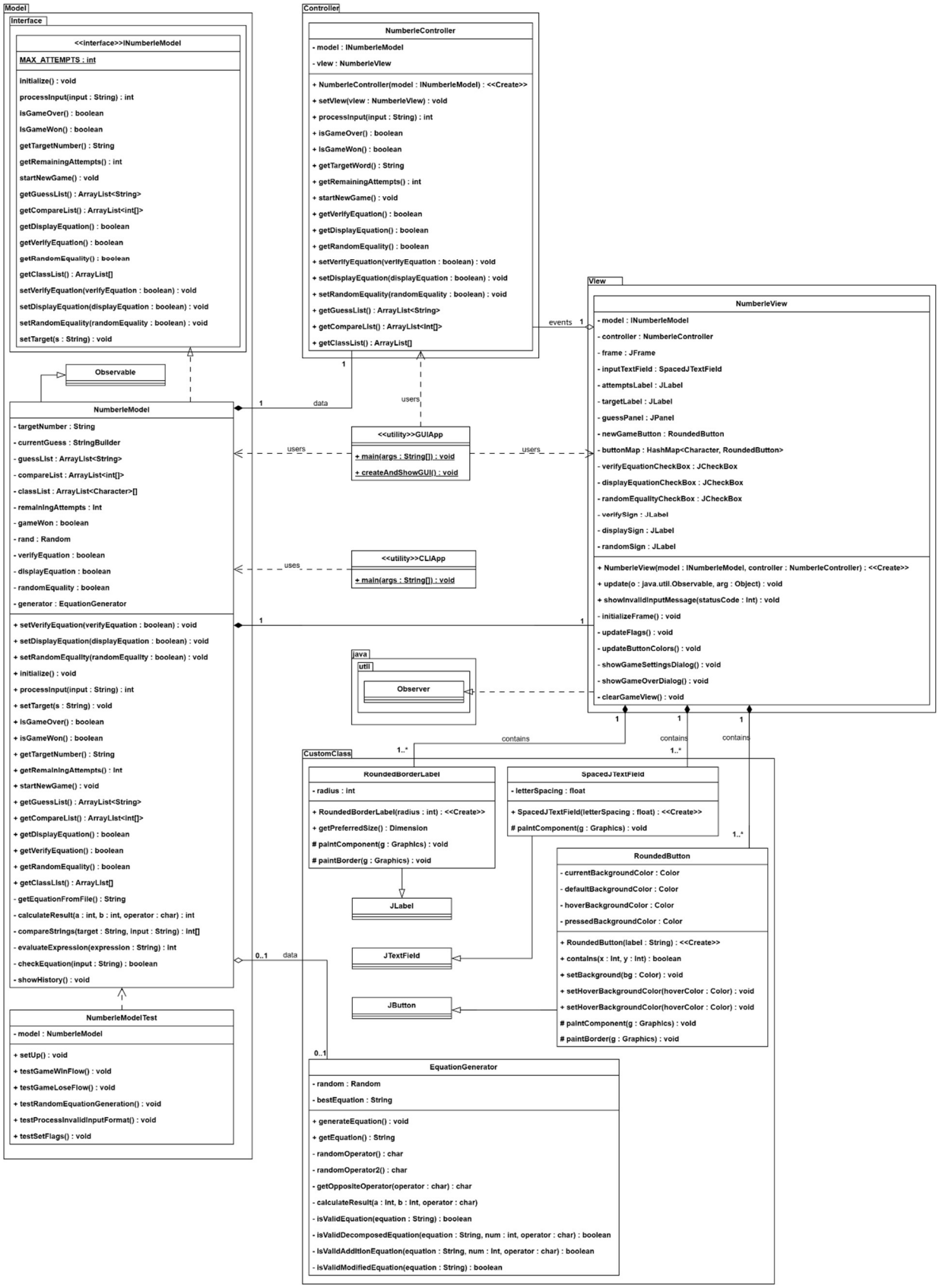
Model
Interface
<<interface>>INumberleModel
MAX_ATTEMPTS : int
initialize() : void
processInput(input : String) : int
isGameOver() : boolean
isGameWon() : boolean
getTargetNumber() : String
getRemainingAttempts() : int
startNewGame() : void
getGuessList() : ArrayList<String>
getCompareList() : ArrayList<int[]>
getDisplayEquation() : boolean
getVerifyEquation() : boolean
getRandomEquality() : boolean
getClassList() : ArrayList[]
setVerifyEquation(verifyEquation : boolean) : void
setDisplayEquation(displayEquation : boolean) : void
setRandomEquality(randomEquality : boolean) : void
setTarget(s : String) : void
Observable
NumberleModel
- targetNumber : String
- currentGuess : StringBuilder
- guessList : ArrayList<String>
- compareList : ArrayList<int[]>
- classList : ArrayList<Character>[]
- remainingAttempts : Int
- gameWon : boolean
- rand : Random
- verifyEquation : boolean
- displayEquation : boolean
- randomEquality : boolean
- generator : EquationGenerator
+ setVerifyEquation(verifyEquation : boolean) : void
+ setDisplayEquation(displayEquation : boolean) : void
+ setRandomEquality(randomEquality : boolean) : void
+ initialize() : void
+ processInput(input : String) : int
+ setTarget(s : String) : void
+ isGameOver() : boolean
+ isGameWon() : boolean
+ getTargetNumber() : String
+ getRemainingAttempts() : Int
+ startNewGame() : void
+ getGuessList() : ArrayList<String>
+ getCompareList() : ArrayList<int[]>
+ getDisplayEquation() : boolean
+ getVerifyEquation() : boolean
+ getRandomEquality() : boolean
+ getClassList() : ArrayList[]
- getEquationFromFile() : String
- calculateResult(a : int, b : int, operator : char) : int
- compareStrings(target : String, input : String) : int[]
- evaluateExpression(expression : String) : int
- checkEquation(input : String) : boolean
- showHistory() : void
NumberleModelTest
- model : NumberleModel
+ setUp() : void
+ testGameWinFlow() : void
+ testGameLoseFlow() : void
+ testRandomEquationGeneration() : void
+ testProcessInvalidInputFormat() : void
+ testSetFlags() : void
Controller
NumberleController
- model : INumberleModel
- view : NumberleView
+ NumberleController(model : INumberleModel) : <<Create>>
+ setView(view : NumberleView) : void
+ processInput(input : String) : int
+ isGameOver() : boolean
+ isGameWon() : boolean
+ getTargetWord() : String
+ getRemainingAttempts() : int
+ startNewGame() : void
+ getVerifyEquation() : boolean
+ getDisplayEquation() : boolean
+ getRandomEquality() : boolean
+ setVerifyEquation(verifyEquation : boolean) : void
+ setDisplayEquation(displayEquation : boolean) : void
+ setRandomEquality(randomEquality : boolean) : void
+ getGuessList() : ArrayList<String>
+ getCompareList() : ArrayList<int[]>
+ getClassList() : ArrayList[]
<<utility>>GUIApp
+ main(args : String[]) : void
+ createAndShowGUI() : void
<<utility>>CLIApp
+ main(args : String[]) : void
java
util
Observer
View
NumberleView
- model : INumberleModel
- controller : NumberleController
- frame : JFrame
- inputTextField : SpacedJTextField
- attemptsLabel : JLabel
- targetLabel : JLabel
- guessPanel : JPanel
- newGameButton : RoundedButton
- buttonMap : HashMap<Character, RoundedButton>
- verifyEquationCheckBox : JCheckBox
- displayEquationCheckBox : JCheckBox
- randomEqualityCheckBox : JCheckBox
- verifySign : JLabel
- displaySign : JLabel
- randomSign : JLabel
+ NumberleView(model : INumberleModel, controller : NumberleController) : <<Create>>
+ update(o : java.util.Observable, arg : Object) : void
+ showInvalidInputMessage(statusCode : Int) : void
- initializeFrame() : void
- updateFlags() : void
- updateButtonColors() : void
- showGameSettingsDialog() : void
- showGameOverDialog() : void
- clearGameView() : void
CustomClass
RoundedBorderLabel
- radius : int
+ RoundedBorderLabel(radius : int) : <<Create>>
+ getPreferredSize() : Dimension
# paintComponent(g : Graphics) : void
# paintBorder(g : Graphics) : void
SpacedJTextField
- letterSpacing : float
+ SpacedJTextField(letterSpacing : float) : <<Create>>
# paintComponent(g : Graphics) : void
RoundedButton
- currentBackgroundColor : Color
- defaultBackgroundColor : Color
- hoverBackgroundColor : Color
- pressedBackgroundColor : Color
+ RoundedButton(label : String) : <<Create>>
+ contains(x : Int, y : Int) : boolean
+ setBackground(bg : Color) : void
+ setHoverBackgroundColor(hoverColor : Color) : void
+ setHoverBackgroundColor(hoverColor : Color) : void
# paintComponent(g : Graphics) : void
# paintBorder(g : Graphics) : void
JLabel
JTextField
JButton
EquationGenerator
- random : Random
- bestEquation : String
+ generateEquation() : void
+ getEquation() : String
- randomOperator() : char
- randomOperator2() : char
- getOppositeOperator(operator : char) : char
- calculateResult(a : Int, b : Int, operator : char)
- isValidEquation(equation : String) : boolean
- isValidDecomposedEquation(equation : String, num : int, operator : char) : boolean
- isValidAdditionEquation(equation : String, num : Int, operator : char) : boolean
- isValidModifiedEquation(equation : String) : boolean
data
users
uses
events
contains
1
0..1
1..*

Figure 1 UML

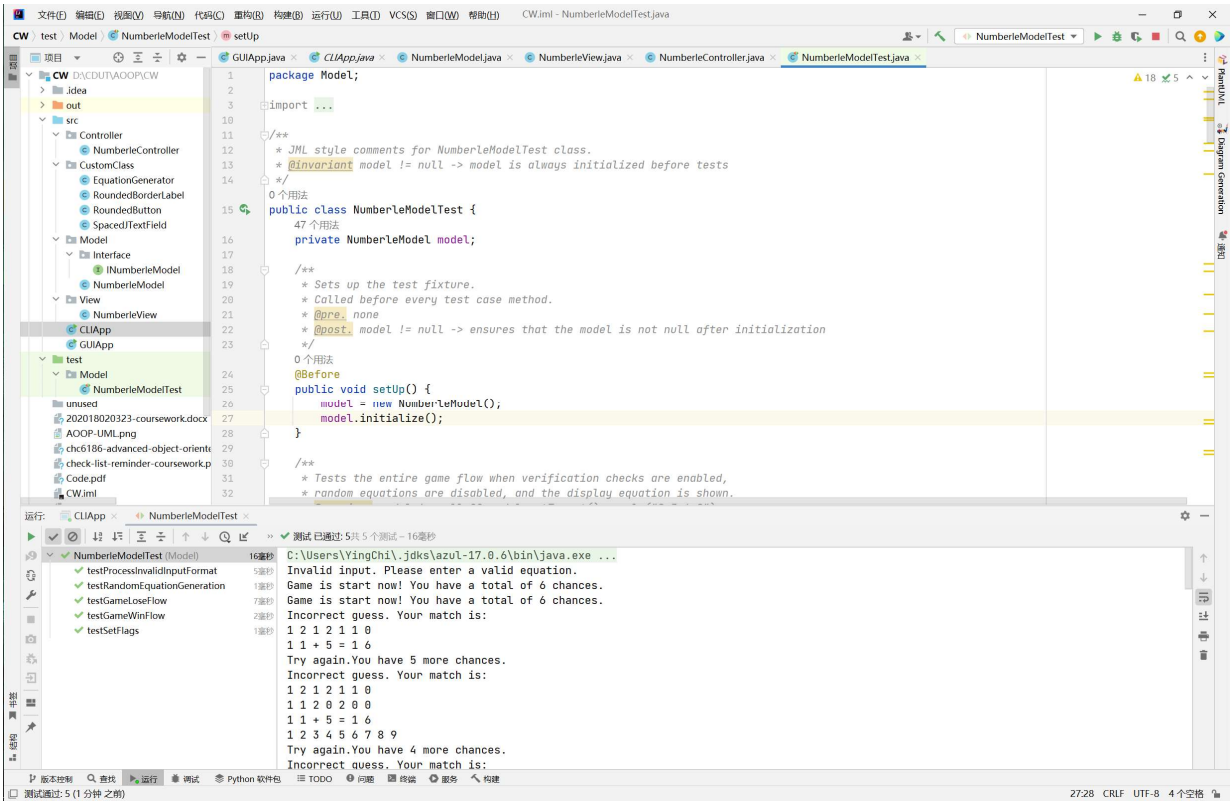

Figure 2 Result for Unit test

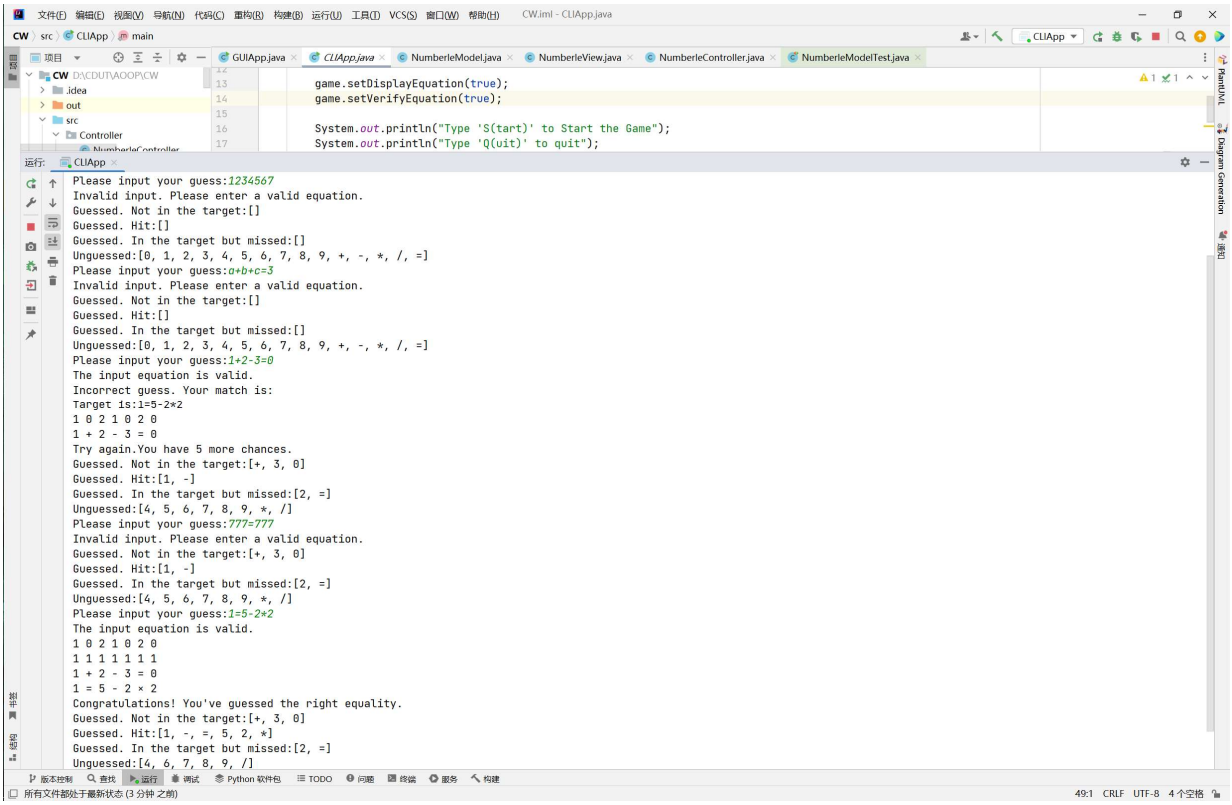

Figure 3 CLIApp program test

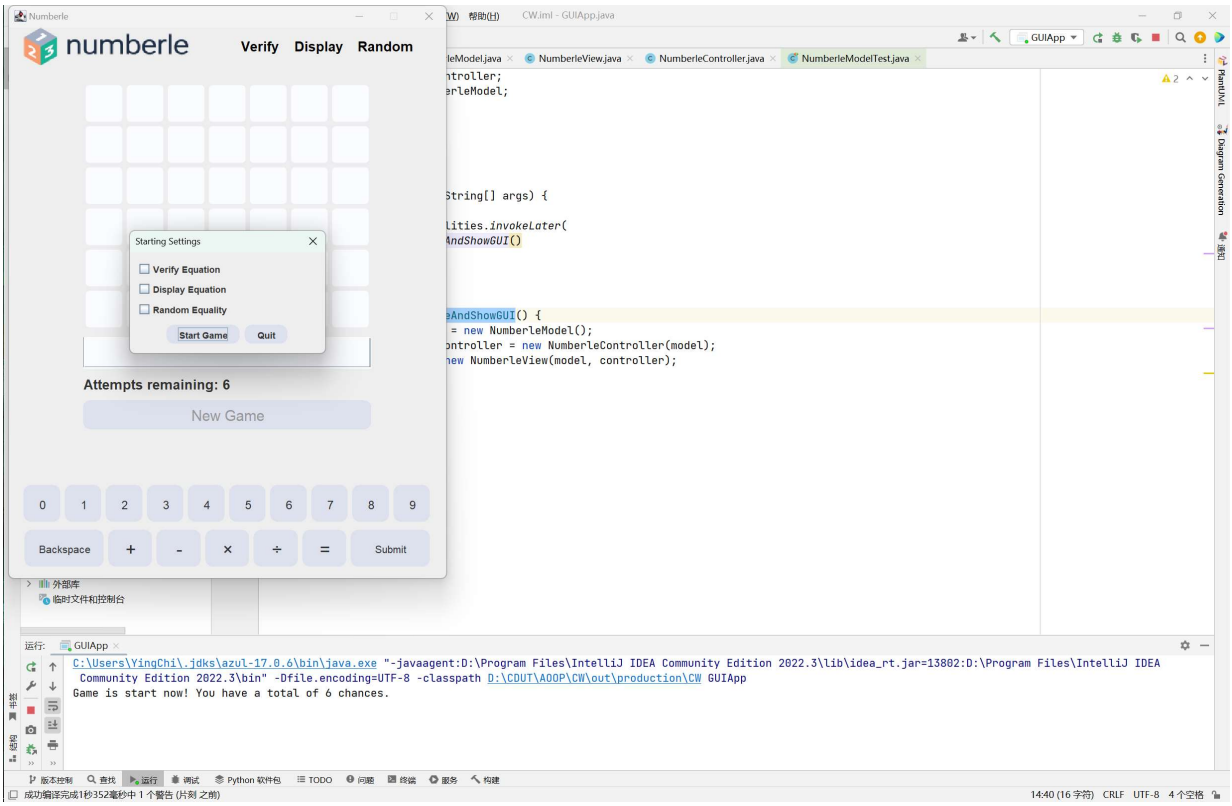

Figure 4 The start of GUIApp

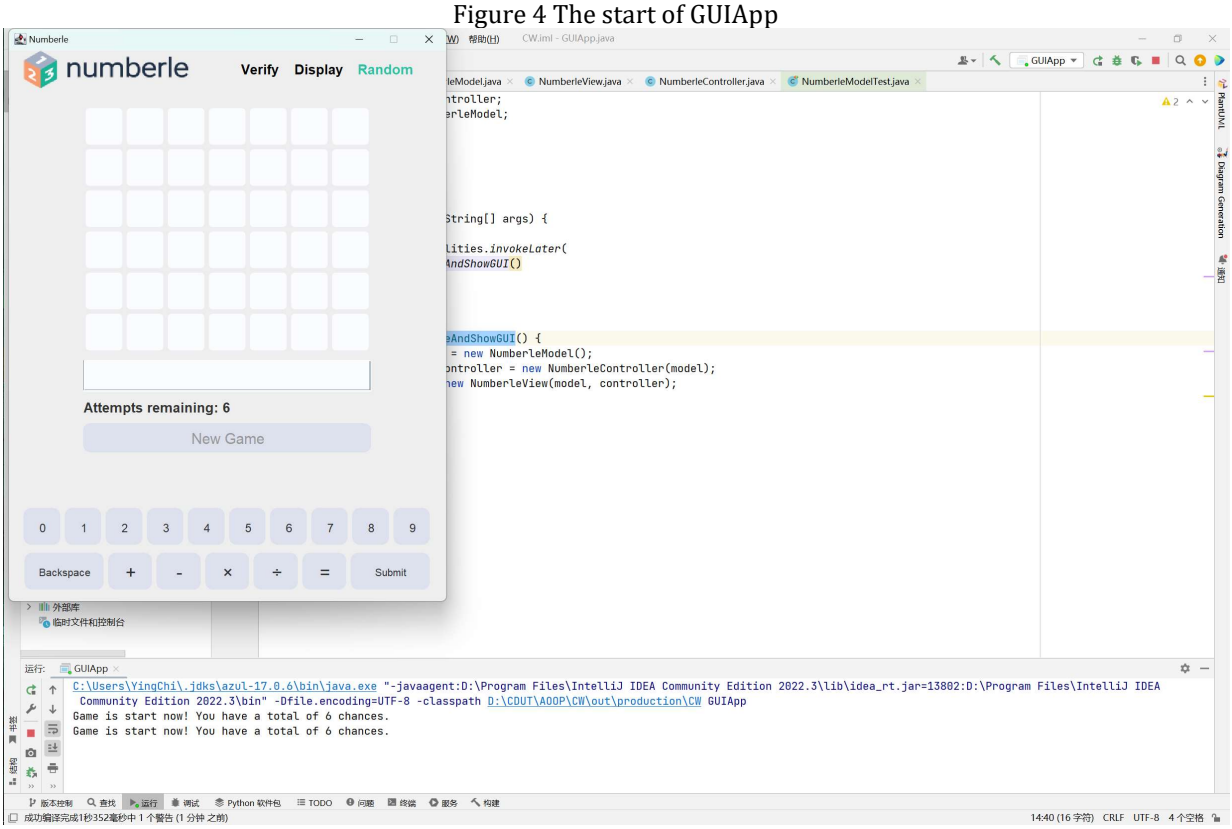

Figure 5 Start the game with unverified equations and random equations.

