Task description
5. Custom game You can develop your own game idea. The minimum criterie is that your game should satisfy the "Common requirement" and it has to have at least one database table, where the highscore of the players are saved. Moreover, it has to be able to read back the saved results from the database, and show them on the screen (rows can be filtered like e.g. top 10 scores).

My idea:

Description of Mechanics

Forest Escape is a simple top-down game played on a 20×20 grid. The player moves through a forest, collects mushrooms, avoids a wolf, and tries to finish the level without losing all lives. Below is a brief explanation of how the main systems work.

Movement and Controls

The player moves using the W, A, S, and D keys. Each key press moves the player one tile up, down, left, or right, as long as the tile is not blocked. The game uses Swing key bindings so movement works even if other interface elements are selected.

Level Structure

Each level is stored in a text file containing 20 lines of 20 characters. Every character represents a tile type such as ground, obstacles, items, the campfire, or starting positions for the player and the wolf. When a level is selected and the game starts, this file is read and converted into the internal map.

Obstacles and Tile Types

The map contains several types of blocking tiles: trees, rocks, and bushes. These tiles cannot be crossed by either the player or the wolf. Ground tiles are walkable. The campfire marks the respawn point.

Collectible Items and Power-Ups

There are several interactable tiles:

- Mushrooms, which increase the player’s score.
- Speed power-ups, which slow down the wolf for a few seconds.
- Invisibility power-ups, which prevent the wolf from harming the player for a short time.

- Extra life tiles, which increase the player’s life count up to a limit.

When the player steps on one of these tiles, it is removed and its effect is applied immediately.

Wolf Behavior

The wolf moves automatically every second. It moves straight in its current direction until it hits an obstacle; if blocked, it chooses a new random direction from the available paths. If the wolf reaches the player or moves directly next to them, the player loses one life unless invisibility is active.

Player Lives and Respawn

The player starts with three lives. When attacked, the player loses one life and respawns at the campfire if lives remain. When lives reach zero, the game ends and the player's score is recorded.

Fog of War

The game uses limited visibility: the player can only see tiles within three tiles around their position. All other areas of the map appear dark, which makes navigation trickier.

