Report for Java Card Game - Noch Verzwichter!

Final report of the ThuG1 Team: FKy for module OOP Java - WiSe 24/25

1st Duc Manh Pham
Computer Science
Frankfurt University of Applied Sciences
Frankfurt am Main, Germany
10422047@student.vgu.edu.vn

2nd Dinh Nam Vu
Computer Science
Frankfurt University of Applied Sciences
Frankfurt am Main, Germany
10422052@student.vgu.edu.vn

3rd Bao Long Pham
Computer Science
Frankfurt University of Applied Sciences
Frankfurt am Main, Germany
10422044@student.vgu.edu.vn

***Abstract*—This document is a report on the writing process and functionalities of our application written in Java for the final project of the module Java OOP WiSe 24/25 - a puzzle card game called "Noch Verzwickter!".**

***Index Terms*—Noch Verzwickter, card game, Game Board, design, architecture, and function.**

I. Introduction

This document will first briefly discuss the premise of this card game, then describe the structure and basic functions of our implementation, followed by the miscellaneous features that we have added to complete this project. This document will also report the distribution of work amongst members of our group.

II. The Card Game "Noch Verzwickter!"

Before we begin to discuss the Java application, we must first go through the specifics of this card game. "Noch Verzwickter!", "Tierisch Verzwickt!", "The Crazy Sheep Game!" and other similar games, henceforth referred to as Icon Matching puzzle games, all have a very simple setup and a set of rules. This is a puzzle game, meant for one player at a time, and, like most puzzle games, have ranging difficulties and a simple end state where the puzzle is solved.

A. Set Up and Rules of the Card Game

Most Icon Matching puzzle games have 9 or 16 square cards arranged in a 3×3 or 4×4 grid pattern, respectively. Each card consists of four icons, positioned at each edge of the card. Each icon belongs to a pair of icons that form a complete picture when put together in the correct orientation. Convention dictates that there are four different pairs of icons, for a total of eight unique icons. There exist versions of this game with different numbers of icon pairs.

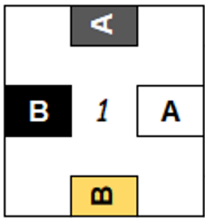

Fig. 1. Example of a card's layout, icons denoted as colored rectangles.

These cards will then be placed face-up in a 3×3 or 4×4 grid pattern, depending on the number of total cards. Any adjacent pair of cards will have one edge from one card line up with one edge from the other card. These edges will then be assessed as one pair by whether or not these edges' icons match up as one of the predetermined icon pairs. A 3×3 grid and a 4×4 grid would thus have a total of 12 and 21 assessable icon pairs, respectively.

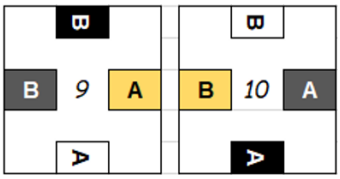

Fig. 2. Example of a matching pair of icons from two adjacent cards.

B. The Goal of the Game

The goal of the game is to orientate and order the cards in such a way that all adjacent edges from all pairs of adjacent cards match, given that the cards stay in the 3×3 or 4×4 grid formation. The player is allowed to swap any card's position with any other's, as well as rotate the cards any given amount of times. If all pairs of adjacent icons match, the puzzle is considered solved. It should be noted that one set of cards may have many different solutions.

III. Initial Analysis and Design

A. Requirements

This implementation follows a specific version of the Icon-Matching puzzle game, published as "Noch Verzwickter!".

Nine cards are played in a 3×3 grid pattern, with 4 different unique pairs of icons, represented as 4 pairs of animals. The project will be a desktop video game application, the majority of which would be built utilizing the JavaFX platform.

This program would be able to generate a puzzle, allow the player to interact with the cards via two different actions: rotating a card or swapping one card with another. Additionally, the player can use a check function to check if their solution is correct, in which case the game ends in a win state. Finally, the player can look up a solution for their given puzzle to serve as a hint.

The original card game only includes 9 predetermined printed cards, and thus can only create a very limited number of possible games. We will implement the functionality of generating new and randomized cards to potentially create a practically unlimited number of different possible games.

B. Application Design

Although the premise of the game suggests that we implement each element of the game (the cards, the icons, the edges, the board, etc...) separately, we found that the fastest and most optimized way was to simply implement the entire game board as one singular object.

Since we have 9 cards arranged in a 3×3 grid pattern, each card can be represented as an integer array of length 4 (for 4 icons) and the entire game board can be an array of these cards(integer arrays) of length 9. This ensures that any element of the game board can be accessed by index alone, making any further logic much simpler to implement. This would be implemented using the ArrayList Java object.

As we also have 4 unique pairs of icons, each icon is represented by an integer between -4 and 4, excluding zero: `{-4, -3, -2, -1, 1, 2, 3, 4}`. This way, determining whether a pair of icons matches is as simple as adding them up and checking if the sum equals zero. Additionally, displaying any icon onto the screen implements a simple hash map linking each integer value with an image URL string. This would be implemented using the HashMap Java object.

The game would be controlled by simple actions with a computer mouse, where dragging and dropping cards will allow the player to swap cards in a game board, and clicking will allow the player to rotate any given card.

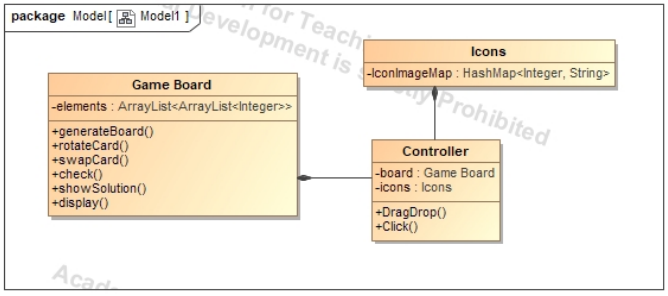

Fig. 3. Class Diagram for initial game design

IV. Implementation Details

This section describes the structure of the game application, key graphical user interface (GUI) elements, system design using a Class Diagram and a Use-case Diagram, external libraries used, and some notable code snippets. Here we will also discuss, in detail, the methods behind certain logical functionalities of the game, whether it serves gameplay purposes, or simply for better user experience.