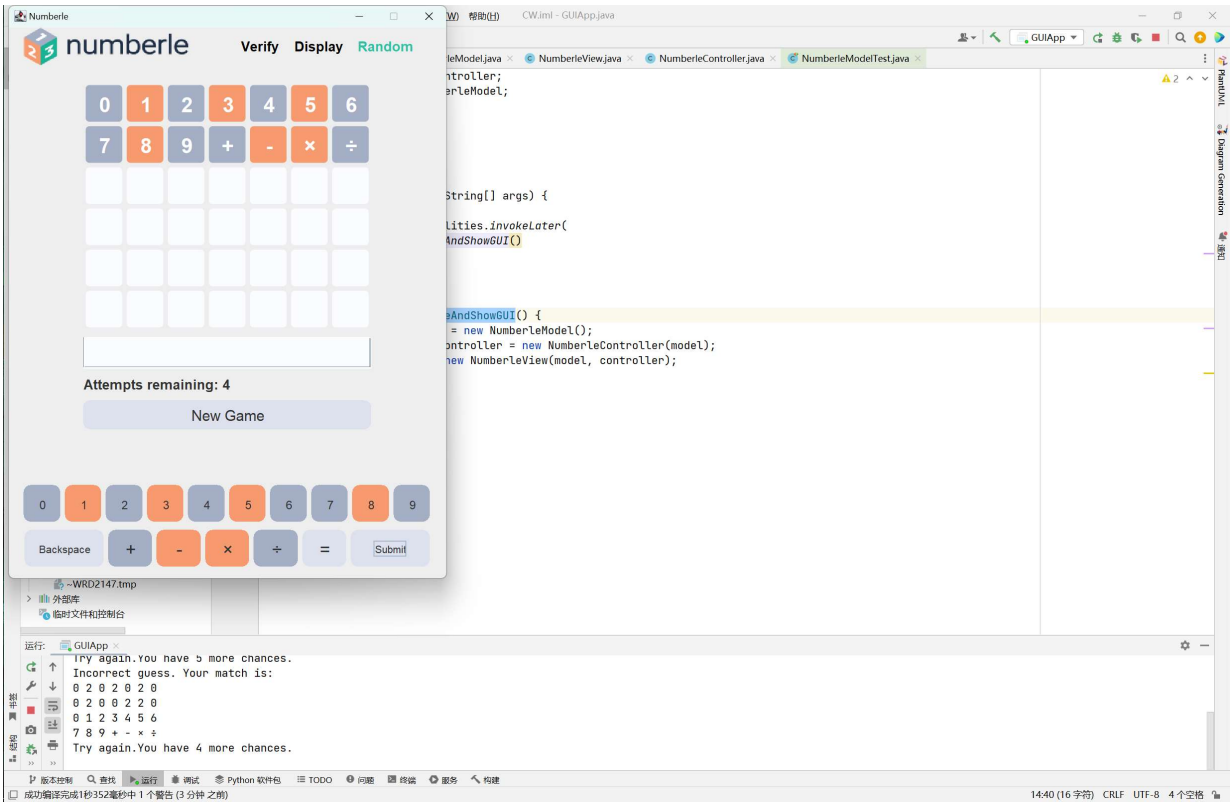

Figure 6 Enter all characters except the equal sign

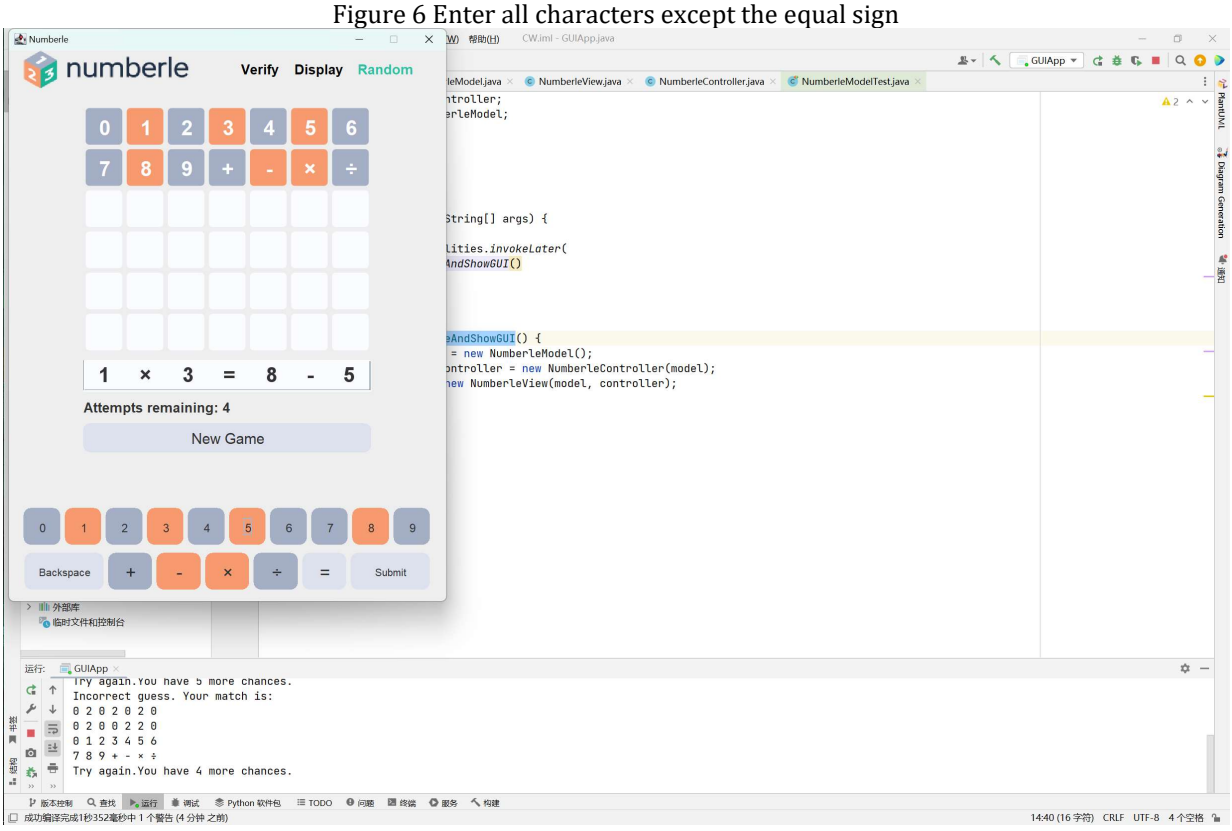

Figure 7 Make the first official guess

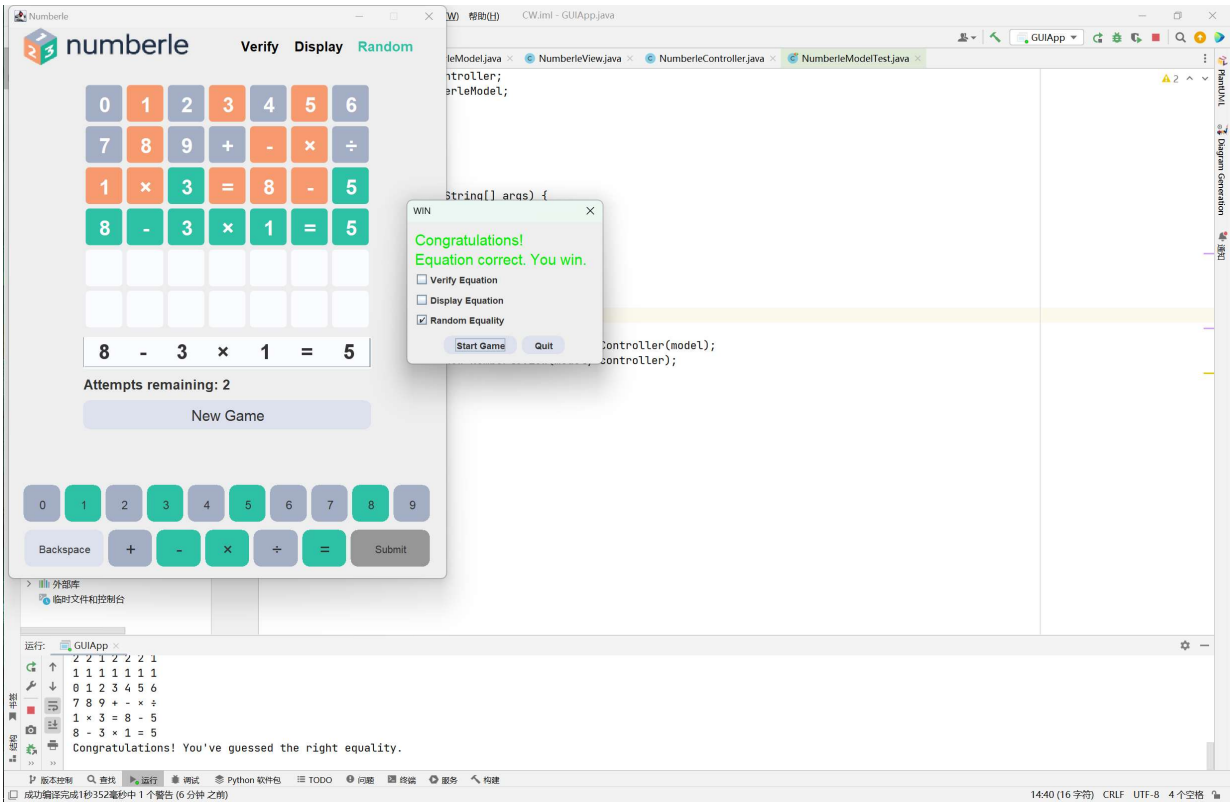

Figure 8 Guess the equation correctly, pop-up prompts, and can set the next game Settings.

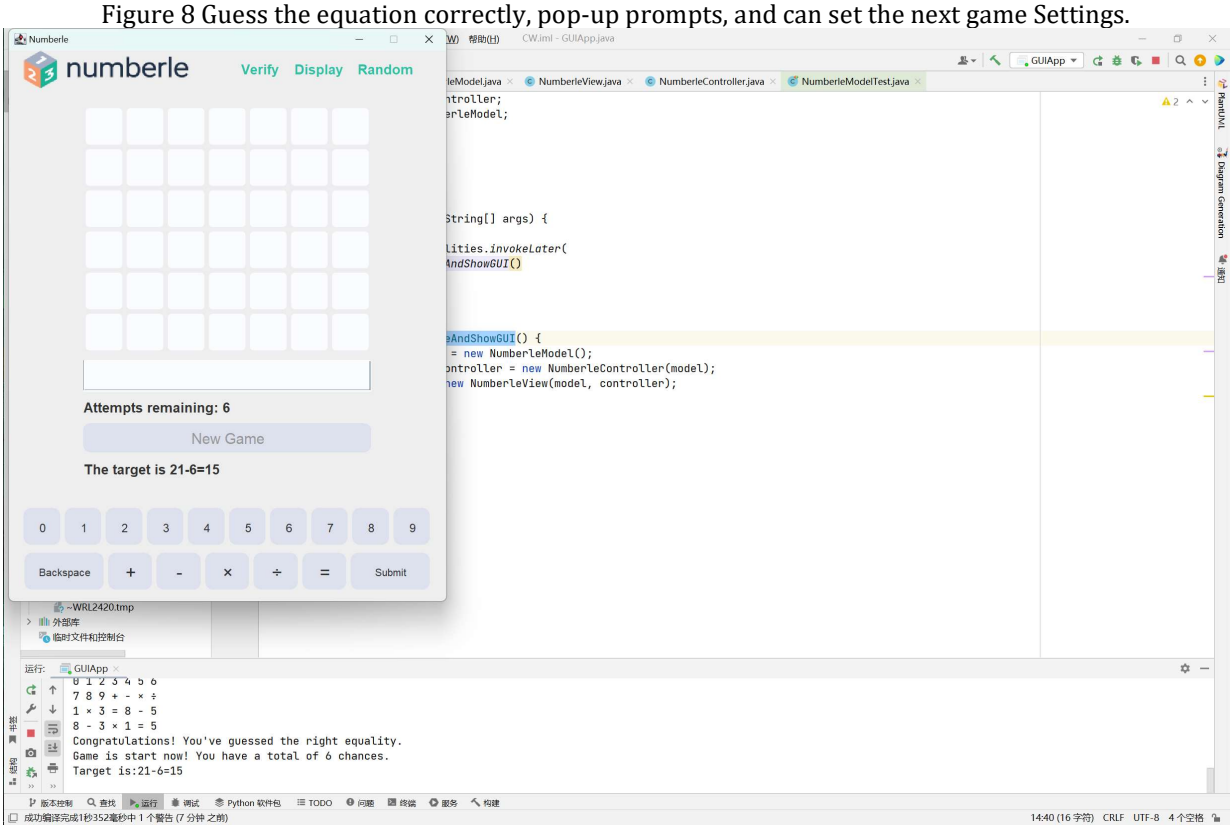

Figure 9 Turn all flags on and restart the game. The equations are displayed and randomly selected.

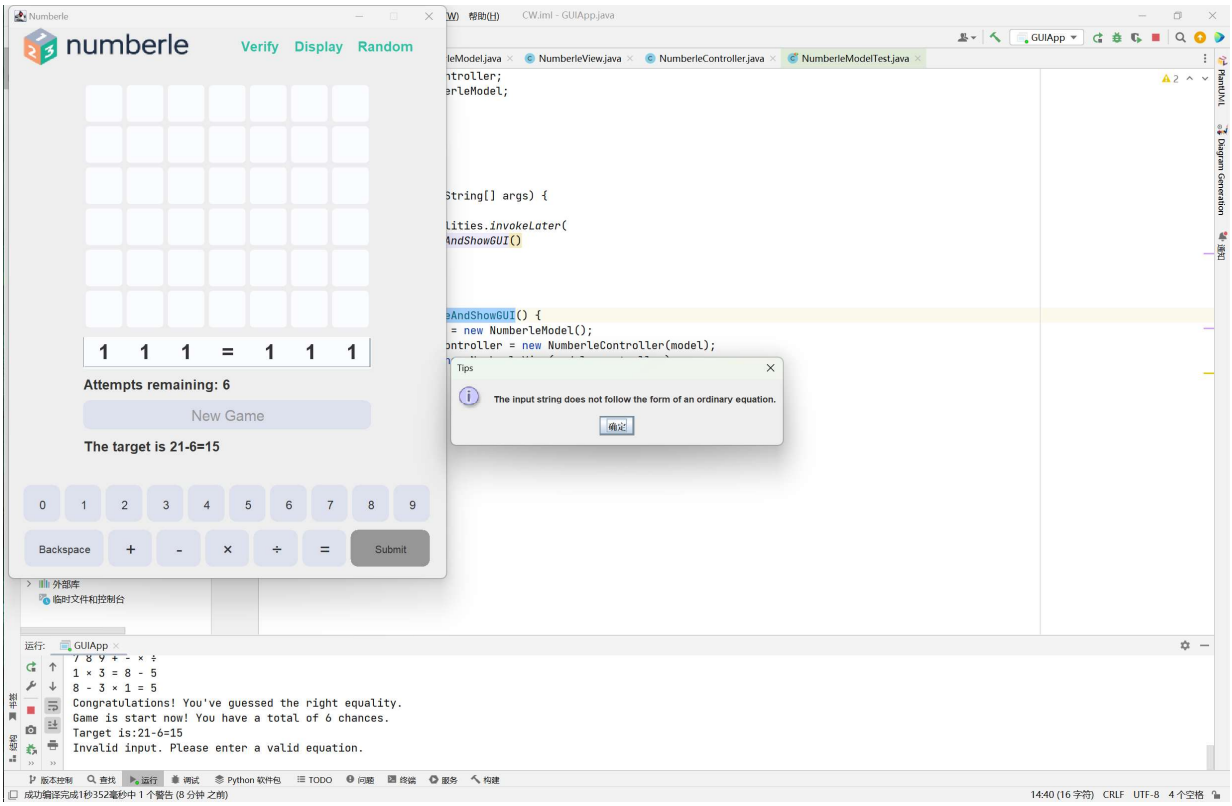

Figure 10 Error input test after equality validation is turned on

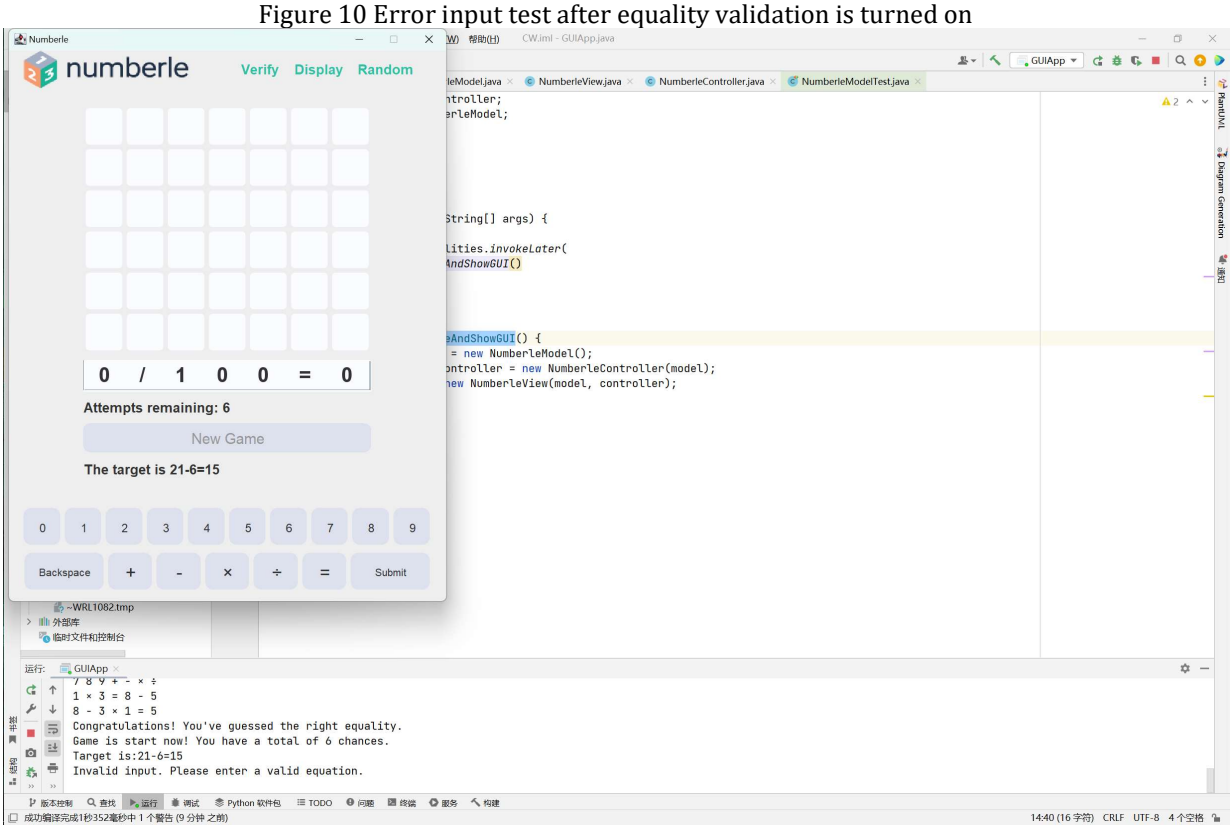

Figure 11 The illegal equation did not reduce the chance of guessing. Take a three-digit guess.

