**DOCUMENTATIE TEMA NR.5**

PROCESSING SENSOR DATA OF DAILY LIVING ACTIVITIES

Student: Bumbu Iulia-Diana

An: II Semestrul: 2

Grupa: 30228

CUPRINS:

1. Obiectivul temei
2. Analiza problemei
3. Proiectare
4. Implementare
5. Rezultate
6. Concluzii
7. Bibliografie
8. Obiectivul temei:

1.1.Obiectiv principal:

Principalul scop al proiectului este de a realiza o aplicatie care sa analizeze comportamentul unei persoane ale carei activitati au fost inregistrate de un set de senzori instalati in casa acesteia. Istoricul activitatilor este extras, analizat si prelucrat conform unor task-uri ulterior prezentate.

1.2.Obiective secundare:

Pentru o dezvoltare optima a unei aplicatii de acest tip trebuie realizati anumiti pasi pentru indeplinirea obiectivului principal.

Principalele **obiective secundare** sunt:

* Realizarea USE CASE-urilor si a scenariilor:

O diagrama a cazurilor de utilizare ofera o descriere generala a modului in care va fi utilizat sistemul, furnizand o privire de ansamblu asupra functionalitatilor care doresc a fi oferite de sistem. (capitolul 2)

* Identificarea structurilor de date:

Structura de date este o metoda sistematica de stocare a informatiilor intr-un calculator astfel incat sa poata fi folosite in mod eficient. (capitolul 3)

* Impartirea claselor:

Pentru a gestiona mult mai usor logica aplicatiei si logica realizarii functionalitatilor dorite proiectul se imparte in mai pachete (capitolul 3)

* Dezvoltarea algoritmilor:

Se va determina strategia necesara pentru a gestiona activitatile persoanei monitorizate(capitolul 3)

* Implementarea solutiei:

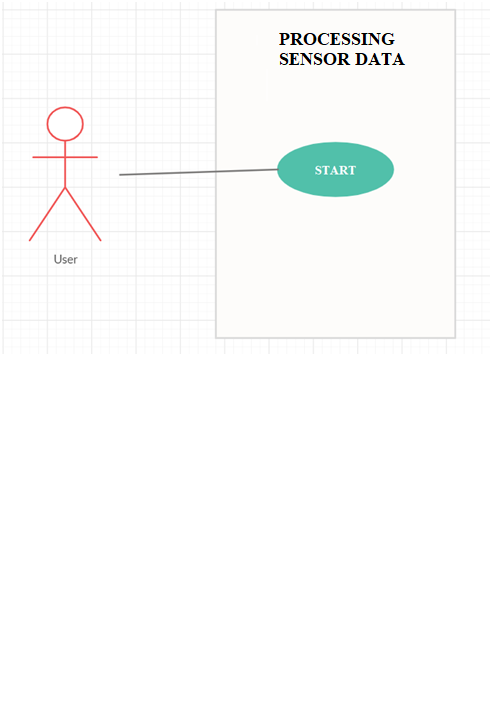
Este reprezentata de descrierea in cod a tuturor claselor anterior explicate, a metodelor necesare gestionarii activitatilor precum si procesarea de fisiere pentru a testa functionalitatea programului creat. (capitolul 4)

* Gestionarea rezultatelor:

Datele de intrare si de iesire se vor procesa folosind streamuri si expresii lambda (capitolul 5)

2.Analiza problemei:

2.1.USE CASE Diagram:



2.2. Scenarii:

O utilizare corecta a programului presupune prelucrarea unui fisier de intrare valid pentru a procesa datele existente. Input valid presupune ca pe fiecare linie exista data de incepere a activitatii, data de incheiere a activitatii, respectiv denumirea acesteia. Data de inceput/sfarsit are formatul "yyyy-MM-dd HH:mm:ss" iar activitatile efectuate au etichete de genul Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming. Activitatile persoanei sunt monitorizate pe perioada mai multor zile, dupa procesare pe aceste date performandu-se urmatoarele cerinte:

TASK\_1 : -fiecare activitate este pastrata sub forma <start\_time, end\_time, activity\_label>, definindu-se astel o clasa MonitoredData care contine aceste campuri. Datele sunt citite din fisierul de intrare si folosind streamuri se genereaza o lista cu obiectele de tipul MonitoredData.