*Note: For some logical functions, only pseudo-code will be provided, as the actual implementation looks a little more complex in order to deal with the somewhat unpredictable of some *fxml* elements in this particular environment.

A. Application Architecture

Built on the JavaFX Platform, the majority of game-play will happen on one main *fxml* file. Starting a new game will initialize a new (Game) Board as well as an instance of the Icons HashMap (AnimalImages). The game board would then be displayed as per specifications discussed below.

The program responds to the player's mouse input by detecting a drag-drop action or a click action. Actions on the Game Board displayed will trigger the logical functions of the game itself. Once an action finishes, the Game Board will be displayed again in order to update the UI with the new state of the board.

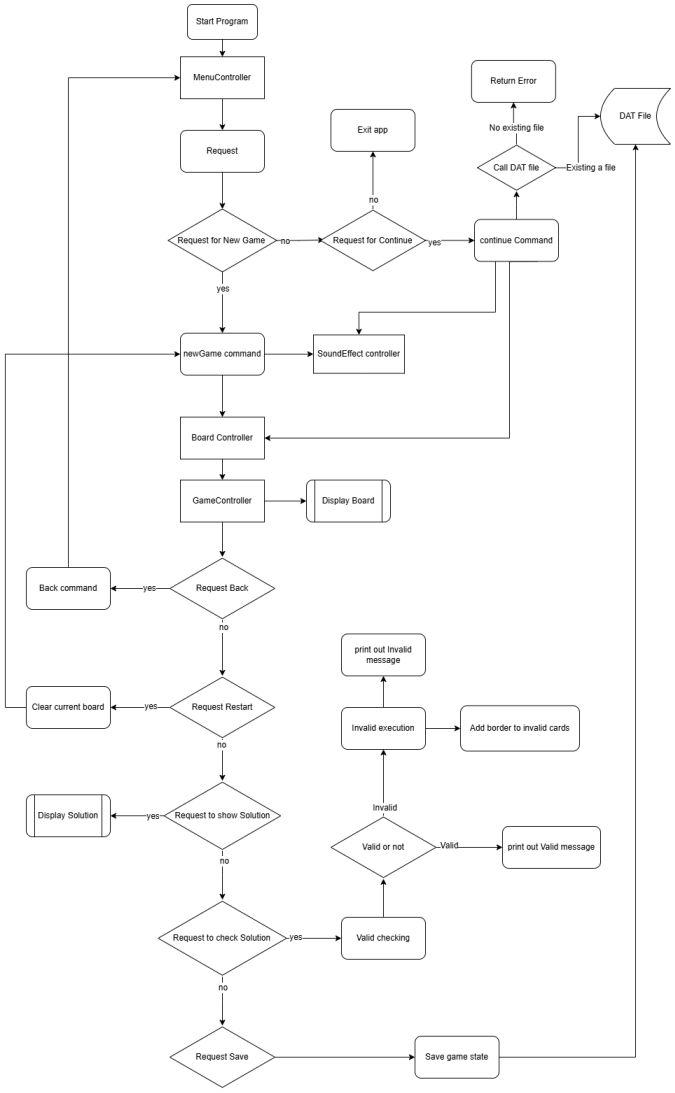

Fig. 4. Process flow diagram of the final Game system

B. Graphical User Interface

1) Main Menu: The **Main Menu** serves as the entry point of the game and provides users with essential navigation options. It features the game logo along with three interactive buttons:

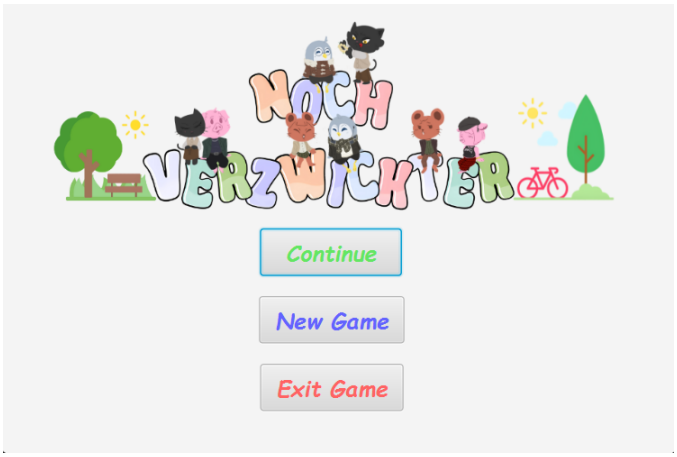

Fig. 5. Main Menu Interface

- **New Game**: Initiate a new game session by closing the current window and immediately launching the `GameBoard` window.
- **Continue**: Resume a previously saved game. It functions similarly to the `New Game` button but loads the last saved state before launching the `GameBoard`.
- **Exit**: Closes the game application by terminating the window.

2) Game Board: The **Game Board** displays a 3×3 grid of shuffled cards, where the player interacts directly by swapping and rotating cards to achieve the correct solution. The interface includes the following interactive buttons:

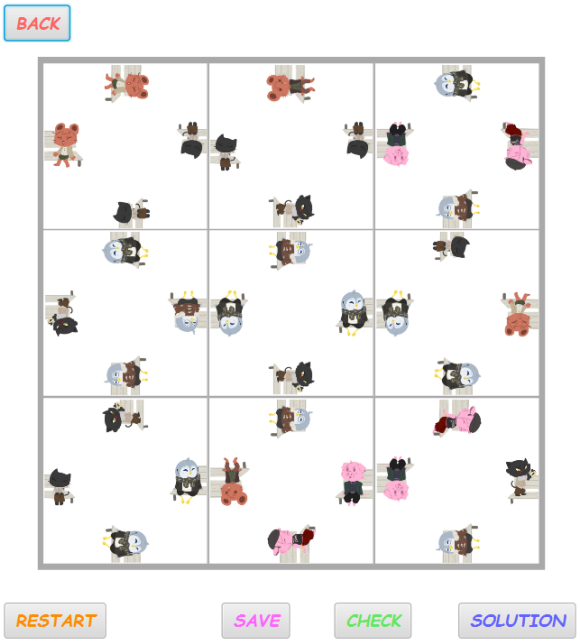

Fig. 6. Game Board Interface

- **Check**: Validates the current board state. If the solution

is correct, the result is shown as in Figure 7; otherwise, it shows the error message in Figure 8.