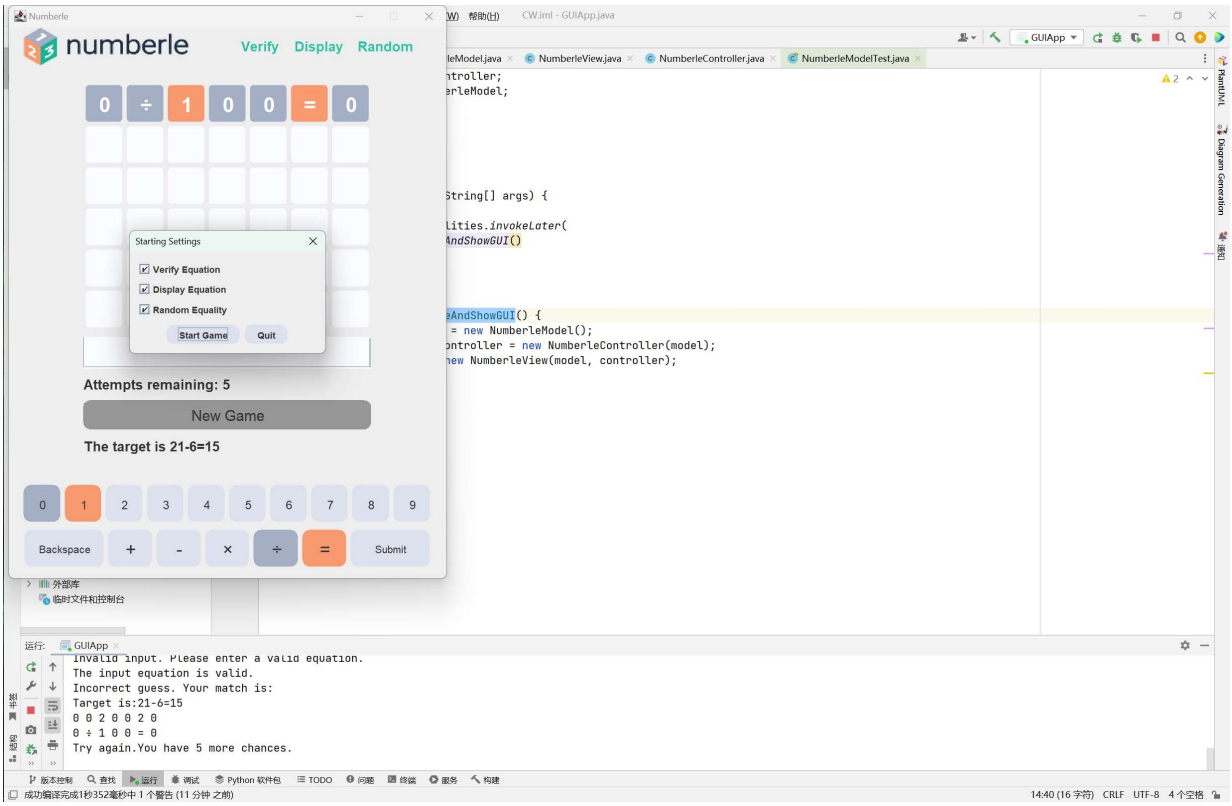

Figure 12 A three-digit guess can be made, and the restart button can be clicked if the guess is legitimate.

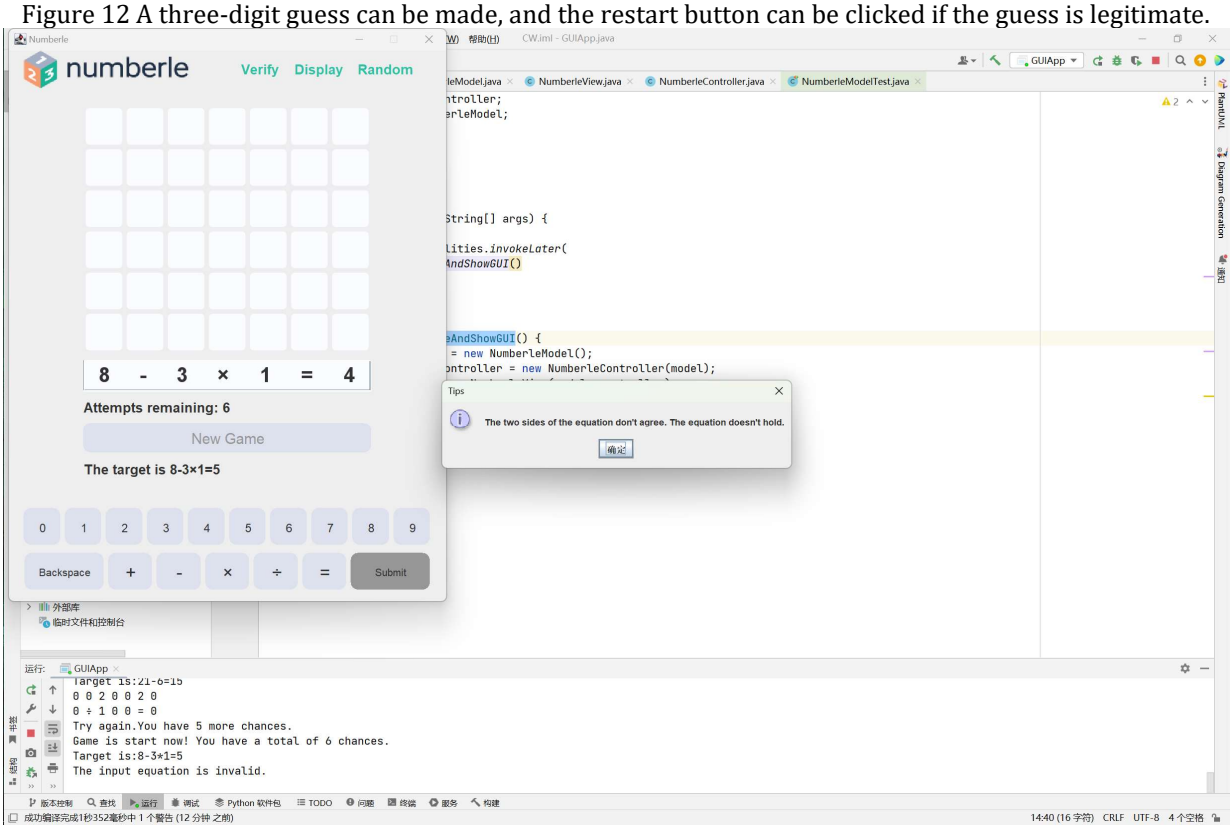

Figure 13 After starting a new game, the equation is randomized. Enter an equation with different results on both sides of the equation.

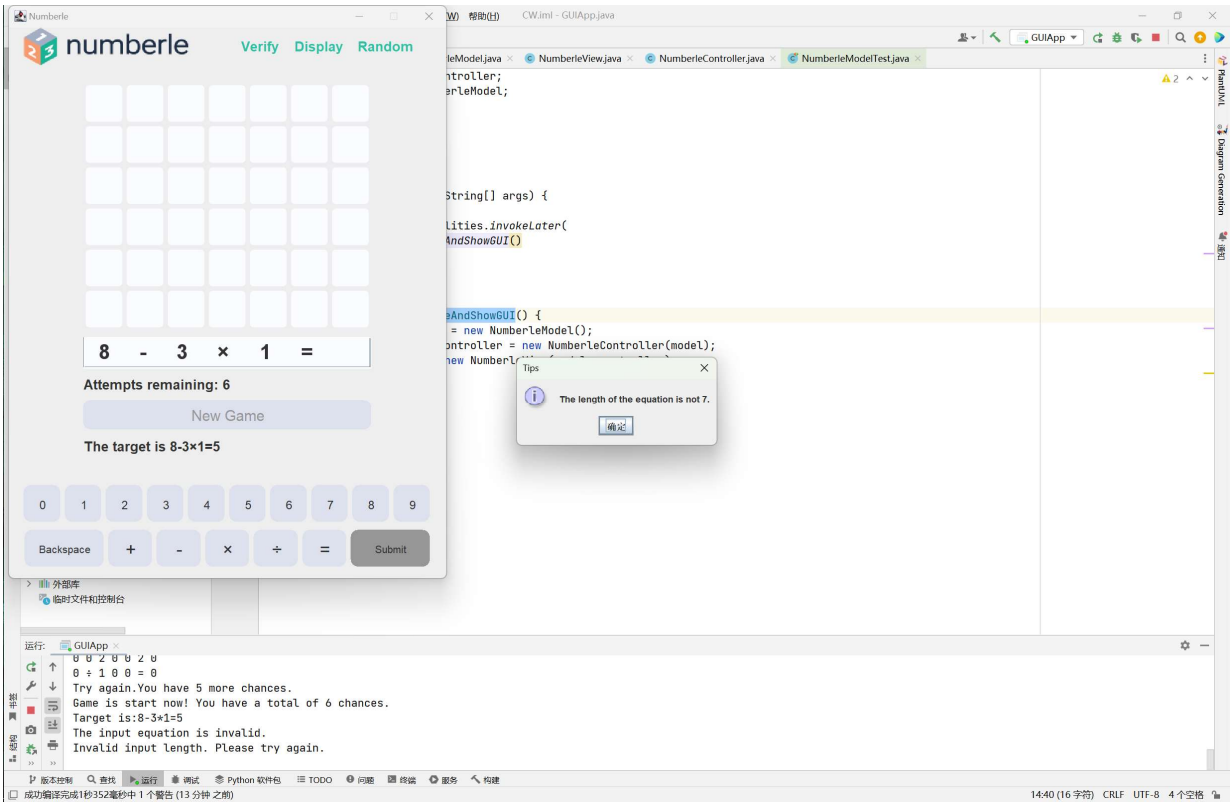

Figure 14 Enter a character string with a length of not 7

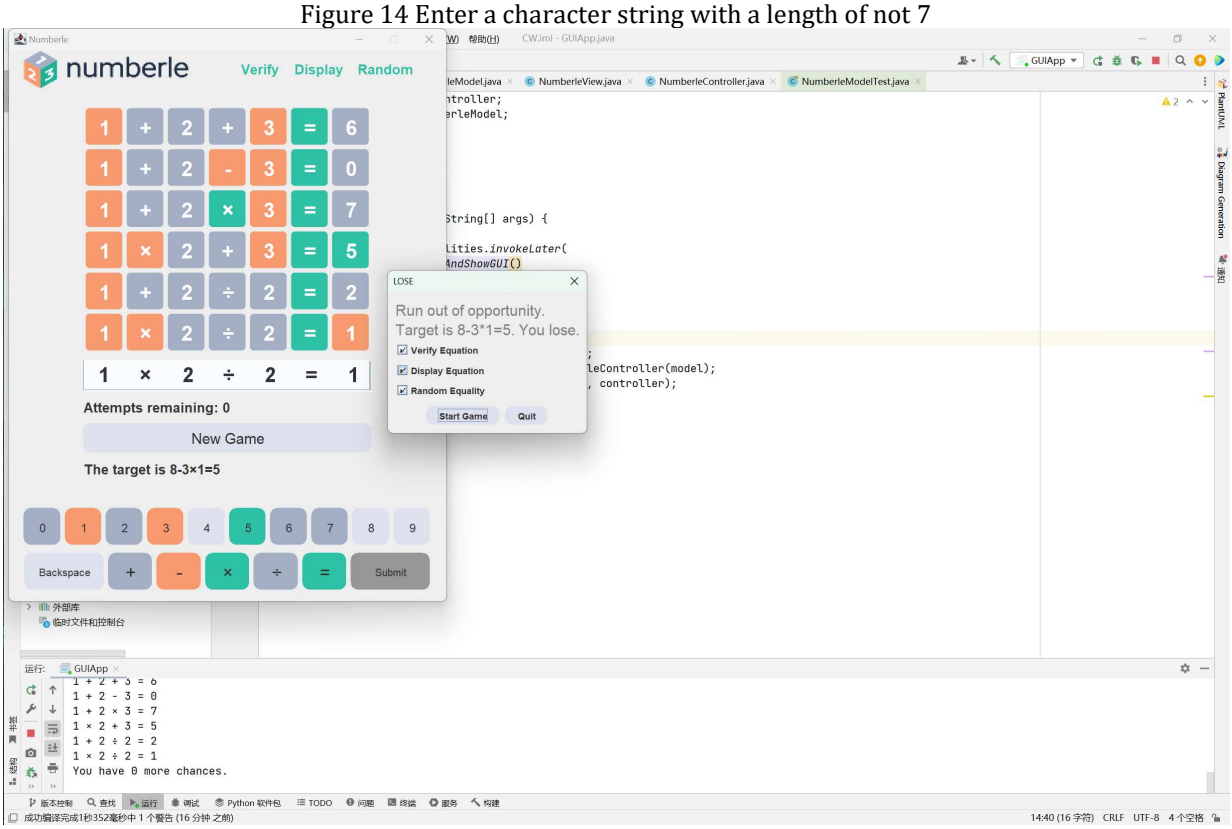

Figure 15 Enter the legal equation 6 times until the game is over and the result is lost.

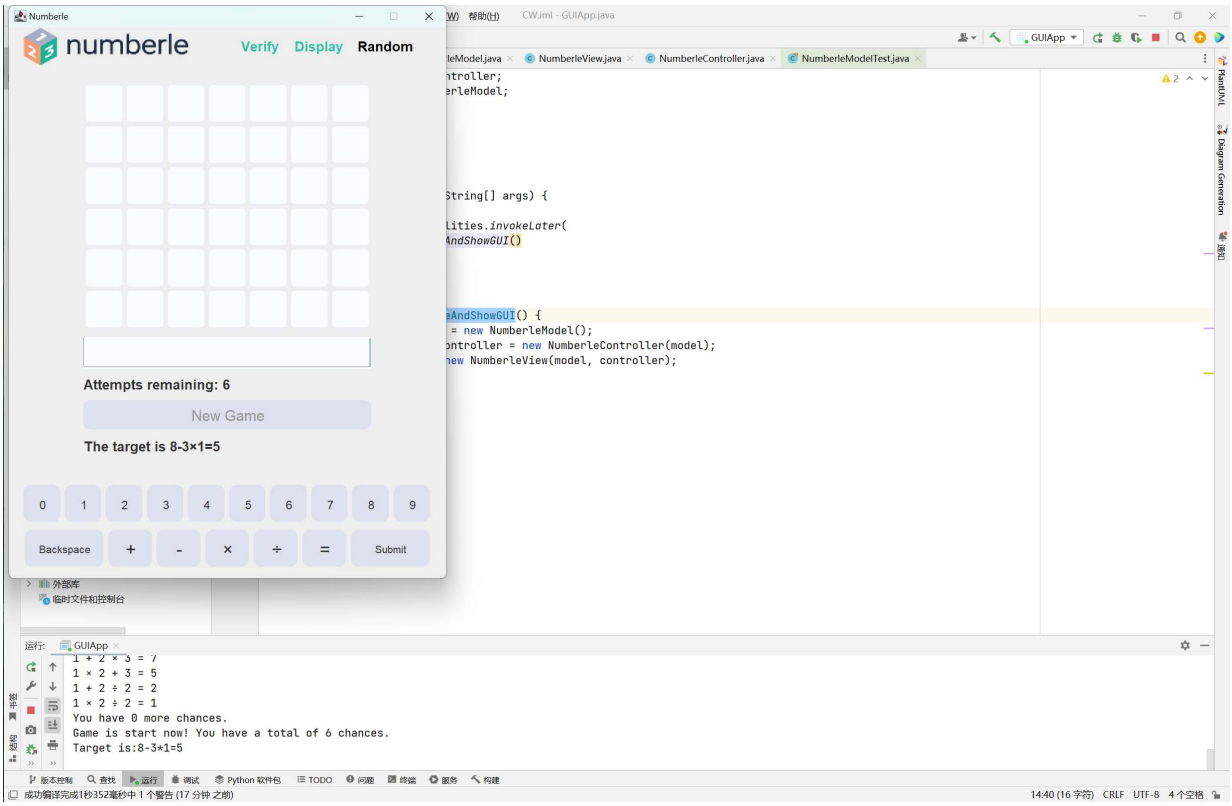

Figure 16 After turning off the random equation flag, the new start game equation does not change.

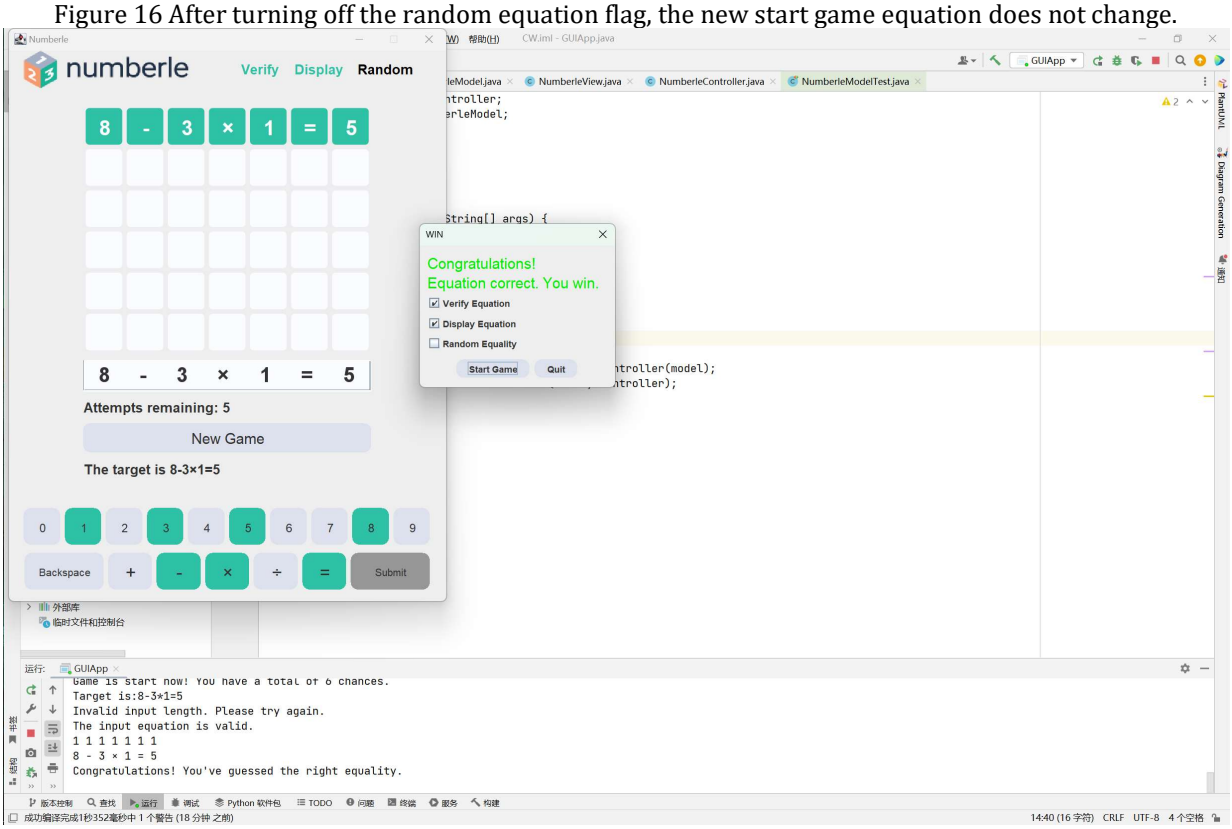

Figure 17 Enter the target equation to verify that it really hasn't changed.

