**Urmatoarele capitole sintetizeaza continutul documentatiei :**

**Capitolul 1.Obiectivul temei (principal)**

Obiectivul principal al lucrarii este intelgerea modului de lucru cu expresiile lambda si stream-uri , din punct de vedere al procesarii , implementarii si manipularii acestora in functie de necesitati (cerintele problemei) .Totodata se dobandesc cunostiinte in privinta stilului de a scrie cod , in sensul ca acesta se diminueaza cu utilizarea expresiilor lambda si a stream-urilor. Printre altele, se intelege modul de prelucrare a datelor in pipeline cu ajutorul stream-urilor.

**1.1.Obiectivele secundare sunt adresate in urmatoarele capitole :**

**Cap.2 : Analiza problemei ,modelare ,scenarii ,cazuri de utilizare**

In cadrul acestei lucrari nu se vor prezenta descrieri de use-case , diagram de clase sau de pachete intrucat aplicatia este construia dintr-o singura clasa care este testata intr-o alta (clasa main va contine rezolvarea cerintelor sub forma de expresii lambda si stream-uri). Aplicatia realizeaza niste statistici care raspund unor cerinte detaliate in capitolul 3. Tema proiectului e data de istoricul de activitati ale unei persoane care sunt inregistrate de un set de senzori.

**Cap.3: Implementarea propriu-zisa**

Se prezinta detalii legate de implementare , de deciziile luate in privinta solutiilor pentru cerintele lucrarii , decizii de structurare a codului si de afisare a rezultatelor. Abordarea problemei este subiectiva , acestea putand avea mai multe solutii. Implementarea se rezuma la aproximativ 100 de linii de cod exceptand liniile de afisare a rezultatelor , de unde se poate observa o prima utilitate a abordarii prin expresii lambda si utilizarea stream-urilor.

**Cap.4. : Concluzii**

In acest capitol , se discuta aspecte legate de concluzii si cunostiinte dobandite precum si realizarea unei comparatii cu posibile metode de rezolvare alternative si traditionale.

**Cap.5. : Bibliografie**

**Cap.2 : Analiza problemei ,modelare ,scenarii ,cazuri de utilizare**

Aplicatia este alcatuita din doua pachete : activityProcess si main. Pachetul activityProcess contine doar clasa MonitoredData . Aceasta contine 3 campuri :

private Date start\_time;   
 private Date end\_time;  
 private String activity;

In aceste campuri se vor salva date din fisierul Activities.txt , fisier care contine activitatile inregistrate de niste senzori , activitati cu timp de inceput si timp de sfarsit . La crearea unui obiect de clasa MonitoredData se citeste o linie din fisier si se salveaza datele corespunzatoare in campurile obiectului nou descris.

Pachetul main contine clasa MainClass al carei rol principal este asamblarea si implementarea diverselor solutii propuse cerintelor :

1. Crearea unor obiecte de tip MonitoredData din liniile fisierului citit (cu ajutorul stream- urilor) pentru manipulari ulterioare

2. Gasirea numarului de zile monitorizate

3. Crearea unui Map<String,Float> din perechi de forma <activitate, numar de aparitii> in care “activitate” inseamna eticheta(numele activitatii din fisier) si “numarul de aparitii” se refera la numarul de aparitii a activitatii respective pe intreaga perioada de monitorizare

4. Determinarea numarului de aparitii a fiecarei activitati pentru fiecare zi

5. Calculul duratei fiecarei activitati(fiecarei linii) din fisier

6. Pentru fiecare activitate durata ei dar pe intreaga durata de monitorizare

7.Filtrarea activitatilor care au 90% din inregistrari cu durata mai mica de 5 minute

Solutiile acestor cerinte se vor implementa exclusiv prin expresii lambda si stream-uri ,pastrandu-se totodata structura clasei MonitoredData , asa cum se va detalia in capitolele urmatoare (fiecare enunt de mai sus va fi explicat individual si se va prezenta strategia de implementare aleasa).