- **Solution**: Opens a separate window displaying the correct solution for the current board (Figure 9).
- **Save**: Saves the current board state, including the exact positions of the cards.
- **Restart**: Generates a new randomized game board for the player.
- **Back**: Returns to the `Main Menu`.

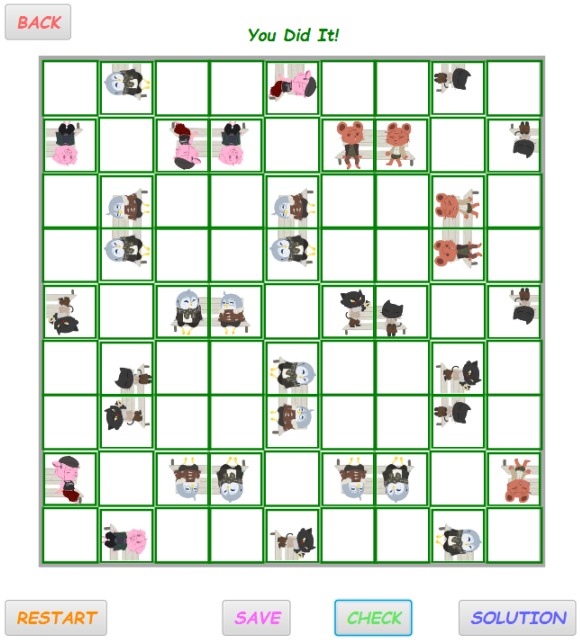

Fig. 7. Win State Achieved

From the win state, the player can continue to find more possible solutions, restart with a new game, or return to the main menu.

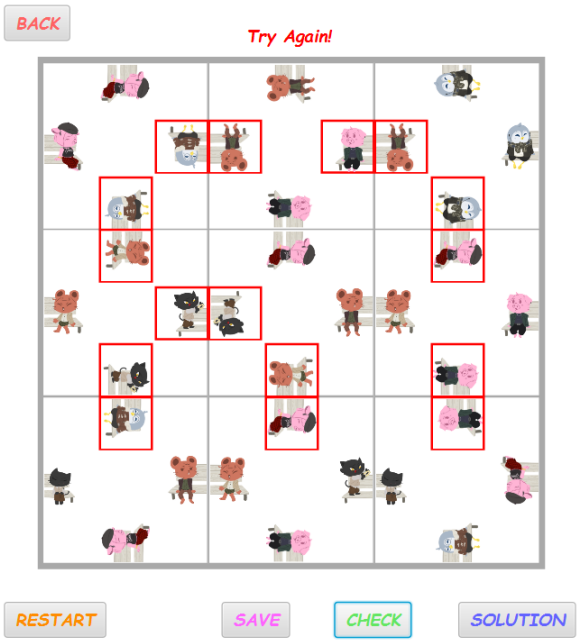

Fig. 8. Puzzle Incorrect Check

Note that only the incorrect icon pairs are highlighted in red, whereas the correct pairs are not. This helps to create a more user-friendly interface to improve the interactive experience.

Fig. 9. Small pop-up of a possible Solution

A small window will appear next to the main game window to allow the player to see an example solution to help guide them towards solving the puzzle.

C. System Design: Class Diagram and Use-case Diagram

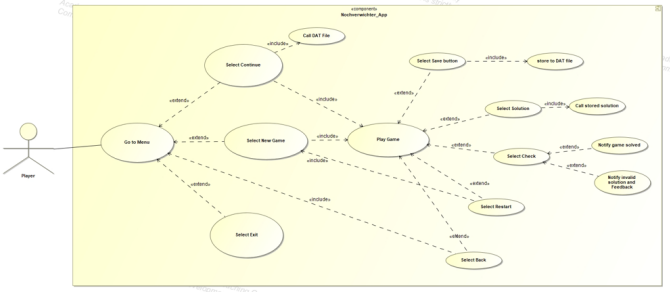

Fig. 10. Use-case diagram for player interactions

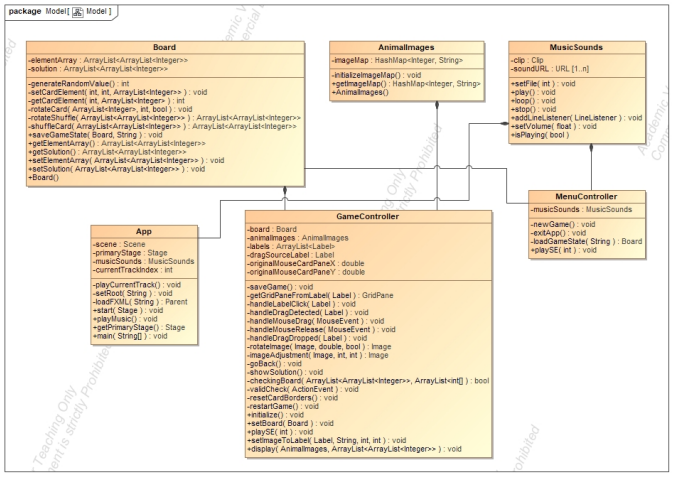

Fig. 11. Class diagram of the actual final Game system

1) Use-Case Diagram Overview: This section describes the interactions between the **Player** and the NochVerzwickter_App system as represented in the use-case diagram. The diagram illustrates the primary functions available to the user and how these functions interrelate. The key elements of the diagram include actors, use cases, and the relationships between them. Below is a detailed breakdown of these components.

a) Actor: The main actor in this system is the **Player**, who interacts with the application to perform various actions. The Player is responsible for initiating the game, navigating through menus, interacting with game elements, and managing game sessions.

b) Primary Use Case: The primary use case is **Go to Menu**, which serves as the entry point for the Player. From the Menu, the player can choose from multiple options such as:

- **Select Continue (extends)**: Resume a previously saved game.
- **Select New Game (extends)**: Start a new game session.
- **Select Exit (extends)**: Exit the application.