File- D:\CDUT\AOOP\CW\src\CLIApp.java

```java
import Model.NumberleModel;

import java.util.ArrayList;
import java.util.Scanner;

public class CLIApp {
    public static void main(String[] args) {
        NumberleModel game = new NumberleModel();
        Scanner sc=new Scanner(System.in);
        String input;
        ArrayList<Character>[] classList;

        game.setDisplayEquation(true);
        game.setVerifyEquation(true);

        System.out.println("Type 'S(tart)' to Start the Game");
        System.out.println("Type 'Q(uit)' to quit");
        input = sc.nextLine();
        while (!(input.equals("quit") || input.equals("Quit") || input.equals("q") || input.equals("Q"))){
            if (input.equals("start") || input.equals("Start") || input.equals("s") || input.equals("S")) {
                game.startNewGame();
                //game.setTarget("7=4*2-1");
                while (!game.isGameOver()) {
                    System.out.print("Please input your guess:");
                    input = sc.nextLine();
                    //System.out.println(input);
                    if ((input.equals("new") || input.equals("New") || input.equals("n") || input.equals("N"))) {
                        game.startNewGame();
                        System.out.print("Please input your guess:");
                        input = sc.nextLine();
                    }
                    if ((input.equals("quit") || input.equals("Quit") || input.equals("q") || input.equals("Q")))
                        break;
                    game.processInput(input);
                    classList = game.getClassList();
System.out.println("Guessed. Not in the target:" + classList[0]. toString());
                    System.out.println("Guessed. Hit:" + classList[1].toString());
                    System.out.println("Guessed. In the target but missed:" + classList[2].toString());
                System.out.println("Unguessed:" + classList[3].toString()); 40}
                if (!game.isGameWon())
                    System.out.println("You LOSE.");
            } else {
            System.out.println("Unknown command! Please check your input"); 45}
            System.out.println("Type 'S(tart)' to Start the Game");
            System.out.println("Type 'Q(uit)' to quit");
        input = sc.nextLine(); 49}
        System.out.println("You quit the game.");
    }
}

```

File- D:\CDUT\AOOP\CW\src\GUIApp.java

```java
import Controller.NumberleController;
import Model.Interface.INumberleModel;
import Model.NumberleModel;
import View.NumberleView;

public class GUIApp {
    public static void main(String[] args) {

        javax.swing.SwingUtilities.invokeLater(
                () -> createAndShowGUI()
        );
    }

    public static void createAndShowGUI()
        { INumberleModel model = new NumberleModel();
        NumberleController controller = new NumberleController(model);
        NumberleView view = new NumberleView(model, controller);
    }
}
```

File- D:\CDUT\AOOP\CW\src\View\NumberleView.java

```java
package View;// View.NumberleView.java
import Controller.NumberleController;
import CustomClass.RoundedButton;
import CustomClass.SpacedJTextField;
import Model.Interface.INumberleModel;
import Model.NumberleModel;

import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Observer;

/**
 * The View.NumberleView class represents the view component in the MVC pattern.
 * It is responsible for displaying the user interface of the Numberle game,
 * including input fields, buttons, and labels to interact with the game.
 */
public class NumberleView implements Observer {
    // Reference to the model component of the MVC pattern.
    private final INumberleModel model;
    // Reference to the controller component of the MVC pattern.
    private final NumberleController controller;
    // Main window frame of the Numberle game.
    private final JFrame frame = new JFrame("Numberle");
    // Text field for user input.
    private  final SpacedJTextField inputTextField = new SpacedJTextField(40);
    //private final JTextField inputTextField = new JTextField(8);
    // Label to display the remaining attempts.
    private final JLabel attemptsLabel = new JLabel("Attempts remaining: ");
    // Label to display the target equation or number.
    private final JLabel targetLabel = new JLabel("The target is ");
    // Panel to display the guesses.
    private JPanel guessPanel;
    // Button to start a new game.
    private RoundedButton newGameButton;
    // Map to store the association between characters and buttons.
    private HashMap<Character, RoundedButton> buttonMap;
    // Checkboxes to toggle game settings.
    private JCheckBox verifyEquationCheckBox;
    private JCheckBox displayEquationCheckBox;
    private JCheckBox randomEqualityCheckBox;
    // Labels for the checkboxes.
    private final JLabel verifySign = new JLabel("Verify");
    private final JLabel displaySign = new JLabel("Display");
    private final JLabel randomSign = new JLabel("Random");

    /**
     * Constructor for View.NumberleView. Initializes the view by setting up the model and controller,
     * starting a new game, and updating the view to reflect the current state of the model.
     *
     * @param model      The model component of the MVC pattern.
     * @param controller The controller component of the MVC pattern.
     */
    public NumberleView(INumberleModel model, NumberleController controller) {
        this.controller = controller;
        this.model = model;
        this.controller.startNewGame();
        ((NumberleModel)this.model).addObserver(this);
        this.controller.setView(this);
        initializeFrame();
```

```
66          update((NumberleModel)this.model, null);
67          showGameSettingsDialog();
68      }
69
70      /**
71       * Initializes the main frame of the application, setting up the layout, panels, and components.
72       * Precondition: None
73       * Postcondition: The main frame and its components are initialized and displayed to the user.
74       */
75      private void initializeFrame() {
76          // Assert that the controller and model are properly initialized before setting up the frame.
77          assert controller != null : "Controller must not be null.";
78          assert model != null : "Model must not be null.";
79
80          // Map to store the association between characters and buttons.
81          buttonMap = new HashMap<>();
82
83          // Set up the main frame properties.
84          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
85          frame.setSize(600, 775);
86          frame.setResizable(false);
87          frame.setLayout(new BorderLayout());
88
89          // Center panel for the main content.
90          JPanel center = new JPanel();
91          center.setLayout(new BoxLayout(center, BoxLayout.Y_AXIS));
92          center.add(new JPanel());
93
94          // Header panel with the game logo.
95          JPanel head = new JPanel();
96          head.setLayout(new BoxLayout(head, BoxLayout.X_AXIS));
97          JLabel  headText = new JLabel ();
98          BufferedImage originalImage;
99          try {
100             // Load and scale the game logo.
101             originalImage = ImageIO.read(new File("logo.png"));
102         } catch (IOException e) {
103             throw new RuntimeException(e);
104         }
105         double scale = 0.15;
106         Image resizedImage = originalImage.getScaledInstance((int)(originalImage.getWidth() * scale), (int)(originalImage.getHeight() * scale), Image.SCALE_SMOOTH);
107         BufferedImage bufferedResizedImage = new BufferedImage((int)(originalImage.getWidth() * scale), (int)(originalImage.getHeight() * scale), BufferedImage.TYPE_INT_ARGB);
108         Graphics2D g2d = bufferedResizedImage.createGraphics();
109         g2d.drawImage(resizedImage, 0, 0, null);
110         g2d.dispose();
111
112         // Assert that the frame and its components are properly initialized.
113         assert frame != null : "Frame must not be null.";
114         assert center != null : "Center panel must not be null.";
115         assert head != null : "Head panel must not be null.";
116         assert headText != null : "Head text label must not be null.";
117         assert originalImage != null : "Original image must not be null.";
118         assert resizedImage != null : "Resized image must not be null.";
119         assert bufferedResizedImage != null : "Buffered resized image must not be null.";
120         assert g2d != null : "Graphics2D object must not be null.";
121
122         //Sets the icon for the header text label and defines its border.
123         headText.setIcon(new ImageIcon(bufferedResizedImage));
124         headText.setBorder(BorderFactory.createEmptyBorder(0, 20, 0, 20));
125         head.add(headText);
```

```
126126
127         //Sets the font and border for the verification, display, and random labels.
128         verifySign.setFont(new Font("Montserrat", Font.BOLD, 18)); // Set the text size
    to 18 point font
129         verifySign.setBorder(BorderFactory.createEmptyBorder(5, 50, 5, 10));
130
131         displaySign.setFont(new Font("Montserrat", Font.BOLD, 18)); // Set the text size
     to 18 point font
132         displaySign.setBorder(BorderFactory.createEmptyBorder(5, 10, 5, 10));
133
134         randomSign.setFont(new Font("Montserrat", Font.BOLD, 18)); // Set the text size
    to 18 point font
135         randomSign.setBorder(BorderFactory.createEmptyBorder(5, 10, 5, 10));
136
137         // Calls a method to update the flags based on the current game settings.
138         updateFlags();
139
140         // Adds the labels to the head panel and the head panel to the frame.
141         head.add(verifySign);
142         head.add(displaySign);
143         head.add(randomSign);
144         frame.add(head, BorderLayout.NORTH);
145
146         // Initializes the guess panel which displays the history of user inputs.
147         guessPanel = new JPanel();
148         guessPanel.setLayout(new GridLayout(6, 7, 5, 5)); // 6 rows, 7 columns, 5px
    spacing
149         //guessPanel.setPreferredSize(new Dimension(380, 325));
150         int leftAndRight = 100;
151         guessPanel.setBorder(BorderFactory.createEmptyBorder(10, leftAndRight, 10,
    leftAndRight));
152         for (int i = 0; i < 6; i++) {
153             for (int j = 0; j < 7; j++) {
154                 // 创建自定义JLabel
155                 CustomClass.RoundedBorderLabel label = new CustomClass.
    RoundedBorderLabel(10); // Set the radius of the rounded corners to 10
156                 label.setHorizontalAlignment(SwingConstants.CENTER);
157                 label.setFont(new Font("Montserrat", Font.BOLD, 28)); // Set the text
    size to 28 point font
158                 label.setBackground(new Color(251,252,255));
159                 label.setForeground(Color.WHITE); // Set the text color to black
160                 label.setPreferredSize(new Dimension(50, 50));
161                 guessPanel.add(label);
162             }
163         }
164         center.add(guessPanel); // Adds the guessPanel to the center panel.
165
166         assert frame != null : "Frame must not be null.";
167         assert headText != null : "Head text label must not be null.";
168         assert bufferedResizedImage != null : "Buffered resized image must not be null."
    ;
169         assert verifySign != null : "Verify sign label must not be null.";
170         assert displaySign != null : "Display sign label must not be null.";
171         assert randomSign != null : "Random sign label must not be null.";
172         assert guessPanel != null : "Guess panel must not be null.";
173         assert inputTextField != null : "Input text field must not be null.";
174
175         // Initializes the input panel which contains the text field for user input.
176         JPanel inputPanel = new JPanel();
177         inputPanel.setLayout(new GridLayout(4, 1));
178         inputPanel.setBorder(BorderFactory.createEmptyBorder(0, leftAndRight, 0,
    leftAndRight));
179         inputTextField.setFont(new Font("Montserrat", Font.BOLD, 28));
180         inputTextField.setHorizontalAlignment(JTextField.CENTER);
181         inputTextField.setBackground(new Color(251,252,255));
182         inputPanel.add(inputTextField); // Adds the inputTextField to the inputPanel.
183
```

```
184        // Assert that the inputPanel is not null after initialization.
185        assert inputPanel != null : "Input panel must not be null.";
186
187        // Set the text for attemptsLabel and add it to the inputPanel.
188        attemptsLabel.setText("Attempts remaining: " + controller.getRemainingAttempts
   ());
189        attemptsLabel.setFont(new Font("Montserrat", Font.BOLD, 18)); // Set the text
   size to 18 point font
190        inputPanel.add(attemptsLabel);
191        // Assert that attemptsLabel is added to the inputPanel.
192 assert inputPanel.getComponent(inputPanel.getComponentCount() -
   1).equals( attemptsLabel) : "attemptsLabel must be added to the inputPanel.";
193
194        // Initialize newGameButton, set its properties, and add it to the inputPanel.
195        newGameButton = new RoundedButton("New Game");
196        //newGameButton = new JButton("New Game");
197        newGameButton.setEnabled(false); // 初始禁用
198        newGameButton.setFont(new Font("Montserrat", Font.PLAIN, 20)); // Set the text
   size to 20 point font
199        newGameButton.addActionListener(e -> showGameSettingsDialog());
200        inputPanel.add(newGameButton);
201        // Assert that newGameButton is added to the inputPanel.
202 assert inputPanel.getComponent(inputPanel.getComponentCount() -
   1).equals( newGameButton) : "newGameButton must be added to the inputPanel.";
203
204        // Set the text for targetLabel and add it to the inputPanel.
205        targetLabel.setText("The target is " + controller.getTargetWord().replaceAll("/"
   ,"÷").replaceAll("\\*","×"));
206        targetLabel.setFont(new Font("Montserrat", Font.BOLD, 18)); // Set the text size
    to 20 point font
207
208        inputPanel.add(targetLabel);
209
210        // Assert that targetLabel is added to the inputPanel.
211 assert inputPanel.getComponent(inputPanel.getComponentCount() -
   1).equals( targetLabel) : "targetLabel must be added to the inputPanel.";
212
213        // Add inputPanel to the center of the frame.
214        center.add(inputPanel);
215        center.add(new JPanel());
216        frame.add(center, BorderLayout.CENTER);
217
218        // Initialize keyboardPanel and add it to the frame.
219        JPanel keyboardPanel = new JPanel();
220        keyboardPanel.setLayout(new BoxLayout(keyboardPanel, BoxLayout.Y_AXIS));
221        keyboardPanel.add(new JPanel());
222        // Assert that keyboardPanel is not null.
223        assert keyboardPanel != null : "Keyboard panel must not be null.";
224
225        // Initialize numberPanel and add number buttons to it.
226        JPanel numberPanel = new JPanel();
227        //numberPanel.setLayout(new GridLayout(1, 10));
228        for (int i = 0; i < 10; i++) {
229            RoundedButton button = new RoundedButton(Integer.toString(i));
230 //             JButton button = new JButton(Integer.toString(i));
2312    1          32
   3 2322       233233
```

```
            button.setEnabled(true);
            button.addActionListener(e -> {
                String text = inputTextField.getText();
                if (text.length() >= controller.getTargetWord().length())
                    inputTextField.setText(text.substring(0, text.length() - 1));
                inputTextField.setText(inputTextField.getText() + button.getText());
            });
            button.setBackground(new Color(220,225,237));
            button.setFont(new Font("Montserrat", Font.PLAIN, 16)); // Set font size to
16.
            button.setPreferredSize(new Dimension(50, 50));
            numberPanel.add(button);
```

```
2422            buttonMap.put((char) ('0' + i), (RoundedButton) numberPanel.getComponent(i
   4));
   2        }

2432
   4
   3
244         // Assert that numberPanel contains 10 buttons.
245         assert numberPanel.getComponentCount() == 10 : "Number panel must contain 10
    buttons.";
246
247         // Initialize operatorPanel and add backspaceButton to it.
248         JPanel operatorPanel = new JPanel();
249         RoundedButton backspaceButton = new RoundedButton("Backspace");
250         backspaceButton.setBackground(new Color(220,225,237));
251         backspaceButton.setFont(new Font("Montserrat", Font.PLAIN, 14)); // Set font
    size to 14.
252         backspaceButton.setPreferredSize(new Dimension(107, 50));
253         backspaceButton.addActionListener(e -> {
254             String text = inputTextField.getText();
255             if (text.length() > 0)
256         inputTextField.setText(text.substring(0, text.length() - 1)); 257
            });
258         operatorPanel.add(backspaceButton);
259         // Assert that operatorPanel contains the backspaceButton.
260         assert operatorPanel.getComponent(operatorPanel.getComponentCount() -
    1).equals( backspaceButton) : "Backspace button must be added to the operator panel.";
261
262         // Assert that the operatorPanel is not null after initialization.
263         assert operatorPanel != null : "Operator panel must not be null.";
264
265         for (String s: new String[]{"+", "-", "×", "÷", "="}){
266             //Initialize and set properties for all the operate buttons, then add it to
    the operatorPanel.
267             RoundedButton button = new RoundedButton(s);
268             button.setBackground(new Color(220, 225, 237));
269             button.setFont(new Font("Montserrat", Font.PLAIN, 24)); // Set the text size
     to 24 point font
270             button.setPreferredSize(new Dimension(60, 50));
271             button.addActionListener(e -> {
272                 String text = inputTextField.getText();
273                 if (text.length() >= controller.getTargetWord().length())
274                     inputTextField.setText(text.substring(0, text.length() - 1));
275             inputTextField.setText(inputTextField.getText() + button.getText()); 276
                });
277             operatorPanel.add(button);
278             // Assert that plusButton is added to the operatorPanel.
279             assert operatorPanel.getComponent(operatorPanel.getComponentCount() - 1).
    equals(button) : "plusButton must be added to the operator panel.";
280
281             // Assert that the buttonMap is not null after initialization.
282             assert buttonMap != null : "Button map must not be null.";
283             // Map the operator buttons to their respective symbols in the buttonMap.
284             buttonMap.put(s.charAt(0), button);
285             if (s.equals("×"))
286                 buttonMap.put('*', button); // For systems that use '*' as the
    multiplication symbol.
287             else if (s.equals("÷"))
288                 buttonMap.put('/', button); // For systems that use '/' as the division
    symbol.
289         }
290
291         // Assert that all the necessary buttons are added to the buttonMap.
292         assert buttonMap.containsKey('+') : "Plus button must be mapped.";
293         assert buttonMap.containsKey('-') : "Minus button must be mapped.";
294         assert buttonMap.containsKey('×') : "Multiply button must be mapped.";
295         assert buttonMap.containsKey('*') : "Asterisk for multiplication must be mapped
    .";
```

```
296         assert buttonMap.containsKey('÷') : "Divide button must be mapped.";
297         assert buttonMap.containsKey('/') : "Slash for division must be mapped.";
298         assert buttonMap.containsKey('=') : "Equal button must be mapped.";
```

```
299
300         RoundedButton submitButton = new RoundedButton("Submit");
301         submitButton.setBackground(new  Color(220,225,237));
