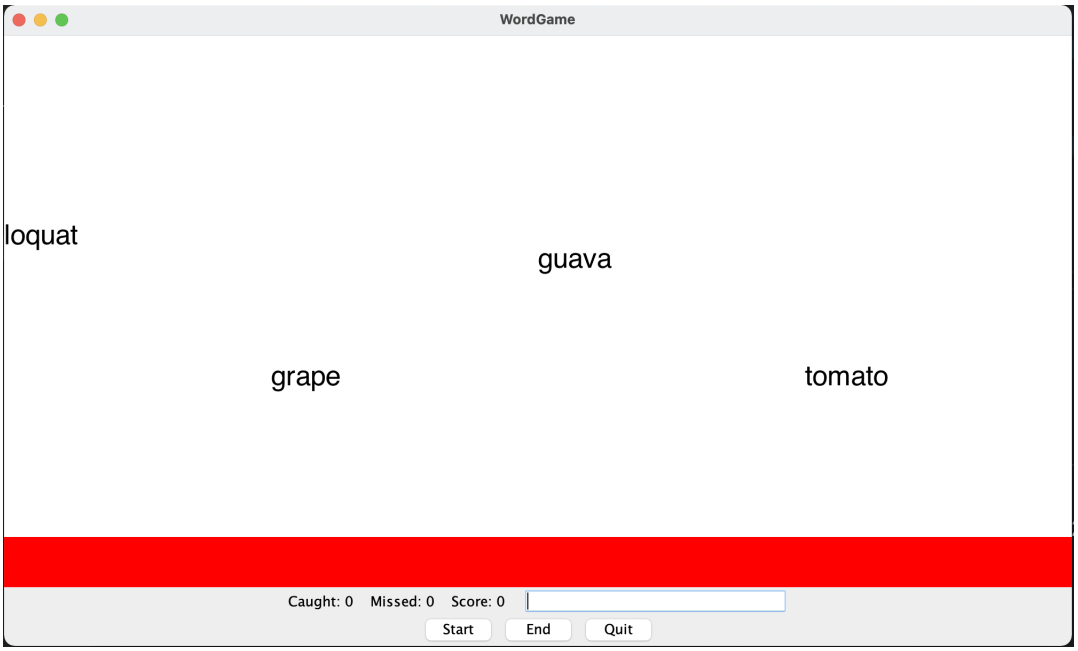
CSC2002S Assignment 2 Write-Up

Jack Montgomery (MNTJAC003)

1 September 2021



Contents

1 Description of Classes

This assignment required the creation of new classes to add to the skeleton as well as added concurrent and functional features to ensure the program runs smoothly as possible.

1.1 WordApp

The *WordApp* class created volatile flags in order for the program to know what state it is in. This assisted when determine whether to clear the current words or start again. The contents of the *SetUpGUI* method was left unchanged until the creation of the buttons and text field. The text field in the App works by using an action listener when the enter button is pressed while in the text field. It then checks if the word entered matches any word thread created. (Word threads will be discussed later) If so the word will be removed and a correct sound will play. It will then check the volatile flag *finished*, if not finished the text field requests focus. The *Start* button is where the program begins in a sense. The program will sit dormant if this button isn't pressed. Once pressed the program will set the flag *started* [1] to true and check if it is in the initial state. If so, the program will continue, if not the program will end its threads. This is because for it to not be an initial run and for the start button to be pressed, it must have been pressed while words are falling. So the game "restarts", but does not stop. The threads are then made, the flags are all set to false, focused is put pack on the text field and then the game is started by calling the *run* method in the *WordPanel* class.

The second button is the *End* button. This button will allow the user to start anther game. It suspends all the threads but does not start until the start button is pressed. The code reset all the scores and sets the flag *ended* and *initialRun* to true, as well as *started* to false. This stops the program but allows it be carry on as if the user is starting a new game. The last button is simply the *Quit* button which ends the program.

[1] This flag prevents the user from entering text before the game has started

The next two methods were added to assist with the functionality that comes when a game is finished. The first is the *closeGUI* method. This method is more of a hide GUI method. It changes the visibility of the GUI to false. This is because when the game finishes you have the option to play again. While the results window is displayed the game GUI is hidden but not disposed of. That then introduces the next method *restartGUI*. This method resets the score and makes the GUI visible again. In order for the user to play again. The next method is *getDictFromFile* method which is unchanged and a method was created *makeWordArray* which was simply to make the creation of the word array cleaner. The last method is *main*. This method controls the command-line input and then creates the word dictionary. Other than the volatile variables, there are not many concurrent features needed for this class because its methods are used for handling user input which is inherently serial. The next few classes are where the concurrency of the program is implemented.

