Hick Hack in Gackelwack

Khoi Nguyen Nguyen
Bachelor of Informatik
Ho Chi Minh, Vietnam
khoi.nguyen@stud.fra-uas.de

Tran Quoc Dat Nguyen
Bachelor of Informatik
Ho Chi Minh, Vietnam
dat.nguyen-tran-quoc@stud.fra-uas.de

Duc Minh Khoa Ngo
Bachelor of Informatik
Ho Chi Minh, Vietnam
duc.ngo@stud.fra-uas.de

***Abstract*—This report presents the design, development and implementation of a Java-based digital adaptation of the renowned board game *Hick Hack in Gackelwack*. The project aims to recreate gameplay experience while leveraging object-oriented programming principles to ensure scalability, maintainability and user experience. The implementation features a modular architecture that separates game logic, user interface and data management, allowing for ease of extension and debugging. Core challenges addressed include simulating the game's strategic depth, implementing the game's algorithm and integrate it into the user interface. Usability testing with continuous development and debugging in order to ensure players' satisfaction. The resulting application successfully replicates the game's mechanics and offers a platform for further exploration. This project demonstrates the potential of Java for creating robust and engaging digital board game adaptations.**

I. Introduction

Board games have long been a source of entertainment and strategic challenge, fostering social interaction and critical thinking. In recent years, digital adaptations of board games have gained popularity, offering players the convenience of virtual play and the opportunity to explore new ways of interacting with classic games. This paper focuses on the development of a Java-based digital version of Hick Hack in Gackelwack, a board game that combines elements of strategy, probability, and player interaction.

The objective of this project is to replicate the mechanics of the original game while enhancing accessibility and user interaction via a digital platform. The development process employed object-oriented programming (OOP) principles to ensure a modular and maintainable codebase. Core features include an intuitive user interface in the front end and a robust game logic engine in the back end. Additionally, the project explored challenges such as implementing randomized events, managing game state transitions, and simulating the unique dynamics of the game's decision-making process.

This paper begins by outlining the rules and mechanics of Hick Hack in Gackelwack to provide a foundation for understanding the implementation. It then details the system architecture, development methodology, and key design decisions. Finally, the paper evaluates the project's outcomes, including its fidelity to the original game, usability, and areas for potential improvement. By showcasing the capabilities of Java for creating digital board game adaptations, this project highlights the intersection of technology and traditional gaming culture.

II. Problem Description

A. General

Hick Hack in Gackelwack is a funny-themed simultaneous card selection board game created by Stefan Dorra [1]. In this game, players will aim to gather the most points either by playing their poultries to feed on corn or play foxes to munch on others' poultries. In the end, the player with the most wealthy stockpile wins. Overall, this is a great game for kids as well as adults to have some fun.

B. Components

The game includes:

- 6 Farm Tiles (each with a distinct color)
- 78 Corn Cubes (26 for each color: green, blue, yellow)
- 60 Cards (Poultries of values -2, 3, 3, 4, 4, 5, 6 and Foxes of values 4, 5, 6 for each of the 6 colors of the corresponding Farm)
- 1 standard 6-sided Die

C. Setup

Deal 5 cards to each player and place a random corn cube on each farm.

D. Gameplay

Each turn is made up of 3 steps:

1) Pick a card: Every player choose a card from their hand and put it face down in front of them. After everyone has chosen a card, they will reveal their cards at the same time.

2) Resolve the cards: Each card has a color and 1 of the following:

- A bird with a value between 3-6
- A fox with a value between 4-6
- A fleeing bird with value -2

The color of the played card will determine the farm that the card goes to in a turn. The way each farm resolves will depend on all of the played cards on that farm:

- If a bird is the only card played on a farm, it will eat all the corn (collect all the corn and add it to score pile).
- If more than 1 bird is played on a farm, they fight for the corn (each player rolls the dice and adds the rolled value to their card value, the one with the highest total points takes all the corn).

Foxes don't eat corn, only birds:

If no birds are on the same farm as a fox, the foxes on that farm don't collect anything

If a fox and at least 1 bird are on the same farm, the fox will eat all the birds on that farm and the birds don't eat anything.

If at least 2 foxes and at least 1 bird are on the same farm, the foxes fight (roll the dice and add it to the fox's value, highest total score eats all the birds).

Birds eaten are kept in the fox players' score pile (birds will be worth the same amount of points as their values)

The Fleeing Bird (-2 card):

If no other cards are in the same farm as the fleeing bird, it will eat all the corn on that farm like a normal bird.

If there are other cards (either birds or foxes), the fleeing bird will first eat a green corn (if available). Then everything else resolves normally. When a fox eats a fleeing bird, it gets -2 points.

3) Pass out new cards and place a corn cube on each farm: After each turn, all the played cards are gathered into a discard pile (excluding the poultry cards eaten by foxes, in which the fox players keep). Then a player passes out a card from the deck to each player. If there are not enough cards in the deck, the discard pile will be re-shuffled back into the deck and the procedure continues normally. Another player should also place a random corn cube on each farm.

E. End of Game / Score

Keep on playing until there are no longer enough corn cubes to place 1 on each farm. After that, count the total score of each player.

Each corn cube is worth 1-3 points depending on the color:

- Green = 1
- Blue = 2
- Yellow = 3

Each poultry card eaten by a fox is worth the amount of points on the card.

The player with the highest total score wins.

The Game Rules can be accessed via [1]

F. Example

Example of a turn with 3 players:

*** New Turn ***
— Farms —
Farm (Black) with corn: [BLUE]
Farm (Blue) with corn: [YELLOW]
Farm (Green) with corn: [BLUE]
Farm (Yellow) with corn: [GREEN]
Farm (Purple) with corn: [BLUE]
Farm (Red) with corn: [YELLOW]
— Players Pick Cards —
Player 1, choose a card to play:
1: BIRD (3, Yellow)
2: BIRD (3, Purple)
3: FOX (4, Purple)
4: BIRD (4, Green)
5: BIRD (4, Green)
4
Player 2, choose a card to play:
1: BIRD (4, Red)
2: BIRD (4, Yellow)
3: FOX (6, Purple)
4: BIRD (4, Red)
5: BIRD (5, Red)
3
Player 3, choose a card to play:
1: BIRD (3, Black)
2: FOX (6, Black)
3: BIRD (6, Purple)
4: BIRD (4, Blue)
5: FOX (4, Black)
4
— Resolving Cards —
Player 2's fox eats nothing!
Player 3's bird eats all the corn!
Player 1's bird eats all the corn!
— Player Scores After This Turn —
Player 1 (Score: 2)
Player 2 (Score: 0)
Player 3 (Score: 3)
— Player Score Pile —
Player 1's score pile:
CornCube: BLUE
Player 2's score pile:
Player 3's score pile:
CornCube: YELLOW
— Corn Count —
Green Corn Count: 25
Blue Corn Count: 23
Yellow Corn Count: 24
— Deck Count —
Deck Count: 45
Discard Pile: 3

Explanation:

1) Intialization: At the beginning, each farm is initialized with a random corn cube and each player is passed out 5 cards.
2) Choose card: Each player choose a card: player 1 chooses "Bird 4 Green", player 2 chooses "Fox 6 Purple" and player 3 chooses "Bird 4 Blue".
3) Resolve cards: Since player 1's bird is alone in the Green Farm, it eats the blue corn in the farm and gets 2 points. Same for player 3, his/her bird is alone on the Blue Farm so it eats the yellow corn in the farm and gets 3 points. Player 2's fox is in the purple farm, which does not have any bird, so the fox eats nothing. Finally, all played cards are put into the discard pile
4) Display scores of each player along with their score pile
5) Display remaining corn count as well as deck and discard pile count.
6) Next turn: each farm is again passed out a random corn cube and each player also draws a random card. The game continues until there are no corn cubes left.

III. Related Work

In this section, we would represent some ideas, which are main ideas for our game.

One of them is the Uno Game [2], which has the interface nearly as same as the idea we are working with

Fig. 1. Intro Scene

Description: These are the interfaces we are trying to achieve. In this interface, each player has a visual on all of their card deck, which they click on a preferred card, and it will be added to the center. And a player can only see their card deck, and the others are folded.

Another related work that gives us idea on working with the winner scene is Kahoot outro [3], which introduces the game winner.

