RAVENSBURGER Lotti Karotti

1st Sujung Lee
Matriculation Number: 1365537
Frankfurt University of Applied Sciences
Frankfurt, Germany
sujung.lee@stud.fra-uas.de

2nd Huynh Thi Kim Quynh
Matriculation Number: 1584541
Frankfurt University of Applied Sciences
Frankfurt, Germany
thi.huynh@stud.fra-uas.de

3rd Duong Bao Ngoc
Matriculation Number: 1589562
Frankfurt University of Applied Sciences
Frankfurt, Germany
bao.duong@stud.fra-uas.de

***Abstract*—This report describes the planning and execution of the old version of the board game project Lotti Karotti, created by a three-person team using the Java programming language. With JavaFX's intuitive graphical user interface, the project aimed to build an enjoyable game. The main issues were achieving perfect synchronization between the GUI and game logic, flexible game flow, and effective implementation of game rules. The development method included research, task organization, brainstorming ideas, and collaborating using version control and bug-tracking technologies. Technical and design problems are solved in detail, emphasizing the tools and algorithms employed. The result is an innovative, fascinating game that maintains the attraction of the original board game. This project provides significant insights into game development, GUI design, and teamwork dynamics, creating the path for future improvements like new features or multiplayer capabilities.**

I. Introduction

Board games have long been a popular form of entertainment as well as a technique to improve our cognitive abilities. When we talk about board games, we mean anything that involves moving, placing, or removing pieces from a marked board. Board games are wonderful teaching instruments, particularly for children's cognitive development. They improve critical thinking, problem-solving ability, and mathematics skills via enjoyable play. When children play these games, they improve their counting skills, scorekeeping, and strategic decision-making abilities. Playing together also teaches children how to share, wait their time, and follow the rules. This digital adaption of Lotti Karotti illustrates that traditional board games can be supported with technology to generate more complete and effective education tools for youngsters.
Creating a Java board game, such as Lotti Karotti, teaches us how to create games and how players interact with them. In the game, players move their pieces on a set path, aiming to get to the end while avoiding obstacles and traps. Players click a special button that randomly creates holes on the 2D game board, causing random holes that players must avoid. For instance, games such as Chess or Snakes and Ladders ask for the use of rules, taking turns, and dealing with chance, which are usual problems in making board games. Furthermore, new board games increasingly contain computer-controlled opponents, options for many players, and user interfaces with graphics, among other features that call for a mix of smart planning and coding skills. By evaluating and handling these problems throughout the production of Lotti Karotti, this project not only aims to recreate the game's main features but also to investigate methods to improve its functionality and user experience using efficient Java programming approaches. This report demonstrates how to build the Lotti Karotti board game with Java. It focuses on a functional and enjoyable game environment. The document aims to go over each aspect of this project. We start with a thorough explanation of the game's rules, how it works, and what you need to win, with clear examples provided. Next, we discuss previous research, specifically approaches and processes related to producing board games, such as JavaFX and effective group strategies. This report outlines possible ways to construct a game, such as how to choose multiplayer, GUI how the game looks, UML diagram, and display stages in simple code.

II. Problem Description

Lotti Karotti is an entertaining game for players of all ages depending on planning, luck, and lively action. The goal is to move all four colorful bunnies (green, yellow, red, and purple) up to the juicy carrot at the top of the hill before their rivals do. But be careful, the racetrack is filled with surprises and challenges, as the ground beneath the bunnies is unstable and may vanish at any time, causing the rabbits to fall into hidden holes and disappear from the game.
The game takes place on a colorful 3D board with a spiraling path that leads to the carrot on top of the hill. Each player starts with a set of four bunnies in the color they were assigned. The number of steps that the player's rabbit move depend on the card shows, and they must wisely choose how to move their bunnies forward. There are two types of cards: Bunny cards and Rabbit cards [1].

- Bunny card:
  - Rabbit cards give players the ability to update their bunny's position by several steps.
  - They can either move an existing rabbit that is already on the board or add a new one to the game.
- Carrot card:
  - Players of carrot cards are instructed to hit a unique carrot button at the top of the hill, which will cause a random area on the path to collapse.
  - Any rabbit that is standing in the collapsing area is eliminated from the game after falling into a hole. A player is removed from the game if all four of their bunnies fall into holes.

For instance, if a player obtains a Bunny card that says ”three steps”, they can select which bunny to advance while still following the guidelines of avoiding occupied areas. When the player draws a carrot card, the player must click the carrot button, which leads to an unplanned twist that could cause a hole in the path and remove their own or a competitor’s rabbit. Once the player’s rabbit gets close to the carrot (1,2, or 3 steps away), if the bunny card appears, that player wins the game, no matter how many steps. The game only ends when a bunny reaches the carrot or when all players are eliminated, in that case, there isn’t a winner. The game is unexpected and exciting because of these dynamic interactions.