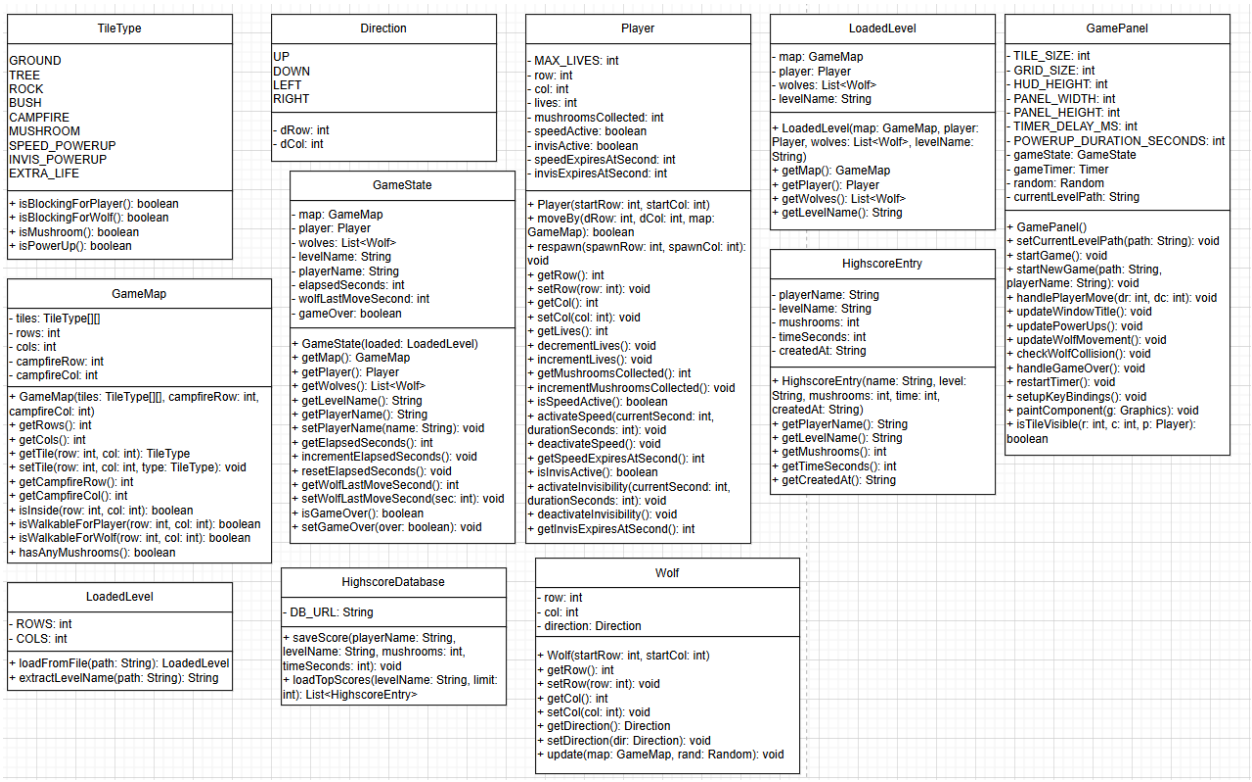
Win and Loss Conditions

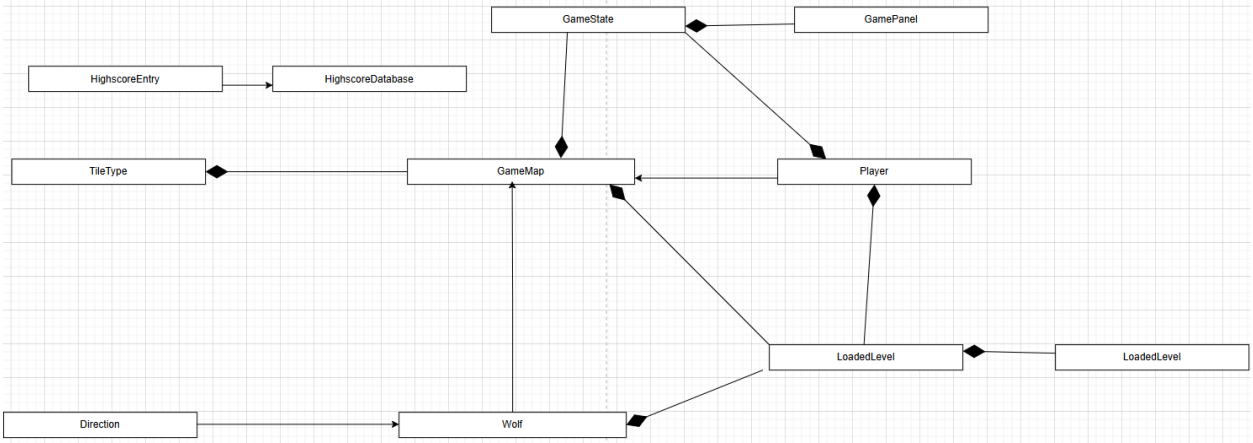
A level is completed when the player collects all mushrooms on the map. The level restarts after completion. The game is lost when the player runs out of lives.

Highscore Recording

After losing, the game asks the player for their name and saves their score (mushrooms collected and time survived) into an SQLite database. The top ten scores can be viewed from the menu.

UML class





Analysis

This chapter explains the structure of the program, the classes used, how they interact, and the main algorithms that control the game. Forest Escape is built with an object-oriented approach and uses several small, focused classes that work together to create the full gameplay.

Program Structure

The program is divided into three main parts:

User Interface Layer (Swing)

- **MainFrame**
  The main window created with NetBeans GUI Builder. It contains the user interface elements such as the level selector, Start Game button, and menu items.
  Its main responsibility is to display the GamePanel and forward user actions (Start Game, New Game, Highscores).

- **GamePanel**
  A custom Swing panel responsible for drawing the game world and handling gameplay. It manages the timer, keyboard input, rendering, and updating the game state.