Fig. 2. Winner Scene

Description: This is the outro that gives us ideas to introduce the winner. It is same as a podium, that introduces the 3rd place first, and then moves to the right, 2nd place later, then moves to the left and finally the 1st player, and move to the center.

IV. Team Work

In this section, we will discuss about the process of team work and the work structure of our project.

A. Who does what?

This team consists of 3 members: Dat, Nguyen, and Khoa. Below is the distribution of work process.

- Dat: Responsible for distribution of work for team members. Also responsible for researching and working with GUI(Javafx & Scene Builder). Moreover Dat is responsible for communications between team members and updates to the project workflow.
- Nguyen: Responsible for determining the classes, the workflow and logic code for score calculation. Working with Dat, Nguyen is responsible for the drawing and explanation of the diagrams.
- Khoa: Responsible for applying Dat's code to create a new version of GUI for single player.

Each week, the team will have an online meeting on Google Meet to update the work process to Dat.

B. Work Structure

Work Timeline: Here are our main timeline of the project:

1) 05/12/2024: Choose board game topic for the project.
2) 08/12/2024 - 10/12/2024: Understand rules, define classes, outline, skeleton of the project.
3) 11/12/2024 - 12/12/2024: Set up the extension, build up the project.
4) 12/12/2024 - 17/12/2024: Work with code, finish the logic code and run the game in Command Line.
5) 18/12/2024 - 30/12/2024: Assign Khoa to implement the GUI of the board game.
6) 01/01/2025 - 03/01/2025: Khoa's realization of errors and rework again with GUI.
7) 04/01/2025 - 12/01/2025: Dat's research, implementation of another version of GUI of the game.
8) 13/01/2025 - 17/01/2025: Dat, Khoa, Nguyen finish the documentation of report.
9) 18/01/2025 - 22/01/2025: Implement the gameplay to check for error of logic and GUI. End the project.

C. Ideas

The initial idea when we brainstorm is to create a board and make cards to move in the center area like poker and some card games. There are other ideas on using numbers from 1 to 5 to choose the preferred card each round. One member is responsible for outlining the game component while the 2 others determine the layout of the game and how to interact with the game.

Then it comes to the set up idea to stick with the second way to work with Command Line to test and implement the logic first. Then 2 members will independently apply the logic and create 2 versions of GUI for gameplay, which is single player and multiplayer

In the maintenance part, the member that is responsible for logic code is responsible for testing the game and reporting back to the member implementing GUI if there are errors in relation to the game components or logics of the game.

V. Proposed Approaches

Algorithms in Pseudocode:

1) Initialize class variables:
   - List of farms
   - List of players
   - List of cards (deck)
   - List of cards (discard pile)
   - Die object
   - List of corn cubes
   - Corn counts (green, blue, yellow)
2) Constructor Game():
   - Initialize cubes list with green, blue, and yellow corn types
3) Method getCubes():
   - Return cubes list
4) Constructor Game(playerCount):
   - Call initializeFarms()
   - Call initializeDeck()
   - Call initializePlayers(playerCount)
   - Call addInitialCorn()
5) Method getDeck():
   - Return deck list
6) Method initializeFarms():
   - Add farms with predefined colors to farms list
7) Method initializeDeck():
   - Add cards of different types and colors to deck
   - Shuffle the deck
8) Method initializePlayers(playerCount):
   - Create players and deal 5 cards to each from the deck
9) Method addInitialCorn():
   - Add initial corn cubes to farms randomly
10) Method startGame():
    - Loop until game ends (no more corns to distribute):
      - Print farms
      - Players pick cards
      - Resolve cards
      - Display player score piles
      - Display corn counts
      - Display deck and discard pile counts
      - Check if game should end
      - Deal new cards to players
11) Method printFarms():
    - Print each farm
12) Method pickCardForPlayer(player, scanner):
    - Allow player to pick a card from their hand
    - Return chosen card
13) Method resolveCards(chosenCards):
    - Group cards by color
    - Resolve each farm based on grouped cards
    - Display player scores after resolving actions
14) Method resolveFarm(cards, farm):
    - Separate cards into birds, foxes, and fleeing birds
    - Resolve fleeing birds if no foxes
    - Resolve foxes and birds if both present
    - Resolve birds if only birds present
    - Resolve foxes if only foxes present
15) Method resolveBirds(birds, farm):
    - Sort birds by player name
    - If one bird, it eats all corn
    - If multiple birds, they fight for corn
    - Add played bird cards to discard pile
16) Method resolveFoxesAndBirds(foxes, birds, fleeingBirds, farm):
    - Sort foxes by player name
    - Determine winning fox
    - Winning fox eats all birds
    - Add played fox and bird cards to discard pile (eaten birds will not be added, the fox player keeps them)
17) Method resolveFoxes(foxes):
    - Sort foxes by player name
    - Add played fox cards to discard pile
18) Method resolveFleeingBird(fleeingBirds, farm, otherBirdsAndFoxes):
    - If fleeing bird is alone, it eats all corn
    - If not alone, it takes one green corn if available
19) Method resolveFleeingBirdDiscard(fleeingBirds, farm, otherBirdsAndFoxes):
    - If fleeing bird is alone, it eats all corn
    - If not alone, it takes one green corn if available
    - Add played fleeing bird cards to discard pile
20) Method addCornToFarms():
    - Add corn cubes to farms randomly based on available corn types
21) Method dealNewCards():
    - Shuffle discard pile into deck if needed
    - Deal new cards to players
22) Method declareWinner():
    - Determine player with highest score and declare winner
23) Main method:
    - Print welcome message
    - Get number of players from user
    - Create Game object with player count
    - Start the game
    - Close scanner

Summary:

This pseudocode outlines how players interact through farms, cards, and corn resources. The game starts by initializing key components, including farms, players, a shuffled deck, a discard pile, and corn cubes of different types. Players are dealt cards and take turns picking and resolving them, with gameplay focusing on how birds, foxes, and fleeing birds

interact at farms to compete for corn. Birds eat corn or fight each other, foxes prey on birds, and fleeing birds act selectively based on the presence of other cards. The game progresses in a loop, where farms are updated, resources are redistributed, and new cards are dealt to players, with the game state displayed at each step. The loop ends when there are not enough corns to distribute, at which point the player with the highest score is declared the winner.

VI. Implementation Details

In this section, we will introduce the application structure, which will presents the game logic layer(Model), User Interface layer(View), Controller layer, and the resources of the project. Moreover, we will explain carefully the GUI details and give instructions about the UML diagrams and libraries used to run the game.

A. Application Structure

1) Game Logic Layer:

a) Role: First and foremost is the Game Logic layer(Model), which contains the core game logic, which contains classes for game rules, objects, and player state management.

b) Responsibility:

- Implement the game rules, logics and mechanics
- Manage and update the cards, cubes states and players' scores.

c) Model components:

- Card.java: This contains the class and the components of a card. It contains the type, color and value of cards, and the constructors, getter, setter function to call the card. A card can be valued from 3 to 6 for Bird, 4 to 6 for Fox, and -2 for Fleeing Bird. A card has 6 colors: Including Yellow, Green, Red, Blue, Purple, and Black.
- CornCube.java: This contains the class and the types of the corn, functions to value the points based on the color, which is 1, 2, or 3.
- Die.java: This contains a parameter to random a score from 1 to 6, which corresponds the values of a die.
- Farm.java: This contains a parameter color and a list of corns, which is defined in CornCube.java. The farm has also 6 colors, as same as Card. It also has methods to add corns and clear corns, which are functions used in game.
- Game.java: This contains methods to initialize the game, declare number of cards for player, add and remove cards, add and remove corns, and calculate the players' scores after each round. Gameplay on Command Line can be played here.
- Player.java: This contains parameter of name of players, which are set to player 1, 2, and 3. It has methods to add new card from Card Deck, add to player's current score.

2) User Interface Layer(View):

a) Role: Next is the User Interface Layer, which defines the Graphical User Interface (GUI) of the application. This layer is built using Scene Builder and added to the application under FXML files for a clean separation of UI design and application logic.

b) Responsibilities:

- Display the game boards, cards images, and other UI components, including the buttons to start and end game, or to function the game.
- Handle User Interaction as real-life board game.

c) Interface Components:

- begin.fxml: Begin menu view, which is the intro of the game.
- rules.fxml: Rule view, where players can understand rules of playing and score calculations.
- game.fxml: The main board game view, where players can interact with the game.
- winner.fxml: The board view to finalize and introduce the winner of the game.
- styles.css: Stylesheets for customization of User Interface elements.
- WinnerAnimation.java: Contains the animations to celebrate the winner in winner board view.

