DOCUMENTATIE

TEMA *2 QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS*

NUME STUDENT: Moga Eduard Mihai
GRUPA: 30224

CUPRINS

- Obiectivul temei

Scopul proiectului este dezvoltarea unei aplicații de gestionare a cozilor, care atribuie unor cozi(Queues) client(Clients) în așa fel încât timpul de așteptare să fie minimizat. Interfața grafică va fi intuitivă pentru utilizatori, facilitând accesul la funcționalități.

- *Procesul începe cu analiza nevoilor utilizatorilor și stabilirea cerințelor (se va detalia în capitolul: 2. Analiza problemei, modelare, scenarii, cazuri de utilizare).*
- *Urmează proiectarea detaliată a arhitecturii software și a interfeței. (se va detalia în capitolul: 3. Proiectare)*
- *Implementarea se concentrează pe scrierea codului și integrarea funcționalităților. (se va detalia în capitolul: 4. Implementare)*
- *Testarea este esențială pentru asigurarea corectitudinii și stabilității aplicației. (scenariile pentru testare vor fi prezentate în capitolul: 5. Concluzii)*

Scopul final este dezvoltarea unei aplicații robuste și ușor de utilizat pentru gestionarea eficientă a cozilor.

Analiza problemei, modelare, scenarii, cazuri de utilizare

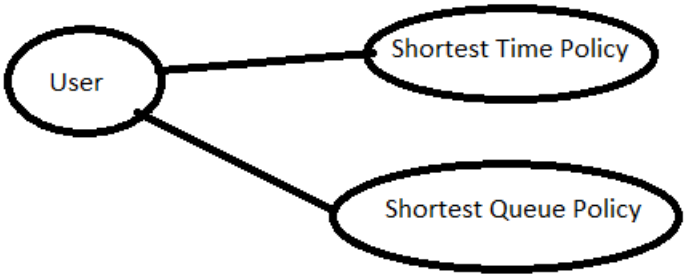
Cerințe Funcționale:

- Shortest Time Policy: Optiunea care va face alegerea distribuirii clientilor in functie de timpul minim din cozi.
- Shortest Queue Policy: Optiunea care va face alegerea distribuirii clientilor in functie de marimea minima a cozilor.
- Interfață Grafică: O interfata grafica usor de inteles care permite utilizatorului sa introduca informatii despre simulare si sa isi aleaga politica dorita.
- Interfata Animata: O interfata grafica ce se modifica real-time si permite utilizatorului sa observe evolutia cozilor.

Cerințe Non-Funcționale:

- Performanță: Aplicatia trebuie sa raspunda rapid si sa se actualizeze rapid in timpul interactiunilor utilizatorului.

- Ușurință de Utilizare: Interfata grafica trebuie sa fie prietenoasa si intuitiva pentru utilizatori de toate nivelurile de experienta.
- Fiabilitate: Aplicația trebuie să fie robustă și să gestioneze corect diversele scenarii de utilizare.



Use-case-urile sunt prezentate mai jos, având utilizatorul ca actor principal:

Shortest Time Policy: Scenariu de Succes:

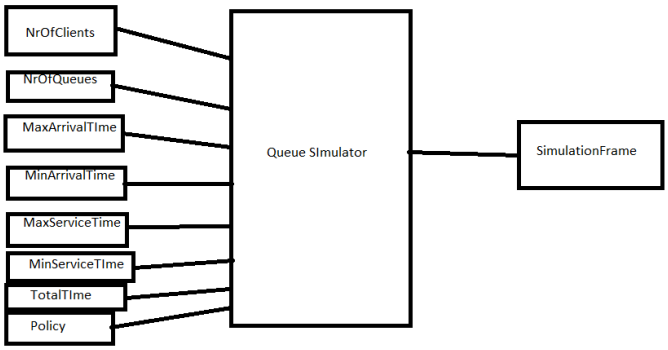
1. Utilizatorul introduce datele simulării în interfața grafică.
2. Utilizatorul selectează "Shortest Time Policy" din meniul derulant și apasă butonul de Start.
3. Se deschide un nou panou în care utilizatorul poate urmări în timp real evoluția cozilor, împreună cu statistici precum Peak Hour, Average service time, Average waiting time.

