**Software Engineering Concepts**

**Assignment 1**

**Due: 20/09/2020**

**Aaron Gangemi**

**Student Number: 19447337**



Contents

[Introduction 3](#_Toc50482529)

[Threaded Design 3](#_Toc50482530)

[Communication between threads 3](#_Toc50482531)

[Avoiding Race Conditions and Deadlocks 3](#_Toc50482532)

[Ending Threads 3](#_Toc50482533)

[Class Responsibilities 4](#_Toc50482534)

[Droid Class 4](#_Toc50482535)

[Thread Pools 4](#_Toc50482536)

[GameController 5](#_Toc50482537)

[Grid Updater Class 6](#_Toc50482538)

[Multiplayer Version issues 6](#_Toc50482539)

[Multiplayer Architectural Decisions 6](#_Toc50482540)

# Introduction

For this assignment, I have implemented a game with the provided pre-existing code which allows droids to move freely across a 5x5 grid space until a droid successfully reaches “the fortress” located at grid space (2,2). The user can shoot the droids by clicking on an associated grid space. In addition, the game successfully tracks the users score which is increased by 10 points for every second that the game is played, or if a droid is shot successfully, then the score is increased by a greater difference. All shooting activity is logged in the text area of the game and the user is notified when the game is over. Furthermore, the implementation of the program required several tasks to be both scheduled and multi-threaded using existing Java features which will be further discussed in this report.

# Threaded Design

## Communication between threads

To communicate between threads, each class is passed an instance of the JFX arena which will allow them to utilize the arena’s accessors for its class fields. The arena contains shared variables which allow threads to read and write to these variables. Examples of these shared variables are the GridTracker which tracks which grid spaces are occupied and the score variable which tracks the users current score.

The score updater thread is required to update the score it a different thread every second, however when a droid is destroyed, the score variable is updated from the firing command thread. To determine when a droid is firing to update the score, another shared variable called “isFiring” is used to indicate that the score is being updated elsewhere in another thread.

The GridTracker is updated in the droid thread, by many different droids at the same time so this variable is updated across many different threads.

## Avoiding Race Conditions and Deadlocks

My program implements four mutex’s which ensures that if a shared variable has the potential to cause a race condition or deadlock, the mutex ensures that only one thread at a time can access a block of code which contains that shared variable. By doing this, only one thread will be able to write to the variable. To implement this, I have used Java’s “synchronized” keyword which uses the mutex to allow one thread to enter, and any other thread that attempts to enter will be blocked.

Correspondingly, to store the list of droids, I have implemented a LinkedBlockingQueue. The linked blocking queue ensures that modifying the queue will not cause a Java concurrent modification exception (Race Condition). This is necessary as when a droid is successfully created, it adds it to the queue, and when a droid is clicked on, it is removed and therefore this action must be controlled. A list could be used to implement this function, however the LinkedBlockingQueue creates a thread-safe space. My reasoning for making the queue a LinkedBlockingQueue is that by having no bounds on the queue, droids will be free to add to the queue at any point in the program.

## Ending Threads

There are various points in my program where I require a thread to end. For example, if a droid is clicked on then it must be removed from the grid space which requires the thread to end. To do this, the firing command will set the droid’s alive status to false which will stop the droid from moving. If the droid stops moving, then the move droid function will Java’s Thread.CurrentThread().interrupt() method to stop to droid from moving. Another instance where the threads must end is if the game is over. Once a droid has reached the fortress, then the ScheduledExecutorService is shutdown, ending all scheduled threads in the ScheduledExecutorService and further clearing the list of droids.

## Class Responsibilities

I have structured my program to appropriately ensure the division of responsibilities.

1. The Droid class is responsible for controlling the droid thread
2. The GameController class is responsible for handling all inputs and outputs in a new thread
3. The GridUpdater class is responsible for ensuring any grid movements are thread safe to ensure that multiple droids cannot move into the same grid space
4. The JFX arena is responsible for handling the layout of the arena and the spawning of the initial droids.

