SEN4013 SOFTWARE VERIFICATION AND VALIDATION

Project group number: 64

Student Name	ID	Responsibilities
Umniyah Sameer Haitham Abbood	1900812	1- **Introduction** (Model checking, Testing embedded systems) 2- **Objectives and scope of the software** 3- **Functional Testing** (Equivalence Class, Test cases table, 3 Finite state machines, 3 Case effect graphs) 4- **Recommendations**
Mohanad El Masri	1900129	1- **Introduction** (Load testing, Functional testing) 2- **Functional Testing** (2 Combinatorial testing (Pairwise approach)) 3- **Structural Testing** (4 Control Flow Graphs with their test cases) 4- **UML**

1. Introduction

Verification and validation are two essential processes in software development that aim to ensure the quality and accuracy of a product. Verification focuses on evaluating whether the product meets the specified requirements, while validation assesses whether the product satisfies the user's needs and expectations. Both processes play a critical role in ensuring that the software meets the intended objectives and provides the desired functionality to users.

Several automated technologies are used for verification and validation; we will discuss four.

1. **Model checking:** this is a formal method of software testing that involves using mathematical logic to examine the behavior of a software model and verify that it meets its desired specifications. The desired system behaviors are defined and then checked against the system model in model checking. If the model satisfies all of the properties, it is considered to be a valid implementation of the system.

- **Usage**:
    - It is commonly used in the domain of computer-aided verification, where it can be used to automatically verify the correctness of a software system.
    - It can be used to validate partial specifications and provide important information about correctness even though the system has not been fully specified
    - It is applicable throughout the process software life cycle, from specification to final implementation.
- **Limitation**: model checking can only be applied to systems with finite state spaces, meaning that it is not suitable for systems with an unbounded number of possible states. Also, model checking can be computationally intensive, so it may not be practical for large or complex systems [1].

Some of the model-checking automated tools

1.1. **Spin:** is an open-source software verification tool that was created in the Computing Science Research Center of Bell Labs in 1980). It is frequently regarded as the most widely used formal verification tool and can be used in these operating systems: Microsoft Windows; Linux; Mac OS X [2].

1.2. **SVM**: Ken McMillian created the first model checker that used symbolic verification [1]. There are currently several varieties of tools available from CMU, such as (NFLSAT), in which Boolean satisfiability (SAT) solvers are heavily utilized in hardware and software verification tools for testing the satisfiability of Boolean circuits [3].

2. **Testing embedded systems** is a type of software testing that focuses on testing the functionality of software running on embedded devices, such as those found in household appliances, medical equipment, and vehicles. The main goal of testing embedded systems is to ensure that the software running on these devices is reliable, efficient, and meets the requirements of the end user. It often involves testing the software in the context of the hardware it will be running on. This means that testers must have a thorough understanding of the hardware components of the embedded device and how they interact with the software which could be a limitation as human evolution is not always accurate. This technique differs from other validation techniques like model-checking, static analysis, and debugging since it dynamically executes the product in a realistic (and yet controllable) environment with actual concrete input data, comparing the actual and expected behavior [1].

- **Usage**:
    - It is typically used in industries where embedded devices play a critical role, such as automotive, aerospace, and healthcare. In these industries, it is important to ensure that the software running on these devices is reliable and operates as intended, as a failure of the software could have serious consequences.
- **Limitation:** of testing embedded systems is that it can be difficult to simulate the real-world conditions in which the software will be running. This means that it can be challenging to test for all possible scenarios and ensure that the software will operate correctly in all situations. Additionally, testing embedded systems can be time-consuming and costly, particularly if the device being tested has many hardware components.

Some of the testing embedded systems' automated tools

2.1. **Simulink Reactis Tester:** This automatically builds test suites from Simulink / Stateflow diagrams. Each test is made up of a series of stimulus/response pairings, with each stimulus assigning an input value to each in-port in the model and each response recording an output value for each out-port. The test suites are produced based on the specification's criteria [1].

2.2. **Conformiq Test Generator:** This automatically builds test cases from UML state chart models. Model simulations can be used to produce batches of test cases that can then be executed. However, the models can be dynamically interpreted to allow for on-the-fly testing [1].

3. **Load testing** is a type of performance testing that involves simulating real-world usage of a software application or system to measure its performance and identify any potential bottlenecks or scalability issues. It is typically used to verify that a system can handle expected levels of usage and user traffic, and to ensure that it meets performance and availability requirements. Load testing can be performed using specialized tools that simulate various load conditions and measure system performance. [4].

- **Usage**: Load testing is typically applied to a software application or system to determine how it performs under a specific load or user activity and it is typically performed at various stages of the development process, including during design, development, and testing, as well as after deployment to ensure that the system continues to perform well under expected load conditions [4]
  Some common scenarios where load testing may be applied include:

- Verifying that a system can handle the expected number of users or transactions
- Identifying performance bottlenecks and scalability issues
- Testing the performance of a system under high load conditions, such as during peak usage periods
- Validating the system's ability to recover from failures or errors
- Ensuring that the system meets performance and availability requirements
- Identifying the maximum capacity of a system and identifying the point at which it may fail or degrade

- **Limitations:**
  - Time-consuming and resource-intensive
  - May not accurately reflect real-world usage patterns
  - May not identify all potential performance issues
  - May not be effective for systems with highly variable or unpredictable load patterns
  - May not fully replicate the complexity and interactions of a real-world environment
  - Tools may have limitations or biases that affect test results

Some of the load testing automated tools

3.1 Apache JMeter: this is an open-source load testing tool developed by the Apache Software Foundation (ASF) that can be used to simulate various load conditions and measure system performance. It is useful for load testing, regression testing, and testing client/server systems. It has a range of features, a GUI, and provides accurate results. However, it requires setup and performs offline reporting. [4]

3.2 LoadRunner: this is a load testing tool developed by Hewlett Packard Enterprise (HPE) that is used to identify bottlenecks and detect and prevent software performance issues. It is particularly useful for testing the performance of web, mobile, and other types of applications, as well as for evaluating network performance. LoadRunner can simulate and measure system performance for multiple users simultaneously but may have challenges with installation or configuration across firewalls. [4]

4. **Functional testing:** this is a type of testing that verifies that a software application or system performs the functions it is designed to perform. It involves testing individual functions or features of the system to ensure that they work as intended and meet specified requirements. This can include testing user interfaces, database interactions, system integrations, and other elements of the system. Functional testing is often automated and is typically used in conjunction with other types of testing to ensure the overall reliability, security, and usability of the system. [4]

- **Usage:** Functional testing is typically applied at various stages of the development process to verify the correct functioning of a software application or system. [4]
  Some common scenarios where functional testing may be applied include:
  - Verifying system behavior when given specific inputs
  - Testing system interfaces and integrations
  - Ensuring the system meets functional and performance requirements
  - Identifying and fixing defects or issues in system functionality
  - Validating system recovery from failures or errors
  - Performing regression testing
  - Functional testing can be performed manually or automated using functional testing tools and is often used in conjunction with other types of testing to ensure system reliability, security, and usability
- **Limitations:**
  - Only tests system functions and features, not other issues such as performance or scalability
  - Relies on predefined test cases and may not fully replicate real-world scenarios
  - Limited in testing system behavior under extreme or unexpected conditions
  - May not fully test system integration with other systems or external dependencies
  - May not identify all defects or issues in the system
  - Testing tools may have limitations or biases that affect test results.

Some of the Functional testing automated tools

4.1 Selenium: is an open-source functional testing tool that is used to test web applications and automate browser actions. It allows testers to use simple scripts to run tests directly in the browser and can be used on different platforms. Selenium also has features such as the ability to edit, record, and debug tests, making it a user-friendly tool to set up and use. [4]

4.2 FitNesse: this is a functional testing tool that is used to support the acceptance testing of software applications. It allows users to define and document acceptance tests in a simple, easy-to-read format, and then run those tests automatically to verify that the system under test meets the specified requirements. FitNesse is well-suited for agile development environments and can be used to test a variety of applications. It is open-source and written in Java. [4]

2. Objectives and scope of the software.

In this project, we will examine the code that I have developed with my colleagues (sukran oyku Erdik- Jawad Ntefeh), in the CMP2004. Our software is about Wordle game in which players are presented with a grid of letters and must create words using the letters in the grid. The objective of the game is to use 5 letters to create valid words and earn points, they have 5 chances to guess the word.

There are 3 cases while playing:

1. If the letter that the user entered is in the word and the correct place, the background of the grid to that specific letter will be green.
2. If the letter that the user entered is in the word but not in the correct place, the background of the grid to that specific letter will be yellow
3. If the letter that the user entered is not in the word, the background of the grid to that specific letter will be gray.

There are 4 modes of the game:

1. Only 1 player uses the keyboard to enter the word (Figure 1)
2. Only 1 player drops and drags the letter to enter the word (figure 2)
3. 2 players using the keyboard to enter the word (figure 3)
4. 2 play drop and drags the letter to enter the word (figure 4)

The player with the highest score at the end of the game wins. In addition to being a fun and engaging way to improve vocabulary and word-building skills, Wordle can also help improve spelling and concentration.

There are 11 java files in the project

1. **Source.java:** it has the main method, and the first thing starts to run
2. **WelcomePage**.**java**: this class extends JFrame and implements ActionListene, to give the user the choice to choose whether they want to play using the keyboard or drag and drop, based on their choice, the corresponding class will be called. (Figure 5)
3. **Player**.**java:** this class extends JFrame and implements ActionListener, to give the user the choice to choose whether the user wants to play alone or as 2 players based on their choice, the corresponding class will be called. (Figure 6)
4. **PlayerParent.java:** this class has 2 attributes which are playerName and playerGuess to keep track of the user's information
5. **WordBase.java:** this class reads the words.txt file and randomly generates a word from the list to be the target word
6. **Keyboard.java:** this class is called when the user wants to play the game using the keyboard, it has 2 constructors, the first one for 1 player, and the second is for 2 players, it has a check object that checks if the player entered the correct word, keyboard class has implemented MouseListener to check if the user entered only 5

characters, otherwise it gives an error as a pop-up message to inform the user that they can only enter 5 characters, afterword it checks if the word that had been entered by the user in our words.txt. (Figure 2) and (Figure 3)

7. **DragAndDrop.java:** this class is remarkably similar to **Keyboard.java,** the only difference here the user will not enter the word using the keyboard, instead it will drag the letter picture and drop it to the grid, the drag and drop feature was implemented using this class **ExportHandler** which extends **TransferHandle**. (Figure 2) and (Figure 4)
8. **AnimationPanel.java:** this class extends JPanel and implements ActionListener, it displays a 2D animation when the user wins and it is linked to the Statistics window. (Figure 7)
9. **AnimationFram.java:** this class only creates the **AnimationPanel** frame
10. **Statistics.java:** this class holds and displays the user's playing time, the user score, highest score, play count, and win rate (%). (Figure 8)
11. **Checks.java:** this class checks if the user entered the target word correctly and calls the corresponding class to show the animation if the user wins, and if the user did not get the word, then the Statistics window will show up

There are 4 text files in the project

1. **Words.txt**: it has 1000 words, which have been used randomly to generate a word from the list to be the target word.
2. **HighestScore.txt:** it keeps the highest score, and it gets overwritten in the **Statistics** class.
3. **PlayCount.txt:** it keeps the number of how many the user who played this game; it increases by 1, and it gets read and written in **Statistics** class.
4. **WinCount.txt:** it keeps the number of how many the user won the game; it increases by 1 each time the user gets the target word correctly, and it gets read and written in **Statistics** class.

