello storico punteggio, questa scelta consente una gestione efficiente e sicura delle informazioni necessarie.

Il sistema presenta le schermate organizzate e visualizzate in questo ordine:

1. **HomeScreen**: titolo del gioco e tre pulsanti dai quali è possibile selezionare la difficoltà (numero di ostacoli presenti sulla mappa)
2. **HighScoreScreen**: visualizzazione dei 50 migliori punteggi in classifica
3. **NameInputScreen**: inserimento dei dati utente
4. **MazeScreen**: visualizzazione grafica del labirinto con counter mosse e visualizzazione strategia di movimento utilizzata. Movimento possibile mediante WASD.

Durante lo sviluppo del progetto, ho incorporato i Design Pattern: **Singleton** (connessione al DB), **Builder** (creazione labirinto), **Factory** (creazione schermate) e **Strategy**

(implementazione strategia movimento). Questa scelta è stata guidata dal voler garantire i principi SOLID, finalizzati alla creazione di una struttura flessibile e manutenibile.

Il labirinto viene generato randomicamente ad ogni avvio partita; ogni mappa generata viene sottoposta ad un controllo con algoritmo FloodFill che garantisce la giocabilità della mappa prima di essere visualizzata a schermo, il controllo viene riproposto ad ogni rigenerazione degli ostacoli, per garantire sempre un percorso verso l’uscita al robot.

Per garantire una comprensione chiara del codice, ho incluso commenti dettagliati mediante JavaDoc e commenti interlinea che offrono spiegazioni esaustive su ciascuna parte del sistema.

La gestione delle eccezioni è stata implementata con cura, assicurando un comportamento robusto anche in situazioni impreviste.

# **Diagramma delle classi**

## **Singleton**

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Il pattern Singleton è un modello di progettazione creazionale che assicura che una classe abbia una sola istanza durante l'intero ciclo di vita di un'applicazione e fornisce un punto di accesso globale a questa istanza. Questo è particolarmente utile quando è necessario un controllo centralizzato su una risorsa, come: una connessione al database, una configurazione, o un logger.

### **DbConnectionSingleton**

La classe DbConnectionSingleton implementa il pattern Singleton per gestire la connessione al database nell'applicazione "The Labirinto". Questo assicura che esista una sola connessione condivisa da tutte le parti dell'applicazione che necessitano di accedere al database.

**Costruttore Privato**

Il costruttore è privato per prevenire la creazione di più istanze. All'interno del costruttore, vengono caricate le proprietà del database e viene registrato il driver JDBC.

**loadProperties**

Carica le proprietà di configurazione del database dal file db.properties, garantendo che tutte le informazioni necessarie per la connessione siano disponibili e valide.

**getInstance**

Restituisce l'unica istanza della classe DbConnectionSingleton. Se l'istanza non esiste ancora, la crea.

**buildConnection**

Crea una nuova connessione al database se non esiste o se è chiusa.

**getConnection**

Ritorna l'oggetto Connection, assicurandosi che sia attivo e utilizzabile.

**closeConnection**

Chiude la connessione al database, rilasciando le risorse.

**switchToDatabase**

Seleziona il database da usare nella connessione corrente, eseguendo il comando SQL appropriato.

**createDatabase**

Crea il database e la tabella player se non esistono, garantendo che la struttura del database sia pronta per l'uso.

**insertPlayer**

Inserisce un nuovo giocatore nella tabella player, utilizzando una query parametrizzata per prevenire SQL injection.

**getTopPlayersByDifficulty**

Recupera i migliori giocatori per una data difficoltà, ordinandoli per il numero di mosse effettuate, e li ritorna in una lista.

## **Factory**

Immagine che contiene schermata, testo, Software multimediale, Software per la grafica

Descrizione generata automaticamente

Il pattern Factory è un modello di progettazione creazionale che consente di delegare l'istanziazione di oggetti ad una classe Factory anziché farlo direttamente nel codice client. Questo approccio favorisce il principio di separazione delle responsabilità e migliora la modularità del codice.

### **ScreenFactory**

La classe ScreenFactory gestisce la creazione di diverse schermate (JPanel) in base al tipo specificato (ScreenType). Utilizza un costruttore che riceve un oggetto MainFrame, necessario per inizializzare le schermate correttamente.