### Droid Class

The droid class is responsible for tracking the state of a droid and firing off a new thread that will allow the droid to move. The droid class will store the following fields:

1. Droid ID: The unique identifier for a given droid
2. Droid delay time: The delay time the droid will have to wait before moving
3. X and Y Coordinates: The current X and Y coordinates that the droid is at on the 5x5 grid space
4. Droid Moving: A Boolean variable indicating whether the droid is moving or not. This is used to ensure that a given droid can not move multiple spaces at any time
5. Is Alive: A Boolean variable indicating whether the thread the current droid is in is still running and the droid exists on the grid space
6. Arena reference: Each droid will require a reference to the provided JFX arena for it to spawn and move around.

The droid class is designed such that when a droid is created, a new thread will also be created which will allow that droid to move while it is alive, and the game is not over. This is implemented in a private class called MoveDroid which is in the droid class. The MoveDroid thread works by adding numbers one to four to a list and shuffling that list every time the droid wishes to move. Each number represents a direction that the droid can move in. The program will then make the appropriate move based on the number provided.

After the droid has successfully completes the move, the droid will be required to make several checks. The first is to check that the droid has completed a move. If the move was completed, then the program will check if that any droid has reached the fortress. If any droid has reached the fortress, then the game over thread is executed. However, if no droid has reached the fortress than the program checks to every second to see if any of the corners do not have droids in them. If a corner is empty, the program will use the mutex provided in the arena class to allow the thread pool to populate each corner with a new droid. The program will then wait for the droids given delay time before moving.

### Thread Pools

The JFX Arena requires a way which allows droids to spawn every second if a corner is empty. I have utilized Java’s ScheduledExecutorService which will schedule for a droid to be spawned in a corner after a given time frame. The implementation of this was completed using the scheduleAtFixedRate(Thread, Initial Delay, Droid delay time, Milliseconds as the time unit).

It is important that a droid cannot move into the grid space when any droid is spawning. To control this race condition, I have used the “mutex” variable which will wait while the droid is moving, and when the move thread in the droid class has realized that a grid space is empty, it will unlock the mutex and allow the spawn droid thread to fill in the grid corners with new droids. Once this operation has successfully been completed, the program will continue to move the droids in their corresponding threads.

The second ExecutorService implemented in the program is used to queue up the firing commands once the user has clicked on a grid space. The ExecutorService ensures that commands are executed with a 1 second wait between execution of firing commands. I have provided the firingService a Synchronous queue to allow it to submit and execute firing commands to be executed.

### GameController

The game controller class is used to handle any inputs and outputs that the program is required to use. The inputs the game controller handles are the clicking of a grid square to generate a firing command. The outputs the game handles are updating the score, logging the text to the text area on the GUI, and informing the user via an alert that the game has finished due to a droid reaching the fortress. Once an instance of the GameController is instantiated, it starts the game over thread and the score thread.

#### Game Over Thread

The game over thread runs constant checks to see if a droid has reached the fortress. If a droid has not reached the fortress, it uses a game over mutex which is stored in JFX Arena to allow other threads to use the move droid thread to check if the game is over. Once this condition has been met, the move droid will wait and allow the game over thread to use the mutex. The game over thread clears the ScheduledExecutorService and list of droids so that no droids can further spawn or move, then it will access the GUI thread using Java’s Platform.runLater method and display an information alert displaying the users final score. This game over thread is then completed.

#### Firing Command Thread

The Firing command class is a private class within the GameController class. It is used to generate a new firing command and update the score after a droid has been removed. Whenever a firing command is executed by clicking on screen, the program will wait 1 second between executions using Java’s Thread.Sleep(1000) method. The firing command will then access the GUI thread to update the new score. As mentioned earlier, the scoring is incrementing every 10 seconds, so I have implemented a mutex called score mutex which will allow the score to increment every 10 seconds but will then wait if a firing command has been requested. Once the firing command is complete, a call to scoreMutex.notify() is made so that the score may continue incrementing every 10 seconds in a separate thread. Following this, the grid coordinates that the droid was at are further set to 0 to indicate that the grid space is empty.