Shortest Length Policy: Scenariu de Succes:

1. Utilizatorul introduce datele simulării în interfața grafică.
2. Utilizatorul selectează "Shortest Length Policy" din meniul derulant și apasă butonul de Start.

3. Se deschide un nou panou în care utilizatorul poate urmări în timp real evoluția cozilor, împreună cu statistici precum Peak Hour, Average service time, Average waiting time Scenariu Alternativ pentru Toate Use-case-urile: Nu sunt introduse date valide pentru simulare.

Proiectare



Implementarea funcționalităților calculatorului polinomial necesită gestionarea a 8 intrări esențiale: parametrii simulării și tipul politicii (Shortest Queue sau Shortest Time).

Pentru a gestiona aceste aspecte, am folosit un architecture design MVC , împărțind aplicația în trei componente distincte:

Model: Aici sunt încapsulate modelele datelor de bază, cum ar fi Clientul și Queue. Clasa Client reprezintă clientul din coadă, iar clasa Queue reprezintă coada în sine și implementează interfata Runnable. Această structură permite gestionarea eficientă a clienților și cozilor.

View (SetupFrame și SimulationFrame): Aceasta componenta se ocupa de afisarea a datelor necesare, utilizatorului. SetupFrame este clasa principala care se ocupa cu prelucrarea datelor de la tastatura, urmand sa fie trimise Controller-ului care se ocupa de procesarea acestora. In final dupa apararea butonului “start simulation” ultilizatorul va primi pe ecran simularea propriu-zisa cu datele introduse.

Controller (QueueStrategy, TimeStrategy, Scheduler, SelectionPolicy, SimulationManager, Strategy): Clasa SimulationManager coordonează fiecare task în parte și îl trimite unei cozi în funcție de politica selectată. Clasa SelectionPolicy este de tip enum pentru o mai bună vizibilitate a politicii selectate în cod. Clasa Scheduler pornește firele de execuție pentru fiecare server în parte și, în funcție de politica selectată, trimite task-ul către un server anume. Pentru a ajunge la o abstractizare cât mai mare, ne folosim și de o interfață "Strategy" implementată de clasele **QueueStrategy** și **TimeStrategy** pentru a implementa diferite modalități de a adăuga un task la serverul trimis ca parametru.