3) Controller Layer:

a) Role: The Controller Layer acts as a bridge between the Game Logic layer (Model) and the Interface layer (View). It contains Java controller classes linked to FXML files, using FXML:Controller atrributes.

b) Responsibilities:

- Handle user interactions (Button clicks, Cards moving, Corns adding, etc.).
- Update the view based on the game state and logic.

c) Controller Components:

- BeginController.java: Manage begin scene and enable users to start the game, change to rules view or exit the game. Moreover, this controller create transition from begin scene to game scene smoothly.
- MainController.java: Manage the game scene and enable user interactions with games, calculate score for players and handle the discarded card. This controller make transition from game scene to winner scene when the condition is satisfied.
- WinnerController.java: Manage the winner scene and apply the animations to the winner scene. It also handles the actions to play again or exit the game.

B. Resources

a) Roles: The resources contain static assets such as images of card, farms and gifs for animation.

b) Components:

- images/: Folder contains farms' images for game decorations.
- begin/: Folder contains image for the begin scene.
- cards/: Folder contains cards' images for game decorations.

- gif/: folder contains gifs and image used in the rules scene and game decorations.
- logo/: Folder contains the logo for the game bar decoration.

C. GUI Details

In this section, we will introduce and explain how the scenes are created and how to handle the interactions in each scene.

First, we take the images of the games, which have the images of cards, farms to create the interface for the main game. As discussed in [5] we take screenshots of each card image and farm image and create an **ImageView** for each area, which will be seen in the game scene.

Fig. 3. HickHack Game Image

Intro Scene: This scene is the introduction of our game.

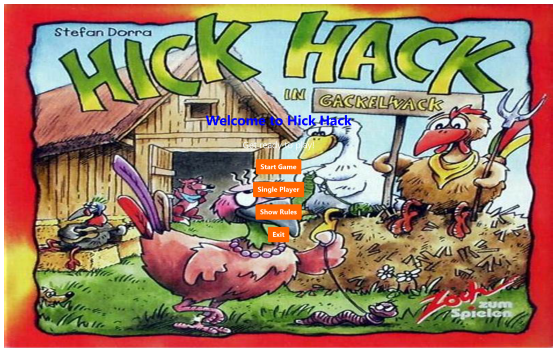

Fig. 4. Intro Scene

The **background image** in the code is the background image in the scene, which is used to decorate the intro. The 2 labels **title Label** and **welcome Label** are the lines appearing in the intro, which there are 3 buttons, including **Start Game**, **Show Rules**, and **Exit** are buttons that make transitions to the corresponding scenes or exit the game.

Listing 1. Start Game Button

```
@FXML
public void onOkButtonPressed(ActionEvent
    event) {
    int playerCount = 3; // Fixed number of
        players
    game = new Game(playerCount); //
        Initialize the game with the number
        of players
    if (mainApp != null) {
        mainApp.showGamePage(game); // Pass
            the game instance to the game
            page
    } else {
        System.err.println("mainApp is null ,
            cannot proceed.");
    }
}
```

Listing 2. Show Rules Button

```
@FXML
private void handleShowRulesAction(
    ActionEvent event) {
    if (mainApp != null) {
        mainApp.showRulesPage();
    } else {
        System.err.println("mainApp is null ,
            cannot proceed.");
    }
}
```

Listing 3. Single Player Button

```
@FXML
 public void handleSinglePlayerButtonAction(
     ActionEvent event) {
     int playerCount = 3;
     game = new Game(playerCount);
     if (mainApp != null) {
         mainApp.showSinglePage(game);
        } else {
            System.err.println("mainApp is
                null ,cannot proceed.");
     }
 }
```

Listing 4. Exit Button

```
@FXML
public void handleExitAction(ActionEvent
    event) {
    Stage stage = (Stage) ((Button) event.
        getSource()).getScene().getWindow();
    stage.close();
}
```

Game Rules Scene:

This is the Rules Scene if players click the **Show Rules** Button

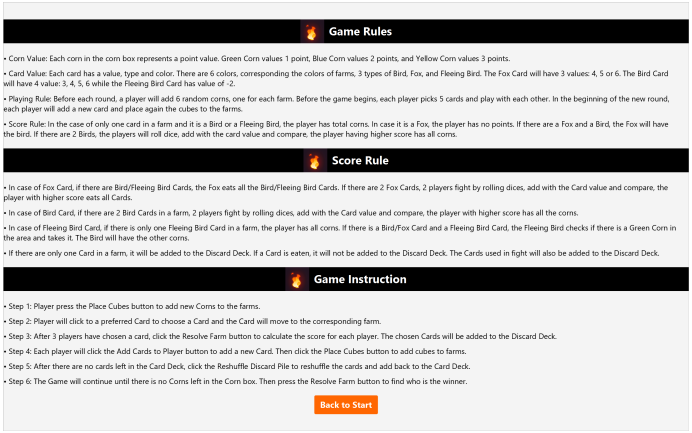

Fig. 5. Rules Scene

In this scene, there are full description of the rules:

- **Game Rules**: show how to play the game.
- **Score Rules**: show how to calculate players' score.
- **Game Instructions**: show how to play the games, click the button and how to choose cards.

If players click the **Back to Start** button, the game will return back to intro scene.

Listing 5. Back to Start Button

```
@FXML
private void handleBackToStartAction(ActionEvent
    event) {
    try {
        // Load the begin.fxml file
        FXMLLoader loader = new FXMLLoader(getClass
            ().getResource("/hellofx/resources/
            begin.fxml"));
        Parent root = loader.load();

        // Get the controller and set the mainApp
            reference
        BeginController beginController = loader.
            getController();
        beginController.setMainApp(mainApp);

        // Get the current stage
        Stage stage = (Stage) ((Button) event.
            getSource()).getScene().getWindow();

        // Set the new scene
        stage.setScene(new Scene(root));
        stage.setFullScreen(true); // Set the stage
            to full screens
        stage.show();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Single Player Scene: This is the game scene, when players click the **Single Player** Button.

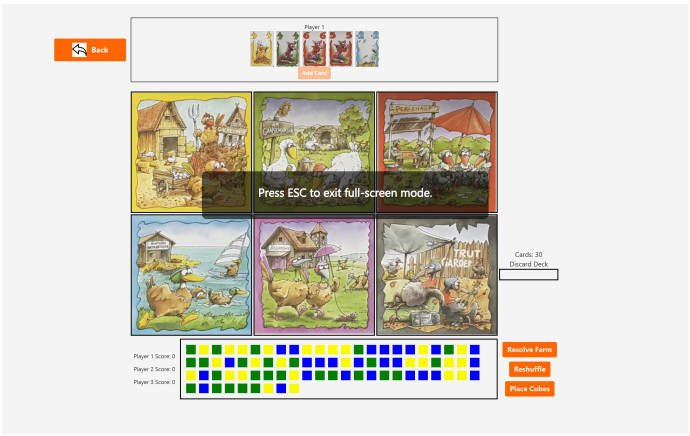

Fig. 6. Single Player Scene

Multiplayer Game Scene: This is the game scene, when players click the **Start Game** Button.

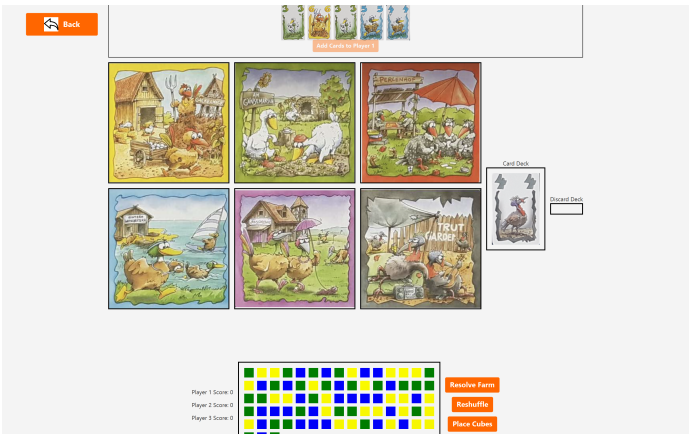

Fig. 7. Game Scene player 1 turn

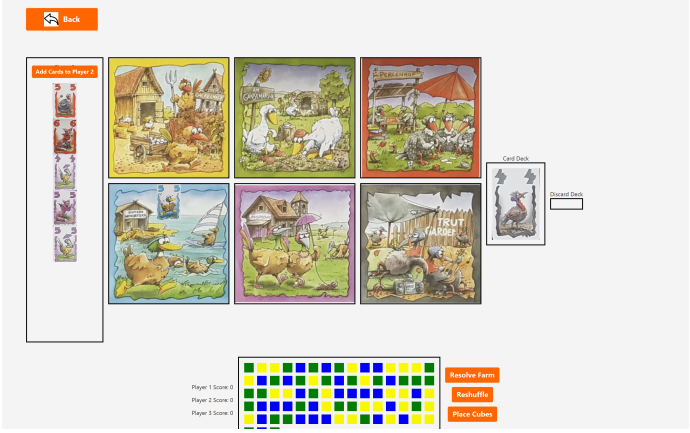

Fig. 8. Game Scene player 2 turn

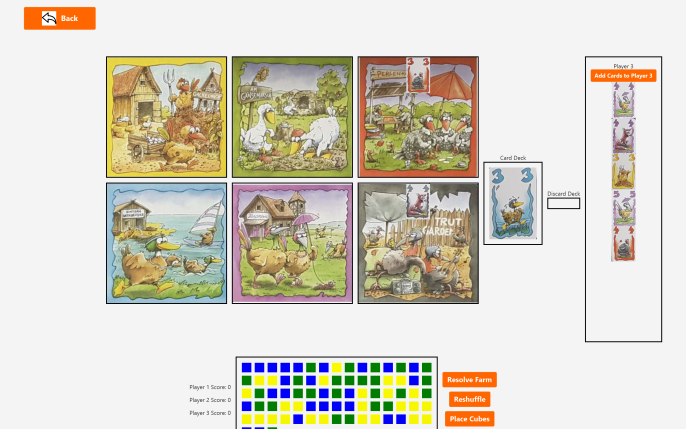

Fig. 9. Game Scene player 3 turn

On the top-left, there is a back button, which if players enter this button, the game returns back to the intro scene. There is also image that decorates the back button.

Listing 6. Back Button

