DOCUMENTATIE TEMA 5

Analysing the behaviour of a person

ROTARU IONUT

Seria B, gr. 302210

**CUPRINS**

1. Obiectivul temei. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3
2. Cazuri de utilizare . . . . . .. 4
3. Clasele si metodele lor.. . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
4. Rezultate si concluzii 8

**1) Obiectivul temei**

Cerinta temei este crearea unui sistem de monitorizare prin senzori pentru a analiza comportamentul unei persone. Activitati precum dormitul, spalatul sau uitatul la televizor se realizeaza in anumite interval de timp si au loc in momente diferite ale unei zile. Scopul este acela de a afla anumite caracteristici ale comportamentului unei persoane.

Cele 6 task-uri ale temei sunt:

a) Citirea datelor dintr-un fisier text si transformarea lor in obiecte ale clasei MonitoredData: Fiecare linie din fisier contine, in ordine, momentul zilei cand o activitate este inceputa si terminate si numele acelei activitati. Fisierul contine mai multe zile,iar o zi poate avea mai multe activitati. Fiecare linie din fisier trebuie citita si separate corespunzator in cele 3 parti. Pentru verificarea corectitudinii, se creaza un fisier de iesire cu aceleasi date.

b) Aflarea numarului de zile distincte care apar in perioada de monitorizare a activitatilor persoanei.

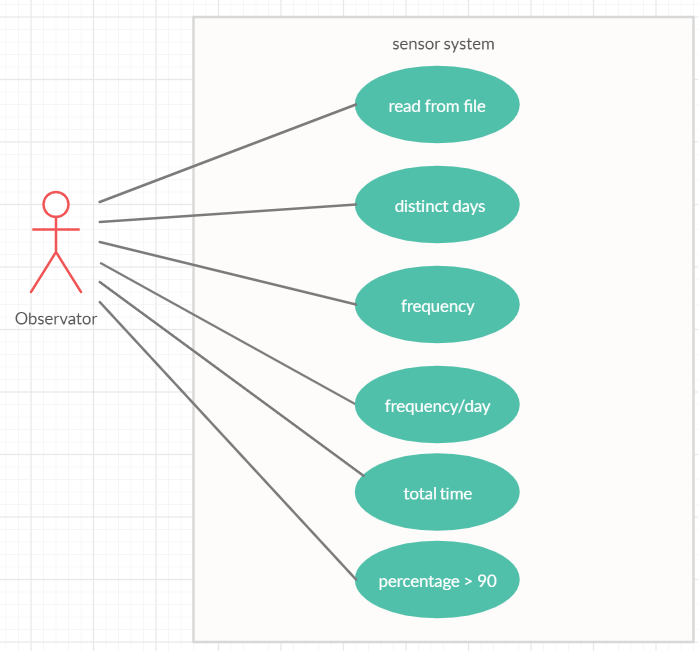
c) Calcularea frecventei de aparitie a fiecarei activitati pe parcursul monitorizarii.

d) Calculare frecventei de aparitie a fiecarei activitati in fiecare zi din perioada de monitorizare.

e) Calculul timpului total petrecut pentru realizarea fiecarei activitati.

f) Gasirea activitatilor care au loc in proportie de 90% din aparitii, sub 5 minute.

Ideea principal a temei este folosirea expresiilor lambda si a streamurilor in realizarea cerintelor.



**Diagrama use-case**

Use case: Pornire simulare:

• Primary Actor: Observator

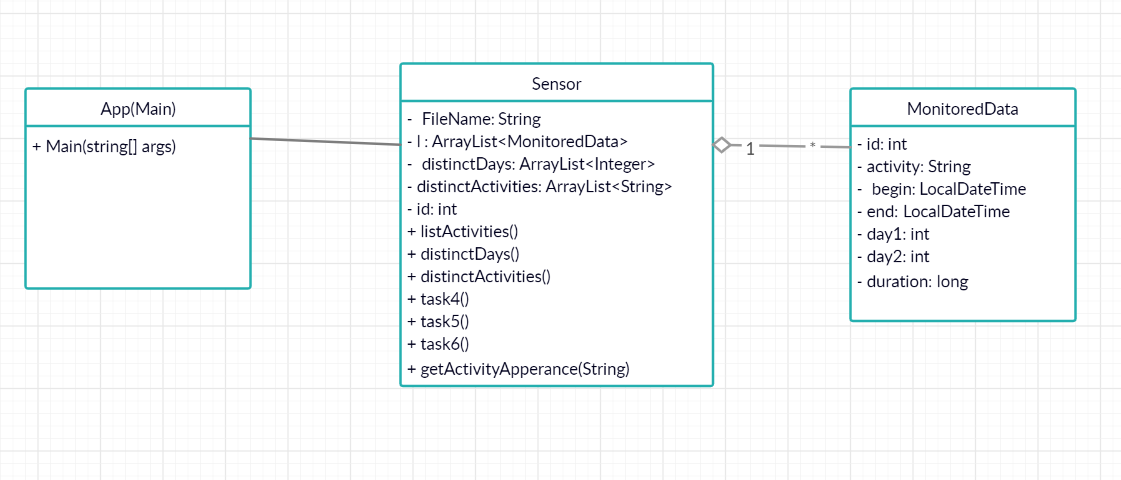
• Main Success Scenario:

1. Programul asteapta sa fie pornit;
2. Utilizatorul porneste aplicatia;
3. Task-urile sunt rezolvate pe rand, afisand un mesaj de validare;
4. Programul se opreste;

**Clase si pachete**

Structura proiectului este cat se poate de simpla. Exista un singur pachet si si trei clase: Main,

MonitoredData si Sensor.



**Clasa MonitoredData:**

Aceasta clasa este create pentru a reprezenta o activitate a persoanei intr-un obiect. Variabilele de clasa stocheaza informatiile necesare: activity, begin si end. Cele 3 campuri detin informatiile esentiale in monitorizare, dar am adaugat si altele noi, care vor ajuta in diferite locuri:

* Id: este folositpentru a da o ordine zilelor, asa cum sunt citite din fisier. Variabila este folosita la cerinta 4 tocmai in acest scop;
* Day1 si day2: reprezinta zilele cand o activitate incepe, respective se termina. Cel mai frecvent activitatea se desfasoara in aceiasi zi ( day1 = day 2 ), dar exista si cateva cazuri opuse.
* Duration: variabila de tip long care salveaza durata activitatii: end – begin;

Metodele clasei sunt urmatoarele:

* Constructorul: primeste id-ul (int) si linia citita din fisierul text. Stringul este impartit in cele 3 componente prin “String[] words = s.split(" ");” . Apoi, words[0] si words[1], adica begin si end, sunt impartite asemanator, pentru a separa ziua activitatii: String[] days = words[0].split("-"); . Day1 si day2 se calculeaza dupa formula x = 31\*m + d: day1 = Integer.parseInt(days[1])\*31 + Integer.parseInt(days[2].substring(0, 2));
* genBegin() si genEnd() sunt metodele care transforma stringurile in tipul LocalDateTime. Din string se selecteaza campurile reprezentand anul, luna, ziua, ora, minutul si secundele, apoi variabila begin va primi un obiect LocalDateTime creat cu detaliile de mai sus.
* Restul metodelor sunt de tip get si ajuta la obtinere datelor sau verificarea corectitudinii.

**Clasa Sensor:**

Clasa Sensor reprezinta clasa principal a proiectului. Aici au loc operatiile pentru rezolvarea celor 6 task-uri. Variabilele clasei sunt urmatoarele:

* filename: un String ce reprezinta numele fisierului text in care sunt stocate informatiile despre persoana monitorizata ( “ Activities.txt ” );
* l: ArrayList unde se vor salva toate liniile din fisier sub forma unor obiecte de tip MonitoredData;
* distinctDays: un ArrayList de intregi unde se salveaza fiecare day1 si day2 din obiectele MonitoredData. Initial vor fi si zile care apar de mai multe ori, dar in taskul 2 se aplica o procedura care va selecta numarul zilelor distincte;
* distinctActivities: un ArrayList de Stringuri care reprezinta numele activitatilor. Initial se introduce toate activitatile, chiar daca unele se repeta, dar in taskul 3 se vor folosi procedee pentru rezolvarea cerintei;
* id: intregul care memoreaza a cata linie din fisier este asociata unei activitati (1,2,…). La crearea unui nou obiect MonitoredData, id se incrementeaza cu o unitate;

Metodele clasei Sensor sunt:

* listActivities(): metoda reprezinta task-ul 1 si are ca scop citirea datelor din fisierul de intrare, crearea unui obiect MonitoredData pentru fiecare linie din fisier si scrierea liniei intr-un fisier de iesire, cu scopul de a verifica corectitudinea metodei. Pentru partea de citire am folosit un stream: Stream<String> stream = Files.lines(Paths.get(fileName)), unde filename este variabila de clasa. Astfel, in stream vom avea toate liniile din fisier. Pentru a crea obiectele MonitoredData, ma folosesc de “stream.forEachOrdered”: fiecare linie din stream va fi trimisa la argument (String) catre constructorul clasei MonitoredData. Obiectul creat este adaugat si in lista l, iar Stringul este scris si in fisierul de iesire. Cand programul isi termina executia, fisierul de iesire este creat, iar activitatile sunt salvate in Arraylist-ul corespunzator;
* distinctDays(): metoda reprezinta task-ul 2, avand rolul de a numara zilele distincte care se regasesc in fisierul de intrare. Primul pas este sa adaugam toate zilele in ArrayListul distinctDays. Acest lucru se face printru-un for, unde sunt adaugate atat day1, cat si day2 ale obiectelor din lista l. Pentru a vedea cate din aceste zile sunt diferite, streamul este cea mai rapida metoda: count = distinctDays.stream().distinct().count(); Astfel, in variabila count de tip long vom avea rezultatul dorit, pe care il si scriu in fisierul de iesire corespunzator task-ului 2; “.distinct()” si ”.count()” sunt metodele care rezolva singure cele 2 probleme in aflarea numarului de zile distincte;
* distinctActivities(): task-ul 3 presupune calcularea frecventei de aparitie a fiecarei activitati. Pentru asta, am salvat mai intai in lista distinctActivities numele tuturor activitatilor din lista l ( am apelat metoda getActivity() pentru a obtine doar numele), la fel ca la task-ul 2. Pentru calculul efectiv, folosesc functiile stream-ului :

Map<String, Integer> counts = distinctActivities.parallelStream().collect(Collectors.toConcurrentMap(w -> w, w -> 1, Integer::sum));

Fiecare activitate trebuie salvata impreuna cu frecventa sa de aparitie, deci trebuie folosit map<String,Integer>. Ceea ce se salveaza in fiecare element din map se calculeaza usor cu functia “.collect”: numele activitatii (Stringul din map) reprezinta chiar Stringul din lista distinctActivities, deci este doar copiat. Pentru a contoriza aparitia activitatii, am folosit 1,Integer::sum, care se salveaza in partea “value” a din map. Astfel, folosind streamul, am rezolvat cerinta task-ului si pot printa in fisierul de intrare datele obtinute;

* task4(): trebuie calculate de cate ori apare fiecare activitate in fiecare zi, iar rezultatul va fi de forma Map<Integer,Map<String,Long>>. Continutul din map este calculate de stream prin:

Map<Integer,Map<String,Long>> a = l.parallelStream().collect(Collectors.groupingBy(

t -> {

return t.getDay1();

},

Collectors.groupingBy(MonitoredData::getActivity,Collectors.counting())));

Prima folosire a functiei “Collectors.groupingBy” asigura informatia din map de tip Integer. Gruparea dupa day1 din ArrayListul de MonitoredData asigura unicitatea (desi day1 este un numar mare, conform formulei month\*31 + day, la scrierea in fisier am folosit id-ul, de la 1 la 14 ).Partea de “value” din map, reprezentata tot de un map, este calculate in a doua utilizare a “collectors.groupingBy”: se obtine fiecare nume de activitate din lista l (String) si se calculeaza frecventa de aparitie (long). Astfel, prin combinarea celor 2 metode, se obtine frecventa de aparifie a fiecarei activitati in fiecare zi. Ultima parte a metodei se ocupa cu buna afisare in fisierul de iesire (detaliile pentru o zi sa fie afisate pe cate o linie) .

