CSC2002S Assignment 4
Concurrent Programming: Falling Words

Callum Tilbury

September 29, 2019

Contents

1 Classes **2**
1.1 Modified . . . 2
1.1.1 WordApp . . . 2
1.1.2 Score . . . 2
1.1.3 WordPanel . . . 2
1.1.4 WordRecord . . . 3
1.2 Created . . . 4
1.2.1 Model . . . 4
1.2.2 View . . . 4
1.2.3 ScoreUpdater . . . 4
1.2.4 Controller . . . 6

2 Concurrency Considerations **6**
2.1 Atomic Variable: Thread Count . . . 6
2.2 Synchronized Methods: Score . . . 7
2.3 Event-Dispatch Thread: Swing . . . 8
2.4 Volatile Variables: 'Done' flag . . . 8
2.5 No thread-safety required . . . 8
2.6 Preventing deadlock & Ensuring liveness . . . 8

3 Validation **9**

4 MVC Design Pattern **10**

5 Additional **10**
5.1 Theme: Word Colours . . . 10
5.2 High Score Record . . . 11
5.3 Levels of Difficulty . . . 11

Overview

This report details the design and creation of a simple typing game: various words fall from the top of the screen, and a user attempts to type each word before it reaches the bottom of the screen. If a word is typed correctly, the number of caught words is incremented, and the user's score increases by the length of the typed word. The word then disappears, and a new one starts falling. If a word is *not* typed correctly before it reaches the bottom, it disappears and the number of missed words is incremented. A new word then starts falling.

Emphasis for this game is placed on its concurrence requirement – a user must be able to interact easily and smoothly with the interface, while not noticing any change in the fluidity of the game's animation (the words falling). Having said that, this introduces the burden on ensuring data integrity when sharing resources and acting simultaneously on multiple threads.

1 Classes

1.1 Modified

The classes mentioned below were given as 'skeleton code' for the assignment, but were modified to various extents.

1.1.1 WordApp

Much of the WordApp functionality was handed to new, separate classes. Originally, it parsed the command-line arguments, set-up and stored the dictionary, created the Graphical User Interface (GUI), and created and stored the WordRecord objects. Notice that this approach conflicts with the desired Model-View-Controller design pattern, as there are elements of each component within this single class. Setting-up the dictionary and creating the WordRecord objects should be done in a Controller class; storing these objects should be done in a Model class; and the GUI setup should be done in a View class.

Hence, separate classes were created for each of these tasks – they are discussed in more detail in section 1.2. The WordApp class instead created the Model, View and Controller objects, handled the parsing of arguments, and then let the controller handle the program flow. This functionality is shown in the pseudocode given in listing 1.

Listing 1: Pseudocode for WordApp functionality

```
public static void main(String[] args) {
  // MVC
  Model m = new Model();
  View v = new View();
  Controller c = new Controller(m, v);

  /* PARSE ARGUMENTS */

  // Start the controller
  c.init();
}
```

1.1.2 Score

The Score class essentially remained the same, other than the addition of a 'high score' field. This enabled the record of the highest score achieved within the current instance of the application, on the current level. Synchronized getter and setter methods were created for this field.

1.1.3 WordPanel

The WordPanel class had one vitally important method added to it. It already implemented the Runnable interface, however its run( ) method was empty – it was not actually doing anything when run on a separate thread. The desired action was for it to update/refresh the panel continually, as to visualise the words falling. This was easily done using the repaint( ) method, as shown in listing 2.

Listing 2: Pseudocode for WordPanel's run( ) method

```
@Override
public void run() {
  while(true) this.repaint();
}
```

Additional changes to this class consisted on simple User Interface improvements. The background colour was changed to a dark grey, and a slightly more subdued red was used for the bottom rectangle. Furthermore, for a nice visual effect, the colour of each word was set-up to be functionally dependent on that word's distance to the bottom of the screen. At the top of the screen, the word starts off green. As it moves down, its red component increases, while its green component decreases, until it is completely red at the bottom of the screen. This is shown in pseudocode in listing 3.

Finally, a simple try/catch block was added to handle 'null' words. This was used in other areas of the program, when the word should no longer be displayed. Hence, the catch block is simply empty. This is also shown in listing 3.