```
@FXML
private void handleBackToStartAction(ActionEvent
    event) {
    try {
        // Load the begin.fxml file
        FXMLLoader loader = new FXMLLoader(getClass
            ().getResource("/hellofx/resources/
            begin.fxml"));
        Parent root = loader.load();

        // Get the controller and set the mainApp
            reference
        BeginController beginController = loader.
            getController();
        beginController.setMainApp(mainApp);

        // Get the current stage
        Stage stage = (Stage) ((Button) event.
            getSource()).getScene().getWindow();

        // Set the new scene
        stage.setScene(new Scene(root));
        stage.setFullScreen(true); // Maximize the
            stage instead of setting it to full
            screen
        stage.show();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

On the top, left, and right of the game, there are players card deck, which in the initial round, players can only see cards of player 1, while player 2 and 3's card decks has been hidden. Inside each player card deck, there are 5 cards, corresponding to the game rule that each player must have 5 cards to start the game. There are also button at each card deck to add a new card to player to keep playing the game.

In the center, there are 6 squares, which represent 6 farms for the cards to be placed into. Besides, there are 2 decks, which on the left side is the deck containing the main card deck to be added to players' card deck. On the right side is the discard deck, which contains the used cards to play again when there are no cards left in the main card deck.

Fig. 10. Center Game Scene

On the bottom, there is a big box, which contains the available corn cubes, which will be added by clicking the **Place Cubes** button. At the end of the card and cube addition phase, click **resolve farm farm** button to calculate players' score after each turn. When there are no cards left in the main card deck, click the **Reshuffle** button to reshuffle the discard deck, and it automatically adds back to the main card deck to continue the process.

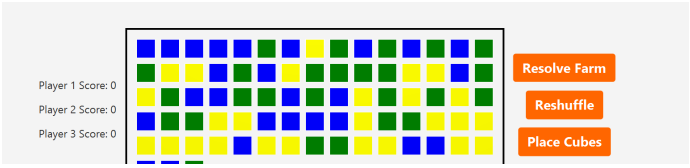

Fig. 11. Bottom Game Scene

Winner Scene: Here is the winner scene of the game

At the end, when there are no corns left in the corn box, press the **Resolve Farm** button once more to transition to the winner scene. Here finalize and give congratulations animation to three players with their 3 scores. There are 2 buttons in the winner scene:

- Press the **Play Again** button to transition back to the intro scene to start a new game.
- PRess the **Exit** button to exit the current game.

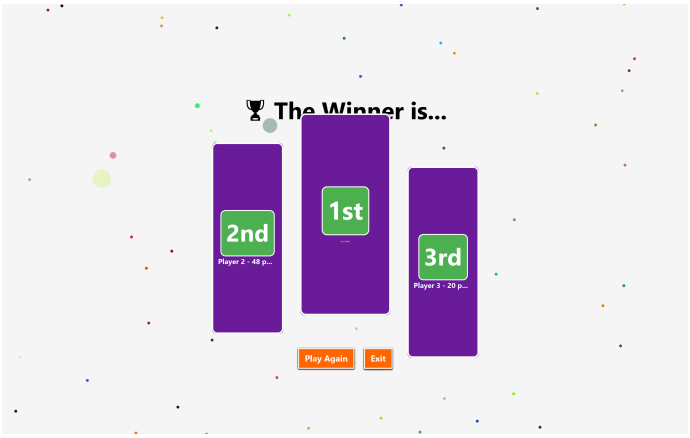

Fig. 12. Winner Scene

D. UML Diagram

1) Class Diagram: The Class diagram illustrates 6 different classes involved in out Hick Hack boardgame, with those being: Card, Player, Game, CornCube, Farm and Die.

Classes:

- Card: includes attributes such as type, value and color to identify each card. Has methods to gets these information and check if it is a fleeing bird.
- Player: has a name, a hand of cards and a score pile. Functions include getters, add cards, calculate scores, add to scores and print score.
- Game: possesses a list of farms, list of players, a deck, a discard pile and a die. Includes a number of functions for initializing the game and its components, for resolving the farms, for printing out the game state after each round and lastly, declare the winner.
- Corncube: each has a color along with methods to get point and get type.
- Farm: associates with a distinct color and includes a list of corncubes inside. Uses functions to add and clear corn, get the color and the list of corncubes.
- Die: attributes include a value and a method to roll.

Relationships:

- Player - Card: since each player has a 5 cards on hand and each card only belongs to 1 player at a time, it is therefore a 1:N relationship.
- Game - Card: a game has a total of 60 cards and each card is only in 1 game, so the relationship is 1:N as well.
- Game - Player: this is a 1:N association because a game has a number of players but a player only play 1 game at a time.
- Game - Farm: A game contains 6 farms and each farm only goes with a single game, so the relationship is 1:N.
- Game - Die: A game only as 1 die and a die is used only in that game, hence it is a 1:1 association.
- Farm - Corncube: a farm may contain any number of cubes but a cube is only contained in 1 farm, therefore, it is a 1:N relationship.