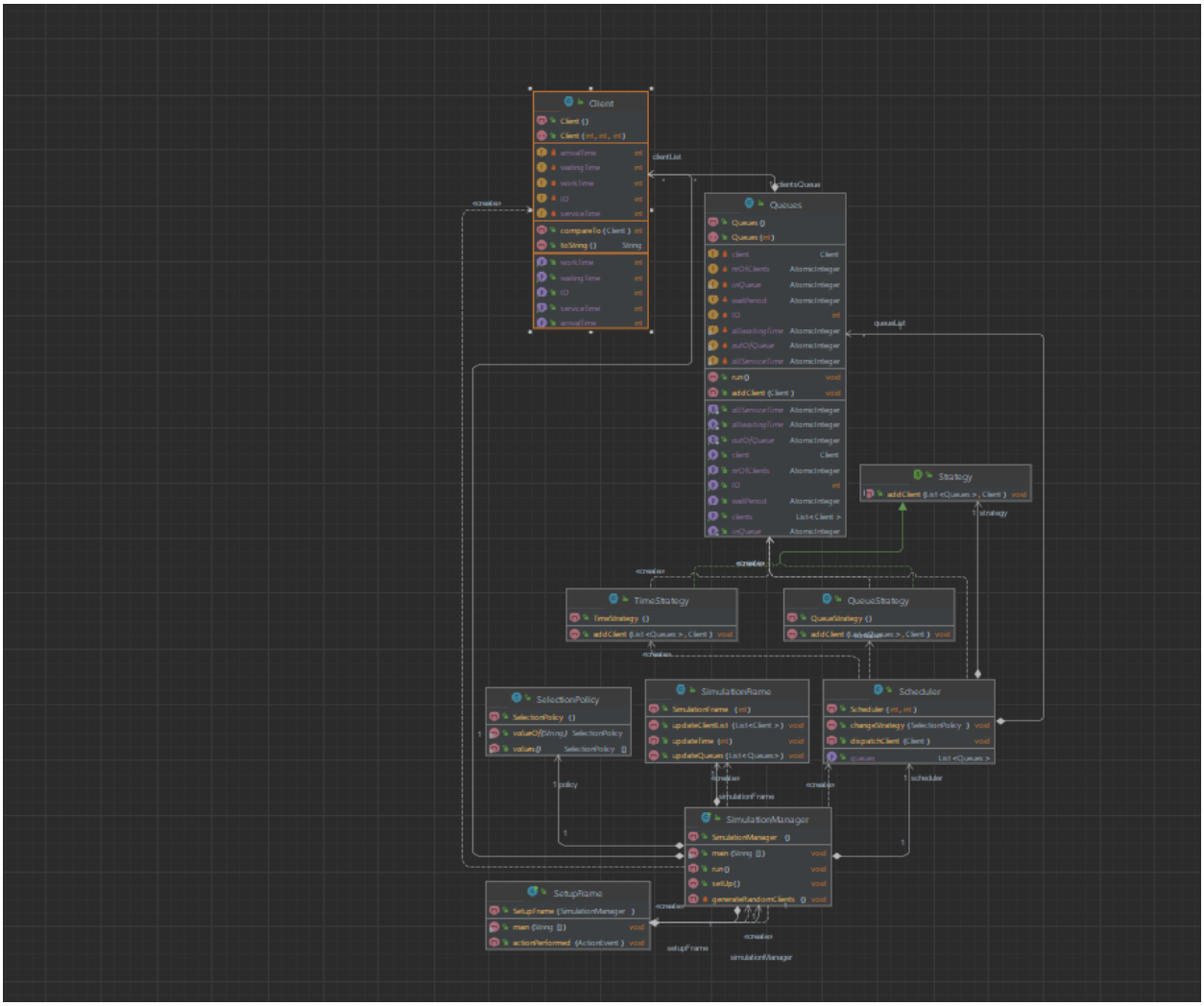
Prin implementarea acestui model arhitectural, putem gestiona eficient interacțiunea utilizatorului cu aplicația, separând View, BusinessLogic și gestionând intrările utilizatorului prin intermediul Controllerului.

Pe lângă cele trei pachete menționate, am introdus și următoarea clasă:

Clasa Main: În această clasă se găsește punctul de intrare în aplicație, de unde este deschis simulatorul de cozi. Clasa Main este responsabilă pentru inițializarea și lansarea aplicației.

Structura de date pentru reprezentarea serverului este un blockingQueue (de tipul Client), fiecare Client având un arrivalTime și un serviceTime (de tip întreg), iar pentru reprezentarea cozilor din Scheduler este un List (de tipul Queue).

Diagrama UML a proiectului este urmatoarea:



• Implementare

Pentru a asigura o înțelegere completă a funcționalității Calculatorului Polinomial, voi descrie detaliat clasele menționate în secțiunile anterioare:

1. Clasa Client din pachetul **org.example.MODEL** reprezintă un task în cadrul unui sistem de simulare a cozilor.

Atributele clasei sunt:

- **int ID**: identificatorul unic al task-ului.

- **int arrivalTime**: timpul de sosire al task-ului.
- **int serviceTime**: timpul necesar pentru a finaliza task-ul.
- **int waitingTime**: timpul de așteptare al task-ului în coadă.
- **int workingTime**: timpul pe care il petrece pentru procesare.

Constructorul clasei **Client** primește ca parametri id-ul task-ului, timpul de sosire și timpul de serviciu, inițializând variabilele membru ale obiectului.

Metodele publice ale clasei includ:

- **Client(int ID, int arrivalTime, int serviceTime): Constructor care inițializează un obiect Client cu un ID dat, timpul de sosire și timpul de serviciu specificat.**
- **Client(): Constructor implicit care inițializează un obiect Client cu un ID implicit (-1).**
- **getArrivalTime(): Returnează timpul de sosire al clientului.**
- **getID(): Returnează ID-ul clientului.**
- **getServiceTime(): Returnează timpul de serviciu al clientului.**
- **toString(): Returnează o reprezentare sub formă de șir de caractere a obiectului Client, formată din ID-ul, timpul de sosire și timpul de serviciu.**
- **compareTo(Client o): Implementează metoda de comparație necesară pentru a sorta clienții după timpul de sosire.**
- **setServiceTime(int serviceTime): Setează timpul de serviciu al clientului la o valoare dată.**
- **getWorkTime(): Returnează timpul total de serviciu sau de lucru al clientului.**
- **setWaitingTime(int waitingTime): Setează timpul de așteptare al clientului la o valoare dată.**
- **getWaitingTime(): Returnează timpul de așteptare al clientului.**