Listing 3: Pseudocode for WordPanel's improved painting method

```
for (int i=0; i < noWords; i++) {
    // Determines how close the word is from the bottom
      float c = (float) (words[i].getY()/(maxY*1.2));

```

```
    g.setColor(new Color(c,1-c,0));

    try {
      g.drawString(words[i].getWord(),words[i].getX(),words[i].getY());
    }
    catch (java.lang.NullPointerException e) {}
}
```

1.1.4 WordRecord

Most of the existing code in the WordRecord class remained the same – fields such as its string value, its co-ordinates, and its falling speed. However, a fair amount of additional (and important) functionality was added. Most notably, it became a child of the SwingWorker class, and this enabled each word to operate on a separate thread.

The reason for extending the SwingWorker class, instead of simply implementing the Runnable interface, is worth discussing. The Swing GUI runs on a single thread – called the Event Dispatch Thread (EDT). Hence, anything called on this thread will remain sequential on the EDT. Since the game begins with clicking the 'Start' button in the GUI, the ActionListener method is called on the EDT. This method then starts the WordRecord threads, and they should begin falling. However, when taking the Runnable interface approach, Swing simply puts the work of these threads onto the EDT – and it becomes sequential. Instead, one must indicate to Swing that this work (making the words fall) should not be done on the GUI thread, but instead on a separate one. This is the job of the SwingWorker – it is built into the Swing library for handling big tasks outside of the Swing GUI thread, thus ensuring a fluid UI is maintained.

Some other fields were added to the WordRecord object, including a static (shared between all the words) reference to a Score object, a static integer indicating the level (difficulty) of the game – in essence, this is a scaling for how quickly the words fall, a boolean for handling whether or not a new word should be created once the current word 'dies' (*running*), and a boolean for handling whether or not the current word has reached the end of the screen (*finished*). These fields culminate in the doInBackground( ) method, shown in listing 4. This method is invoked when the thread is created.

Listing 4: Pseudocode for WordRecord's doInBackground( ) method

```
public class WordRecord extends SwingWorker<Boolean, Integer> {

  .
  .
  .

  @Override
  protected Boolean doInBackground() throws Exception {
    // Checks if it should keep falling
    loop: while(running) {
      // If it has reached the end
      if (dropped) {
        score.missedWord(); // Update the score
        resetWord();    // Reset the word
        if (finished) break loop; // If the game is over, stop loop
      } else {
        // Move word down 1 step
        drop(1);
        // Speed determined by level
        try { Thread.sleep(fallingSpeed/level); }
        catch (InterruptedException e) {}   // Keep the compiler happy
      }
    }
    return true;
  }

}
```

Finally, a static counter of the number of threads running was added, and this was used to check if the game was over in the Controller class. The done( ) method is automatically invoked after the doInBackground( ) method finishes. This functionality is shown in listing 5.

Listing 5: Pseudocode for WordRecord's done( ) method

```
public static AtomicInteger threadCounter = new AtomicInteger(0);

@Override
protected void done() {
  super.done();
  // Setting the word to null ensures it is not painted to the screen
  setWord(null);
  // Increment the number of threads that have finished
```

```
    threadCounter.incrementAndGet();
}
```

1.2 Created

The classes mentioned below were created from scratch to meet the assignment requirements.

1.2.1 Model

The Model was essentially handed all the data components that were originally being stored in the WordApp class – including the word dictionary, the WordRecord array, and the Score. Getter and setter methods were added as required. This is all that the Model is responsible for – which aligns with the MVC design pattern. Consider listing 6 for pseudocode summarising the Model. Notice that the getScore( ) method is the only one with the 'synchronized' tag, as it is the only one that might be accessed by multiple threads at a single time.