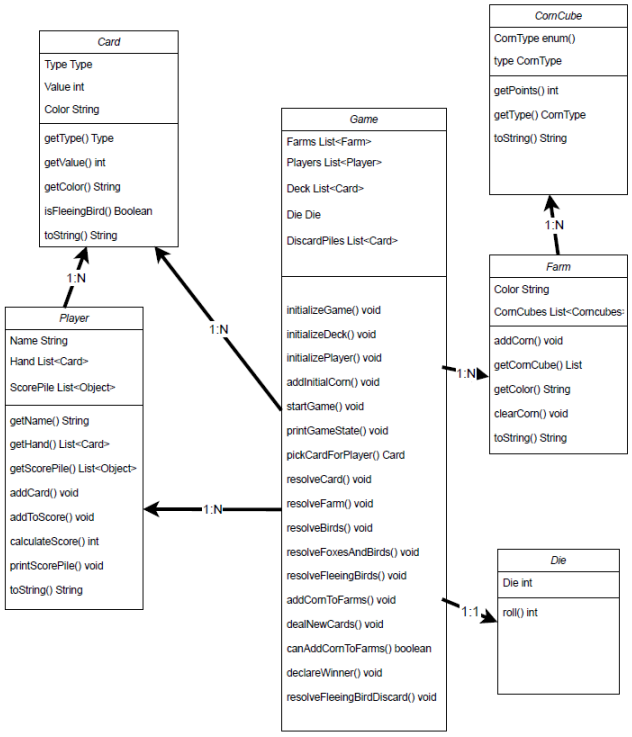

Fig. 13. Class Diagram

2) Use Case Diagram: The Use Case diagram demonstrates a number of actors and their use cases involved in the Hick Hack boardgame. The primary actor is the Game and secondary actors include CornCube, Farm, Deck, Card, DiscardPile and Player.

Main use cases:

The game begins with Start Game, which initializes essential components like the deck, farms, players, and discard pile. Use cases such as Create Farm, Add Initial Corns, and Create Deck handle the setup by defining farms, distributing corn resources, and preparing cards. During gameplay, players interact with use cases like Place Card, which moves cards to the discard pile, and Resolve Card, which manages the outcomes of actions on farms, birds, foxes, and scores. The diagram also includes tasks for calculating scores and maintaining game progression, ensuring modular and efficient execution.

Relationships: The relationships in the Use Case diagram showcase a structured and interconnected system where higher-level processes depend on smaller, modular tasks. The primary actor, Game, interacts with most use cases to initiate and manage gameplay, while supporting actors like CornCube, Farm, Deck, Card, DiscardPile, and Player perform specialized roles. Use cases are linked through include and extend relationships, ensuring modularity and reusability. For example, foundational use cases like Create Farm and Create Deck rely on smaller tasks such as Get Card Type or Add Corn Cubes for setup, while gameplay resolutions like Resolve Card extend into specific actions such as Resolve Birds or Add To

Score. This interconnected structure ensures a seamless flow of actions while maintaining clarity and flexibility in the system.

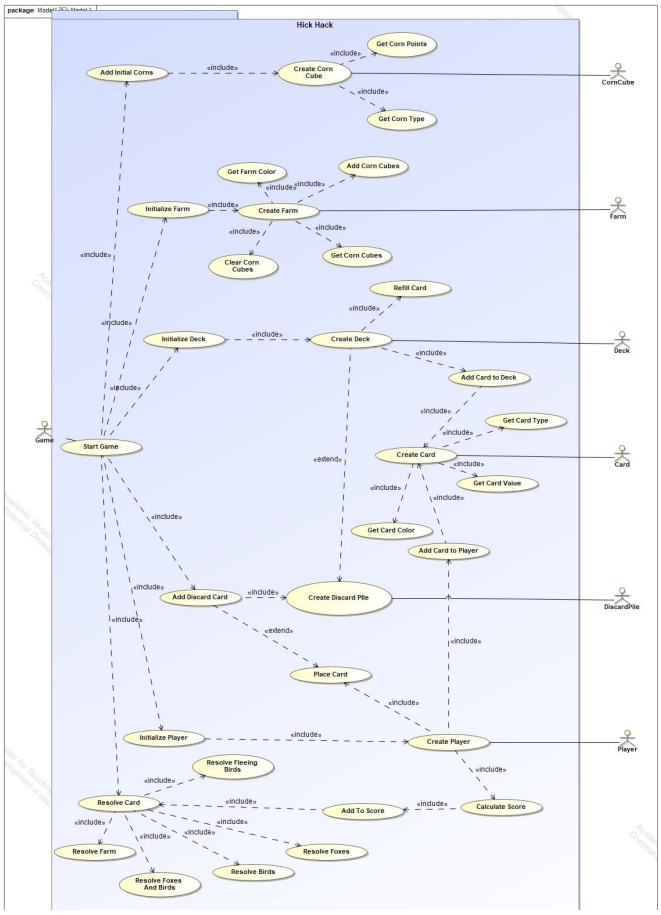

Fig. 14. Use Case Diagram

3) Sequence Diagram: The Sequence Diagram illustrates the ordered sequence of actions performed by the Hick Hack game. A total of 7 object lifecycles are shown: Game, Farm, CornCube, Card, Deck, Discard Pile and Player.

Sequence:

- The Game starts by initialize the 6 farms, each with a distinct color.
- Next, we will initialize the deck by adding cards to it.
- Initialize players by adding cards to their hand.
- Add a random cube to each farm.
- After initialization, the game enters a while loop until there are not enough corn cubes to be added.
- Each player choose a card from their hand.
- Resolve the cards.
- Resolve the farms.
- Add played cards to the discard pile.
- Display the game state, such as the information regarding the deck, the discard pile, number of corn cubes remaining and each player's score pile.
- After each turn, if there are enough cards in the deck, refill each player's hand with 1 card. If not enough, reshuffle the discard pile back to the deck and distribute a card to each player.
- Also check if there are enough corn to distribute to each farm for the next turn, if not enough, the game ends and the player with the highest score wins.

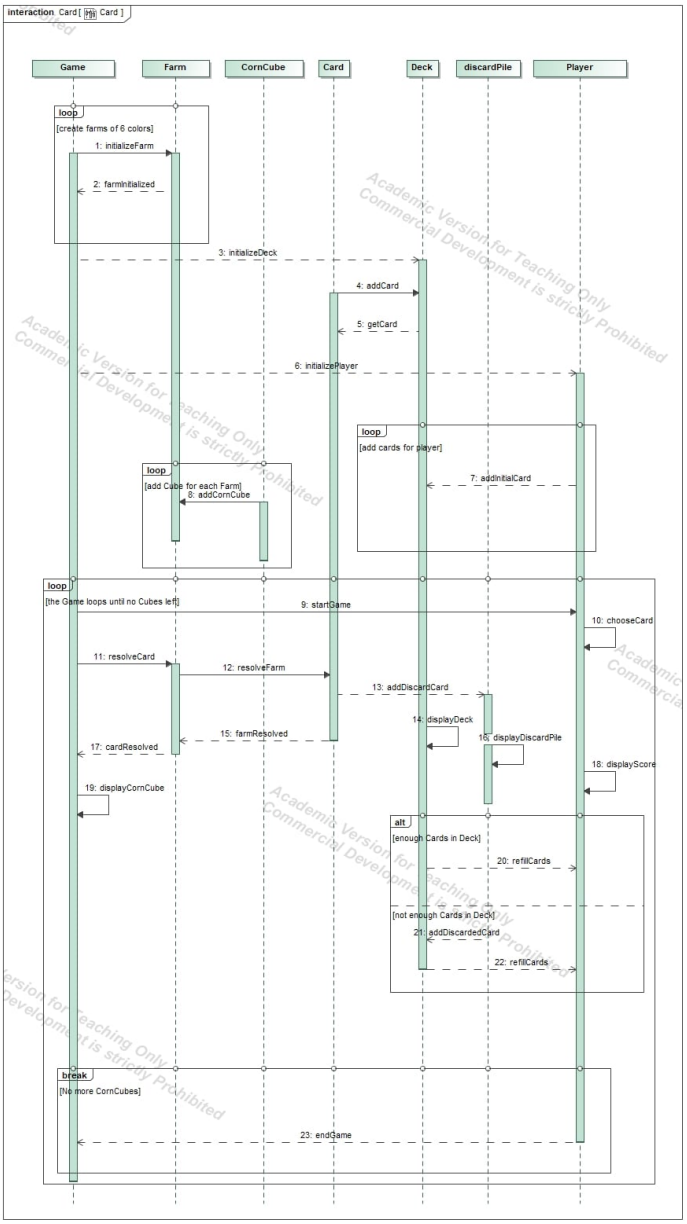

Fig. 15. Sequence Diagram

E. Used Libraries

- import javafx.fxml.FXML;
- import javafx.fxml.FXMLLoader;
- import javafx.geometry.Bounds;
- import javafx.geometry.Insets;
- import javafx.geometry.Pos;
- import javafx.scene.Parent;
- import javafx.scene.Scene;