Clasa **Client** implementează interfața **Comparable<Client>**, permițând compararea task-urilor pe baza timpului lor de sosire. Astfel, task-urile pot fi sortate în ordinea sosirii lor în sistem.

2. Clasa Queue din pachetul **org.example.MODEL** reprezintă un server în cadrul unui sistem de simulare a cozilor.

Atributele clasei sunt:

- **private BlockingQueue<Client> clientsQueue: O coadă blocantă care conține clienții care așteaptă să fie serviți în această coadă.**
- **private AtomicInteger waitPeriod: Un atom care reprezintă perioada de așteptare curentă a cozi. Este actualizat în timp ce clienții sunt adăugați și serviți.**
- **private AtomicInteger nrOfClients: Un atom care stochează numărul curent de clienți din coadă. Este actualizat în timp ce clienții sunt adăugați și serviți.**
- **private int ID: Un identificator unic al cozii.**
- **private Client client: O referință către clientul curent care este servit din coadă. Este actualizată în timpul procesului de servire a clienților.**

- **private static AtomicInteger allServiceTime: Un atom care stochează timpul total de serviciu pentru toți clienții din toate cozile.**
- **private static AtomicInteger allwaitingTime: Un atom care stochează timpul total de așteptare pentru toți clienții din toate cozile.**
- **private static AtomicInteger inQueue: Un atom care stochează numărul total de clienți care se află în cozi într-un moment dat.**
- **private static AtomicInteger outOfQueue: Un atom care stochează numărul total de clienți serviți din toate cozile.**

Constructorul clasei **Queue** primește ca parametru timpul de așteptare între procesarea fiecărui unitate de timp a task-urilor și inițializează coada de task-uri și numărul total de secunde de așteptare.

Metodele publice ale clasei includ:

- **Queues(int ID): Constructor care inițializează o coadă cu un ID dat și setează perioada de așteptare la 0.**
- **Queues(): Constructor implicit care inițializează o coadă și setează perioada de așteptare la 0.**
- **addClient(Client c): Adaugă un client în coadă. Actualizează perioada de așteptare și numărul total de clienți.**
- **run(): Implementează logica de tratare a clienților din coadă într-un fir de execuție separat. Extrage clienții din coadă și îi servește în ordinea sosirii lor.**
- **getClients(): Returnează o listă de clienți din coadă.**
- **getNrOfClients(): Returnează numărul de clienți din coadă.**
- **getWaitPeriod(): Returnează perioada de așteptare actuală a cozi.**
- **getID(): Returnează ID-ul cozii.**
- **getClient(): Returnează clientul curent servit din coadă.**
- **getOutOfQueue(): Returnează numărul total de clienți serviți.**
- **getAllServiceTime(): Returnează timpul total de serviciu pentru toți clienții.**
- **getAllwaitingTime(): Returnează timpul total de așteptare pentru toți clienții.**
- **getInQueue(): Returnează numărul total de clienți aflați în coadă.**

Clasa **Queue** implementează interfața **Runnable**, permițându-i să fie executată într-un fir de execuție separat. Aceasta preia task-uri din coadă pentru procesare și actualizează lista de task-uri procesate.

3. Clasa **SimulationManager** din pachetul **org.example.LOGIC** este responsabilă pentru gestionarea simulării sistemului de cozi.

