PCP ASSIGNMENT TWO

A MULTI-THREADED WORD GAME

COMPLETED BY

DAVID PULLINGER

The University of Cape Town

CSC2002S
6 September 2021

TABLE OF CONTENTS

Introduction

The aim of this assignment was to create a multithreaded word game which ensured thread safety and sufficient concurrency. The game is simple: it requires the player to type falling words into a text field before they hit the bottom of the screen. If the player correctly types one of the words that are falling, their score is incremented by the length of the typed word. The game ends when a specified number of words have "dropped" or been "caught" (correctly typed by the user).

This report presents my implementation of this game and how it ensured thread safety, thread synchronization and liveness.

Development

Program Structure

Before discussing what methods and classes were added to the skeleton code in order for the game to function correctly, it is important to visualize how the various classes interacted. To this end, the following UML class diagram has been constructed and discussed below:

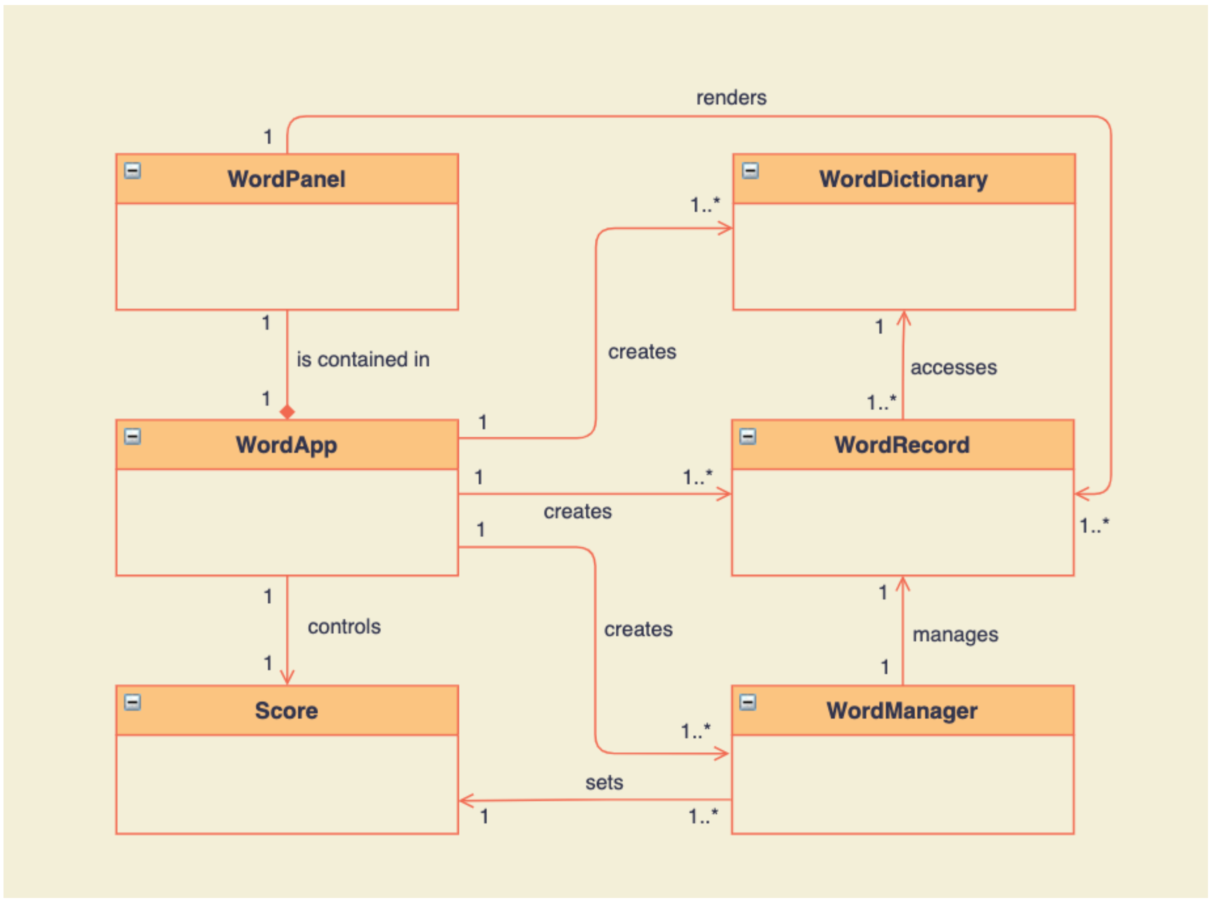

Figure 1: UML diagram displaying relationships between classes

WordApp

The main controller of the application was the WordApp, which executed the following steps in succession:

1. It read in the words to be used for the duration of the game from a text file
2. It used these words to initialize a WordDictionary
3. It created and displayed the main GUI of the game, including creating a WordPanel
4. It created and started a Thread which would be in charge of refreshing the WordPanel
5. It created and initialized an array of WordRecords
6. It created and started an array of WordManager Threads which would each manage all aspects of a single WordRecord

The above steps were all executed in the beginning of the application. However, throughout the game, the WordApp class still controlled what had to be updated when the player pressed any buttons or submitted text via the text field. It also handled all game logic, such as ending and restarting the game.

WordDictionary

The WordDictionary class was very simple. A WordDictionary object was created when the game started, using words read in from a text file. The WordDictionary class held these words in a local array which allowed other classes (specifically the WordRecord class) to access a random word when requested.

WordRecord

The WordRecord class was the games representation of a word, and it held and modified the position, falling speed and text of the word. The WordRecord could be "reset" once it fell into the elimination zone at the bottom of the screen. When it was reset, it would reset its y position, generate a random falling speed and use the WordDictionary class to generate a new random text.

WordPanel

The WordPanel class extended the JPanel class, and it was the component of the main GUI which would actually draw the WordRecords to the screen. It implemented the Runnable interface so that a Thread could be started which would continuously re render the WordPanel GUI.

WordManager

The WordManager class controlled the "lifecycle" of the falling WordRecords. In this class's `run` method, it would continuously drop the WordRecord (increment its y coordinate) and perform various checks. These checks, which are described in better detail below, ensured that the WordRecord was destroyed or reset when it was supposed to. The WordManager class implemented the Runnable interface and so it was possible to create a Thread which would execute the WordManager's `run` method.

Score

The Score class was the games representation of a score, and it held and modified various variables, namely the missed words, caught words and game score (defined by the length of the caught words).

Note

Hopefully the above description of the classes coupled with the UML diagram in Figure 1 are sufficient for the reader to understand the program structure, so no idiosyncrasies arise later in this report.

Added Classes

WordManager

A single class, WordManager, was added to the skeleton code. As mentioned above, this class implemented the Runnable interface (so that it could be executed on a new Thread), and it controlled the "lifecycle" of the falling WordRecords. The following UML class diagram highlights the various methods and variables of the WordManager class:

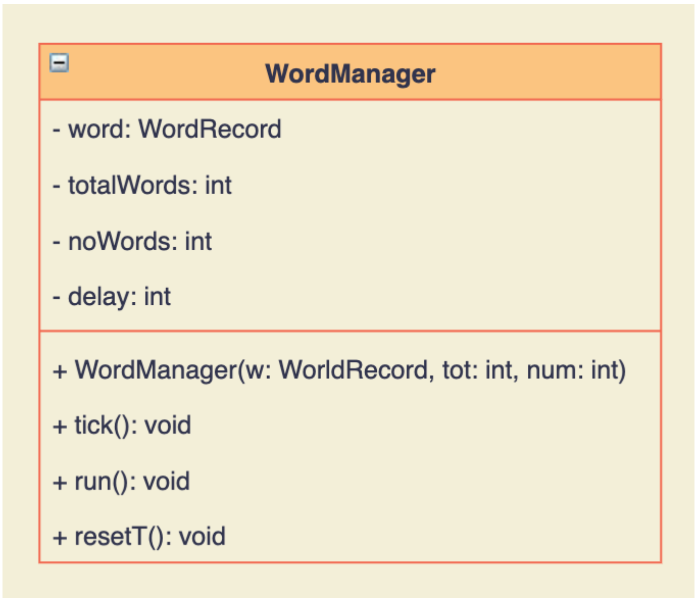

Figure 2: UML diagram showing WordManager class structure

A single WordManager controlled a single WordRecord, called `word`. When the WordApp started the WordManager Threads, their `run` method would be called. The `run` method would be continuously executed and, if the game was playing (it wasn't paused), the `tick` function would be called. The `run` method would then cause the Thread to sleep (temporarily cease execution) for a calculated amount of time, so that the words would **smoothly** fall at the **correct** speed. This is also where the `delay` variable is used.

