**VIETNAM NATIONAL UNIVERSITY—HO CHI MINH CITY**

**INTERNATIONAL UNIVERSITY**

**PROJECT REPORT**

**FINANTASY**

**MINIGAME**

**ALGORITHMS & DATATRUCTURES (IT013IU)**

Course by

**Dr. Vi Chi Thanh and MSc. Thai Trung Tin**

**Members\_ID Student**

Phan Trần Thanh Huy - ITCSIU22056

Nguyễn Đình Khánh Ngân - ITCSIU22236

**TABLE OF CONTENTS**

**Chapter 1. INTRODUCTION 3**

[1. Gaming in the Field 3](#_1fob9te)

2[. Our game project 4](#_2et92p0)

3. References 5

4[. Developer team 6](#_uq12be2nigd4)

**Chapter 2. SOFTWARE REQUIREMENTS 7**

1. [Working tools, platform 7](#_o3vuobbsjh7k)

2[. Design Pattern 8](#_ie0z4yebe6ul)

3[. Class diagram 8](#_ejhxh9jy0ces)

[**Chapter 3. DESIGN & IMPLEMENTATION 14**](#_14kam3i4ogul)

[1. Programming Languages and Tools 14](#_hkjjntwyirs6)

2. Data [Structures and Algorithms 22](#_613qns7irp5p)

[a.](#_4nanbqic7cac) Core Data Structures [22](#_4nanbqic7cac)

[b. Algorithms 22](#_97qovu7586db)

[c. Complexity Analysis 23](#_kgsqzpx8rhxo)

3[. Conclusion 24](#_ugojiob0qu8n)

[**Chapter 4. FINAL APP GAME 27**](#_obunkzs6to41)

[1. Title Screen 27](#_5831rvg74e7z)

[2. Gameplay Tetris Screen 28](#_lbae40vg7wod)

3[. Gameplay Maze Screen 28](#_lbae40vg7wod)

4[. End Screen 31](#_j1phoqw1890v)

# **Chapter 1. INTRODUCTION**

## **Gaming in the Field**

The realm of Data Structures and Algorithms (DSA) plays a pivotal role in the development of efficient software systems. For our four-credit "Data Structures and Algorithms" course, which contributes to our degree, we have chosen to design and implement a problem-solving application that focuses on optimizing algorithmic performance using various data structures. This project allows us to deepen our understanding of essential concepts such as trees, graphs, dynamic programming, and sorting algorithms.

Our goal is to build a system that not only performs efficiently but also adapts to varying complexities in input size and problem constraints. Through this project, we aim to improve our problem-solving and coding skills while also applying theoretical knowledge to practical scenarios. The project will challenge us to optimize solutions, analyze time and space complexity, and ultimately refine our understanding of algorithmic design. It also provides an opportunity to collaborate with peers to tackle real-world computational challenges and test our skills in a comprehensive, hands-on way.

## **Our game project**

Our inspiration comes from classic puzzle games and maze-solving challenges, with a modern twist. We sought to combine the thrill of problem-solving and spatial awareness seen in iconic games like *Tetris* and *Maze Runner* while implementing innovative programming techniques to enhance the player experience.

In our game, players navigate through various mazes while also solving Tetris-like puzzles, blending strategy, timing, and spatial reasoning into a single cohesive experience. We have incorporated several key improvements to ensure the game is both challenging and engaging:

* Game Balance and Difficulty Adjustments: We have carefully tuned the difficulty levels of both the maze and Tetris puzzles, ensuring that players are constantly challenged but not overwhelmed.
* Enhanced Visuals and Audio: The game now features polished graphics, along with immersive sound effects and background music to heighten the overall gaming experience.
* Efficient Game Mechanics: By leveraging advanced object-oriented programming concepts, such as the factory pattern and strategy pattern, we have ensured that the game is not only fun but also scalable and maintainable.

This project is designed to appeal to players of all skill levels, offering both casual gameplay and more complex challenges as players advance.

## **References**

Images from

* 1. <https://www.pinterest.com/>
  2. [Top game assets - itch.io](https://itch.io/game-assets)
  3. <https://stackoverflow.com/>

How to make a 2D Game in Java

* + [How to Make a 2D Game in Java](https://www.youtube.com/playlist?list=PL_QPQmz5C6WUF-pOQDsbsKbaBZqXj4qSq)
  + [Object Oriented Programming (OOP) in Java Course](https://www.youtube.com/playlist?list=PL9gnSGHSqcno1G3XjUbwzXHL8_EttOuKk)

Turn-based Game idea

* 1. [Tutorial Series: Turn-Based RPG](https://www.youtube.com/watch?v=Sp623fof_Ck&list=PLPRT_JORnIurSiSB5r7UQAdzoEv-HF24L&index=2)
  2. [How Do You Improve Turn-Based Combat?](https://www.youtube.com/watch?v=ktogjiX3eI4)
  3. [Make a Turn-based RPG Combat Tutorial](https://www.youtube.com/playlist?list=PLFU1tNlBFT4tlmbnsmgrf796g-_6GDNpx)
  4. [More Engaging Turn-Based Combat in RPGs](https://www.youtube.com/watch?v=0Rebci_7ABQ)
  5. [Final Fantasy 1 100% walkthroug](https://www.youtube.com/playlist?list=PLHTKUO2aC5S5khJQCJ8BsN4nBCEVOoYpH)

## **Developer team**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name - Github** | **Student ID** | **Jobs and tasks** | **Contribution** |
| Nguyen Dinh Khanh Ngan [FanchonSora](https://github.com/FanchonSora) | ITCSIU22236 | Design the core algorithm for generating and solving mazes. Implement pathfinding algorithms for navigating through the maze, ensuring optimal routes for players. | 50% |
| Phan Tran Thanh Huy [Alexspector123](https://github.com/Alexspector123) | ITCSIU22056 | Design the core mechanics of the Tetris game, including the falling blocks, rotation system, and line clearing. Implement the logic for detecting block collisions and boundaries, ensuring smooth gameplay. Adjust the speed of the falling blocks and game progression to create a balanced. | 50% |

# **Chapter 2. SOFTWARE REQUIREMENTS**

## **1. Working tools, platforms**

* Visual Studio Code with addition library
* JRE System Library [JavaSE-17] implementation ‘Eclipse Temurin JDK with Hotspot 17.0.3+7 (x64)’
* IntelliJ IDEA for Java and Kotlin
* JRE System Library [JavaSE-17] implementation ‘Eclipse Temurin JDK with Hotspot 17.0.3+7 (x64)’
* Adobe Illustrator 2023
* Adobe Photoshop 2023

## **2. Design Pattern**

**Singleton**

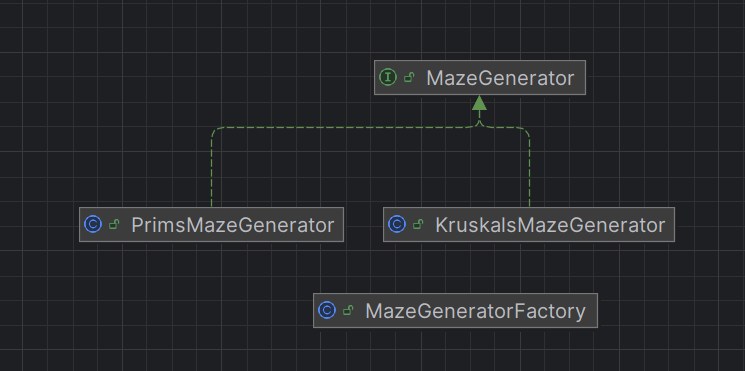
****

The Singleton Pattern ensures that a class has only one instance, and provides a global point of access to that instance. This is useful when you need to control access to shared resources, such as configuration settings, database connections, or in this case, managing a game instance.

### Purpose of Singleton Pattern:

* Control Access to Shared Resources: In a game application, only one instance of the GameManager is needed to manage the game state, ensuring a consistent state and resource access across the entire application.
* Global Access Point: It provides a global point of access to the instance, so no matter where the game state or game logic needs to be accessed, the same instance is used.
* Lazy Initialization: The instance is created only when it is needed, which can help in reducing the initial overhead if not all parts of the application use the singleton.
* Prevents Multiple Instances: It prevents the creation of multiple instances of the GameManager class, which might cause inconsistent behavior or resource conflicts in the application.

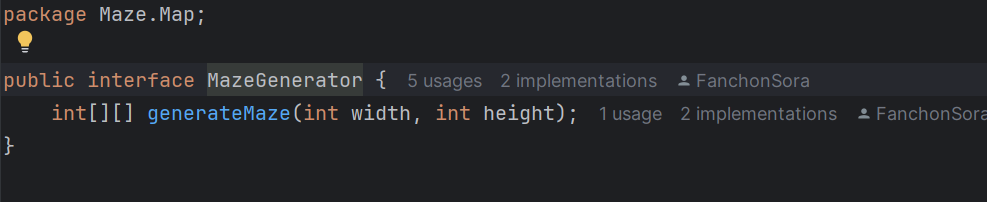
**Factory Method**



The Factory Design Pattern provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. It promotes loose coupling by eliminating the need to bind application-specific classes into the code.