**CreateScreen (ScreenType screenType)**:

Utilizza uno switch-case per determinare quale tipo di schermata creare in base al ScreenType fornito. Ritorna un nuovo oggetto JPanel corrispondente alla schermata richiesta.

### **HomeScreen**

HomeScreen è una schermata iniziale che mostra il titolo del gioco e i pulsanti per selezionare la difficoltà del labirinto.

**initializeLayout()**:

Configura il layout della schermata e aggiunge il titolo, un'immagine e tre pulsanti per le difficoltà.

**ProceedToNextScreen (Difficulty difficulty)**:

Metodo privato chiamato dai pulsanti per passare alla schermata successiva (HighScoreScreen) e comunicare la difficoltà selezionata.

### **NameInputScreen**

Raccoglie il nome e il cognome del giocatore prima di avviare il gioco.

**ProceedToNextScreen (MainFrame mainFrame)**:

Passa alla schermata successiva (MazeScreen) dopo aver validato l'input del giocatore.

### **MazeScreen**

Visualizza il labirinto e gestisce l'interazione con il giocatore tramite tastiera.

**loadImages()**:

Carica le immagini utilizzate per rappresentare il labirinto.

**initializeLayout()**

Configura il layout della schermata, inclusi i pannelli per il labirinto e per le informazioni di gioco.

**setupKeyBindings()**

Imposta i key bindings per gestire l'input da tastiera.

**handleKeyPress()**

Gestisce gli eventi di pressione dei tasti per il movimento nel labirinto.

**handleMovement(int dx, int dy)**

Gestisce il movimento del giocatore nel labirinto, aggiorna la posizione e controlla se il giocatore ha raggiunto l'uscita.

**proceedToNextScreen(MainFrame mainFrame)**

Passa alla schermata successiva (HighScoreScreen) dopo aver completato il labirinto.

### **HighScoreScreen**

### **HighScoreScreen** mostra i punteggi più alti per la difficoltà selezionata.

**loadHighScores(JPanel scorePanel, MainFrame mainFrame)**

Carica e visualizza i punteggi più alti per la difficoltà corrente utilizzando il Singleton della connessione al database (DbConnectionSingleton).

**proceedToNextScreen(MainFrame mainFrame)**:

Torna alla schermata precedente (NameInputScreen) per permettere al giocatore di iniziare una nuova partita.

## **Builder**

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Il pattern Builder è un design pattern creazionale che viene utilizzato per separare la costruzione di un oggetto complesso dalla sua rappresentazione. Questo permette di creare oggetti complessi passo dopo passo, controllando il loro processo di costruzione e consentendo configurazioni diverse senza dover modificare direttamente la classe dell'oggetto che si sta costruendo.

### **MazeBuilder**

**addExit()**:

Aggiunge un'uscita al labirinto. Questo metodo posiziona l'uscita in una posizione casuale all'interno del labirinto e la marca sulla mappa con il valore 2.

**addRobot()**:

Aggiunge un robot al labirinto. Il robot viene posizionato nella parte inferiore del labirinto, evitando i bordi, e viene marcato sulla mappa con il valore 3.

**addWalls()**

Aggiunge le pareti al labirinto. Le pareti vengono posizionate in posizioni casuali non occupate da robot, uscita o bordi, e sono rappresentate con il valore 1 sulla mappa.

**addEdges()**

Aggiunge i bordi al labirinto. I bordi sono posizionati lungo il perimetro del labirinto e sono rappresentati con il valore 1 sulla mappa.

**setPlayer(String name, String surname)**

Imposta il giocatore con nome e cognome. Questo metodo crea un oggetto Player associato al labirinto con le informazioni del giocatore fornite.

**build()**

Costruisce e restituisce un oggetto Maze completo con tutte le configurazioni applicate. Questo metodo finalizza la costruzione del labirinto, creando un'istanza di Maze che incorpora la mappa, la posizione del robot, l'uscita, le pareti, i bordi e il giocatore.

### **Maze**

**updateRobot(Position newRobotPosition)**

Aggiorna la posizione del robot nel labirinto. Questo metodo cambia la posizione del robot sulla mappa, aggiornando sia la vecchia posizione a 0 che la nuova posizione a 3.