The `tick` function handled what should happen each time the WordRecord was updated. It would first check if the word had been dropped (it had hit the elimination zone at the bottom of the screen). If it had, it would then check if the word should be reset (i.e., the maximum number of words for the game, `totalWords`, had not been reached). If it should be reset, it would call the `resetWord` function of the WordRecord class and continue. If it should not be reset, it would "deactivate" the word so it would no longer `tick` (be updated). If, however, the word had not been dropped in the first place, it would check if the word matched the word that the user had entered into the text field. If it did, the word would be checked and reset in the same way as mentioned above. It would also reset the `currentWord` that had been entered by the player. If the word had not been caught or dropped, it would simply `drop` (y coordinate would be incremented). This is all laid out in the pseudocode below:

```
def tick:
    If word is deactivated
        return
    Else if word is dropped
        missedWord()
        If word can be reset
            resetWord()
        Else
            deactivateWord()
            If this is the last word on screen
                endgame("Oof, couldn't get the last one")
            End if
        End if
    Else if word is equal to user input
        caughtWord()
        clearUserInput()
        If word can be reset
            resetWord()
        Else
            deactivateWord()
            If this is the last word on screen
                endgame("Well done on getting the last one")
            End if
        End if
    Else
        drop()
```

While it may not be pleasant to look at, all these checks had to be performed in order to ensure that the WordRecord was reset when it was supposed to and that the game ended when it had to.

Besides this `tick` function, the WordManager class had no meaningful methods, besides the `resetT` function. This function was implemented so that the WordApp class could tell the WordManagers to reset the WordRecord they were managing when the game was ended or restarted. It simply called the `resetWord` and `activate` methods of the WordRecord class (if the game had ended, all WordRecord's would obviously be deactivated and thus had to be reactivated and allowed to `tick`).

Added Methods or Variables

<u>WordApp</u>

WordApp was the class that was modified the most since it was the main controller of the game. The first things that were changed were the ActionListeners that were added to all the buttons. Several buttons had to be implemented: a Start Game button, End Game button and Quit button. As an extension to these and for a (slightly) better game experience, a Pause/Resume button and Dark Mode button were also added.

Since the WordManager threads had already been started, the Start Game button would simply set a boolean called `gameIsPlaying` to true. This would allow the WordManager threads to call the `tick` function and thus fall down the screen. Since the End Game and Pause/Resume buttons were disabled by default, the Start Game button would also enable these.

The EndGame button would call `endGame` which would pause the game (set `gameIsPlaying` to false) and cause a JOptionPane to be created and displayed, which would allow the player to restart, end, or continue the game. If the game was ended, the application would simply close. If the game was to be continued, `gameIsPlaying` would simply be set to true and thus the WordManagers would be able to continue calling the `tick` function. If the game was restarted, the WordApp class would simply call its `reset` method, which would reset the WordManager threads and reset the score.

The Pause/Resume button would toggle its text when pressed (from "Pause" to "Resume" and vice-versa) and would toggle the `gameIsPlaying` boolean. It would then enable or disable the Start Game button, End Game button and text field depending on what the state of `gameIsPlaying` was.

The DarkMode button would also toggle its text when pressed (from "Dark Mode" to "Light Mode" and vice-versa). It would then call a function called `toggleAppearance` which is discussed below.

Finally, the Quit button would simply close the application.

Besides the functions mentioned above (namely, `endGame`, `reset` and `toggleAppearance`), three other functions were added, called `getCurrentWord`, `setCurrentWord` and `clearCurrentWord` which would return, set and clear the word that the user had most recently entered. The get and clear functions were called by the WordManager Threads as discussed above. These functions were all `synchronized` since they were prone to thread interference and memory inconsistency. This is further discussed in the Concurrency section below.

Finally, the `setCurrentWord` function was called by the text field when the user hit the Enter button on their keyboard after typing the word. Again, the synchronization of the `currentWord` variable is discussed in the Concurrency section below.

