OOP PROJECT REPORT VALIYYADIN ALIYEV,FUAD ALIZADA,NADIR ASKAROV

Introduction

This project implements a simple maze game where the player navigates through a grid-like maze, interacting with various objects such as keys, chests, doors, and monsters. The game logic is modularized through an interface and multiple classes representing different game objects, each with specific behaviors triggered by player interactions. The game maintains the player's state, including position and health, and provides a graphical interface for visualization. Interaction feedback is displayed using dialog boxes, and the game responds dynamically to the player's actions within the maze.

Specificaton:

1. CurseOfTheMaze() constructor

Sets up the JFrame window and layout.

Initializes objectMap by placing unique game objects (Key, Chest, Door, Monster1, Monster2) based on the maze layout.

Adds a key listener to handle player movement and interactions.

Updates the player's position, checks collisions, and triggers interactions with objects.

Displays win message on reaching the exit.

2. Door.onPlayerInteract(int x, int y)

Checks if player has the matching key for this door's ID.

If yes, opens (removes) the door from the maze and allows player to move through.

If no, shows a message that the matching key is needed and blocks movement.

3. Key.onPlayerInteract(int x, int y)

Adds this key's unique ID to the player's collected keys list.

Removes the key from the maze after picking it up.

Shows a message confirming the key pickup.

4. MazeData.movePlayer(int newX, int newY)

Moves the player’s position in the maze from old coordinates to new ones.

Updates the maze array to reflect the player’s new position ('P') and clears the old one ('.').

5. MazeData.isValidMove(int x, int y)

Checks if coordinates are inside the maze boundaries and not a wall ('#').

Returns true if the move is valid, false otherwise.

6. MazeData.getMonsterType(int x, int y)

Returns a string identifying the monster type at given coordinates ("M1" for M, "M2" for B).

Returns empty string if no monster.

7. MazePanel.MazePanel() constructor

Loads images for player, keys, chests, monsters, and doors from resources.

If image loading fails, sets image variables to null.

8. MazePanel.paintComponent(Graphics g)

Draws the entire maze grid on the panel, cell by cell.

Paints walls, player, keys, chests, monsters, doors with colors or images.

Shows player info (health, keys collected, monsters killed) at the top.

9. Monster1.onPlayerInteract(int x, int y)

If player health > 30, reduces health by 30, removes monster, increments kill count, and lets player move.

Otherwise, shows death message and exits the game.

10. Monster2.onPlayerInteract(int x, int y)

If player health > 20, reduces health by 20, removes monster, increments kill count, and lets player move.

Otherwise, shows death message and exits the game.

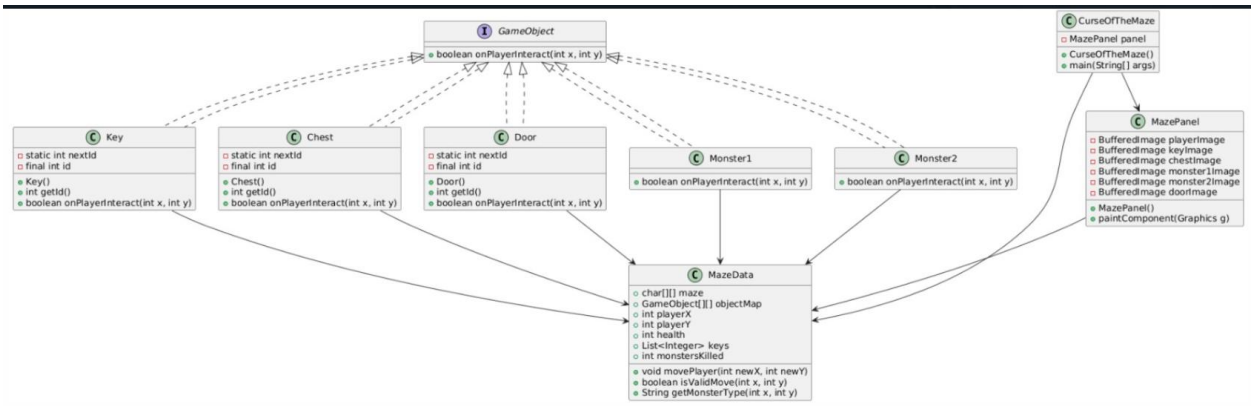
11. Chest.onPlayerInteract(int x, int y)

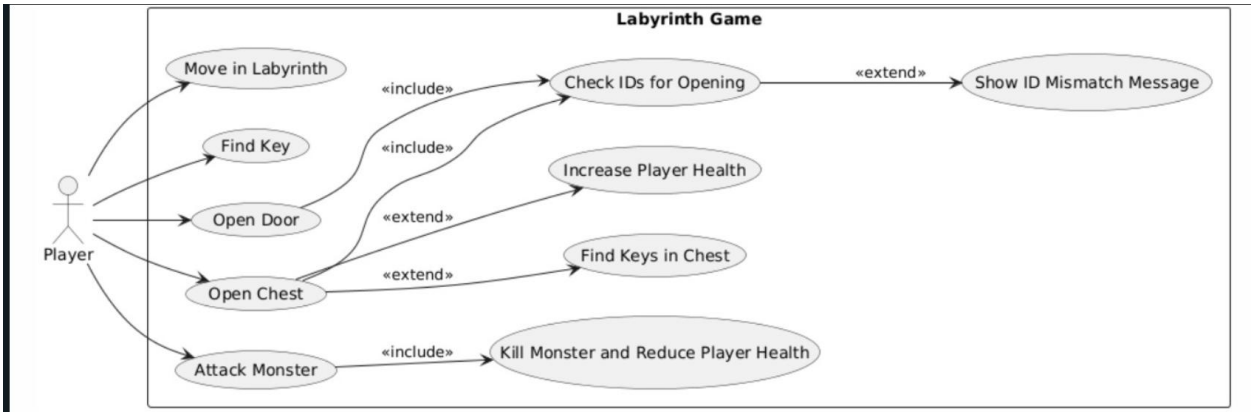
Checks if player has a key matching the chest's ID.