Key Characteristics:

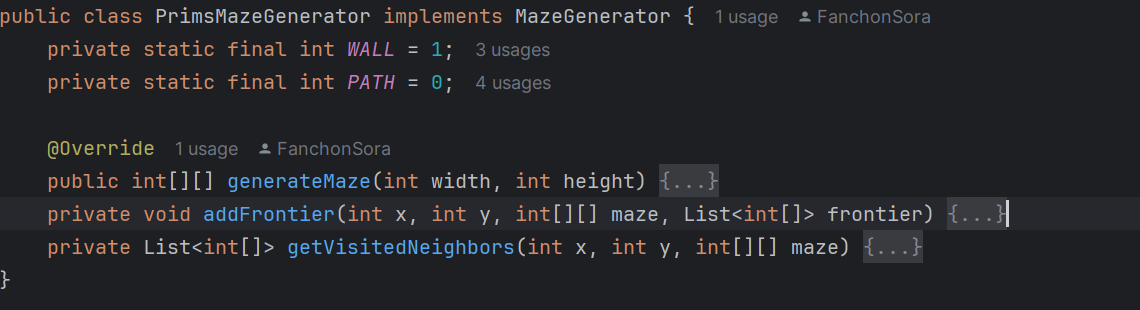
* MazeGenerator (Product Interface/Abstract Class): Defines the interface for objects the factory method creates



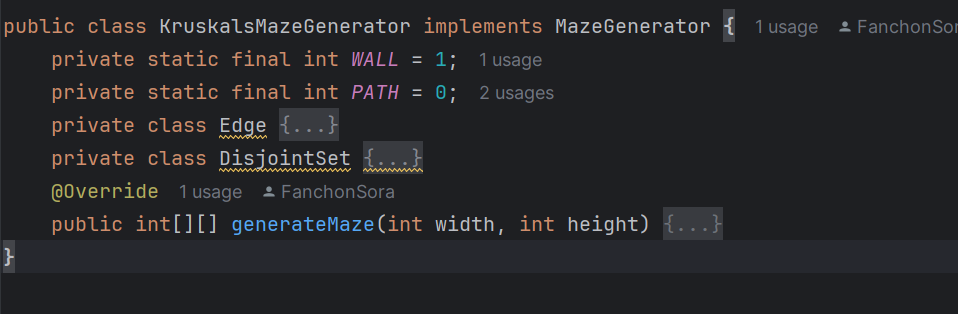
* MazeGeneratorFactory (Creator Abstract Class or Interface): Declares the factory method, which returns an object of type Product.



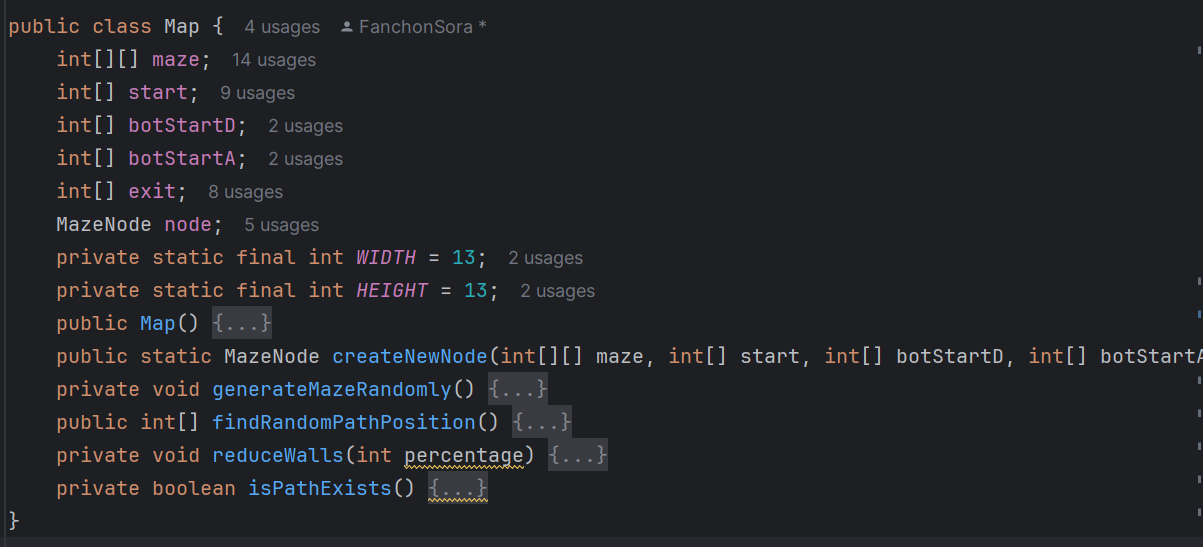
* PrimsMazeGenerator (Concrete Classes): Implement the Product interface.



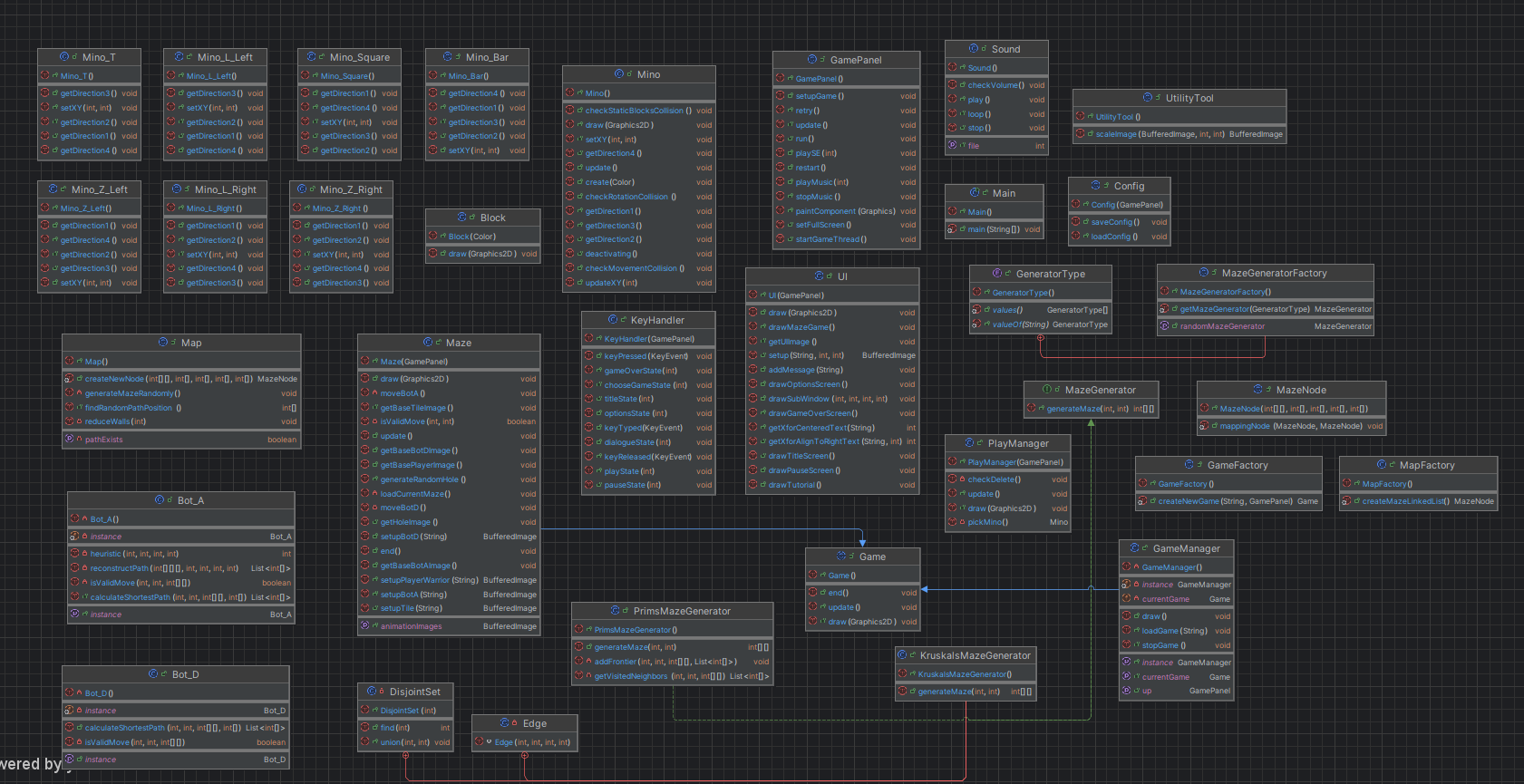
* KruskalsMazeGenerator (Concrete Classes):



* Map Class (Client):



## **3. Class Diagram**



# **Chapter 3. DESIGN & IMPLEMENTATION**

## **Programming Languages and Tools**

For this project, my group has decided to use IntelliJ IDEA and VSCode as an IDE to develop the application which members have frequently used, and excellent community support when they get stuck. This IDE and VSCode also provides tools like, Terminal, and Debugging Tool for testing and assurance purposes when tackling hard-to-achieve tasks, Tool for creating UML. In particular, IntelliJ IDEA and Visual Studio Code support adding external libraries easily.