Listing 6: Pseudocode for the Model class

```
public class Model {
  int noWords;    // Words on the screen at a point in time
  int totalWords;   // Total words that will fall

  // Dictionary of words
  WordDictionary dict = new WordDictionary();
  // Actual array of word (threads)
  WordRecord[] words;
  // Shared indicator of program status
  static volatile boolean done;  // Volatile to ensure freshness
  // Object to keep track of scores
  Score score = new Score();

  public int getNoWords() {...}
  public void setNoWords(int noWords) {...}
  public int getTotalWords() {...}
  public void setTotalWords(int totalWords) {...}
  public WordDictionary getDict() {...}
  public void setDict(WordDictionary dict) {...}
  public WordRecord[] getWords() {...}
  public void setWords(WordRecord[] words) {...}

  synchronized public Score getScore() {...}
}
```

1.2.2 View

The View, in line with the MVC design pattern, is responsible for all GUI-related setup and creation. It holds all the Swing components, and has a constructor for creating the graphic window. Most of this code came directly from the old WordApp class, along with a handful of GUI additions – such as a quit button, and a 'level' (difficulty) selector. A sample of the modified GUI is shown in figure 1.

Also, a JOptionPane was designed to pop up at the end of the game. Code for this is given in listing 7, and a sample of the displayed pop-up is given in figure 2.

Listing 7: Pseudocode for the View's end-of-game pop-up method

```
public void popUpEndOfGame(String scoreString) {
  JOptionPane.showMessageDialog(frame, scoreString
      + "\nLevel: " + levelCombo.getSelectedItem().toString(),
      "Game Over", JOptionPane.PLAIN_MESSAGE);
}
```

1.2.3 ScoreUpdater

The ScoreUpdater is a simple class that implements the Runnable interface, and runs on a separate thread for the lifetime of the game. It contains a reference to the Model and View (as it is essentially an component of the Controller), and has a simple run( ) method. Its pseudocode is given in listing 8.

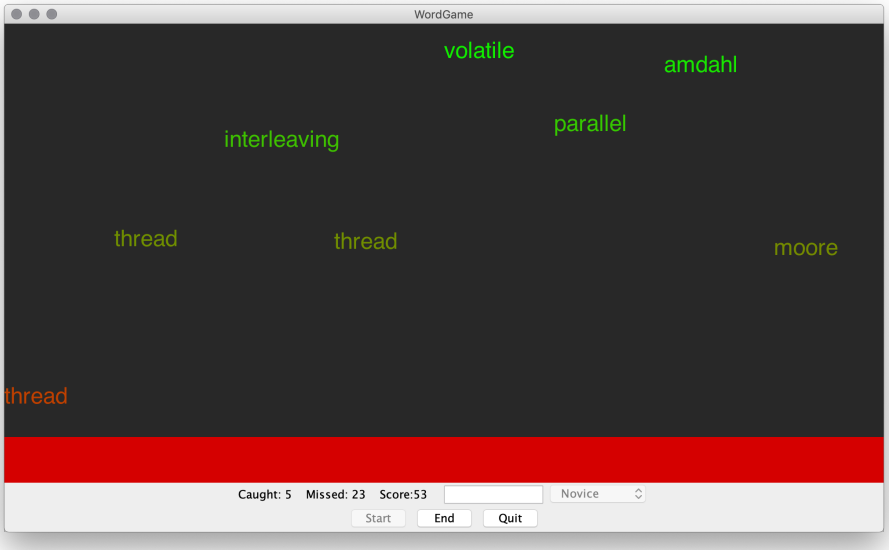

Figure 1: Sample of modified GUI

Figure 2: Sample of end-of-game pop-up message

Listing 8: Pseudocode for the ScoreUpdater class

```
public class ScoreUpdater implements Runnable {
  // Has a reference to both the model and the view
  private Model m;
  private View v;

  /* Constructor */
  // ...

  @Override
  public void run() {
    // Always update the score
    while (true) {
      updateScore();
      // If the total number of words has exceeded the desired amount
      if ((m.getScore().getCaught() + m.getScore().getMissed() + m.getNoWords())
          > (m.getTotalWords()-1)) {
        // Tell the model to finish up
        Model.done = true;
        // Tell the WordRecord class to stop creating new words
        WordRecord.finished = true;
      }
    }
  }

}
```

1.2.4 Controller

The Controller is definitely the most substantial class written for the game. In line with the MVC pattern, it controls all the starting and stopping of threads, all the processing of data, and it communicates with the Model and the View accordingly. The key methods are shown in the pseudocode of listing 9. None of the functional code is actually shown, and some simple methods are omitted, but the essence of the Controller's role in the broader MVC pattern is conveyed. It essentially acts as a mediator between the Model and the View. Further understanding of the MVC pattern can be seen in figure 8.