The **Go to Menu** use case is crucial as it sets the stage for all subsequent actions the Player can perform within the game.

2) Class Diagram Overview: The following class diagram represents the architecture of the game application, illustrating the relationships and functionalities of various classes within the system. It provides an overview of how different components interact to manage game logic, user interface, and multimedia elements.

This class diagram illustrates the final architecture of the game application, highlighting the relationships and interactions among various components. It provides a structured view of the system, showing how different classes contribute to game logic, user interface management, and multimedia integration.

The `Board` class is responsible for managing the game board using a two-dimensional `ArrayList<Integer>`. It provides methods for generating random configurations, shuffling, rotating elements, and saving game states. Additionally, it maintains the correct solution to validate user actions.

The `GameController` class serves as the central logic controller, integrating multiple components such as the `Board` and `AnimalImages`. It handles user interactions, including card movement, validation, and saving or loading game progress. Furthermore, it manages graphical user interface elements like labels and images, ensuring a seamless gaming experience.

The `App` class oversees the application's primary window and scene management. It is also responsible for handling background music using the `MusicSounds` class. This class initializes the game by loading *fxml* layouts and configuring essential components.

The `AnimalImages` class plays a crucial role in managing game assets. It maintains a mapping of game element images using a `HashMap<Integer, String>` and retrieves images dynamically when needed.

The `MusicSounds` class functions as the audio controller, managing background music and sound effects. It provides controls for playing, looping, stopping, and adjusting volume, and it supports event handling through `LineListener`.

The `MenuController` class is responsible for handling the main menu actions, such as starting a new game, exiting, and loading saved states. It also integrates with the `MusicSounds` class to manage sound effects associated with menu navigation.

In terms of relationships, the `GameController` serves as a bridge between the game logic (`Board`) and user interface elements. The `App` class manages the application's lifecycle, including window transitions and multimedia controls. The `MenuController` enables seamless navigation between different game states while ensuring smooth sound integration.

The class diagram effectively models the core functionalities of the game application. It demonstrates how users interact with a shuffled game board, save or load progress, and validate their solutions. Additionally, it highlights the role of multimedia elements, such as images and audio, in enhancing the gaming experience. By organizing these components efficiently, the system ensures smooth gameplay, persistence, and user feedback mechanisms.

D. Notable Logical Functions

This section discusses some of the more convoluted functions that might require some explanation for better comprehension of the inner workings of the entire application.

1) Generation of a new Board: Traditionally, this card game has a fixed collection of cards which can only lead to a fixed number of possible puzzles and solutions. To create a potentially endless number of possible games, all we need to do is find a way to generate random collections of solvable cards.

An intuitive solution is to randomly create a Board and check if it is solvable, then decide whether or not to generate a new Board. However, upon second inspection, we should be able to recognize that this approach is quite inefficient, as, in fact, the algorithm to determine whether an Icon-Matching puzzle is solvable or not is extremely complex with a high time complexity. Furthermore, if we force the player to wait for a valid solvable Board to generate every time they play, the wait time can vary wildly between relatively fast and potentially endless.

The solution we unanimously agreed upon was to generate a board with valid icon pairs first, where edges on the borders of the board (defined by the fact that they do not have any adjacent edges from other adjacent cards) can be randomized. Additionally, the valid icon pairs themselves are also random only apart from the fact that they match each other.

Since the game state of the entire Board can be represented as an ArrayList of an ArrayList of Integers, we can use indexing not only as a quick way to look up any icon's position from any card. Generating the board, then, follows that we first randomly generate all of the First Card's icons, then the same for the Second Card, except we make sure the Second Card's icon which is adjacent to the First Card matches said corresponding icon from the First Card. We do this procedurally for all nine Cards, making sure that every new card has matching icons with adjacent Cards that has already been generated.

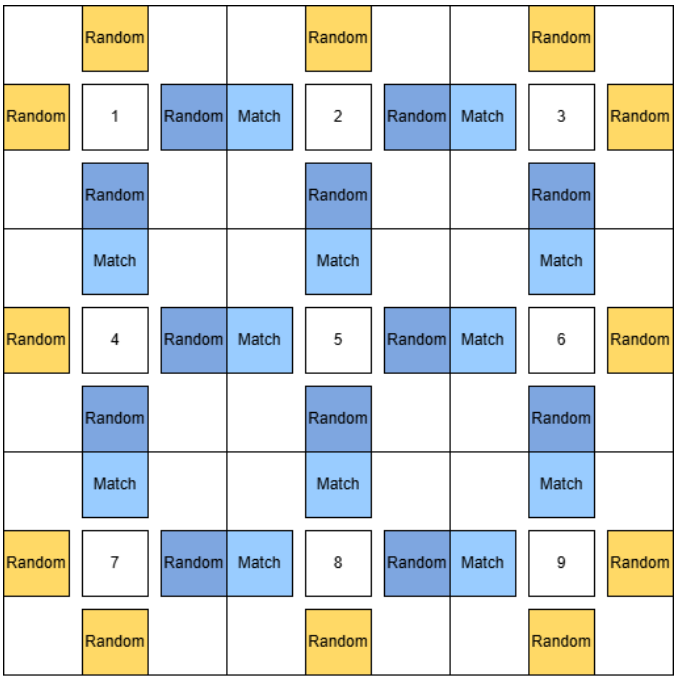

Fig. 12. Puzzle generation pattern, each card made in the denoted sequence.