Atributele clasei sunt:

- **public int timeLimit: Reprezintă limita de timp pentru simulare, specificată în unități de timp.**
- **public int maxProcessingTime: Reprezintă timpul maxim de procesare a unui client în simulare.**
- **public int minProcessingTime: Reprezintă timpul minim de procesare a unui client în simulare.**
- **public int nrOfQueues: Reprezintă numărul de cozi disponibile în simulare.**

- **public int nrOfClients: Reprezintă numărul total de clienți care vor fi generați pentru simulare.**
- **public int maxArriveTime: Reprezintă timpul maxim de sosire a unui client în simulare.**
- **public int minArriveTime: Reprezintă timpul minim de sosire a unui client în simulare.**
- **public SelectionPolicy policy: Reprezintă politica de selecție a cozilor pentru servirea clienților în simulare.**

Constructorul clasei primește parametrii necesari pentru inițializarea simulării și initializează planificatorul și listele de task-uri.

setUp(): Metodă pentru inițializarea datelor dorite pentru simulare. Creează un obiect Scheduler, schimbă politica de selecție a cozilor și generează clienți aleatorii.

generateRandomClients(): Metodă pentru generarea de clienți aleatori cu timp de sosire și timp de serviciu aleatorii. Acești clienți sunt sortați după timpul de sosire și adăugați în lista de clienți.

run(): Implementează logica principală a simulării. Parcurge intervalul de timp specificat și actualizează starea clienților și a cozilor în fiecare unitate de timp. De asemenea, calculează și înregistrează statistici precum ora de vârf, timpul mediu de serviciu și timpul mediu de așteptare într-un fișier de ieșire..

4. Clasa **Scheduler** din pachetul **org.example.LOGIC** este responsabilă pentru gestionarea alocării task-urilor către servere în funcție de politica de selecție specificată.

Atributele clasei sunt:

- **queueList: Este o listă de obiecte Queues, reprezentând cozile gestionate de acest planificator.**
- **maxQueues: Este un întreg care indică numărul maxim de cozi permise de către acest planificator.**
- **maxClients: Este un întreg care specifică numărul maxim de clienți în cadrul sistemului de cozi gestionat de acest planificator.**
- **strategy: Este o referință către o strategie de selecție a cozilor, inițializată cu o strategie implicită (QueueStrategy) în momentul construirii unui obiect Scheduler.**
- **executor: Este un ExecutorService utilizat pentru gestionarea firelor de execuție pentru fiecare coadă în sistem.**

Constructorul clasei primește ca parametri numărul maxim de servere și timpul de așteptare între procesarea fiecărui unitate de timp a task-urilor și inițializează lista de servere și strategia de alocare a task-urilor.

Metodele publice ale clasei includ:

- **changeStrategy(SelectionPolicy policy)**: schimbă strategia de alocare a task-urilor în funcție de politica specificată.

- **checkFinished()**: verifică dacă toate serverele au terminat de procesat toate task-urile aflate în coada lor.
- **dispatchTask(Task t)**: trimite un task către serverele disponibile pentru alocare în conformitate cu strategia specificată.
- **getServers()**: returnează lista de servere disponibile.

Clasa **Scheduler** integrează serverele și politica de selecție a acestora pentru a asigura o alocare eficientă și coerentă a task-urilor în cadrul sistemului de simulare a coziilor.

În clasa **SetupFrame** din pachetul org.example.View, sunt definite următoarele atribute:

grid: Un obiect GridBagConstraints utilizat pentru a specifica modul de poziționare și aliniere a componentelor în cadrul containerului.

clText, qText, simText, minAText, maxAText, minSText, maxSText: Obiecte JTextField utilizate pentru a permite utilizatorului să introducă valori pentru numărul de clienți, numărul de cozi, durata simulării, timpul minim și maxim de sosire al clienților și timpul minim și maxim de serviciu al clienților.

startSimulation: Un obiect JButton care reprezintă butonul "Start Simulation", utilizat pentru a iniția simularea atunci când utilizatorul dă clic pe el.

strategyList: Un obiect JComboBox care permite utilizatorului să selecteze strategia de selecție a cozilor pentru servirea clienților în timpul simulării.

simulationManager: O referință către un obiect SimulationManager care va gestiona simularea.

Aceste atribute sunt utilizate pentru a crea interfața grafică a ferestrei de configurare a simulării, permițând utilizatorului să introducă parametrii necesari pentru simulare și să înceapă simularea atunci când sunt completate toate câmpurile.

