DOCUMENTATIE

TEMA *2*

NUME STUDENT: Nicoara Cristian-Cătălin

GRUPA: 30228

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

[3. Proiectare 3](#_Toc95297887)

[4. Implementare 10](#_Toc95297888)

[5. Rezultate 16](#_Toc95297889)

[6. Concluzii 16](#_Toc95297890)

[7. Bibliografie 16](#_Toc95297891)

# Obiectivul temei

* Obiectivul principal al acestei teme este de a crea o aplicatie in care sa simulam intr-un mod cat mai eficient faptul ca clientii unei companii sunt imaprtiti pe cozi pentru a fi deserviti, iar pana a ajunge randul lor, trebuie sa astepte toti oamenii din fata.
* Obiective secundare :

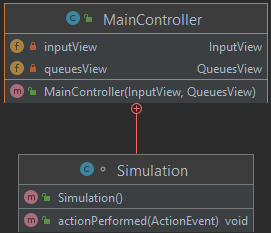
1. Crearea unei clase pentru client (cap.4)
2. Crearea unei clase pentru cozi, care sa reprezinte threaduri, pentru a lucra cat mai rapid sim ai eficient (cap.4)
3. Crearea uneia, sau a mai multor clase de control a simularii, care impart clientii in cel mai bun mod posibil (cap.4)
4. Crearea interfetelor grafice (GUI) (cap.4)

# Analiza problemei, modelare, scenarii, cazuri de utilizare

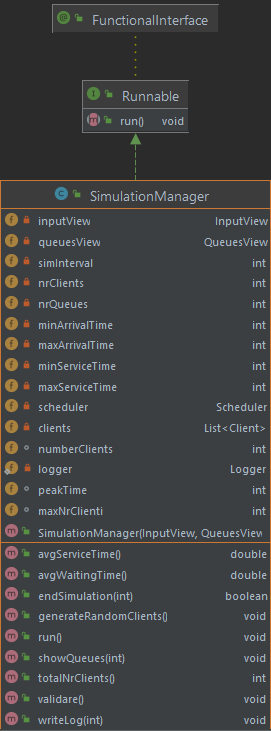
* Problema principala a temei este de a sincroniza firele de lucru ( thread-urile ), pentru a nu avea cazuri in cand doua sau mai multe fire de lucru lucreaza cu acelasi obiect. De exemplu , in aceasta tema , din cauza nesincronizarii thread-urilor un client poate fi pus in mai multe cozi, sau cand se calculeaza average waiting time, sa se adune timpul de asteptare a unui client de mai multe ori la mai multe cozi , ceea ce ar duce la rezultatae gresite. Pentru a rezolva aceasta problema , am pus “synchronized” la metoda dispatchClient(), astfel ea se apeleaza doar daca este libera, adica nu este utilizata de alt thread , deci se adauga clientul doar o data , in coada cu timpul de asteptare minim. De asemenea, in clasa Queue, am folosit BlockingQueue si AtomicInteger pentru a fi sigur ca fiecare coada are atributele unice, adica in calcularea timpului de asteptare la o coada, nu intra si clientii din alta coada. Aceste tipuri de date au acelasi rol pentru atribute ca si “synchronized” are pentru metode.

# Proiectare

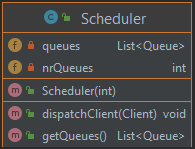
* **Package: controller**
  1. **Class: Main Controller**
* Aceasta clasa controleaza aparitia interfetelor. La apasarea butonului “View Queues Evolution ”, inchide prima interfata si o porneste pe a doua. De asemenea portneste firul de lucru al clasei SimulationManager.
* Diagrama :



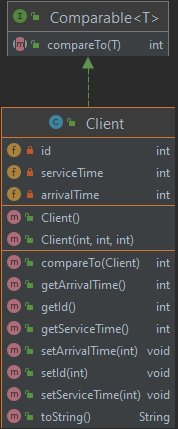
* 1. **Class: SimulationManager**
* Aceasta clasa se ocupa de generarea clientilor, apelarea metodei de adaugare a clientilor la momentul ptrivit timpului de intrare si scrie datele in interfata.
* Diagrama :