#### Score Updater Thread

The final class is GameController is the private ScoreUpdater class. The purpose of this class is to allow the score to update every 10 seconds in its own thread, so it does not interfere with the scoring additions when a firing command is executed. The score mutex is used to protect the variable “isFiring” so that only one droid can access it at any time. If “isFiring” is true, then the score is updating elsewhere and the ScoreUpdater thread will wait. Once the “isFiring” is false, the score will continue to update every 10 seconds. This thread then accesses the GUI thread using Platform.runLater to modify the score on the GUI label.

### Grid Updater Class

The grid updater class is used to update the coordinates of the shared variable “gridTracker” in a thread-safe environment. The GridUpdater contains a class field called “gridUpdateLock” which acts as a mutex to only allow one droid thread to update any given grid space at a time. By using this mutex, we ensure that a race condition cannot be caused from 2 robots attempting to update the same grid space at the same time.

The class contains 4 functions. Two functions to update the grid if the robot attempts to move up or down, and two function to update the grid if the robot attempts to move left or right. A robot may only access one of these functions at any given time.

Communication between these functions is achieved using the shared integer array “gridTracker”, which stores 25 integers of either 1’s or 0’s. If the grid space in the array is set to 1, then the space is occupied by a given droid and no other droid can move there. However, if a space is set to 0 then the grid space is empty, and any given droid is free to move there.

## Multiplayer Version issues

For a multiplayer version of the game to be created, there are several arising issues that would need to be addressed.

1. In a multi-player version, there will be required to be multiple users clicking on the same grid causing droids to be removed on all screens that are currently playing the game. To implement this, each users game will have to be in sync and be updating in real-time once either an event such as a droid moving, or a droid being killed has occurred.
2. The multiplayer version may require a scoreboard to be implemented across each user that is connected to the game. There will need to be a separate thread constantly updating and further sorting the scoreboard to ensure that it is up to date with each connection of the game. In correspondence to this feature, each user will need to enter a username or set of login credentials to recognize the user on the scoreboard.
3. To ensure that the game is up to date in real time, each user will be required to have an internet connection to constantly retrieve game data. Therefore, the program will need to have a feature that will only allow the user to play while they are connected to the internet. From here, the program could implement a different thread to fetch data in real-time.
4. The risk of implementing a multiplayer version is that some users may experience lag time. The program will need a way of refreshing a user’s connection so that if they are to experience any lag due to connection issues, then they will be able to recover from it and continue playing.
5. For the program to track the scoreboard, it will need to know which players have successfully killed which droids. However, in doing this, there is also the possibility that 2 players may click on the same grid square at the same time. The program must have a way of handling concurrent events across multiple users.

## Multiplayer Architectural Decisions

1. Implementing the game with a client-server architecture: As identified above, many users will be connecting to the server at different times and will need to retrieve data at different times. By storing this data using a server architecture, each client will be able to make a request to fetch new data. The server will be able to keep each client up to date with the game in real-time which will handle many issues.
2. Implementing the scoreboard: The scoreboard will need to be implemented using multi-threading combined with the server architecture mentioned above. Firstly, the server will need to track which users are currently playing the game and then further track their current scores. If a user kills a droid in the game, the game must ping the server to update there score so that each other user connected is aware of the updated scoreboard. This will have to be done in a separate thread to the main game, hence why multi-threading is required.
3. Controlling networking and lag times: To ensure that users can connect to the server, a networking feature will need to be implemented successfully. To ensure that the server is not overloaded with users, one may limit the number of users that can connect to the server at a given time. An alternative to this would be to run the game through the cloud. By doing this, the architecture may be able to handle more requests from more users making the game more available.