TASK\_2: - se numara zilele distincte care apar pe parcursul monitorizarii

TASK\_3: - se numara de cate ori un tip de activitate a aparut pe durata intregii monitorizari, returnandu-se o structura de tip Map<String, Long>

TASK\_4: - se numara de cate ori o activitate a aparut intr-o zi, pentru fiecare zi a perioadei de monitorizare, returnandu-se o structura de tip Map<String, Map<String, Long >>

TASK\_5: - pentru fiecare activitate se calculeaza intreaga durata pe perioada de monitorizare, returnandu-se o structura de tip Map<String, Duration>

TASK\_6: - se filtreaza activitatile care au mai mult de 90% din inregistrari cu durata de mai putin de 5 minute, returnandu-se o lista de String-uri care contine numele activitatilor distincte

De asemenea, rezultatul fiecarui task este scris intr-un fisier txt separat, numit sub forma task\_number.txt.

* Proiectare:

3.1.Structuri de date:

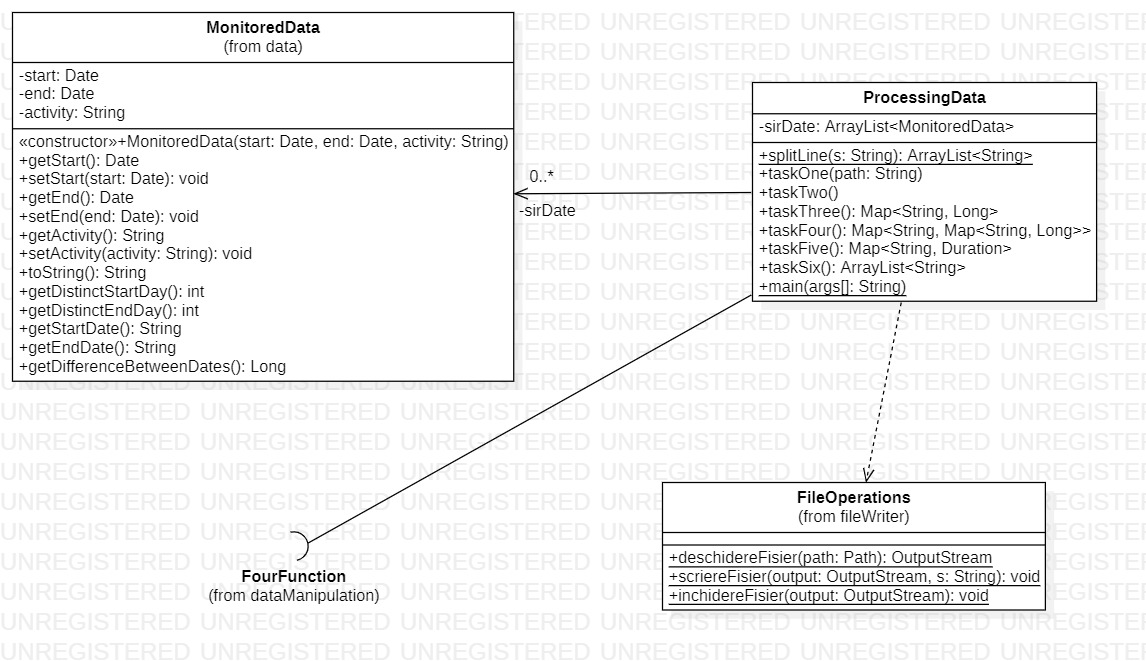
Deoarece multiple date monitoriate trebuie procesate in mai multe metode adesea este folosita ca structura de date ArrayList deoarece faciliteaza lucrul cu vectori prin metodele preimplementate. De exemplu:

ArrayList<MonitoredData> sirDate = **new** ArrayList<MonitoredData>();

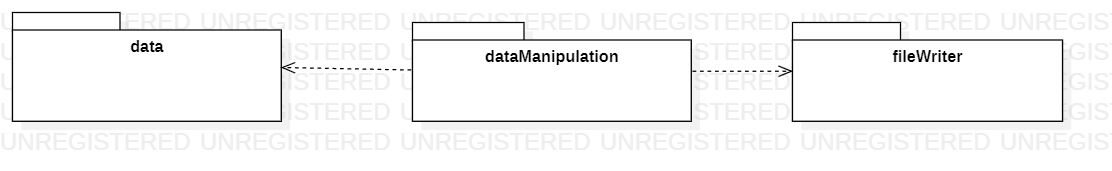
De asemenea se individualizeaza folosirea unor structuri de tipul Map pentru a stoca informatiile legate de activitatile procesate la anumite taskuri, printre care: Map<String, Long> map, Map<String, Map<String, Long>> map, Map<String, Duration> newMap . Acestea au fost prelucrate cu ajutorul streamurilor.