Once generated, the solution will then be saved as its own array in the Board (for if the player wants to reference the solution). The array itself will then be shuffled and displayed to the player as their puzzle. The player can now start and try to solve the puzzle.

2) Automatic Image Rotation: The following method is designed to automatically rotate image from every edge in the proper way before displaying it in a label. To ensure that, images are oriented correctly based on their position and key parameter, which determines whether the image needs additional rotation.

We know that the order of labels on every card remains the same. The positions North, East, South, and West are represented by the variable `labelCount`, ranging from 0 to 3, respectively. Additionally, each image in our game is encoded with a key value ranging from -4 to 4 (excluding zero). These two factors, position and key value, are crucial in determining how the images should be rotated.

Images with keys greater than zero are considered right images, meaning the subject, such as an animal, is positioned on the right side. Let us first examine the North position (`labelCount = 0`). If the key is greater than zero, the image is rotated 90 degrees clockwise to ensure the correct orientation. Conversely, if the key is negative, the image is rotated 90 degrees counterclockwise.

Moving to the South position (`labelCount = 2`), the logic is applied in reverse. Here, images with keys less than zero, known as left images, are rotated 90 degrees clockwise, while those with positive keys are rotated 90 degrees counterclockwise to maintain consistency.

Now, let us consider the East and West positions (`labelCount = 1, 3`), which differ slightly from the previous cases. In the East position, if the key is negative, no rotation is applied, as the image is already in the correct orientation. However, if the key is positive, the image is rotated 180 degrees to align properly. To implement this, a conditional check ensures that only left images (`key > 0`) placed in the East position are rotated. For the West position, the transformation is applied in reverse: right images (`key < 0`) undergo a 180-degree rotation to achieve the correct alignment. This approach ensures that all images are displayed with the correct orientation based on their predefined key values and positional constraints.

```
// Method to adjust the image correctly in the label
private Image imageAdjustment(Image img, int
    keyImage, int labelCount) {
    if (labelCount == 0) {
        return rotateImage(img, 90, (keyImage > 0));
    } else if (labelCount == 1 && keyImage > 0) {
        return rotateImage(img, 180, true);
    } else if (labelCount == 2) {
        return rotateImage(img, 90, (keyImage < 0));
    } else if (labelCount == 3 && keyImage < 0) {
        return rotateImage(img, 180, false);
    } else {
        return img;
    }
}
```

Fig. 13. Images before rotation

Fig. 14. Images after rotation

3) Displaying the Game Board: The display of the board is a unique challenge, as the game generates 9 cards, each containing 9 labels, resulting in a total of 81 labels being managed simultaneously. However, as observed in the game, images are displayed only in specific positions on each card, namely North, East, South, and West. To handle this display mechanism efficiently, each card is assigned a numeric identifier ranging from 1 to 9, while labels are sequentially numbered from 0 to 80, as illustrated in Figures 15 and 16. By leveraging the order numbers of both cards and labels, specific conditions are applied to ensure a proper display of images.

The solution is structured around two primary conditions: handling odd and even order cards. A variable, `trackingCard`, is used to track the order number of each card. Additionally, the order number of each label plays a crucial role in determining which labels will contain and display images.

For cards with an odd order number (`trackingCard` is odd), only labels with an odd index (`i` is odd) will include an image. Conversely, for even-order cards, only labels with an even index (`i` is even) will display an image.

A separate variable, `labelCount`, is maintained to track the number of labels that have been assigned images. Since each card is designed to contain exactly four labeled images, once `labelCount` reaches 4, it is reset to 0. Simultaneously, `trackingCard` is incremented to move to the next card, and the label index is incremented accordingly to ensure the next label index will start at the new card.

For every label index that satisfies the display conditions, the image key is retrieved using a composite calculation based on the order number of the card (`trackingCard - 1`) and the label's tracking index (`labelCount`). Furthermore, each qualified label index (`i`) is used to call the `Label` object, which retrieves the corresponding label intended to store the image. Finally, the image is assigned to the appropriate label using the `setImageToLabel` function, ensuring the correct placement of images on the board.

