Report

̡̨̨̢̧̨̣̩

# Section 1:

## 1:

A screen shot of a computer program

Description automatically generated

The classes responsible for starting threads is the GameBoard.java Class, this class creates 5 threads. These threads are used to run specific game tasks. The thread pool, also known as threadPool is used to control the robot movement, in which each robot has their own thread to move at their own pace, specified by their delay. This thread will check the interactions between robots, and the environment such as robots colliding with the walls, as well as changing and updating the robot positions on the board. The use of a cached thread pool is due to not having a limit on the number of robots that can be spawned and that can move simultaneously to other robots or placements of walls. Therefore, this allows a large number of threads to be created and used for robots.

Another thread is used to schedule the creation of robots, this scheduler uses a scheduleAtFixedRate, to determine the speed at which robots are created. Specifically, the robots are designed to be created every 1500 milliseconds. However, the scheduler will attempt to create said robots, after checking the 4 spawn locations specified. The thread will randomly choose a spawn location, but if the spawn location is already occupied, the spawn will be cancelled, and no robot will spawn. After another 1500 milliseconds the thread will attempt another robot creation, in which if the randomly chosen spawn location is free, a robot will be created and spawned on the map. This thread is created on a single thread, as only one robot can be spawned at a time, with a delay of 1500 milliseconds.

The next thread is called the wallSchedule, this thread is used to schedule the creation and placement of the walls. First the wall creation will be put into a queue, where the wallSchedule will then create a thread to take a wall out of the queue, check whether the wall can be placed, and then add the wall to the board position. This thread is set to be delayed 2000 milliseconds as described in the problem statement. Every time a wall is queued, the thread will take 2000 milliseconds to build the wall, in which case if there are more walls in the queue, the thread will take the next wall and attempt to create that in another 2000 milliseconds. This thread is used on a single thread, as only one wall can be built at a time with a 2000 millisecond delay, and only one wall can be taken from the queue at a time. Therefore, it was determined that a single thread would be sufficient to demonstrate the creation of the walls.

The final two threads are the wallLabelUpdater, and the scoreUpdater, these two threads are both single threads, that update the JavaFX GUI label that displays the score, and the number of walls being queued to build. Both these labels update every 1000 milliseconds and use the Platform.runlater method to invoke the JavaFX Thread, as they both update the JavaFX GUI. The scoreUpdater will increase by 10 every second and increase by 100 whenever a robot is destroyed by a wall.

## 2:

The thread communication that occurs, happens between the scheduler. The communication between the scheduler and threadpool occur after a robot is created by the scheduler. After a robot is created by the scheduler, the scheduler will send the data of the robot object to the threadpool, in which the threadpool will execute the robot movement until the game is over.

The threads share resources through the use of lists, board positions and the arena object. There are 3 lists and a 2D array of Props. The shared resources are the list allProps, the 2D Prop array and the arena object. The allProps list is used to store a list of all the props, this includes walls, robots, and the citadel, when a wall, robot or citadel is created, this list is accessed and added to by the threads, and therefore when it is being accessed is put into a synchronized(mutex) statement until the accessing of the list is complete.

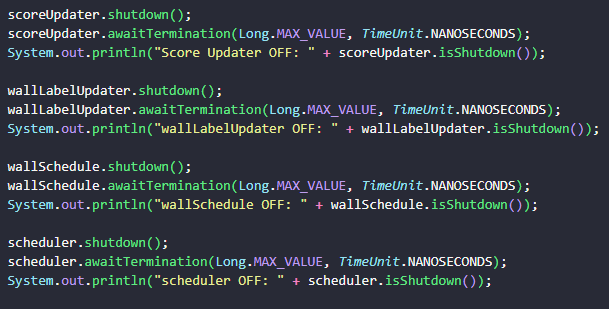
The 2D Prop array is used to determine the locations of the props. Whenever a robot is spawned on the map, moves or when a wall is placed, this 2D array is accessed and edited. The threads use this to update the X and Y coordinate of the specific prop, it is used to either remove the previous location, to provide a new free square for another prop to move or be placed in, or for the prop to move into a new X and Y location.

The final shared resource is the arena object. The arena object is used to requestlayout, in which the GUI to display the game on, will be updated. Whenever a thread completes one of its functions (move, creation, etc) the requestLayout function will be called through the arena object to update the GUI. Whenever these shared resources are being accessed or edited by threads, the use of synchronized(mutex) statements are used to prevent race conditions and deadlocks in the program, allowing the threads to run simultaneously.

A computer screen shot of text

Description automatically generated

## 3:



All threads check if the game over Boolean is true, once the game over is true, the threads will interrupt, and the endThread function will be called. This function uses the thread.shutdown() method as well as the thread.awaitTermination() method. This tells the thread once the game is over, to start shutting down and the thread.awaitTermination will allow the thread to gracefully end the task and shut down the thread. Once all threads are shutdown, it will be displayed in the terminal: for example.

A screenshot of a computer screen

Description automatically generated

# Section 2:

## 1:

A non-functional requirement that could be considered is performance. With a larger number of game objects and shorter time intervals, the application's performance might be a concern. This is because it needs to handle updates between the objects, lists, and gameboard as well as the interactions between the robots and the walls a lot more efficiently, to maintain an experience that doesn’t incur race conditions or deadlocks when continuously updating the Java GUI and the shared resources.

Another non-functional requirement that could be considered is the use of concurrency. Managing concurrent access to shared resources, such as the game board or game objects, becomes more challenging with many objects. Ensuring thread safety and avoiding deadlocks would be extremely important and I think it would be the largest problem that will be presented if the program was to increase in scalability. Since there will be thousands or millions of game objects with shorter intervals, these shared resources will need to be identified, and minimised or at least minimise the amount of locking that is being used in the program to reduce the lock contention as many objects will be running simultaneously. I would also need to introduce thread-safe data structures or lock free data structures, in which I have not implemented in the current version.

2:

If I was to re-engineer the application’s architecture, I would attempt to make the method calls and data structures different. I would like to remove or minimise the number of shared resources that are accessed or edited at the same time, such as making it so that when a robot moves, it does not need to directly access the game boards 2d array, allowing the robot to not hold the game boards array in a synchronized lock. This will allow the robot to rum much more efficiently, and smoothly for all other robots to move simultaneously in a much more efficient way. This can also be done to anything that needs to be placed or moved on the board, so the locking of this shared resource can be removed. I would also like to attempt to change these structures into thread-safe data structures so that concurrent accessing is done safely and attempt to implement lock-free data structures to reduce contention for the resources.

I would also like to re-structure methods, so that methods do small and simple tasks instead of large and/or more complex tasks that take up more time. For example, my robot movement does more than 1 task, and calls many functions, which may have affected the performance of the program. If I can make methods do small tasks fast, then the program will run a lot faster and smoother.

If these are implemented correctly, the program will run many times faster than it currently is, and will allow for more scalability, in which it MAY currently not, as I have not tested for thousands of objects or shorter intervals. Although I have not tested for a larger grid, I don’t believe that will be much of an issue to increase in the current program.