3.2.Diagrama de clase

Diagrama de clase este folosita in modelarea orientata obiect pentru a descrie structura statica a sistemului si a modului in care este el structurat. De asemenea, aceasta ofera o notatie greafica pentru reprezentarea claselor (entitati care au caracteristici comune) si a relatiilor (legaturi dintre doua sau mai multe clase) dintre ele.



3.3.Diagrama de pachete:



UML furnizeaza mijloace de grupare a elementelor din cadrul diagramelor numite pachete. Pachetele constituie baza pentru controlul configuratiei si stocare, contribuind la reutilizarea codului.

3.4.Algoritmi:

In cazul acestui proiect nu au fost utilizati algoritmi preimplementati insa au fost utilizate expresii lambda si interfete functionale si predicate cu implementare inline care au usurat logica codului. Unele dintre expresiile folosite sunt:

s -> s.replaceAll("\\s{2,}", " ").trim()

FourFunction<Date, Date, String, MonitoredData> funct4 = (start, end, name)-> **new** MonitoredData(start, end, name);

BiPredicate<Long, Long> predicate1 = (x, y) -> y > 0.9 \* x;

* Implementare:

4.1.Descrierea claselor din pachetul Data:

* Clasa MonitoredData:

Clasa are rolul de a retine datele definitorii pentru o activitate monitorizata deoarece fiecare atribut este atribuit unui camp:

**private** Date start;

**private** Date end;

**private** String activity;

Se remarca existenta contructorului specific cu campuri,a metodelor get si set precum si a metodelor:

**public** String toString() –utilizata pentru a afisa detaliile datelor monitorizate, specificand care e data de inceput, data de sfarsit si denumirea activitatii

**public** String getStartDate() -responsabila cu manipularea datei de inceput pentru a o returna sub forma de String. Astfel, din formatul implicit al tipului Date am extras ziua, luna si anul folosindu-ne de clasa Calendar si le-am concatenat pentru a obtine formatul dorit apeland la constantele clasei DAY\_OF\_MONTH, MONTH, YEAR.

**public** String getEndDate() -responsabila cu manipularea datei de sfarsit pentru a o returna sub forma de String. Astfel, din formatul implicit al tipului Date am extras ziua, luna si anul folosindu-ne de clasa Calendar si le-am concatenat pentru a obtine formatul dorit apeland la constantele clasei DAY\_OF\_MONTH, MONTH, YEAR.

**public** **int** getDistinctStartDay()–utilizata pentru a obtine o codificare unica pentru ziua de inceput. Daca am folosi doar ideea de la metoda anterioara cu folosirea clasei Calendar respectiv a constantei DAY\_OF\_MONTH pentru a gasi ziua din data nu s-ar face diferenta dintre ziua x luna y si ziua x luna z, asadar codificam ziua sub forma nr = luna \* 31 + zi; pentru a asigura unicitatea numarului returnat.

**public** **int** getDistinctEndDay()–utilizata pentru a obtine o codificare unica pentru ziua de sfarsit. Daca am folosi doar ideea de la metoda anterioara cu folosirea clasei Calendar respectiv a constantei DAY\_OF\_MONTH pentru a gasi ziua din data nu s-ar face diferenta dintre ziua x luna y si ziua x luna z, asadar codificam ziua sub forma nr = luna \* 31 + zi; pentru a asigura unicitatea numarului returnat.

