|  |  |
| --- | --- |
| **IS303 Concurrency Programming Project**  **PROJECT WRITE-UP** | |
| Group ID |  |
| Name | *Huiyeon Kim* |
|  | |
| Item | Comments |
| Overall description | The project is based on **a MOBA** game (Multiplayer online battle arena) where **multiple players** in 2 teams play in one arena until **one team wins** the game.  In this program there are 3 different GameCharacters – Healer, Attacker and Tank – with 2 teams playing a game at a time. Each team consists of 1 Healer, 1 Tank and the remaining space is filled with Attackers. This game is NOT turn based and **every character executes its actions simultaneously**.  Each Characters 2 different moves which can be executed: 1. Normal Attack 2. Special attack.   * **Healer**: Healer is a key character which holds the key to healing the players health and mana.   + Normal Attack: Randomly selects 1 team member and heals their mana   + Special Attack: Selects all alive team members and heals their health * **Tank**: Tank is the bulkiest character.   + Normal Attack: Randomly selects 1 opponent and reduces the health of that opponent by the attack power of tank.   + Special Attack: Select all alive team members and increments the attack power of all team members. * **Attacker**: Attacker are the common character in the game   + Normal Attack: Randomly selects 1 opponent and reduces the health of that opponent by the attack power of Attacker   + Special Attack: Select all alive opponents and reduces the health of those opponents by the attack power of Attacker   The game ends when every character in 1 team dies. |
| Justification for multi-threading | This program has to be coded in a multithreaded manner as it involves many characters attacking and performing **different actions at the same time**. If coded in a single threaded manner, the winner of the game will be determined by which team attacks first and **will always result in the same way** (Unless using Random class to randomize the team which starts) or it would make the program run **for a very long time**. By making this a multithreaded program, the winning team of the game is not pre-defined, and the attacks and actions are **performed at real time**. Not only that, the speed at which the game is played will **actually simulate a MOBA game such as League of Legends**. |
| Transactional integrity | There were plenty race conditions to be cleared in this program as it involves actions like *reducing health/mana, increasing mana* and much more. These were the main race conditions to be fixed   1. **Receiving Damage/Reducing Mana**: There was a possibility of race condition in a situation where Character A and B attacks Character C at the same time. If Character B takes the CPU time while Character A is decreasing the health of Character C, it leads to Race Condition. This was fixed by using a **Read Write Lock** on each attribute (Health, Mana and attack power) 2. **Increasing Health while receiving Damage**: There was a possibility where a Healer heals Attacker B while Attacker B was getting attacked by Attacker C, this may lead to race condition but again was fixed using a **Read Write Lock**. 3. **Wait() & Notify()** – The program uses wait() and notify() methods to pause all Character Threads in order for the game announcer to display mid game character information. The wait function has to be synchronized due to possibility of spurious wake ups from the waiting state.   All of the race conditions were fixed. |
| Performance | The performance was generally improved by the fact that multithreading was used. The “Liveness” of the program was maximised with the usage of Read/Write locks on ONLY the critical sections such as Reducing health/mana. There were no class locks or locks which lock a massive chunk of code. The code inside the locks were kept to bare minimum. |
| Innovation | This game is pretty innovating to make using Java Multithreading as it is a super popular game genre in todays world. Providing a game like this using Multithreading would not only increase the speed of the game but also increase the liveliness of it. |
| Adherence to coding conventions & good practices | 1. Usage of Abstract Class which maximises code re-usability (GameCharacter.java) 2. Usage of properly encapsulated classes using “private” fields with getter methods 3. Usage of Packaging for proper maintenance 4. Usage of Comments and proper indentations. |
| Good documentation | Used comments |
| Evidence of exploration | There are a few things which were added such as:   1. Wait() & Notify() – These two methods were used to pause threads until another thread finished its execution (MidGameAnnouncer) 2. ThreadLocalRandom – Used instead of the Random class as it increases performance. Random class is unnecessarily synchronized even if there was an instance of random. ThreadLocalRandom increases performance. |
| Others | *<Any other comments here?>* |
| Acknowledgements/References | I got help from the TA – Dion Ang  1. <https://stackoverflow.com/questions/2779484/why-must-wait-always-be-in-synchronized-block>  2. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html |