Fig. 1: Board game view

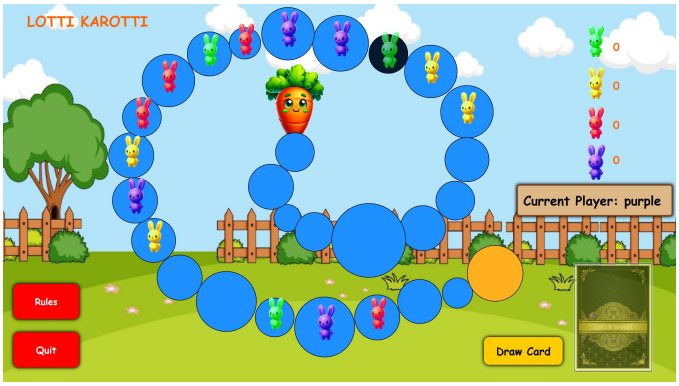

Fig. 2: Rabbit falls into the hole

III. Related Work

A. Algorithmic Strategies in Luck-Based Board Games: Lotti Karotti and Its Alternatives

Several classic board games, including Lotti Karotti, Pop-Up Pirate[2], Don’t Break the Ice[3], and Gino Pilotino[4], share common game mechanics such as randomness, suspense, and interactivity. These elements can be implemented in Java using object-oriented programming(OOP), randomization techniques, and event-driven logic. A key feature of these games is randomness, which determines unpredictable outcomes. In Lotti Karotti, trapdoors open at random positions, similar to Pop-Up Pirate, where the pirate jumps when a randomly assigned sword slot is triggered. In Java, this can be handled using the Random class to assign probabilities to specific actions. Don’t Break the Ice works similarly by having ice blocks collapse based on their structural connections, which can be modeled using grid-based data structures. In Gino Pilotino, obstacles appear unpredictably, which can be implemented using randomized movement patterns. Suspense is another important factor. In Lotti Karotti, players do not know when a trapdoor will open, just like in Pop-Up Pirate, where the pirate jumps at an unexpected moment. In Java, event-driven programming can be used to delay outcomes, for example, by using Timer or ScheduledExecutorService to create suspenseful pauses before revealing a game-changing event. In Don’t Break the Ice, the suspense builds as more ice blocks are removed, which can be simulated using collision detection in Java’s physics libraries. Gino Pilotino creates suspense by making players react quickly to obstacles, which can be implemented using real-time event listeners. Interactivity is another common feature in these games. Lotti Karotti uses a turn-based system, where players draw cards and move accordingly. This can be implemented in Java using class-based models, where players, tiles, and movements are separate objects interacting through game logic. Pop-Up Pirate and Don’t Break the Ice rely on player actions triggering physical changes, which can be simulated in Java with object interactions and conditional logic. Gino Pilotino requires real-time user input, which Java handles with event listeners for keyboard or mouse controls. From a Java game development perspective, these games can be built using simple class structures, random event triggers, and interactive game loops. Java frameworks like LibGDX or JavaFX Canvas API provide tools for creating smooth animations, handling player input, and managing game logic. By applying these programming techniques, digital versions of these board games can maintain the same fun and excitement as their physical counterparts.

B. Applying Evolutionary Algorithms to Lotti Karotti

Lotti Karotti is a board game with strong randomness, making it difficult to apply fixed strategies. Players must move their rabbits toward the goal while avoiding trapdoors that appear unexpectedly. Because of this uncertainty, traditional rule-based approaches are ineffective. Instead, evolutionary algorithms(EAs)[5] can help AI adapt and improve its strategy over time. An evolutionary algorithm mimics natural selection by testing different strategies, selecting the most successful ones, and refining them through mutation (random changes) and crossover (combining strategies). In Lotti Karotti, AI can start with random movement strategies, play multiple games, and gradually learn which decisions lead to better results. Over time, this approach helps AI avoid traps and optimize movement. Additionally, Monte Carlo Tree Search (MCTS)[6] can improve decision-making by simulating multiple possible moves and selecting the best one. Instead of following a fixed plan, MCTS allows AI to explore different options and choose safer paths based on probabilities. This approach helps AI improve its gameplay through experience, much like how

human players learn from trial and error. By analyzing past moves and outcomes, AI can make better choices in future games. This learning process allows AI to gradually develop more effective strategies without needing pre-programmed rules.

C. Key Algorithms in Lotti Karotti

The Lotti Karotti game integrates multiple algorithms to ensure fairness, randomness, smooth movement, and efficient game state management. The turn management algorithm is implemented using a round-robin scheduling mechanism within the PlayerController and Player classes, ensuring equitable turn rotation while skipping players with no valid moves. Randomization algorithms[7] introduce variability, with the Card class utilizing random sampling for card selection and the Board class generating holes dynamically through probabilistic methods. Pathfinding and movement algorithms determine valid rabbit movements based on board constraints and card values, ensuring legal navigation. The goal detection algorithm monitors rabbit positions and triggers the win condition upon reaching the carrot. Validation algorithms are distributed across the system, verifying player actions in the Bunny, PlayerController, and Player classes to enforce game rules and manage resources correctly. Game state management algorithms initialize and maintain dynamic components, configuring board elements, paths, and players while detecting win conditions. Animation algorithms [8], implemented using JavaFX, employ timeline-based motion control, translation, scaling, and fading effects to enhance interactivity. Resource management algorithms utilize thread-safe structures such as ConcurrentHashMap to maintain real-time synchronization of rabbit positions in multiplayer scenarios. Event-handling algorithms process user interactions using event-driven logic within the PlayerController and Bunny classes, allowing responsive input handling for gameplay actions. Preloading algorithms in the WinnerView class optimize asset loading to reduce latency and improve transition speed. Visualization and feedback algorithms apply opacity transitions, shadow effects, and dynamic color updates to indicate game events and enhance player engagement. The combination of these algorithms enables a seamless, dynamic, and interactive gameplay experience. Their modular implementation ensures maintainability and scalability while preserving fairness and unpredictability in Lotti Karotti's game mechanics.

IV. Team Work

Our project team is a collaboration of three persons, each with responsibility and unique skills that contribute to the success of the project. After discussion, we divided tasks for each member based on their talent and knowledge. Member Sujung worked as the principal coder, developing the game logic and ensuring a precise and seamless game structure. Member Quynh and Ngoc were tasked with designing the game's graphical user interface (GUI) in JavaFX, debugging code, and ensuring a consistent user experience. Collaborative assignments, such as integrating the GUI with the game logic, were distributed across the whole group.

