EÖTVÖS LORÁND UNIVERSITY
INFORMATICS FACULTY
PROGRAMMING TECHNOLOGY COURSE
(Template belongs to Vilnius University)

Rubik Clock

Documentation of the Assignment No. 2, Task 10

Author: Monika Mirbakaite (YS0EG6)

ELTE email: ys0eg6@inf.elte.hu

Evaluated by: Lecturer Szendrei Rudolf,
Instructor Török Zoltán Ákos

Budapest
2024

1. DESCRIPTION

Create a game, which implements the Rubik clock. In this game there are **9 clocks**. Each clock can show a time between 1 and 12 (hour only). Clocks are placed **in a 3x3 grid**, and initially they set randomly. Each four clocks on a corner has a button placed between them, so we have **four buttons** in total. Pressing a button increase the hour on the four adjacent clocks by one. The player wins, if all the clocks show 12.

Implement the game, and let the player restart it. The game should recognize if it is ended, and it has to show in a message box how much steps it took to solve the game. After this, a new game should be started automatically.

2. ANALYSIS

2.1. Task Analysis (Key Points)

2.1.1. Game Elements

- A *3x3 grid* of *clocks* that can show an hour *between 1 and 12*.
- *Four buttons* placed between the clocks in the corners of the grid. *Each button affects four clocks* (clocks on its corner).
- The *goal* is to adjust *all clocks to show 12*.

2.1.2. Game Flow

- Each clock starts with a random hour (between 1 and 12).
- Pressing a button increases the hour on the four adjacent clocks by 1.
- The player wins by setting all clocks to 12.
- When all clocks show 12, the game ends and a message box shows the number of steps taken to solve it.
- After the message, a new game should automatically restart, randomizing the clocks again.

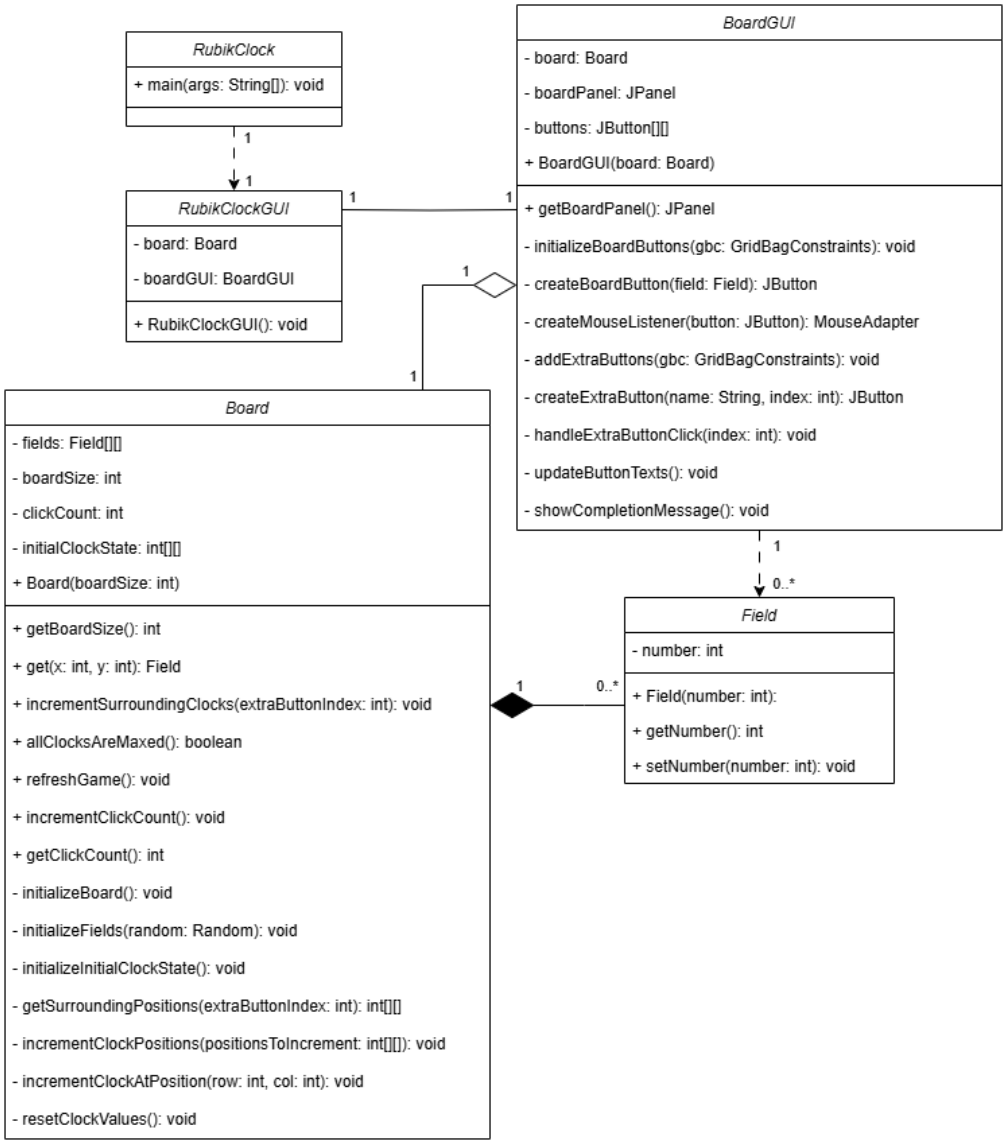
2.2. Solution Plan (Implementation Steps)