Game Logic Layer (Core Classes)

- **GameState**
  Stores the current map, the player, the wolf, the timer values, and game flags (such as game over or elapsed time).

- **Player**
  Stores the player's position, lives, collected mushrooms, and power-up status. Contains movement logic for the player.

- **Wolf**
  Stores the wolf's position and direction. Contains the logic that controls how the wolf moves and chooses new directions.

- **GameMap**
  Stores the 2D tile array for the level. Provides functions for checking walkability, retrieving tiles, and detecting whether mushrooms remain.

Data and Loading

- **TileType**
  An enum describing the type of each tile (ground, tree, rock, bush, mushroom, power-ups, campfire).

- **Direction**
  An enum describing the four movement directions used by the wolf.

- **LevelLoader**
  Reads a 20×20 text file and converts it into a GameMap and starting positions.

- **LoadedLevel**
  A small class that bundles the map and starting positions into one object.

Highscores

- **HighscoreManager**
  Handles saving and loading scores using SQLite.
- **HighscoreEntry**
  A simple data class that represents one score record.

Class Interactions

- MainFrame creates a GamePanel and embeds it into the window.
- When the player starts a new game, GamePanel calls LevelLoader to load the selected level.
- LevelLoader returns a LoadedLevel which is used to create a new GameState.
- GamePanel uses GameState to access the Player, Wolf, and GameMap.
- The javax.swing.Timer inside GamePanel updates the wolf, timers, and UI.
- When the game ends, HighscoreManager saves the score.
- The Highscores menu reads scores through HighscoreManager.

This structure keeps the code organized and prevents logic from being mixed with UI code.

Data Representation

Map Representation

The map is stored as a 20×20 array of TileType. Each tile in the level file is mapped to a tile type:

- Blocking tiles (trees, rocks, bushes)
- Special tiles (campfire, mushrooms, power-ups)
- Empty ground

Characters like P and W do not remain in the array; they are replaced with GROUND and their coordinates are stored separately.

Player and Wolf Positioning

Both entities store their current row and column. The map does not contain player or wolf tiles; they are drawn directly on top of the map in GamePanel.

Algorithms Used in the Game

Player Movement

Player movement checks:

1. Compute new row/column.
2. Check map.isWalkableForPlayer()
3. If walkable, update row/column.
4. Check for mushrooms or power-ups and apply effects.

Wolf Movement

The wolf:

1. Attempts to move forward in its current direction.
2. If blocked, checks all four directions to find possible new directions.
3. Picks a random valid direction and moves once.
4. If no valid direction exists, it stays still.

Collision Detection

After each movement:

- Compute distance between player and wolf using Manhattan distance.
- If the distance is 0 or 1, the player takes damage unless invisibility is active.

Fog of War

A tile is visible if:

- max(abs(playerRow - tileRow), abs(playerCol - tileCol)) <= 3
  This is called Chebyshev distance.

Level Completion

The game checks whether any mushrooms remain by scanning the map.
If none remain:

- Show “Level Cleared” message
- Restart the same level

Respawn Logic

If the player has at least one life:

- Move player to campfire coordinates
- Continue the game

If lives reach zero:

- Game ends
- Highscore is recorded

Event Flow

The main events in the game occur in this sequence:

1. User selects a level.
2. User presses “Start Game”.
3. MainFrame tells GamePanel to load the selected level.
4. GamePanel loads the map and initializes GameState.
5. The timer starts.
6. Every second:
    - Time increases
    - Wolf may move
    - Power-ups may expire
    - Fog of war and screen are updated
7. User moves with W, A, S, D.
8. Player interacts with items and obstacles.
9. Collision checked after each move.
10. When level ends or player dies:
    - Show result dialog
    - Save score
    - Load same level again or return to UI

Implementation Details

This section explains the most important parts of the implementation.

Level Loading

Levels are stored as simple text files.
LevelLoader reads the file line by line and converts each character into a TileType.
The start positions for the player and the wolf are also read from the file.
The result is wrapped in a LoadedLevel, which is passed into GameState.