1.2 WordPanel

The constructor of the *WordPanel* class and the *paintComponent* method remained fairly unchanged. Aside from the creation of a timer. This timer was for the scheduling of tasks as well as making the GUI thread-safe. The timer was constructed so that the main thread checked the validity of the state of the program very certain time interval. This check consisted of validating the program, then checking that the volatile flag *finished* was false. If it was true then all the *WordThreads* would be stopped and the *EndGUI* would be shown to the user. The *run* method consisted of starting up each *WordThread* the number of threads would be determined by user input. The method then started each thread by calling the *start*. The last two methods are *endThreads* and *WordThreads*. *TestWord* simply checks that the String passed is equal to any of the words in the threads. The *endThreads* ends all the *WordThreads* in execution. Once again this class does not have many concurrent components as it doesn't have multiple threads running through it. Only the main thread will see this class. However, this class created the parallel component of this program.

1.3 WordThread

The *WordThread* class is the first class outside the skeleton created. The class had one constructor the the type *WordRecord*. Then *WordThread* places each word on its own thread. The most important aspect of this class is the timer. This timer allowed each change to the GUI to be run through the event dispatcher thread which made the *Swing* library thread-safe. When the *ActionPerformed* is triggered the program checks whether its String was empty. This would only happen if there were too many threads created compared to the user input. If so, the thread would be ended. It would also check that the volatile flag *ended* was false, if true it would end the thread. After these validity checks were done, the thread would carry like in the normal course of the game. This means that each thread would drop and check if its in the red. If it is then the *missed* score is incremented and the error sound is played. Then it checks whether there are too many words in the screen. If so it stops the current thread. If not, it resets the word. The rest of the methods in the class are fairly simple. The *UpdateTotals* method simply changes the text in the GUI to account for the changes in score. The *match* method tests whether a String passed through is equal to the word in the current thread. If so it updates the totals. The *run* method starts the timer for each thread and is invoked when we call *start* in the *WordPanel* class. Then *endThread* ends the current thread by stopping the timer. We have seen the timer used as a concurrency feature to ensure thread safety in this class.

1.4 WordRecord

The *WordRecord* also employs concurrency features. This is needed because each thread that is being created needs these methods in order for the game to function as it should. The *WordThead* class creates the parallel component but the *WordRecord* class does the work for each thread. This class moves the words down on the screen, it allows the threads to get and set positions, Strings and speeds. And to ensure thread safety all of these methods need to be synchronized. The contents of each method have remained largely unchanged until the *matchWord* method which has been changed to increment the score and do an validity test. This method , *validTest* checks whether the number of total words is less than the words in the screen plus the total score (missed + caught). If this is false, then it has not been exceeded. If true, it checks whether the total score is equal to the total number of number of words fall. In which case the game is finished, so it change the volatile flag *finished* to true. If the score total is not equal to the number of words to fall then the words in play have exceeded and the method returns true.

1.5 Score

The *score* class has been modified in order to maintain thread safety. This was done by modifying all methods to be synchronized other than the *resetScore* method and changing the score variables to atomic. This is because we can only have threads accessing these methods one at a time. This is because the methods in this class all call on shared variables, the scores. The use of shared variables can create situations where there are dirty reads or inconsistent analysis. The use of synchronized methods as well as atomic variables means that these problems will not occur.

1.6 Endgui

The *Endgui* class was made to create a results window when the game was finished. This window shows the scores in the game and gives the user the option to play again or to close the window. Play again will start the game again with the same parameters and *Close* will end the whole program.

1.7 AppAudio

The *AppAudio* class has two methods that play two different noises. One noise for every word that has been entered correctly the other if the entered text does not match any words on the screen.

2 Concurrency Features

2.1 Thread Safety

The main concerns for the thread safety of this program are the shared variables and the *Swing* library. The score variables were of most concern. [2] This is because these variables were shared as well as mutable. Variables that are immutable and not shared do not threat the safety of the threads. Thread safety was ensured by synchronizing each method that was reading/writing to these variables. In other words, variables that were getting or setting these variables. As well as these variables being declared as atomic. This meant that the variables would be stored in main memory instead of in the cache of each thread. It also ensures only one thread can access it at a time. In this way we can ensure that when the data is in a mutable state it will not be subject to data races or bad interleaving.