302         submitButton.setFont(new Font("Montserrat", Font.PLAIN, 14)); // Set the text
    size to 14 point font
303         submitButton.setPreferredSize(new Dimension(107, 50));
304         submitButton.addActionListener(e -> {
305             controller.processInput(inputTextField.getText());
306             inputTextField.setText("");
307         });
308         operatorPanel.add(submitButton);
309         // Assert that submitButton is added to the operatorPanel.
310         assert operatorPanel.getComponent(operatorPanel.getComponentCount() -
    1).equals( submitButton) : "submitButton must be added to the operator panel.";
311
312         // Assert that keyboardPanel is not null and contains all necessary subpanels.
313         assert keyboardPanel != null : "Keyboard panel must not be null.";
314         assert keyboardPanel.getComponentCount() >= 7 : "Keyboard panel must contain all
     the components.";
315
316         // Add numberPanel and operatorPanel to the keyboardPanel.
317         keyboardPanel.add(numberPanel);
318         keyboardPanel.add(operatorPanel);
319
320         // Assert that numberPanel and operatorPanel are added to the keyboardPanel.
321 assert keyboardPanel.getComponent(keyboardPanel.getComponentCount() -
    3).equals( numberPanel) : "Number panel must be added to the keyboard panel.";
322 assert keyboardPanel.getComponent(keyboardPanel.getComponentCount() -
    2).equals( operatorPanel) : "Operator panel must be added to the keyboard panel.";
323
324         // Add an empty JPanel as a spacer.
325         keyboardPanel.add(new JPanel());
326
327         // Assert that the spacer panel is added to the keyboardPanel.
328 assert keyboardPanel.getComponent(keyboardPanel.getComponentCount() - 1)
    instanceof JPanel : "Spacer panel must be added to the keyboard panel.";
329
330         // Add the keyboardPanel to the frame and make the frame visible.
331         frame.add(keyboardPanel, BorderLayout.SOUTH);
332         frame.setVisible(true);
333         // Assert that the frame is set to be visible.
334         assert frame.isVisible() : "Frame should be visible.";
335     }
336
337     /**
338      * Updates the view to reflect the current state of the model.
339      * This method is called whenever the observed object is changed.
340      *
341      * @param o   The observable object.
342      * @param arg An argument passed to the notifyObservers method.
343      */
344     @Override
345     public void update(java.util.Observable o, Object arg) {
346         // Assert that the controller and guessPanel are not null.
347         assert controller != null : "Controller must not be null.";
348         assert guessPanel != null : "Guess panel must not be null.";
349
350         // Update the attempts label with the remaining attempts from the controller.
351         attemptsLabel.setText("Attempts remaining: " + controller.getRemainingAttempts
    ());
352         // Assert that the attemptsLabel is not null and has been updated.
353         assert attemptsLabel != null : "Attempts label must not be null.";
354 assert attemptsLabel.getText().contains(String.valueOf(controller.
    getRemainingAttempts())) : "Attempts label text must be updated with remaining attempts
    .";
355
356         // Update the target label with the target word from the controller, formatting
```

```
356 division and multiplication symbols for display.
357         targetLabel.setText("The target is " + controller.getTargetWord().replaceAll("/"
    ,"÷").replaceAll("\\*","×"));
358         // Assert that the targetLabel is not null and has been updated.
359         assert targetLabel != null : "Target label must not be null.";
360         assert targetLabel.getText().contains(controller.getTargetWord().replaceAll("/",
    "÷").replaceAll("\\*","×")) : "Target label text must be updated with the target word.";
361
362         // Retrieve the list of guesses and comparison results from the controller.
363         ArrayList<String> guessList = controller.getGuessList(); // List of guesses made
     by the user.
364         ArrayList<int[]> compareList = controller.getCompareList(); // List of
    comparison results for each guess.
365         // Assert that the guessList and compareList are not null and are synchronized
    in size.
366         assert guessList != null : "Guess list must not be null.";
367         assert compareList != null : "Compare list must not be null.";
368         assert guessList.size() == compareList.size() : "Guess list and compare list
    must be of the same size.";
369
370         // Update the state of the new game button.
371         newGameButton.setEnabled((!controller.getGuessList().isEmpty()));
372
373         // Update the flags for verifying, displaying, and randomizing equations.
374         updateFlags();
375
376         // Update the guess panel with the guesses and comparison results.
377         for (int i = 0; i < guessList.size(); i++) {
378             String guess = guessList.get(i);
379             int[] compare = compareList.get(i);
380             for (int j = 0; j < compare.length; j++) {
381                 JLabel label = (JLabel) guessPanel.getComponent(i * 7 + j);
382                 // Assert that each label is not null.
383                 assert label != null : "Label at index " + (i * 7 + j) + " must not be
    null.";
384                 if (j < guess.length()) {
385                     // Set the text of the label to the current character of the guess.
386                     label.setText(String.valueOf(guess.charAt(j)));
387                     // Assert that the label text is set correctly.
388                     assert label.getText().equals(String.valueOf(guess.charAt(j))) : "
    Label text must be set to the character at index " + j + " of the guess.";
389                     // Change the background color of the label based on the comparison
    result.
390                     switch (compare[j]) {
391                         case 0 -> label.setBackground(new Color(164, 174, 196));
392                         case 1 -> label.setBackground(new Color(47, 193, 165));
393                         case 2 -> label.setBackground(new Color(247, 154, 111));
394                     }
395                     // Assert that the label background is set correctly.
396                     assert label.getBackground() != null : "Label background must be set
     based on the comparison result.";
397                 } else {
398                     // If there is no character at this position, set the label to blank
    .
399                     label.setText(" ");
400                     label.setBackground(Color.WHITE);
401                     // Assert that the label is reset correctly for empty guess
    positions.
402                     assert label.getText().equals(" ") : "Label text must be set to a
    space for empty guess positions.";
403                     assert label.getBackground().equals(Color.WHITE) : "Label background
     must be set to white for empty guess positions.";
404                 }
405             }
406         }
407         updateButtonColors();
408
```

```
409         // Check if the game is over
410         if (controller.isGameOver()) {
411             showGameOverDialog();
412         }
413     }
414 
415     /**
416      * Updates the visual flags for the verify, display, and random settings based on the current game state.
417      * Precondition: None
418      * Postcondition: The color of the flags is updated to reflect the current settings.
419      */
420     private void updateFlags() {
421         // Assert that the controller is not null.
422         assert controller != null : "Controller must not be null.";
423         // Update the verifySign color based on the controller's state.
424         if (controller.getVerifyEquation())
425             verifySign.setForeground(new Color(47,193,165));
426         else
427             verifySign.setForeground(Color.BLACK);
428         // Assert that verifySign is not null.
429         assert verifySign != null : "Verify sign must not be null.";
430 
431         // Update the displaySign and targetLabel visibility based on the controller's state.
432         if (controller.getDisplayEquation()) {
433             displaySign.setForeground(new Color(47,193,165));
434             targetLabel.setVisible(true);
435         }
436         else {
437             displaySign.setForeground(Color.BLACK);
438             targetLabel.setVisible(false);
439         }
440         // Assert that displaySign and targetLabel are not null.
441         assert displaySign != null : "Display sign must not be null.";
442         assert targetLabel != null : "Target label must not be null.";
443 
444         // Update the randomSign color based on the controller's state.
445         if (controller.getRandomEquality())
446             randomSign.setForeground(new Color(47,193,165));
447         else
448             randomSign.setForeground(Color.BLACK);
449         // Assert that randomSign is not null.
450         assert randomSign != null : "Random sign must not be null.";
451     }
452 
453     /**
454      * Updates the colors of the buttons based on the classification of characters in guesses.
455      * Precondition: The model must provide valid classification lists.
456      * Postcondition: The buttons' colors are updated to reflect the classification of characters.
457      */
458     private void updateButtonColors() {
459         // Assert that model and buttonMap are not null.
460         assert model != null : "Model must not be null.";
461         assert buttonMap != null : "Button map must not be null.";
462 
463         // Classification lists used for tracking character matches.
464         ArrayList<Character>[] classList = controller.getClassList();
465         // Assert that classList is not null and has the expected number of lists.
466         assert classList != null : "Class list must not be null.";
467         assert classList.length == 4 : "Class list must contain four lists for character classification.";
468 
469         // Reset all button colors.
470         buttonMap.values().forEach(button -> {
```

```
471             button.setBackground(new Color(220, 225, 237));
472             button.setHoverBackgroundColor(new Color(200, 200, 200));
473             button.setPressedBackgroundColor(new Color(150, 150, 150));
474         });
475         // Set colors for characters not present in the target.
476         for (char c : classList[0]) {
477             buttonMap.get(c).setBackground(new  Color(164,174,196));
478             buttonMap.get(c).setHoverBackgroundColor(new Color(144, 154, 176));
479             buttonMap.get(c).setPressedBackgroundColor(new Color(124, 134, 156));
480         }
481         // Set colors for characters present but in the wrong position.
482         for (char c : classList[2]) {
483             buttonMap.get(c).setBackground(new  Color(247,154,111));
484             buttonMap.get(c).setHoverBackgroundColor(new Color(227, 134, 81));
485             buttonMap.get(c).setPressedBackgroundColor(new Color(207, 114, 61));
486         }
487         // Set colors for correctly positioned characters.
488         for (char c : classList[1]) {
489             buttonMap.get(c).setBackground(new  Color(47,193,165));
490             buttonMap.get(c).setHoverBackgroundColor(new Color(67, 213, 185));
491             buttonMap.get(c).setPressedBackgroundColor(new Color(27, 173, 145));
492         }
493         // Assert that each character in classList has a corresponding button in
    buttonMap.
494         for (ArrayList<Character> classGroup : classList) {
495             for (char c : classGroup) {
496                 assert buttonMap.containsKey(c) : "Button map must contain a button for
    character: " + c;
497             }
498         }
499     }
500
501     /**
502      * Displays a dialog for game settings before starting a new game.
503      * Precondition: The controller must provide the current settings for verification,
    display, and random equality.
504      * Postcondition: The game settings are updated based on user input, and a new game
    is started or the application is exited.
505      */
506     private void showGameSettingsDialog() {
507         // Assert that the controller is not null.
508         assert controller != null : "Controller must not be null.";
509
510         // Initialize checkboxes with current settings from the controller.
511         verifyEquationCheckBox = new JCheckBox("Verify Equation", controller.
    getVerifyEquation());
512         displayEquationCheckBox = new JCheckBox("Display Equation", controller.
    getDisplayEquation());
513         randomEqualityCheckBox = new JCheckBox("Random Equality", controller.
    getRandomEquality());
514
515         // Create and configure the start game button.
516         RoundedButton startButton = new RoundedButton("Start Game");
517         startButton.addActionListener(e -> {
518             // Update model flags based on checkbox selections.
519             controller.setVerifyEquation(verifyEquationCheckBox.isSelected());
520             controller.setDisplayEquation(displayEquationCheckBox.isSelected());
521             controller.setRandomEquality(randomEqualityCheckBox.isSelected());
522             // Dispose of the dialog and start a new game.
523             Window dialog = SwingUtilities.getWindowAncestor(startButton);
524             if (dialog != null) {
525                 dialog.dispose();
526             }
527             controller.startNewGame();
528             clearGameView();
529             initializeFrame();
530             updateFlags();
```

```
531         });
532
533         // Create and configure the quit button.
534         RoundedButton exitButton = new RoundedButton("Quit");
535         exitButton.addActionListener(e -> System.exit(0));
536
537         // Display the dialog with the game settings options.
538         Object[] options = {startButton, exitButton};
539         JOptionPane.showOptionDialog(frame,
540                 new Object[]{verifyEquationCheckBox, displayEquationCheckBox,
    randomEqualityCheckBox},
541                 "Starting Settings",
542                 JOptionPane.YES_NO_OPTION,
543                 JOptionPane.PLAIN_MESSAGE,
544                 null,
545                 options,
546                 options[0]);
547     }
548
549     /**
550      * Displays a dialog indicating the end of the game with a message based on the
    outcome.
551      * Precondition: The controller must provide the game outcome and target word if the
     game is lost.
552      * Postcondition: The dialog is displayed with the game outcome message and options
    to start a new game or quit.
553      */
554     private void showGameOverDialog() {
555         // Assert that the controller is not null.
556         assert controller != null : "Controller must not be null.";
557
558         // Determine the message and title based on the game outcome.
559         String message = controller.isGameWon() ? "<html>Congratulations!<br>Equation
    correct. You win.</html>" : "<html>Run out of opportunity.<br>Target is " + controller.
    getTargetWord().replaceAll("/","÷").replaceAll("\\*","×") + ". You lose. </html>";
560         String title = controller.isGameWon() ? "WIN" : "LOSE";
561
562         // Create and configure the message label.
563         JLabel messageLabel = new JLabel(message);
564         messageLabel.setFont(new Font("Montserrat", Font.PLAIN, 20));
565         if (controller.isGameWon())
566             messageLabel.setForeground(new  Color(0,240,0));
567         else
568             messageLabel.setForeground(Color.GRAY);
569
570         // Create and configure the start game button.
571         RoundedButton startButton = new RoundedButton("Start Game");
572         startButton.addActionListener(e -> {
573             // Update model flags based on checkbox selections.
574             controller.setVerifyEquation(verifyEquationCheckBox.isSelected());
575             controller.setDisplayEquation(displayEquationCheckBox.isSelected());
576             controller.setRandomEquality(randomEqualityCheckBox.isSelected());
577             // Dispose of the dialog and start a new game.
578             Window dialog = SwingUtilities.getWindowAncestor(startButton);
579             if (dialog != null) {
580                 dialog.dispose();
581             }
582             clearGameView();
583             controller.startNewGame();
584             initializeFrame();
585         });
586
587         // Create and configure the quit button.
588         RoundedButton exitButton = new RoundedButton("Quit");
589         exitButton.addActionListener(e -> System.exit(0));
590
591         // Display the dialog with the game outcome message and options.
```

```
592         Object[] options = {startButton, exitButton};
593         Object[] dialogContent = {messageLabel, verifyEquationCheckBox,
    displayEquationCheckBox, randomEqualityCheckBox};
594         JOptionPane optionPane = new JOptionPane(
595                 dialogContent,
596                 JOptionPane.PLAIN_MESSAGE,
597                 JOptionPane.YES_NO_OPTION,
598                 null,
599                 options,
600                 options[0]);
601601
602         // Create a dialog and set the JOptionPane as its content pane.
603         JDialog dialog = optionPane.createDialog(frame, title);
604         // Set the default close operation to DO_NOTHING_ON_CLOSE.
605         dialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
606         // Display the custom dialog.
607         dialog.setVisible(true);
608     }
609
610     /**
611      * Displays a message dialog to inform the user of invalid input based on the status
     code.
612      * Precondition: statusCode must be an integer representing a specific type of input
     validation error.
613      * Postcondition: A message dialog is displayed with information about the input
    error.
614      *
615      * @param statusCode The code representing the type of input error.
616      */
617     public void showInvalidInputMessage(int statusCode) {
618         // Assert that the statusCode is within the expected range.
619         assert statusCode > 0 && statusCode <= 4 : "Status code must be between 1 and 4
    .";
620
621         // Determine the message based on the status code.
622         String message = switch (statusCode) {
623             case 2 -> "The length of the equation is not 7.";
624             case 3 -> "The input string does not follow the form of an ordinary equation
    .";
625             case 4 -> "The two sides of the equation don't agree. The equation doesn't
    hold.";
626         default -> "The input is valid."; 627
627         };
628
629         // Display the message dialog with the determined message.
630         JOptionPane.showMessageDialog(null, message, "Tips", JOptionPane.
    INFORMATION_MESSAGE);
631         // Assert that the message is not null or empty.
632         assert message != null && !message.isEmpty() : "Message must not be null or
    empty.";
633     }
634
635     /**
636      * Clears the game view by removing all components from the frame.
637      * Precondition: None
638      * Postcondition: The frame is cleared of all components and repainted.
639      */
640     private void clearGameView() {
641         // Assert that the frame is not null before attempting to clear it.
642         assert frame != null : "Frame must not be null.";
643         // Remove all components from the frame.
644         frame.getContentPane().removeAll();
645         // Repaint the frame to update the display.
646         frame.repaint();
647         // Assert that the frame's content pane is empty after clearing.
648         assert frame.getContentPane().getComponentCount() == 0 : "Frame's content pane
    should be empty after clearing.";
```