1. *Class diagram*. It will help to figure out the elements and how they interact with each other.
2. *Implementing the classes*. Define the classes with source code.
3. *Implementing the responsibilities in the game of each class*. Define the purpose of each class.
4. *Implementing GUI*. To be able to successfully start and play the game in a user-friendly manner.
5. *Implementing needed testing*. To ensure that the program is working properly.

3. CLASS DIAGRAM

In Picture 1 a class diagram of Rubik Clock game is displayed:

- *Rubik Clock* has exactly one *user interfa*ce, *Rubik Cloc*k depends on Rubik Clock *user interface* because it creates an instance of *Rubik Clock GUI* to start the application's GUI.
- One *Rubik Clock user interface* has one *user interface of Board*.
- One *Board* has one *user interface of Board*. *Board GUI* has a reference to *Board*, but *Board* could exist independently of *Board GUI*.
- One *Board* can have many *Field*s. The *Field* objects are created and managed by *Board* and cannot exist independently, making this a composition.
- Because *Board GUI* is connected to *Board* it also can have many *Field*s. It does not own *Field* instances directly but depends on them to function correctly.



Picture 1. Rubik Clock Class Diagram.

4. IMPLEMENTATION

4.1. Package gameLogic

4.1.1. RubikClock (Main)

```
package gameLogic;

import gameGUI.RubikClockGUI;

/**
 * The RubikClock class contains the main method to launch the Rubik's Clock game.
 * It creates an instance of the RubikClockGUI to display the user interface.
 */
public class RubikClock {

    /**
     * The main method to launch the Rubik's Clock game.
     * Initializes the GUI by creating an instance of RubikClockGUI.
     *
     * @param args
     */
    public static void main(String[] args) {
        RubikClockGUI gui = new RubikClockGUI();
    }
}
```

4.1.2. Field

```
package gameLogic;

/**
 * The Field class represents a single field or cell on the game board.
 * Each Field object holds an integer value representing a number, which can be
updated.
 */
public class Field {
    private int number;

    /**
     * Constructs a Field with a specified initial number.
     *
     * @param number The initial number for this field.
     */
    public Field(int number) {
        this.number = number;
    }

    /**
     * Gets the current number in this field.
     *
     * @return The number in this field.
     */
    public int getNumber() {
        return number;
    }

    /**
     * Sets a new number for this field.
     *
     * @param number The new number to set in this field.
     */
    public void setNumber(int number) {
```

```
        this.number = number;
    }
}
```