- import javafx.scene.control.Button;
- import javafx.scene.image.Image;
- import javafx.scene.image.ImageView;
- import javafx.scene.layout.StackPane;
- import javafx.scene.layout.VBox;
- import javafx.scene.layout.Background;
- import javafx.scene.layout.BackgroundImage;
- import javafx.scene.layout.BackgroundPosition;
- import javafx.scene.layout.BackgroundRepeat;
- import javafx.scene.layout.BackgroundSize;
- import javafx.scene.layout.FlowPane;
- import javafx.scene.layout.GridPane;
- import javafx.scene.layout.Pane;
- import javafx.scene.control.Label;
- import javafx.scene.paint.Color;
- import javafx.scene.shape.Box;
- import javafx.stage.Stage;
- import javafx.scene.paint.PhongMaterial;
- import javafx.animation.KeyFrame;
- import javafx.animation.KeyValue;
- import javafx.animation.Timeline;
- import javafx.animation.TranslateTransition;
- import javafx.event.ActionEvent;
- import javafx.util.Duration;
- import java.util.Iterator;
- import java.io.FileInputStream;
- import java.io.FileNotFoundException;
- import java.io.InputStream;
- import java.util.ArrayList;
- import java.util.Arrays;
- import java.util.Collections;
- import java.util.HashMap;
- import java.util.List;
- import java.util.Map;
- import java.util.Random;
- import java.util.stream.Collectors;
- import javafx.animation.*

F. Code Snippets

Here represents the codes to calculate player score.

- Case 1: When there is one unique card in a farm

```
// If there is only one card in the VBox
if (cardVBox.getChildren().size() == 1) {
    ImageView cardImageView = (ImageView) cardVBox.getChildren().get(0);
    Card card = (Card) cardImageView.getUserData();
    int playerNumber = getPlayerNumberByCard(card);

    // Check if the card belongs to the player and is a bird, fox, or fleeing bird
    if (card != null) {
        if (card.getType() == Card.Type.BIRD) {
            int totalScore = playerScores.getOrDefault(playerNumber, defaultValue:0);
            totalScore += areaTotalValueMap.getOrDefault(area, 0);
            playerScores.put(playerNumber, totalScore);
            System.out.println("Single bird card in VBox for player " + playerNumber + ", total score: " + totalScore);
            discardPile.getChildren().add(cardImageView);
            // Clear the cubes in the VBox
            clearCubesInVBox(area);
        } else if (card.getType() == Card.Type.FOX) {
            System.out.println("Single fox card in VBox for player " + playerNumber + ", no score added.");
            // Add fox card to the discard pile
            discardPile.getChildren().add(cardImageView);
        } else if (card.getType() == Card.Type.FLEEING_BIRD) {
            int totalScore = playerScores.getOrDefault(playerNumber, defaultValue:0);
            totalScore += areaTotalValueMap.getOrDefault(area, 0);
            playerScores.put(playerNumber, totalScore);
            System.out.println("Single fleeing bird card in VBox for player " + playerNumber + ", total score: " + totalScore);
            // Add fleeing bird card to the discard pile
            discardPile.getChildren().add(cardImageView);
            // Clear the cubes in the VBox
            clearCubesInVBox(area);
        }

        // Remove the VBox of cards
        area.getChildren().remove(cardVBox);
    }
}
```

Fig. 16. Case 1

- Case 2: When there are 2 birds in a farm

```
if (birdCards.size() == 2) {
    Map<Card, Integer> cardTotalValues = new HashMap<>();
    boolean tie;

    do {
        tie = false;
        cardTotalValues.clear();

        for (Card card : birdCards) {
            int diceRoll = random.nextInt(bound:6) + 1; // Roll a dice for the card
            int totalValue = card.getValue() + diceRoll;
            cardTotalValues.put(card, totalValue);
            int cardPlayerNumber = getPlayerNumberByCard(card);
            System.out.println("Player " + cardPlayerNumber + " rolled " + diceRoll + " for card with value " + card.getValue() + ", total: " + totalValue);
        }

        // Determine the card with the highest total value
        Card winningCard = null;
        int maxTotalValue = 0;
        int maxCount = 0;
        for (Map.Entry<Card, Integer> entry : cardTotalValues.entrySet()) {
            if (entry.getValue() > maxTotalValue) {
                maxTotalValue = entry.getValue();
                winningCard = entry.getKey();
                maxCount = 1;
            } else if (entry.getValue() == maxTotalValue) {
                maxCount++;
            }
        }

        // Check for a tie
        if (maxCount > 1) {
            tie = true;
            System.out.println(x:"Tie detected, rolling dice again...");
        } else {
            // The card with the highest total value gets the total value of the area
            if (winningCard != null) {
                int winningPlayerNumber = getPlayerNumberByCard(winningCard);
                int totalScore = playerScores.getOrDefault(winningPlayerNumber, defaultValue:0);
                totalScore += areaTotalValueMap.getOrDefault(area, 0);
                playerScores.put(winningPlayerNumber, totalScore);
                System.out.println("Winning card for player " + winningPlayerNumber + ", total score: " + totalScore);
```

Fig. 17. Case 2

- Case 3: When there are 2 foxes in a farm

```
else if (foxCount == 2) {
    System.out.println(x:"Two fox cards in VBox, no score added.");
    for (javafx.scene.Node node : cardVBox.getChildren()) {
        ImageView cardImageView = (ImageView) node;
        Card card = (Card) cardImageView.getUserData();
        if (card != null && card.getType() == Card.Type.FOX) {
            discardPile.getChildren().add(cardImageView);
        }
    }
    area.getChildren().remove(cardVBox);
}
```

Fig. 18. Case 3

- Case 4: When there are 1 bird and 1 fleeing bird

```
else if (birdCards.size() == 1 && fleeingBirdCard != null) {
    int fleeingBirdPlayerNumber = getPlayerNumberByCard(fleeingBirdCard);
    int birdPlayerNumber = getPlayerNumberByCard(birdCards.get(0));
    int totalValue = areaTotalValueMap.getOrDefault(area, 0);
    boolean hasGreenCube = false;

    // Check for the presence of a green cube
    for (javafx.scene.Node node : area.getChildren()) {
        if (node instanceof VBox && "cubes".equals(node.getId())) {
            VBox cubeVBox = (VBox) node;
            for (javafx.scene.Node cubeNode : cubeVBox.getChildren()) {
                if (cubeNode instanceof StackPane) {
                    StackPane cubeStack = (StackPane) cubeNode;
                    if (cubeStack.getChildren().size() > 1) {
                        PhongMaterial material = (PhongMaterial) ((Box) cubeStack.getChildren().get(1)).getMaterial();
                        Color cubeColor = (Color) material.getDiffuseColor();
                        if (Color.GREEN.equals(cubeColor)) {
                            hasGreenCube = true;
                            break;
                        }
                    }
                }
            }
        }
        if (hasGreenCube) {
            break;
        }
    }
```

Fig. 19. Case 4

```
if (hasGreenCube) {
    int fleeingBirdScore = playerScores.getOrDefault(fleeingBirdPlayerNumber, defaultValue:0);
    fleeingBirdScore += 1;
    playerScores.put(fleeingBirdPlayerNumber, fleeingBirdScore);
    System.out.println("Fleeing bird card in VBox for player " + fleeingBirdPlayerNumber + ", score: 1, total score: " + fleeingBirdScore);
    // Bird gets the remaining total value

    int birdScore = playerScores.getOrDefault(birdPlayerNumber, defaultValue:0);
    birdScore += (totalValue-1);
    playerScores.put(birdPlayerNumber, birdScore);
    System.out.println("Bird card in VBox for player " + birdPlayerNumber + ", score: " + (totalValue - 1) + ", total score: " + birdScore);
    // Add bird card to the discard pile
    ImageView birdCardImageView = findCardImageView(birdCards.get(0));
    if (birdCardImageView != null) {
        discardPile.getChildren().add(birdCardImageView);
    }
    ImageView fleeingCardImageView = findCardImageView(fleeingBirdCard);
    if (fleeingCardImageView != null) {
        discardPile.getChildren().add(fleeingCardImageView);
    }
    // Remove the VBox of cards
    area.getChildren().remove(cardVBox);

    // Clear the cubes in the VBox
    clearCubesInVBox(area);
}
else
{
    // Bird gets the remaining total value
    int birdScore = playerScores.getOrDefault(birdPlayerNumber, defaultValue:0);
    birdScore += totalValue;
    playerScores.put(birdPlayerNumber, birdScore);
    System.out.println("Bird card in VBox for player " + birdPlayerNumber + ", score: " + (totalValue) + ", total score: " + birdScore);

    // Add bird card to the discard pile
    ImageView birdCardImageView = findCardImageView(birdCards.get(0));
    if (birdCardImageView != null) {
        discardPile.getChildren().add(birdCardImageView);
    }
    ImageView fleeingCardImageView = findCardImageView(fleeingBirdCard);
    if (fleeingCardImageView != null) {
        discardPile.getChildren().add(fleeingCardImageView);
    }
    // Remove the VBox of cards
    area.getChildren().remove(cardVBox);

    // Clear the cubes in the VBox
    clearCubesInVBox(area);
```