Listing 9: Pseudocode for the Controller class

```
public class Controller {
  private Model model;
  private View view;

  private ScoreUpdater su;
  Thread scoreUpdateThread;

  Thread refreshViewThread;

  Thread waitForGameOverThread;
  private boolean waitingForGameOver = false;

  public void initView() {
    /* Start refreshView and updateScore threads */
  }

  public void initController() {
    /* Give the WordRecord objects to the View */
    /* Add ActionListeners to buttons, textfield, and level selector. */
  }

  public void startGame() {
    /* Start the word threads */
    /* Start waiting for the game to finish on a new thread */
  }

  public void endGame() {
    /* Stops the WordRecord threads and resets score */
  }

  private void enterText() {
    /* Action Listener for TextBox -- checks input against WordRecords, and acts accordingly */
        }

  public String[] getDictFromFile(String filename) {
    /* Loads words from dictionary file into dictionary object in the Model */
  }

  public void createWords() {
    /* Creates the word threads from a sample of the dictionary */
  }
}
```

2 Concurrency Considerations

2.1 Atomic Variable: Thread Count

There is one instance in the codebase where concurrency is critical and a single variable defines the class state. This situation is ideal for the Atomic Variable approach to thread-safety, as it is most efficient solution.

The situation occurs in the WordRecord class. As a reminder, a WordRecord is a falling word in the game. Each object contains a piece of text, its co-ordinates, and some other fields. Most importantly, each word can be invoked as a separate thread. This invocation causes the word's y-coordinate to change, as it 'falls' down the screen. The thread stops, however, if the word reaches the end of the screen (the red zone), or is matched to the word typed by the user. Once all the word threads have stopped, the game is over and appropriate actions should occur – a pop-up message is displayed, the buttons are enabled/disabled correctly, and the score is reset.

In order to know when all the threads have stopped, a static (shared between all WordRecord variables) counter is incremented as each WordRecord thread dies. This counter is compared to the total number of threads initialised, and when the values are equal, the game is over.

Notice the possible concurrency issue here – suppose two threads die at similar times, and their respective done( ) invocations occur simultaneously. The situation shown in figure 3 might occur. By the time all the threads have died, the threadCount value would be one (or more) less than the total word count – thus, the 'game over' routine would never occur. This is a classic race condition.

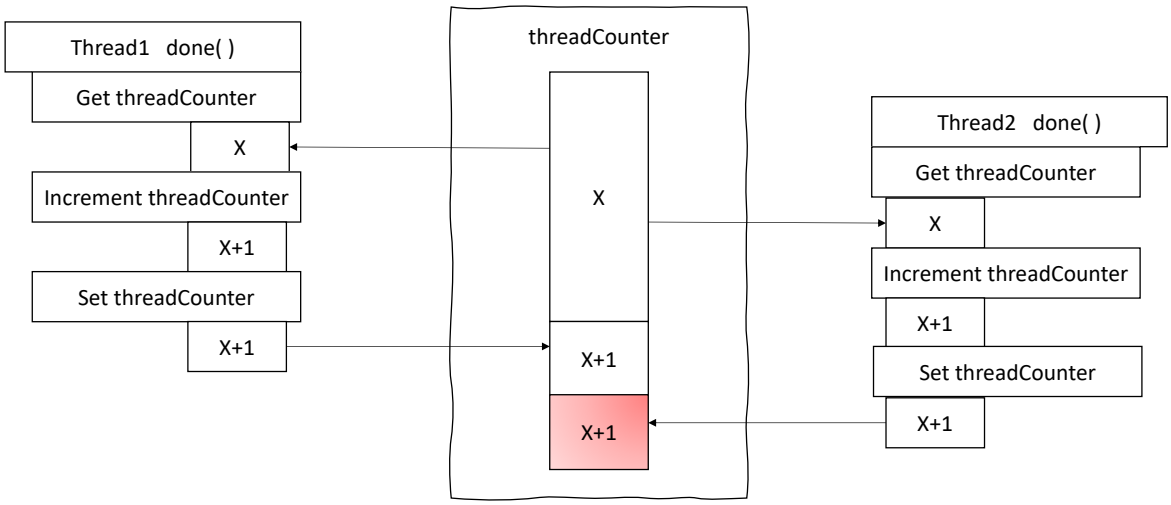

Figure 3: Depiction of possible concurrency issue with threadCount variable [Author's own diagram]