**public** Long getDifferenceBetweenDates()–calculeaza si returneaza durata in milisecunde dintre data de sfarsit si data de inceput

4.2. Descrierea claselor din pachetul DataManipulation:

* Interfata FourFunction:

Reprezinta o interfata functionala care poate primi trei parametri si contine antetul metodei:

**public** Four apply(One one, Two two, Three three);

* Clasa ProcessingData:

Clasa contine ca atribut un sir de obiecte de tip MonitoredData corespunzator datelor din fisierul de tip txt care trebuie prelucrat:

**private** ArrayList<MonitoredData> sirDate = **new** ArrayList<MonitoredData>();

Principala functionalitate a clasei este de a implementa taskurile specificate in sectiunea 2.2 valorificand operatiile cu streamuri si expresii lambda. In acest sens, se individualizeaza metodele:

**public** **static** ArrayList<String> splitLine(String str) –stringul trimis ca parametru este separat dupa doua spatii “ “ prin apelul finctiei split iar noile subsiruri obtinute sunt mapate si colectate intr-un ArrayList<String> returnat ca parametru folosind functionalitatea Stream.of

**public** **void** taskOne(String path)–metoda debuteaza prin colectara fiecarei linii din fisierul intr-un ArrayList<String> numit list folosind streamurile. Avand in vedere ca intr-o linie din fisier intre data de inceput/ sfarsit sau activitate exista un numar variabil de spatii pentru prelucrarea datelor acestea au fost eliminate formand un nou ArrayList<String> replacedList in care am mapat liniile lasand maxim doua spatii intre caractere apicand expresia lambda s -> s.replaceAll("\\s{2,}", " ").trim().Tinand cont de validitatea datelor de intrare s-a creat formatul de data "yyyy-MM-dd HH:mm:ss". Pentru a valorifica cunostintele nou dobandite s-a creat

FourFunction<Date, Date, String, MonitoredData> funct4 = (start, end, name)-> **new** MonitoredData(start, end, name); cu rol de instantiere a unui nou obiect de tip MonitoredData. Pentru fiecare String din noua lista obtinuta corespunzator liniei cu spatii eliminate s-a apelat functia anterior prezentata splitLine pentru a obtine componentele data de inceput, data de sfarsit si denumire activitate, astfel putand sa se instantieze si adauge in sirul de date un nou obiect prin apelul lui funct4.apply. Pentru a verifica corectitudinea manipularii datelor acestea au fost scrise intr-un nou fisier dupa instantierea tuturor obiectelor.

**public** **static** <T> Predicate<T> distinctByKey(Function<? **super** T, Object> keyExtractor) –predicatul determina daca cheia trimisa este unica (preluata din sursa [2] prezentata la bibliografie)

**public** **void** taskTwo(String path)–pentru a determina numarul de zile distincte se pastreaza in variabila count numarul de zile cu zi de start distincta si in variabila count2 numarul de zile cu zi de end distincta. Rezultatul este cea mai mare dintre cele doua valori returnate. Pentru a se realiza corect numaratorile pentru streamul sirului de date se filtreaza dupa cheia unica folosind predicatul anterior prezentat dupa care se utilizeaza functionalitatea count care returneaza un long echivalent numarului de aparitii distincte obtinute pentru cheia trimisa. Dupa obtinerea rezultatului se creeaza un nou fisier prin apelul metodei deschidereFisier, dupa care se scrie in el numarul obtinut si se inchide fisierul.

**public** Map<String, Long> taskThree()–pentru a contoriza de cate ori apare o activitate se returneaza un map in care cheia este reprezentata de numele activitatii iar valoarea contorizarea aparitiilor. Pentru a returna aceasta structura s-a folosit Collectors.groupingBy pt a grupa dupa activitate si Collectors.counting pt a numara aparitiile (deoarece aceasta metoda returneaza un long s-a schimbat tipul valorii mapului din cerinta). Dupa ce s-a obtinut map-ul aceasta este parcurs si fiecare componenta este scrisa intr-un nou fisier.

**public** Map<String, Map<String, Long>> taskFour()–pentru a contoriza de cate ori fiecare activitate a aparut de-a lungul unei zile se creeaza doua noi siruri de MonitoredData sirS-contine toate datele de inceput si SirE-contine doar datele de final pentru activitatile care incep intr-o zi si se termina in alta prin filtrarea streamului, dupa care se colecteaza intr-un nou map toate obiectele obtinute prin concatenare streamurilor celor doua siruri folosindu-ma de metoda din Collectors groupingBy. Cheia mapului returnat este data de inceput procesata ca string prin apelul metodei getStartDate, iar valoarea un nou map cu cheie numele activitatii si valoare numarul aparitiilor sale. Rezultatul obtinut se scrie intr-un nou fisier