This makes level editing simple and allows easy debugging.

Game Loop

The game uses a Swing Timer that runs once every second.
Each tick updates:

- elapsed time
- wolf movement
- power-up expiration
- collision checks
- window title

This timer controls the entire game progression.

Rendering

GamePanel draws everything in its paintComponent method:

1. HUD
2. Tiles
3. Fog of war
4. Player and wolf

Everything is drawn using basic Graphics2D operations.

Player Movement

Movement uses Swing key bindings.
Each key press calls handlePlayerMove, which:

1. Checks collisions
2. Moves the player

3. Applies power-ups
4. Removes mushrooms
5. Checks whether the level is completed

Wolf Behavior

The wolf tries to move forward. If blocked, it picks a random new direction that is walkable. This gives simple but unpredictable wandering, making the game challenging.

Fog of War

The player can only see tiles within a distance of 3.
Everything else is darkened.
This is checked with Chebyshev distance.

Highscore Handling

When the player dies, the game shows a dialog asking for a name.
The system stores:

- player name
- mushrooms collected
- time survived

Then the database is updated and the top ten scores can be displayed from the menu.

Event Handling

Event	**Component**	**Handler Method**
User selects a level from dropdown	JComboBox levelComboBox	Updates currentLevelPath in GamePanel
User presses Start Game	JButton startGameButton	gamePanel.startGame()
User chooses New Game	MenuItem "New Game"	gamePanel.startGame()
User opens Highscores	MenuItem "Highscores"	Shows highscores dialog
User presses Exit	MenuItem "Exit"	handleExitConfirmation()

User opens How to Play	MenuItem "How to Play"	Shows instructions
User opens About	MenuItem "About"	Shows about dialog

Event	Component	Handler
W key press	Key Binding	handlePlayerMove(-1,0)
A key press	Key Binding	handlePlayerMove(0,-1)
S key press	Key Binding	handlePlayerMove(1,0)
D key press	Key Binding	handlePlayerMove(0,1)
Timer tick (1 second)	Swing Timer	Updates wolf, time, power-ups
Player moves	GamePanel	Checks items, collisions, level complete
Wolf moves	GamePanel	Triggered inside timer loop
Game Over	GamePanel	Shows dialog, records score

Test cases

Test Case 1: Valid Level File

Condition: Level file is correctly formatted (20×20).
Action: Start game with this level.
Expected Result:

- Game loads without errors.
- Player, wolf, and campfire positions are set correctly.
- Tiles match the content of the file.

Test Case 2: Missing Player Position

Condition: Level contains no 'P'.
Action: Start game.
Expected Result:

- Player spawns at campfire tile.

Test Case 3: Missing Wolf Position

Condition: Level contains no 'W'.
Action: Start game.
Expected Result:

- Wolf spawns at fallback position (1,1).

Test Case 4: Malformed Level Size

Condition: Level file has fewer than 20 lines or short lines.
Action: Start game.
Expected Result:

- Error dialog is shown (“Level Load Error”).

Test Case 5: Walkable Ground

Condition: Player stands next to a ground tile.
Action: Press movement key.
Expected Result:

- Player moves into the tile successfully.

Test Case 6: Blocked Tile

Condition: Next tile is TREE, ROCK, or BUSH.
Action: Press movement key.
Expected Result:

- Player does not move.

Test Case 7: Map Boundary

Condition: Player is at the edge of the map.
Action: Move outside the map.
Expected Result:

- Player remains in place.

Test Case 8: Wolf Moves Forward

Condition: Wolf's forward tile is walkable.
Action: Timer ticks.
Expected Result:

- Wolf moves one step in the same direction.

Test Case 9: Wolf Hits Obstacle

Condition: Forward tile is blocked.
Action: Timer ticks.
Expected Result:

- Wolf chooses a new random direction and moves into it.

Test Case 10: Wolf Surrounded

Condition: All neighboring tiles are blocked.
Action: Timer ticks.
Expected Result:

- Wolf stays in place.