3. Test cases and test results.

3.1. Functional Testing

- **Equivalence Class**

EC1: The entered word is alphabetic (valid)

EC2: The entered word is not alphabetic (invalid)

EC3: The entered word is 5 characters in length (valid)

EC4: The entered word is less than 5 characters in length (invalid)

EC5: The entered word greater than 5 characters in length (invalid)

EC6: The entered letter is in the word and the correct place, the background of the grid to that specific letter turns green (valid)

EC7: The entered letter is in the word but not in the correct place, the background of the grid to that specific letter turns yellow(valid)

EC8: The entered letter is not in the word, the background of the grid to that specific letter turns gray(valid)

EC9 The user wins the game and the WinCount increases by 1(valid)

EC10 The user wins the game, and the WinCount does not increase by 1(invalid)

EC11 The user plays the game, and the PlayCount increases by 1(valid)

EC12 The user wins the game, and the PlayCount does not increase by 1 (invalid)

EC13 The entered word is lowercase(valid)

EC14 The entered word is not lowercase(invalid)

EC15 The entered word is in the Words.txt(valid)

EC16 The entered word is not in the Words.txt (invalid)

EC17 The user clicks on Keyboard button then 1 player button, the corresponding does page open(valid)

EC18 The user clicks on Keyboard button then 1 Player button, the corresponding page does not open (invalid)

EC19 The user clicks on Keyboard button then 2 Players button, the corresponding page does open (valid)

EC20 The user clicks on Keyboard button then 2 Players button, the corresponding page does not open (invalid)

EC21 The user clicks on Drag and Drop button then 1 player button, the corresponding does page open (valid)

EC22 The user clicks on Drag and Drop button then 1 Player button, the corresponding page does not open (invalid)

EC23 The user clicks on Drag and Drop button then 2 Players button, the corresponding page does open (valid)

EC24 The user clicks on Drag and Drop button then 2 Players button, the corresponding page does not open (invalid)

EC25 The user guesses the target word correctly, animation page does open (valid)

EC26 The user guesses the target word correctly, animation page does not open (invalid)

EC27 the user clicks on the square in the animation pages, Statistics page does open (valid)

EC28 the user clicks on the square in the animation pages, Statistics page does not open (invalid)

EC29 The user does not guess the target word correctly, Statistics page does open (valid)

EC30 The user does not guess the target word correctly, Statistics page does not open (invalid)

Test cases, input values, valid conditions, and invalid conditions table

Test case identifier	Input value	Valid conditions	Invalid Conditions
1	hello, target word is wheel	EC1, EC3, EC6, EC7, EC8, EC13, EC15	
2	Apple	EC1, EC3	EC14, EC16
3	play	EC1, EC13	EC4, EC16
4	Write2		EC2, EC6, EC13, EC16
5	banana	EC1, EC13	EC5, EC16
6	cArRoT	EC1	EC14, EC16
7	whale, target word is wheel	EC1, EC3, EC6, EC7, EC8, EC13, EC15	
8	fluke	EC1, EC3, EC6, EC7, EC8, EC13	EC16

Table 1: Test cases

- **Finite state machine**
- **Test Cases:**

1) Checks if the high score is updated correctly in "HighestScore.txt" File (Test Passed)

```
2)    public static String highestScoreReplace() {
3)          String highestScore="";
4)          String replaceHighetScore=yourScoreCount();
5)          String h;
6)          File hadi= new File("HighestScore.txt");
7)          BufferedReader br = null;
8)          try {
9)              br = new BufferedReader(new FileReader(hadi));
10)             } catch (FileNotFoundException e) {
11)             e.printStackTrace();
12)             }
13)             try {
14)             String st="";
15)             st = br.readLine();
16)             while (st != null) {
17)             if(st.contains("Highest Score:") ) {
18)                         highestScore+=st.substring(14);
19)             }
20)             break;
21)             }
22)             if(Integer.parseInt(replaceHighetScore) >
        Integer.parseInt(highestScore)) {
23)              st = st.replace(highestScore, replaceHighetScore);
24)             highestScore=st.substring(14);
25)             }
26)
27)             FileWriter fw = new FileWriter(hadi);
28)             fw.write(st);
29)             fw.close();
30)
31)             } catch (IOException e) {
32)             e.printStackTrace();
33)             }
34)             return highestScore;

  }
```

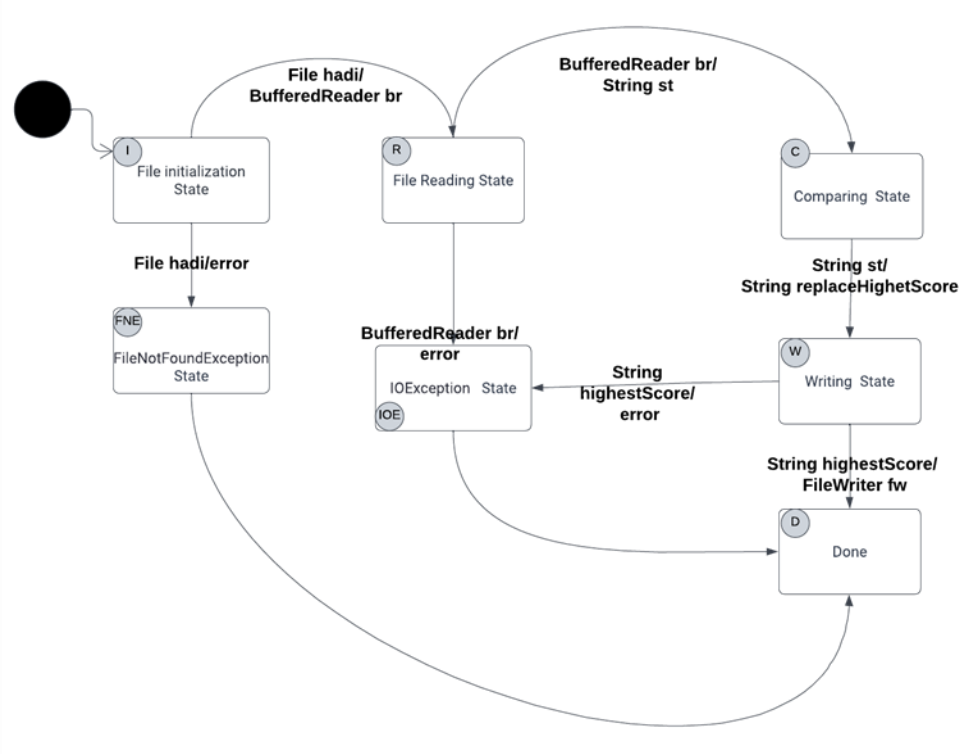


Figure 9: Graph representation (Mealy machine) for test case 1

A Mealy machine is a type of finite state machine that is defined by a finite set of states, a set of input symbols, a set of output symbols, and a state transition function that takes as input a current state and an input symbol and produces an output symbol and a next state.

In the given code, the code starts by reading a file called "HighestScore.txt" and checks if the current score, represented by the string returned by the "yourScoreCount()" method, is higher than the current highest score recorded in the file. If it is, the highest score in the file is replaced with the current score. The output of the code is the highest score after the potential replacement.

The states of the Mealy machine in this code include

1. **File initialization state**: In this state, the program creates a new *File* object and initializes a *BufferedReader* to read from the file.
2. **File reading state:** In this state, the program reads a line from the file using the *BufferedReader*.
3. **Comparing state:** In this state, the program compares the current highest score, stored in the *highestScore* variable, with the user's current score, stored in the *replaceHighestScore* variable. If the user's current score is greater than the current highest score, the *highestScore* variable is updated to the user's current score.

4. **Writing state**: In this state, the program writes the current highest score to the file using a *FileWriter*.
5. **FileNotFoundException state**: If a *FileNotFoundException* is thrown when trying to initialize the *BufferedReader*, the program enters this state and prints the exception to the console.
6. **IOException state**: If an *IOException* is thrown when trying to read from the file or write to the file, the program enters this state and prints the exception to the console.
7. **Done state**: After the program has finished executing all of its logic, it enters the done state and returns the current highest score to the caller.

The state transition function of the Mealy machine in this code includes

1. **File initialization state**:
    - If the file is successfully initialized, the next state is the file reading state, and the output is a File object that initializes a BufferedReader to read from the file.
    - If a *FileNotFoundException* is thrown, the next state is the FileNotFoundException state, and the output is an error.
2. **File reading state:**
    - If a line is successfully read from the file, the next state is the comparing state, and the output *String st*.
    - If an *IOException* is thrown, the next state is the IOException state, and the output is an error.
3. **Comparing state:**
    - If the current score is greater than the current highest score, the next state is the writing state, and the output is the current score.
    - If the current score is not greater than the current highest score, the next state is the writing state, and the output is the current highest score.
4. **Writing state:**

    - If the file is successfully written to, the next state is the done state, and the output is current highest score that has been stored to the file using a FileWriter.

    - If an IOException is thrown, the next state is the IOException state.

5. **FileNotFoundException state:** The next state is the done state.
6. **IOException state**: The next state is the done state.

	File hadi	BufferedReader br	String st	String highestScore
I	R/ BufferedReader br			
R		C/ String st		
C			W/ String replaceHighetScore	
W				D/ FileWriter fw
FNE	D/error			
IOE		D/error		D/error

Table 2: Tabular representation for test case 1

In this tabular representation of a Mealy machine Each row of the table represents a state of the machine, and each column represents an input symbol. The entries in the table contain the next state and the output for the machine when it is in the given state and receives the given input symbol.

Figure 10: Before updating the highest score is 40%

Figure 11: After updating the highest score is 100%

Figure 12: no updating happened on the highest score is 100% as the user score this time is lower than the highest score

Finally, There are several additional constraints or requirements that may affect the finite state testing for the code such as:

- The code assumes that the "HighestScore.txt" file contains a line that starts with "Highest Score: " followed by an integer value representing the current highest score. If the file does not have this format, the code may not work correctly.
- The code uses the "yourScoreCount()" method to get the current score, so the tester should ensure that this method is functioning correctly and returns a valid integer value.