SetupFrame(SimulationManager simulationManager): Este constructorul clasei SetupFrame, care primește ca parametru un obiect de tip SimulationManager. Acest constructor inițializează toate componentele ferestrei de configurare a simulării și stabilește layout-ul și aspectul acesteia. De asemenea, adaugă ascultător pentru evenimentul de clic pe butonul "Start Simulation".

actionPerformed(ActionEvent e): Este o metodă implementată din interfața ActionListener, care este apelată atunci când utilizatorul dă clic pe butonul "Start Simulation". Această metodă preia valorile introduse de utilizator pentru parametrii simulării (numărul de clienți, numărul de cozi, durata simulării, timpul minim și maxim de sosire al clienților, timpul minim și maxim de serviciu al clienților și strategia de selecție a cozilor) și le transmite obiectului SimulationManager pentru a iniția simularea.

Aceste două metode sunt esențiale pentru gestionarea și inițializarea ferestrei de configurare a simulării și pentru comunicarea cu obiectul SimulationManager pentru inițierea simulării cu parametrii specificați de utilizator.

În clasa SimulationFrame din pachetul org.example.View, sunt definite următoarele atribute și metode:
Atribute:
private JPanel simulationPanel: Un panou în care sunt afișate cozile și clienții din simulare.
private Map<Integer, JLabel> queueLabels: Un map care asociază fiecărei cozi unui obiect JLabel, pentru afișarea informațiilor despre clienții din fiecare coadă.
private JLabel timeLabel: Un label pentru afișarea timpului curent al simulării.
private JLabel clientLabel: Un label pentru afișarea listei de clienți din simulare.
Metode:
SimulationFrame(int nrOfQueues): Constructorul clasei care inițializează fereastra de simulare. Construiește panoul de simulare și setează layout-ul și dimensiunile ferestrei.
updateQueues(List<Queues> queues): Metodă pentru actualizarea afișării cozilor în panoul de simulare. Primește o listă de obiecte Queues și actualizează label-urile corespunzătoare fiecărei cozi cu informațiile despre clienții din acea coadă.
updateTime(int time): Metodă pentru actualizarea afișării timpului curent al simulării.
updateClientList(List<Client> clients): Metodă pentru actualizarea afișării listei de clienți din simulare.
Aceste metode și atribute sunt utilizate pentru gestionarea afișării grafice a simulării sistemului de cozi. Atributul queueLabels este folosit pentru a actualiza informațiile despre cozile din simulare, iar celelalte atribute sunt utilizate pentru a afișa timpul și lista de clienți în cadrul ferestrei de simulare.

- # Rezultate

Test1:

```
Peak hour 23
Avg service time 2.5
Avg waiting time 0.0
```

Test2:

```
Peak hour 15
Avg service time 4.0
Avg waiting time 1.36
```

Test3:

Concluzii

Experiența acumulată în timpul implementării acestui proiect a adus la lumină câteva concluzii și idei relevante, care pot fi de folos în viitoarele proiecte:

- **Structurarea clară a obiectivelor:** Divizarea proiectului în obiective mai mici și mai ușor de gestionat, cum ar fi crearea claselor `Client`, Queue sau `SimulationManager`, ne-a permis să ne concentrăm pe implementarea fiecărei componente în mod individual și să avansăm treptat către obiectivul final.
- **Utilizarea adecvată a structurilor de date:** Alegerea structurilor de date potrivite, cum ar fi `BlockingQueue` pentru gestionarea task-urilor sau listele pentru stocarea polinoamelor, a contribuit la eficiența și performanța aplicației noastre.
- **Abstractizarea și modularitatea:** Utilizarea claselor abstracte, interfețelor și a conceptelor de modularitate ne-au ajutat să abstractizăm și să izolăm funcționalitățile distincte ale aplicației. Acest lucru facilitează extinderea, întreținerea și testarea componentelor individuale ale sistemului.
- **Lizibilitatea și gestionarea codului:** Menținerea codului lizibil și bine structurat este esențială pentru o dezvoltare eficientă și colaborativă. Limitarea dimensiunii metodelor și a claselor, precum și respectarea principiilor de codificare și documentare, au contribuit la un cod mai ușor de înțeles și de întreținut.

Bibliografie

1. *https://dsrl.eu/courses/pt/*
2. *https://www.w3schools.com/java/java_threads.asp*
3. *https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html*
4. *https://www.geeksforgeeks.org/atomic-variables-in-java-with-examples/*
5.