**Cap.3: Implementarea propriu-zisa**

Dupa cum am explicat si mai sus implementarea e execlusiv oritentata pe stream-uri si expresii lambda ceea ce pune in evidenta puterea acestor structuri , iar dimensiunea codului le pune intr-o lumina si mai buna , acesta rezumandu-se la aproximativ 100 de linii de cod (fara contorizarea liniilor de afisare a rezultatelor). Printre liniile de cod pentru afisare se afla si blocuri try-catch pentru ca rezultatele sunt scrise in fisiere pentru a imparti solutiile si pentru a nu incarca prea mult consola , vizualizarea si verificarea rezultatelor putandu-se observa mai usor din fisiere.

Explicatia implementarii aduse se va structura pe cerinte astfel:

**1. Crearea unor obiecte de tip MonitoredData din liniile fisierului citit (cu ajutorul stream-urilor) pentru manipulari ulterioare**

List<MonitoredData> monitoredData=null;  
 try {  
 Stream<String> lines=Files.*lines*(Paths.*get*("Activities.txt"));  
 monitoredData=lines  
 .map(line-> new MonitoredData(line.split("\\s\\s+")))  
 .collect(Collectors.*toList*());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 try {  
 fw=new FileWriter("monitoredData.txt");  
 } catch (IOException e) {  
 e.printStackTrace();  
 }

for(int i=0;i<monitoredData.size();i++)  
 {  
 fw.append(monitoredData.get(i)+"\n");  
 }  
 fw.close();

La prima linie se creeaza o lista de MonitoredData si se initializeaza cu null. Urmeaza un block de try-catch intrucat vom incerca obtinerea unei cai valide catre fisierul care contine datele inregistrate de senzori (activitatea, timpul de start,timpul de final) , din interiorul caruia se poate arunca o exceptie de tipul IOException in cazul in care acest fisier nu exita. In cazul nostru fisierul exista.

Metoda lines din Files returneaza un stream de String-uri pentru prelucrare a fisierului . Aceasta linie practic va parcurge ulterior pentru prelucrare toate liniile , cu ajutorul unui stream , in cazul nostru stream-ul de String-uri “lines”. Odata ce am obtinut acest stream putem prelucra printr-un pipeline operatiile pe aceste linii.

Ne intereseaza sa mapam fiecare linie la un obiect de tip MonitoredData. Pentru aceasta va trebuie sa impartim fiecare linie ( cuvantul “fiecare” e asigurat de modul de functionare a unui stream care va parcurge pana la urma fiecare element care i se da spre prelucrare) pe campuri . Aceasta impartire se realizeaza prin regexul "[\\s\\s](../../../../../..//s)+" care va imparti string-ul acolo unde exista cel putin 2 spatii.

Expresia lambda va returna un obiect care va primi ca argument un String[] exact ce returneaza metoda split din String. In final cu ajutorul metodei collect vom colecta fiecare obiect nou creat intr-o lista pe care o vom putea asigna la finalul pipeline-ului listei intializate cu null la inceput.

Explicatia pentru pasarea unui String[] in constructorul lui MonitoredData este ca acolo se va face parsarea datelor . Acest lucru se evidentia aici:

SimpleDateFormat format=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
 try {  
 this.start\_time = format.parse(data[0]);  
 this.end\_time = format.parse(data[1]);  
 } catch (ParseException e) {  
 e.printStackTrace();  
 }  
 this.activity = data[2];

Campurile start\_time si end\_time sunt de tipul Date deci e nevoie de parsarea String-ului intr-o data valida dupa formatul furnizat mai sus ( anul-luna-ziua ora:minutul:secunda) . Activitatea e un String deci e posibila o simpla asignare. Ceea ce nu s-a mentionat este ca ordinea de transmitere a String-urilor conteaza : primul element din tablou fiind timpul de inceput, al doilea fiind timpul de final si ultimul fiind activitatea.