2.2 Synchronized Methods: Score

Thread-safety is also vitally important in the Score class. In this case, however, multiple *classes* will be accessing the same data, possibly performing multiple actions on the data values, and the Atomic Variable approach is hence no longer sufficient. Instead, the synchronized keyword – essentially using a lock – on the getter and setter methods is required. Consider the pseudocode in listing 10 – this shows the Score class' methods, and their respective synchronized modifiers. The exception to this is the toString( ) method – this need not be thread safe because it is only ever called once, from a single thread, when the game is over.

Listing 10: Pseudocode for the Score's getter and setter methods

```
public class Score {
  private int missedWords;
  private int caughtWords;
  private int gameScore;
  private int highScore;

  synchronized public int getMissed() { return missedWords; }

  synchronized public int getCaught() { return caughtWords; }

  synchronized public int getScore() { return gameScore; }

  synchronized public void updateHighScore() {
    if (gameScore > highScore) highScore = gameScore;
  }

  synchronized public void resetHighScore() { highScore = 0; }

  synchronized public void missedWord() { missedWords++; }

  synchronized public void caughtWord(int length) {
    caughtWords++;
    gameScore+=length;
  }

  synchronized public void resetScore() {
    caughtWords=0;
    missedWords=0;
    gameScore=0;
  }

  public String toString() {
    return "Caught: " + caughtWords +
        "\nMissed: " + missedWords +
        "\nScore: " + gameScore +
```

```
36            "\nHigh Score: " + highScore;
37    }
38 }
```

The synchronized keyword performs as depicted in figure 4.

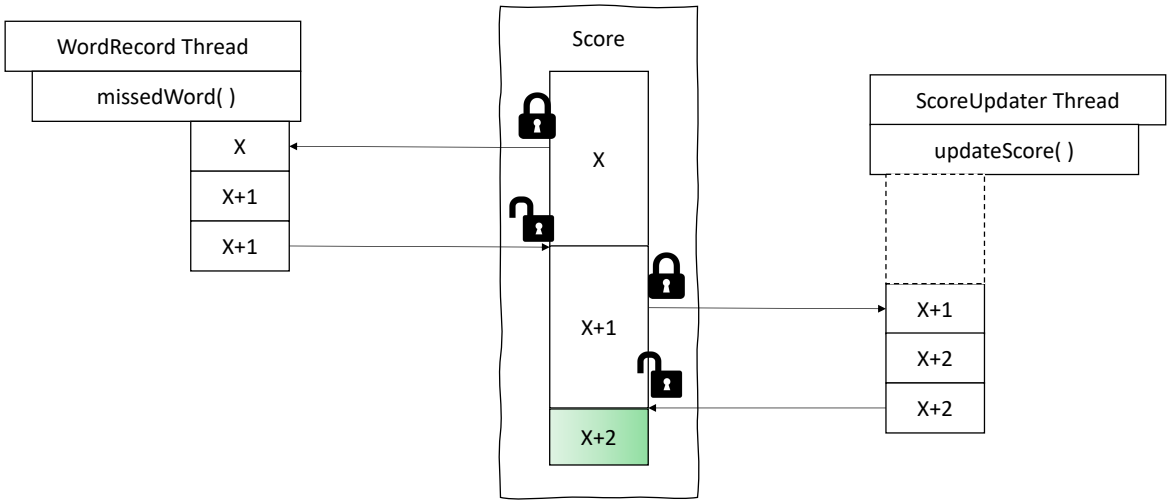

Figure 4: Depiction of locking using the synchronized keyword [Author's own diagram]

2.3 Event-Dispatch Thread: Swing

The Swing GUI uses a single thread – the Event Dispatch Thread – because many of its objects are inherently not thread-safe. This is not problematic if one operates serially, but when one creates separate threads that alter the GUI in some way, problems can creep in. In order to ensure thread-safety for the game, the WordRecord threads (which use the SwingWorker library) do not directly access the GUI. Instead, they update the Model, and then the Controller in turn updates the GUI – when it is safe to do so (on the EDT thread).