File- D:\CDUT\AOOP\CW\src\View\NumberleView.java

```
649         }
650 }
```

File- D:\CDUT\AOOP\CW\src\Model\NumberleModel.java

```java
package Model;// Model.NumberleModel.java
import CustomClass.EquationGenerator;
import Model.Interface.INumberleModel;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.*;

/**
 * The Model.NumberleModel class represents the model component in the MVC pattern.
 * It maintains the game state for the Numberle game, including the target number,
 * current guess, list of guesses, and comparison results.
 *
 * @invariant ("The target number must always be a valid equation or number.")
*targetNumber.matches("[0-9]+") || targetNumber.matches("valid equation regex");
 * @invariant ("The remaining attempts should be non-negative.")
 *             remainingAttempts >= 0;
 * @invariant ("The guessList and compareList should have the same size.")
 *             guessList.size() == compareList.size();
 * @invariant ("The game is won if and only if the currentGuess equals the targetNumber."
)
 *             gameWon == targetNumber.equals(currentGuess.toString());
 */
public class NumberleModel extends Observable implements INumberleModel {
    private String targetNumber; // The target number or equation to guess
    private StringBuilder currentGuess; // The current guess input by the user
    private ArrayList<String> guessList; // List of all guesses made
    private ArrayList<int[]> compareList; // List of comparison results for each guess
    private ArrayList<Character>[] classList; // Used for storing character
classifications
    private int remainingAttempts; // Number of attempts left for the user
    private boolean gameWon; // Indicates if the game has been won
    //@ requires newRandom != null;
    //@ ensures rand != null;
    private final Random rand = new Random(); // Random number generator

    //@ ensures verifyEquation == true;
private boolean verifyEquation = false; // Flag to verify the correctness of
equations

    //@ ensures displayEquation == false;
    private boolean displayEquation = false; // Flag to display the equation

    //@ ensures randomEquality == false;
private boolean randomEquality = false; // Flag to use random equality in the target
number

    //@ requires generator != null;
    //@ ensures (\result instanceof EquationGenerator);
private final EquationGenerator generator = new EquationGenerator(); // Generates
equations

    /**
     * Sets the flag to verify the correctness of the equation.
     *
     * @param verifyEquation The new value for the verifyEquation flag.
     * @pre The method can be called without a precondition.
     * @post The verifyEquation flag is updated to the value of the parameter.
     */
    public void setVerifyEquation(boolean verifyEquation) {
        assert remainingAttempts >= 0 : "Remaining attempts must be non-negative.";
    this.verifyEquation = verifyEquation; 59
    }

    /**
```

```
62      * Sets the flag to display the equation.
63      *
64      * @param displayEquation The new value for the displayEquation flag.
65      * @pre The method can be called without a precondition.
66      * @post The displayEquation flag is updated to the value of the parameter.
67      */
68     public void setDisplayEquation(boolean displayEquation) {
69         assert remainingAttempts >= 0 : "Remaining attempts must be non-negative.";
70     this.displayEquation = displayEquation; 71 }
72
73     /**
74      * Sets the flag to use random equality in the target number.
75      *
76      * @param randomEquality The new value for the randomEquality flag.
77      * @pre The method can be called without a precondition.
78      * @post The randomEquality flag is updated to the value of the parameter.
79      */
80     public void setRandomEquality(boolean randomEquality) {
81         assert remainingAttempts >= 0 : "Remaining attempts must be non-negative.";
82     this.randomEquality = randomEquality; 83
       }
84
85     /**
86      * Initializes the game state, including setting up the target number,
87      * initializing the current guess, and preparing data structures.
88      *
89      * @pre The method can be called without a precondition.
90      * @post guessList and compareList are empty.
91      * @post classList is initialized with empty ArrayLists.
92      * @post targetNumber is set based on the file or generated equation.
93      * @post remainingAttempts is set to the maximum allowed attempts.
94      * @post gameWon is false.
95      */
96     @Override
97     public void initialize() {
98         guessList = new ArrayList<>();
99         compareList = new ArrayList<>();
100        classList = new ArrayList[4];
101        // Read the equation from the file.
102        targetNumber = getEquationFromFile();
103 //        // This section of code is used to generate random equations and should be
       used as an alternative to the getEquationFromFile method.
104 //        if (randomEquality)
105 //            generator.generateEquation();
106 //        else {
107 //            if (targetNumber == null)
108 //                generator.generateEquation();
109 //        }
110 //        targetNumber = generator.getEquation();
111
112        currentGuess = new StringBuilder("       ");
113        remainingAttempts = MAX_ATTEMPTS;
114        gameWon = false;
115
116        // The assertion list is initialized to empty
117        assert guessList.isEmpty() : "guessList Should be empty";
118        assert compareList.isEmpty() : "compareList Should be empty";
119
120        for (int i = 0; i < 4; i ++) {
121            classList[i] = new ArrayList<>();
122            assert classList[i] != null : "The elements in the classList should be
       initialized";
123        }
124
125        // Asserts that targetNumber is not empty
126        assert targetNumber != null : "targetNumber should not be empty";
```

```
127
128         // Apply characters to classList[3]
129         String a = "0123456789+-*/=";
130         for (int i = 0; i < a.length(); i ++) {
131             classList[3].add(a.charAt(i));
132         }
133
134         // Assert that remainingAttempts are set correctly
135 assert remainingAttempts == MAX_ATTEMPTS : "remainingAttempts should be set to
    MAX_ATTEMPTS";
136         // assert the game unwon
137         assert !gameWon : "gameWon should be initialized to false";
138
139         // Display the target equation if needed
140         if (displayEquation)
141             System.out.println("Target is:" + getTargetNumber());
142         // Notifying the observer that the model has changed
143         setChanged();
144         notifyObservers();
145     }
146
147     /**
148      * Processes the user input, updates the game state, and provides feedback.
149      *
150      * @param input The user's input (equation or number).
151 * @requires ("The input must be a non-null and non-empty string representing a
    valid equation or number.")
152      *           input != null && !input.isEmpty() && isValidEquationOrNumber(input);
153      * @ensures ("The current guess is updated based on the input.")
154      *          currentGuess.toString().equals(input);
155      * @ensures ("guessList contains the input, formatted as needed.")
156      *          guessList.contains(input.replaceAll("/", "÷").replaceAll("\\*", "×"));
157      * @ensures ("compareList contains the comparison results for the guess.")
158      *          compareList.size() == guessList.size();
159      * @ensures ("remainingAttempts is decremented.")
160      *          remainingAttempts < \old(remainingAttempts);
161      * @ensures ("If the input matches the targetNumber, gameWon is set to true.")
162      *          (input.replaceAll("÷", "/").replaceAll("×", "*").equals(targetNumber
    )) ==> gameWon;
163 * @ensures ("If the input is invalid, appropriate messages are displayed and the
    observer(s) are notified of the state change.")
164 *     // This is handled within the method body and is not directly
    represented in JML.
165      * @return An integer code representing the result of processing the input:
166      *         - 1: Correct guess
167      *         - 2: Invalid input length
168      *         - 3: Invalid input format (not a valid equation)
169      *         - 4: Invalid equation (does not satisfy the equation rules)
170      */
171     @Override
172     public int processInput(String input) {
173         // Prerequisite assertion: The input should be a valid equation or number
174         assert input != null : "Input should not be null";
175         assert !input.isEmpty() : "The input should not be empty";
176
177         if (verifyEquation) {// Check that the input length is correct
178             if (input.length() != 7) {
179                 System.out.println("Invalid input length. Please try again.");
180                 setChanged();
181                 notifyObservers();
182                 return 2;
183             }
184
185             // Use regular expressions to check that the input conforms to the form of
    ordinary equations
186             if (!input.replaceAll("÷","/").replaceAll("×","*").matches("^(?!\\b(\\d{1,3
    })\\b=\\1$)(\\d{1,3}[+\\-*/]){0,2}-?\\d{1,3}=(\\d{1,3}[+\\-*/]){0,2}-?\\d{1,3}$")) {
```

```
187                 System.out.println("Invalid input. Please enter a valid equation.");
188                 setChanged();
189                 notifyObservers();
190                 return 3;
191             }
192 
193             // Check for validity
194             if (checkEquation(input.replaceAll("÷","/").replaceAll("×","*"))) {
195                 System.out.println("The input equation is valid.");
196             } else {
197                 System.out.println("The input equation is invalid.");
198                 return 4;
199             }
200         }
201         remainingAttempts--;
202         // Update current guesses
203         currentGuess = new StringBuilder(input);
204         guessList.add(input.replaceAll("/","÷").replaceAll("\\*","×"));
205 
206         // Match the target with the guessed character
207 int[] compared = compareStrings(getTargetNumber(),input.replaceAll("÷","/").
replaceAll("×","*"));
208         compareList.add(compared);
209 
210         // Postcondition assertion
211 assert remainingAttempts < MAX_ATTEMPTS : "The number of remaining attempts should
be reduced";
212 assert guessList.contains(input.replaceAll("/","÷").replaceAll("\\*","×")) : " The
guessList should contain the input";
213 assert compareList.size() == guessList.size() : "The size of compareList should be the
same as that of guessList";
214 assert currentGuess.toString().equals(input) : "currentGuess should be updated to the
value entered";
215 
216         // Check if your guesses are correct
217         if (input.replaceAll("÷","/").replaceAll("×","*").equals(targetNumber)) {
218             gameWon = true;
219             showHistory();
220             System.out.println("Congratulations! You've guessed the right equality.");
221         } else {
222             System.out.println("Incorrect guess. Your match is:");
223             if (displayEquation)
224                 System.out.println("Target is:" + getTargetNumber());
225             showHistory();
226             if (!isGameOver())
227                 System.out.print("Try again.");
228             System.out.println("You have " + getRemainingAttempts() + " more chances.");
229         }
230 
231         // If the guess is correct, the game-winning flag should be true
232 assert !input.replaceAll("÷", "/").replaceAll("×", "*").equals(targetNumber) ||
gameWon : "If the input matches targetNumber, gameWon should be true";
233 
234         // Notifying the observer that the model has changed
235         setChanged();
236         notifyObservers();
237 
238         return 1;
239     }
240 
241     /**
242      * Displays the history of guesses and their comparison results.
243      *
244      * @pre The method can be called without a precondition.
245      * @post The history of guesses and comparison results is printed to the console.
246      */
247     private void showHistory() {
```

```
248         assert compareList != null : "The comparison list should not be null.";
249         assert guessList != null : "The guess list should not be null.";
250 
251         for (int[] row : compareList) {
252             assert row != null : "The row in the comparison list should not be null.";
253             for (int value : row) {
254                 System.out.print(value + " ");
255             }
256             System.out.println();
257         }
258         for (String row : guessList) {
259             assert row != null : "The row in the guess list should not be null.";
260             for (int i = 0; i < row.length(); i++) {
261                 System.out.print(row.charAt(i) + " ");
262             }
263             System.out.println();
264         }
265     }
266 
267     /**
268      * Checks if the given input string is a valid equation by comparing the evaluated
    results of both sides.
269      *
270      * @param input The input string containing the equation to check.
271 * @requires ("The input must be a non-null and non-empty string representing an
    equation.")
272      *           input != null && !input.isEmpty();
273 * @ensures ("Returns true if the equation is valid, which means the evaluated
    results of both sides are equal.")
274      *          \result == (evaluateExpression(leftSide) == evaluateExpression(rightSide
    ));
275      * @return boolean indicating whether the equation is valid.
276      */
277     private boolean checkEquation(String input) {
278         assert input != null : "Input equation should not be null.";
279         assert !input.isEmpty() : "Input equation should not be empty.";
280 
281         // Remove the equal sign and split the equation
282         String[] parts = input.split("=");
283         if (parts.length != 2) {
284             System.out.println("Invalid equation format. Please enter a valid equation."
    );
285             return false;
286         }
287 
288         String leftSide = parts[0].trim();
289         String rightSide = parts[1].trim();
290 
291         // Evaluates the value of the expression on the left
292         int leftValue = evaluateExpression(leftSide);
293         // Evaluates the value of the expression on the right
294         int rightValue = evaluateExpression(rightSide);
295 
296         return leftValue == rightValue;
297     }
298 
299     /**
300      * Evaluates the given mathematical expression and returns the result.
301      *
302      * @param expression The mathematical expression to evaluate (e.g., "2 + 3 * 4").
303  * @requires ("The expression must be a non-null and non-empty string representing a
     valid mathematical expression.")
304      *           expression != null && !expression.isEmpty();
305      * @ensures ("The result of evaluating the expression is returned.")
306      *          \result == (\old(evaluateExpression(expression)));
307      * @return The result of evaluating the expression.
308      */
```

```
309     private int evaluateExpression(String expression) {
310         // Precondition: The expression should be a non-null and non-empty string.
311         assert expression != null : "The expression cannot be null.";
312         assert !expression.isEmpty() : "The expression cannot be empty.";
313 
314         // Remove all spaces
315         expression = expression.replaceAll("\\s+", "");
316         // Split the expression into tokens (numbers and operators)
317         List<String> tokens = new
    ArrayList<>(Arrays.asList(expression.split( "(?=[+*/-])|(?<=[+*/-])")));
318 
319         // Precondition: The tokens list should not be empty after splitting.
320         assert !tokens.isEmpty() : "The tokens list cannot be empty.";
321 
322         // Handle multiplication and division
323         for (int i = 1; i < tokens.size(); i++) {
324             String token = tokens.get(i);
325             if (token.equals("*") || token.equals("/")) {
326                 int result = calculateResult(Integer.parseInt(tokens.get(i - 1)),
    Integer.parseInt(tokens.get(i + 1)), token.charAt(0));
327                 // Replace the calculated part
328                 tokens.set(i - 1, Integer.toString(result));
329                 tokens.remove(i); // Remove the operator
330                 tokens.remove(i); // Remove the second number
331                 i--; // Backtrack one step to handle consecutive multiplications/
    divisions
332             }
333         }
334 
335         // Handle addition and subtraction
336         String char1 = tokens.get(0);
337         int result;
338         if (!char1.equals("-")) {
339             result = Integer.parseInt(tokens.get(0));
340             for (int i = 1; i < tokens.size(); i += 2) {
341                 result = calculateResult(result, Integer.parseInt(tokens.get(i + 1)),
    tokens.get(i).charAt(0));
342             }
343         } else {
344             result = -Integer.parseInt(tokens.get(1));
345             for (int i = 2; i < tokens.size(); i += 2) {
346                 result = calculateResult(result, Integer.parseInt(tokens.get(i + 1)),
    tokens.get(i).charAt(0));
347             }
348         }
349 
350         return result;
351     }
352 
353     /**
354      * Compares two strings and returns an array of integers indicating the match status
    .
355      * Each index in the result array corresponds to the character in the input string.
356 * A value of 1 indicates an exact match, 2 indicates a character match but in a
    different position,
357      * and 0 indicates no match.
358      *
359      * @param target The target string to compare against.
360      * @param input The input string to be compared.
361      * @requires ("Both target and input must be non-null strings.")
362      *           target != null && input != null;
363 * @requires ("classList[3] must contain all characters that can be used in the
    input.")
364      *           classList[3] != null;
365 * @ensures ("Returns an array of integers representing the match status for each
    character in the input.")
366      *          \result != null;
```

```
367 * @ensures ("classList[0] contains characters found in the input but not in the target.")
368 *     (\forall char c; input.contains(c) && !target.contains(c); classList[0].contains(c));
369      * @ensures ("classList[1] contains characters with an exact match.")
370 *     (\forall int i; 0 <= i && i < input.length(); input.charAt(i) == target. charAt(i) ==> classList[1].contains(input.charAt(i)));
371 * @ensures ("classList[2] contains characters that match but are in different positions.")
372 *     (\forall int i; 0 <= i && i < input.length(); (\exists int j; 0 <= j && j < target.length(); input.charAt(i) == target.charAt(j) && i != j) ==> classList[2].contains(input.charAt(i)));
373      * @ensures ("classList[3] contains characters not yet matched.")
374      *          (\forall char c; !input.contains(c) || !target.contains(c); classList[3].contains(c));
375 * @return An array of integers indicating the match status for each character in the input.
376      */
377     private int[] compareStrings(String target, String input) {
378         // Precondition: The target and input strings should not be null.
379         assert target != null : "Target string cannot be null.";
380         assert input != null : "Input string cannot be null.";
381 
382         // Precondition: The classList[3] should contain characters for input.
383         assert classList[3] != null : "classList[3] should not be null.";
384 
385         // Initialize the result array with zeros
386         int[] result = new int[target.length()];
387 
388         // Mark characters from input that are present in classList[3]
389         // boolean[] matched = new boolean[target.length()];
390 
391         int minEquationLength = Math.min(target.length(), input.length());
392         for (int i = 0; i < minEquationLength; i++){
393             if (classList[3].contains(input.charAt(i))){
394                 classList[0].add(input.charAt(i));
395                 classList[3].remove(Character.valueOf(input.charAt(i)));
396             }
397         }
398 
399         // Check for exact matches (value 1)
400         for (int i = 0; i < input.length(); i++) {
401             if (i < target.length() && input.charAt(i) == target.charAt(i)) {
402                 result[i] = 1;
403                 if (!classList[1].contains(target.charAt(i))) {
404                     classList[1].add(target.charAt(i));
405                     classList[0].remove(Character.valueOf(target.charAt(i)));
406                 }
407                 //matched[i] = true;
408             }
409         }
410 
411         // Check for character matches but different positions (value 2)
412         for (int i = 0; i < minEquationLength; i++) {
413             if (result[i] != 1) { // If the current position is not an exact match
414                 for (int j = 0; j < target.length(); j++) {
415                     if (input.charAt(i) == target.charAt(j)) {//!matched[j] &&
416                         result[i] = 2;
417                         if (!classList[2].contains(target.charAt(j))) {
418                             classList[2].add(target.charAt(j));
419                             classList[0].remove(Character.valueOf(target.charAt(j)));
420                         }
421                           matched[j] = true;    标记匹配避免重复匹配
422                         break;
423                     }
424                 }
425             }
```

```
426         }
427 
428         // Postcondition: The result array should be non-null.
429         assert result != null : "Result array cannot be null.";
430 
431         return result;
432     }
433 
434     /**
435      * Calculates the result of a binary operation between two integers.
436      *
437      * @param a The first operand.
438      * @param b The second operand (must not be zero if operator is '/').
439      * @param operator The operator ('+', '-', '*', '/').
440      * @requires ("The second operand 'b' must not be zero if the operator is division."
)
441      *           operator != '/' || b != 0;
442      * @ensures ("Returns the result of the operation, or Integer.MAX_VALUE if division
by zero occurs.")
443      *          (\result == a + b && operator == '+') ||
444      *          (\result == a - b && operator == '-') ||
445      *          (\result == a * b && operator == '*') ||
446      *          ((operator == '/' && b != 0) ==> \result == a / b) ||
447      *          ((operator == '/' && b == 0) ==> \result == Integer.MAX_VALUE);
448      * @return The result of the operation.
449      */
450     private int calculateResult(int a, int b, char operator) {
451         // Precondition: 'b' should not be zero if the operator is division.
452         assert operator != '/' || b != 0 : "Second operand must not be zero for division
.";
453         // The switch statement handles the calculation based on the operator.
454         int result = switch (operator)
455             { case '+' -> a + b;
456             case '-' -> a - b;
457             case '*' -> a * b;
458             case '/' -> a / b; // Division by zero is checked by the precondition.
459             default -> throw new IllegalArgumentException("Invalid operator");
460         };
461 
462         // Postcondition: The result should be within the range of integers.
463         assert Integer.MIN_VALUE <= result && result <= Integer.MAX_VALUE : "Result out
of range.";
464 
465         return result;
466     }
467 
468     /**
469      * Sets the target number or equation for the game.
470      *
471      * @param target The new target number or equation.
472      * @requires ("The target must be a non-null and non-empty string.")
473      *           target != null && !target.isEmpty();
474      * @ensures ("The targetNumber is updated to the new target.")
475      *          targetNumber.equals(target);
476      */
477     public void setTarget(String target){
478         // Precondition: The target should be a valid equation or number.
479         assert target != null && !target.isEmpty() : "Target must be a non-null and non-
empty string.";
480         targetNumber = target;
481         // Postcondition: The targetNumber is updated.
482         assert targetNumber.equals(target) : "Target number not updated correctly.";
483     }
484 
485     /**
486      * Checks if the game is over, which occurs when no attempts remain or the game has
been won.
```

```
487      *
488      * @return boolean indicating whether the game is over.
489      * @ensures ("Returns true if the game is over, which is when no attempts remain or
    the game has been won.")
490      *          \result == (remainingAttempts <= 0 || gameWon);
491      */
492     @Override
493     public boolean isGameOver() {
494         // The method checks if the game is over based on remaining attempts or if the
    game has been won.
495         boolean gameOver = remainingAttempts <= 0 || gameWon;
496 
497         // Postcondition: The method returns a boolean indicating the game's over status
    .
498         assert gameOver == (remainingAttempts <= 0 || gameWon) : "Game over status
    incorrect.";
499 
500         return gameOver;
501     }
502 
503     /**
504      * Checks if the game has been won.
505      *
506      * @return boolean indicating whether the game has been won.
507      * @ensures ("Returns true if the game has been won, false otherwise.")
508      *          \result == gameWon;
509      */
510     @Override
511     public boolean isGameWon() {
512         // The method checks if the game has been won.
513         // Postcondition: The method returns a boolean indicating the game's won status.
514         assert gameWon == this.gameWon : "Game won status incorrect.";
515 
516         return gameWon;
517     }
518 
519     /**
520      * Retrieves the target number or equation for the game.
521      *
522      * @return The target number or equation.
523      * @ensures ("Returns the current target number or equation.")
524      *          \result.equals(targetNumber);
525      */
526     @Override
527     public String getTargetNumber() {
528         // The method retrieves the target number or equation for the game.
529         // Postcondition: The method returns the current target number or equation.
530         assert targetNumber != null : "Target number cannot be null.";
531 
532         return targetNumber;
533     }
534 
535     /**
536      * Retrieves the remaining attempts left for the user.
537      *
538      * @return The number of remaining attempts as an integer.
539      * @ensures ("The returned number of remaining attempts is non-negative.")
540      *          \result >= 0;
541      */
542     @Override
543     public int getRemainingAttempts() {
544         // Postcondition: The remaining attempts should be non-negative.
545         assert remainingAttempts >= 0 : "Remaining attempts cannot be negative.";
546         return remainingAttempts;
547     }
548 
549     /**
```

```
550      * Starts a new game by initializing the game state.
551      *
552      * @ensures ("The game state is reset, and a new game begins.")
553      */
554     @Override
555     public void startNewGame() {
556         System.out.println("Game is start now! You have a total of 6 chances.");
557         initialize();
558     }
559 
560     /**
561      * Retrieves the list of guesses made by the user.
562      *
563      * @return An ArrayList containing the user's guesses.
564      * @ensures ("The returned list is not null and contains the user's guesses.")
565      *          \result != null;
566      */
567     @Override
568     public ArrayList<String> getGuessList() {
569         // Postcondition: The guess list should not be null.
570         assert guessList != null : "Guess list cannot be null.";
571         return guessList;
572     }
573 
574     /**
575      * Retrieves the list of comparison results for each guess.
576      *
577      * @return An ArrayList containing int arrays representing comparison results.
578      * @ensures ("The returned list is not null and contains the comparison results for
    each guess.")
579      *          \result != null;
580      */
581     @Override
582     public ArrayList<int[]> getCompareList() {
583         // Postcondition: The compare list should not be null.
584         assert compareList != null : "Compare list cannot be null.";
585         return compareList;
586     }
587 
588     /**
589      * Retrieves the flag indicating whether equations should be displayed.
590      *
591      * @return boolean indicating if equations should be displayed.
592      * @ensures ("Returns the value of the displayEquation flag.")
593      *          \result == displayEquation;
594      */
595     @Override
596     public boolean getDisplayEquation() {
597         return displayEquation;
598     }
599 
600     /**
601      * Retrieves the flag indicating whether equations should be verified for
    correctness.
602      *
603      * @return boolean indicating if equations should be verified.
604      * @ensures ("Returns the value of the verifyEquation flag.")
605      *          \result == verifyEquation;
606      */
607     @Override
608     public boolean getVerifyEquation()
{ 609         return verifyEquation;
610     }
611 
612     /**
613      * Retrieves the flag indicating whether the equality in the target number should be
     random.
```

```
614       *
615       * @return boolean indicating if the equality should be random.
616       * @ensures ("Returns the value of the randomEquality flag.")
617       *          \result == randomEquality;
618       */
619      @Override
620      public boolean getRandomEquality() {
621          return randomEquality;
622      }
623
624      /**
625       * Retrieves an equation from a file or the current target number based on the
    randomEquality flag.
626       *
627       * @return String containing a valid equation or null if an error occurs.
628       * @requires ("The file \"equations.txt\" must exist and be readable.")
629       *           Files.exists(Paths.get("equations.txt")) && Files.isReadable(Paths.get
    ("equations.txt"));
630       * @ensures ("Returns a valid equation from the file or the current target number.")
631       *          (\result != null) && (randomEquality ? allEquations.contains(\result
    ) : \result.equals(targetNumber));
632       */
633      private String getEquationFromFile()
634      {
635          List<String> allEquations;
636          try {
637              allEquations = Files.readAllLines(Paths.get("equations.txt"));
638          } catch (IOException e) {
639              e.printStackTrace();
640              // Postcondition: If an exception occurs, the method returns null.
641              assert false : "IOException occurred while reading the file.";
642              return null; // Or handle the error appropriately.
643          }
644
645          // Postcondition: The list of all equations should not be empty.
646          assert allEquations != null && !allEquations.isEmpty() : "List of equations
    cannot be empty.";
647
648          if (randomEquality)
649              return allEquations.get(rand.nextInt(allEquations.size()));
650          else {
651              if (targetNumber != null)
652                  return targetNumber;
653              else
654                  return allEquations.get(rand.nextInt(allEquations.size()));
655          }
656      }
657
658      /**
659       * Retrieves the classification lists used for tracking character matches.
660       *
661       * @return ArrayList[] containing classification lists.
662       * @ensures ("Returns the array of ArrayLists representing character classifications
    .")
663       *          \result != null;
664       */
665      @Override
666      public ArrayList[] getClassList(){
667          // Postcondition: The class list should not be null.
668          assert classList != null : "Class list cannot be null.";
669          return classList;
670      }
671  }
```

```java
package Model.Interface;

import java.util.ArrayList;

/**
 * The Model.Interface.INumberleModel interface defines the core functionalities of the model component
 * in the MVC pattern for the Numberle game. It declares methods to manage the game state,
 * process user input, and retrieve game information.
 *
 * Invariant:
 * - MAX_ATTEMPTS should be a positive integer representing the maximum number of attempts allowed.
 */
public interface INumberleModel {
    // Maximum number of attempts a player has to guess the correct number or equation.
    int MAX_ATTEMPTS = 6;

    /**
     * Initializes or resets the game state to start a new game.
     * Precondition: None
     * Postcondition: The game state is initialized and ready for a new game.
     */
    void initialize();

    /**
     * Processes the user's input and updates the game state accordingly.
     * Precondition: input is a non-null string representing a valid guess.
     * Postcondition: Returns an integer indicating the result of processing the input.
     *
     * @param input The user's guess to be processed.
     * @return An integer code representing the outcome of the input processing.
     */
    int processInput(String input);

    /**
     * Checks if the game is over, which occurs when no attempts remain or the game has been won.
     * Precondition: None
     * Postcondition: Returns true if the game is over, false otherwise.
     *
     * @return boolean indicating whether the game is over.
     */
    boolean isGameOver();

    /**
     * Checks if the game has been won.
     * Precondition: None
     * Postcondition: Returns true if the game has been won, false otherwise.
     *
     * @return boolean indicating whether the game has been won.
     */
    boolean isGameWon();

    /**
     * Retrieves the target number or equation for the game.
     * Precondition: None
     * Postcondition: Returns the current target number or equation.
     *
     * @return The target number or equation.
     */
    String getTargetNumber();

    /**
     * Retrieves the remaining attempts left for the user.
     * Precondition: None
```

```
64      * Postcondition: Returns the remaining attempts as an integer.
65      *
66      * @return The remaining attempts.
67      */
68     int getRemainingAttempts();
69
70     /**
71      * Starts a new game by initializing the game state.
72      * Precondition: None
73      * Postcondition: The game state is reset, and a new game begins.
74      */
75     void startNewGame();
76
77     /**
78      * Retrieves the list of guesses made by the user.
79      * Precondition: None
80      * Postcondition: Returns an ArrayList containing the user's guesses.
81      *
82      * @return The list of guesses.
83      */
84     ArrayList<String> getGuessList();
85
86     /**
87      * Retrieves the list of comparison results for each guess.
88      * Precondition: None
89  * Postcondition: Returns an ArrayList containing int arrays representing comparison
    results.
90      *
91      * @return The list of comparison results.
92      */
93     ArrayList<int[]> getCompareList();
94
95     /**
96      * Retrieves the flag indicating whether equations should be displayed.
97      * Precondition: None
98      * Postcondition: Returns the value of the displayEquation flag.
99      *
100     * @return boolean indicating if equations should be displayed.
101     */
102    boolean getDisplayEquation();
103
104    /**
105     * Retrieves the flag indicating whether equations should be verified for
    correctness.
106     * Precondition: None
107     * Postcondition: Returns the value of the verifyEquation flag.
108     *
109     * @return boolean indicating if equations should be verified.
110     */
111    boolean getVerifyEquation();
112
113    /**
114     * Retrieves the flag indicating whether the equality in the target number should be
    random.
115     * Precondition: None
116     * Postcondition: Returns the value of the randomEquality flag.
117     *
118     * @return boolean indicating if the equality should be random.
119     */
120    boolean getRandomEquality();
121
122    /**
123     * Retrieves the classification lists used for tracking character matches.
124     * Precondition: None
125     * Postcondition: Returns the array of ArrayLists representing character
    classifications.
126     *
```

```
127      * @return ArrayList[] containing classification lists.
128      */
129     ArrayList[] getClassList();
130 
131     /**
132      * Sets the flag to verify the correctness of the equation.
133      * Precondition: None
134      * Postcondition: The verifyEquation flag is updated.
135      *
136      * @param verifyEquation The new value for the verifyEquation flag.
137      */
138     void setVerifyEquation(boolean verifyEquation);
139 
140     /**
141      * Sets the flag to display the equation.
142      * Precondition: None
143      * Postcondition: The displayEquation flag is updated.
144      *
145      * @param displayEquation The new value for the displayEquation flag.
146      */
147     void setDisplayEquation(boolean displayEquation);
148 
149     /**
150      * Sets the flag to use random equality in the target number.
151      * Precondition: None
152      * Postcondition: The randomEquality flag is updated.
153      *
154      * @param randomEquality The new value for the randomEquality flag.
155      */
156     void setRandomEquality(boolean randomEquality);
157 
158     /**
159      * Sets the target number or equation for the game.
160      * Precondition: The target should be a String.
161      * Postcondition: The targetNumber is updated.
162      *
163      * @param s The new target number or equation.
164      */
165     void setTarget(String s);
166 }
```