If yes, chest opens: player gains random keys and health, chest removed from maze.

If no, shows locked message and blocks movement.

DESIGN:





TECHINCAL CHOICES:

Frameworks and Libraries

Swing

The graphical user interface (GUI) uses Java Swing, a built-in GUI toolkit in Java. Swing is lightweight and provides components like JFrame, JPanel, and event listeners, allowing you to build windowed desktop applications with custom rendering and input handling.

Java AWT

Used alongside Swing for event handling (KeyEvent, KeyAdapter) and basic graphics (Graphics, Toolkit). AWT provides low-level support for graphical and event-driven programming.

javax.imageio.ImageIO

Used for loading images (player, keys, doors, monsters, etc.) which enhances the visual appeal beyond simple colored shapes.

Architectural Decisions

Separation of Concerns

The project is structured around different classes representing distinct responsibilities:

CurseOfTheMaze handles the main game window and user input.

MazePanel handles rendering the maze and UI elements.

MazeData acts as a centralized data model holding maze state, player position, keys, health, and objects.

Individual game objects (Door, Key, Chest, Monster1, Monster2) implement a common interface (GameObject) and encapsulate their own interaction logic.

Object-Oriented Design

Each interactive entity in the maze implements the GameObject interface, allowing polymorphic interaction logic via the onPlayerInteract() method. This makes it easy to add or modify behavior for new types of objects without changing the core game logic.

Game State Management

The maze grid (MazeData.maze) holds the base map, while a parallel objectMap holds references to GameObject instances. This decouples static map layout from dynamic interactive objects, simplifying updates and interactions.

Event-Driven Input Handling

Player movement is handled by Swing's key event listeners, reacting immediately to arrow keys and updating game state accordingly.

Resource Loading

Images are loaded from resources for visual representation, with fallbacks to simple shapes if images are missing. This enhances user experience while maintaining robustness.

Simple Game Mechanics Embedded in Object Methods

Damage calculation, key collection, door opening, and chest rewards are handled inside each object's interaction method, keeping the main game loop clean and delegating responsibility appropriately.

Possible Enhancements Influenced by This Architecture

Extendability

Thanks to the GameObject interface and separate classes, new interactive objects or enemies can be added easily by implementing the interface and adding their logic.

Maintainability

The clear separation of GUI rendering, game state, and interaction logic helps maintain and update the codebase.

Sure! Here's a discussion about the programming language,frameworks, libraries, and architectural decisions that influenced your maze game implementation:

The entire project is written in Java, a versatile, platform-independent, object-oriented language well-suited for desktop applications. Java's mature ecosystem and rich standard libraries make it ideal for GUI development and game logic implementation.

Frameworks and Libraries

The graphical user interface (GUI) uses Java Swing, a built-in GUI toolkit in Java. Swing is lightweight and provides components like JFrame, JPanel, and event listeners, allowing you to build windowed desktop applications with custom rendering and input handling.

Used alongside Swing for event handling (KeyEvent, KeyAdapter) and basic graphics (Graphics, Toolkit). AWT provides low-level support for graphical and event-driven programming.

Used for loading images (player, keys, doors, monsters, etc.) which enhances the visual appeal beyond simple colored shapes.

The project is structured around different classes representing distinct responsibilities:

* CurseOfTheMaze handles the main game window and user input.
* MazePanel handles rendering the maze and UI elements.
* MazeData acts as a centralized data model holding maze state, player position, keys, health, and objects.
* Individual game objects (Door, Key, Chest, Monster1, Monster2) implement a common interface (GameObject) and encapsulate their ow interaction logic.

* *Object-Oriented Design*

  Each interactive entity in the maze implements the GameObject interface, allowing polymorphic interaction logic via the onPlayerInteract() method. This makes it easy to add or modify behavior for new types of objects without changing the core game logic.

* *Game State Management*

  The maze grid (MazeData.maze) holds the base map, while a parallel objectMap holds references to GameObject instances. This decouples static map layout from dynamic interactive objects, simplifying updates and interactions.

* *Event-Driven Input Handling*

  Player movement is handled by Swing's key event listeners, reacting immediately to arrow keys and updating game state accordingly.

* *Resource Loading*

  Images are loaded from resources for visual representation, with fallbacks to simple shapes if images are missing. This enhances user experience while maintaining robustness.

* *Simple Game Mechanics Embedded in Object Methods*

  Damage calculation, key collection, door opening, and chest rewards are handled inside each object’s interaction method, keeping the main game loop clean and delegating responsibility appropriately.

Possible Enhancements Influenced by This Architecture

* *Extendability*

  Thanks to the GameObject interface and separate classes, new interactive objects or enemies can be added easily by implementing the interface and adding their logic.

* *Maintainability*

  The clear separation of GUI rendering, game state, and interaction logic helps maintain and update the codebase.

* *Decoupling and Testability*

  Although tightly coupled in this small project, the use of interfaces and encapsulation lays groundwork for potential future refactoring toward more modular, testable components.