2)Checks if the playCount is updated correctly in “playCountScore.txt” File (Test Failed)

```
1)   public static String playCountRep() {
2)         String playCount="";
3)         int playCountint = 0;
4)         File hadi= new File("playCountScore.txt");
5)         BufferedReader br = null;
6)         try {
7)             br = new BufferedReader(new FileReader(hadi));
8)         } catch (FileNotFoundException e) {
9)             e.printStackTrace();
10)        }
11)        try {
12)              String st=br.readLine();
13)              while (st  != null) {
14)
15)              if(st.contains("Play Count:")) {
16)                    playCount+=st.substring(11);
17)                    playCountint=Integer.parseInt(playCount);
18)                   playCountint++;
19)                   break;
20)
21)              }
22)              }
23)              st = st.replaceAll(playCount,playCountint+"");
24)              playCount=st.substring(11);
25)
26)
27)              FileWriter fw = new FileWriter(hadi);
28)              fw.write(st);
29)              fw.close();
30)
31)        } catch (IOException e) {
32)              e.printStackTrace();
33)        }
34)
35)        return playCount;
```

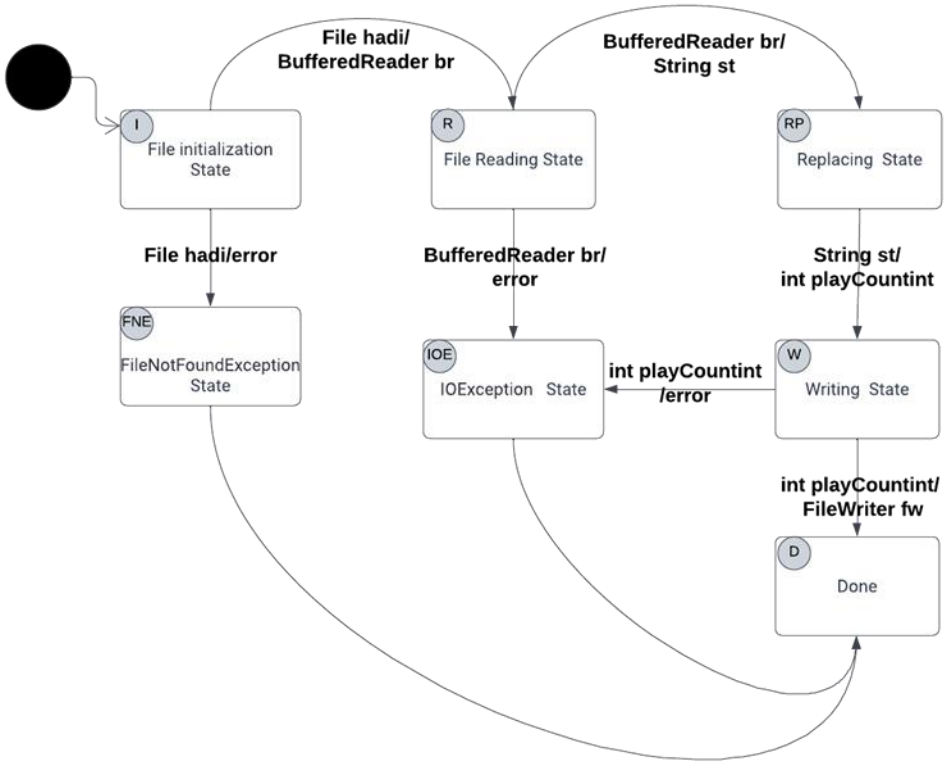


Figure 13: Graph representation (Mealy machine) for test case 2

In the given code, the code starts by reading a file called "playCountScore.txt" and increments the play count recorded in that file by 1. The output of the Mealy machine is the updated play count.

The states of the Mealy machine in this code include

1. **File initialization state**: In this state, the program creates a new *File* object and initializes a *BufferedReader* to read from the file.
2. **File reading state:** In this state, the program reads a line from the file using the *BufferedReader* which is "playCountScore.txt" file and retrieves the play count recorded in it
3. **Replacing state:** In this state, the program increased play count recorded in the file by 1.
4. **Writing state:** In this state, the program writes the incremented play count to the file using a *FileWriter*.

5. **FileNotFoundException state:** If a *FileNotFoundException* is thrown when trying to initialize the *BufferedReader*, the program enters this state and prints the exception to the console.
6. **IOException state:** If an *IOException* is thrown when trying to read from the file or write to the file, the program enters this state and prints the exception to the console.
7. **Done state:** After the program has finished executing all of its logic, it enters the done state and returns the updated play count to the caller.

The state transition function of the Mealy machine in this code includes

1. **File initialization state:**
    - If the file is successfully initialized, the next state is the file reading state, and the output is a File object that initializes a BufferedReader to read from the file.
    - If a *FileNotFoundException* is thrown, the next state is the FileNotFoundException state, and the output is an error.

2. **File reading state:**
    - If a line is successfully read from the file, the next state is the comparing state, and the output *String st*.
    - If an *IOException* is thrown, the next state is the IOException state, and the output is an error.

3. **Replacing state:**
    - The next state is the writing state, and the output is the updated play count (play count + 1).

4. **Writing state:**
    - If the file is successfully written to, the next state is the done state, and the output is the updated play count.
    - If an IOException is thrown, the next state is the IOException state, and the output is an error.

5. **FileNotFoundException state:** The next state is the done state.
6. **IOException state:** The next state is the done state.

	File hadi	BufferedReader br	String st	int playCountint
I	R/ BufferedReader br			
R		RP/ String st		
RP			W/ int playCountint	
W				D/ FileWriter fw
FNE	D/error			
IOE		D/error		D/error

Table 3: Tabular representation for test case 2

Figure 14: Before updating the play, count is 202

Figure 15: After updating the play count is 204, this is wrong it should be 203, and the play count is increased by 2.

After the test failed, I analyzed the code to identify the cause of the issue. I initially suspected that there was a problem with the method, but upon further investigation, I determined that the method was implemented correctly. The issue was that the method was being called twice in different parts of the program, which caused the play count to be incremented by 2 instead of 1. To fix the issue, I refactored the code to store the result of the method in a variable and used the variable instead of calling the method multiple times. This resolved the issue, and the code is now functioning correctly.

Figure 16: Before updating the play, count is 205- after fixing the code

Figure 17: After updating the play count is 206, and the code is behaving in an expected way

In the first and second test cases, the **Sensitivity principle** is not being followed because both methods create a BufferedReader to read from a file, but they do not close the BufferedReader after they are done reading from it. This means that the program may not always fail when it is run in different environments, as the BufferedReader may not always be closed properly. This goes against the principle of Sensitivity, which states that it is better for a program to fail every time rather than sometimes, as it makes it easier to identify and fix problems.

In test cases 1 and 2, the **Redundancy principle** is applied by implementing the same steps of opening the file, reading it, and writing it in both methods. This makes it easier to identify the source of any errors that may occur. When debugging the issue of the play count being increased by 2 instead of 1, the redundancy in the implementation helped to clarify that the issue was not with the method itself, but with something else.

In test cases 1 and 2, the **Visibility principle** is applied through the use of try-catch blocks, which allow for the program to display errors if something unexpected occurs. This makes information about the program's execution more accessible, making it easier to identify any issue.

3. **Checks if the correct interface gets displayed. (Test Passed)**

```
1) public static String playCountRep() {
2)     public void actionPerformed(ActionEvent e) {
3)         WelcomePage w = new WelcomePage();
4)         if(e.getSource() == player1 && w.isKeyPressed()) {
5)             PlayerParent p1 = new PlayerParent("player 1");
6)             Keyboard k = new Keyboard(p1);
7)             k.setVisible(true);
8)             this.dispose();
9)         }
10)         else if(e.getSource() == player1 && w.isDragPressed()) {
11)             PlayerParent p1 = new PlayerParent("player 1");
12)             DragAndDrop d = new DragAndDrop(p1);
13)             d.setVisible(true);
14)             this.dispose();
15)         }
16)         else if(e.getSource() == player2 && w.isKeyPressed()) {
17)             PlayerParent p1 = new PlayerParent("player 1");
18)             PlayerParent p2 = new PlayerParent("player 2");
19)             Keyboard k = new Keyboard(p1, p2);
20)             k.setVisible(true);
21)             this.dispose();
22)         }
23)         else if(e.getSource() == player2 && w.isDragPressed()) {
24)             PlayerParent p1 = new PlayerParent("player 1");
25)             PlayerParent p2 = new PlayerParent("player 2");
26)             DragAndDrop d = new DragAndDrop(p1,p2);
27)             d.setVisible(true);
28)             this.dispose();
29)         }
30)     }
```

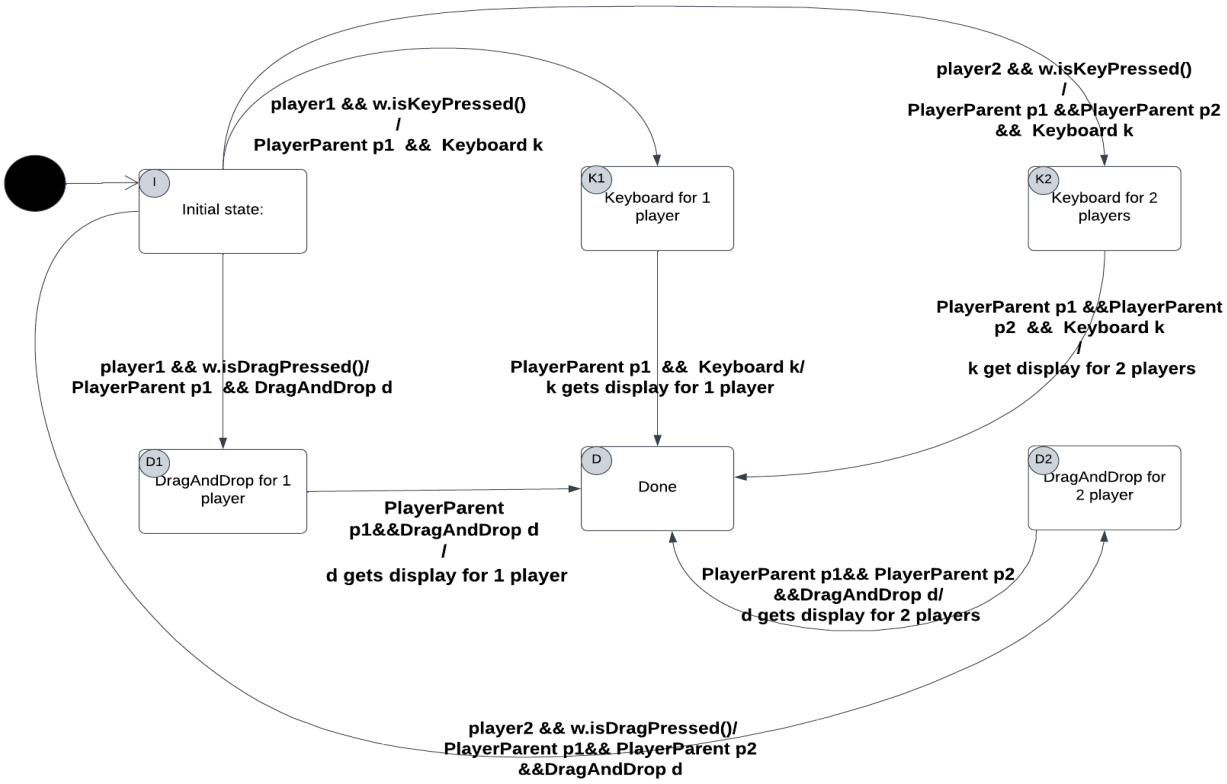


Figure 18: Graph representation (Mealy machine) for test case

In the given code, the code processes user actions and determines what action to take based on the source of the action and the state of the system. The output of the code is the display of a specific interface.