File- D:\CDUT\AOOP\CW\src\Controller\NumberleController.java

```java
package Controller;

import Model.Interface.INumberleModel;
import View.NumberleView;

import java.util.ArrayList;

/**
 * Controller for the Numberle game following the MVC pattern.
 * It mediates the communication between the view and the model.
 * Invariants:
 * - model should not be null after construction.
 * - view can be null initially but should be set before processing input.
 */
public class NumberleController {
    private final INumberleModel model;
    private NumberleView view;

    /**
     * Constructor for Controller.NumberleController.
     * @param model The model of the Numberle game.
     * Precondition: model should not be null.
     * Postcondition: this.model is initialized.
     */
    public NumberleController(INumberleModel model) {
        assert model != null : "Model cannot be null";
    this.model = model; 28
    }

    /**
     * Sets the view for the controller.
     * @param view The view of the Numberle game.
     * Precondition: view should not be null.
     * Postcondition: this.view is set.
     */
    public void setView(NumberleView view) {
        assert view != null : "View cannot be null";
    this.view = view; 39
    }

    /**
     * Processes the user input.
     * @param input The user's guess.
     * Precondition: input should not be null or empty.
     * Postcondition: The view is updated based on the input's validity and game state.
     */
    public void processInput(String input) {
        assert input != null && !input.isEmpty() : "Input cannot be null or empty";
        int statusCode = model.processInput(input);
        if (statusCode > 1)
    view.showInvalidInputMessage(statusCode); 52 }

    /**
     * Checks if the game is over.
     * @return true if the game is over, false otherwise.
     */
    public boolean isGameOver() {
    return model.isGameOver(); 60}

    /**
     * Checks if the game is won.
     * @return true if the game is won, false otherwise.
     */
    public boolean isGameWon() {
        return model.isGameWon();
```

```
68      }
69
70      /**
71       * Gets the target number for the game.
72       * @return The target number.
73       */
74      public String getTargetWord() {
75      return model.getTargetNumber(); 76
        }
77
78      /**
79       * Gets the remaining attempts.
80       * @return The number of remaining attempts.
81       */
82      public int getRemainingAttempts() {
83      return model.getRemainingAttempts(); 84
        }
85
86      /**
87       * Starts a new game.
88       * Postcondition: The game state is reset.
89       */
90      public void startNewGame() {
91      model.startNewGame(); 92
        }
93
94      /**
95       * Gets the value of the "Verify Equation" setting.
96       * @return true if equation verification is enabled, false otherwise.
97       */
98      public boolean getVerifyEquation() {
99          return model.getVerifyEquation();
100     }
101
102     /**
103      * Gets the value of the "Display Equation" setting.
104      * @return true if equation display is enabled, false otherwise.
105      */
106     public boolean getDisplayEquation(){
107         return model.getDisplayEquation();
108     }
109
110     /**
111      * Gets the value of the "Random Equality" setting.
112      * @return true if random equality is enabled, false otherwise.
113      */
114     public boolean getRandomEquality() {
115         return model.getRandomEquality();
116     }
117
118     /**
119      * Sets the value of the "Verify Equation" setting.
120      * @param verifyEquation true to enable equation verification, false otherwise.
121      * Precondition: None.
122      * Postcondition: The model's verifyEquation state is set.
123      */
124     public void setVerifyEquation(boolean verifyEquation) {
125         model.setVerifyEquation(verifyEquation);
126     }
127
128     /**
129      * Sets the value of the "Display Equation" setting.
130      * @param displayEquation true to enable equation display, false otherwise.
131      * Precondition: None.
132      * Postcondition: The model's displayEquation state is set.
133      */
134     public void setDisplayEquation(boolean displayEquation) {
```

```
135         model.setDisplayEquation(displayEquation);
136     }
137 
138     /**
139      * Sets the value of the "Random Equality" setting.
140      * @param randomEquality true to enable random equality, false otherwise.
141      * Precondition: None.
142      * Postcondition: The model's randomEquality state is set.
143      */
144     public void setRandomEquality(boolean randomEquality) {
145         model.setRandomEquality(randomEquality);
146     }
147 
148     /**
149      * Gets the list of user guesses.
150      * @return An ArrayList of Strings representing the user's guesses.
151      * Precondition: None.
152      * Postcondition: Returns the current list of guesses.
153      */
154     public ArrayList<String> getGuessList() {
155         return model.getGuessList();
156     }
157 
158     /**
159      * Gets the list of comparisons for each guess.
160 * @return An ArrayList of int arrays representing the comparison of each guess
    against the target number.
161      * Precondition: None.
162      * Postcondition: Returns the current list of comparisons.
163      */
164     public ArrayList<int[]> getCompareList() {
165         return model.getCompareList();
166     }
167 
168     public ArrayList<Character>[] getClassList() {
169         return model.getClassList();
170     }
171 }
```

File- D:\CDUT\AOOP\CW\src\CustomClass\RoundedButton.java

```java
package CustomClass;
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.geom.RoundRectangle2D;

// Create a custom button class that inherits from JButton
public class RoundedButton extends JButton {
    private Color currentBackgroundColor; // Stores the current background color
    private Color defaultBackgroundColor; // Stores the default background color
    private Color hoverBackgroundColor; // The background color when the mouse is hovering
    private Color pressedBackgroundColor; // The background color when the button is pressed

    public RoundedButton(String label) {
        super(label);
        // Set the button content area not to be filled with the default background color
        setContentAreaFilled(false);
        currentBackgroundColor = defaultBackgroundColor = new Color(220,225,237); // Initialize to the default color
        hoverBackgroundColor = new Color(200, 200, 200); // Initialize to the default hover color
        pressedBackgroundColor = new Color(150, 150, 150); // Initialize to default pressed color

        // Add mouse event listener to change button color
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseEntered(MouseEvent e) {
                currentBackgroundColor = hoverBackgroundColor;
                repaint();
            }

            @Override
            public void mouseExited(MouseEvent e) {
                currentBackgroundColor = defaultBackgroundColor; // Reset to default background color
                repaint();
            }

            @Override
            public void mousePressed(MouseEvent e) {
                currentBackgroundColor = pressedBackgroundColor;
                repaint();
            }

            @Override
            public void mouseReleased(MouseEvent e) {
                currentBackgroundColor = hoverBackgroundColor;
                repaint();
            }
        });
    }

    // Override the paintComponent method to draw the button contents
    @Override
    protected void paintComponent(Graphics g) {
        g.setColor(currentBackgroundColor); // Draw using the stored background color
        g.fillRoundRect(0, 0, getSize().width - 1, getSize().height - 1, 20, 20);
    super.paintComponent(g); 57
    }

    // Override the paintBorder method to draw the button border
    @Override
    protected void paintBorder(Graphics g) {
```

```
62          g.setColor(getForeground());
63      }
64
65      // Override the contains method to change the click area of the button
66      @Override
67      public boolean contains(int x, int y) {
68          // Determine whether the point clicked is inside the rounded rectangle
69          Shape shape = new RoundRectangle2D.Float(0, 0, getWidth(), getHeight(), 20, 20);
70          return shape.contains(x, y);
71      }
72
73      @Override
74      public void setBackground(Color bg)
75          { currentBackgroundColor = bg; // Update background
76          color defaultBackgroundColor = bg;
77          super.setBackground(bg); // Call the setBackground method of the parent class
78          repaint(); // Request to redraw the button
79      }
80
81      // Set the background color when the mouse is hovering
82      public void setHoverBackgroundColor(Color hoverColor)
83          { hoverBackgroundColor = hoverColor;
84      }
85
86      // Set the background color when the button is pressed
87      public void setPressedBackgroundColor(Color pressedColor)
88          { pressedBackgroundColor = pressedColor;
89      }
90  }
```

File- D:\CDUT\AOOP\CW\src\CustomClass\SpacedJTextField.java

```java
package CustomClass;

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

// Create a custom text box class that inherits from JTextField
public class SpacedJTextField extends JTextField {
    private final float letterSpacing;

    // Constructor to specify word spacing
    public SpacedJTextField(float letterSpacing) {
        this.letterSpacing = letterSpacing;
        // Set the background color to white
        this.setBackground(Color.WHITE);
        // Set to opaque to show the background color
    this.setOpaque(true); 17
    }

    // Override the paintComponent method to customize text painting
    @Override
    protected void paintComponent(Graphics g) {
        // Manual background drawing
        g.setColor(getBackground());
        g.fillRect(0, 0, getWidth(), getHeight());

        Graphics2D g2d = (Graphics2D) g.create();
g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.
VALUE_TEXT_ANTIALIAS_ON);
        g2d.setFont(getFont());

        FontMetrics fm = g2d.getFontMetrics();
        String text = getText();
        int x = (int)letterSpacing/2;

        g2d.setColor(getForeground()); // Use the component's foreground color

        // Draws each character according to letterSpacing
        for (char c : text.toCharArray()) {
            String s = String.valueOf(c);
            g2d.drawString(s, x, fm.getAscent());
        x += fm.stringWidth(s) + letterSpacing; 41
        }

        g2d.dispose();
    }
}