To ensure effective communication, we regularly texted and attended online meetings to discuss progress, handle difficulties, and discover solutions. Critical decision-making and design revisions were carried out in person in class meetings. Our team uses GitHub to manage versions, enabling code collaboration, sharing, and bug tracking. Besides that, Google Drive is utilized to save shared documents, notes, and pictures. During brainstorming sessions in class, we analyzed new ideas and came up with alternative solutions together, thus enhancing the team's creativity. This systematic approach established a productive working environment, allowing us to successfully resolve issues and ensure that all members had a voice in important decisions. Building strong teams and leveraging each member's capabilities helped us meet project deadlines and produce high-quality results.

V. Proposed Approaches

The proposed approach introduces algorithms that establish a structured and efficient framework for implementing the Lotti Karotti game. These algorithms systematically define the input parameters, processing logic, and expected outputs, ensuring a well-organized and seamless game flow. By modularizing the game mechanics, the approach enhances clarity, maintainability, and scalability while ensuring correct execution of game logic.

A. Game Initialization

This algorithm sets up the Lotti Karotti game before it starts. It takes the number of players as input and initializes the game board, paths, and player rabbits. Each player is assigned a rabbit color and a starting number of rabbits. The game begins with Player 0, and the interface displays the current player. This setup ensures the game starts in a clear and organized state.

Algorithm 1 Initialize Game State

Require: TotalPlayers {Number of players in the game}
Ensure: Initialized game state with board, players, paths, and rabbits

```
Initialize Board
Initialize Paths as List
for each PathID in range(1 to TotalPaths) do
    Add Path to Paths
end for
for each PlayerIndex in range(0 to TotalPlayers - 1) do
    Assign Rabbit Color (e.g., Green, Yellow, Pink, Purple)
    Set Initial Rabbit Count
end for
Set CurrentPlayer ← 0
Display CurrentPlayerLabel
        ("Current Player: Green")
```

B. Player Turn Management

This algorithm ensures that the game moves to the next player who can take a turn. It checks each player in order to see if they have rabbits to move or add. If a valid player is found, their turn begins, and the game updates the display. If no valid players are available, an alert is shown. The algorithm also resets the card-drawing status, ensuring smooth and fair gameplay.

Algorithm 2 Player Turn Management

Require: CurrentPlayer, TotalPlayers, RabbitCounts
Ensure: Updated CurrentPlayer
InitialPlayer ← CurrentPlayer
HasValidPlayer ← False
repeat
  Increment CurrentPlayer by 1 (modulo TotalPlayers)
  Reset CardDrawn Status
  PlayerColor ← GetPlayerColor(CurrentPlayer)
  RabbitCount ← CheckRabbitCounts(PlayerColor)
  HasRabbitsToAdd ← CheckPlayerHasRabbitsToAdd(PlayerColor)
  if $RabbitCount > 0$ **or** $HasRabbitsToAdd$ **then**
    HasValidPlayer ← True
    break
  end if
  if $CurrentPlayer = InitialPlayer$ **then**
    break
  end if
until False
if NOT $HasValidPlayer$ **then**
  ShowAlert("No Valid Players")
  return
end if
Display("Current Player: " + PlayerColor)
ResetCardView()

C. Rabbit Movement

This algorithm moves a rabbit based on the number of steps drawn from a card. It first checks if the rabbit has a valid position. If not, an error is shown. Next, it calculates the next position. If the rabbit reaches the winning spot, the player wins. Otherwise, the rabbit moves forward. If it lands on a hole, it is removed. Otherwise, its position is updated. The game then switches to the next player's turn, ensuring smooth gameplay.

Algorithm 3 Rabbit Movement

Require: Rabbit, CardSteps
Ensure: Updated Rabbit Position
CurrentPath ← RabbitPositionMap[Rabbit]
if CurrentPath is NULL **then**
  Print "Rabbit has no current path!"
  return
end if
NextPath ← GetNextPath(CurrentPath, CardSteps)
if NextPath is NULL **then**
  TriggerWin(Rabbit)
  return
end if
Animate Rabbit Movement from CurrentPath to NextPath
if NextPath is a Hole **then**
  FadeOutAndRemoveRabbit(Rabbit)
else
  Update RabbitPositionMap[Rabbit] ← NextPath
end if
Update PlayerTurn

D. Random Card Drawing

This algorithm lets a player draw a random card from the deck. If a card was already drawn this turn, an alert appears, preventing another draw. If allowed, the algorithm selects a random card, displays it with an animation, and triggers a special effect if it's a carrot. This ensures randomness and fairness in the game.

Algorithm 4 Random Card Drawing

Require: CardDeck (Array of Cards)
Ensure: Displayed Card and Updated Game State
if CardAlreadyDrawn **then**
  ShowAlert("Card Already Drawn", "You can only draw once per turn!")
  return
end if
RandomIndex ← Generate Random Integer in range(0 to Length(CardDeck) - 1)
SelectedCard ← CardDeck[RandomIndex]
Animate Card (Translation and Rotation)
Display SelectedCard on CardView
if SelectedCard is Carrot **then**
  ShakeCarrot()
end if
Mark CardAsDrawn

E. Random Hole Creation

This algorithm randomly creates holes on the game board, adding unpredictability. First, it resets all paths to their default color. Then, it selects a random number of paths and turns them into holes by changing their appearance. If a rabbit is on a selected path, it is removed from the game. This makes the game more challenging and strategic.