The states of the Mealy machine in this code include

1. **Initial state:** before any user action has been taken
2. **Keyboard for 1 player state**: after the user has pressed the player 1 button and the "isKeyPressed()" method returns true
3. **Keyboard for 2 players' state**: after the user has pressed the player 2 button and the "isKeyPressed()" method returns true
4. **DragAndDrop for 1 player state**: after the user has pressed the player 1 button and the "isDragPressed()" method returns true
5. **DragAndDrop for 2 players' state**: after the user has pressed player 2 button and the "isDragPressed()" method returns true
6. **Done state**: after the appropriate interface will be displayed and the current window will be disposed, and the specific interface will be displayed

The state transition function of the Mealy machine in this code includes

1. **Initial state:**
    - If the user presses the player 1 button successfully and the "isKeyPressed()" method returns true, the next state is Keyboard for 1 player state
    - If the user presses the player 2 button successfully and the "isKeyPressed()" method returns true, the next state is Keyboard for 2 players' state
    - If the user presses the player 1 button successfully and the "isDragPressed()" method returns true, the next state is DragAndDrop for 1 player state
    - If the user presses the player 2 button successfully and the "isDragPressed()" method returns true, the next state is DragAndDrop for 2 players' state
2. **Keyboard for 1 player state**
    - If the "setVisible()" method is called successfully and the current window of Keyboard for 1 player is displayed , the next state is Done state
3. **Keyboard for 2 players' state**
    - If the "setVisible()" method is called successfully and the current window of Keyboard for 2 player is displayed , the next state is Done state
4. **DragAndDrop for 1 player state**
    - the "setVisible()" method is called successfully and the current window of Drag and drop for 1 player is displayed , the next state is Done state
5. **DragAndDrop for 2 players' state**
    - "setVisible()" method is called successfully and the current window of Drag and drop for 2 players is displayed , the next state is Done state

	player1&& w.isKeyPressed()	p1&& k	player2 && w.isKeyPressed()	p1 && p2 && k	player1 && w.isDragPressed()	p1 && d	player2 && w.isDragPressed()	p1&& p2 && d
I	K1/ p1 && k		K2/ p1 && p2 && k		D1/ p1 && d		D2/ p1&& p2 && d	
K1		D/k gets display for 1 player						
K2				D/ k get display for 2 players				
D1						D/ d gets display for 1 player		
D2								D/ d gets display for 1 player

Table 4: Tabular representation for test case 3

The case test has passed as all interfaces were displayed correctly.

- **Cause–Effect Graphs**

1. Test whether the correct message was written to the log file (test passed)

```
2. private void scoreTwoPlayers() throws IOException { // Keeps a
   log of the player's previous attempts
3. BufferedWriter writer = new BufferedWriter(new FileWriter(log,
   true));
4. writer.write("(Two Player Game!)\n");
5. if (victory && (player1Tries > player2Tries)) {
6. writer.write("\nPLAYER 1 GOT IT!" + "\nGuessed in " +
   player1Tries + " tries!\n" + "The word was: " + target +
7. "\n_____\n");
8. }else if(victory){
9. writer.write("\nPLAYER 2 GOT IT!" + "\nGuessed in " +
   player2Tries + " tries!\n" + "The word was: " + target +
10.      "\n_____\n");
11.      }else {
12.      writer.write("\nUnfortunately, neither of you got it. The
   word was: " + target + "\nBetter luck tomorrow!\n" +
13.      "\n_____\n");
14.      }
15.      writer.close();
16.      }
```

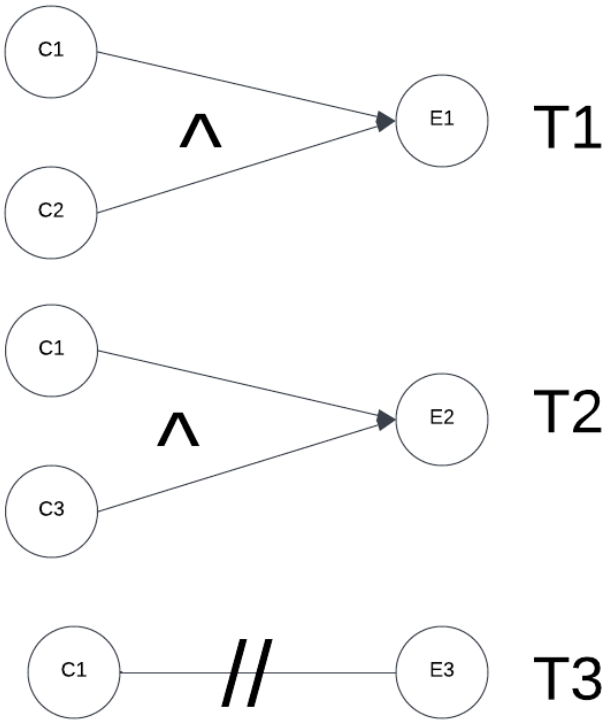


Figure 19: Cause–Effect Graph test case 1

The input conditions or causes are as follows:

- **C1**: Victory
- **C2**: Player 1 Tries > Player 2 Tries
- **C3**:Player 1 Tries < Player 2 Tries

The output conditions or effects are:

- **E1:** Write to log file: "PLAYER 1 GOT IT"
- **E2:** Write to log file: "PLAYER 2 GOT IT"
- **E3:** Write to log file: "Neither player got it"

The rules or relationships can be described as follows:

- If C1 and C2, then E1.
- If C1 and C3, then E2.
- If not C1, then E3

	T1	T2	T3
C1	1	1	0
C2	1	0	-
C3	0	1	-
E1	1	0	0
E2	0	1	0
E3	0	0	1

.

Table 5 : decision table for test case 1

To determine if the test has passed or failed, we checked the output of the program, which in this case is the log file. the log file contained the appropriate messages for all given input causes; therefore, the test has passed.

2. **Checks if the correct interface gets displayed (test passed)**

```
1)    public static String playCountRep() {
2)        public void actionPerformed(ActionEvent e) {
3)            WelcomePage w = new WelcomePage();
4)            if(e.getSource() == player1 && w.isKeyPressed()) {
5)                PlayerParent p1 = new PlayerParent("player 1");
6)                Keyboard k = new Keyboard(p1);
7)                k.setVisible(true);
8)                this.dispose();
9)            }
10)           else if(e.getSource() == player1 && w.isDragPressed()) {
11)               PlayerParent p1 = new PlayerParent("player 1");
12)               DragAndDrop d = new DragAndDrop(p1);
13)               d.setVisible(true);
14)               this.dispose();
15)           }
16)           else if(e.getSource() == player2 && w.isKeyPressed()) {
17)               PlayerParent p1 = new PlayerParent("player 1");
18)               PlayerParent p2 = new PlayerParent("player 2");
19)               Keyboard k = new Keyboard(p1, p2);
20)               k.setVisible(true);
21)               this.dispose();
22)           }
23)           else if(e.getSource() == player2 && w.isDragPressed()) {
24)               PlayerParent p1 = new PlayerParent("player 1");
25)               PlayerParent p2 = new PlayerParent("player 2");
26)               DragAndDrop d = new DragAndDrop(p1,p2);
```

```
27)                 d.setVisible(true);
28)                 this.dispose();
29)             }
30)         }
```

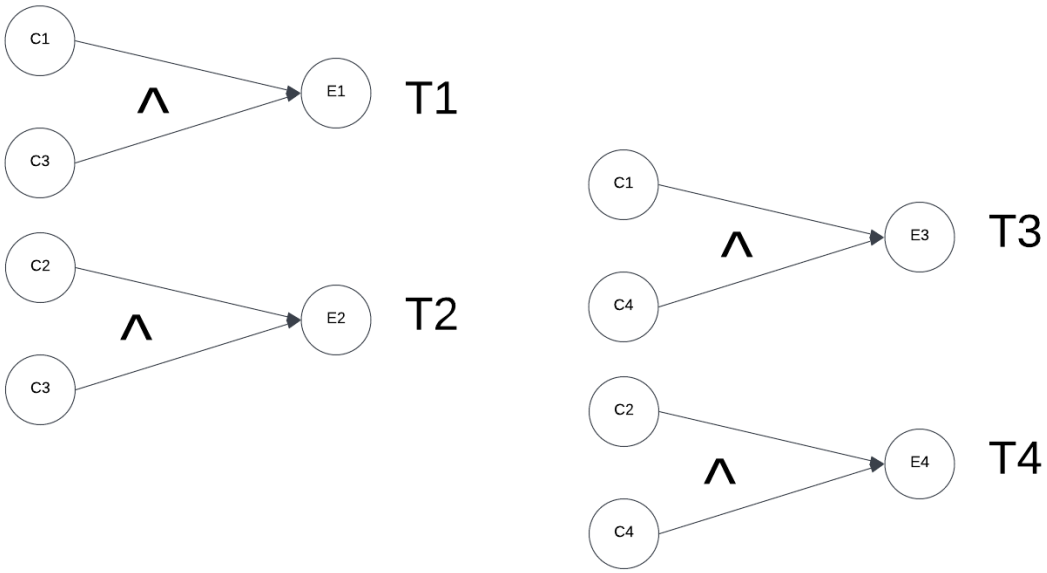


Figure 20: Cause–Effect Graph test case 2

The input conditions or causes are as follows:

- **C1**: ActionEvent (triggered by player1 button)
- **C2**: ActionEvent (triggered by player2 button)
- **C3**: WelcomePage.isKeyPressed() returns true
- **C4**: WelcomePage.isDragPressed() returns true

The output conditions or effects are:

- **E1:** keyboard gets displayed for 1 player
- **E2:** keyboard gets displayed for 2 players
- **E3:** Drang and drop gets displayed for 1 player
- **E4:** Drang and drop gets displayed for 2 players

The rules or relationships can be described as follows:

- If C1 and C3, then E1.
- If C2 and C3, then E2.
- If C1 and C4, then E3
- If C2 and C4, then E4

	T1	T2	T3	T4
C1	1	0	1	0
C2	0	1	0	1
C3	1	1	0	0
C4	0	0	1	1
E1	1	0	0	0
E2	0	1	0	0
E3	0	0	1	0
E4	0	0	0	1

Table 6 : decision table for test case 2

To determine if the test has passed or failed, we checked the program's behavior and checked if the appropriate objects are created and set as visible when the relevant button is pressed, and the corresponding conditions are met. the program behaves as expected; therefore, the test has passed.

3. **It checks if the animation is moving diagonally and bouncing off the edges of the frame when it reaches them and changing direction accordingly .(test passed)**

```
1. public void actionPerformed(ActionEvent e) {
2. if( x >= 500 - anim.getWidth(null) || x<0) {
3. xVelocity= xVelocity *-1;
4. }
5. x+=xVelocity;
6. if( y >= 500 - anim.getHeight(null) || y<0) {
7. yVelocity= yVelocity *-1;
8. }
9. y+=yVelocity;
10.      repaint();
11.      }
```

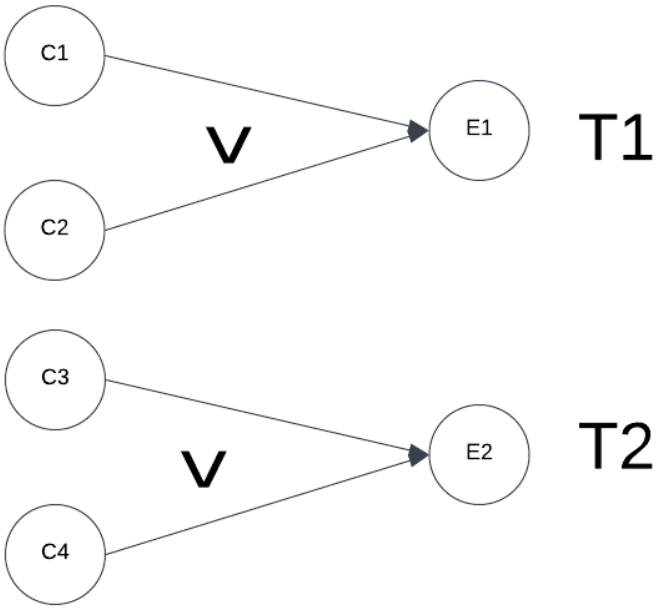


Figure 21: Cause–Effect Graph test case 3

The input conditions or causes are as follows:

- **C1**: x >= 500 - anim.getWidth(null)
- **C2**: x < 0
- **C3**: y >= 500 - anim.getHeight(null)
- **C4**: y < 0

The output conditions or effects are:

- **E1:** Update x by adding xVelocity
- **E2:** Update y by adding yVelocity

The rules or relationships can be described as follows:

- If C1 or C2, then E1.
- If C3 or C4, then E2.