```

File- D:\CDUT\AOOP\CW\src\CustomClass\EquationGenerator.java

```java
package CustomClass;

import java.util.Random;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * The EquationGenerator class is responsible for generating random mathematical equations
 * that conform to specific validity criteria. It maintains the best equation generated so far.
 *
 * @invariant bestEquation != null ==> isValidEquation(bestEquation); // If bestEquation is not null, it must be a valid equation according to the game's rules.
 */
public class EquationGenerator {
    private final Random random = new Random();
    private String bestEquation;

    /**
     * @ensures // The postcondition ensures that a valid equation according to the game's rules is stored in bestEquation.
     *   (\exists String validEquation; isValidEquation(validEquation); bestEquation.equals(validEquation));
     * @assignable bestEquation; // The assignable clause specifies that bestEquation may be assigned a value during the execution of the method.
     * @return void
     */
    public void generateEquation() {
        int a, b, result;
        char operator;
        String equation;
        do {
            // Assert that the random object is not null.
            assert random != null : "Random object must not be null.";

            // Generate random numbers and an operator to form an equation.
            if (new Random().nextInt(10) <= 1) {
                a = random.nextInt(100); // 0-99
                b = random.nextInt(100); // 0-99
            } else {
                a = random.nextInt(15); // 0-14
                b = random.nextInt(15); // 0-14
            }
            operator = randomOperator();
            // Ensure division by zero does not occur.
            while (b == 0 && operator == '/')
                b = random.nextInt(50);
            result = calculateResult(a, b, operator);
            equation = a + "" + operator + "" + b + "=" + result;
            bestEquation = equation;
            // Assert that the equation is not null or empty.
            assert equation != null && !equation.isEmpty() : "Equation must not be null or empty.";
    } while ((operator == '/' && (a % b != 0)) || !isValidEquation(equation)); 49
    }

    /**
     * @requires generateEquationCalled; // The generateEquation method must have been called first.
     * @ensures \result != null; // The postcondition ensures that the result is not null, indicating an equation has been generated.
     * @return The best generated equation.
     */
    public String getEquation(){
        // Assert that the bestEquation is not null or empty.
        assert bestEquation != null && !bestEquation.isEmpty() : "Best equation must not
```

```
be null or empty.";
        return bestEquation;
    }

    /**
     * @ensures (\result == '+' || \result == '-' || \result == '*' || \result == '/'
); // The postcondition ensures that the result is one of the four basic operators.
     * @return A random operator, one of '+', '-', '*', or '/'.
     */
    private char randomOperator() {
        char[] operators = {'+', '-', '*', '/'};
        char op = operators[new Random().nextInt(operators.length)];
        // Assert that the operator is valid.
assert new String(operators).contains(String.valueOf(op)) : "Operator must be one of
the valid operators.";
    return op; 72
    }

    /**
     * @ensures \result == '+' || \result == '-'; // The postcondition ensures that the
result is either '+' or '-'.
     * @return A random operator, either '+' or '-'.
     */
    private char randomOperator2() {
        char[] operators = {'+', '-'};
        char op = operators[new Random().nextInt(operators.length)];
        // Assert that the operator is valid.
        assert new String(operators).contains(String.valueOf(op)) : "Operator must be
one of the valid operators.";
    return op; 84
    }

    /**
     * @requires (operator == '+' || operator == '-' || operator == '*' || operator ==
'/'); // The operator must be one of '+', '-', '*', '/'.
     * @ensures // The postcondition ensures that the opposite operator is returned.
     *   ((operator == '+') ==> (\result == '-')) &&
     *   ((operator == '-') ==> (\result == '+')) &&
     *   ((operator == '*') ==> (\result == '/')) &&
     *   ((operator == '/') ==> (\result == '*'));
     * @param operator The operator for which to find the opposite.
     * @return The opposite operator.
     */
    private char getOppositeOperator(char operator) {
        char opposite = switch (operator) {
            case '+' -> '-';
            case '-' -> '+';
            case '*' -> '/';
            case '/' -> '*';
            default -> '\0'; // Invalid operator
        };
        // Assert that the opposite operator is valid.
        assert opposite != '\0' : "Opposite operator must be valid.";
        return opposite;
    }

    /**
     * @requires (operator != '/' || b != 0) && (operator == '+' || operator == '-' ||
operator == '*' || operator == '/'); // The operator must not be division if b is zero,
and it must be a valid operator.
     * @ensures // The postcondition ensures that the result is correct according to the
 operation.
     *   (operator == '+' ==> \result == a + b) &&
     *   (operator == '-' ==> \result == a - b) &&
     *   (operator == '*' ==> \result == a * b) &&
     *   (operator == '/' ==> \result == a / b);
     * @param a The first integer.
```

```
117      * @param b The second integer, which must not be zero if the operator is division.
118 * @param operator The operator to apply, which must be one of the valid
    mathematical operators: '+', '-', '*', or '/'.
119      * @return The result of the operation.
120      */
121     private int calculateResult(int a, int b, char operator) {
122         // Assert that the operator is valid before calculation.
123         assert "+-*/".indexOf(operator) != -1 : "Operator must be one of the valid
    operators.";
124         int result = switch (operator) {
125             case '+' -> a + b;
126             case '-' -> a - b;
127             case '*' -> a * b;
128             case '/' -> b != 0 ? a / b : 0; // Prevent division by zero
129         default -> throw new IllegalArgumentException("Invalid operator"); 130
            };
131         // Assert that the result is a valid integer.
132         assert result == (int)result : "Result must be a valid integer.";
133         return result;
134     }
135 
136     /**
137      * @requires equation != null && equation.length() > 0; // The equation must be a
    non-null and non-empty string.
138      * @ensures \result == true <==> // The result is true if and only if the equation
    is valid.
139      *           (equation.matches("[0-9+-/*()]+") && checkEquationFormat(equation
    )); // The equation must match a valid mathematical expression format.
140      * @param equation The equation to validate.
141      * @return True if the equation is valid, false otherwise.
142      */
143     private boolean isValidEquation(String equation) {
144         // Assert that the equation is not null or empty.
145         assert equation != null && !equation.isEmpty() : "Equation must not be null or
    empty.";
146         int length = equation.length();
147         if (length == 7) {
148             return true;
149         } else if (length == 6) {
150             return isValidModifiedEquation(equation); // Needs to be regenerated
151         } else if (length == 5) {
152             // Take a random 0 or 1
153             if (new Random().nextInt(2) == 0) {
154                 // The result is decomposed so that the formula forms an equation with a
     symbol b=c symbol d
155                 return isValidDecomposedEquation(equation, -1, '0');
156             } else {
157                 // Look for two numbers that satisfy the form of a symbol b symbol c=d
158                 return isValidAdditionEquation(equation, -1, '0');
159             }
160         } else if (length > 7) {
161             // Replace random symbols with + or -
162             return isValidModifiedEquation(equation);
163         }
164         return false; // If still not valid, needs to be regenerated
165     }
166 
167     /**
168      * @requires equation != null && equation.length() > 0; // The equation must be a
    non-null and non-empty string.
169      * @ensures \result == true <==> // The result is true if and only if a valid
    decomposed equation is found.
170      *           (\exists String validEquation; validEquation.length() >= 0 &&
    validEquation.length() <= equation.length();
171      *           validEquation.matches("[0-9+-/*()]+") &&
    checkDecomposedEquation( validEquation)); // There exists a string that is a valid
    decomposed equation of appropriate length.
```

```
172      * @param equation The equation to be decomposed.
173      * @param num The number to be used as an operand in the decomposition, if known.
174      * @param operator The operator to be used in the decomposition, if known.
175      * @return True if a valid decomposed equation is found, false otherwise.
176      */
177     private boolean isValidDecomposedEquation(String equation, int num, char operator) {
178         // Assert that the input equation is not null or empty.
179         assert equation != null && !equation.isEmpty() : "Input equation must not be
    null or empty.";
180         // Assert that the input number is within the valid range if provided.
181         assert num >= -1 && num <= 9 : "Input number must be between -1 and 9.";
182         // Assert that the input operator is one of the valid operators.
183         assert "+-*/".indexOf(operator) != -1 : "Input operator must be one of
     '+', '-', '*', '/'.";
184
185         // Split the equation into left and right sides.
186         String[] parts = equation.split("=");
187         // Assert that the equation splits into exactly two parts.
188         assert parts.length == 2 : "Equation must split into exactly two parts.";
189         if (parts.length != 2) {
190             return false; // Incorrect equation format.
191         }
192         String leftSide = parts[0].trim();
193         String rightSide = parts[1].trim();
194
195         // Initialize operands and operator.
196         int a = num;
197         char operator2 = operator;
198         // If the operand is not known, generate a random number and operator.
199         if (a < 0) {
200             a = random.nextInt(10);
201             operator2 = randomOperator();
202         }
203         int result = Integer.parseInt(rightSide);
204
205         // Ensure valid operands for multiplication and division.
206         while (a == 0 && (operator2 == '/' || operator2 == '*')) {
207             a = random.nextInt(10);
208             operator2 = randomOperator();
209         }
210         // Calculate the result of the operation.
211         int result2 = calculateResult(result, a, operator2);
212
213         // Check if the operation is valid and create a new equation.
214         if (operator2 != '/' || result % a == 0) {
215             String newEquation = leftSide + "=" + result2 + "" +
    getOppositeOperator( operator2) + "" + a;
216             // If the new equation is too long, attempt to decompose further.
217             if (newEquation.length() > 7)
218                 return isValidDecomposedEquation(equation, a-1, operator2);
219             // Set the best equation found.
220             bestEquation = newEquation;
221             // Assert that the best equation is not null or empty.
222             assert bestEquation != null && !bestEquation.isEmpty() : "Best equation must
     not be null or empty.";
223             return true;
224         } else {
225             // If the result is not an integer, try with a different operand.
226             if (a < 9) {
227                 return isValidDecomposedEquation(equation, a+1, operator2);
228             }
229             // If no valid decomposition is found, indicate the need to regenerate the
    equation.
230             return false;
231         }
232     }
233
```

```
234     /**
235      * @requires equation != null && equation.length() > 0 && num < 9; // The equation must be a non-null, non-empty string and num must be less than 9.
236      * @ensures \result == true <==> // The result is true if and only if a valid addition equation is found.
237      *           (\exists String validEquation; validEquation.length() <= equation.length();
238      *           validEquation.contains("+") && validEquation.matches("[0-9+]+") &&
239      *           validEquation.length() == 7); // There exists a string that is a valid addition equation of length 7.
240      * @param equation The equation to be modified.
241      * @param num The number to be used as an operand in the modification, if known.
242      * @param operator The operator to be used in the modification, if known.
243      * @return True if a valid addition equation is found, false otherwise.
244      */
245     private boolean isValidAdditionEquation(String equation, int num, char operator) {
246         // Assert that the input equation is not null or empty.
247         assert equation != null && !equation.isEmpty() : "Input equation must not be null or empty.";
248         // Assert that the input number is within the valid range if provided.
249         assert num >= -1 && num <= 9 : "Input number must be between -1 and 9 inclusive.";
250         // Assert that the input operator is one of the valid operators.
251         assert "+-".indexOf(operator) != -1 : "Input operator must be one of '+', '-'.";
252 
253         // Split the equation into left and right sides.
254         String[] parts = equation.split("=");
255         // Assert that the equation splits into exactly two parts.
256         assert parts.length == 2 : "Equation must split into exactly two parts.";
257 
258         if (parts.length != 2) {
259             return false; // Incorrect equation format.
260         }
261         if (num >= 9)
262             return false; // Prevent infinite recursion.
263         String leftSide = parts[0].trim();
264         String rightSide = parts[1].trim();
265 
266         // Initialize operands and operator.
267         int a = num;
268         char operator2 = operator;
269         int result2;
270         String newEquation;
271         int result = Integer.parseInt(rightSide);
272         // Calculate the result of the operation.
273         if (new Random().nextInt(2) == 0) {
274             if (a < 0) {
275                 a = random.nextInt(10);
276                 operator2 = randomOperator2();
277             }
278             result2 = calculateResult(result, a, operator2);
279             newEquation = leftSide + "" + operator2 + "" + a + "=" + result2;
280         } else {
281             if (a < 0) {
282                 a = random.nextInt(10);
283                 operator2 = '+';
284             }
285             result2 = calculateResult(a, result, operator2);
286             newEquation = a + "" + operator2 + "" + leftSide + "=" + result2;
287         }
288         // If the new equation is too long, attempt to modify further.
289         if (newEquation.length() > 7)
290             return isValidAdditionEquation(equation, a-1, '+');
291         else {
292             // Set the best equation found.
293             bestEquation = newEquation;
294             // Assert that the best equation is not null or empty.
```

```
295             assert bestEquation != null && !bestEquation.isEmpty() : "Best equation must
 not be null or empty.";
296             return true;
297         }
298     }
299 
300     /**
301      * @requires equation != null && equation.length() > 0; // The equation must be a
non-null and non-empty string.
302      * @ensures \result == true <==> (\exists int i; 0 <= i && i < equation.length();
303      *           (\old(equation.charAt(i)) == '+' || \old(equation.charAt(i)) == '-') &&
304      *           equation.length() == 7); // The result is true if and only if the
equation contains '+' or '-' and its length is 7 after modification.
305      * @param equation The equation to be modified.
306 * @return True if the equation is successfully modified to a valid form, false
otherwise.
307      */
308     private boolean isValidModifiedEquation(String equation) {
309         // Assert that the input equation is not null or empty.
310         assert equation != null && !equation.isEmpty() : "Input equation must not be
null or empty.";
311 
312         // Use regex to parse the equation and extract operands and the operator.
313         Pattern pattern = Pattern.compile("(\\d+)([+\\-*/])(\\d+)=(-?\\d+)");
314         Matcher matcher = pattern.matcher(equation);
315         int a, b, result;
316         char operator2;
317         // Attempt to find a match for the equation pattern.
318         if (matcher.find()) {
319             a = Integer.parseInt(matcher.group(1)); // Extract number a
320             b = Integer.parseInt(matcher.group(3)); // Extract number b
321         } else {
322             // If the equation does not match the pattern, return false.
323             return false;
324         }
325         // Randomly choose an operator from '+' or '-'.
326         operator2 = randomOperator2();
327         // Calculate the result with the new operator.
328         result = calculateResult(a, b, operator2);
329         // Form a new equation with the calculated result.
330         String newEquation = a + "" + operator2 + "" + b + "=" + result;
331 
332         // Check if the new equation has a valid length.
333         if (newEquation.length() == 7) {
334             bestEquation = newEquation;
335             return true;
336         } else {
337             // Try the opposite operator if the length is not valid.
338             result = calculateResult(a, b, getOppositeOperator(operator2));
339             newEquation = a + "" + getOppositeOperator(operator2) + "" + b + "=" +
result;
340             if (newEquation.length() == 7) {
341                 bestEquation = newEquation;
342                 return true;
343             } else {
344                 // Try swapping the operands and re-calculating.
345                 result = calculateResult(b, a, operator2);
346                 newEquation = b + "" + operator2 + "" + a + "=" + result;
347                 if (newEquation.length() == 7) {
348                     bestEquation = newEquation;
349                     return true;
350                 } else {
351                     // Try with the opposite operator after swapping.
352                     result = calculateResult(b, a, getOppositeOperator(operator2));
353                     newEquation = b + "" + getOppositeOperator(operator2) + "" + a + "="
 + result;
354                     if (newEquation.length() == 7) {
```

```
355                             bestEquation = newEquation;
356                             return true;
357                         } else {
358                             // If the length is 5, validate the equation.
359                             if (newEquation.length() == 5)
360                                 return isValidEquation(newEquation);
361                             else {
362                                 // If no valid modification is found, return false.
363                                 return false;
364                             }
365                         }
366                     }
367                 }
368             }
369         }
370 }
371
```

File- D:\CDUT\AOOP\CW\src\CustomClass\RoundedBorderLabel.java

```java
package CustomClass;
import javax.swing.*;
import java.awt.*;

// Create a custom class for JLabel with rounded borders
public class RoundedBorderLabel extends JLabel {
    private final int radius;

    public RoundedBorderLabel(int radius) {
        this.radius = radius;
    setOpaque(false); 12
    }

    // Override the paintComponent method to draw rounded corners
    @Override
    protected void paintComponent(Graphics g) {
        Dimension arcs = new Dimension(radius, radius);
        int width = getWidth();
        int height = getHeight();
        Graphics2D graphics = (Graphics2D) g;
        graphics.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.
VALUE_ANTIALIAS_ON);

        // Draw rounded rectangles
        graphics.setColor(getBackground());
        graphics.fillRoundRect(0, 0, width-1, height-1, arcs.width, arcs.height);
        graphics.setColor(getForeground());
        // Center text
    super.paintComponent(g); 29
    }

    // Override the paintBorder method to draw a border
    @Override
    protected void paintBorder(Graphics g) {
    g.setColor(Color.BLACK); 35
    }

    @Override
    public Dimension getPreferredSize() {
    return new Dimension(50, 50); 40
    }

}
```

File- D:\CDUT\AOOP\CW\test\Model\NumberleModelTest.java

```
1 package Model;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import java.util.ArrayList;
8
9 import static org.junit.Assert.*;
10
11 /**
12  * JML style comments for NumberleModelTest class.
13  * @invariant model != null -> model is always initialized before tests
14  */
15 public class NumberleModelTest {
16     private NumberleModel model;
17
18     /**
19      * Sets up the test fixture.
20      * Called before every test case method.
21      * @pre. none
22      * @post. model != null -> ensures that the model is not null after initialization
23      */
24     @Before
25     public void setUp() {
26         model = new NumberleModel();
27     model.initialize(); 28
       }
29
30     /**
31      * Tests the entire game flow when verification checks are enabled,
32      * random equations are disabled, and the display equation is shown.
33      * @requires model != null && model.getTarget().equals("2*3-6=0")
34      *           && model.isVerifyEquation() && model.isDisplayEquation()
35      *           && !model.isRandomEquality()
36      *           // Requires that the model is initialized with specific settings
37      * @ensures (model.isGameWon() && model.isGameOver()) == true
38      *          // Ensures that the game is won and over at the end of the test
39      */
40     @Test
41     public void testGameWinFlow() {
42         // Test enabled check, not enabled random equation when the entire game flow
43         model.setTarget("2*3-6=0");
44         model.setVerifyEquation(true);
45         model.setDisplayEquation(true);
46         model.setRandomEquality(false);
47
48         /**
49          * @pre input != null
50          *           // Requires that the input string is not null
51 * @post \result == (\exists int[] compareList;
   model.getCompareList().contains( compareList);
52 *        \forall int i; 0 <= i && i < compareList.length; compareList[i] ==
   compared[i])
53 *        // Ensures that the result is an array that matches the expected comparison
   array
54          */
55         // At first guess, the length is correct but not an exact match
56         model.processInput("5+15=20");
57         int[] compare1 = model.getCompareList().get(0);
58         int[] compared = new int[]{0, 0, 0, 0, 2, 2, 1};
59         for (int i = 0; i < compare1.length; i ++){
60             assertEquals("Partial matches should return a specific array", compared[i],
   compare1[i]);
61         }
62
63         // Second guess, wrong length
```

```
64          int result2 = model.processInput("10+10");
65          int sizeOfCompareList= model.getCompareList().size();
66          assertEquals("Formatting errors should return 2", 2, result2);
67          assertEquals("No equality comparison is made, so the list does not grow", 1,
    sizeOfCompareList);
68
69          // Third guess, the length is correct but not the equation
70          int result3 = model.processInput("777=777");
71          sizeOfCompareList= model.getCompareList().size();
72          assertEquals("Not the equation should return 3", 3, result3);
73          assertEquals("No equality comparison is made, so the list does not grow", 1,
    sizeOfCompareList);
74
75          // Third, fourth and fifth guesses, the length is correct and the format is
    correct but not equal
76          String[] invalidEquations = {"10+5=14", "2*2-2=5", "100/1=0"};
77          for (String equation : invalidEquations) {
78              int result = model.processInput(equation);
79          assertEquals("Invalid equations should return 4", 4, result); 80
            }
81          assertEquals("No equality comparison is made, so the list does not grow", 1,
    sizeOfCompareList);
82
83          // Sixth guess. Right guess
84          int result4 = model.processInput("2*3-6=0");
85          sizeOfCompareList= model.getCompareList().size();
86          assertEquals("Perfect matches should return 1", 1, result4);
87          assertEquals("After two comparisons, the list length should be 2", 2,
    sizeOfCompareList);
88          assertTrue("The game should be won", model.isGameWon());
89      assertTrue("The game should be over", model.isGameOver()); 90
        }
91
92      /**
93       * Tests the game lose flow by processing inputs that do not match the target
    equation,
94       * and verifying the classification of characters.
95       * @requires model != null && model.getTarget().equals("12+3=15")
96       *           && !model.isVerifyEquation() && !model.isDisplayEquation()
97       *           && !model.isRandomEquality()
98       *           // Requires that the model is initialized with specific settings
99       * @ensures !model.isGameWon() && model.isGameOver()
100      *          // Ensures that the game is lost and over at the end of the test
101      */
102     @Test
103     public void testGameLoseFlow() {
104         // Test character classification
105         model.setTarget("12+3=15");
106         model.setVerifyEquation(false);
107         model.setDisplayEquation(false);
108         model.setRandomEquality(false);
109
110         /**
111          * @pre input != null && input.equals("11+5=16")
112          *           // Requires that the input string is not null and equals "11+5=16"
113 * @post \exists ArrayList<Character>[] classList; model.getClassList() ==
    classList
114          *          && classList[0].contains('6') && classList[1].contains('1')
115          *          && classList[2].contains('5') && classList[3].contains('2')
116 *        // Ensures that the classification list is not null and contains specific
    characters
117          */
118         // First guess, part match, part wrong
119         model.processInput("11+5=16");
120         ArrayList<Character>[] classList = model.getClassList();
121         assertNotNull("The category list should not be null", classList);
122         assertTrue("Non-existent character should contain '6'", classList[0].contains('6
```

```
122 '));
123         assertTrue("Hit character should contain '1'", classList[1].contains('1'));
124         assertTrue("Characters present but missed should contain '5'", classList[2].
    contains('5'));
125         assertTrue("Unguessed characters should contain '2'", classList[3].contains('2'
    ));
126 
127 // Second guess, parts that exceed the length of the target equation should not be
    compared and classified
128         model.processInput("123456789");
129         classList = model.getClassList();
130         assertNotNull("The category list should not be null", classList);
131         assertTrue("Non-existent character should contain '4'", classList[0].contains('4
    '));
132         assertTrue("Hit character should contain '2'", classList[1].contains('2'));
133         assertTrue("Characters present but missed should contain '3'", classList[2].
    contains('3'));
134         assertTrue("Unguessed characters should contain '8'", classList[3].contains('8'
    ));
135         assertTrue("Unguessed characters should contain '9'", classList[3].contains('9'
    ));
136 
137         // Test the close but incorrect equation 4 times until the game is lost
138         String[] invalidEquations = new String[]{"12+3=14", "3*5=15", "1234=15", "30/2=
    15"};
139         for (String equation : invalidEquations) {
140             model.processInput(equation);
141         }
142         int sizeOfCompareList= model.getCompareList().size();
143         assertEquals("Having guessed 6 times resulted in losing the game, the list
    length should be 6", 6, sizeOfCompareList);
144         assertFalse("The game should be lost", model.isGameWon());
145         assertTrue("The game should be over", model.isGameOver());
146 
147     }
148 
149     /**
150      * Tests the random equation generation to ensure that each new game has a unique
    target number.
151      * @requires model != null && model.isRandomEquality()
152      *           // Requires that the model is initialized and random equality is set to
     true
153      * @ensures \forall String equation; model.getTargetNumber() != equation
154      *          // Ensures that each generated target number is unique
155      */
156     @Test
157     public void testRandomEquationGeneration() {
158         // Test random equation generation
159         model.setRandomEquality(true);
160         String equation1 = model.getTargetNumber();
161         assertNotNull("Random equations should not be null", equation1);
162         model.startNewGame();
163         String equation2 = model.getTargetNumber();
164         assertNotNull("Random equations should not be null", equation2);
165         assertNotEquals("The random equations generated continuously should be different
    ", equation1, equation2);
166         model.setRandomEquality(false);
167         model.startNewGame();
168         String equation3 = model.getTargetNumber();
169         assertNotNull("Equations should not be null", equation3);
170         assertEquals("The non-random equations generated continuously should be the same
    ", equation2, equation3);
171     }
172 
173     /**
174      * Tests the processInput method to ensure it returns the correct error code for
    invalid input format.
```

```
175      * @requires model != null && model.isVerifyEquation()
176      *           // Requires that the model is initialized and equation verification is
    set to true
177      * @ensures model.processInput("abcdefg") == 3
178      *          // Ensures that the method returns 3 for invalid input format
179      */
180     @Test
181     public void testProcessInvalidInputFormat() {
182         // The test handles invalid input formats
183         model.setVerifyEquation(true);
184         int result = model.processInput("abcdefg");
185         assertEquals("Invalid input format, should return 3", 3, result);
186     }
187 
188     /**
189      * Tests the setFlags methods to ensure that the boolean flags are set correctly.
190      * @requires model != null
191      *           // Requires that the model is initialized
192      * @ensures model.getVerifyEquation() == true && model.getDisplayEquation() == true
193      *          && model.getRandomEquality() == true
194      *          // Ensures that the flags are set to true after calling the set methods
195      */
196     @Test
197     public void testSetFlags() {
198         // Test setting flag
199         assertFalse("verifyEquation should be false before setVerifyEquation is true",
    model.getVerifyEquation());
200         model.setVerifyEquation(true);
201         assertTrue("verifyEquation should be true after setVerifyEquation is true",
    model.getVerifyEquation());
202 
203         assertFalse("displayEquation should be false before setDisplayEquation is true"
    , model.getDisplayEquation());
204         model.setDisplayEquation(true);
205 assertTrue("displayEquation should be true after setDisplayEquation is true",
    model.getDisplayEquation());
206 
207 assertFalse("randomEquality should be false before setRandomEquality is true",
    model.getRandomEquality());
208         model.setRandomEquality(true);
209 assertTrue("randomEquality should be true after setRandomEquality is true",
    model.getRandomEquality());
210     }
211 }
```