Assignment 5
Operatii pe stream-uri

Student: Găvenea Radu
grupa 30292

1. Obiectivul temei

1.1. Cerinta

O casa inteligenta functioneaza folosind un set de senzori folositi pentru colectarea datelor cu privire la comportamentul persoanelor ce locuiesc in aceasta. Istoricul activitatilor desfasurate in cadrul casei inteligente este stocat intr-un fisier sub forma unei tuple (startTime, endTime, activityLabel), unde startTime si endTime sunt timpii de inceput si de sfarsit a unei activitatii, iar activityLabel reprezinta denumirea activitatii desfasurate de o persoana: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare_Time/TV, Grooming.

Propuneti si implementati o aplicatie de procesare a datelor colectate prin intermediul unui strem de citire dintr-un fisier cu informatii ce reprezinta date provenite de la senzorii unei case inteligente, precum cea de mai sus specificata.

Citirea din fisier se face cu ajutorul stream-urilor, iar datele provenite din stream sunt stocate in cadrul aplicatiei intr-o structura de tipul: List<MonitoredData>, unde *MonitoredData* contine variabilele de instanta *startTime*, *endTime* si *activityLabel.*

Folosind operatii de procesare pe stream-uri si expresii lambda, implementati urmatoarele cerinte:

1. Numarati zilele distincte care apar in data monitorizata
2. Generati o colectie de tipul Map<String,Integer> care mapeaza pentru fiecare activitate, numarul de aparitii din data monitorizata. Scrieti rezultatele intr-un fisier.
3. Generati o structura de date de tipul Map<Integer, Map<String,Integer>> care contine numarul de aparitii a unei activitatii realizate pentru fiecare zi in parte. Scrieti rezultatele intr-un fisier.
4. Determinati o structura dee date de tipil Map<String, DateTime> care mapeaza pentru fiecare activitate durata totala insumata pentru intreaga perioada de timp monitorizata. Filtrati activitatile cu cele care au un timp total mai mare de 10 ore. Scrieti rezultatele intr-un fisier.
5. Filtrati activitatile care au 90% din numarul de activitati monitorizate ce au timpul de desfasurare sub 5 minute si colectati rezultatele intr-o lista List<String> ce contine numele activitatilor respective.

2. Analiza problemei

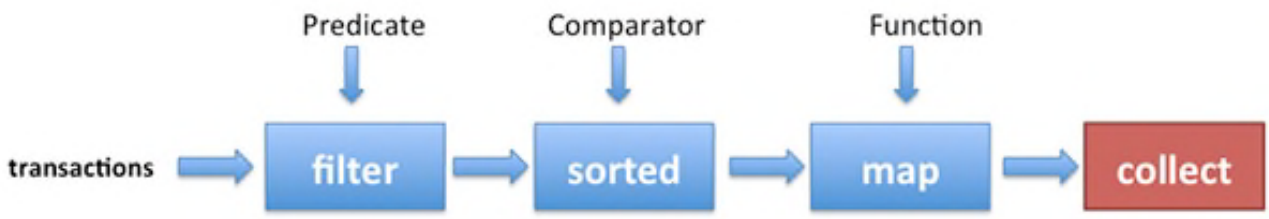
Pentru rezolvarea celor 5 cerinte propuse vom analiza datele provenite din fisier cu privire la activitatiile inregistrate si le vom procesa folosind operatii agregate pe stream-uri si expresii lambda, tehnologii disponibile in java 8.
Aceste tipuri de operatii agregate pe stream-uri sunt expresii sofisticate de procesare de queri-uri asupra datelor stocate in structuri puse la dispozitie de framework-ul Java Collections.
Spunem ca putem efectua operatii de procesare de tip query, deoarece modul de adresare a acestor operatii pe stream-uri sunt foarte asemanatoare ca si concept si ca si functionare cu operatii de tip SQL. Asemanator cu operatiile de tip SQL, cu aceste operatii agregate disponibile in java 8 putem sa cautam si sa filtram colectii (“finding” in SQL), sa grupam obiectele in diferite structuri de date (“gouping”) etc.

Procesarea informatiilor cu ajutorul stream-urilor disponibile in java 8 se face asadar intr-un mod declarativ, iar iteratiile asupra colectiilor sunt interne, si nu externe cum erau pana la java 7, unde dezvoltatorul software avea responsabilitatea de face iteratiile si de a tine cont de limitele de cautare asupra colectiilor pentru a nu primi erori de cautare in afara structurii.