	T1	T2
C1	1	0
C2	1	0
C3	0	1
C4	0	1
E1	1	0
E2	0	1

Table 7: decision table for test case 3

To determine if the test has passed or failed, we observed the behavior of the animation and checked if it is moving and bouncing off the edges of the frame as expected. The animation behaved as expected, therefore, the test has passed.

- **Combinatorial Testing (Pairwise approach)**

1) If the Player chooses Keyboard or Drag & Drop (Test Passed)

```
2)    public void actionPerformed(ActionEvent e) {
3)    if(e.getSource() == butKeyboard) {
4)    keyPressed = true;
5)    Player p = new Player();
6)    p.setVisible(true);
7)    this.dispose();
8)    }
9)    else if(e.getSource() == butDrag) {
10)   dragPressed = true;
11)   Player p = new Player();
12)   p.setVisible(true);
13)   this.dispose();
14)   }
15)   }
```

Combinatorial testing using a pairwise approach involves creating test cases that cover all possible combinations of input values. In this case, the input values are the two possible actions that can be taken when the actionPerformed method is called: either the butKeyboard button is clicked or the butDrag button is clicked.

to create a set of test cases using a pairwise approach, we generated a matrix of all possible combinations of input values.

	butKeyboard	butDrag
1	true	false
2	false	true

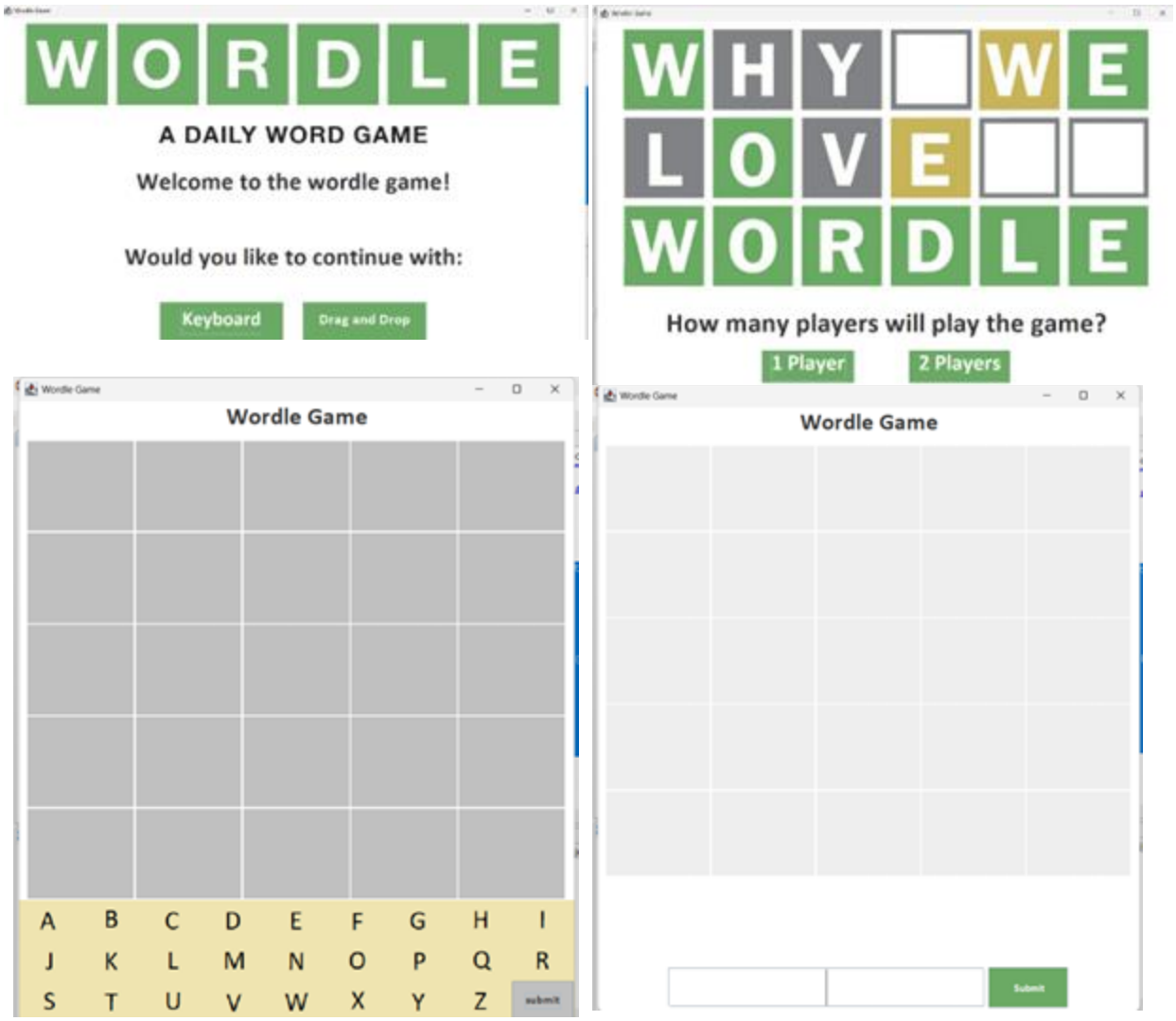
Table 8: matrix of all possible combinations of input values.

Each row in the matrix represents a test case, and the values in the columns indicate the input values for that test case.

To consider these test cases as passed, we would need to execute each test case and verify that the expected behavior occurs. For example, in test case 1, we would click the `butKeyboard` button and verify that the Player window is displayed and the `this.dispose()` method is called. Similarly, in test case 2, we would click the `butDrag` button and verify that the same expected behavior occurs.

We also considered additional constraints or requirements that may affect the test cases. For example, whether the Player window is displayed correctly We also considered the performance of the code under test and ensured that it performs as expected under different input combinations.

The test case applies the **principle of partitioning** by separating the behavior triggered by each button press into distinct branches of the if statement. This allows the code to handle each button press separately and ensures that the correct behavior is triggered for each button.



Figures 22: Wordle game Screenshots

Combinatorial testing PASSED

1) Velocity / Movement of the animation **(Test Passed)**

```
2) public void actionPerformed(ActionEvent e) {
3) if( x >= 500 - anim.getWidth(null) || x<0) {
4) xVelocity= xVelocity *-1;
5) }
6) x+=xVelocity;
7) if( y >= 500 - anim.getHeight(null) || y<0) {
8) yVelocity= yVelocity *-1;
9) }
10) y+=yVelocity;
11) repaint();
12) }
```

Two testing principles can be applied here:

Partitioning: This means dividing the input domain into distinct groups or partitions and testing the functionality of the code for each partition. In the case of this code, we could partition the input domain for the x and y variables into different ranges, such as x < 250, 250 <= x < 500, and x >= 500. Then, we could test the code to ensure that it is functioning correctly for each of these partitions.

Another testing principle that could be applied is the **principle of redundancy.** This means adding additional checks or tests to the code to ensure that it is functioning correctly. For example, we could add additional if statements to check that the x and y variables are within the valid range (0 <= x < 500 and 0 <= y < 500). This can help to catch any errors that may occur and improve the reliability of the code.

To perform combinatorial testing using the pairwise approach, we need to consider the different input values that could be passed to the ActionEvent object, as well as the state of the variables x, y, xVelocity, and yVelocity.

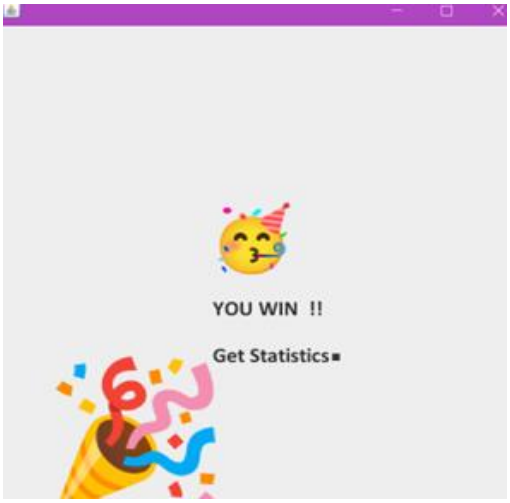
To summarize the test cases in a matrix, we focused on the most important combinations of input values. This could include combinations that represent edge cases or common usage scenarios.

- x and y: 0 and 500 (representing the minimum and maximum values)
- xVelocity and yVelocity: -1, 0, and 1 (representing the minimum and maximum values, as well as the value 0)
- Expected Output: the expected behavior of the code for the given input values

This matrix includes test cases that cover the minimum and maximum values for x, y, xVelocity, and yVelocity, as well as test cases that represent common usage scenarios**.**

x	y	xVelocity	yVelocity	Expected Output
0	0	-1	-1	x should be set to -1, y should be set to -1, and the screen should be repainted.
0	0	1	1	x should be set to 1, y should be set to 1, and the screen should be repainted.
500	500	-1	-1	x should be set to 499, y should be set to 499, and the screen should be repainted.
500	500	1	1	x should be set to 501, y should be set to 501, and the screen should be repainted.
0	500	0	0	x and y should remain unchanged, and the screen should not be repainted.
500	0	0	0	x and y should remain unchanged, and the screen should not be repainted.
-1	500	1	0	x should be set to 0, y should remain unchanged, and the screen should be repainted.
501	500	-1	0	x should be set to 500, y should remain unchanged, and the screen should be repainted.
500	-1	0	1	x should remain unchanged, y should be set to 0, and the screen should be repainted.
500	501	0	-1	x should remain unchanged, y should be set to 500, and the screen should be repainted.

Table 9: Matrix of input variables



Figures 23: Winning animation

Combinatorial testing PASSED

3.2 Structural Testing:

- **Test Cases:**

1) **Checks if the word is available and buffers it from the "Words.txt" File (Test Passed)**

```
2) protected WordBase(){
3) String line;
4) int i = 0;
5) try {
6) BufferedReader reader = new BufferedReader(new
   FileReader("Words.txt"));
7) while ((line = reader.readLine()) != null){
8) words[i] = line;
9) i++;
10)    }
11)    }catch (FileNotFoundException f){
12)    f.printStackTrace();
13)    }
14)    catch (IOException e){
15)    e.printStackTrace();
16)    }
17)    }
```

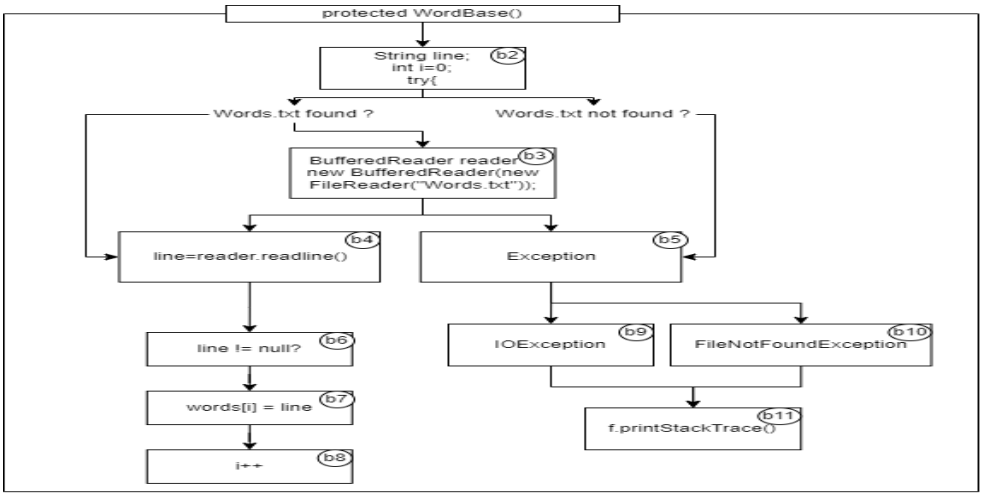


Figure 24: Control Flow Graph

The control flow graph represents the flow of control through the given Java code.