Algorithm 5 Random Hole Creation

Require: Paths, RabbitPositionMap
Ensure: Updated Board with Holes and Removed Rabbits
for each Path in Paths **do**
  Set Path Color to Default (Blue)
end for
HoleCount ← Random Integer in range(1 to MaxHoles)
SelectedPaths ← Randomly Select HoleCount Paths from Paths
for each Path in SelectedPaths **do**
  Animate Path Color to Black
  if RabbitPositionMap contains a Rabbit on Path **then**
    FadeOutAndRemoveRabbit(Rabbit)
  end if
end for
Update PlayerTurn

F. Validate Rabbit Action

This algorithm ensures that a rabbit move is valid before allowing the action. It checks if the rabbit belongs to a player, if it is the current player's turn, and whether a card has been drawn. If any condition is not met, an alert is shown, and the move is rejected. If the conditions are sat- isfied, the algorithm confirms the action as valid, allowing the player to proceed. This ensures fair and rule-based gameplay.

Algorithm 6 Validate Rabbit Action

Require: Rabbit
Ensure: Boolean (True if action is valid, False otherwise)
RabbitPlayer ← GetRabbitPlayer(Rabbit)
if RabbitPlayer == -1 **then**
  Print "Unknown rabbit."
  return False
end if
if CurrentPlayer ≠ RabbitPlayer **then**
  ShowAlert("Invalid Turn", "It's not " + GetPlayerColor(RabbitPlayer) + "'s turn!")
  return False
end if
if NOT CardDrawn **then**
  ShowAlert("Invalid Action", "Please draw a card first.")
  return False
end if
if Card is Carrot **then**
  ShowAlert("Carrot Card", "Click on the carrot to proceed.")
  return False
end if
return True

G. Win Detection

The Win Detection Algorithm checks if any rabbit has reached the winning position. It goes through all rabbits in the RabbitPositionMap and compares their position with the WinningPosition. If a match is found, the algorithm triggers a win state using TriggerWin and stops checking further. This ensures the game recognizes a win as soon as possible, making it efficient.

Algorithm 7 Win Detection

Require: RabbitPositionMap, WinningPosition
Ensure: Triggered Win State
for each Rabbit in RabbitPositionMap **do**
  if Rabbit Position == WinningPosition **then**
    TriggerWin(Rabbit)
    break
  end if
end for

H. Set Winner

The Set Winner Algorithm updates and animates the winning rabbit's image. It sets the WinnerId as the current winner, updates the rabbit's image using UpdateRabbitImage, and plays a congratulatory animation with PlayCongratulationAnimation. This ensures the winner is visually highlighted.

Algorithm 8 Set Winner

Require: WinnerId, RabbitImages, RabbitImageView
Ensure: Winner's Rabbit Image Displayed and Animated
Set CurrentWinnerId ← WinnerId
Call UpdateRabbitImage(WinnerId, RabbitImages, RabbitImageView)
Call PlayCongratulationAnimation(RabbitImageView)

I. Congratulatory Animation

The Congratulatory Animation highlights the winning rabbit with a gold DropShadow and three animations: scaling, rotation, and fading. The rabbit grows and shrinks, tilts left and right, and fades in and out, each repeating 4 times. All effects play simultaneously, creating a celebratory visual.

Algorithm 9 Congratulatory Animation

Require: RabbitImageView
Ensure: RabbitImageView Animations (Scaling, Rotation, Fade)
Apply DropShadow Effect to RabbitImageView (Color: GOLD, Radius: 10)
Create ScaleAnimation:
  ScaleX, ScaleY Transition: [1 → 1.2 → 1]
  Duration: 1 Second
  Repeat: 4 Times
Create RotationAnimation:
  RotateProperty Transition: [0° → 10° → -10° → 0°]
  Duration: 1.5 Seconds
  Repeat: 4 Times
Create FadeAnimation:
  Opacity Transition: [1 → 0.5 → 1]
  Duration: 1 Second
  Repeat: 4 Times
Play ScaleAnimation, RotationAnimation, FadeAnimation Simultaneously

J. Restart Game

The Restart Game algorithm switches the game to the main menu. It runs SceneChanger.ChangeScene to load the MainView scene, resetting the game. This lets players restart the game without closing the application.

Algorithm 10 Restart Game

Require: None
Ensure: Transition to MainView Scene
1: Call SceneChanger.ChangeScene("View/MainView.fxml")

K. Close Application

The algorithm shuts down the game. It calls System.Exit(0), which immediately closes the application. No input is required, and it ensures that the program stops running completely.

Algorithm 11 Close Application

Require: None
Ensure: Application Closed
1: Call System.Exit(0)