4.1.3. Board

```
package gameLogic;

import java.util.Random;

/**
 * The Board class represents the game board for a Rubik's Clock puzzle.
 * It manages a grid of Field objects, each representing a clock with a number from 1
to 12.
 * The board handles initialization, incrementing clocks, and checking the completion
status.
 */
public class Board {
    private Field[][] fields;
    private int boardSize;
    private int clickCount = 0;
    private int[][] initialClockState;

    /**
     * Constructs a Board of a specified size and initializes its state.
     *
     * @param boardSize The size of the board (e.g., 3 for a 3x3 board).
     */
    public Board(int boardSize) {
        this.boardSize = boardSize;
        fields = new Field[boardSize][boardSize];
        initialClockState = new int[boardSize][boardSize];
        initializeBoard();
    }

    /**
     * Initializes the board with random clock values and saves the initial state.
     */
    private void initializeBoard() {
        Random random = new Random();
        initializeFields(random);
        initializeInitialClockState();
    }

    /**
     * Initializes each Field on the board with a random clock value between 1 and 12.
     *
     * @param random Random object used to generate random clock values.
     */
    private void initializeFields(Random random) {
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                fields[i][j] = new Field(random.nextInt(12) + 1);
            }
        }
    }

    /**
     * Saves the initial clock state of each Field to allow game reset.
     */
    private void initializeInitialClockState() {
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                initialClockState[i][j] = fields[i][j].getNumber();
            }
        }
```

```
    }

    /**
     * Returns the size of the board.
     *
     * @return The board size.
     */
    public int getBoardSize() {
        return boardSize;
    }

    /**
     * Retrieves the Field at a specific location on the board.
     *
     * @param x The row index of the field.
     * @param y The column index of the field.
     * @return The Field object at the specified position.
     */
    public Field get(int x, int y) {
        return fields[x][y];
    }

    /**
     * Increments the clocks in positions surrounding the given extra button's index.
     *
     * @param extraButtonIndex The index of the extra button clicked.
     */
    public void incrementSurroundingClocks(int extraButtonIndex) {
        int[][] positionsToIncrement = getSurroundingPositions(extraButtonIndex);
        incrementClockPositions(positionsToIncrement);
    }

    /**
     * Returns the positions of the fields surrounding the given extra button index.
     *
     * @param extraButtonIndex The index of the extra button clicked.
     * @return An array of integer pairs representing the surrounding positions.
     */
    private int[][] getSurroundingPositions(int extraButtonIndex) {
        int[][][] surroundingPositions = {
                {{0, 0}, {0, 1}, {1, 0}, {1, 1}},
                {{0, 1}, {0, 2}, {1, 1}, {1, 2}},
                {{1, 0}, {1, 1}, {2, 0}, {2, 1}},
                {{1, 1}, {1, 2}, {2, 1}, {2, 2}}
        };
        return surroundingPositions[extraButtonIndex];
    }

    /**
     * Increments the clock values at specified positions on the board.
     *
     * @param positionsToIncrement An array of row and column indices to increment.
     */
    private void incrementClockPositions(int[][] positionsToIncrement) {
        for (int[] pos : positionsToIncrement) {
            int row = pos[0];
            int col = pos[1];
            incrementClockAtPosition(row, col);
        }
    }

    /**
     * Increments the clock value at a specified position, wrapping back to 1 if it
exceeds 12.
     *
     * @param row The row index of the clock.
```

```
     * @param col The column index of the clock.
     */
    private void incrementClockAtPosition(int row, int col) {
        if (row < boardSize && col < boardSize) {
            Field field = fields[row][col];
            int currentNumber = field.getNumber();
            if (currentNumber < 12) {
                field.setNumber(currentNumber + 1);
            }
        }
    }

    /**
     * Checks if all clocks on the board are set to 12.
     *
     * @return True if all clocks are set to 12; otherwise, false.
     */
    public boolean allClocksAreMaxed() {
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                if (fields[i][j].getNumber() < 12) {
                    return false;
                }
            }
        }
        return true;
    }

    /**
     * Resets the game to its initial clock values and resets the click count.
     */
    public void refreshGame() {
        resetClockValues();
        clickCount = 0;
    }

    /**
     * Resets all clock values to their initial state.
     */
    private void resetClockValues() {
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                fields[i][j].setNumber(initialClockState[i][j]);
            }
        }
    }

    /**
     * Increments the click count by one.
     */
    public void incrementClickCount() {
        clickCount++;
    }

    /**
     * Returns the current number of clicks made.
     *
     * @return The number of clicks.
     */
    public int getClickCount() {
        return clickCount;
    }
}
```