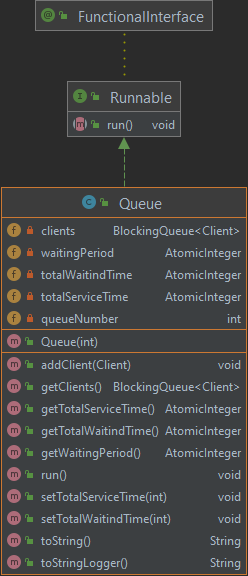
* 1. **Class: Scheduler**
* Adauga clientii la coada cu cel mai mic timp de asteptare (waiting time)
* Diagrama :



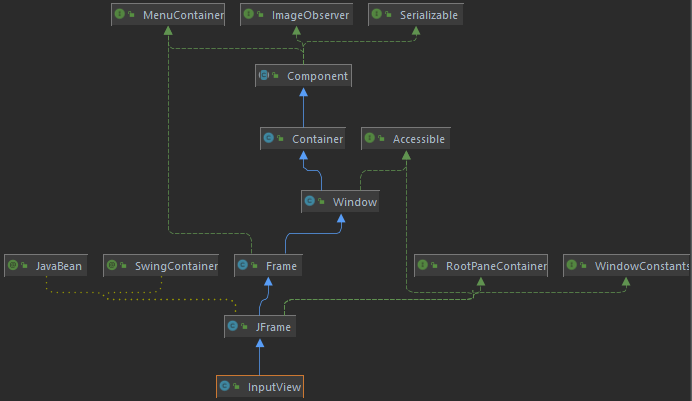
* **Package: model**
  1. **Class: Client**
* Aceasta clasa contine atributele unui client (id, serviceTime, arrivalTime) si implementeaza metoda compareTo(), pentru a putea sorta coada
* Diagrama :



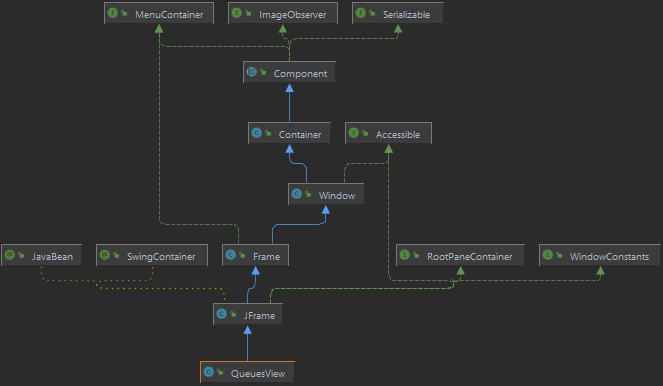
* 1. **Class: Queue**
* Aceasta clasa reprezinta coada in aplicatie, ce implementeaza interfata Runnable, deci este un thread. Contine o Coada de clienti de tip BlockingQueue, waitingPeriod, totalWaitingPeriod si totalServiceTime de tip AtomicInteger. Toate sunt de acest tip pentru ca atunci cand o coada foloseste unul dintre attribute, alte cozi sa nu-l poata folosi. Implementeaza metoda de adaugare a unui client in coada si metoda run(), specifica pentru threaduri.
* Diagrama :



* **Package: view**
  1. **Class: InputView**
* Aceasta clasa reprezinta interfata unde utilizatorul introduce datele de intrare, precum numarul de client, numarul de cozi etc. si apasa butonul “View Queues Evolution ” pentru a incepe simularea si a trece la a doua interfata.
* Diagrama :



* 1. **Class: QueuesView**
* Aceasta clasa reprezinta interfata unde se poate observa precesul de simulare, la fiecare perioada de timp fiind scrise cozile si clientii corespunzatori fiecareia.
* Diagrama :



* **Class: MainClass**
  + Aceasta clasa instantiaza interfetele grafice, pe care le transmite ca argumente constructorului clasei MainController.
* **Packages**
* controller