* task5(): in aceasta metoda trebuie calculate timpul total petrecut pentru fiecare activitate inparte, adica adunarea tuturor timpilor. Informatiaprivitoare la durata unei activitati este calculate in clasa MonitoredData, folosind metodeleclaseiLocalDateTime: Duration.between ( begin, end ).getSeconds() .

Rezultatele acestui task sunt salvate intr-o structura map< String, long >, formata din numele activitatii si durata acesteia. Valorile celor doua campuri sunt calculate prin folosirea stream-ului urmator:

l.parallelStream().collect(Collectors.groupingBy(t ->t.getActivity(),

Collectors.reducing(0L,t -> t.getDuration(),

(x,y) -> x+y)));

Numele activitatii se obtine usor cu ajutorul metodei getActivity. Pentru calculul duratei totale am folosit “Collectors.reducing”: 0L reprezinta o valoare initiala cu care este incarcat strimul la inceput, apoi calculul se face prin obtinerea duratei activitatii intr-o anumita zi ( t.getDuration() ) si adunarea sa cu durata totala a celorlalte zile in care apare activitatea ( (x,y) -> x+y ). Rezultatul se salveaza in map si poate fiscris in fisierul de iesire.

-task6(): pentru gasirea activitatilor care au in proportie de macar 90% durata mai mica de 5 minute la fiecare aparitie se calculeaza in 2 etape:

Mai intai numar pentru fiecare activitate de cate ori apare cu o durata de maxim5 minute, apoi le verific sa vad care respecta criterial de procentaj.

Pentru prima parte am folosit streamul urmator:

Map<String,Long> activities5 = l.parallelStream()

.filter(t->t.getDuration() <= 300)

.collect(Collectors.groupingBy(t -> t.getActivity(),

Collectors.counting()));

Cheia din map este obtinuta ca sipana acum, prin metoda getActivity(), iar frecventa de aparitie este obtinuta prin folosirea “Collectors.counting()”. Nu sunt numerate toate aparitiile pentru ca exista criterial de filtrare care ia in calcul acele activitati care au durata mai mica de 5 inute, adica 300 de secunde. Acum in activities5 sunt salvate activitatile care dureaza mai putin de 5 minute.

Pentru a doua parte a problemei, am folosit tot un stream:

java.util.List<String> res = l.parallelStream().map(t -> t.getActivity())

.filter(s -> activities5.containsKey(s))

.filter(s -> activities5.get(s) >= 0.9 \* getActivityApperance(s))

.distinct()

.collect(Collectors.toList());

In lista de Stringuri res se cauta in lista tuturor activitatilor acele activitati care se regasesc si in activities5 ( .filter(s -> activities5.containsKey(s) ), care au procentajul cerut (.filter(s -> activities5.get(s) >= 0.9 \* getActivityApperance(s))) si sunt distincte . Metoda getActivityApperance() primeste ca argument numele unei activitati si returneaza numarul total de aparitii.

Lista obtinuta reprezinta activitatile cautate si este scrisa in fisierul de iesire.

**Rezultate si concluzii:**

Streamurile si expresiile lambda reprezinta mecanisme foarte utile in java, care ajuta la calculul unor rezultate de rutina intr-un mod mult mai rapid. Consider ca aceasta tema a fost o buna practica in intelegerea acestor concepte, unele cerinte necesitand un timp de gandire mai indelungat.

Metodele implementate de mine returneaza valorile asteptate si sunt salvate in fisierele cu nume corespunzatoare.