We can summarize the folders and files' purpose as follows:

* **.idea:** for storing IDE-specific project configurations (JDK level, the pattern, the structure, and many more).
* **out:** the compiled code to run the application.
* **src:** the project's source code mainly contains the class and.fxml files used by G.U.I. Designer.
* **.gitgnore, README.md:** used for project setup on GitHub Version Control System and provide based configuration per developer.
* **target:** is the Maven build directory, which means that all generated content should be placed under that folder.
* config text to save the parameter using for option setting
* **pom.xml:** is where everything related to the project configured through maven is declared, such as declaring dependencies

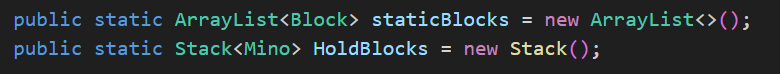
## **Data Structures and Algorithms**

## **Core Data Structures**

### **Tetris**

Efficient data management is pivotal for real-time games like Tetris, where swift rendering and responsive controls are paramount. The following core data structures are integral to the implementation

#### **ArrayList:**



**Usage:**

* + Representing the game grid and positioning of blocks.
  + Storing direction vectors for movement and collision detection.
  + Dynamically storing and managing collections of blocks and effects.

**Characteristics:**

* + Provides indexed access, useful for iteration and manipulation
  + Dynamic resizing capability, facilitating the addition and removal of elements

#### **Stack**



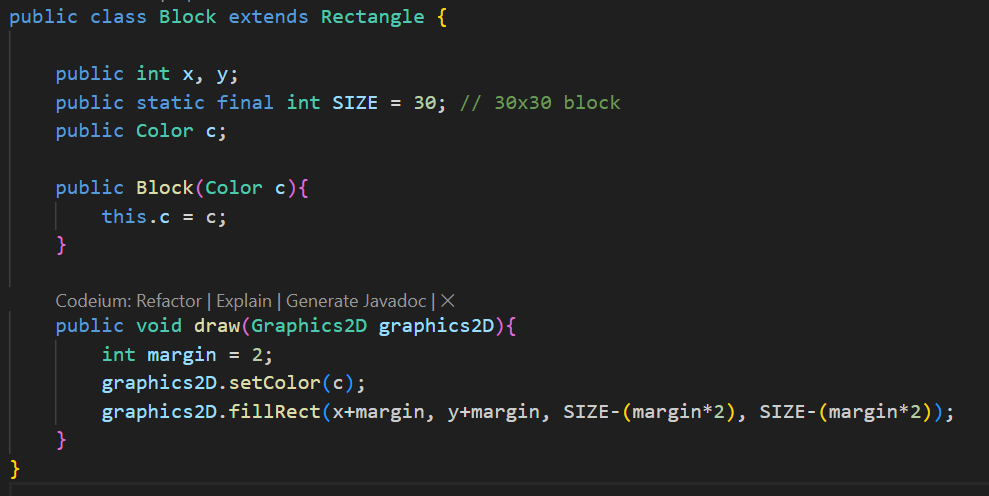
**Usage:**

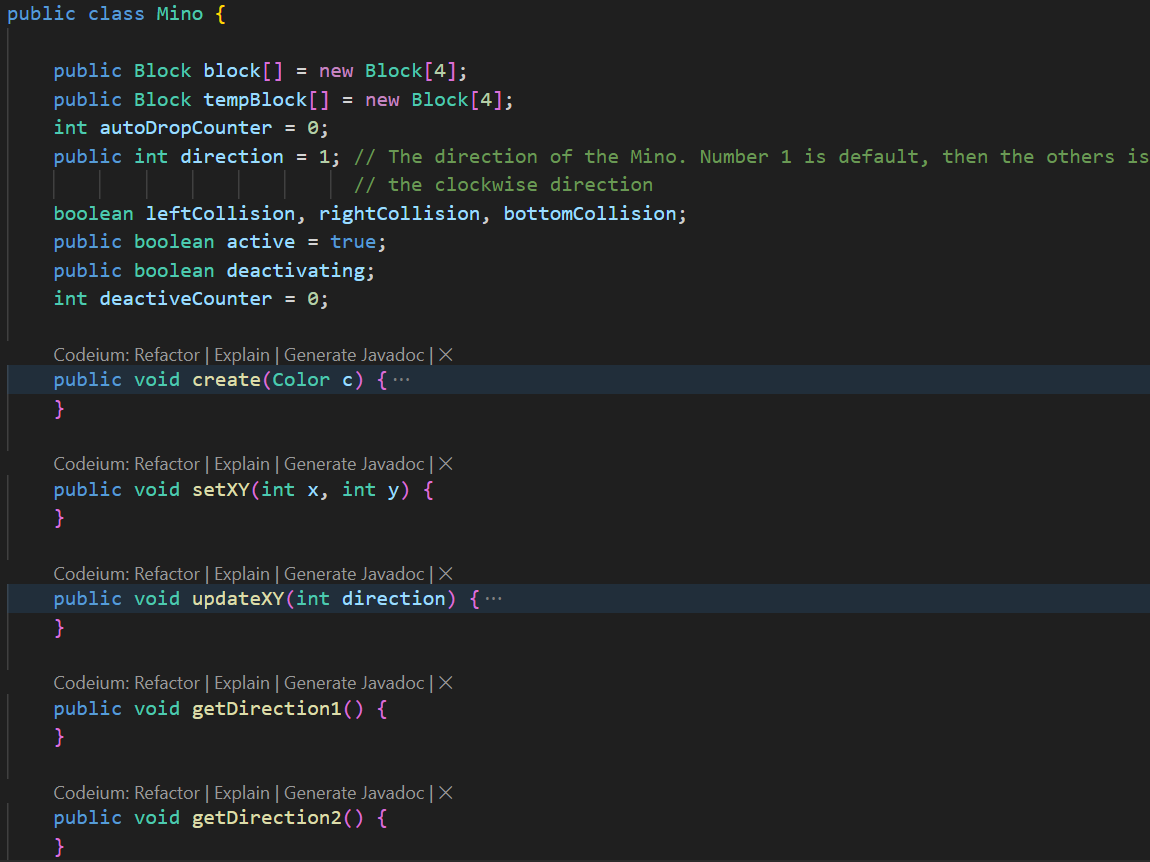
* + - * + Implementing the Hold mechanic to store previously held pieces.

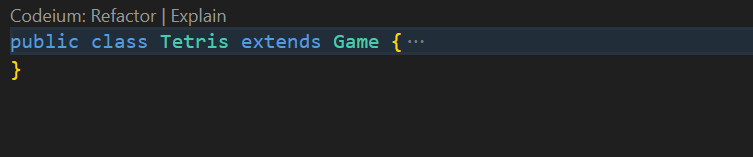
**Characteristics:**

* + Last-In-First-Out (LIFO) structure, ensuring that the most recently held piece is accessible.
  + Efficient for operations that require reversal or backtracking.

#### **Custom Classes**







**Usage:**

* + Encapsulating game entities and their behaviors.

**Characteristics:**

* + Object-oriented approach promotes encapsulation, inheritance, and modularity.
  + Facilitates the management of complex behaviors and interactions between game entities.

