
# Club Simulation 2023

24 AUGUST 2023

**Authored by: Best Nkhumeleni**

**Authored by: NKHBES001**

# Contents

# Introduction

Concurrency in computer science refers to the ability of a computer system to handle multiple tasks or processes simultaneously, without necessarily executing them at the exact same time. Concurrency is essential for efficiently utilizing modern multi-core processors and for designing systems that can handle a large number of users or tasks concurrently.

This report comprehensively outlines the creation and simulation of a bar environment, incorporating the principles of concurrency. The simulation encompasses patrons and a bartender and focuses on concurrent interactions.

# Methodology

## How the program works

For each patron:

- They walk into the club one at a time.
- Head straight to the bar, where they wait for Andre the bar man to serve them.
- If they haven't been served, they don't leave the bar.
- They then wonder around the bar, dance a bit.
- If they want a drink again, they go back to the bar, and wait to be served.

For Andre the barman:

- He moves from side-to-side checking if there is anyone at the bar.
- If he finds someone, he stops near them for a bit and serves them.

## How each rule was adhered to

• **The Start button initiates the simulation.**

- Each instance of clubgoer and Andre have an AtomicBoolean named start, which is set to false initially.
- When the start button is pressed, a for loop goes through all the patrons and sets the start Boolean to true.

• **The Pause button pauses/resumes the simulation.**

- The way the pause button is set up is that, there is a shared Boolean variable called pause, that instantiated in the club simulation class.
- The pause button is initially instantiated as false.
- Within the club goer class. Each thread has to check if the program is paused.
- If the program is paused, they go into a while loop and wait for the pause variable to change.
- When the pause button Is pushed an odd number of times, it sets the pause variable to true and notifies all waiting threads.
- When the pause button Is pushed an even number of times, it sets the pause variable to false and notifies all waiting threads.

- **The Quit button terminates the simulation (it does).**
  - This works
- **Patrons enter through the entrance door and exit through the exit doors.**
  - Each patron checks what type of GridBlock they are interacting with before they interact with it.
- **The entrance and exit doors are accessed by one patron at a time.**
  - Before a thread enters the club, they go into a spin wait if the club entrance is occupied. When the entrance is no longer occupied, they get notified.
  - Before a thread exits the club, they check if the exit is occupied, in which case they wait for a few milliseconds and check again, then they proceed to exit.
- **The maximum number of patrons inside the club must not exceed a specified limit.**
  - In the PeopleCounter class, in the people entered method (which has been synchronized to prevent unwanted interleaving).
  - Each thread has to check if the club is overcapacity or not
  - If it isn't overcapacity or at capacity they'll get into the club
  - If it is overcapacity or at capacity the thread will wait on the PeopleCounter instance.
  - In the personLeft method of the PeopleCounter class, when someone exits the threads waiting on the PeopleCounter instance are all notified.
- **Patrons must wait if the club limit is reached, or the entrance door is occupied.**
  - Before a thread enters the club, they go into a spin wait if the club entrance is occupied. When the entrance is no longer occupied, they get notified.
  - In the PeopleCounter class, in the people entered method (which has been synchronized to prevent unwanted interleaving).
  - Each thread has to check if the club is overcapacity or not
  - If it isn't overcapacity or at capacity they'll get into the club
  - If it is overcapacity or at capacity the thread will wait on the PeopleCounter instance.
  - In the personLeft method of the PeopleCounter class, when someone exits the threads waiting on the PeopleCounter instance are all notified.

**• Inside the club, patrons maintain a realistic distance from each other (one per grid block).**

- If two patrons want to access a block and its occupied, they go to a different block

**• Patrons move block by block and simultaneously to ensure liveness.**

- Each patron and barman can move more than one block at a time.

**• The simulation must be free from deadlocks.**

- If two threads want to go to each other's block when they are next to each other. This can cause a deadlock as they are both trying to access what the other has without giving up what they have to the other block. To counter this **when a block tries to access an occupied block, they give up and try to access a different block instead.**

- When a patron is trying to come in and a patron is trying to leave at the same time. This can cause a deadlock as they will both be trying to access the PeopleCounter and change the tally. To counter act this, **the PeopleCounter class instance is locked in the clubgoer trying to exit** i.e. of a clubgoer wants to leave, they access PeopleCounter and lock everyone else out until they are done decrementing people inside.

## Andre design approach

The implementation includes a distinct class called "AndreTheBarman," derived from the "ClubGoer" class and utilizing the same "start," "pause," and "resume" methods. Within the "AndreTheBarman" class, two separate while loops are employed to determine his position within the club. If he is not at either end, he moves left until reaching one end, then shifts to the right until the opposite end is reached. At each movement step, he assesses the presence of patrons at the bar. Upon encountering a patron, he halts, serves them, and subsequently resumes his movement.

# References

Git repo: https://github.com/BestNkhumeleni/BarSimulation

All this code is stored in a private git repository, to prevent plagiarism.