1. MainController
2. Scheduler
3. SimulationManager

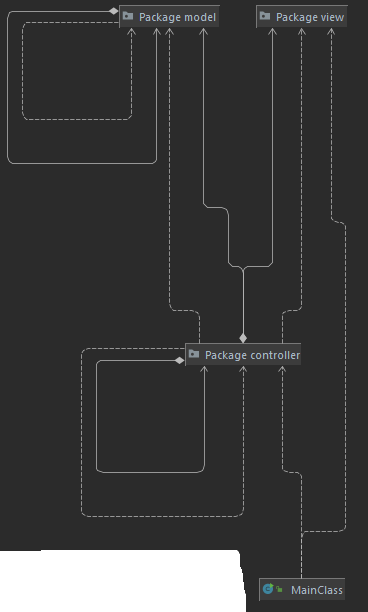
* model

1. Client
2. Queue

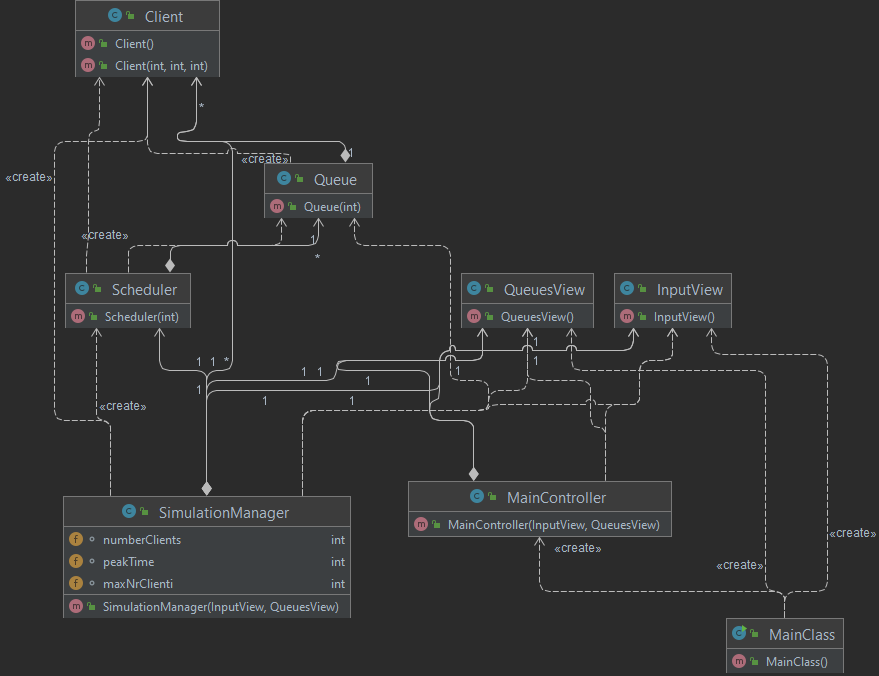
* view

1. InputView
2. QueuesView

* Diagrama :