In cadrul procesarii datelor pe strea-uri avem disponibile o multitudine de operatii pe care le putem folosi intr-o maniera declarativa, secvential asupra structurii de date. Aceste operatii se impart in doua mari categorii:

- operatii intermediare: aceste sunt operatii de filtrare, sortare, mapare etc
- operatii terminale: acestea sunt operatii de colectare(reducere) a datelor pentru a fi stocate in structure finala obtinuta.

Aceste tipuri de operatii pot fi observate in exemplul din imaginea de mai jos:



http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/2179048.jpg

Diferentierea tipurilor de operatii este foarte importanta pentru a intelege modul de functionare a operatiilor agregate pe stream-uri, intrucat aceste operatii pipeline sunt operatii de tip “lazy”. Acest lucru inseamna ca operatiile de tip intermediar nu se efectueaza decat in ultimul momentul in care este nevoie ca acest lucru sa se petreaca si anume in momentul in care se executa o operatie de tip terminal. Acest lucru se intampla deoarece operatiile intermediare pot adesea sa fie procesate impreuna in paralel pentru o eficienta sporita.

Un avantaj mare pentru a folosi operatii pe stream-uri pentru procesarea datelor din colectii este acela ca se pot face operatii in paralel pe acesta, maximizand performanta in functie denumarul de nuclee pe care le are un dispozitiv. Acest lucru inseamna ca in cazul in care avem o masina de lucru cu mai multe core-uri si apeland metoda *parallelStream()* in loc de *stream()* vom obtine o performanta sporita a timpului de executie a procesarii datelor.

Un exemplu de astfel de operatii agregate pe stream-uri este urmatoarea bucata de cod:

```
List<Integer> transactionsIds =
    transactions.parallelStream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed(
))
                .map(Transaction::getId)
                .collect(toList());
```

Bucata de cod de mai sus construieste o lista de numere intregi dintr-o structura *transaction,* asupra careia se aplica o metoda de filtrare pentru a se pastra doar elementele ce au proprietatea ca ssunt de tipul Transaction.GROCERY. Apoi se aplica o metoda de sortare in ordine descrescatoare pe baza valorii tranxzactiei, se mapeaza doar id-ul tranzactiei ce urmeaza a fi colecta. In sfarsit, se apeleaza o metoda de colectare a tuturor elementelor care au fost obtinute in urma aplicarii operatiilor intermediare si se stocheaza intr-o colectide tip lista.

Cu ajutorul codului mai sus mentionat, putem observa simplittea de folosire a operatiilor agregate in cadrul procesarii de date cu stream-uri, prin maniera declarativa, usor de citit si de inteles, mai ales daca se respecta regula ca fiecare metoda sa fie scrisa pe un rand nou(sa avem o linie verticala de puncte).

3. Proiectarea

Proiectarea aplicatiei este destul de simpla si directa. Avem un nivel de dataAccessLayer si un nivel de executie a aplicatiei.

In nivelul <dataAccessLayer> sunt prezente clasele folosite pentru accesarea datelor din fisier. Asadar avem o clasa ce are scopul de citi informatia din fisier prin citirea acestuia linie cu linie sub forma unui stream. Acest lucru este disponibil folosind biblioteva *java.nio.file.*,* prin folosirea urmatoarei linii de cod:

```
Stream<String> stream = Files.lines(Paths.get(filename));
```

Obtinem astfel datele din fisier sub forma unui steam, puntand apela in continuare metode de prelurare a fiecarei linii in parte. In acest sens, pentru fiecare linie de cod se construieste un obiect de tipul structurii cerute si se adauga intr-o lista.

Avem si o clasa de salvare in fisiere a rezultatelor obtinute in cadrul implementarii celor 5 cerinte pentru prelucrarea datelor din stream. In cadrul acestei clase sunt prezente metode de scriere in fisier particularizate cu implementare corespunzatoare fiecarui tip de structura in parte.
Tot in cadru lacestei clase avem metode folosite pentru creerea fisierelor in cazul in care acestea nu exista si suprascrierea acestira in cazul in care exista.

O alta clasa prezenta in partea de model a aplicatiei este o clasa de tip entitate de date unde sunt prezente toate atributele pe care aceasta entitate le are si metodele de get si set necesare pentru accesarea acestor atribute.