**isValidMove(int x, int y)**

Verifica se una mossa è valida nel labirinto. Questo metodo controlla se le coordinate fornite sono all'interno dei limiti del labirinto e se la casella è attraversabile (0) o rappresenta l'uscita (2).

**regenerateMap(Difficulty difficulty)**

Rigenera la mappa del labirinto in base alla difficoltà specificata. Questo metodo reimposta la mappa del labirinto aggiungendo nuove pareti in posizioni casuali finché non è più possibile raggiungere l'uscita dal robot. Questo garantisce che il labirinto sia sempre giocabile.

### **Player**

**Metodi getter e setter**

Per name, surname, moves, difficulty e altri attributi associati al giocatore.

**addMoves()**:

Incrementa il contatore delle mosse del giocatore.

### **Position**

**Metodi getter e setter**

Per x e y.

**Metodi equals(Object o) e hashCode()**

Per confrontare le posizioni.

## **Strategy**

Immagine che contiene testo, schermata, Software multimediale, Software per la grafica

Descrizione generata automaticamente

Il pattern Strategy è un design pattern comportamentale che consente di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Questo pattern permette di selezionare l'algoritmo da utilizzare in fase di esecuzione, favorendo la flessibilità e la riusabilità del codice. Nel contesto delle classi inviate, il pattern Strategy è applicato per gestire diverse strategie di movimento all'interno del labirinto.

### **Interfaccia MovementStrategy**

**getNextPosition(Maze maze, Position currentPosition)**

Metodo astratto che deve essere implementato dalle classi concrete. Questo metodo prende come input il labirinto e la posizione corrente del robot, e restituisce la prossima posizione del robot in base alla strategia di movimento specifica.

### **AStarMovementStrategy**

**getNextPosition(Maze maze, Position currentPos)**

Implementa l'algoritmo A\* per trovare il percorso ottimale dalla posizione corrente del robot alla posizione dell'uscita. Restituisce la prossima posizione nel percorso.

**heuristicCostEstimate(Position a, Position b)**

Calcola la stima del costo euristico tra due posizioni utilizzando la distanza di Manhattan.

**getLowestFScore(Set<Position> openSet, Map<Position, Integer> fScore)**:

Trova la posizione con il punteggio F più basso nell'insieme aperto.

**getNeighbors(Position pos, Maze maze)**

Ottiene le posizioni adiacenti valide dalla posizione corrente.

**reconstructPath(Map<Position, Position> cameFrom, Position current)**

Ricostruisce il percorso dal punto di partenza all'uscita utilizzando la mappa cameFrom.

### **PlayerMovementStrategy**

**PlayerMovementStrategy(int dx, int dy)**

Costruttore che inizializza la direzione del movimento.

**setDirection(int dx, int dy)**

Imposta una nuova direzione di movimento.

**getNextPosition(Maze maze, Position currentPos)**

Calcola la prossima posizione del giocatore sommando i valori di dx e dy alle coordinate della posizione corrente.

### **RandomMovementStrategy**

**getNextPosition(Maze maze, Position currentPosition)**

Seleziona casualmente una delle posizioni adiacenti valide come prossima mossa. Se non ci sono posizioni valide disponibili, il robot rimane nella sua posizione attuale.

# **Database**

Il database è organizzato tramite una singola tabella, in quanto durante la progettazione ho pensato che potendo l’utente inserire solo nome e cognome, ci potessero essere casi di omonimia, la tabella è organizzata come segue:

Immagine che contiene testo, Carattere, numero, schermata

Descrizione generata automaticamente

In future implementazioni, si può pensare di aggiungere un campo username in input che sarà univoco per ogni player, e dividere la tabella in Player e Score, in relazione 1:N con chiave primaria il campo username.

# **Bibliografia e sitografia**

* <https://www.marcobehler.com/guides/jdbc>, Marco Behler
* <https://stackoverflow.com/>
* <https://www.geeksforgeeks.org/java-swing-jpanel-with-examples/>
* <https://www.youtube.com/watch?v=dvzAuq-YDpM&list=PLZPZq0r_RZOMhCAyywfnYLlrjiVOkdAI1&index=53&ab_channel=BroCode>
* <https://www.apachefriends.org/it/index.html>