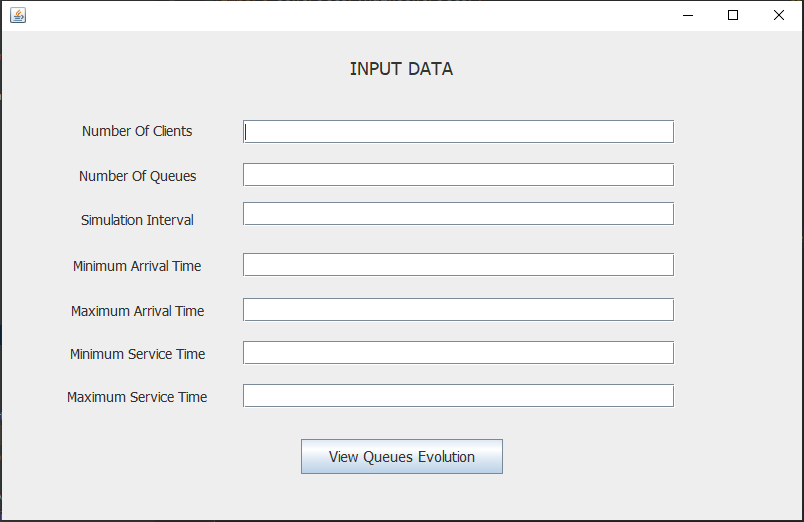
****

* **Diagrama UML**

****

# Implementare

* 1. **Clasa MainController**
* Atribute
* inputView de tip InputView, ce reprezinta interfata unde se introduce datele de intrare.
* queuesView de tip QueuesView, ce reprezinta interfata unde se poate observa datele si evolutia cozilor in timpul simularii
* Metode
* public MainController(), ce reprezinta contructorul clasei. In contructor se mai adauga listener pentru butonul din InputView. Metoda de *actionPerformed( ActionEvent e )* se afla intr-o clasa interioara, unde se inchide prima interfata si se porneste a doua daca au fost introduce date in toate text field-urile. De asemenea se instantiaza clasa SimulationManager si se creeaza si se porneste un fir de lucru cu aceasta clasa.
  1. **Clasa Scheduler**
* Atribute
* Queues de tipul *List<Queues>* : ce reprezinta o lista de cozi, fiecare coada avand anumiti client.
* nrQueues de tipul *int* :unde memoreaza numarul de cozi.
* Metode
* public Scheduler(int nrQueues) : care este contructorul clasei date. In acesta se creeaza lista de cozi, si cu fiecare coada se creeaza un nou fir de lucru care este pornit, apeland metoda *start()*.
* public List<Queue> getQueues() : ce este getter pentru atributul queues.
* public synchronized void dispatchClient(Client client) : ce adauga clientul in coada cu cel mai mic timp de asteptare (waitingTime). Intai gasesete aceasta coada folosind for each, iar apoi tot cu un for each, se adauga clientul in coada gasita. De asemenea, inainte sa adauge clientul in coada, adauga la timul total de astepare a cozii timpul de asteptare in acel moment de timp si adauga la timpul total de deservire a cozii, timpul de deservire a clientului ce urmeaza sa fie adaugat.
  1. **Clasa SimulationManager**
* Atribute
* inputView de tipul *InputView* : ce reprezinta interfata cu datele de intrare.
* queuesView de tipul *QueuesView* : ce reprezinta interfata unde are loc afisarea datelor cozilor in timpul simularii.
* simInterval de tip *int* : unde se memoreaza timpul simularii introdus de utilizator in interfata numarul 1
* nrClients de tip *int* : unde se memoreaza numarul de clienti introdus de utilizator in interfata numarul 1
* nrQueues de tip *int* : unde se memoreaza numarul de cozi introdus de utilizator in interfata numarul 1
* minArrivalTime de tip *int* : unde se memoreaza timpul minim de intrare introdus de utilizator in interfata numarul 1, utilizat mai tarziu la generarea random a clientilor
* maxArrivalTime de tip *int* : unde se memoreaza timpul maxim de intrare introdus de utilizator in interfata numarul 1, utilizat mai tarziu la generarea random a clientilor
* minServiceTime de tip *int* : unde se memoreaza timpul minim de servire introdus de utilizator in interfata numarul 1, utilizat mai tarziu la generarea random a clientilor
* maxServiceTime de tip *int* : unde se memoreaza timpul maxim de servire introdus de utilizator in interfata numarul 1, utilizat mai tarziu la generarea random a clientilor
* scheduler de tip *Scheduler* : ce are rolul de a crea si porni firele de lucru cu clasa Queue si de a introduce clientul in coada corecta (cu cel mai mic timp de asteptare)
* clients de tip *List<Client>* : unde se memoreaza toti clientii ce urmeaza sa fie repartizati pe cozi
* numerClients de tip *int* : contine numarul initial de clienti, care se foloseste la calcularea timpului mediu de asteptare si a timpului mediu de sosire. Pe parcursul simularii, numarul de clienti scade si ajunge la 0, de aceea avem nevoie si de acest atribut, altfel am avea impartirea la 0.
* logger de tip *Logger* : care are rolul de a scrie intr-un fisier .txt datele simularii.
* Metode
* *public SimulationManager(InputView inputView, QueuesView queuesView)* : ce reprezita constructorul acestei clase, unde se scrie in variabilele de intrare, datele introduse de utilizator in prima interfata. De asemenea se instantiaza Scheduler si se apeleaza metoda *generateRandomClients()* .
* *public void generateRandomClients()* : ce creeaza nrClienti client ce au id de la 1 pana la numarul de client, iar timpul de intrare in coada si timpul de deservire se genereaza random folosind *Math.random()* intre intervalele din minim si maxim care au fost introduce de utilizator in prima interfata.
* *public Boolean endSimulation(int nrClients)* : aceasta metoda verifica daca este timpul sa terminam simularea. In acest proiect, simularea se termina daca sa ajuns la finalul timpului de simulare introdus de utilizator sau daca au fost deserviti toti clientii ce sunt in cozi. De exemplu daca timpul de simulare s-a terminat, dar inca mai sunt clienti in cozi ce asteapta sa fie deserviti, aplicatia va astepta pana cand si ei vor fi ascultati.
* *public void showQueues(int clock)* : care scrie in interfata grafica numarul doi, la fiecare secunda, ce clienti se afla in fiecare coada.
* *public void writeLog(int clock) :* care scrie in fisierul .txt datele simularii, la fel la fiecare secunde, folosind atributul logger si apeland *logger.info()*.
* *public double avgWaitingTime()* : care calculeaza timpul medie de astepare a unui client. Face suma dintre totalWaitingTime a fiecarei cozi si imparte la numarul de client.
* *public int totalNrClienti()* : care calculeaza numarul total de clienti in toate cozile la un anumit timp. Se foloseste pentru a calcula momentul de timp in care au fost cei mai multi clienti in cozi.