VI. Implementation Details

A. Application Structure

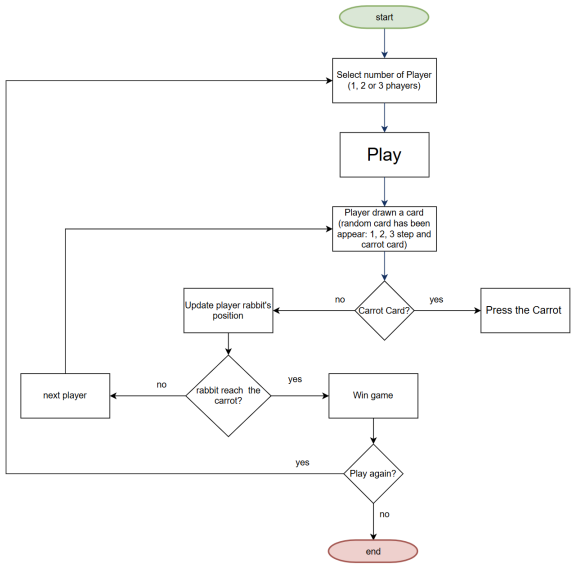

Fig. 3: Flowchart of Lotti Karotti game

Here are the easy steps to play Lotti Karotti. Firstly, the user chooses the number of players (1, 2, or 3) and hit the "Play" button. Each player gets 4 rabbits at the beginning. In each turn, the player draws a card. It can show 1, 2, or 3 steps, or a carrot card randomly. If you get steps, move your rabbit. If you get a carrot, press the carrot on the center of the board which creates a hole randomly. The first player's rabbit reaches the carrot, and that player wins. After the win, the player can choose to play again or stop. If no rabbit reaches the carrot, the turn shifts to the next player, and the game continues.

B. GUI Details

The GUI, or graphical user interface, was created with the Java FX framework. The GUI's objective was to offer a simple and easy-to-use user interface for communicating with the platform. Below, the UI is explained in detail with the help of screenshots.

Fig. 4: Main View

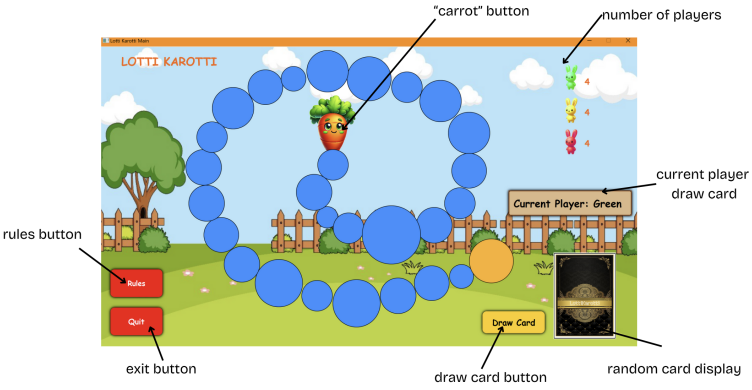

Fig. 5: Board View

Fig. 6: Winner View

C. UML Class Diagram

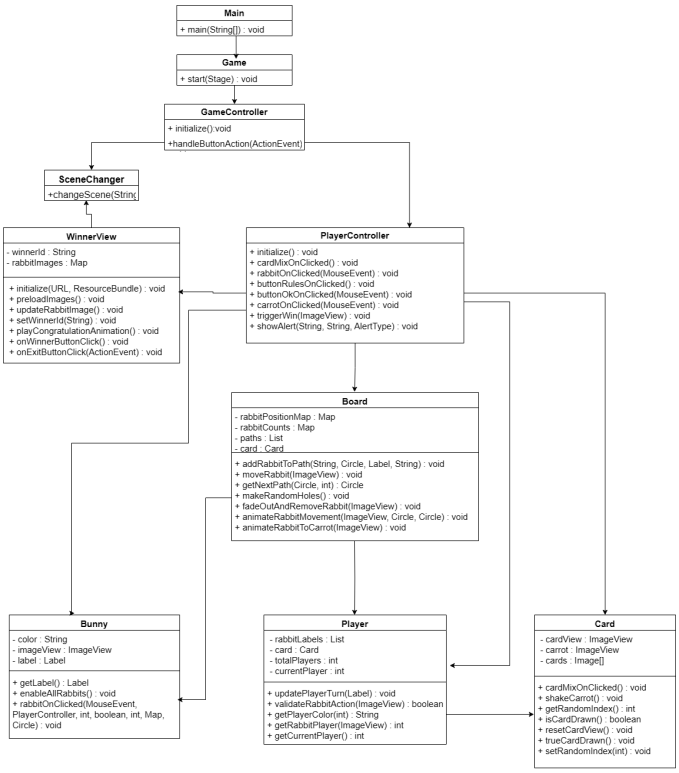

Fig. 7: UML Class Diagram

D. Used Libraries

Implementing the Lotti Karotti game relies on a combination of JavaFX APIs, standard Java libraries, and custom utility classes. These APIs work together to create a dynamic and interactive graphical user interface while handling game logic, animations, and state transitions seamlessly [9].

JavaFX serves as the core framework for the game's GUI and event-driven functionalities. Core UI components such as 'Button', 'Label', and 'ImageView' are employed to create interactive elements, where buttons facilitate actions like restarting or exiting the game, labels display dynamic information such as the current player's turn or rabbit counts, and 'ImageView' is used to render visual elements like rabbits and cards. Layout and graphical elements, including 'AnchorPane' and 'Circle', provide the structural foundation for positioning game components and representing paths on the board.

Event handling is achieved using JavaFX's 'MouseEvent', capturing user interactions such as clicks on rabbits and cards, which trigger game logic. JavaFX animation utilities are extensively used to create smooth, visually appealing effects. Animations such as rabbit movements, card rotations, and celebratory winner displays are implemented using classes like 'Timeline', 'KeyFrame', 'KeyValue', and transitions such as 'TranslateTransition', 'RotateTransition', and 'FadeTransition'. Visual effects, including 'DropShadow' and 'Color', add emphasis to UI elements, such as highlighting the winning rabbit. The 'Duration' class ensures precise timing for these animations, making transitions consistent and responsive.

