Document version: 1.1 (2015-11-15)

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:	Shannon-Innes	Student ID:	18529687
Other name(s):	Kalon		
Unit name:	Software Engineering Concepts	Unit ID:	COMP3003
Lecturer / unit coordinator:	David Cooper	Tutor:	
Date of submission:	Sept. 20, 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: Kalon J Shannon-Innes

Date of signature: Sept. 20, 2020

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

Kalon Shannon-Innes 18529687

Software Engineering Concepts
Assignment 1, 2020

Project Summary

On top of meeting the threading requirements for this project, I attempted to minimize coupling, maximize polymorphism, and ensure high testability via dependency injection. To this end, I implemented a single class called "ThreadHandler", to be responsible for managing all threads (including its own, in a way). No other class is aware that it might be running on a separate thread, as ThreadHandler takes care of all Runnable and Executor considerations. I also implemented a Game class which handles all the game specific state, and several other classes each responsible for managing the scoring, player attacks, entity movement, and more.

I will go into more detail later regarding the relevant classes and how they communicate, but first I will talk about my general multi-threading strategies.

Note: I deliberately left out a lot of comments throughout my code because I feel my method and variable names are very self-explanatory. I attempted to use in-line comments where I felt something may not have been very clear, or otherwise frustrating to read through.

Thread Management

ThreadHandler is the class responsible for all threading related concerns. My main goal in the implementation of this class was to ensure that no other class would ever be aware that it might be running on a different thread from the rest of the program. ThreadHandler maintains 4 ScheduledExecutorService's. The first 3 are only for maintaining a single thread each, for ScoreHandler, AttackHandler, and SpawnHandler. The 4th Executor is a pool of 20 threads, specifically for MovementHandler, which handles the movement of spawned entities.

To further promote encapsulation of thread management, none of my classes are specifically Runnables. Instead, ThreadHandler creates anonymous Runnables, inside of which each of the Handlers performs their specific task. This benefits encapsulation in two ways. First, because none of the Handlers are Runnables, they are all easily switched from a multi-threaded to a single-threaded environment by just implementing a ThreadManager that doesn't create new threads at all, and instead just operates the Handler's inside loops. The second benefit is that, because we're using Runnable lambdas, ThreadHandler is able to manage the creation and destruction of MovementHandler futures whenever an entity is created or destroyed.

When a game is ended via the player closing the GUI window, or by pressing the stop button, the GameManage.stop method is called. This method begins the shutdown of the game by first calling ThreadManager.stop. This method gracefully ends all threads, almost without the need for using InterruptedExceptions. However there are 2 exceptions (heh) to this rule. The first is while awaitingTermination of the executor managing the MovementManager's. This is because in order to gracefully shut down the game, the MovementManager's should finish their current move in full.

The second is when processing the AttackManager's FireCommand queue. Because it's a poor experience when a player has to wait an entire second before their first attack hits, I decided that the schedule period for AttackManager would be 16ms (this was a deliberate choice, take a guess why!), with an internal Thread.sleep call of 1000ms to ensure that *queued* attacks would only be processed once per second. Catching InterruptedException is required due to the Thread.sleep call, however nothing needs to actually be done when this exception is thrown, as it would only be thrown when the game is shutting down. This would be different if shutdown events were logged.

Race Conditions and Deadlocks

While ThreadHandler manages the starting and stopping of the individual threads, the problem of race conditions and deadlocks remained. I employed immutable models, a synchronized list, and a blocking queue (for the queueing of attacks), in order to prevent these issues.

There are three separate models in this project. "Entity" represents a generic enemy, "FireCommand" represents the player attacks, and "Position", represents any position on the board. All of these models were written to be completely immutable which, while presenting their own minor issues, completely solved the problem of race conditions. The only shared state in this program is the list of active Entity objects, and that is protected from race conditions by virtue of it being a synchronized list.

Since both Entity and Position are immutable, adding and removing Entity objects from the list of active entities doesn't present any opportunity for race conditions. Both the add and remove methods of a synchronized list are internally wrapped in a synchronized statement, protecting concurrent modifications to the list. This means adding and removing entities are effectively atomic operations.

Due to the immutability factor, when moving an Entity we must first create a new Entity with identical attributes, but with a new Position that is slightly altered from the previous one. Once this is done, we pass the new entity to the GameHandler.moveEntity method, which removes the old version from the active entities list, then adds the new version with the altered position.

