Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Nguyen	Student ID:	19247171
Other name(s):	Phi Long		
Unit name:	Software Engineering Concepts	Unit ID:	COMP3003
Lecturer / unit coordinator:	Dr. David Cooper	Tutor:	Dr.David Cooper
Date of submission:	20 September 2020	Which assignment?	1 (Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Phi Long Nguyen Date of signature: 20/09/2020

(By submitting this form, you indicate that you agree with all the above text.)

• Explain your design in regards to threading

Threads in the design:

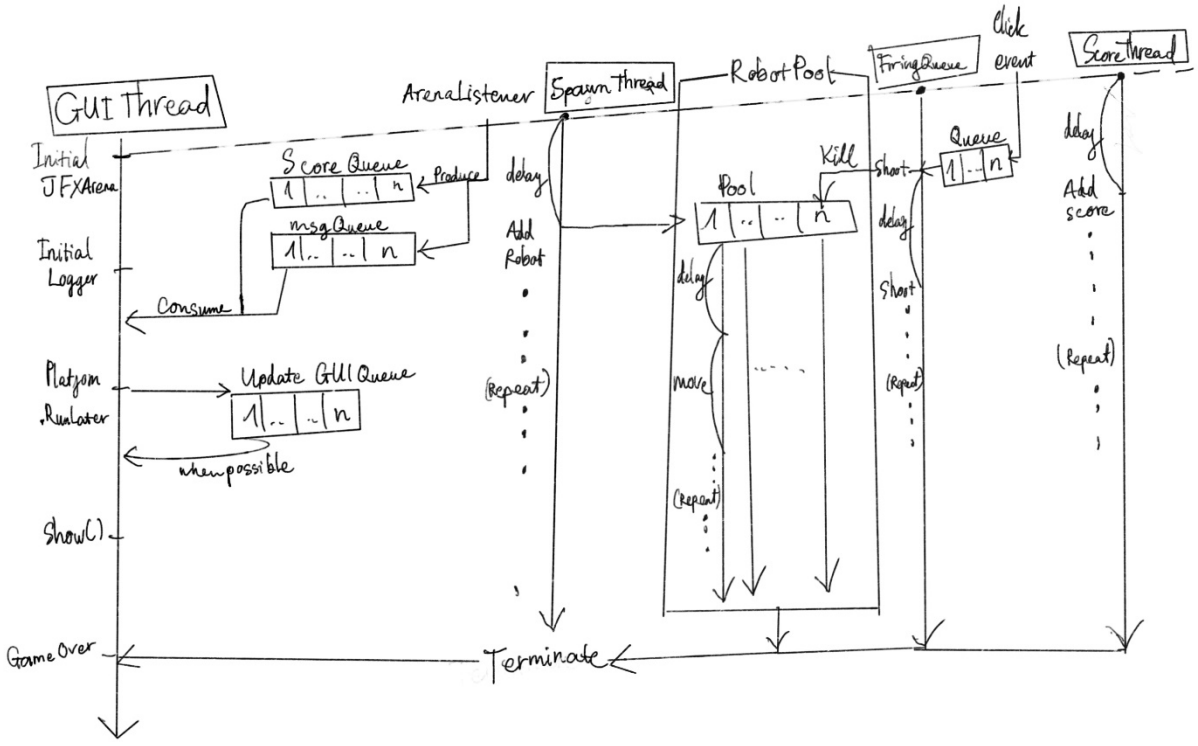
-Spawn Thread(1), Robot threads(width*height), Firing Thread(1), Score Thread(1), GUI thread

Note: if width grid is 5, height grid is 5, numbers of robot threads is 25.

Shared resources: List<Robot> robotArmy

Blocking Queue: firingQueue, msgQueue, scoreQueue

Threads Flow Chart



– Which classes are responsible for starting threads, and what are these threads used for?

The JFXArena constructor responsible for initial canvas, based on this feature, I want the robot start spawning, score start increasing, and ready to consume if any bullet reload as soon as the initial stage is ready. Therefore, the JFXArena class have startGame() function which start Spawn

Thread, Score Thread and Firing Thread, and the App class is responsible when to call it.

```
arena.setMinWidth(300.0);
arena.startGame();

BorderPane contentPane = new BorderPane();
contentPane.setTop(toolbar);
contentPane.setCenter(splitPane);

Scene scene = new Scene(contentPane,  width: 800,  height: 800);
stage.setScene(scene);
stage.show();
```

These threads(spawn, score, firing) are single thread because it only have 1 long task running.

```
//spawnPool will spawn robot
private ExecutorService spawnPool = Executors.newSingleThreadExecutor();

//scorePool will handle record and increase score
private ExecutorService scorePool = Executors.newSingleThreadExecutor();
//firingPool will handle reload and shoot bullet
private ExecutorService firingPool = Executors.newSingleThreadExecutor();
```

The thread pools handle a single runnable task:

- spawnTask (JFXArena): spawn a robot at a corner randomly every 2 seconds.

```
Runnable spawnTask = () -> {
    try {
        while (true) {
            Thread.sleep( millis: 2000); //default n value
            spawnRobot();
        }
    } catch (InterruptedException e) {
        System.out.println("Stop Spawning Robots");
    }
};
```

Note: because it is random, sometimes it takes longer than expected for the random algorithm to find empty corner if at least 1 corner is occupied. E.g. keep generate the last corner (4th position), so cannot check another corner.