2.4 Volatile Variables: 'Done' flag

Within the model class, there is a shared boolean 'done'. This is used to check if the game is still going and words should still be falling. To guarantee each time it is accessed (by various classes) it reflects the true state of the program, the volatile keyword is used. This ensures the variable is not cached, and must be reloaded from memory every time it is used. This assists in correct operation.

2.5 No thread-safety required

There are certain variables in this game that do *not* require thread-safety measures, simply because the nature of the program guarantees no conflict occurring. For example, within the WordRecord class, the getter and setter methods for each word's coordinates, text, and falling-speed need not be synchronized – they are only ever accessed from a single thread (from itself as a SwingWorker thread).

2.6 Preventing deadlock & Ensuring liveness

It is worth noting that the minimum amount of required thread-safety measures were put in place to ensure correct concurrency. Moreover, the synchronized methods are all very short – only involving simple get- and set- methods. Thus, deadlock will not occur – purely due to the simplicity of the associated locks. Furthermore, the liveness of data is ensured considering the 'separate' nature of the work that the threads are doing. These things are affirmed by looking at the actual threads that are created in the program – see figure 5. Notice how there is minimal dependency between the respective threads.

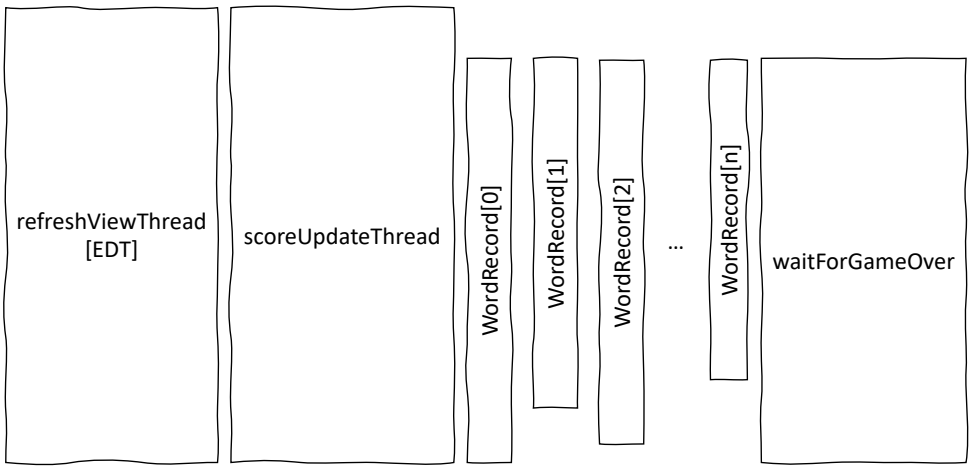

Figure 5: Depiction of threads in the program [Author's own diagram]

3 Validation

To validate the correct operation of the game, three approaches were taken.

Firstly, where concurrency features were used, they were removed – and the effect of this was noted. For example, all the synchronized keywords were removed from the Score class. The game was run with the total number of words falling set to 100. Figure 6a shows a game result before the keywords were removed, and figure 6b shows a game result afterwards. Notice how the total number of words (caught + missed) is correct in the former, but incorrect in the latter. One can thus infer the correct effect of these concurrency tools.

(a) Before removing synchronized keywords

(b) After removing synchronized keywords

Figure 6: Game results before- and after- removing the synchronized keywords with a total number of words = 100

Another example tested was changing the Atomic Integer mentioned in section 2.1 to a normal integer field. Multiple iterations were performed, and several times the game would freeze once the final word had fallen – all the threads had finished, and yet the threadCount was an incorrect value due to race conditions.

The second approach taken to validate the correctness of the system entailed using the correct codebase (unlike the first approach) and simply testing corner cases. For example, a dictionary with a single word allows the testing of matching multiple words at a single point in time – see figure 7.

The third approach taken to validate the system's correctness is via theoretical considerations. Through careful analysis of the shared resources, the respective threads, and the latter accessing the former, an sufficient level of confidence can be maintained in the thread-safety of the codebase.