The graph begins with the WordBase() constructor, represented by a rectangle. This is the entry point of the control flow.

The control flow then proceeds to a try block, represented by a rectangle. The try block contains code that may throw an exception.

Inside the try block, the control flow creates a BufferedReader object named reader that reads from the Words.txt file. The control flow then enters a while loop, represented by a rectangular shape. The loop continues iterating as long as there are lines left to read in the file.

For each iteration of the loop, the control flow reads a line from the file and stores it in the line variable. It then stores the line variable in the words array at the current index i and increments i.

If an exception is thrown while reading the file, the control flow follows the path marked with the label "Exception" and enters either the FileNotFoundException block or the IOException block, depending on the type of exception thrown. These blocks contain code to print the stack trace of the exception.

Finally, the control flow returns to the beginning of the while loop and continues iterating until there are no more lines left to read in the file. When the loop terminates, the control flow returns to the WordBase() constructor, and the constructor terminates.

The **Visibility principle** is utilized in this test case through the usage of try-catch blocks, which allow the program to report errors if anything unexpected happens.

Figure 25: Word is found from txt file

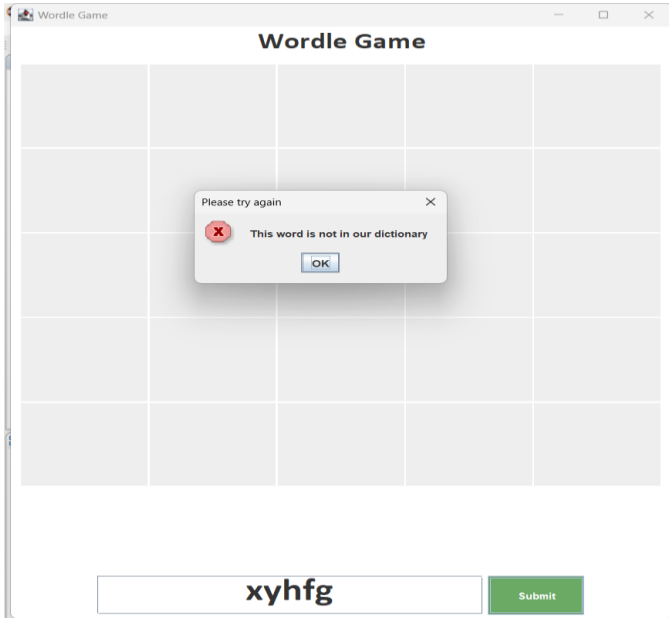


Figure 26: Word is not found from txt file

2) **If the Player chooses Keyboard or Drag & Drop (Test Passed)**

```
3) public void actionPerformed(ActionEvent e) {
4) if(e.getSource() == butKeyboard) {
5) keyPressed = true;
6) Player p = new Player();
7) p.setVisible(true);
8) this.dispose();
9) }
10)       else if(e.getSource() == butDrag) {
11)       dragPressed = true;
12)       Player p = new Player();
13)       p.setVisible(true);
14)       this.dispose();
15)       }
16)       }
```

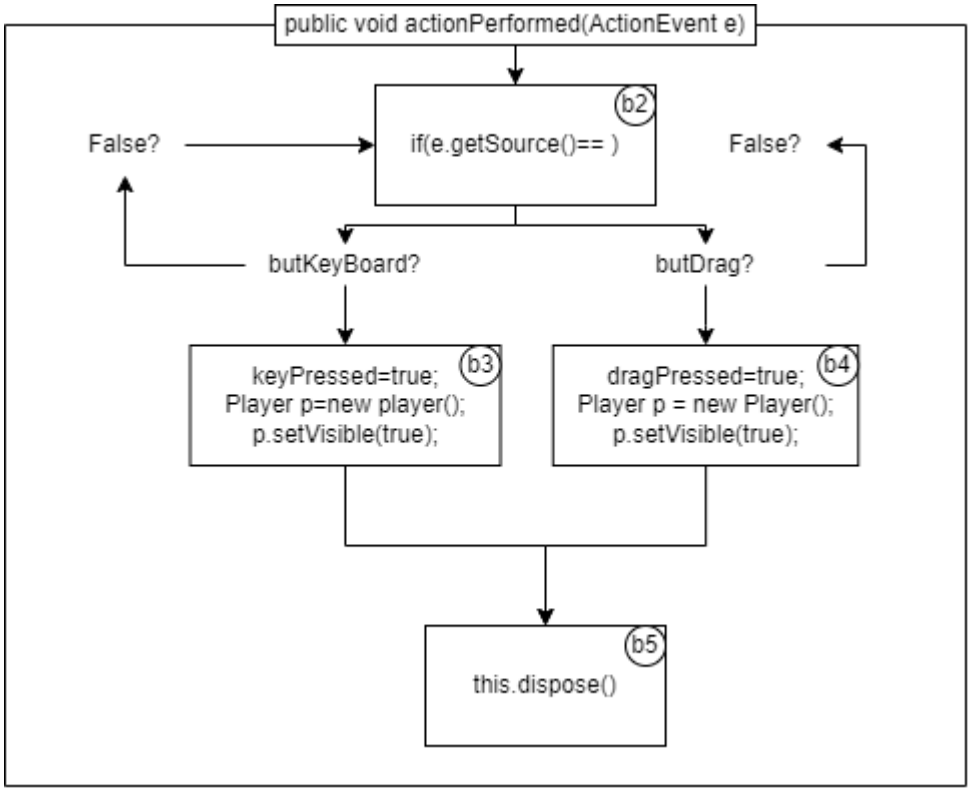


Figure 27: Control Flow Graph

The control flow graph represents the flow of control through the given Java code.

The graph begins with the `actionPerformed()` method, represented by a rectangle. This is the entry point of the control flow.

The control flow then proceeds to an `if` statement, represented by a rectangle shape. The `if` statement checks if the source of the `ActionEvent` object is the `butKeyboard` button.

If the source of the `ActionEvent` object is the `butKeyboard` button, the control flow follows the path marked with the label "True" and sets the `keyPressed` variable to `true`. It then creates a new `Player` object, makes it visible, and disposes of the current GUI window.

If the source of the `ActionEvent` object is not the `butKeyboard` button, the control flow follows the path marked with the label "False" and proceeds to an `else if` statement. The `else if` statement checks if the source of the `ActionEvent` object is the `butDrag` button.

If the source of the `ActionEvent` object is the `butDrag` button, the control flow follows the path marked with the label "True" and sets the `dragPressed` variable to `true`. It then creates a new `Player` object, makes it visible, and disposes of the current GUI window.

If the source of the `ActionEvent` object is neither the `butKeyboard` nor the `butDrag` button, the control flow follows the path marked with the label "False" and the method terminates.

Finally, the control flow returns to the `actionPerformed()` method, and the method terminates.

Figure 28: Welcome Page

3) **Checks if the right letter is in the wrong position. (Test Passed)**

```
4) public boolean isYellow(char letter, String targetWord) { //
   Check if the right letter is in a wrong position
5) for (int index = 0; index < targetWord.length(); index++) {
6) if (targetWord.charAt(index) == letter) {
7) return true;
8) }
9) }
10)     return false;
11)     }
```

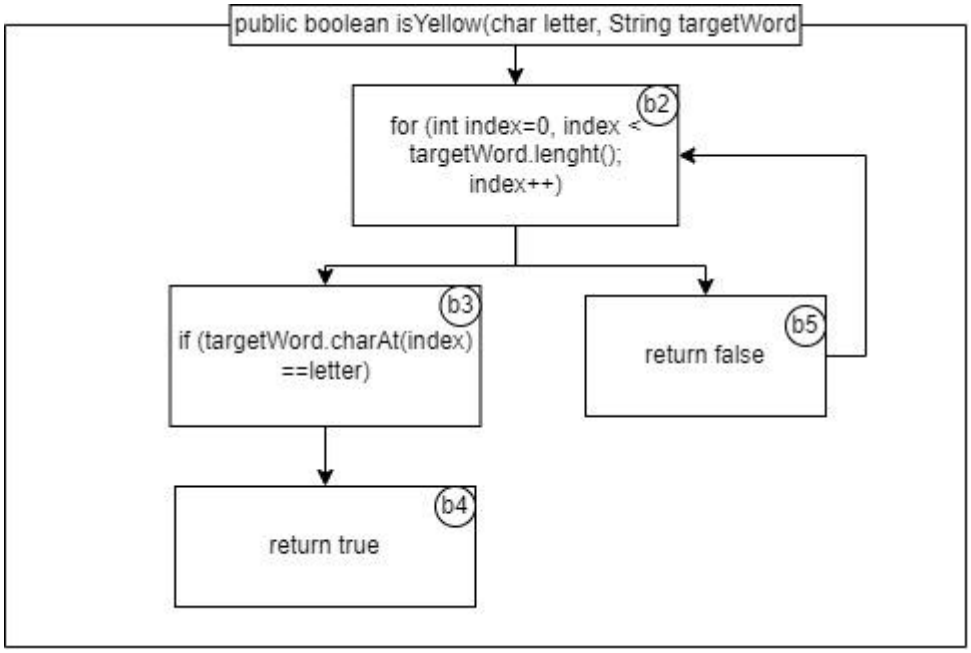


Figure 29: Control Flow Graph

The control flow graph represents the flow of control through the given Java code.

The graph begins with the isYellow() method, represented by a rectangle. This is the entry point of the control flow.

The control flow then proceeds to the for loop, represented by a rectangular shape. The for loop iterates through the characters in the targetWord string.

Inside the loop, the control flow branches out into two different paths based on the result of the comparison between the current character in the loop and the letter character.

If the characters match, the control flow follows the path marked with the label "True" and returns true.

If the characters do not match, the control flow follows the path marked with the label "False" and returns to the beginning of the loop. The loop continues iterating until all characters in the targetWord string have been processed.

If none of the characters in the targetWord string match the letter character, the control flow follows the path marked with the label "False" and returns false.

Finally, the control flow returns to the isYellow() method, and the method terminates.