- firingTask (JFXArena): this thread is a firingQueue's consumer, it will shoot the bullet (if any) every 1 second. So, when the game starts this task will keep asking the queue if there is any bullet in the queue else wait.

Note: no limit number of producers for bullet. Click event on any square will generate a bullet and then add it to the firingQueue.

```
Runnable firingTask = () -> {
    try {
        while (true) {
            Bullet bullet = firingQueue.take(); //wait until new bullet load
            shootBullet(bullet);
            Thread.sleep( millis: 1000); //wait for 1 second for next bullet
        }
    } catch (InterruptedException e) {
        System.out.println("Stop Fire Queue");
    }
};
```

- score Task (Score class): start add the score every second, add 10 points for the player for keeping fortress save.

```java
@Override
public void run() {
    try {
        while (true) {
            Thread.sleep( millis: 1000);
            addScore( plusScore: 10.0);
        }
    } catch (InterruptedException e) {
        System.out.println("Stop Counting Score");
    }
}
```

The 3 threads mentioned above running same time when the game started.

About robot Threads, I treat each robot as a task because they behave unpredictable and they have different characteristic, so it every hard to have one or 2 threads controls fully occupied (worst case) robots. Therefore, 10 robots need 10 threads and so, if the cores are not sufficient for number of robots the robot will just stand and wait for an available resource from pool. Because of many threads, I use ThreadPool fixed size (not just single threadpool) to handle robot threads, the reserved cores I used are gridWidth*gridHeight. When a robot is killed, I can reuse the thread in thread pool, this will save a lot of resources.

```java
private List<Future> robotThreadIndicator = new ArrayList<>();
// robotPool control robot characteristic: one each
private ExecutorService robotsPool = Executors.newFixedThreadPool( nThreads: gridWidth * gridHeight); //fully occupied
```

So the robot should only start moving when it is spawn.

```java
if (!isOccupied(newRobot)) {
    robotArmy.add(newRobot);
    robotCounter += 1;
    robotThreadIndicator.add(robotsPool.submit(newRobot)); //start robot movement(task) thread
    for (ArenaListener listener : listeners) {
        listener.spawnRobot(newRobot.getId()); //notify App class to log
    }
    requestLayout();
}
```

The single robot thread will perform move from old position to new position every robot's delay.

```java
@Override
public void run() {
    try {
        while (true) {
            Thread.sleep(delay); //wait for next available move
            setDirectionFlag(true); //do not move this line
            support.firePropertyChange( propertyName: "directionFlag",  oldValue: false, directionFlag);
            //find the next square to move to
            //stop finding new direction when have desire square to go.
            if ((directionX != 0 || directionY != 0) && !getDirectionFlag()) {
                processOfMoving();
            }
        }
    } catch (InterruptedException e) {
        System.out.println("(thread shutdown) Robot class: Robot " + id + "  killed");
    }

}
```

The processOfMoving() is a smooth transition of the robot

```
//take 500ms to move
private void processOfMoving() {
    //start process of moving
    //move 10% every 50ms
    setDirectionFlag(false);
    try {
        for (int i = 0; i < 10; i++) {
            if(getDirectionFlag() || (directionX == 0 && directionY == 0) ) //being shutdown
                Thread.interrupted();
            this.move( directionX / 10.0,  directionY / 10.0);
            Thread.sleep( 50);
        }
    } catch (InterruptedException e) {
        System.out.println("(thread shutdown) Robot " + id + " Killed While moving");
    }

}
```

Lastly GUI thread, I need to log message to log area whenever event happened and show current score by update label for toolbar. However, GUI thread cannot handle heavy task, it is bad idea and impossible to have 2 non-GUI threads (one to busy waiting and consume message, one for showing current message). Therefore, I use Platform.Runlater() which allows to queue up runnable tasks (GUI tasks) and execute one by one whenever GUI thread is available, so it won't freeze the GUI.

```
logger.addNewMessage("!!!!!!!!!!!!!!!!!!!!!!!RELOAD BULLET!!!!!!!!!!!!!!!!!!!!!!!!!\n"
        + "Aim at ( " + x + "," + y + ")\n" +
        "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
String newMessage = logger.takeMessage();
if (newMessage != null) {
    Platform.runLater(() -> loggerArea.appendText(newMessage));
}
```

Logger is the container for new message and current score. The messages and score will be stored before GUI ready to consume, this process does not take long, so the race condition or freeze GUI does not happen. But to make sure the order is correct I use Blocking Queue to maintain threadsafe.

```
private BlockingQueue<String> msgQueue = new LinkedBlockingQueue<>(); //Log area is responsible for printing msgQueue
private BlockingQueue<String> scoreQueue = new LinkedBlockingQueue<>();//label on tool bar is responsible for printing scoreQueue
```

– How do they communicate?