The standard Java libraries complement JavaFX by managing backend functionality. Collections such as 'Map', 'HashMap', 'List', and 'ArrayList' organize and manage data efficiently. For instance, 'HashMap' maps player colors to their corresponding rabbit images, while 'List' dynamically manages paths and labels. The 'Random' class introduces stochastic elements like random card drawing and hole generation on the board, enhancing gameplay variability. Resource handling is facilitated by the 'URL' class, which locates assets like images and FXML layouts, while the 'IOException' class handles errors that may occur during resource loading. The use of 'ConcurrentHashMap' ensures thread-safe management of shared resources, such as rabbit positions, which is particularly important for maintaining stability in multiplayer scenarios.

A custom utility class, 'SceneChanger', facilitates scene transitions. It abstracts and simplifies navigation logic, enabling seamless transitions between the main game view and the winner screen. Its modular design enhances reusability and maintainability.

The APIs are distributed across several key game components. The 'PlayerController' integrates JavaFX components like 'Button', 'MouseEvent', and 'Label' to manage player interactions, update the game state and control UI behaviors. The 'Board' class utilizes JavaFX shapes such as 'Circle' and animation utilities to handle rabbit movements and visualize random events like hole creation. The 'Bunny' class uses JavaFX elements like 'ImageView' and 'Label' to represent rabbits and their labels while managing visibility and interactions. The 'Player' class validates player actions and tracks game state using JavaFX 'Label' and standard Java collections. The 'Card' class employs JavaFX animation utilities for card drawing and integrates the 'Random' class to implement randomness in gameplay. The 'WinnerView' class incorporates JavaFX animation tools and visual effects to deliver an engaging and celebratory display for the winner.

Together, the integration of JavaFX and standard Java libraries ensures a robust and engaging gameplay experience. JavaFX provides the tools necessary for designing, animating, and interacting with a visually dynamic GUI, while the standard libraries manage backend operations such as data organization, randomization, and resource handling. This cohesive combination of APIs enables the seamless execution of gameplay mechanics, responsive animations, and interactive user experiences. The modular and scalable design ensures that the game remains maintainable and adaptable for future enhancements.

E. Code Snippets

1) Dynamic Path Initialization

```
1 private void initializePaths() {
2    for (int i = 1; i <= 27; i++) {
3        Circle path = (Circle) anchorPane.lookup
             ("#path" + i);
4        if (path != null) {paths.add(path); }}}
5        //paths.add(1)...paths.add(27)
```

Listing 1: Dynamic Path Initialization

The game board's paths are dynamically initialized during runtime using the initializePaths method. JavaFX Circle objects, representing path nodes, are identified through their unique IDs (path1, path2, etc.) and stored in a list. This dynamic approach allows seamless updates or modifications to the path structure without hardcoding, improving scalability and maintainability.

2) Rabbit Movement

```
public void moveRabbit(ImageView rabbit) {
    if (!playerManager.validateRabbitAction(
        rabbit)) {
        return;
        // Abort move if validation fails
        }

    Circle currentPath = rabbitPositionMap.get(
        rabbit);

    // Validate that the rabbit has a current
        path
    if (currentPath == null) {
        System.err.println("Rabbit has no
            current path!");
        return;
        }

    // Determine the next path based on card
        logic, skipping occupied paths
    Circle nextPath = getNextPath(currentPath,
        card.getRandomIndex());


    if (card.getRandomIndex() > 0) { // Ensure
        movement only happens if steps > 0
        if (nextPath == null) {
         //   animateRabbitToCarrot(rabbit);
            controller.triggerWin(rabbit);
        } else {
            // Normal movement
            animateRabbitMovement(rabbit,
                currentPath, nextPath);
            // Check if the next path is black
            if (nextPath.getFill().equals(javafx
                .scene.paint.Color.BLACK)) {
                // Remove the rabbit if the path
                    is black
                fadeOutAndRemoveRabbit(rabbit);
            } else {
                // Update the rabbit's position
                    if the path is not black
                rabbitPositionMap.put(rabbit,
                    nextPath);
            }
        }
    } else {
        System.out.println("Random index is 0,
            no movement.");
    }

    // Update the turn after the move
    if (card.isCardDrawn()) {
        playerManager.updatePlayerTurn(
            currentPlayerLabel);
        card.setRandomIndex(4);
        //initial randomIndex
    }
}
```

Listing 2: Rabbit Movement

The moveRabbit method is a key function that handles the movement of a player's rabbit based on the drawn card and ensures that the game rules are followed. The method begins by validating the player's action using the playerManager.validateRabbitAction method. This ensures that only valid moves, such as moving the current player's rabbit, are executed. If the validation fails, the method terminates without further processing, enforcing fair play and compliance with game rules.

Next, the method retrieves the rabbit's current position on the game board using the rabbitPositionMap. This map maintains a dynamic association between each rabbit and its corresponding path. If the rabbit is not assigned to a valid path, an error message is logged, and the method terminates. This validation prevents undefined behavior and ensures that all game objects maintain consistent states.

The method then calculates the rabbit's next path using the getNextPath function, which determines the target position based on the card drawn and the current position. The logic ensures that occupied paths are skipped, adhering to the movement rules of the game. If the drawn card's value (random index) is zero, no movement occurs, and the method logs this outcome for debugging purposes.

If the rabbit is eligible to move and a valid nextPath is determined, the method executes the movement logic. For rabbits reaching the final goal, indicated by a null value for nextPath, the controller.triggerWin method is invoked, marking the player as the winner. For normal movements, the method animates the rabbit's transition from the current position to the next path using the animateRabbitMovement method. If the next path is a "hole" (represented by a black color), the rabbit is removed from the game using the fadeOutAndRemoveRabbit method. Otherwise, the rabbit's new position is updated in the rabbitPositionMap.