Fig. 20. Case 4

- Case 5: When there is 1 bird and 1 fox

```
else if (birdCards.size() == 1 && fleeingBirdCard != null) {
    int fleeingBirdPlayerNumber = getPlayerNumberByCard(fleeingBirdCard);
    int birdPlayerNumber = getPlayerNumberByCard(birdCards.get(0));
    int totalValue = areaTotalValueMap.getOrDefault(area, 0);
    boolean hasGreenCube = false;

    // Check for the presence of a green cube
    for (javafx.scene.Node node : area.getChildren()) {
        if (node instanceof VBox && "cubes".equals(node.getId())) {
            VBox cubeVBox = (VBox) node;
            for (javafx.scene.Node cubeNode : cubeVBox.getChildren()) {
                if (cubeNode instanceof StackPane) {
                    StackPane cubeStack = (StackPane) cubeNode;
                    if (cubeStack.getChildren().size() > 1) {
                        PhongMaterial material = (PhongMaterial) ((Box) cubeStack.getChildren().get(1)).getMaterial();
                        Color cubeColor = (Color) material.getDiffuseColor();
                        if (Color.GREEN.equals(cubeColor)) {
                            hasGreenCube = true;
                            break;
                        }
                    }
                }
            }
        }
        if (hasGreenCube) {
            break;
        }
    }
```

Fig. 21. Case 5

- Case 6: When there is 1 fox and 1 fleeing bird

```
else if (fleeingBirdCard != null && foxCard != null) {
    int foxPlayerNumber = getPlayerNumberByCard(foxCard);
    int fleeingBirdCardValue = fleeingBirdCard.getValue();
    int totalScore1 = playerScores.getOrDefault(foxPlayerNumber, defaultValue:0);
    totalScore1 += fleeingBirdCardValue;
    playerScores.put(foxPlayerNumber, totalScore1);
    System.out.println("Fox card in VBox for player " + foxPlayerNumber + ", score added by fleeing bird card value: " + fleeingBirdCardValue + "

    boolean hasGreenCube = false;

    // Check for the presence of a green cube
    for (javafx.scene.Node node : area.getChildren()) {
        if (node instanceof VBox && "cubes".equals(node.getId())) {
            VBox cubeVBox = (VBox) node;
            for (javafx.scene.Node cubeNode : cubeVBox.getChildren()) {
                if (cubeNode instanceof StackPane) {
                    StackPane cubeStack = (StackPane) cubeNode;
                    if (cubeStack.getChildren().size() > 1) {
                        PhongMaterial material = (PhongMaterial) ((Box) cubeStack.getChildren().get(1)).getMaterial();
                        Color cubeColor = (Color) material.getDiffuseColor();
                        if (Color.GREEN.equals(cubeColor)) {
                            hasGreenCube = true;
                            break;
                        }
                    }
                }
            }
        }
        if (hasGreenCube) {
            break;
        }
    }

    // Fleeing bird gets 1 point only if there is a green cube
    if (hasGreenCube) {
        int fleeingBirdNumber = getPlayerNumberByCard(fleeingBirdCard);
        int totalScore2 = playerScores.getOrDefault(fleeingBirdNumber, defaultValue:0);
        totalScore2 += 1;
        playerScores.put(fleeingBirdNumber, totalScore2);
        System.out.println("Fleeing bird card in VBox for player " + fleeingBirdNumber + ", score: 1, total score: " + totalScore2);
    }

    // Add fox card to the discard pile before removing the VBox
    ImageView foxCardImageView = findCardImageView(foxCard);
    if (foxCardImageView != null) {
        discardPile.getChildren().add(foxCardImageView);
```

Fig. 22. Case 6

- Method to add delay for bot

```
private void playCardAfterDelay(int playerNumber, VBox playerPile) {
    List<ImageView> cardImageViews = playerPile.getChildren().stream()
            .filter(node -> node instanceof ImageView)
            .map(node -> (ImageView) node)
            .collect(Collectors.toList());

    if (!cardImageViews.isEmpty()) {
        Random random = new Random();
        int randomIndex = random.nextInt(cardImageViews.size());
        ImageView cardImageView = cardImageViews.get(randomIndex);
        Card card = (Card) cardImageView.getUserData();

        PauseTransition playCardPause = new PauseTransition(Duration.seconds(2)); // 2-second delay before playing a card
        playCardPause.setOnFinished(event -> {
            handleCardSelection(card, playerPile, cardImageView, playerNumber);

            if (playerNumber == 2) {
                player2CardSelected = true;
                currentPlayerTurn = 3;
                transitionToNextPlayer(player3CardPile, player1CardPile, player2CardPile);
            } else if (playerNumber == 3) {
                player3CardSelected = true;
                currentPlayerTurn = 1;
                transitionToNextPlayer(player1CardPile, player2CardPile, player3CardPile);
            }

            if (player1CardSelected && player2CardSelected && player3CardSelected) {
                resolveFarmButton.setDisable(false);
            }
        });
        playCardPause.play();
    }
}
```

Fig. 23. Add Delay Method for Bot

The **playCardAfterDelay** method manages the delayed selection and playing of a card for a given player in the game. It starts by identifying all **ImageView** nodes in the player's card pile (**playerPile**), representing the available cards. If the pile is not empty, it randomly selects one card using a random index. The method retrieves the corresponding Card object from the ImageView. A 2-second delay is introduced with a **PauseTransition**, after which the selected card is processed via **handleCardSelection**, which likely manages game-specific actions related to the card.

The method updates the turn to the next player based on playerNumber (e.g., transitioning from player 2 to 3 or player 3 to 1). If all three players have selected their cards, it enables the resolveFarmButton, allowing further gameplay progression.

- Method to add random method as a bot for single player. The playAutomaticallyForPlayer method automates a player's actions in a card game. It determines the player's card pile (**player2CardPile** or **player3CardPile**) based on the provided **playerNumber**. It collects all the **ImageView** nodes in the pile, representing the player's cards. If the player has exactly four cards, a 2-second delay is initiated using a **PauseTransition** before adding a card to the pile with **addCardsToPlayerPile** and subsequently playing a card with **playCardAfterDelay**. If the player has fewer than four cards, the method directly triggers the **playCardAfterDelay** action without delay. This ensures appropriate pacing and card-playing behavior for the player.

```
private void playAutomaticallyForPlayer(int playerNumber) {
    VBox playerPile = (playerNumber == 2) ? player2CardPile : player3CardPile;
    List<ImageView> cardImageViews = playerPile.getChildren().stream()
            .filter(node -> node instanceof ImageView)
            .map(node -> (ImageView) node)
            .collect(Collectors.toList());

    if (cardImageViews.size() == 4) {
        PauseTransition addCardPause = new PauseTransition(Duration.seconds(2)); // 2-second delay before adding a card
        addCardPause.setOnFinished(event -> {
            addCardsToPlayerPile(1, playerPile, playerNumber);
            playCardAfterDelay(playerNumber, playerPile);
        });
        addCardPause.play();
    } else {
        playCardAfterDelay(playerNumber, playerPile);
    }
}
```

Fig. 24. Automatic Play Method for Bot

These are basic cases that handle basic score rules that are implemented here. To summarize, these are basic summaries we finalize after writing codes:

- When there is only 1 and unique card in the farm:
  - If it is a bird or a fleeing bird, it eats all the corns.
  - If it is a fox, it eats nothing.
- When there are 2 cards in a farm
  - If there are 2 bird cards, it fights to eat all the corns.
  - If there are 2 fox cards, it eats nothing.
  - If there are 1 bird and 1 fox card, fox eats the bird.
  - if there are 1 bird and 1 fleeing bird card, fleeing bird eat one green corn if there is one in the farm. If else, bird eats all.
  - IF there are 1 fox and 1 fleeing bird card, fleeing bird eat one green corn if there is one in the farm. Then the fox card eats the fleeing bird.
- When there are 3 cards in a farm
  - If there are 3 bird cards, they fight to eat all the corns.
  - If there are 3 fox cards, they eat nothing.
  - If there are 2 bird and 1 fox card, the fox eats all the birds.
  - If there are 2 fox and 1 bird card, the foxes fight to eat the bird.
  - If there are 2 fox and 1 fleeing bird card, the fleeing bird eat one green corn if there is one in the farm. Then the foxes fight to eat the fleeing bird
  - If there are 1 fox, 1 bird, and 1 fleeing bird card, the fleeing bird eat one green corn if there is one in the farm. Then the fox eats the bird and fleeing bird.

