Author: Samukelile Jama

Date: 31 August 2023

Multithreaded Concurrent ClubSimulation Report

Introduction

Concurrency and thread safety are vital aspects of application development involving multiple concurrent threads. This report delves into the implementations for various classes within a club simulation application.

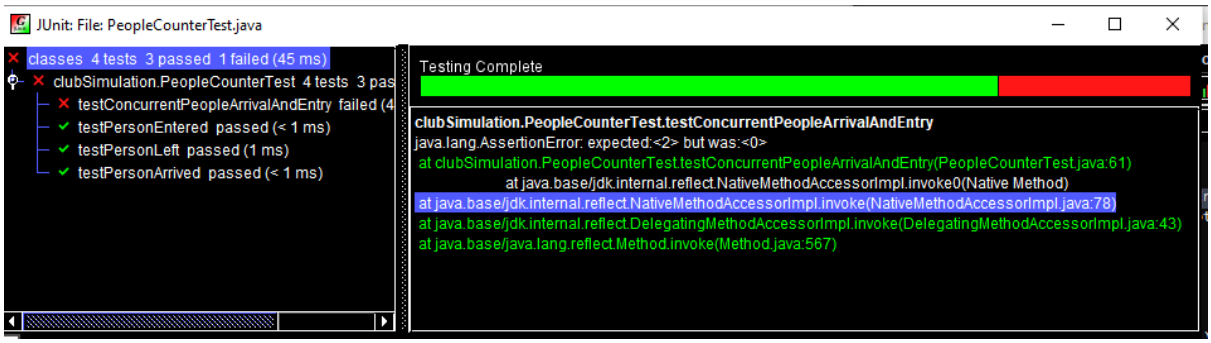
PeopleCounter, GridBlock, ClubGrid, and Clubgoer classes, as well as the distinctions between the two versions of the ClubSimulation class.

PeopleCounter Class

The implementation of the PeopleCounter class centres on resolving concurrency issues through synchronization mechanisms. The initial version suffered from data inconsistency and race conditions due to multiple threads simultaneously accessing shared counter values. In the improved version, methods responsible for retrieving counter values (getInside(), getTotal(), getLeft(), getMax()) are synchronized. This ensures exclusive access by one thread at a time, thereby preventing conflicts and discrepancies.

The personArrived() method, which modifies the shared counter, is also synchronized. This safeguard prevents potential data corruption that could arise when multiple threads modify the counter simultaneously. Additionally, the introduction of a notify() call in the personEntered() method establishes a notification system for waiting threads. This enhancement fosters better thread coordination.

JUnit testing was pivotal in confirming the simulation's reliability, particularly in relation to diverse thread interactions. Notably, the testing phase exposed an unforeseen anomaly in the PeopleCounter class when dealing with simultaneous patron arrival and entry, arising from a concurrency-related glitch. This emphasized the critical requirement for robust synchronization mechanisms to uphold data integrity. The report is augmented by accompanying images that visually depict the test outcomes. While one image validates successful synchronization management through four passing test cases, another image highlights a test failure, underscoring the importance of addressing such issues for an accurate and dependable simulation.



GridBlock Class

The implementation of the GridBlock class addresses thread safety concerns by synchronizing critical methods. The initial version lacked synchronization, opening doors to race conditions and data inconsistencies. In the upgraded version, the release() and occupied() methods, involving the isOccupied variable, are synchronized. This ensures that only one thread can modify the occupancy status of a block or check its occupancy state at any given time.

Non-shared state methods like getX() and getY() remain untouched, as they do not involve shared data and are inherently thread-safe. These changes guarantee accurate maintenance and controlled access to the occupancy status of each block in a multi-threaded environment.

ClubGrid Class

The second implementation of the ClubGrid class enhances thread safety and concurrency management through synchronization and coordination mechanisms. To prevent concurrency issues, a synchronization lock (entranceLock) is introduced for synchronized access to the entrance block. A boolean flag (atCapacity) is utilized to effectively control the club's capacity. The incorporation of wait() and notifyAll() facilitates coordinated thread behaviour during entry and exit operations. This minimizes potential conflicts and ensures orderly handling of these actions.

These modifications effectively address concurrency challenges, prevent capacity exceedance, and create a more synchronized environment for the club simulation. The outcome is improved reliability and accuracy in the simulation's behaviour.

Clubgoer Class

The provided version of the Clubgoer class introduces significant synchronization and responsiveness improvements. The initial version contained empty methods, which are now replaced with active synchronization mechanisms. The checkPause() method ensures that threads respond appropriately to the simulation's pause state before continuing execution. The startSim() method delays a thread's activities until the simulation officially starts. This enhances synchronization and realism by aligning thread behaviour with the simulation's state. These changes lead to better coordination and interaction between threads, resulting in an enhanced and more realistic simulation of clubgoers' activities within the club environment.

ClubSimulation Class

The provided code includes two versions of the ClubSimulation class, both simulating a club environment with multiple patrons. The second version enhances user interaction by implementing effective start and pause functionality through GUI buttons.

In the first version, the "Start" and "Pause" buttons lacked functionality, while the "Quit" button could exit the program.

The provided version improves user control by introducing proper start and pause functionality. The "Start" button sets a boolean flag (started) to initiate the simulation, while the "Pause" button toggles the pause flag for pausing and resuming the simulation. The button text dynamically changes between "Pause" and "Resume" to reflect the pause state. The "Quit" button retains its functionality to exit the program.

JUnit Testing and Justification

The results of the JUnit testing process instilled confidence in the robustness of the implemented synchronization mechanisms and thread coordination strategies. The tests highlighted anomalies or unexpected behaviours, enabling prompt resolution before deployment. By ensuring the simulation's intended behaviour in various scenarios, JUnit testing contributes to a more reliable and accurate simulation.

Lessons learned from these implementations underscore the importance of synchronized access to shared resources, like counters and occupancy statuses, to maintain data integrity and prevent conflicts. The distinction between shared and non-shared state methods emphasizes the need to optimize synchronization for performance efficiency. The use of synchronization mechanisms, such as locks and synchronized methods, contributes to controlled and synchronized thread execution, eliminating race conditions and ensuring orderly operation.

Ensuring liveness and preventing deadlock in this club simulation was achieved by combining strategic synchronization, resource management, and coordinated thread behaviour. Locks and synchronized methods shielded critical code sections from concurrent access, reducing deadlock risk from resource contention. Proper allocation and management of shared resources prevented unnecessary thread blocking, promoting liveness.

Conclusion

The second implementations of various classes in the club simulation exhibit substantial improvements in thread safety, synchronization, and concurrency management. By addressing potential concurrency issues, introducing synchronization mechanisms, and enhancing thread coordination, these implementations create a more reliable, realistic, and controlled simulation of a club environment with multiple threads. Proper synchronization and coordination mechanisms are pivotal for data integrity and accurate behaviour in multi-threaded applications.