Finally, if the card has been drawn during the move, the method updates the game state by advancing the turn to the next player using the playerManager.updatePlayerTurn method. It also resets the card's random index to its initial state, ensuring readiness for the next player's turn. This method demonstrates dynamic state management, rule enforcement, and interactive gameplay, serving as a foundation for the game's functionality.

3) Make Random Holes

```java
public void makeRandomHoles() {
    // Reset all paths to blue
    for (Circle path: paths) {
        path.setFill(Color.DODGERBLUE);
    }

    Random random = new Random();
    // Create a set to keep track of selected
        random indices
    Set<Integer> selectedIndices = new HashSet
        <>();

    while (selectedIndices.size() < 1) {
        int randomPathIndex = random.nextInt(
            paths.size());
        selectedIndices.add(randomPathIndex);
    }

    // Change the color of the selected paths to
         black and remove rabbits if present
    for (int index : selectedIndices) {
        Circle path = paths.get(index);

        // Create a color transition animation
        Timeline colorTransition = new Timeline(
                new KeyFrame(Duration.ZERO, new
                    KeyValue(path.fillProperty()
                    , Color.DODGERBLUE)),
                new KeyFrame(Duration.millis
                    (2000), new KeyValue(path.
                    fillProperty(), Color.BLACK)
                    )
        );

       colorTransition.play(); // Play the
           animation

        // Check if any rabbit is on this path
        rabbitPositionMap.entrySet().removeIf(
            entry -> {
            if (entry.getValue() == path) {
                // Remove the rabbit with a fade
                    -out animation
                fadeOutAndRemoveRabbit(entry.
                    getKey());
                return true; // Remove this
                    rabbit from the map
            }
            return false;
        });
    }
    card.trueCardDrawn(); //true;
    playerManager.updatePlayerTurn(
        currentPlayerLabel);
}
```

Listing 3: Make Random Holes

The makeRandomHoles method introduces an element of randomness and dynamic gameplay by converting certain paths into ”holes,” visually represented as black paths. Initially, all paths are reset to their default blue color to ensure a consistent game state. This is achieved by iterating through the paths list and updating the fill property of each Circle object to Color.DODGERBLUE. To introduce holes, the method utilizes Java’s Random class to generate unique indices corresponding to paths. A Set is used to ensure that the selected indices are unique, and the randomness adds unpredictability to the game, enhancing replayability.

Once the random indices are determined, the method changes the color of the corresponding paths to black using JavaFX’s Timeline animation. The transition starts from blue (Color.DODGERBLUE) and smoothly changes to black (Color.BLACK) over a duration of 2 seconds. This animation provides visual feedback to the players, making the gameplay more engaging and interactive.

If any rabbits are present on the newly black paths, they are removed from the game. The method iterates through the rabbitPositionMap, checking for rabbits located on the affected paths. For each match, the fadeOutAndRemoveRabbit method is called to animate the rabbit’s removal. This ensures that players are penalized for landing on a path that becomes a hole.

Finally, the method updates the game’s turn by invoking the playerManager.updatePlayerTurn method. Additionally, it resets the card state to indicate that a valid action has been completed, maintaining the turn-based structure of the game.

The makeRandomHoles method exemplifies event-driven programming by dynamically modifying the game board based on specific actions. Its use of randomness adds variability and excitement, while the smooth animations enhance user experience. By dynamically updating the rabbit positions and resetting the paths, the method ensures fairness and consistency in gameplay. This approach demonstrates effective resource management and reinforces the interactive nature of the game.

VII. Experimental Results, Statistical Tests, Running Scenarios

A. *Statistical tests*

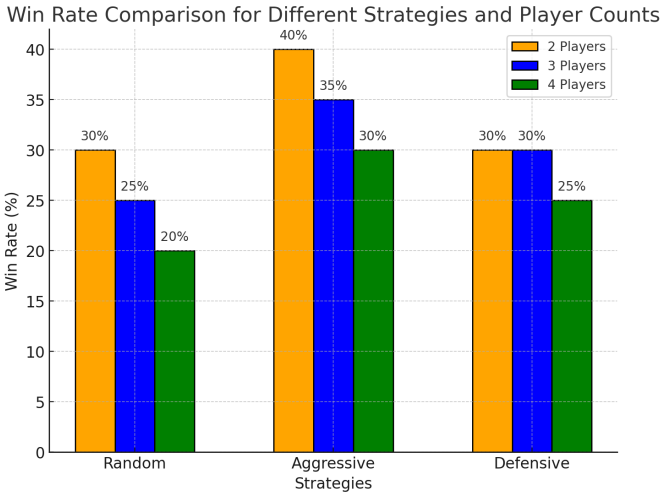

Fig. 8: Bar chart Win rate comparison between strategies with player counts

The bar chart demonstrates the standard deviation of percentages of wins for the three separate approaches (Defensive,

Aggressive, and Random) based on the number of competitors (two, three, and four). The approaches are shown on the X-axis, and the corresponding winning rates are shown on the Y-axis as ratios.

According to the figure, the Aggressive strategy routinely wins the most games, especially when there are just two players (40%) since it has a far higher chance of obstructing the opponent. This method becomes somewhat less effective as the number of participants increases because of the increasing competition and unpredictability. The defensive strategy, on the other hand, is consistent across player counts (30% for two and three players, but lower to 25% for four), suggesting that it performs well under balanced circumstances but falters with additional players. In every scenario, the Random approach has the lowest victory rate, falling from 30% (2 players) to 20% (4 players), confirming the notion that lower results are the result of a lack of strategic forethought[10].