4.2. Package gameGUI

4.2.1. RubikClockGUI

```
package gameGUI;

import gameLogic.Board;

import javax.swing.*;
import java.awt.*;

/**
 * The RubikClockGUI class represents the graphical user interface for displaying
 * a Rubik's Clock board.
 */
public class RubikClockGUI {

    /**
     * Constructs a RubikClockGUI instance and initializes the main application
window.
     * Sets up a 3x3 board and displays it in a JFrame.
     */
    public RubikClockGUI() {
        // Create a new 3x3 game board instance
        Board board = new Board(3);

        // Create a BoardGUI to represent the board visually
        BoardGUI boardGUI = new BoardGUI(board);

        // Set up the JFrame containing the game board
        JFrame frame = new JFrame("Rubik's Clock Board");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());

        // Add the board panel to the center of the frame
        frame.add(boardGUI.getBoardPanel(), BorderLayout.CENTER);

        // Adjust frame settings and make it visible
        frame.pack();
        frame.setMinimumSize(new Dimension(400, 400));
        frame.setVisible(true);
    }
}
```

4.2.2. BoardGUI

```
package gameGUI;

import gameLogic.Board;
import gameLogic.Field;

import javax.swing.*;
import java.awt.*;

/**
 * The BoardGUI class represents the graphical user interface for a game board.
 */
public class BoardGUI {
    private Board board;
    private final JPanel boardPanel;
    private JButton[][] buttons;

    /**
     * Constructs a new BoardGUI instance for a game board.
     *
     * @param board
```

```
     */
    public BoardGUI(Board board) {
        this.board = board;
        int boardSize = board.getBoardSize();
        buttons = new JButton[boardSize][boardSize];

        boardPanel = new JPanel(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.fill = GridBagConstraints.BOTH;
        gbc.insets = new Insets(5, 5, 5, 5);
        gbc.weightx = 1.0;
        gbc.weighty = 1.0;

        initializeBoardButtons(gbc);
        addExtraButtons(gbc);
    }

    /**
     * Initializes the buttons representing the fields on the board and adds them to
the panel.
     *
     * @param gbc The GridBagConstraints used for layout management.
     */
    private void initializeBoardButtons(GridBagConstraints gbc) {
        int boardSize = board.getBoardSize();
        for (int i = 0; i < boardSize; ++i) {
            for (int j = 0; j < boardSize; ++j) {
                Field field = board.get(i, j);
                JButton button = createBoardButton(field);
                buttons[i][j] = button;
                gbc.gridx = j * 2;
                gbc.gridy = i * 2;
                boardPanel.add(button, gbc);
            }
        }
    }

    /**
     * Creates a JButton representing a field on the game board.
     *
     * @param field The Field object containing the data for this button.
     * @return A JButton configured with the field's properties.
     */
    private JButton createBoardButton(Field field) {
        JButton button = new JButton();
        button.setText(String.valueOf(field.getNumber()));
        button.setFont(new Font("Arial", Font.BOLD, 20));
        button.setEnabled(false);
        button.setPreferredSize(new Dimension(90, 90));
        button.setBackground(new Color(230, 230, 250));
        button.setForeground(Color.BLACK);
        button.setFocusPainted(false);
        button.setCursor(new Cursor(Cursor.HAND_CURSOR));

        // No MouseListener added here, so no color change on hover.
        return button;
    }

    /**
     * Adds extra control buttons to the board panel at specific positions.
     *
     * @param gbc The GridBagConstraints used for layout management.
     */
    private void addExtraButtons(GridBagConstraints gbc) {
        String[] extraButtonNames = {"Button 1", "Button 2", "Button 3", "Button 4"};
        int[][] positions = {
```

```
                {1, 1}, {1, 3}, {3, 1}, {3, 3}
        };

        for (int i = 0; i < 4; i++) {
            JButton extraButton = createExtraButton(extraButtonNames[i], i);
            int x = positions[i][0];
            int y = positions[i][1];
            gbc.gridx = y;
            gbc.gridy = x;
            boardPanel.add(extraButton, gbc);
        }
    }

    /**
     * Creates an extra control button with a specified label and action index.
     *
     * @param name The label to display on the button.
     * @param index The index used to determine which action to perform on button
click.
     * @return A JButton configured with the specified properties.
     */
    private JButton createExtraButton(String name, int index) {
        JButton extraButton = new JButton(name);
        extraButton.setFont(new Font("Arial", Font.BOLD, 14));
        extraButton.setEnabled(true);
        extraButton.setPreferredSize(new Dimension(100, 50));
        extraButton.setBackground(new Color(255, 223, 186));
        extraButton.setForeground(Color.BLACK);
        extraButton.setFocusPainted(false);
        extraButton.setCursor(new Cursor(Cursor.HAND_CURSOR));

        extraButton.addActionListener(e -> handleExtraButtonClick(index));
        return extraButton;
    }

    /**
     * Handles the click action for the extra control buttons.
     * Increments the surrounding fields' clocks and updates the board display.
     * Checks if all clocks are set to 12 and shows a completion message if so.
     *
     * @param index The index of the clicked extra button.
     */
    private void handleExtraButtonClick(int index) {
        board.incrementSurroundingClocks(index);
        board.incrementClickCount();
        updateButtonTexts();
        if (board.allClocksAreMaxed()) {
            showCompletionMessage();
        }
    }

    /**
     * Updates the text displayed on each button to reflect the current state of the
board.
     */
    private void updateButtonTexts() {
        for (int i = 0; i < board.getBoardSize(); ++i) {
            for (int j = 0; j < board.getBoardSize(); ++j) {
                buttons[i][j].setText(String.valueOf(board.get(i, j).getNumber()));
            }
        }
    }

    /**
     * Displays a completion message when all clocks are set to 12,
     * and provides options to restart the game or exit.
```