### **Maze**

The efficiency and effectiveness of the algorithms are heavily reliant on the underlying data structures. The primary data structures used across the classes include:

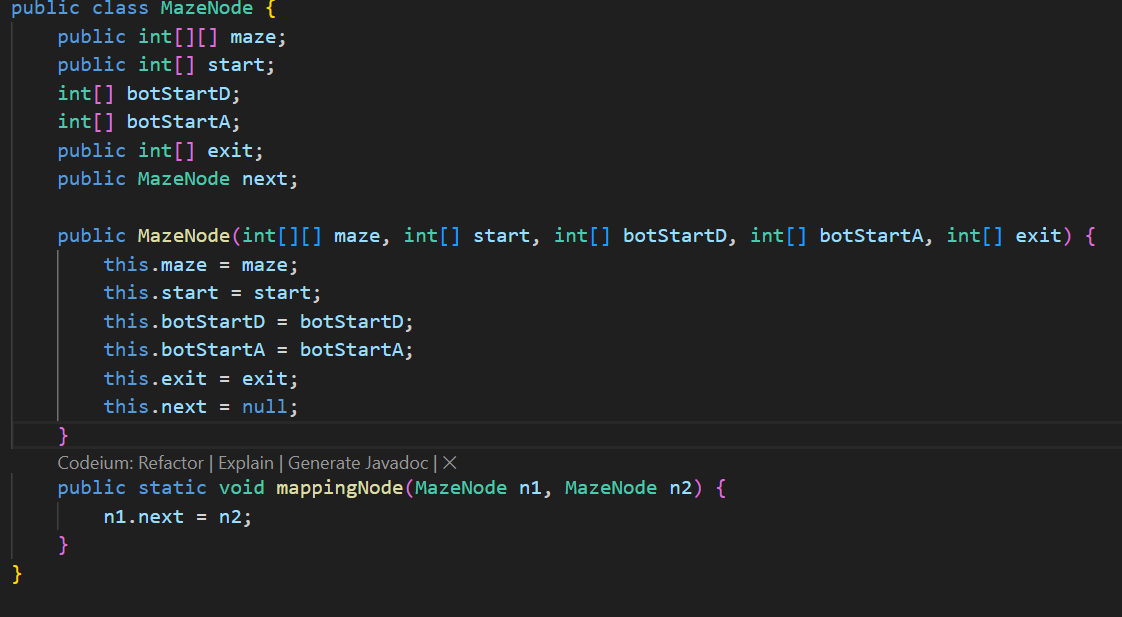
**Arrays (int[][], int[]): **

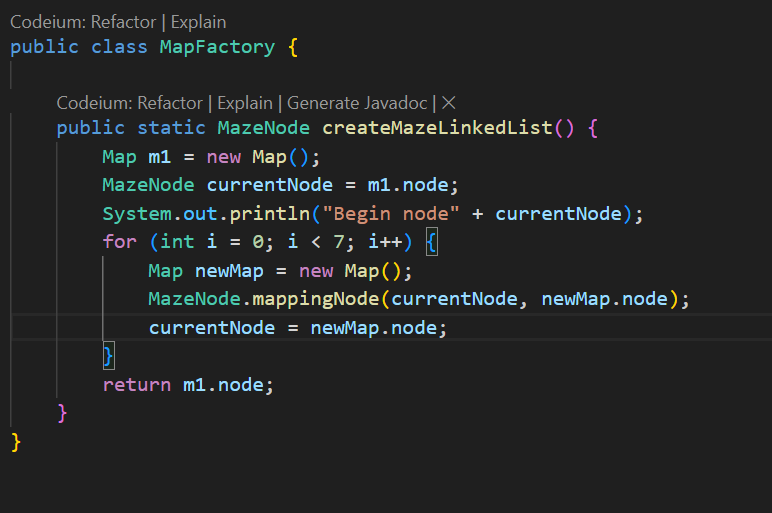
****

**Usage:** Represent the maze grid, coordinates, parent tracking for path reconstruction, and other grid-based data.

**Characteristics:** Fixed-size, indexed access, efficient for grid-based operations.

**Lists (MazeNode currentMaze):**

****

****

**Usage:** Generate a series of mazes (each MazeNode contains one maze). After the player (or bot) solves the current maze, the system checks if there is a next maze (by checking the next reference). If a next maze exists, it loads the next MazeNode in the sequence and updates the game state accordingly. In this way, the MazeNode class is useful for managing and linking multiple mazes together in a chain, making it easy to progress from one maze to the next in a game or simulation

**Characteristics:** Dynamic resizing, linked map elements, sequential access.

**Linked List (List<int[]>, List<int[][]>, List<Edge>):**

****

****

****

**Usage:** Manage dynamic collections such as the maze population in genetic algorithms, edges in Kruskal's algorithm, and frontier cells in Prim's algorithm.

**Characteristics:** Dynamic resizing, ordered elements, sequential access.

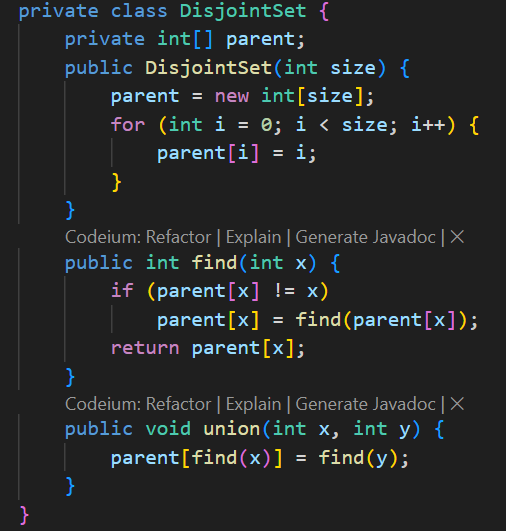
**Priority Queues (PriorityQueue<int[]>):**

****

**Usage:** Implement the open set in pathfinding algorithms, prioritizing nodes based on distance or heuristic values.

**Characteristics:** Maintains elements in a specific order based on a comparator, efficient retrieval of the highest priority element.

**Disjoint Set (Union-Find):**

****

**Usage:** Manage connectivity in Kruskal's algorithm to avoid cycles when building the maze's spanning tree.

**Characteristics:** Efficient union and find operations, used for cycle detection.

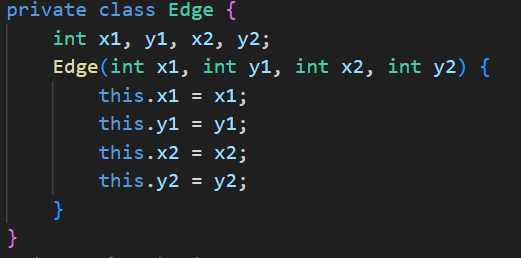
**Booleans (boolean[][]):**

****

**Usage:** Track visited cells in maze generation and pathfinding to prevent redundant processing.

**Characteristics:** Simple flags, efficient storage for visited states.

**Custom Classes (Edge, FitnessMaze):**

****

**Usage:** Encapsulate maze edges for Kruskal's algorithm and store maze instances with their corresponding fitness scores in genetic algorithms

**Characteristics:** Object-oriented encapsulation, facilitating organized data management.

## **Algorithms**

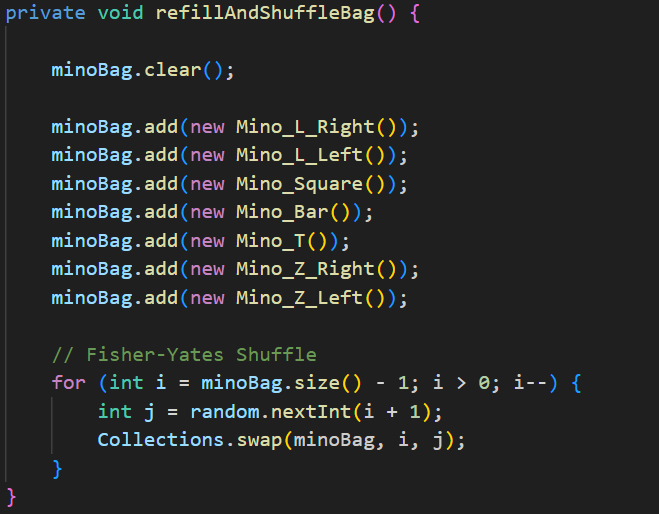
### **Tetris**

#### **Fisher-Yates Shuffle for Mino Bag**

**Purpose:**

Randomly shuffling the sequence of Tetrominoes (Minos) to ensure a fair and unpredictable distribution.

**Implementation:**



**Role:**

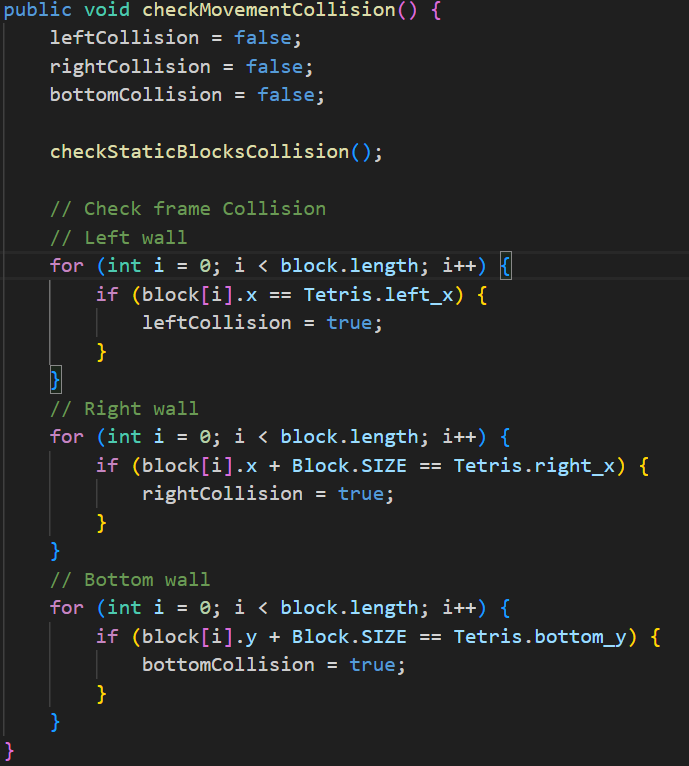
Ensures that each Tetromino appears exactly once before any are repeated, promoting fairness.