**public** Map<String, Duration> taskFive()–pentru a obtine durata activitatilor pe intreaga perioada de monitorizare initial se colecteaza intr-un Map<String, Long> datele grupate dupa numele activitatii ca si cheie, iar ca valoare suma duratelor. Durata unei singure aparitii a activitatii a fost calculata in milisecunde apeland metoda getDifferenceBetweenDates, iar insumarea tuturor duratelor obtinute s-a facut apeland Collectors.summingLong. Pentru a returna rezultatul sub forma Map<String, Duration> din streamul map-ului anterior obtinut s-au colectat datele cu aceleasi chei si valoare cu nr de milisecunde transformate in Duration prin apelul metodei Duration.ofMillis. La scrierea rezultatului in fisier fiecare element al mapului returnat a fost prelucrat extragand din Duration orele, minutele si secundele pentru o afisare mai frumoasa. S-a ales tipul Duration in locul lui LocalTime deoarece exista activitati monitorizate a caror durata totala depaseste 24 de ore.

**public** ArrayList<String> taskSix()–pentru a filtra datele care au mai mult de 90% din inregistrari cu durata de mai putin de 5 minute s-a colectat initial intr-un Map<String, Long> map toate activitatile cu numarul aparitiei lor, iar in Map<String, Long> mapLess5 aceleasi activitati insa filtrate dupa conditia p -> p.getDifferenceBetweenDates()/60000 < 5 pentru a ne asigura ca duratele activitatilor obtinute nu depasesc 5 minute. S-a construit si bipredicatul

BiPredicate<Long, Long> predicate1 = (x, y) -> y > 0.9 \* x; care ne ajuta sa testam conditia cu mai mult de 90%. Avand toate aceste date obtinute, din streamul mapului mapLess5 s-au colectat intr-un ArrayList<String> denumirile activitatilor obtinute dupa filtrarea in care s-a aplicat bipredicatul predicate1, asigurandu-ne astfel ca cel putin 90% din activitatile date au durata de maxim 5 minute. Lista obtinuta este returnata ca rezultat, iar elementele sale scrise intr-un nou fisier.

**public** **static** **void** main(String args[])–in aceasta metoda se apeleaza toate taskurile anterior prezentate si se specifica calea fisierului de intrare care trebuie prelucrat, acesta fiind primit ca parametru.

* 1. Descrierea claselor din pachetul FileWriter:
* Clasa FileOperations:

Clasa responsabila cu efectuarea operatiilor asupra fisierului de iesire, continand metodele:

**public** **static** OutputStream deschidereFisier(Path path) – creeaza fisierul cu calea trimisa ca parametru

**public** **static** **void** scriereFisier(OutputStream output,String s) – scrie sirul s in fisierul cu OutputStream trimis ca parametru. Inainte de a se scrie, se realizeaza conversia caracterelor sirului in biti pentru a putea folosi metoda preimplementata write

**public** **static** **void** inchidereFisier(OutputStream output) – se inchide fisierul cu OutputStream trimis ca parametru prin apelul functiei close

1. Rezultate:

Pentru realizarea verificarii rezultatelor programul este configurat sa ruleze din terminal si sa proceseze un fisier cu date primit ca parametru. In acest sens s-a configurat un fisier .jar. La inchiderea aplicatiei noile rezultate obtinute dupa procesarea datelor sunt scrise in fisiere txt separate. Corectitudinea rezultatelor se poate verifica prin analizarea fisierelor generate.

1. Concluzii:

In concluzie, aceasta tema mi-a oferit oportunitatea de a invata cum sa folosesc streamurile si de a utiliza expresii lambda, interfete functionale si predicate. Astfel, am sedimentat o parte din conceptele de baza ale lucrului cu acestea si am aprofundat alte cunostinte in legatura cu POO . Proiectul poate avea alte dezvoltari ulterioare prin adaugarea de noi functionalitati, folosirea rezultatelor furnizate de un task in alte scopuri sau prin creerea unei interfete grafice pentru a manipula mult mai interactiv datele de intrare.

7. Bibliografie:

1. <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_5/Assignment_5.pdf>

2. <https://www.baeldung.com/java-streams-distinct-by>

3. <https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>