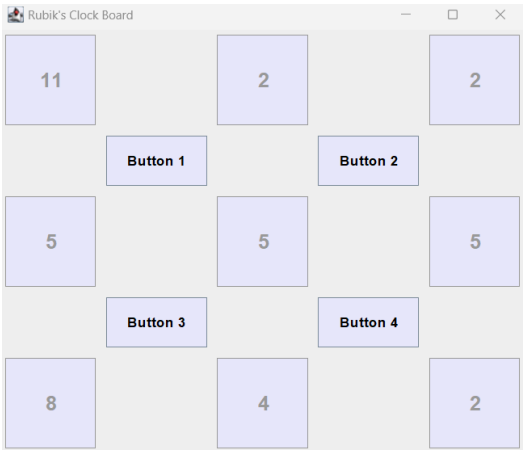
```
     */
    private void showCompletionMessage() {
        Object[] options = {"OK", "Exit"};
        int choice = JOptionPane.showOptionDialog(
                null,
                "All clocks are set to 12! It took " + board.getClickCount() + "
clicks.",
                "Completion",
                JOptionPane.YES_NO_OPTION,
                JOptionPane.INFORMATION_MESSAGE,
                null,
                options,
                options[0]
        );

        if (choice == JOptionPane.YES_OPTION) {
            board.refreshGame();
            updateButtonTexts();
        } else {
            System.exit(0);
        }
    }

    /**
     * Returns the main panel containing the board layout.
     *
     * @return The JPanel representing the board's graphical layout.
     */
    public JPanel getBoardPanel() {
        return boardPanel;
    }
}
```

TEST REPORT

4.3. Black Box (Running Application)

4.3.1. Initializing the Game

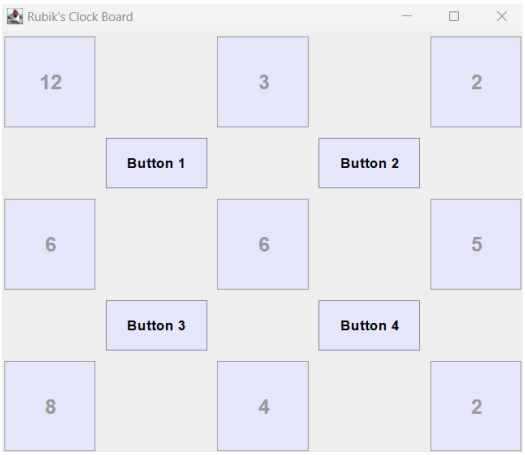
In Picture 2 the start of the game is displayed. On screen, user sees 9 clocks in 3x3 grid and in the middle of every four clocks there are four buttons.