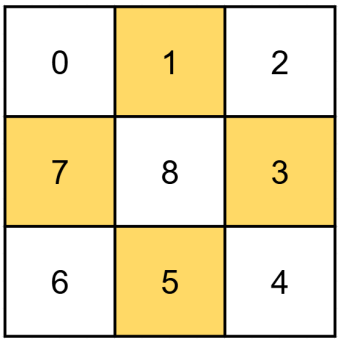

Fig. 15. Sample of odd card's order of labels

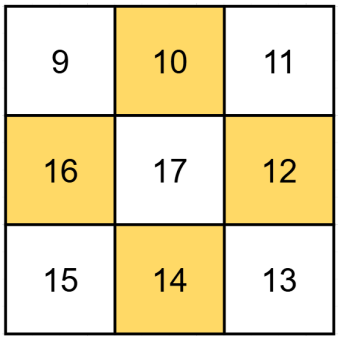

Fig. 16. Sample of even card's order of labels

```
// Method to display the board
public void display(AnimalImages animalImages,
    ArrayList<ArrayList<Integer>> elementArray) {
    /*
     * The new grid will be
     * 0 1(image) 2
     * 7(image) 8 3(image)
     * 6 5(image) 4
     */
    // Get the image map from the AnimalImages
        instance
    HashMap<Integer, String> imageMap = animalImages
        .getImageMap();

    // Iterate through the flattened elements and
        set images to labels
    int labelCount = 0;
    int trackingCard = 1;
    for (int i = 0; i < labels.size(); i++) {
        // Stop if trackingCard exceeds the number
            of groups in elementArray
        if (trackingCard > elementArray.size())
            break;

        if (trackingCard % 2 == 1 && i % 2 == 1) {
            int key = elementArray.get((trackingCard
                - 1)).get(labelCount);
            Label label = labels.get(i);
            if (imageMap.containsKey(key)) {
                // Get image from the map
                String imagePath = imageMap.get(key)
                    ;
                setImageToLabel(label, imagePath,
                    key, labelCount); // Set the
                    image to the label
            } else {
                label.setGraphic(null); // Clear the
                    label graphic if key not found
            }
            labelCount++;
            if (labelCount == 4) {
                labelCount = 0;
                trackingCard++;
                i++;
            }

        } else if (trackingCard % 2 == 0) {
            if (i % 2 == 0) {
                int key = elementArray.get((
                    trackingCard - 1)).get(
                    labelCount);
                Label label = labels.get(i);
                if (imageMap.containsKey(key)) {
                    // Get image from the map
                    String imagePath = imageMap.get(
                        key);
                    setImageToLabel(label, imagePath
                        , key, labelCount); // Set
                        the image to the label
                } else {
                    label.setGraphic(null); // Clear
                        the label graphic if key
                        not found
                }
                labelCount++;
                if (labelCount == 4) {
                    labelCount = 0;
                    trackingCard++;
                    i++;
                }
            }
        }
    }
}
```

4) Validation of Player's Solution: In this part, we present the method used to determine the validity of a player's given solution. As illustrated in Figure 7 and Figure 8, which depict the winning and invalid states respectively, our validation method ensures that the provided solution is checked thoroughly. If the solution is incorrect, the system highlights the invalid pairs, providing visual feedback to guide the player in making necessary adjustments.

The core idea of this validation method is based on the predefined encoding of images, which ranges from -4 to 4 (excluding zero). By leveraging this pattern, we can systematically validate the placement of images. Figures 17 and 18 illustrate how each image position is represented as a pair of numbers $[a, b]$, where a denotes the card index and b represents the label index. Using these pairs, we retrieve the corresponding encoded values (keys) assigned to each image.

To determine the validity of an image pair, we extract the keys associated with two adjacent labels, for example, $[0, 1]$ and $[1, 3]$. A pair is considered valid if the key of the left label is the negative value of the key of the right label. This ensures that the images align correctly, as demonstrated in Figure 17. Conversely, if this condition is not met, the pair is classified as invalid, as seen in Figure 18.

The code sample below demonstrates the validation logic for a single pair of labels. However, the complete validation

process involves checking all 12 pairs of labels on the board. A solution is deemed correct only if all 12 pairs satisfy the validation condition. If any pair is invalid, the solution is rejected.

```
@FXML
private boolean checkingBoard(ArrayList<ArrayList<
    Integer>> elementArray, ArrayList<int[]>
    invalidConditions) {
    boolean[] validCond = {
            (elementArray.get(0).get(1) == (-
                elementArray.get(1).get(3))),
            /* The rest of code*/
            }
    invalidConditions.clear();

        for (int i = 0; i < validCond.length; i++) {
            if (!validCond[i]) {
                switch (i) {
                    case 0:
                        invalidConditions.add(new
                            int[] { 0, 1, 1, 3 });
                        break;
                    /*The rest of code */
            }
        }
// Method to validate the board and display the
    result
@FXML
private void validCheck(ActionEvent event) {
    ArrayList<int[]> invalidCards = new ArrayList<
        int[]>();
    boolean isValid = checkingBoard(board.
        getElementArray(), invalidCards); // Call
        the validation method
    /*The rest of code*/
     // Add border to invalid cards
        for (int[] invalidPairs : invalidCards) {
            if (Arrays.equals(invalidPairs, new int
                [] { 0, 1, 1, 3 })) {
                labels.get(3).setStyle("-fx-border-
                    color:␣red;␣-fx-border-width:␣2"
                    );
                labels.get(16).setStyle("-fx-border-
                    color:␣red;␣-fx-border-width:␣2"
                    );

            }
            /*The rest of code*/
        }
```

To enhance user experience, our system provides feedback by marking incorrect pairs. This is implemented in two functions: `checkingBoard` and `validCheck`. The `checkingBoard` function evaluates each pair and stores invalid ones in an array list called `invalidConditions`. Additionally, it returns a boolean value indicating whether the board configuration is valid. The `validCheck` function then performs two tasks: first, it updates the game interface to reflect an invalid solution if necessary, and second, it highlights the incorrect pairs stored in the `invalidConditions` list by applying a visual border around them.

This validation approach ensures that players receive clear and immediate feedback, facilitating a better gameplay experience and making the solution-checking process more intuitive.

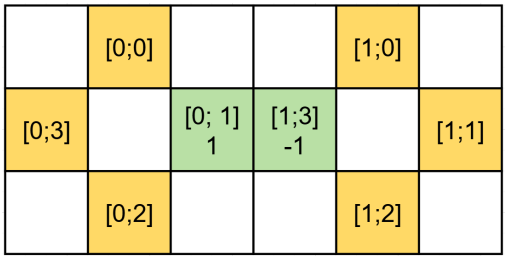

Fig. 17. Example of a valid pair

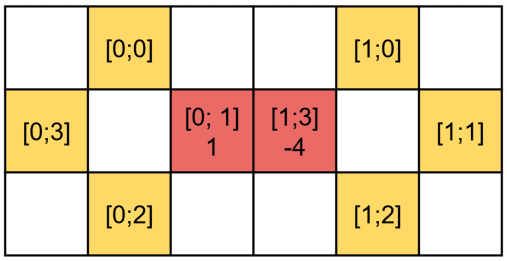

Fig. 18. Example of an invalid pair

5) Interface Interaction Implementation: As depicted in Figures 15 and 16, each card itself consists of 9 label elements, despite us using only 4 of which to actually display the image icons. This is because all 9 are necessary to detect mouse click and drag drop events. These labels are predefined *fxml* elements in the `GameController` class, and thus are appropriately indexed to efficiently find the card in which they belong.