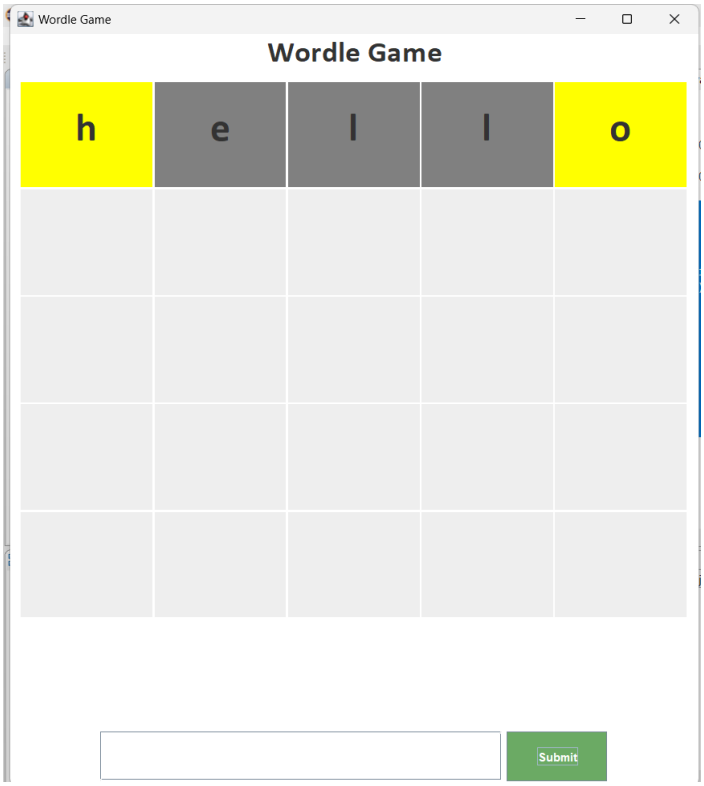


Figure 30: Letter Check

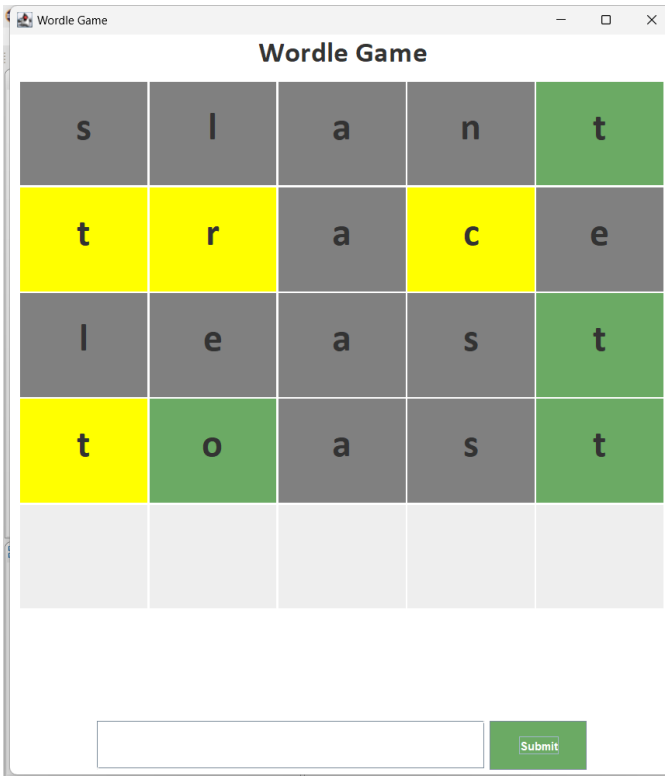


Figure 31: Letter Check

4) **Checks if the right letter is in the correct/wrong place and the wrong letter.**

```
5) public void checkWord(JLabel[] trial, String guess) { //
   checks if the right letter is in the correct position
6) for (int j = 0; j < target.length(); j++) {
7) if (trial[j].getText().equals(target.charAt(j) + "")) {
8) trial[j].setBackground(newGreen);
9) } else if (isYellow(guess.charAt(j), target)) {
10)    trial[j].setBackground(Color.yellow);
11)    } else {
12)    trial[j].setBackground(Color.gray);
13)    }
14)    }
15)    }
```

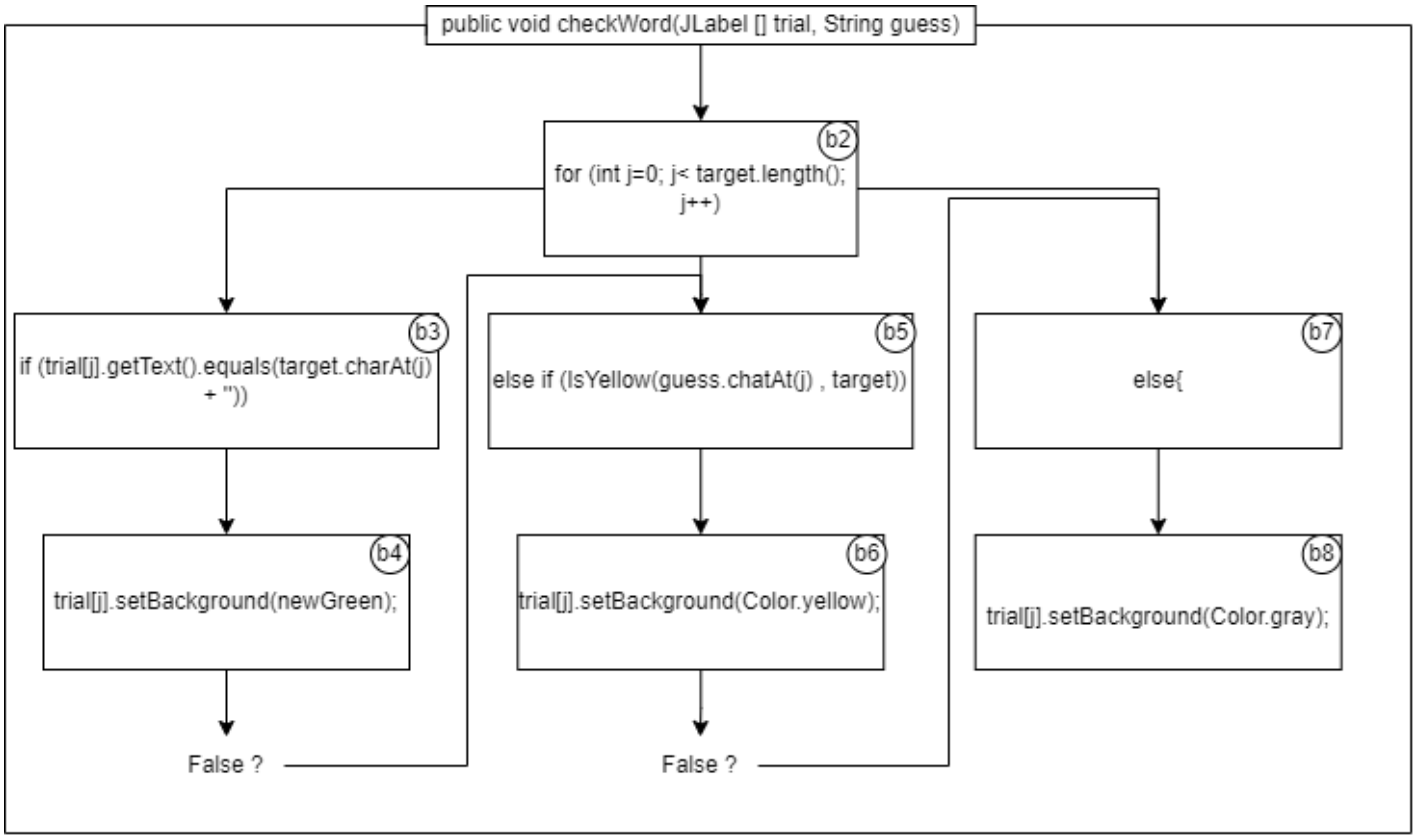


Figure 32: Control Flow Graph

The control flow graph represents the flow of control through the given Java code.

The graph begins with the checkWord() method, represented by a rectangle. This is the entry point of the control flow.

The control flow then proceeds to the for loop, represented by a rectangular shape. The for loop iterates through the elements of the trial array.

Inside the loop, the control flow branches out into three different paths based on the result of the comparison between the text of the JLabel object at the current index and the character at the same index in the target string.

If the text of the JLabel object matches the character in the target string, the control flow follows the path marked with the label "True" and sets the background color of the JLabel object to newGreen.

If the text of the JLabel object does not match the character in the target string, the control flow follows the path marked with the label "False" and calls the isYellow() method. If the isYellow() method returns true, the control flow follows the path marked with the label "True" and sets the background color of the JLabel object to Color.yellow.

If the isYellow() method returns false, the control flow follows the path marked with the label "False" and sets the background color of the JLabel object to Color.gray.

Finally, the control flow returns to the beginning of the loop and continues iterating until all elements of the trial array have been processed.

At the end of the loop, the control flow returns to the checkWord() method, and the method terminates

Figure 33: Letter Check

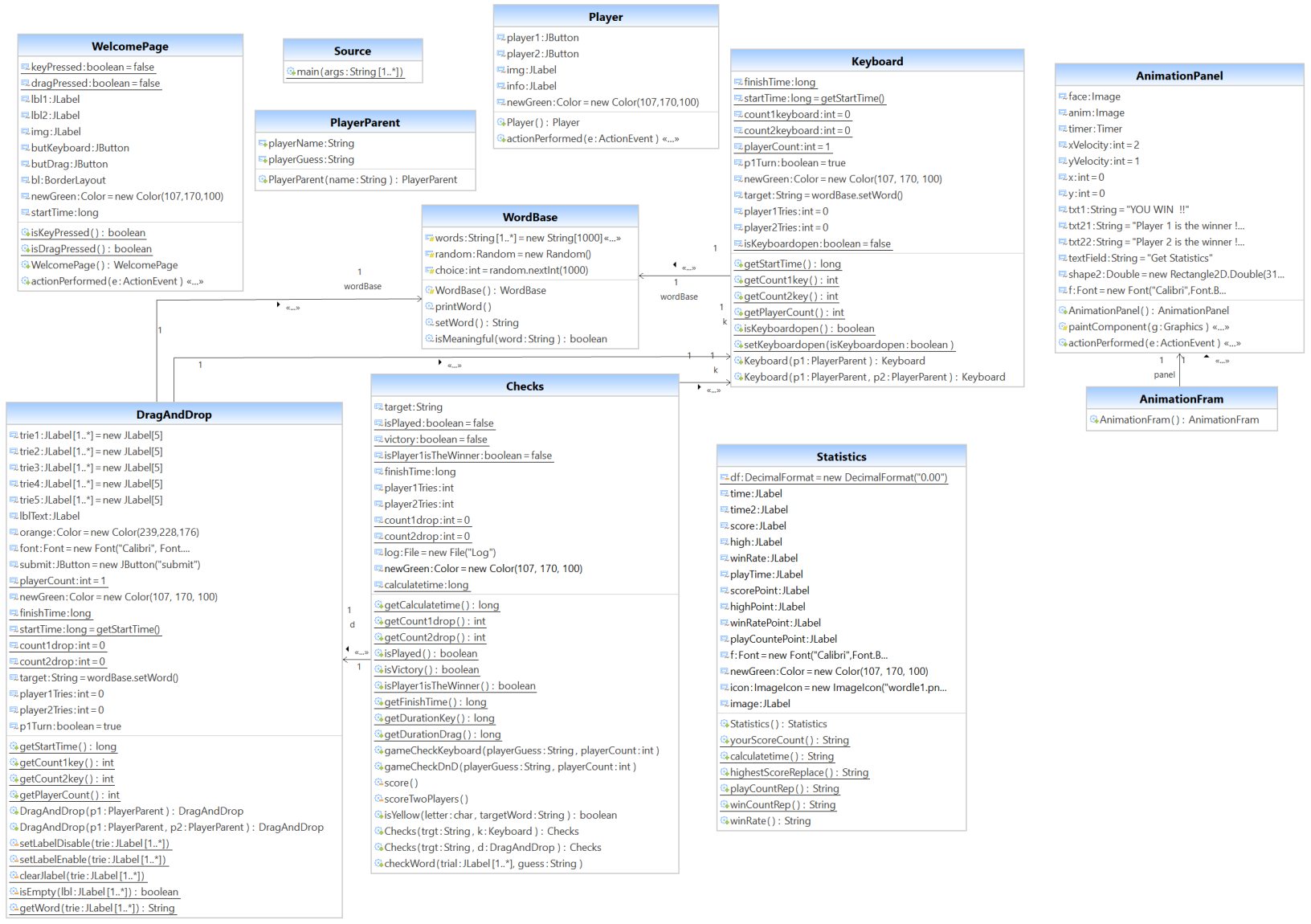


Figure 34: UML Diagram

5. Recommendations.

Some of the methods were extremely hard to evaluate as they have a lot of branching logic, with multiple conditions to check and different code paths to follow depending on the result of those checks. As well as some methods have multiple side effects, such as updating the text of multiple labels and displaying dialog boxes, which makes it harder to track its behavior and verify that it is working correctly.