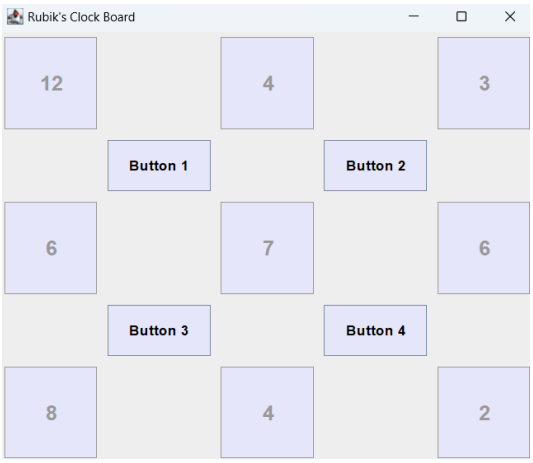
Picture 2. Starting Point of the Game

4.3.2. Purpose of Buttons

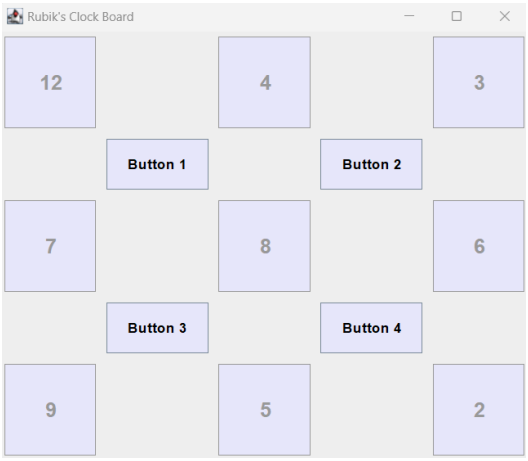
In Picture 3, Picture 4, Picture 5, Picture 6 the incrementing of clocks is displayed. Every button increments four clocks around it by one.



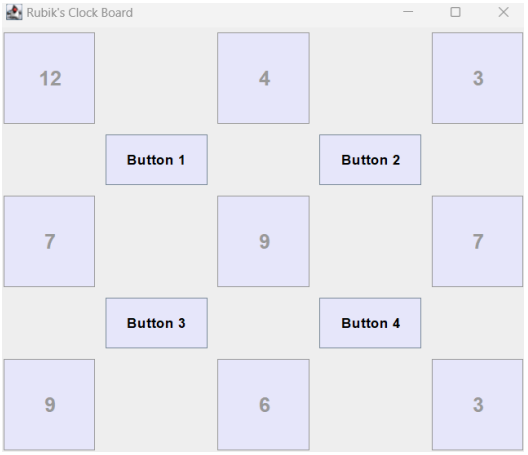
Picture 3. Button 1 (1).



Picture 4. Button 2.



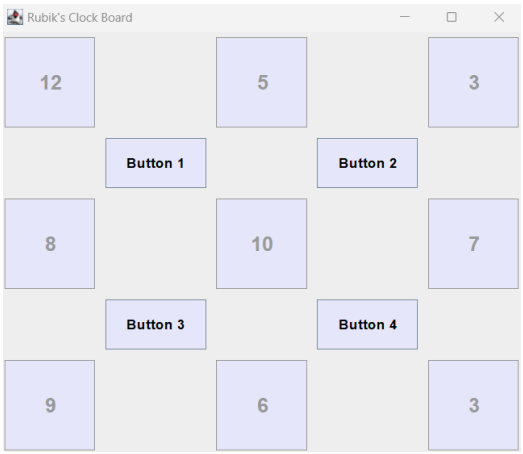
Picture 5. Button 3.