**2. Gasirea numarului de zile monitorizate**

Stream stream=Stream

.*concat*(monitoredData.stream().map(start-> start.getStart\_time().getDate()),  
 monitoredData.stream().map(end->end.getEnd\_time().getDate()));  
 long nrOfDays=stream.distinct().count();

Numarul de zile monitorizate depine de timpul de inceput si sfarsit , insa in unele situatii o activitate incepe intr-o zi si se termina in alta . Solutia propusa este identificarea datilor , mai exact a zilelor lunii pentru timpul de inceput si sfarsit si colectarea lor intr-un singur stream . In final cu ajutorul metodelor distinct (elimina posibilitatea aparitiei a aceleasi zile in rezultat ) si count(), care este apelat in pipeline dupa eliminarea duplicatelor se va returna numarul de zile.

**3. Crearea unui Map<String,Float> din perechi de forma <activitate, numar de aparitii> in care “activitate” inseamna eticheta(numele activitatii din fisier) si “numarul de aparitii” se refera la numarul de aparitii a activitatii respective pe intreaga perioada de monitorizare**

Solutia propusa este urmatoarea:

Map<String, Long> occurrences = monitoredData.stream()  
 .collect(Collectors.groupingBy(a -> a.getActivity(),  
 Collectors.counting()));

Metoda groupingBy() de mai sus din Collectors va returna un obiect de tip Map in care cheia e un String si anume eticheta activitatii si valoarea va fi reprezentata de un Long care reprezinta numarul de aparitii a activitatii respective pe intreaga perioada de monitorizare. Acest numar e procesat in felul urmator. Atunci cand se gasesc doua activitati cu acelasi nume deci trebuies mapate la aceeasi cheie 2 valori diferite de fapt se va aduna la aceeasi valoare +1 la fiecare activitate cu aceeasi cheie gasita prin metoda counting din Collectors.

**4. Determinarea numarului de aparitii a fiecarei activitati pentru fiecare zi**

Map<String,Map<String,Long>> s1=monitoredData.stream()  
 .collect(Collectors.groupingBy(a->a.getStartIdentifier(),  
 Collectors.groupingBy(a- > a.getActivity(),Collectors.counting())));  
 //count end days  
 Map<String,Map<String,Long>> s2=monitoredData.stream()  
 .collect(Collectors.groupingBy(a->a.getEndIdentifier(),  
 Collectors.groupingBy(a-> a.getActivity(),Collectors.counting())));  
 //merge start and end days  
 s2.forEach((k,v)->s1.merge(k,v,  
 (m1,m2)->{  
 Map<String,Long> m=new HashMap<>(m1);  
 for(Map.Entry<String,Long> entry:m2.entrySet()) {  
 String key=entry.getKey();  
 if(m.containsKey(key) && m2.get(key)>m.get(key))  
 m.put(key,m2.get(key));  
 if(!m.containsKey(key))  
 {  
 m.put(key,m2.get(key));  
 }  
 }  
 return m;  
 }  
 ));

Acest caz e mai special in sensul in care solutia este partitionata in 2 din cauza urmatorului motiv: o activitate poate incepe intr-o zi si sa se termine in alta astfel va trebui sa se tina cont atunci cand se numara aparitiile / zile ,si de timpul de final de exemplu , urmatoarea linie din fisier :2011-11-28 20:21:15 2011-11-29 02:06:00 Spare\_Time/TV. Incepe in 28 si se termina in 29 . In cazul in care activitatea incepe intr-o zi si se termina in aceeasi zi se va numara o singura data pentru ziua respectiva. Acestea se traduc in urmatoarea explicatie relativa la cod:

Cream 2 stream-uri , unul pentru momentele de inceput si unul pentru cele de final care numara similar ca la punctul 3 activitatile pentru fiecare moment. Urmeaza reuniunea celor 2 stream-uri si maparea de forma Map<String,Map<String,Long>> care va asocia fiecarei zile (ca si String) activitatile (ca si String) si numarul de aparatii in acele zile (ca si long) . Metoda forEach din final este poate cea mai importanta pentru ca functioneaza in felul urmator: Se parcurge fiecare pereche de forma <String,Map> din reuniunea de stream-uri rezultata din metoda “merge()” , dupa care cu ajutorul expresiilor lambda se cauta perechile de forma <String,Long> iar acolo unde exista deja o mapare , mai exact acolo unde am contorizat numarul de aparitii pentru o activitate vom avea un conflict care se solutioneaza in felul urmator: daca numarul de activitati inregistrate de o un moment (de inceput de ex) este mai mare decat celalalt moment(de sfarsit) atunci vom salva(mapa) pentru acea activitate doar numarul de aparitii pentru inceput.