Lastly, to handle the queueing of player attacks, I implemented a blocking queue in the AttackHandler class. Because blocking queues are inherently thread-safe by design, this meant that despite all player attacks originating on the GUI thread (via the onMouseClick listener that AttackHandler is subscribed to), race conditions are prevented.

Class Relationships

App, GameHandler, and ThreadHandler

The program starts off in the App class, instantiating the UI, creating all the necessary objects, and injecting them to where they need to go. It's also here that the start and stop buttons are initialized. Once the player clicks on "Start", the GameHandler.start method is called. This method is essentially just a wrapper around ThreadHandler.start, as GameHandler is simply responsible for the current game state (started/stopped, and the entity locations). All of this happens on the UI thread, while the actual running functionality happens via periodic invocation of the SpawnHandler, MovementHandler, AttackHandler, and ScoreHandler classes on separate threads.

LogHandler

Implements: LogManager
Responsibility: Logging events to the screen

All calls to LogManager.log are run on the same thread it was called by, and all calls from LogManager.log to TextArea.appendText are run on the GUI thread via Platform.runLater.

SpawnHandler

Implements: SpawnManager
Responsibility: Creation of entities

SpawnHandler requires a reference to the GameManager, so as to filter the possible positions an entity can be spawned in (preventing entities spawning on top of entities), as well as to actually add the entity to the game board. Calls to the GameManager.filterPositions and GameManager.addEntity methods are run on the same thread as the SpawnHandler, however as mentioned earlier in the report since the list of active entities is a synchronized list, this still does not allow for race conditions to happen.

MovementHandler

Implements: MovementManager
Responsibility: Movement of entities

MovementHandler requires references to the GameManager and the Entity it will move. It makes calls to GameManager.filterPositions, GameManager.moveEntity, and Entity.getPosition. These calls happen on the same thread as the MovementHandler, similarly to the SpawnHandler. Also similarly to SpawnHandler, moveEntity acts on a synchronized list and thus is immune to race conditions when calling List.add or List.remove. However, because moveEntity needs to first remove an existing entity from the list, this requires iteration. According to the Java documentation for synchronized lists, it is required to manually synchronize when iterating over a synchronized list. As such, moveEntity uses the synchronize statement to protect the List.removeIf call, which internally iterates over the list.

AttackHandler

Implements: AttackManager
Responsibility: Queueing and processing of player attacks

AttackHandler requires references to the GameManager and ScoreManager. It makes calls to GameManager.findEntity, GameManager.removeEntity, and ScoreHandler.enemyKilled methods. AttackManager extends the ArenaListener interface, so as to register itself as a listener to the onMouseClick event from JFXArena. The squareClicked method is always run on the GUI thread, as JFXArena is the only class calling it. Since the only code that AttackHandler.squareClicked actually contains is creating immutable objects and adding them to a Blocking Queue, there is no potential for race conditions. To ensure that the GUI is never held up from rendering the graphics, I made sure to use the BlockingQueue.offer call in squareClicked. If the queue is full, the attack should just fail to be queued at all.

ScoreHandler

Implements: ScoreManager
Responsibility: The updating of the player's score

ScoreHandler requires a reference to the user interface. All calls to ScoreManager.enemyKilled happen on the same thread as the caller. ScoreManager.enemyKilled is protected via synchronization of an internal mutex. This prevents the score being updated or read by multiple threads at the same time. I had heard from a couple sources that Platform.runLater and/or Label.setText were internally synchronized and thus I didn't need to synchroninze the reading of ScoreManager.score, but since I couldn't verify this, I decided to use a synchronized statement to assign ScoreManager.score to a local variable, guaranteeing there wouldn't be any race conditions.

SingleLosingPosition

Implements: LossManager
Responsibility: Determining when the player has lost

All calls to ScoreManager.enemyKilled happen on the same thread as the caller. ScoreManager.enemyKilled is protected via synchronization of an internal mutex. This prevents the score being updated or read by multiple threads at the same time. I had heard from a couple sources that Platform.runLater and/or Label.setText were internally synchronized and thus I didn't need to synchroninze the reading of ScoreManager.score, but since I couldn't verify this, I decided to use a synchronized statement to assign ScoreManager.score to a local variable, guaranteeing there wouldn't be any race conditions.