WordPanel
The WordPanel was modified to continuously update itself. This was done by calling the `repaint` method of the Java Component class in the WordPanel's `run` method. The WordPanel class had to implement this `run` method since, as mentioned above, it implemented the Runnable interface. Thus, when WordApp started a Thread with WordPanel, it called its `run` method which would, in turn continuously update the GUI, thus redrawing all the WordRecords. It would also re render the score text fields that were initialized by WordApp.

The last thing that the WordPanel implemented was for testing purposes. It would continuously detect deadlocks by using Java 5's ThreadMXBean API. This API allows the developer to detect any deadlocked threads and print them out. This is discussed in the Testing section below.

WordRecord
The last class that was modified was the WordRecord class. A Boolean called `active` was added, as well as synchronized methods called `deactivate`, `activate`, and `isActive`, which would respectively set and get the `active` variable. This Boolean was added to allow the WordManager to decide whether a WordRecord should be allowed to fall down the screen.

The methods that would set and get the `active` variable were made synchronized since they were prone to thread interference and memory inconsistency. This is further discussed in the Concurrency section below.

Concurrency

Features Used and Implementation in Code

WordApp
The WordApp class made use of two concurrency features: a volatile Boolean (`gameIsPlaying`) and 3 synchronized methods which were able to modify the `currentWord` variable.

A variable is made volatile if it is constantly changing during runtime and should thus never be caches by the compiler for any reason. The Boolean `gameIsPlaying` had to be volatile, as it was constantly being accessed by the various WordManager threads, and they needed the absolute latest state of the game. For example, the player could press the pause button, thus setting `gameIsPlaying` to false. However, if the WordManager Threads have a stale value of `gameIsPlaying` where it is still true, they will continue to animate the falling of the words, if for a short while. This is obviously not something that the player wants, and so making this variable volatile ensures that this will not happen. The `gameIsPlaying` variable is only ever modified by the WordApp class and so it did not need to be set via a synchronized method.

The `currentWord` variable on the other hand, was continuously being accessed and modified by all the WordManager threads, as well as the WordApp. Therefore, methods to get and set it were synchronized, ensuring that no Threads would read or modify it at the same time, thus ensuring there would be no race conditions. The entire bodies of the functions were synchronized, as the entire function was a critical section, and no sub part could be done asynchronously.

<u>WordDictionary</u>
The WordDictionary made use of a synchronized method as a concurrency feature. It's synchronized method was called `getNewWord` and it would return a random word from the array of words that it held. Since this method was also constantly being called by the WordManager threads, it had to be synchronized.

If two threads simultaneously tried to generate a new random word, it could lead to bad interleaving. The method had two statements, one where it generated a random integer and stored it in a variable, and a second where it returned the word at the index of this random integer. So, Thread A could generate a random integer while Thread B could interleave and get the word that Thread A was supposed to.

Since the method was synchronized however, such interleaving would not occur. In the above scenario, Thread B would have to wait for Thread A to release its lock on the method before it could execute it. This ensured thread safety in the WordDictionary class.

<u>WordRecord</u>
The WordRecord class had synchronized methods for all getters and setters in the skeleton code. However, through the implementation of the WordManager class, they were made redundant. This is because each WordManager would manage a single WordRecord, and so when it called any methods of the WordRecord class, it would only be in the scope of the WordRecord that it was managing. In other words, there were no shared variables (from the WordRecord class) between the WordManager Threads.

<u>WordPanel and Word Manager</u>
These classes had no implementations of concurrency as they were not being accessed by external classes in the same way that the WordDictionary was, for instance.

Score

The Score class made use of synchronized methods as a concurrency feature. All its methods were either getters and setters and since there was a single Score object for the entire game (which was held in the WordApp class), these had to be synchronized to avoid any race conditions or bad interleaving.

The Score was, again, being continuously accessed by numerous WordManager Threads and so it had to ensure that only a single Thread could access or modify any of its members at any given time. For example, if Thread A and Thread B were managing words that had entered the elimination zone nearly simultaneously, it could happen that Thread A would modify the Score object to say that it was no longer active on the screen. But, before it could **also** let the Score object know that it had been dropped, Thread B would read the Score's state and see that it could potentially be reset (the total number of words had not been reached yet). Therefore, even though the game should have ended, Thread B would reset and begin dropping another word down the screen.