**5. Calculul duratei fiecarei activitati(fiecarei linii) din fisier**

Map<MonitoredData, Long> durations = monitoredData.stream()  
 .collect(Collectors.toMap(a->a,(a->a.getEnd\_time().getTime()/1000- a.getStart\_time().getTime()/1000)));

Aici are loc o simpla mapare cu ajutorul expresiilor lambda si a metodei toMap. Metoda getTime obtine timpul in milisecunde si de aceea se va imparti la 1000.Durata se calculeaza ca si diferenta dintre timpul de final si timpul de inceput (in secunde) . Atribuirea a->a are sens pentru ca stream-ul parcurge elemente de tip MonitoredData si salveaza ca si chei obiecte de acelasi tip.

**6. Pentru fiecare activitate durata ei dar pe intreaga durata de monitorizare**

Map<String,Long> durationsGlobal=durations.entrySet().stream()  
 .collect(Collectors.groupingBy(entry-> entry.getKey().getActivity(),Collectors.summingLong(entry- > entry.getValue())));

Pentru aceasta cerinta de folosim de solutia 5 unde vom parcurge fiecare pereche de forma <MonitoredData, Long> si vom stabili daca exsita chei egale , in caz afirmativ valorile se vor insuma (cu metoda summingLong din Collectors) si mapa pe prima cheie ,cealalta fiind stearsa.

**7.Filtrarea activitatilor care au 90% din inregistrari cu durata mai mica de 5 minute**

Map<String,Long> durationLess5Min= durations.entrySet().stream()  
 .filter(entry->entry.getValue()< 300).collect(Collectors.groupingBy(a -> a.getKey().getActivity(),  
 Collectors.counting()));  
Map<String,Long> \_90percent=durationLess5Min.entrySet().stream()  
 .filter(entry->entry.getValue() >= 90\*occurrences.get(entry.getKey())/100).collect(Collectors.toMap(entry->entry.getKey(),entry->entry.getValue()\*100/occurrences.get(entry.getKey())));

Aici ne vom folosi tot de rezultatul de la 5 pe care vom crea un stream care sa filtreze actvitatile cu durata mai mica decat 5 min si sa le grupeze dupa numele activitatii cu groupingBy(). Va trebui sa selectam acum doar acele activitati care au 90% din inregistrari cu durate < 5min , deci vom aplica un stream pe rezultatul intermediar produs anterior si de rezultatul produs la 3. Primul pas a fost numararea activitatile care au inregistrari cu durate < 5 min de unde rezulta un numar x. Al doilea pas este sa obtinem numarul total de aparitii a unei activitati cu 5 deci pentru o activitate rezulta de aici un numar y. Predicatul filtrarii consta in verificarea x>=90/100\*y si maparea procentului la activitatile care verifica cu adevarat predicatul prin metoda toMap discutata ca si mai sus.

**Cap.4. : Concluzii**

Rezolvarea cerintelor a adus urmatoarele cunostiinte dobandite : intelegerea modului de functionare a expresiilor lambda si a stream-urilor , modului de procesare si de abordare a unei solutii prin aceste expresii precum si formarea unei gandiri in directia scrisului de cod cu aceste expresii, acesta fiind foarte putin in comparatie cu metodele de rezolvare traditionale care din punct de vedere al dimensiunii codului sunt mai mari. Datorita acestui fapt creste lizibilitatea codului . Totodata cu o anumita subtilitate a aparut si pipeline-ul de prelucrare al stream-ului care prin eficientizari ar putea sa execute operatii intermediare(neterminale) in paralel.

**Cap.5. : Bibliografie**

**- FlatMap:**

**https://stackoverflow.com/questions/26684562/whats-the-difference-between-map-and-flatmap-methods-in-java-8**