Multiplayer Support

Because I focused so heavily on decoupling in this project, implementing multiplayer support would actually be quite easy! To rewrite this for multiplayer, the user interface would need to be moved to a javascript client running in a user's browser. I would then write a class responsible for managing websocket connections. I would tweak the GameManager such that instead of updating a UI, it updated the WebsocketManager class with the location of entities, and lastly I would implement a new LossChecker class such that it had multiple losing positions (equivalent to each player's fortress). The game flow would involve the browser clients sending player attack commands to the server via their websockets, and the WebsocketManager class forwarding those commands onto the AttackManager. The server would send UI updates back through the websockets to the connected clients, which would then render the positioning of the entities on their respective boards.

One non-functional requirement that this design addresses would be game integrity. Because the server is the sole authority on the game state while the browser clients are only responsible for GUI rendering, there is no way for a player to cheat. They can render whatever they like on their screen, but all decisions regarding entity positioning are maintained server-side, so this would really only hinder the player.

Another requirement solved with this design is scalability. Because the server-side can scale the number of threads up with minor tweaks to the ThreadHandler class, multiplayer support could be scaled to accomodate many players at once. Of course this would also involve writing a class that handles which clients are part of which game, but this would be as trivial as just maintaining a collection of "Lobby" objects which themselves each contain a collection of Player objects. The LobbyManager would just assign new websocket clients to their own Lobby when they connect.
While not directly related to the multiplayer support, I would also like to point out that my design supports very high maintainability. My methods are well named, responsibilities are well divided, the code is highly testable through the extensive use of dependency injection and programming to polymorphic interfaces.

Thread Communication Diagram

Note: Only cross-thread communication is shown in this diagram. If a class is not shown inside a thread boundary, that means it runs on whichever thread invoked its methods.

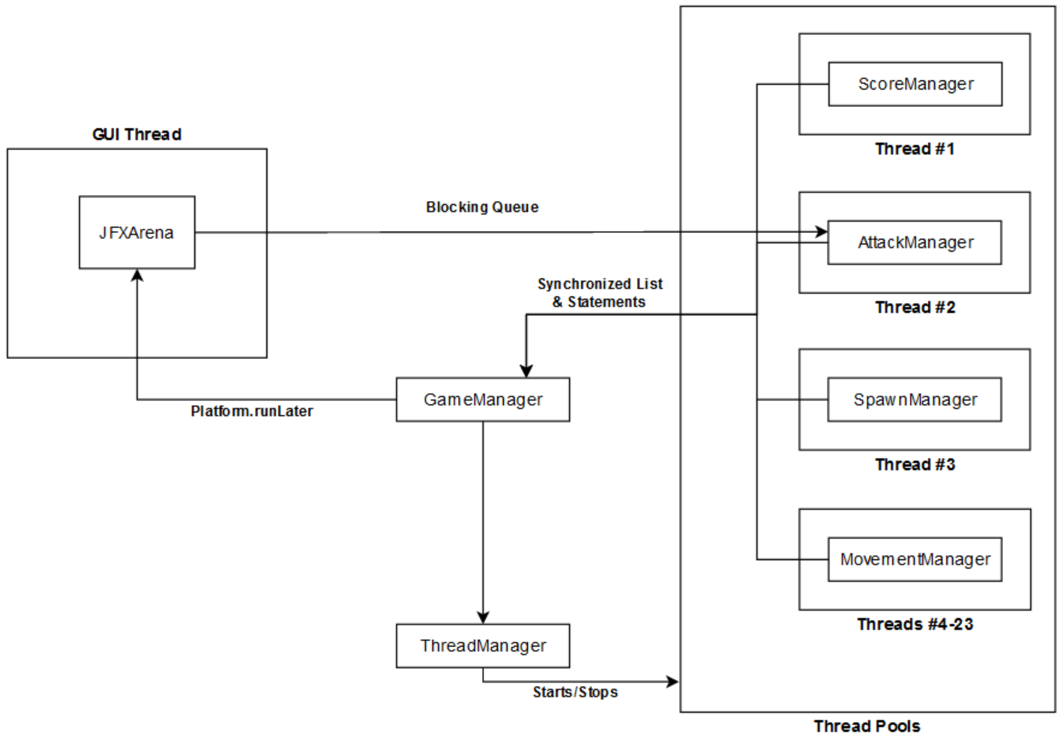

Figure 1: Thread Communication Diagram