Fortunately, since all the methods were synchronized, only one Thread would be allowed to modify or read the Score object at any time. This ensured (together with testing, mentioned below) that the Score object was being safely accessed by multiple Threads and was thus, thread safe.

Swing Library

Unfortunately, the Swing Library is not thread safe. Therefore, to ensure that proper concurrency was maintained, I was sure to keep the model and controller separate from the view, so that the Swing Library had no state or complex execution; it simply had to render the GUI elements. Also, there was only ever a single thread that was accessing the two GUIs; the main Thread handled the MainApp GUI while a separate Thread was created to handle the WordPanel GUI.

Note

While I have mentioned thread safety and synchronization, I have not mentioned deadlock. Deadlock was not an issue since I never had methods which would need to acquire locks that were already acquired. I therefore had no deadlock and I attempted to verify this (mentioned below).

Validation

Score

The first validation that was done was to ensure that the thread synchronization implemented in the Score class ensured thread safety. To this end, the application was run with ridiculous parameters, such as the number of words falling being set to 100 and the total number of words being set to 1000. This was done to detect any incorrect behavior when many Threads attempt to modify/read from the Score object at once. Therefore, the application was run with the above-mentioned parameters and all

WordRecords were given the same speed so that they entered the elimination zone almost instantaneously.

So, to test if the score was working, it remained to check that the number of missedWords + typedWords = totalWords. Indeed, it was. To ensure that this was not a fluke, the application was run multiple times and observed the same result.

Furthermore, the methods of the Score class were made not synchronized and, sure enough, missedWords + typedWords != totalWords, which ensured that the thread synchronizsion was running as expected.

Deadlock

It was already strongly assumed that no deadlock was occurring since the code had been designed in such a way that a scenario where two threads were waiting for each other's locks was just not possible. However, to ensure that this was the case, the `findDeadlockedThreads` function of Java's ThreadMXBean API was run on the WordPanel Thread so that it would continuously check whether a deadlock was occurring. No deadlocked threads were reported.

Race conditions

As mentioned above, the relevant classes were made synchronized and so race conditions (multiple threads attempting to read/modify the same resource) were not possible as the Thread had to have a lock on the entire object.

MVC Framework

The Model-View-Controller framework is an architectural pattern that separates an application into three main logical components Model, View, and Controller. This application conformed to this pattern by splitting the classes into the logical components as follows:

Model

The classes which fell into the Model component were the Score, WordDictionary and WordRecord classes. This is because these classes were the data components of the application and had no complicated logic. They did not modify any other classes but were simply used as a representation of the data that could be used in the application.

View

The class which fell into the View component was the WordPanel class. This is because it simply pulled the application's state from the Controller components and represented this state to the player. More specifically, the WordPanel class was initialized via the WordApp with the WordRecords and each time it was updated, it would fetch this data (state) from memory and display it on a GUI by reading the WordRecords position and text. It did not modify the data, but simply broke it down into its various components and displayed this on a GUI.

Controller

The classes which fell into the Controller components were the WordApp and WordManager classes. This is because these classes together were able to handle the players interaction with the game (typing in the text field and click buttons), and subsequently alter the Model which would ultimately be rendered to the View to reflect the user's input (e.g., a word disappearing once it has been correctly typed). These classes did not update the View themselves; rather, they updated the Model which would in turn be fetched and reflected by the View, thus conforming with the MVC architectural pattern.

Extra Credit

Pause/Resume Button

While it may not have been very far out of scope, the Pause/Resume button did add an extra level of useful user interaction, in turn improving the usability of the game. It was not difficult to implement although it did imply having to rethink under what conditions the falling WordRecords should be stopped.

Dark/Light Mode Button

This was an enjoyable feature to implement. A custom dark grey color was used for the dark background (rgb(24,26,27)) as well as a custom red color for the elimination zone (rgb(222,10,2)).

All that occurred when the player pressed the button is that the text and background colors would switch on all components, except the text field which was kept white.

Ending Dialog and WPM

The dialog at the end of the game had an ImageIcon to make it look slightly better. I also added timing to record how long the game was (excluding paused time) so that I could report to the player how quick their typing speed was.

High Score

I implemented a high score system which would read and store previous scores (WPM typing speeds) and write new scores. It also tells the player what their placement is overall. This has an InputMismatchException error which is discussed in the README.md file in the root directory of this project.