B. Effects on the Strategy

These findings show how crucial it is to consider the game's strategies. A more defensive or balanced approach may work better in larger groupings, although aggressive playstyles are stronger in smaller ones. The results also show that the choice of strategy has a major impact on the game's outcome and that a well-planned strategy increases the likelihood of winning[11].

C. Evaluations

Performance Evaluation

- The game plays smoothly when the number of players is increased to four due to the usage of Java and suitable data structures.
- Each turn has a response time of less than one second, offering a continuous playing experience.

Algorithm Evaluation

- Offensive tactics help players take control of the game, especially when there are few players.
- Defensive tactics are suitable for difficult scenarios because they reduce the risk from random factors.

Players stated that the game is enjoyable because of the random nature of the cards. However, if specific cards, such as "Drop Hole," are used too frequently, the deck may become unbalanced.

The game offers a competitive and enjoyable experience while meeting its original design goals. To improve balance, however, the frequency of special cards must be adjusted.

VIII. Conclusions and Future Work

A. What you have learned

We had an amazing experience with Java. The more we explore this programming language, the more we're amazed by how many opportunities it offers for software development. Compared to other languages like Python, C/C++, or JavaScript, Java has more built-in capabilities and is also simpler to use and recognize. Thanks to its developer-friendly syntax and comprehensive documentation, Java makes it easier for programmers to take advantage of all of its features when creating web apps, mobile apps, or backend services. Due to its multithreading capabilities, Java enables us to achieve amazing heights of efficiency. Java's object-oriented concepts, including abstraction, encapsulation, inheritance, and polymorphism, have changed the way we approach coding. These important OOP properties enable us to create and implement classes and objects that are flexible and reusable and improve code management efficiency. From extending the Java Libraries to utilizing powerful web frameworks, Java developers will be able to leverage Java effectively to build full-fledged front-end, back-end, and web stacks. Java multithreading features have taken us to a new level of efficiency. With OOP principles, classes and objects may be designed and implemented to create modular and manageable code. Our coding processes have improved, resulting in software that exceeds today's standards of performance, scalability, and maintainability.

The project has significantly improved our teamwork experience. We learned so much about working together effectively and managing workflows effectively. The project has significantly improved our teamwork experience. We learned so much about working together effectively and managing workflows effectively. Our code management platform was GitHub, which allowed us to maintain a clear record of any code modifications. GitHub offers many excellent reasons to be the preferred tool for many development teams. Our team was able to perfectly work together on development projects while maintaining complete insight into code modifications. With GitHub's branching mechanism, several developers may work on different features at the same time. The pull request mechanism assisted in carefully reviewing code and guaranteeing quality through comments provided by others. Google Drive was extremely useful in organizing materials and sharing documents, providing version control, and quick access to related materials. This comprehensive toolbox not only enhanced productivity but also generated a strong sense of cooperation, assisting the team in staying on target while submitting outstanding products on time.

B. Ideas for the future development of your application, new algorithms

Future plans for the Lotti Karotti game app could focus on improving the game's feel, improving AI, and adding new features to keep it entertaining. One idea is to help AI adversaries think more clearly by using improved mathematical forms. To make the game interesting, add various levels and events. Players are promoted, have a choice of modes, etc. Additionally, allowing live play allows users to play online with friends. This can include features like private rooms, player lists, adding friends, global leaderboards, and tournament modes where multiple players can participate at the same time. Features like levels and the ability to customize the player's appearance can all add to the fun of the game. To increase player creativity, the game can allow players to change the appearance of their characters, board themes,

backgrounds, etc. Integrating animations and sound effects also increases players experience. These advancements will increase the app's appeal to more players while also improving the game experience.

REFERENCES

[1] "Lotti Karotti Rule," n.d.

[2] "Pop-up Pirate," Wikipedia. Sep. 23, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Pop-up_Pirate&oldid=1247164801 [Accessed: Jan. 31, 2025].

[3] "Don't Break the Ice," Wikipedia. Dec. 22, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Don%27t_Break_the_Ice&oldid=1264545229 [Accessed: Jan. 31, 2025].

[4] "Gino Pilotino," Wikipedia. Sep. 08, 2024. [Online]. Available: https://it.wikipedia.org/w/index.php?title=Gino_Pilotino&oldid=141029556 [Accessed: Jan. 31, 2025].

[5] "Evolutionary algorithm," Wikipedia. Jan. 15, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Evolutionary_algorithm&oldid=1269617352 [Accessed: Jan. 31, 2025].

[6] "Monte Carlo tree search," Wikipedia. Dec. 30, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Monte_Carlo_tree_search&oldid=1266139796 [Accessed: Jan. 31, 2025].

[7] "Random (Java Platform SE 8)," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html. [Accessed: Jan. 24, 2025].

[8] "Javafx.animation (JavaFX 2.2)," [Online]. Available: https://docs.oracle.com/javafx/2/api/javafx/animation/package-summary.html[Accessed: Jan. 24, 2025]

[9] "Java Core Libraries," Oracle Help Center. Accessed: Jan. 24, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/22/core/java-core-libraries1.html

[10] "Lotti Karotti – perfekte Mischung aus Glück, einfachen Regeln und Taktik," Galaxus. Accessed: Jan. 27, 2025. [Online]. Available: https://www.galaxus.de/en/page/lotti-karotti-perfect-mix-of-luck-simple-rules-and-tactics-27181

[11] manaiaclemim, "Lotti Karotti or Funny Bunny!," Tame The Board Game. Accessed: Jan. 27, 2025. [Online]. Available: https://tametheboardgame.com/2016/12/21/lotti-karotti-or-funny-bunny/https://tametheboardgame.com/2016/12/21/lotti-karotti-or-funny-bunny/.