* *public double avgServiceTime()* : care calculeaza timpul mediu de deservire a unui client. Se face suma dintre timpul total de deservire a fiecarei cozi si se imparte la numarul de client.
* *public void run()* : Aceasta metoda este suprascrisa de aceasta clasa, deoarece ea implementeaza Runnable. Metoda verifica daca s-a ajuns la timul de intrare (arrival time) a unui client, si daca s-a ajuns, atunci il scoate din lista de clienti si il apeleaza metoda *dispatchClient* din clasa Scheduler, care il introduce in coada potrivita. De asemenea, aceasta verifica daca un client dintr-o coada a ajuns la timpul de deservire = 0, ceea ce inseamna ca a fost deservit, si il scoate daca. Metoda data mai caluleaza peakTime folosind metoda *totalNrClienti()* sim ai actualizeaza interfata numarul 2. La sfarsitul simularii, metoda *run()* afiseaza un pop-up cu mesajul de finisare si cele 3 statistici.
  1. **Clasa Client**
* Atribute
* id de tip *int* : ce reprezinta identificatorul unic al unui client
* serviceTime de tip *int* : ce reprezinta timpul de deservire a unui client
* arrivalTime de tip *int* : ce reprezinta timpul in care clientul intra in coada
* Metode
* *public Client(int id, int serviceTime, int arrivalTime)* : ce reprezinta contructorul principal al clasei Client
* *public Client()* : ce reprezinta un constructor gol
* *getter’s and setter’s* pentru fiecare atribut
* *public int compareTo(Client client)* : este o metoda din interfata Comparable suprascrisa pantru a putea sorta clientii dupa timpul de intrare in coada si dupa timpul de deservire
* *public String toString()* : metoda suprascrisa pentru a scrie frumos in fisierul .txt creat de Logger.
  1. **Clasa Queue**
* Atribute
* clients de tip *BlockingQueue<Client>* : care tine lista de clienti ce se afla intr-o coada
* waitingPeriod de tip *AtomicInteger* : care memoreaza timpul de asteptare a fiecarii cozi, pentru a sti in care coada sa se insereze clientul
* totalWaintingTime de tip *AtomicInteger* : care memoreaza timpul total de astepare a unei cozi si se va folosi la calcularea timpului mediu de asteptare
* totalServiceTime de tip *AtomicInteger* : care memoreaza timpul total de deservire a unei cozi si se va folosi la calcularea timpului mediu de deservire
* queueNumber de tip *int* : se folosete la scriere pentru a deosebi in care coada se duce un client
* Metode
* *public Queue(int queueNumber)* : constructorul clasei
* *getters and setter* pentru fiecare atribut
* *public void run()* : metoda ce este suprascrisa, fiind implementata interfata Runnable. Aceasta scoate ia elementul ce este deservit la moment si pune threadul in strea de sleep pe durata deservirii.
* *public String toStringLogger()* : se foloseste la scrierea frumoasa si pe inteles in fisierul .txt creat de logger
* *public String toString()* : metoda suprascrisa care ajuta la scrierea in interfata grafica numarul 2
  1. **Clasa InputView**
* Atribute
* nrClientiField de tip *JTextField* : campul unde utilizatorul va scrie numarul de clienti
* nrQueuesField de tip *JTextField* : campul unde utilizatorul va scrie numarul de cozi
* simIntervalField de tip *JTextField* : campul unde utilizatorul va scrie timpul de simulare
* minArrivalTimeField de tip *JTextField* : campul unde utilizatorul va scrie timpul minim de sosire
* maxArrivalTimeField de tip *JTextField* : campul unde utilizatorul va scrie timpul maxim de sosire
* minServiceTimeField de tip *JTextField* : campul unde utilizatorul va scrie timpul minim de deservire
* maxServiceTimeField de tip *JTextField* : campul unde utilizatorul va scrie timpul maxim de deservire
* queuesEvolutionBtn de tip *JButton* : butonul care la apasare va deschide cea de-a doua interfata si va porni simularea
* Metode
* *public InputView()* : constructorul care creeaza interfata
* *getter* pentru atribute care se va folosi la verificarea daca toate campurile au fost completate
* *public int getNrClients()* : returneaza numarul introdus de utilizator scris in campul dat
* *public int getNrQueues()* : returneaza numarul introdus de utilizator scris in campul dat
* *public int getSimInterval()* : returneaza numarul introdus de utilizator scris in campul dat
* *public int getMinArrivalTime()* : returneaza numarul introdus de utilizator scris in campul dat
* *public int getMaxArrivalTime()* : returneaza numarul introdus de utilizator scris in campul dat
* *public int getMinServiceTime()* : returneaza numarul introdus de utilizator scris in campul dat
* *public int getMaxServiceTime ()* : returneaza numarul introdus de utilizator scris in campul dat
* *public void addButtonListener()* : metoda care asculta actiunea butonului de trecere la cealalta interfata
* *public void showMessage(String message)* : aceasta metoda arata un pop-up cu un mesaj primit ca atgument
  1. **Clasa QueuesView**
* Atribute
* simArea de tim *JTextArea* : campul unde se va scrie datele simularii
* Metode
* *public QueuesView()* : contructorul care creeaza interfata de simulare
* *public void setSimArea()* : metoda care va scrie datele simularii
* *public void showMessage(String message)* : aceasta metoda arata un pop-up cu un mesaj primit ca atgument
* *public void refresh()* : metoda care sterge sectiunea unde se scrie dupa fiecare secunda
  1. **Clasa MainClass**
* Are doar metoda main, unde se instantiaza cele doua interfete si clasa MainController.
  1. **GUI 1 ( Input Data )**



