**CSC2002S ASSIGNMENT4**

**REPORT**

***INTRODUCTION***

Concurrent computing is a form of computing in which several computations are executed during overlapping time periods. In concurrence our main concern is correctly and efficiently managed access to shared resource.

In this project we are designing a multithreaded java program, ensuring both thread safety and sufficient concurrency for it to work. We have a main GUI window with words falling, and the user has to type these words on the text field, if the typed word matches any falling word then the score increases by the length of that word and the word is caught (removed from the window). If the word is missed (i.e. the word reached a red zone) we increment missed by 1 and remove the word. This GUI window has a start, end and quit buttons.

***CLASS DESCRIPTION***

**Added classes**:

**-Simulate class:** this is a thread responsible for starting simulation. The given classes only display words at the top of the window, this class starts dropping these words with each word falling at a different speed. This class extends Thread class and overwrites the run method.

**-CheckWord class:** a thread responsible for comparing the typed word to the words falling. It extends thread class and overwrites the run method. An array of type WordRecord is created which contains all words falling to the window. If the typed word matches any falling word, this class calls the resetWord method from WordRecord and this method removes the word from the window and sets a new word falling at a different speed. This class also sets a new caught value and new score.

**Modifications on existing classes**: ***WordPanel class*** modifications: After drawing each word on the GUI, the y position of the word is checked, if its position is equal to maximum y position we say the word has reached the red zone we remove it a and increment missed words then write new missed in to the GUI. In the run method of this class we start a thread that starts the simulation by creating a thread and pass the **simulate class.**

There is also an added clear Boolean variable. This variable is false if words are being painted on the GUI panel and true when the end button is pressed, and this variable is true it clears the screen and restarts the game.

***WordApp class*** modifications: In this class we created three setText methods for caught label, missed label and score label. The reason for these functions is to be able to set new caught, missed and score values outside this class by calling these methods. We use these methods in the **ChechWord class** and in WordPanel class.

***Score class*** modifications: in this class we synchronize all methods that are setters and getters.

***JAVA CONCURRENCY FEATURES***

In the WordRecord class all methods are synchronized. Java synchronized blocks are used to avoid **race condition** a special condition that may occur inside a **critical section**. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the results of the concurrent execution. The reason for synchronizing these methods is because they all set and return values, which may cause race condition if not handled.

In the simulate class, a volatile Boolean variable called flag is created. Volatile has semantics for memory visibility. This volatile flag is visible to all readers (threads) after a write operation completes on it. So the threads using this flag will read the current updated status of this flag. Without volatile threads could read non-updated value.

In the WordPanel class, Boolean done and clear variables are also declared as volatile, for the same reason as flag variable. And these variables are also used in other classes.

1. **Thread safety**: when accessing words from the array and displaying them to a GUI thread safety is needed. Without thread safety, threads could try to access the word and that may lead to these thread halting. Thread safety is also implemented when checking for matching words and when updating the score. SWING threads are busy executing on the background, so thread safety ensures that threads for shared variables and SWING threads do not interrupt each other.
2. **Thread synchronization**: when starting two threads in the WordApp (CheckWord and Simulate threads), there may be a situation where these threads try to access the same resource, words or WordRecord values and Score values and finally can produce unforeseen result due to concurrency issues. So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is accomplished by using synchronized blocks in WordRecord and Score classes.
3. **Liveness**: set of properties of concurrent systems that require a system to make progress despite the fact that it’s concurrently executing components may have to take turns in critical sections. Two classes (CheckWord and Simulate) are created separately and when they are called, two different threads are created which call these classes and start execution as specified by these classes. By this we ensure code liveness.
4. **Deadlock**: a state where each thread is waiting for another thread, including itself, to take action. This is avoided by synchronized blocks, synchronized methods in the WordRecord class.

**Model-View-Controller pattern:** the model comprise of classes **WordDictionary**, the array of **WordRecords** and the Score. The view is the GUI which shows animation. Controllers comprise of classes **CheckWord** and **Simulate**, these classes alter the model and the view, moving the word position, performing animation, adding and removing words as necessary and updating the caught and missed counters and the score.