Thread Safety for the *Swing* class was done by the *Swing.Timer* library. This meant that every action that needed to be performed to the GUI was performed through the Event Dispatches Thread. This was necessary because the *java.swing* class is not thread safe. Making the changes through the *actionPerformed* event handler made sure that it was executed on the Event Dispatcher Thread which ensures thread safety.

2.2 Thread Synchronization

Thread synchronization was achieved through the use of the *synchronized* keyword. This meant that every method that was getting or setting a shared variable did this in a thread synchronized fashion. The variables that did not need synchronization have been discussed.

[2] The volatile flags were not of concern for safety because flags could not be changed back in the method. So threads could not interleave and leave the program in a state with an incorrect flag

2.3 Liveness

The main features of liveness is the absence of dead-lock (which will be discusses) and starvation. Starvation occurs threads cannot gain regular access to a shared variables, in this case the score variables. This was avoided starvation in this program by creating short critical sections that are synchronized. This meant that threads never held locks for long, because of this no resource was held for too long meaning that each thread has equal access to the variables they needed. Which created the liveness and smooth flow of the program.

2.4 Deadlock

The absence of deadlock was achieved by the acquisition one lock for each operation. This was particularly pertinent in the *Score* class. There could be a situating where a thread a thread wanted to read the total score and other thread has the lock for the *missed* or *caught* counter. This could have resulted in a deadlock but by making each get and set operation atomic we avoid this situation.

3 Testing

Testing of parallel program can be very difficult. Because deadlocks can happen in very particular thread execution order that won't happen for the first 10000 running's of the program it is difficult to know when exactly your program is race-condition free. However, we can try get push the program to its limits and see how it behaves. Testing was a few different steps.

1. The first step was a manual one. It required a detail evaluation of each method trying to determine if there was any situation conceivable where there would be race-condition would occur. Once we determined that in our capacity we could not see a dead lock situation we moved on to the empirical tests.

2. The first empirical tests was to test the volatile flags, namely *ended* and *exceeded*. This done by using more words then the program had handled before, then missing words and catching words. This was a test of the program capacity to handle exceeded words as well as testing if there were any missing writes or reads to the scores.

3. The second test was concerning the ability of the program to reset or restart. The testing consisted again of a large dictionary. We let words fall at similar speeds to test the concurrent ability of the misses counter. As a second part to this test the *Start* and *End* buttons were operational. We were arbitrarily pressing them when the program was in different states to see if the buttons reacted as expected.

By repeating these three steps to test each model/modification that was made we ensured that the game was valid.

4 Model-View-Controller

The model-view-controller design can be clear seen by the operations performed by each class.

The "controller" in this program would be the *WordApp* as well as the *WordPanel* classes. These classes interact with the user as well as control the manipulation and restricted visibility of information in the program. This control would be the different action when button are pressed and when text is typed. It is up to these classes to send these requests off to the logical classes in the program. These logical classes make up the "model" aspect in the design. These classes consist of the *WordThread*, *WordDictionary*, *WordRecord* and *Score* classes. These classes perform the logical operation based on the information send by the controller classes. After this logical manipulation the necessary information is then updated. After being updated the controller classes will read it and allow the user to view the necessary information.

This is perhaps best understood through an example, take the user typing a correct word in the text field:

1. The *WordApp* class displays the current GUI and handles the input by the user. It then sends the word to *WordPanel*

2. *WordPanel* then checks each word against each *WordThread* that has been created. It does this by calling a *match* method on an array of *WordThreads*

3. *WordThread* checks if the word is equal to its *WordRecord* insatnce varible. It does this by calling a match method in the *WordRecord* class.

4. The *WordRecord* class tests if the inputted word is equal to its String instance variable. Since it is, it then calling an increment on the *Score* class *caught* atomic instance variable. The value is then updated.

5. The Boolean true is then rolled back to the *WordApp* class

6. Then the updated value of the *caught* varible is shown.

In this example, it is clear to see the MVC design in action.

5 Additional Features

This program has the additional feature of sounds being played when a word is caught, missed or typed incorrectly.

As well as a results window that shows the scores and allows the user to play again or quit.

6 README

6.1 Implementation

In order to run the program the following command like prompts should be executed: Make sure to be in the root directory.

- *make*
- *cd bin*
- *java Word App <Total Words> <Words on Screen>* <Dictionary File Name>

In other words the program should be run from the bin directory after the make command has been run.

6.2 To Note

- All dictionary files should be placed in the dictionaries directory.
- Audio file that you may like to play should placed in the audio directory.