* 1. **GUI 2 ( Simulation Stats )**



# Rezultate

* Se pornește aplicația, dând run proiectului. Se deschide prima interfață, unde clientul introduce datele de intrare (numărul de clienți , numărul de cozi, timpul total de simulare, timpul de intrare minim și maxim și timpul de deservire minim și maxim). Daca utilizatorul nu a introdus nimic intr-unul dintre text field-uri, atunci aplicatia il va avertiza si va astepta sa introduca datele lipsa. În această interfață este un buton care, la apăsare, închide interfața și pornește o altă interfață unde se observă datele de simulare(ce clienți se află în fiecare coadă la fiecare secundă). Dacă nu mai sunt clienți sau timpul de lucru al aplicației s-a terminat, se închide simularea. La final, datele simulării sunt salvate într-un fișier text, unde se poate analiza cum au fost împărțiți clienții pe cozi la fiecare moment de timp.

# Concluzii

* După terminarea acestei teme, pot spune că la final mi-a ieșit sa fac o simulare destul de bună a împărții clienților pe cozi, pentru a avea un timp de așteptare cât mai mic. De asemenea mi-am îmbunătățit cunoștințele în crearea de Graphical User Interface (GUI) și utilizare MVC (Model View Controller). Însă, efectuarea acestei teme m-a ajutat mult să înțeleg mai bine cum lucrează firele de lucru (threads) și sincronizarea acestora, cum ar fi utilizarea metodelor synchronized sau ar tipurilor AtomicInteger sau BlockingQueue. La final am folosit și log4j pentru prima dată pentru a afișa într-un fișier datele simulării. Ca dezvoltări ulterioare, as putea îmbunătăți interfața grafică, pentru a fi capabilă să arate datele pentru un număr mult mai mare de cozi.

# Bibliografie

1. https://m.youtube.com/watch?v=rbuR9Q\_55h4 (log4j)
2. https://dsrl.eu/courses/pt/materials/A2\_Support\_Presentation.pdf
3. https://springframework.guru/log4j-2-configuration-using-properties-file/