A click event on any given card will trigger the rotate function on said card. This function is purely logical and thus only rotates the values in the Board's element array. To actually create a smooth rotating animation, we used the built-in animation package from `JavaFX`. However, due to the unpredictable behavior of the built-in animation function within our environment, we have the animation quickly reset after playing, before calling the actual logical rotate function that we have already written, as to keep the animation purely cosmetic.

```
@FXML
private void handleLabelClick(Label label) {
    // Derive the card element in the FXML file -
        represented as a GridPane element - from a
        label
    GridPane cardPane = getGridPaneFromLabel(label);

    // Card rotation animation class
    RotateTransition rotateTransition = new
        RotateTransition(Duration.millis(200),
        cardPane);
    rotateTransition.setByAngle(90);
    rotateTransition.setCycleCount(1);
    rotateTransition.setInterpolator(Interpolator.
        EASE_BOTH);
```

```
    rotateTransition.setOnFinished(event -> {
        // Find the label's index
        int labelIndex = labels.indexOf(label);

        // Determine the current section
        // Each section spans 9 labels (0-8, 9-17,
            etc.)
        // Section index can thus determine card's
            index
        int section = labelIndex / 9;
        ArrayList<Integer> card = board.
            getElementArray().get(section);
        board.rotateCard(card, 1, true);
        // Reset the animation to keep the function
            behaving corrrectly
        cardPane.setRotate(0);
        // Update the display
        display(animalImages, board.getElementArray
            ());
    });

        rotateTransition.play();
}
```

To detect Dragging and Dropping for any given card, we used the same methodology as in Clicking, where dragging on any given label element drags the entire Card attached to said label. The Drag and Drop logic is simple to implement logically, but to allow the player to see the card physically move as they drag and drop, the process is a little more convoluted.

As dragging one card and dropping it into another card triggers the swap function to exchange the positions of said two cards, we need the drag method to save the dragged card into a global variable. Once dropped (a mouse release event), we can then pass the aforementioned variable into the swap function with the card where the mouse release event happened.

Similar to the way Rotation works, and also due to similar issues, what the player is seeing is purely visual, as the dragged card is actually quickly returned right back to its original position before the logical swap is called. This is also useful, as the logical swap takes care of edge cases where a player might drag a card but not drop it onto another card, but instead some other undefined part of the screen.

```
@FXML
// Method to handle drag detection
private void handleDragDetected(Label label) {
    // Save the dragged label as a global variable
        called dragSourceLabel
    dragSourceLabel = label;

    // Get the corresponding card for that label
    GridPane cardPane = getGridPaneFromLabel(label);

    // Set behaviour for mouse drag
    cardPane.setOnMouseDragged(this::handleMouseDrag
        );
    cardPane.setOnMouseReleased(this::
        handleMouseRelease);
}
```

```
// Method to handle mouse drag
private void handleMouseDrag(MouseEvent event) {
    GridPane cardPane = (GridPane) event.getSource()
        ;

    // Makes sure the card follows the mouse's
        movement
    // Note: actual implementation looks a little
        different due to some clipping that happens
        with the mouse when the card is dragged
        along, making it impossible to drop the card
         otherwise.
    cardPane.setLayoutX(event.getSceneX() -
        originalMouseCardPaneX);
    cardPane.setLayoutY(event.getSceneY() -
        originalMouseCardPaneY);
}


// Method to handle mouse release
private void handleMouseRelease(MouseEvent event) {
    // Get the card that the mouse dropped on
    GridPane cardPane = (GridPane) event.getSource()
        ;

    // Stop listening for mouse drag events after
        release
    cardPane.setOnMouseDragged(null);
    cardPane.setOnMouseReleased(null);

    // Sets the cardPane back to where it was from,
        for the actual logical swap to happen
    // X and Y happens to always be -1, since this
        is the card's coordinate relative to it's
        parent container
    cardPane.setLayoutX(-1);
    cardPane.setLayoutY(-1);
}
```

Please do note that the above code are not exactly how the functions are implemented, as *fxml* elements can be quite unpredictable when moving around. The actual code for these sections are a little more complicated.

E. Used Libraries

This section provides an overview of the external libraries used in the game application, detailing their roles in functionality and development. JavaFX libraries are utilized for user interface components, animations, and event handling, ensuring a smooth and interactive experience.

The `java.util` package offers essential data structures like `ArrayList` and `Collections`, which manage game elements efficiently. Additionally, `javax.sound.sampled` handles audio playback for in-game sound effects. Other libraries, such as `java.io` and `java.net`, manage file input/output operations and network resources. By leveraging these libraries, the application achieves modularity, reusability, and an efficient coding structure, enhancing both performance and maintainability.

TABLE I
LIST OF USED LIBRARIES AND PURPOSE

Imported Library	Purpose
`java.util.*`	Core Java utilities
`java.util.ArrayList`	Manage dynamic lists
`java.util.Collections`	Sort and shuffle lists
`java.util.Random`	Generate random values
`javafx.application.Application`	JavaFX application entry point
`javafx.fxml.FXMLLoader`	Load FXML UI files
`javafx.scene.Parent`	Base class for UI nodes
`javafx.scene.Scene`	Define window content
`javafx.stage.Stage`	Create main application window
`javafx.scene.control.Button`	Handle button interactions
`javafx.scene.control.Label`	Display text and images in UI
`javafx.scene.image.Image`	Load images into UI
`javafx.scene.image.ImageView`	Manipulate images in UI
`javafx.scene.text.Font`	Manage text styles
`javafx.animation.FadeTransition`	Create fade effects
`javafx.animation.Interpolator`	Smooth animations
`javafx.animation.PauseTransition`	Delay actions
`javafx.animation.RotateTransition`	Rotate UI elements
`javafx.event.ActionEvent`	Handle button events
`javafx.scene.input.MouseEvent`	Capture mouse interactions
`javafx.scene.layout.GridPane`	Create grid-based layouts
`javafx.scene.layout.Pane`	Manage UI components
`javafx.scene.canvas.Canvas`	Draw graphics manually
`javafx.scene.canvas.GraphicsContext`	Render shapes and images
`javafx.application.Platform`	Manage JavaFX threads
`javax.sound.sampled.AudioInputStream`	Load audio data
`javax.sound.sampled.AudioSystem`	Manage sound playback
`javax.sound.sampled.Clip`	Play short sound effects
`javax.sound.sampled.LineListener`	Monitor audio events
`javax.sound.sampled.LineEvent`	Handle sound events i.e. running, stopping, etc...
`java.net.URL`	Handle sound files as URL paths
`java.io.IOException`	Handle input/output errors