VII. Experimental Results, Statistics Tests, Running Scenarios

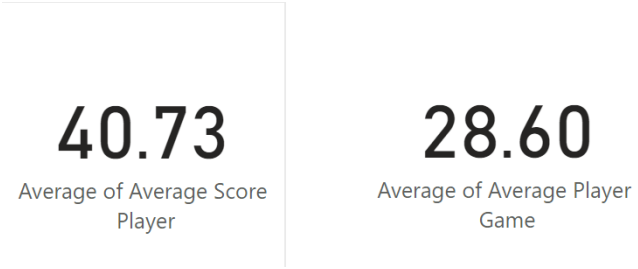

Fig. 25. Average scores of Player on Multiplayer vs. Singleplayer

Based on this numbers, a player choosing Multiplayer mode get an average of approximately 41 points to compete with other players. On the other hand, if player chooses Singleplayer mode, a player can get an average of 28.6 points to compete with 2 bots. The reason behind is that when playing multiplayer, the chance of not challenging each other is low and calculations to get points are easier. However, calculations to get points from bots are more challenging to solve, which results in a lower chance to win the game.

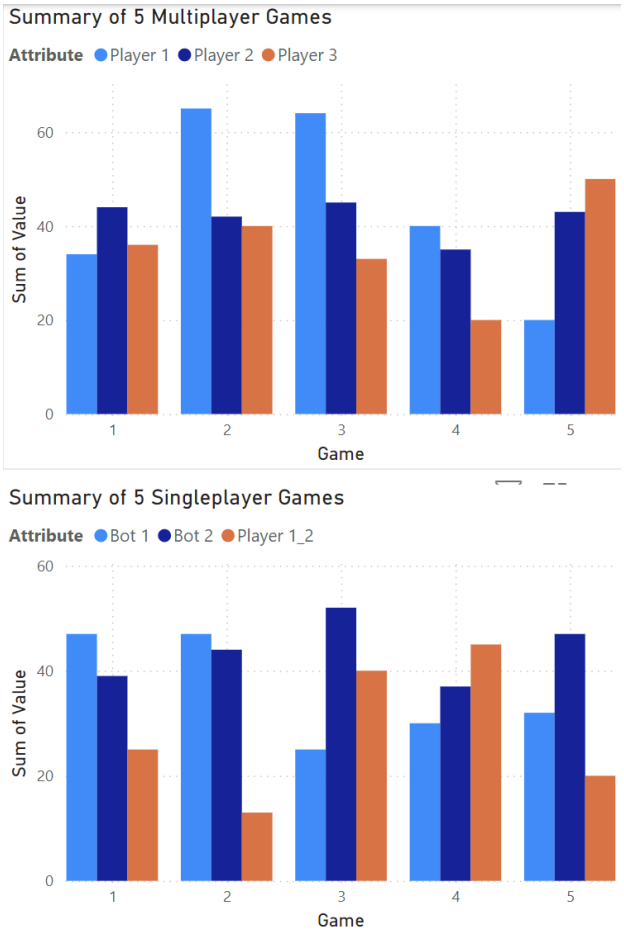

Fig. 26. Score Summary of 5 Games

In these clustered column charts, there are different trends between 2 charts. On the charts of Multiplayer games, the score of players are mostly higher than 30 points and the probability of chance to win the game of each player is same. On the other hand, the average score of singleplayer is relatively lower, compared to the players in Multiplayer mode, and there are less chances for player to finish the game in 1st place

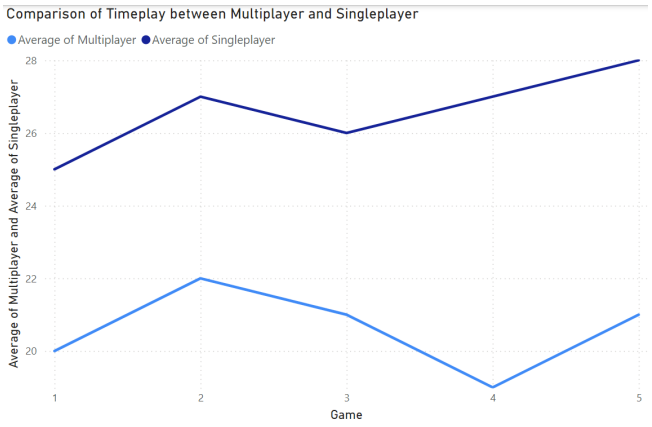

Fig. 27. Timeplay Comparison

In this line chart, there is a clear difference in the timeplay between Singleplayer and Multiplayer, with the former always higher than 25 minutes and the latter fluctuating around 21 minutes.

VIII. Conclusions and Future Work

A. How was the teamwork

The teamwork of my team is great, despite the problem of distance. The members succeeded in overcoming the problem of distance and excellently communicated with other members for a good and beautiful logic and User Interface. Although there are setbacks in the end of December when team had some problems with code, the team managed to finish and documented the report on time and members finished the part of tasks given before deadline of team leader.

B. What you have learned

After this project, this is the summary of the knowledge we have achieved:

- How to create classes.
- How to apply the classes components to the game.
- How to write codes clearly and coherently.
- How to understand clearly the codes.
- How to help readers to understand the codes by adding comments
- Seperate the classes of objects to separate files to reduce the total length
- How to debug and fix codes.
- How to use Scene Builder to create scene for the game.
- How to use and fix the FXML files.
- How to work with controller to control the scenes and apply transitions between scenes smoothly.
- How to add animations to the scenes for animations.

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

In the future, we would love to make improvements and developments to our games. Our features are currently functioning normally; however we have a visual of making the players' card deck become folded when it comes to a player turn and become unfolded when it comes to other players' turn instead of making them invisible at the moment. moreover, we would like to make the transitions of the cards more smoothly and correctly between the card deck and the farms, the farms and the discard deck. There are also animations we would like to make for the introduction scene, game scene and the winner scene, which will make the game more animated and interested for players.

References

[1] O'Sullivan, Kevin, "Mini Review - Hick Hack in Gackelwack," Blogspot.com, 2025. Available at: https://tinyurl.com/yv5wnd4v. Accessed 22 Jan. 2025.

[2] Ubisoft, "UNO by Ubisoft Für Windows," Softonic, 2023. Available at: https://uno-by-ubisoft.de.softonic.com/?utm_source=bing&utm_medium=paid&utm_campaign=Bing_DE_CH_DSA_Desktop_6524&msclkid=30db4ea61dd11289a7ac29377322bf15. Accessed 10 Jan. 2025.

[3] D'Arcy, Sean, "Introducing Kahoot!+ Make Learning Awesome for the Entire Family," Kahoot!, 19 May 2021. Available at: https://kahoot.com/blog/2021/05/19/kahoot-plus-make-learning-awesome-entire-family/. Accessed 22 Jan. 2025.

[4] GeeksForGeeks, "JavaFX Tutorial," GeeksforGeeks, 5 Oct. 2021. Available at: https://www.geeksforgeeks.org/javafx-tutorial/.

[5] "Pick Picknic," BoardGameGeek, 2020. Available at: https://tinyurl.com/372abme5. Accessed 22 Jan. 2025.