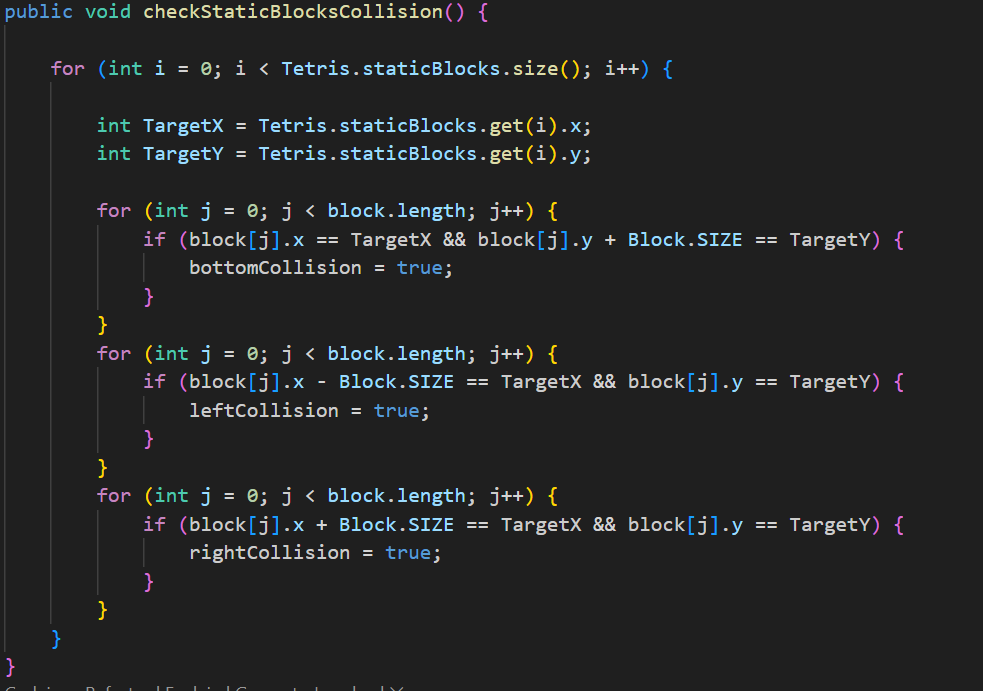
#### **Collision Detection**

**Purpose:**

Determining whether a Tetromino can move in a desired direction without intersecting existing blocks or the boundaries of the play area.

**Implementation:**





**Role:**

Prevents Tetrominoes from overlapping with static blocks or moving outside the play area.

Ensures the game's integrity by enforcing movement constraints.

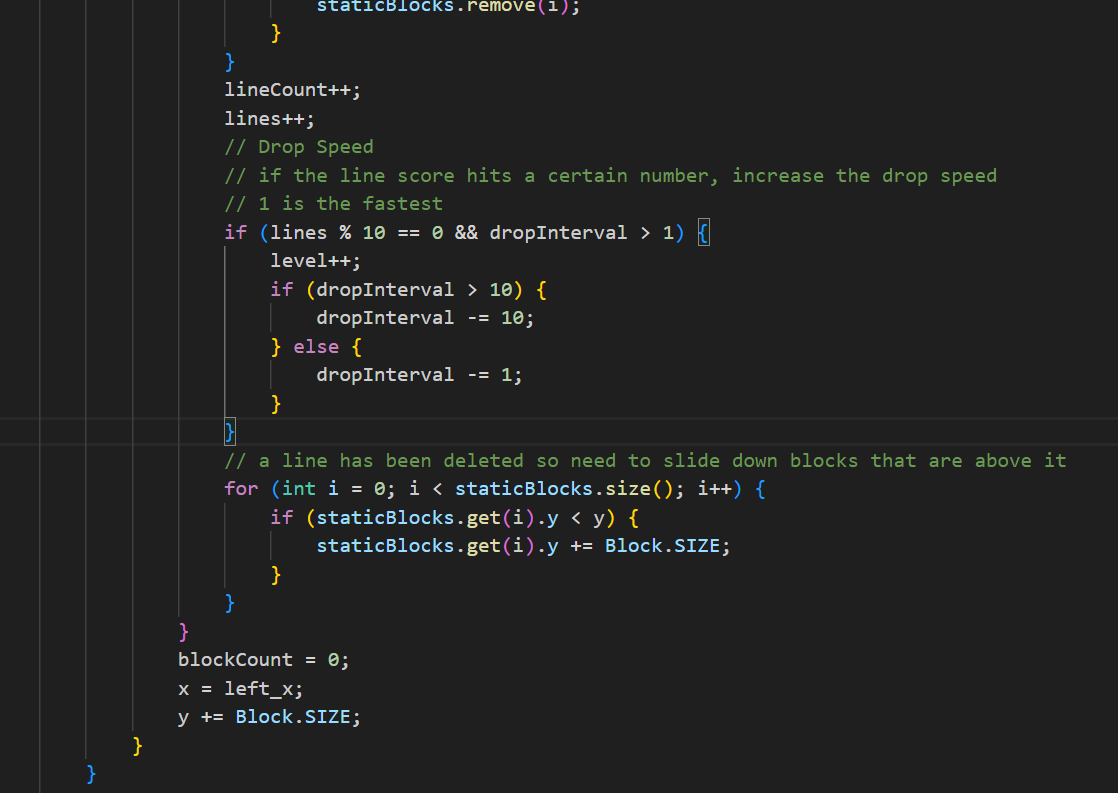
#### **Line Clearing and Scoring**

**Purpose:**

Detecting and clearing fully occupied lines to prevent the play area from filling up, thereby increasing the player's score and level.

**Implementation:**





**Role:**

Enhances gameplay by introducing challenges and rewards.

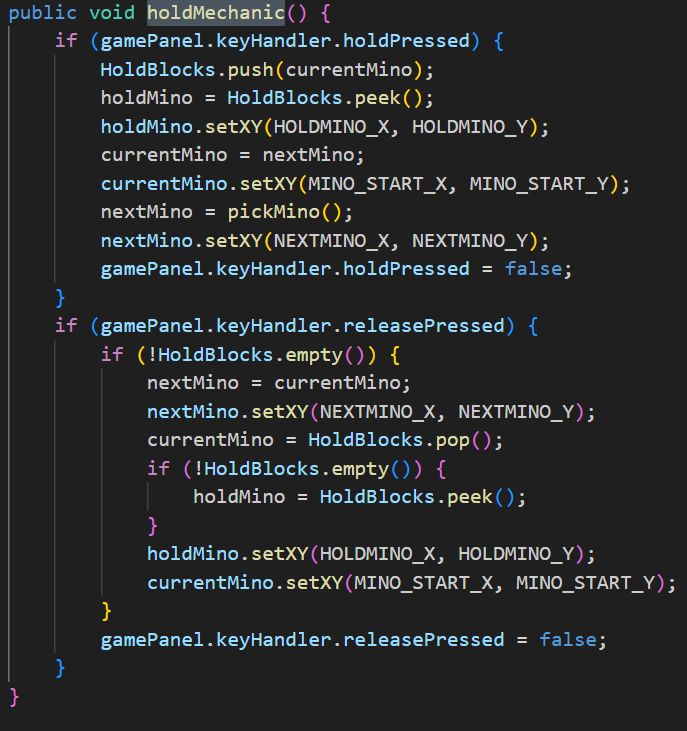
Manages the game's difficulty progression through level increments and speed adjustments.

#### **Hold Mechanic**

**Purpose:**

Allows players to temporarily store a Tetromino for later use, adding strategic depth to gameplay.

**Implementation:**



**Role:**

Enhances player strategy by allowing the management of upcoming Tetrominoes.

Prevents gameplay stagnation by offering flexibility in Tetromino utilization.