The solution I use for communication is add listener:

- the JFXArena class will listen to robot position and total score.
- the App class listen to JFXArena class when particular events happen: robot spawned, bullet reload, shoot miss/hit, increase score.

In additional, as shown in the Thread flow graph. We can see how the threads communicate to each other's.
Spawn Thread could add new robot Thread to robots' thread pool, Firing Thread could kill/terminate a robot thread if condition met. Whenever this event happens, it will notify GUI thread by listeners.

– How do they share resources (if they do at all) without incurring race conditions or deadlocks?

As we mention, robot can be adding or removing from different threads, they share the same resource which is List<Robot> robotArmy. Therefore, this list should be threadsafe

```
private List<Robot> robotArmy = Collections.synchronizedList(new ArrayList<>());
```

BlockingQueue helps prevent race condition when produce/consume bullets and messages

– How do your threads end?

When the bullet hit a robot, the robot should be killed and remove from the game. To do so I have a Future robotThreadIndicator to keep track of the robot task. So, when I need to terminate a robot thread, I only need to refer to the Future the robot task return when robot thread pool submits.

```
robotThreadIndicator.add(robotsPool.submit(newRobot)); //start robot movement(task) thread
```

```
//remove robot from the game
robotThreadIndicator.get(targetedRobot.getId() - 1).cancel( mayInterruptIfRunning: true); //terminate thread.
robotArmy.remove(targetedRobot);
requestLayout();
```

When the game is over, one of the robots in robotArmy occupied the fortress, all threads should be clean after this condition is met, only keep GUI thread running since we need to show final message.

```
//End Game Condition
if(isOccupied(fortress.getPositionX(),fortress.getPositionY())){
    for (ArenaListener listener : listeners) {
        listener.gameOver(score.getTotalScore()); //notify App class to log
    }
    scorePool.shutdownNow();
    firingPool.shutdownNow();
    spawnPool.shutdownNow();
    robotArmy.clear();
    for (Future e :robotThreadIndicator) {
        e.cancel( mayInterruptIfRunning: true);
    }
    robotsPool.shutdownNow();
}
```

All the task including running tasks and not beginning tasks should be terminated. The shutdownNow() built-in ExecutorService would interrupt any task in the pool, need to make sure the task must be running. The not beginning tasks could be ignored by shutdownNow(). As for robot threads, it is very difficult to terminate all thread with shutdownNow() alone, so I use robotThreadIndicator to interrupt all robot first then shutdown the threadpool.

- **What would be the best way (architecturally speaking) to transform this app into a cooperative multi-player game, where each player has their own separate fortress, and they must work together to stop the robots.**
- **Consider some *plausible* non-functional requirements for such a system, and say what architectural decisions could help satisfy them.**
  To transform this app into a cooperative multi-player, first I will add the matchmaker feature to the game. Matchmaker will define how many players are minimum/maximum in a lobby, and ensure that initial stage of all players are the same.

Making this game local co-op, only need to make another set of input event for 2nd/3rd/4th players. For example, one player uses mouse, one use arrow and 1 (to shoot), another use WSAD and F (to shoot), etc.

Making this game online co-op, we will need a distributed system. The game logic would be sat in business tier, players can interact/communicate with the game server at presentation tier using Socket.io or Node.js or JRE, and we will need a database server to upload and record players' stage/ move/ results this is our game data tier.

Depends on how we define the maximum number of players per lobby we will need to have a dedicated server strong enough run our game. Because we need to broadcast data to all players in current room/server. A cheaper approach is Peer to peer model; however, one player's connection can influence the game experience for the others, P2P are also open to DDoS attacks.

As I mentioned above, the game needs to broadcast to other players, depends on how many players in one lobby, we need a dedicated server to run the game. So, for this game I would say its maximum 10 players per lobby (map 50x50 - normal difficulty). Higher max players are too expensive to run.

Since this is a shooting game and require a real-time communication, UDP suits best to transmit the data.

The game is potential to deploy cross platform to play. Beside that we should consider OS consumption for different platform. Depends on what platform and how many game sessions we need a CPU requirement to play the game. 5 game sessions for this game require 28 cores minimum because we are using 5.5 threads per session for the current game design.