Finally, it is worth noting that the validation of thread-safety is an inherently painful process – due to the uncertain nature of thread execution, bugs come and go sporadically. Nevertheless, the discussed methods are a good start in ensuring a robust game.

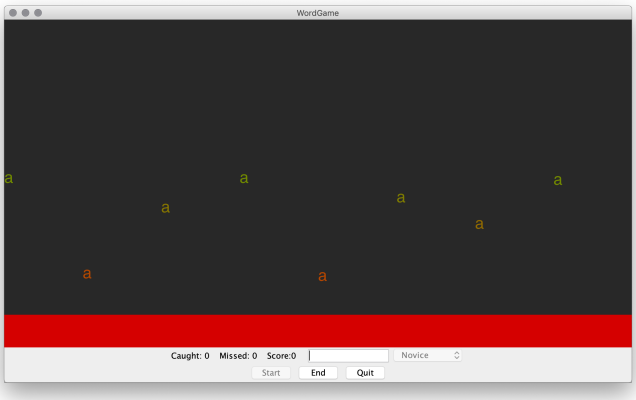

Figure 7: Testing the game with a dictionary containing a single word ('a')

4 MVC Design Pattern

The Model-View-Controller (MVC) pattern was used for the game. Explicitly named classes emphasize the respective components of this pattern – the Model, View and Controller classes. Figure 8 shows how the other designed classes fit into this pattern.

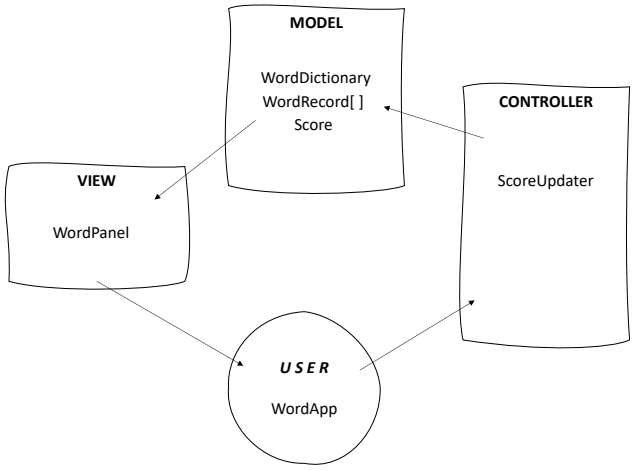

Figure 8: Overview of MVC design pattern structure for this game [Author's own diagram]

Much of the design considerations in section 1.2 concern the alignment of the created classes with the MVC pattern, and should be referred to now. In short, however, all data is stored in the Model (dictionary, word records, etc.), all GUI components are in the View (JPanels, JButtons, etc.), and all functionality is performed by the Controller. The user initialises the controller through the WordApp.

Much inspiration was drawn from here for the MVC design, and serves as a good reference.

5 Additional

There are three improvements that were added to the scope of the assignment which could deserve additional credit.

5.1 Theme: Word Colours

This improvement to the user interface experience is described in section 1.1.3, with code shown in listing 3. An example of the change is clearly visible in figure 1.

5.2 High Score Record

As mentioned in section 1.1.2, a high score feature was added to the game. This enables the user to keep track of their highest recorded score within the current game instance, on the current level that they are playing. The high score result can be seen in the pop-up in figure 2.

5.3 Levels of Difficulty

A difficulty selector was added to the game, and this selector can be seen to the right of the text-entry box (in figure 1 for example). This selector is enabled when a game is not in progress, and there are four options: Novice, Master, Expert, Superhero. Each option changes the amount of sleeping time between the WordRecord moving downwards – this directly influences the perceived 'falling speed'.

This functionality is achieved by the code shown in listing 11. Since the above options have indices 0, 1, 2, 3 in the JComboBox, the respective 'level' values are 50, 150, 250, 350. As the level number increases, the thread's sleep time decreases $\left(\frac{1}{\text{level}}\right)$, and the effective speed increases.

Listing 11: Pseudocode for the game's difficulty selection

```
// Within the WordRecord class:
Thread.sleep(fallingSpeed/level);

// Within the Controller class:
WordRecord.level = 50 + ((view.getLevelCombo().getSelectedIndex())*100);
```