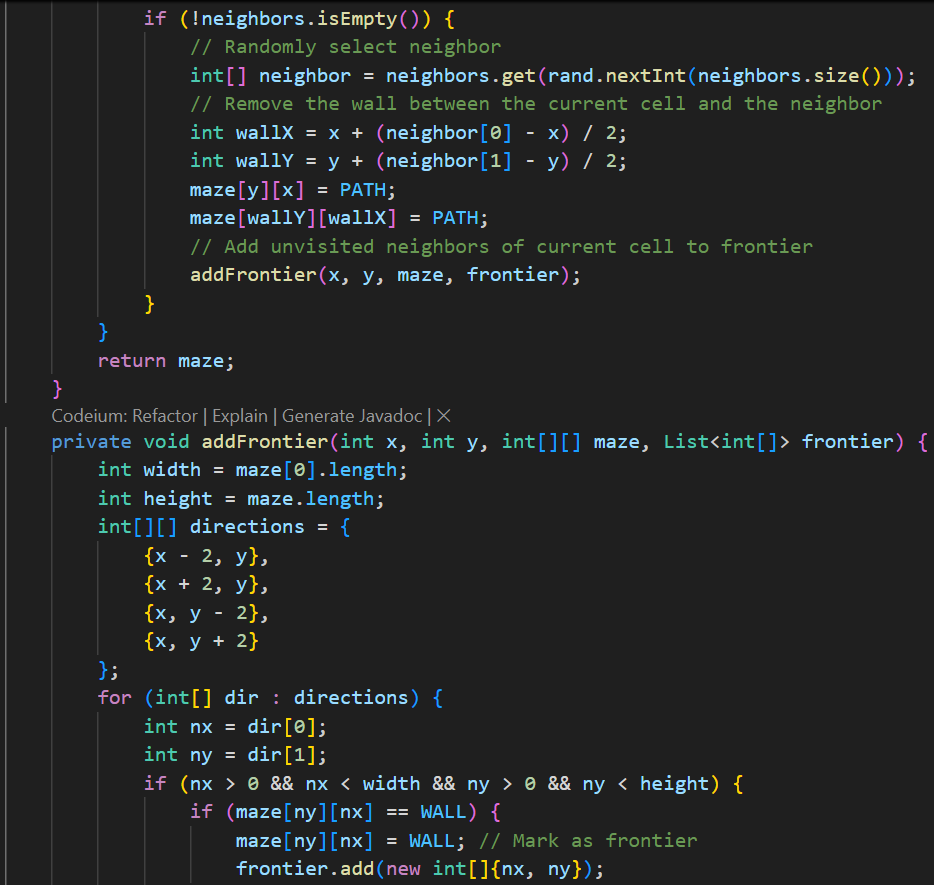
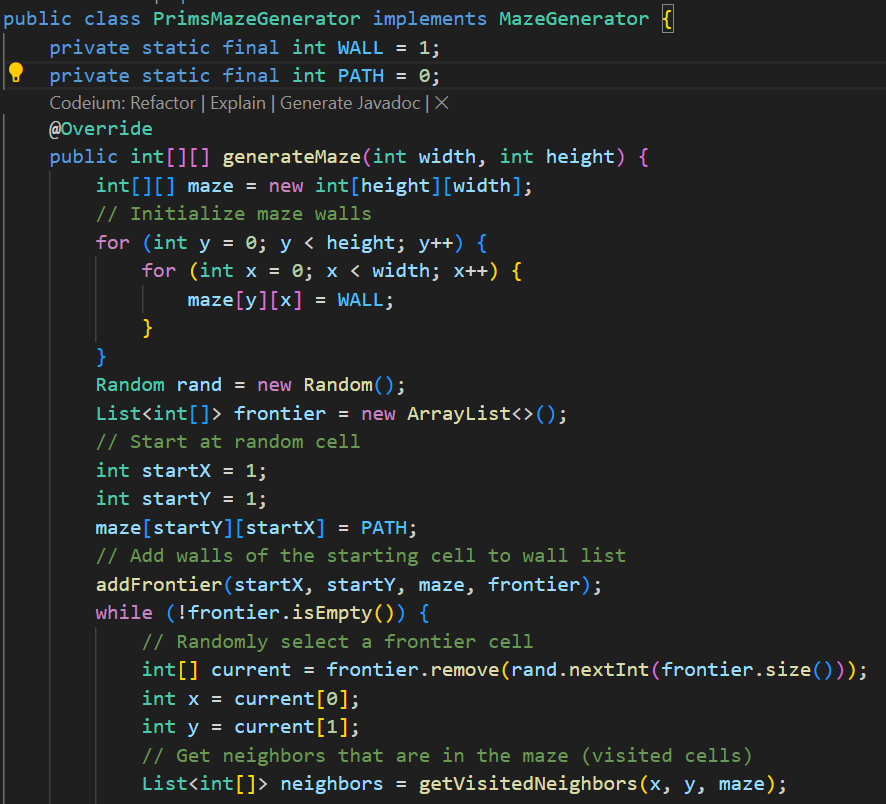
### **Maze**

#### **Maze Generation Algorithms**

##### **Prim's Algorithm**

**Description:** Prim's algorithm builds the maze by starting from an initial cell and iteratively adding adjacent frontier cells, ensuring that the maze remains a connected spanning tree without cycles.

**Implementation Highlights:**

****

****

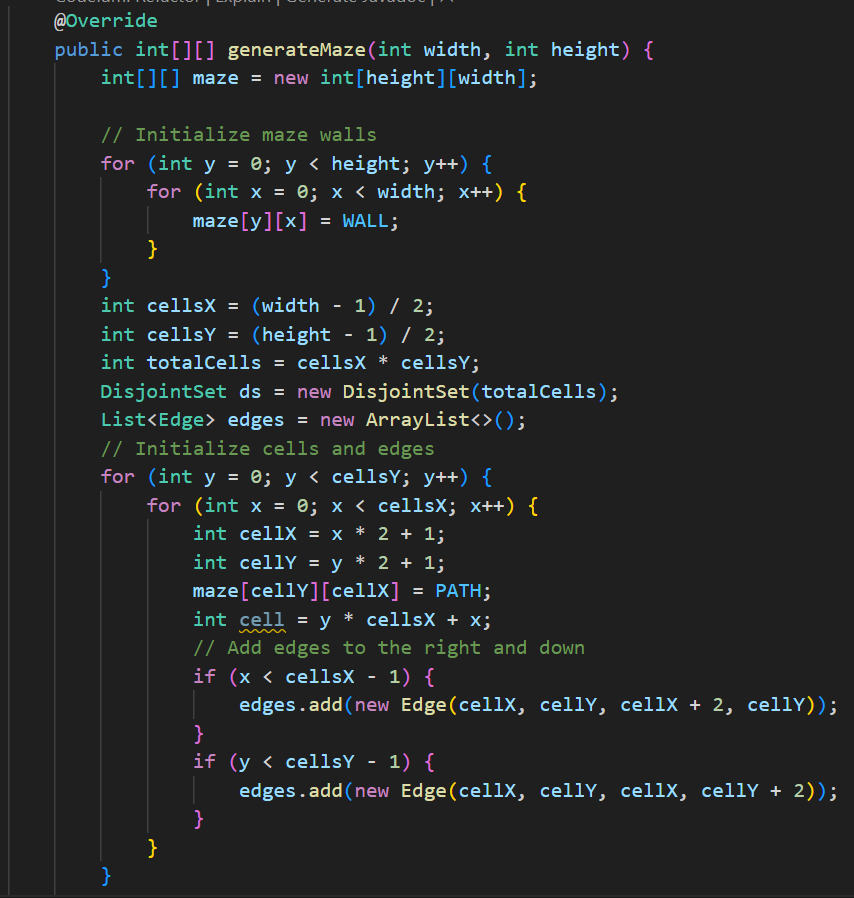
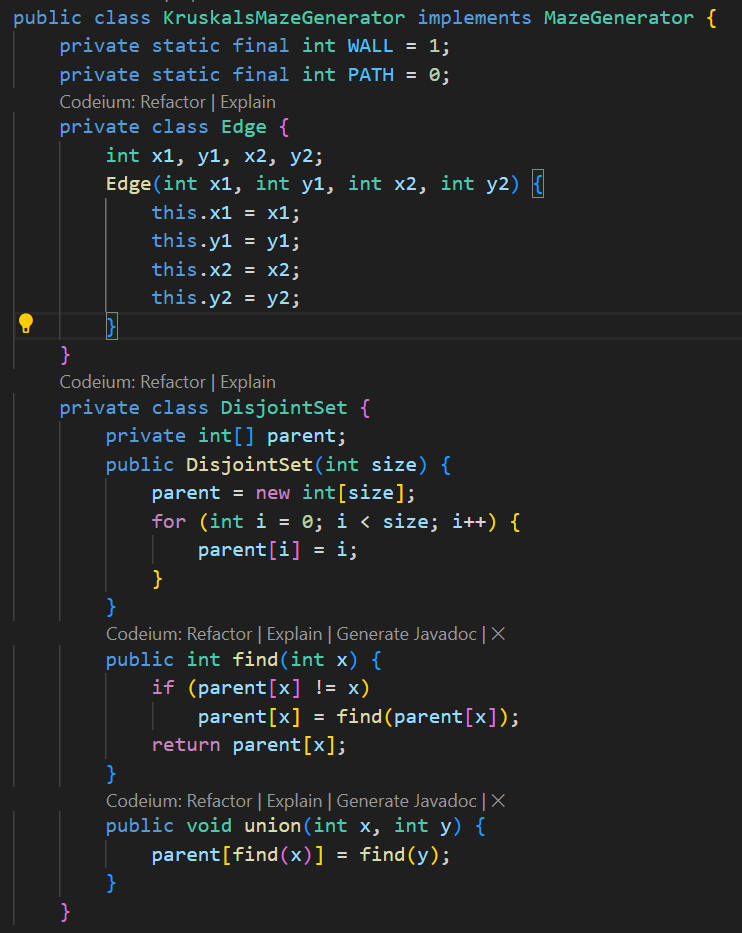
**Frontier List:** Maintains cells adjacent to the current maze. Cells are randomly selected from the frontier to ensure maze variability.