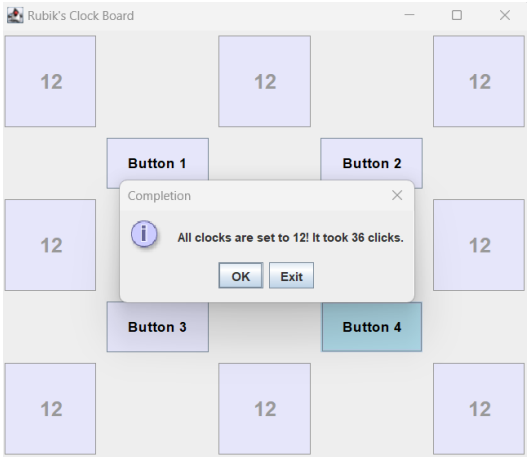
Picture 6. Button 4.

4.3.3. Winning the Game

In Picture 7 user view after pressing button 1 again is displayed. User sees that when the clock is set on 12 it stops incrementing. In Picture 8 user view is displayed when the user have won: it happens when all clocks are set to 12. User sees a pop-up window from which they can decide what to do next.



Picture 7. Button 1 (2).



Picture 8. Click Count and Menu Window.

4.3.4. Exiting the Game

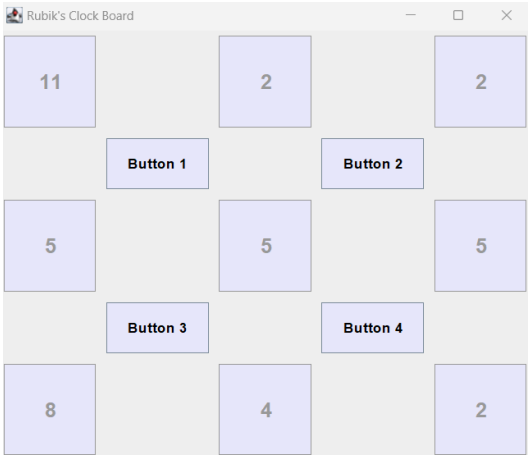
In Picture 9 user view is displayed if the user decides to click “Exit”. User returns to the main class of the program.

```
package gameLogic;

import gameGUI.RubikClockGUI;

public class RubikClock {

    public static void main(String[] args) {

        RubikClockGUI gui = new RubikClockGUI();
    }
}
```

Picture 9. Returning to Code (After “Exit”).

4.3.5. Restarting the Game

In Picture 10 user view is displayed if the user decides to click “OK”. User sees the new game that they can play. All clocks are randomized again and click count reset to 0.



Picture 10. New Game (After "OK").