Here are some recommendations

1. Split the methods into smaller, more focused methods that each handle a specific task or responsibility. For example, methods to update the labels with the player's guess, and a method to check if the player has won the game. This will make it easier to understand and test the different parts of the function.

2. Avoid using multiple return statements in the same function. Having multiple return statements can make it harder to understand the control flow of a method and can make it more difficult to test.

Other methods were lacking try and catch block or missing close() function, such as opening a file for reading but never closing it.

Here are some recommendations

1. Make sure to include try-catch blocks around any code that could throw an exception, and make sure to include a catch block for each type of exception that could be thrown. This will help to prevent your program from crashing and will allow you to handle exceptions in a more graceful way.

2. Consider using a static code analysis tool to automatically detect potential issues with try-catch blocks and resource management in your code. This can help to identify potential problems early on and make it easier to fix them.

Finally, this code was run using the Eclipse development environment and may behave differently when using other integrated development environments (IDEs).

References

[1] Bouyssounouse, B., & Sifakis, J. (2005, January). (PDF) tools for verification and validation - researchgate. ResearchGate. Retrieved December 13, 2022, from https://www.researchgate.net/publication/251372215_Tools_for_Verification_and_Validation.

[2] Holzmann, G. (n.d.). Formal verification. Spin. Retrieved December 13, 2022, from https://spinroot.com/spin/whatispin.html

[3] Jain, H. (n.d.). Satisfiability checking of non-clausal formulas using DPLL, graphs, and watched cuts: NFLSAT homepage. Retrieved December 13, 2022, from https://www.cs.cmu.edu/~hjain/nflsat-web/

[4] Alferidah, S. K., & Ahmed, S. (2020, September). (PDF) Automated Software Testing Tools - researchgate. ResearchGate. Retrieved December 16, 2022, from https://www.researchgate.net/publication/344311956_Automated_Software_Testing_Tools

Appendix

Figure 1: Only 1 player uses the keyboard to enter the word

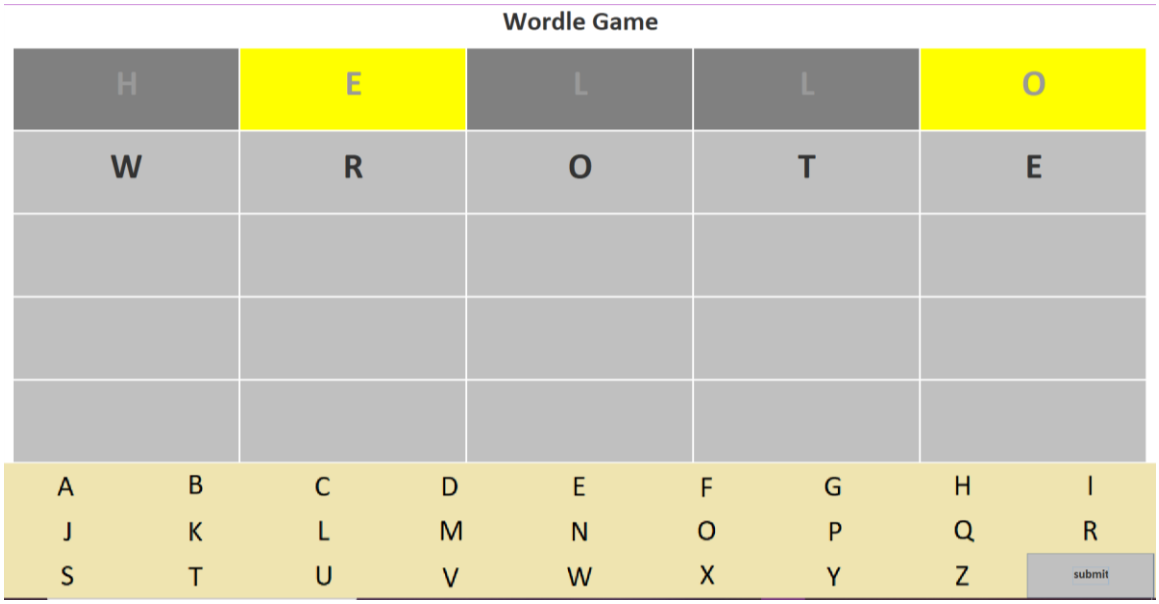


Figure 2: Only 1 player drops and drags the letter to enter the word

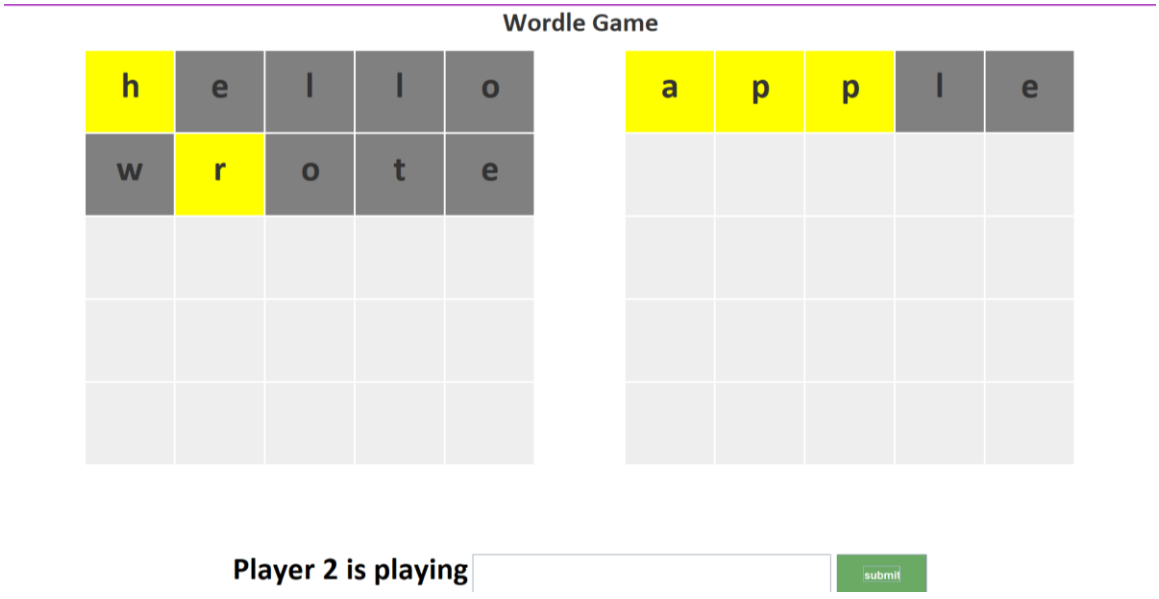


Figure 3: 2 players using the keyboard to enter the word

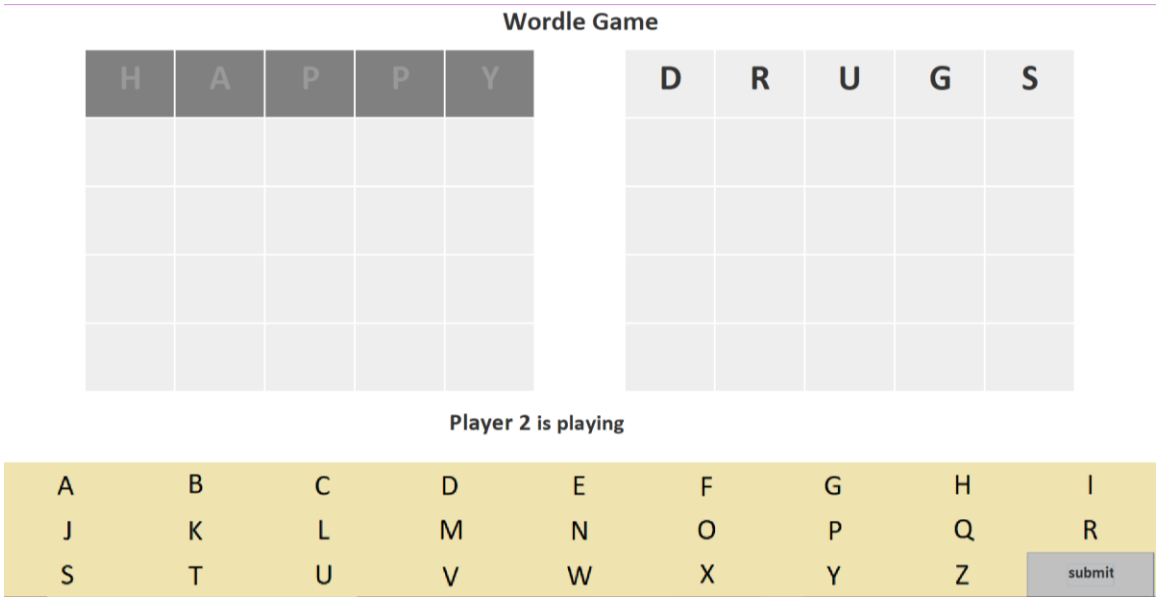


Figure 4: 2 players drop and drags the letter to enter the word

Figure 5: WelcomePage

Figure 6: Player.java

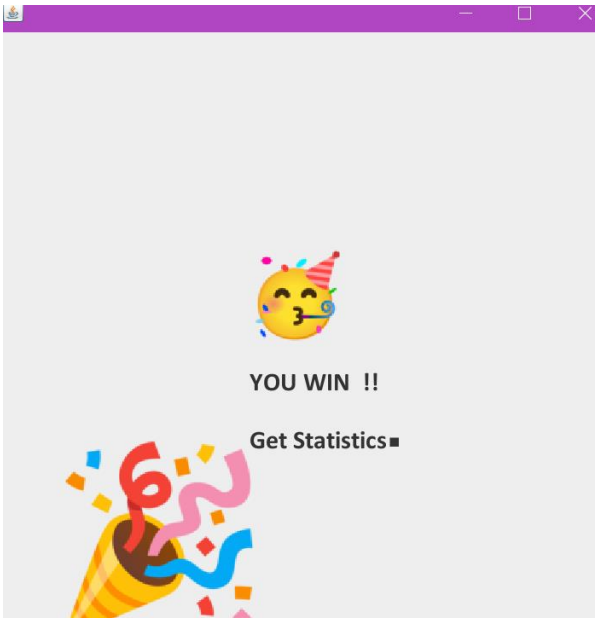


Figure 7: AnimationPanel.java

Figure 8: Statistics.java