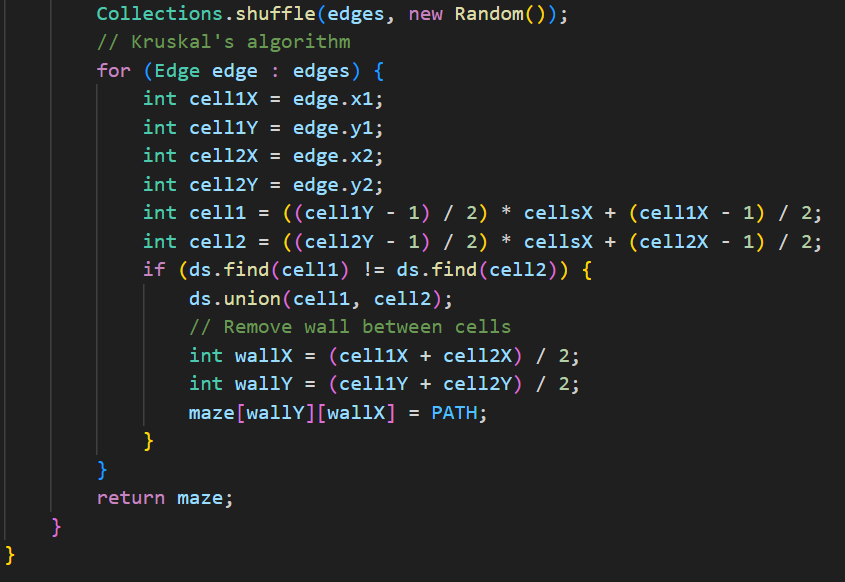
**Wall Removal:** When adding a new cell, the wall between the current cell and the selected frontier cell is removed, creating a passage.

##### **Kruskal's Algorithm**

**Description:** Kruskal's algorithm generates the maze by treating each cell as a separate set and merging sets by removing walls between adjacent cells, ensuring no cycles are formed.

**Implementation Highlights:**

****

****

**Edge List:** All possible walls (edges) between cells are listed and shuffled to introduce randomness.

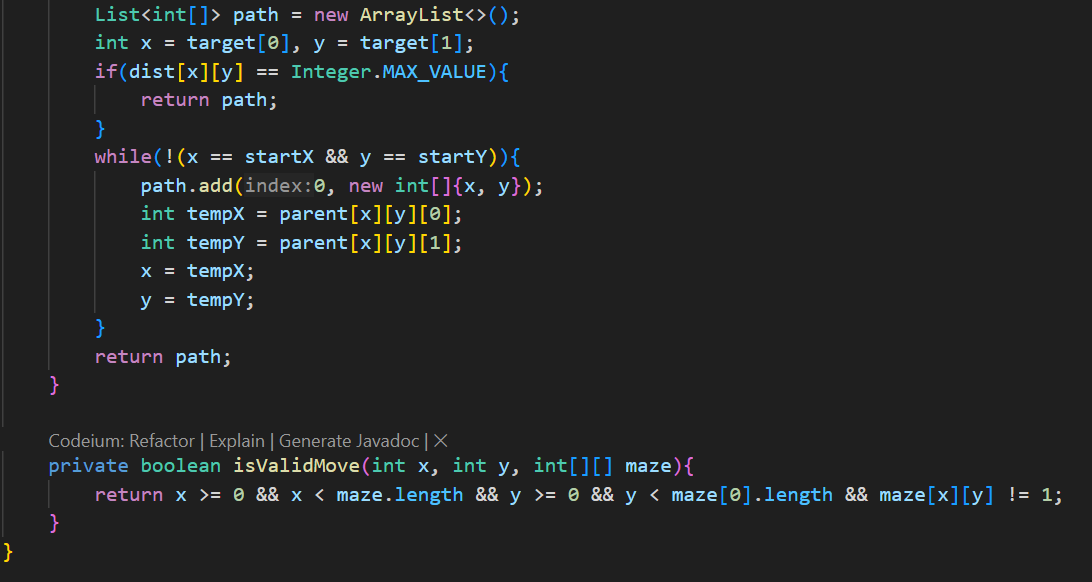
**Disjoint Set (Union-Find):** Ensures that adding a wall removal does not create a cycle by checking if two cells belong to different sets before merging.

#### **Pathfinding Algorithms**

##### **Dijkstra's Algorithm (Bot\_D)**

**Description:** Dijkstra's algorithm finds the shortest path from a starting node to a target node in a graph with non-negative edge weights. It systematically explores the nearest unvisited node until the target is reached.