4.4. White Box

```
package gameTests;

import gameLogic.*;
import gameGUI.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.io.IOException;

/**
 * This class provides white-box testing for the game logic in the {@code Board}
class.
 */
class WhiteBox {

    private Board board;
    private final int boardSize = 3;

    /**
     * Sets up a new board instance with a specified size before each test.
     */
    @BeforeEach
    void setUp() {
        board = new Board(boardSize);
    }

    /**
     * Tests the initialization of the board to ensure it has the correct size and all
fields
     * contain values between 1 and 12.
     */
    @Test
    void testBoardInit() {
        assertEquals(boardSize, board.getBoardSize());
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                assertNotNull(board.get(i, j));
                int fieldValue = board.get(i, j).getNumber();
                assertTrue(fieldValue >= 1 && fieldValue <= 12, "Field values should
be between 1 and 12");
            }
        }
    }
```

```
    /**
     * Tests the {@code incrementSurroundingClocks} method to verify that it correctly
increments
     * the values of surrounding fields.
     */
    @Test
    void testIncrementSurroundingClocks() {
        int[][] initialValues = {
                {board.get(0, 0).getNumber(), board.get(0, 1).getNumber()},
                {board.get(1, 0).getNumber(), board.get(1, 1).getNumber()}
        };

        board.incrementSurroundingClocks(0);

        assertEquals((initialValues[0][0] % 12) + 1, board.get(0, 0).getNumber());
        assertEquals((initialValues[0][1] % 12) + 1, board.get(0, 1).getNumber());
        assertEquals((initialValues[1][0] % 12) + 1, board.get(1, 0).getNumber());
        assertEquals((initialValues[1][1] % 12) + 1, board.get(1, 1).getNumber());
    }

    /**
     * Tests the {@code allClocksAreMaxed} method to verify that it returns false when
     * not all clocks on the board are set to 12.
     */
    @Test
    void testAllClocksAreMaxedWhenNotAllAreMaxed() {
        board.get(0, 0).setNumber(11);
        assertFalse(board.allClocksAreMaxed(), "Not all clocks should be set to 12");
    }

    /**
     * Tests the {@code allClocksAreMaxed} method to verify that it returns true when
     * all clocks on the board are set to 12.
     */
    @Test
    void testAllClocksAreMaxedWhenAllAreMaxed() {
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                board.get(i, j).setNumber(12);
            }
        }
        assertTrue(board.allClocksAreMaxed(), "All clocks should be set to 12");
    }

    /**
     * Tests the {@code incrementClickCount} method to verify that it correctly
     * increments the click count each time it is called.
     */
    @Test
    void testClickCountIncrementsCorrectly() {
        assertEquals(0, board.getClickCount(), "Initial click count should be zero");

        board.incrementClickCount();
        assertEquals(1, board.getClickCount(), "Click count should be 1 after one
increment");

        board.incrementClickCount();
        assertEquals(2, board.getClickCount(), "Click count should be 2 after two
increments");
    }

    /**
     * Tests the {@code refreshGame} method to verify that it resets the click count
to zero
     * and sets all fields on the board to their initial values.
     */
```

```
    @Test
    void testRefreshGameResetsClocksAndClickCount() {
        board.get(0, 0).setNumber(12);
        board.incrementClickCount();

        board.refreshGame();

        assertEquals(0, board.getClickCount(), "Click count should be reset to 0 after
refresh");

        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                assertEquals(board.get(i, j).getNumber(), board.get(i, j).getNumber(),
"Clock values should be reset to initial state");
            }
        }
    }
}
```

5. USER STORIES

5.1. Core Gameplay

5.1.1. Clock Initialization

As a player, I want each clock to start with a random time between 1 and 12, so that each game begins with a unique setup.

5.1.2. Button Mechanics

As a player, I want to press a button to increase the hour on the four adjacent clocks by one, so I can strategically align all clocks to 12.

5.1.3. Hour Wrapping

As a player, I want clocks to don't want wrap around from 12 back to 1 when incremented, so I can win the game.

5.1.4. Winning Condition

As a player, I want the game to automatically check if all clocks show 12 after each button press, so that I can know if I've won immediately.

5.1.5. Move Counting

As a player, I want each button press to count, so that I can know how many steps it took to solve the puzzle.

5.1.6. Victory Message

As a player, I want a message to appear when I win the game, displaying how many moves I took, so I can celebrate my success and track my progress.

5.2. User Interface

5.2.1. Clock Display

As a player, I want each clock to display the current hour clearly, so that I can easily see the game state and plan my moves.

5.2.2. Button Placement

As a player, I want each button to be clearly placed between four clocks, so that I understand which clocks each button will affect.

5.2.3. Move Counter Display

As a player, I want to see the move counter displayed on the screen after I win, so I can monitor how many moves I have made to get there.

5.2.4. Restart Button

As a player, I want a “Restart” button after I win, so that I can press to reset the clocks to a new random setup to start a new game.

5.3. Game Flow

5.3.1. Automatic Game Reset

As a player, I want the game to automatically reset after I win, so I can play again without navigating menus or pressing extra buttons.

5.3.2. Button Colors

As a player, I want to see which clocks are affected when I press a button, so I can see my impact on the game.

5.3.3. Random Clock Setup

As a player, I want the clocks to be randomized differently each time a new game starts, so each game presents a unique challenge.

5.3.4. One-Step Feedback

As a player, I want visual feedback each time I press a button, so I can see which clocks have changed and understand the impact of each move.

5.3.5. Win Condition Persistence

As a player, I want the game to recognize that I’ve won, so that the game doesn't reset until I start a new game.