V. EXPERIMENTAL RESULTS, STATISTICAL TESTS, RUNNING SCENARIOS

A. Simulations and Parameter Tuning

To evaluate the performance of our puzzle game, we conducted simulations by adjusting key parameters such as the time constraints and user interaction patterns. The game was tested exclusively with a 3x3 puzzle grid, as it is the only available level in this version. Through extensive testing and user experience analysis, we observed that no player was able to successfully solve the puzzle at this difficulty level. Based on these findings, we have decided to refine and update the puzzle mechanics in the next version to improve playability and balance.

TABLE II
NOCH VERZWICHTER CONFIGURATION PARAMETERS

Parameter	Value Range	Description
Grid Size	3x3	Number of puzzle pieces
Time Limit	Unknown	Maximum time per puzzle
Hints Allowed	0	Number of hints available

B. Input Data and Algorithm Comparisons

This puzzle game does not rely on user-provided input data; instead, the only data utilized are the images that contribute to the game's visual representation.

Regarding the algorithms, all implementations within the game were conceptualized and developed entirely by our team, without external references. As a result, there are currently no alternative algorithms available for direct comparison. However, we assume that the existing algorithms are optimized for efficient gameplay. To facilitate meaningful algorithmic comparisons, further research and development are required. Future updates will introduce additional algorithmic approaches, allowing for more comprehensive evaluations aimed at enhancing the overall gaming experience.

C. Performance Results and Statistical Tests

As this game is a puzzle-based application, we do not currently collect performance data or conduct statistical tests to evaluate its efficiency. Since the gameplay does not involve algorithmic benchmarking or complex computational processes, traditional performance metrics such as execution time, memory usage, or success rates have not been analyzed.

However, in future updates, we plan to integrate user performance tracking, including completion times, the number of moves taken, and success rates. This data will allow for statistical analysis to assess difficulty levels and optimize the gaming experience. Additionally, comparative studies on different algorithmic approaches may be conducted to enhance efficiency and responsiveness in future iterations of the game.

VI. CONCLUSIONS AND FUTURE WORK

A. Work Distribution and Lessons

The workload has been very well distributed among the three members of our team. Each of us contributed to the planning, drafting, researching, and building the application. Every file in this project has been authored by all three team members to some extent. Each member shared responsibility with writing and debugging nearly every part of the program.

Music used for the game are royalty-free, from open sources. The sound effects used are hand-crafted and

recorded by us. All art work are also original, designed and completed by members of our team.

Throughout this project, we have learned an extraordinary amount about Java development and application building in general. We learned some of the most efficient and thoroughly used techniques in the industry as well as many of the pitfalls and drawbacks of using certain tools and libraries. This has been a challenging experience for all of us and it has taught us a lot about problem solving with code as well as communicating ideas to be implemented in code.

B. Potential Future Work

As the premise of this puzzle card game is relatively simple, we are mostly satisfied with the current state of the project. The most basic and essential features have been implemented and many of the play features are already present. There are, however, more that can potentially be done for this video game.

This game has a similar puzzle structure to some of the most popular puzzles in the world like Sudoku and the Rubik's Cube. This means that we can theoretically create algorithms to generate games with different levels of difficulty, controlling which aspects of the board can be shuffled and by how much.

Furthermore, solving algorithms can be developed, to allow the program to give players hints of useful moves instead of simply letting the player see the solution. As far as we have researched, unlike Sudoku and the Rubik's Cube, no agreed-upon algorithm has been documented for reliably solving Icon Matching puzzle games.

Once difficulty levels can be reliably and consistently generated, we could potentially add an online competitive element to the project, allowing different players to compete for better solving times against each other, updating high scores for different difficulty categories online.

New and more dynamic animations can also be applied. As this is a puzzle game revolving around matching up anthropomorphic animal couples, there is a lot of room for creativity when it comes to the art work and potentially, cut scenes and character animations. The theme of the game can also change as we wish. Since the icon pairs can potentially be anything, we could also change out art assets to create different looking Game Boards without changing anything about the functionalities of the game.

As the game currently only includes 3×3 boards, as to match the original paper card game, new levels for 4×4, 5×5 or even bigger Game Boards can also be added for increased difficulty. The logic would stay relatively similar though it would take a more generalized pattern of algorithmic functions, which might take a long time to realistically implement.

C. Conclusions

This project has been a challenging learning experience for all of us. Since there has been no documented attempts at digitalizing this particular game, almost the entirety of our code base has been originally planned and written, with barely any pre-existing guides and tips. We have worked very hard to complete this program and we hope it can provide a great experience for anyone interested in puzzle games.

ACKNOWLEDGMENT

We would like to extend our gratitude towards Dr. Doina Logofatu, Mr. Sheikh Sharfuddin Mim, and Mr. Jiban Kumar Ray for your guidance and assistance throughout the process of working on our project.You have provided us with helpful and relevant documentations and tutorials, as well as personal advice and proofreading our concepts. We couldn't have done it without your help.

REFERENCES

No external references were used in this report. All concepts, algorithms, and implementations were originally written and developed by members of our team.