**Implementation Highlights:**

****

**Distance Array (dist):** Stores the shortest known distance from the start to each cell.

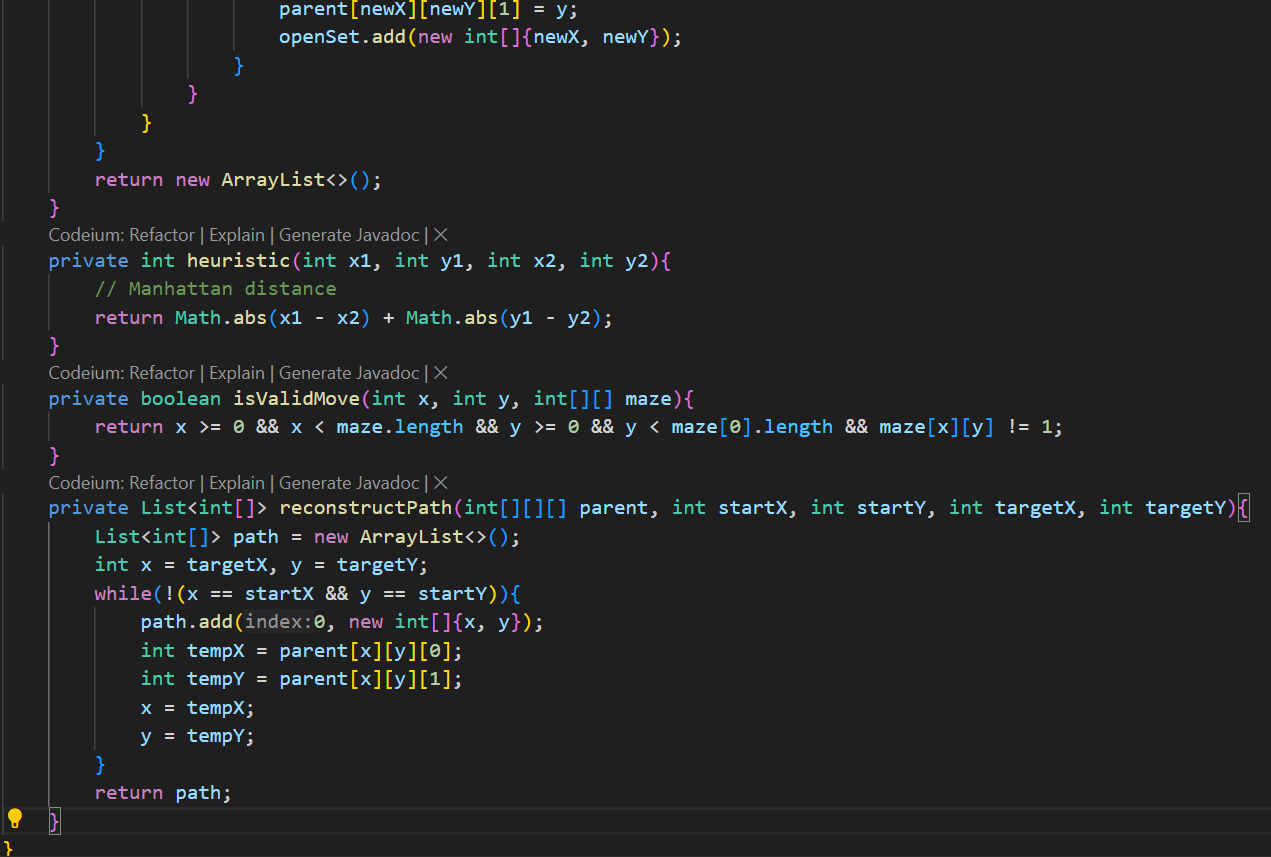
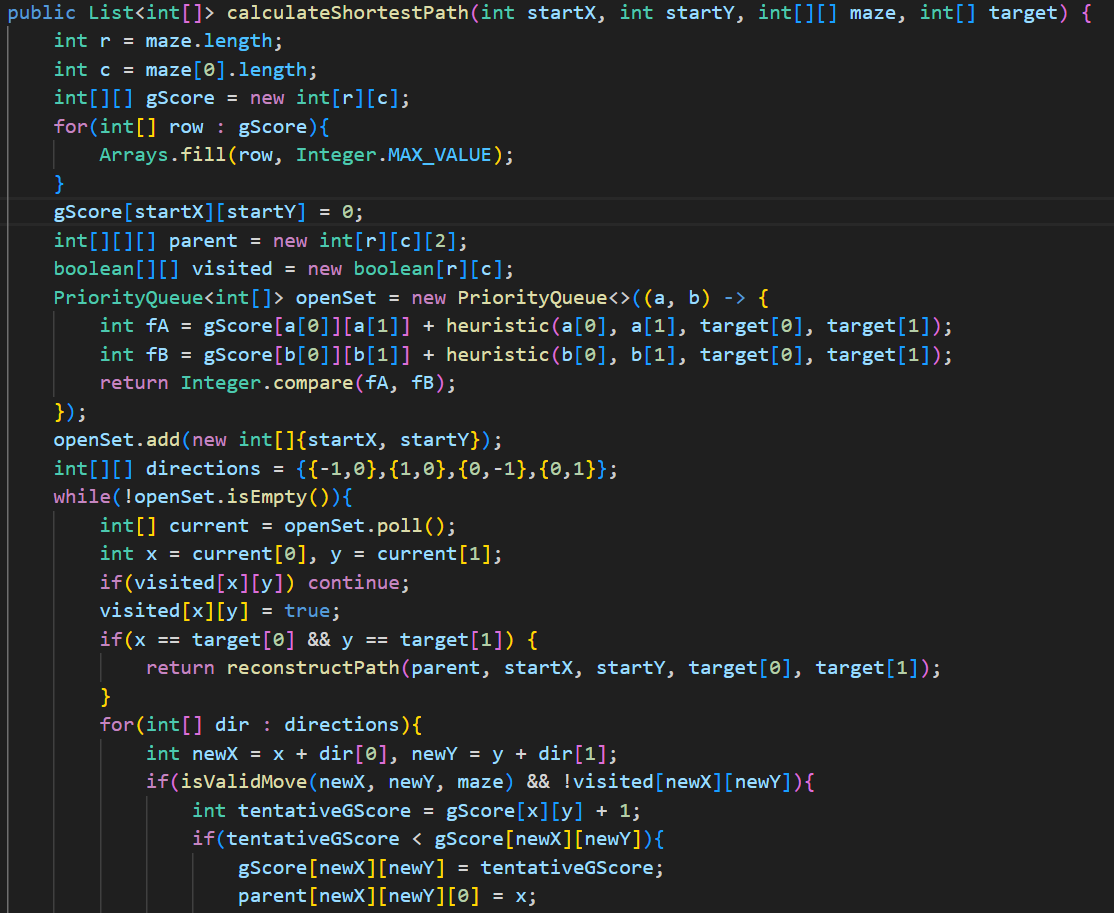
**Priority Queue:** Prioritizes cells with the smallest tentative distance.

**Parent Array:** Tracks the path by storing predecessors for path reconstruction.

##### ***A (A-Star) Algorithm (Bot\_A)*\***

**Description:** A\* algorithm enhances Dijkstra's by incorporating heuristics to estimate the cost from a node to the target, thereby optimizing the search process and reducing computation.

**Implementation Highlights:**

****

**gScore Array:** Similar to Dijkstra's dist, stores the cost from the start to each cell.

**Heuristic Function:** Uses Manhattan distance to estimate the cost from the current cell to the target.

**Priority Queue:** Orders cells based on the sum of gScore and heuristic (fScore), prioritizing nodes closer to the target.

## **Complexity Analysis**

### **Tetris**

#### **Fisher-Yates Shuffle**

**Purpose:** Randomly shuffles the Tetrominoes in the Mino Bag to ensure an unbiased distribution.

**Time Complexity:**

* + **O(n):** Each element in the array is swapped exactly once.

**Space Complexity:**

* + **O(1):** The shuffling is done in place without requiring additional memory proportional to the input size.

**Justification:**

* + The algorithm iterates through the list of Tetrominoes from the end to the beginning, swapping each element with another randomly selected element that comes before it (including itself). This ensures each possible permutation is equally likely.

#### **Collision Detection**

**Purpose:** Determines if a Tetromino can move or rotate without intersecting existing blocks or exceeding play area boundaries.

**Time Complexity:**

* + **O(k):** Where k is the number of blocks in a Tetromino (constant, typically 4).

**Space Complexity:**

* + **O(1):** Uses a fixed number of variables to track collision states.

**Justification:**

* + Each collision check iterates through the Tetromino's blocks and compares their positions against static blocks and boundaries. Given that the number of blocks per Tetromino is constant, the complexity remains constant.

#### **Line Clearing and Scoring**

**Purpose:** Identifies and removes fully occupied lines, updates the score, and adjusts game difficulty accordingly.

**Time Complexity:**

* + **O(m \* n):** Where m is the height of the play area (number of rows) and n is the width (number of columns).

**Space Complexity:**

* + **O(1):** Utilizes a fixed amount of additional memory regardless of the play area size.

**Justification:**

* + The algorithm scans each row to count the number of blocks. If a row is fully occupied, it removes the blocks and adjusts the positions of the remaining blocks. Since each cell is visited exactly once, the time complexity is proportional to the total number of cells.

#### **Hold Mechanic**

**Purpose:** Manages the storage and retrieval of Tetrominoes that the player chooses to hold.

**Time Complexity:**

* + **O(1):** Push and pop operations on the stack are constant-time operations.

**Space Complexity:**

* + **O(p):** Where p is the number of Tetrominoes held (typically 1, but implemented as a stack for potential extensions).

**Justification:**

* + The stack data structure allows for efficient management of held Tetrominoes, ensuring that operations do not depend on the number of elements held.

### **Maze**

#### **Maze Generation Algorithms**

##### **Prim's Algorithm**

**Time Complexity:**

* + **O(V + E log V)**, where V is the number of vertices (cells) and E is the number of edges (walls).
  + In a grid-based maze, V = (width \* height) and E = 4 \* V approximately.
  + Therefore, Time Complexity ≈ O(V log V).

**Space Complexity:**

* + **O(V)** for storing the maze grid, frontier list, and visited cells.

##### **Kruskal's Algorithm**

**Time Complexity:**

* + **O(E log E)** due to sorting the edges.
  + In a grid-based maze, E ≈ 4 \* V.
  + Therefore, Time Complexity ≈ O(V log V).

**Space Complexity:**

* + **O(E + V)** for storing the edge list, disjoint sets, and the maze grid.

#### **Pathfinding Algorithms**

##### **Dijkstra's Algorithm (Bot\_D)**

**Time Complexity:**

* + **O(V log V + E)**, where V is the number of cells and E is the number of possible moves (edges).
  + In a grid-based maze with 4-directional movement, E ≈ 4 \* V.
  + Therefore, Time Complexity ≈ O(V log V).

**Space Complexity:**

* + **O(V)** for storing distances, parent pointers, and the priority queue.

##### ***A (A-Star) Algorithm (Bot\_A)*\***

**Time Complexity:**

* + **O(V)** for storing gScores, parent pointers, visited states, and the priority queue.**O(V log V + E)** similar to Dijkstra's, but often performs better in practice due to the heuristic guiding the search.
  + The heuristic can reduce the number of nodes processed, especially in large mazes.

**Space Complexity:**

* + **O(V)** for storing gScores, parent pointers, visited states, and the priority queue.

## **Conclusion**

# **Chapter 4. FINAL APP GAME**

**Source code (link to Github):**

|  |
| --- |
| https://github.com/Alexspector123/DSA-Project.git |