In nivelul de executie, avem implementata o singura clasa din cadrul careia se apeleaza toate metodele de prelucarare a datelor stocate local in structura de date corespunzatoare. Toate rezultatele sunt salvate local in variabile de instanta locale si salvate ulterior in fisiere separate pentru fiecare cerinta in parte.

4. Implementare si testare

Aplicatia este impartita in doua pachete principale: cel de accesare a datelor din fisier, pachetul <dataAccessLayer> si cel de executie a tuturor cerintelor proiectului implementate in cadrul pachetului <program>.

4.1. Impartirea in pachete

4.1.1. Pachetul <dataAccessLayer>

In cadrul acestui pachet sunt prezente toate clasele necesare pentru citirea, respectiv scrierea daterlor in fisier. Atat citirea, cat si scrierea datelor se face folosind metode din disponibile in cadrul manipularii colectilor folosind stream-uri.

Clasa <ReadFile> contine ca variabiala de instata o structura de date(lista de obiecte de tip MonitoredData) si metode necesare pentru citirea datelor din fisier si salvarea acetora corespunzator in stuctura de date anterior precizata.
Citirea de din fisier se face folosind metode de procesare a stream-urilor disponibile in biblioteca *java.nio.file.**, prin folosirea urmatoarei linii de cod:

```
Stream<String> stream = Files.lines(Paths.get(filename));
```

In urma folosiri acestei metode se pot efectua operatii asupra strem-ului pentru a salva fiecare linie din fisier intr-un obiect de tipul *MonitoreData* ce va fi adaugat la lista declarata ca variabila instanta. Acest lucru poate fi realizat cu urmatoara linie de cod:

```
stream.forEach(line -> addToMonitoredDatas(line));,
```

unde addToMonitoredDatas(line) este o metoda ce are rolul de a mapa o linie primita ca argument la un obiect de tip MonitoredData, apoi sa il adauge in lista.

Clasa<WriteFile> este o clasa folosita in cadrul aplicatiei pentru a scrie rezultatele obtinute in fisiere diferite salvate de disk-ul local. In acest sens avem prezente 4 metode diferite(prima cerinta din cele 5 nu necesita salvare in fisier) de scriere a rezultatelor in cate un fisier, particularizate in functie de structura de date ce trebuie salvata.
De asemenea, in cadrul acestei clase se afla si o metoda de creare a fisierelor(cu tot cu structura de directoare necesare) in cazul in care acestea nu exista si suprascrierea acestora in cazul in care acestea exista.

Clasa <MonitoredData> este o clasa de tip entitate ce are rolul de a pastra informatii cu privire la o activitate primita din fisier prin cadrul stream-ului. Aceasta contine atribute private in care sunt pastrate data de inceput a activitatii, data de finalizare a activitatii si numele activitatii. De asemenea, in cadrul acestei clase sunt prezente si metodele set si get necesare pentru accesarea acestor atribute in parte.

4.1.2. Pachetul <programs>

In cadrul acestui pachet este prezenta o singura clasa in care sunt incluse toate implementarile cerintelor de prelucrare a datelor provenite din fisierul de inregistrarea a activitatilor. Toate cele cinci implementari sunt executate in cadrul aceleasi metode main() la rularea programului, iar rezultatele local obtinute in variabilele corespunzatoare sunt memorate in fisiere apeland metodele corespunzatoare.

4.2. Implementarea cerintelor de prelucrare a datelor pe stream-uri

4.2.1.

Cerinta:

Count the distinct days that appear in the monitoring data.

Implementare:

```
rf.getMonitoredDatas().stream()
                .map(e -> e.getStartTime().getDayOfYear())
                .distinct()
                .count());
```

rf.getMonitoredDatas() este metoda apelata care returneaza structura salvata local cu datele provenite din fisier: List<MonitoredData>. Asupra acestei liste se executa toate operatiilor agregate. Acest lucru este valabil si pentru urmatoarele cerinte, accesarea listei de date fiind identica identica.

- metoda *stream()* transforma lista de obiecte de tip MonitoredData intr-un stream asupra caruia se pot executa operatii agregate.
- metoda *map(e -> e.getStartTime().getDayOfYear())* este folosita pentru a obtine o noua colectie generata doar cu zilele anului in care o activitate incepe
- metoda *distinct()* este folosita pentru pastra doar elementele distincte, si anume doar zilele diferite ale anului anterior generate
- metoda *count()* este o metoda de reducere folosita pentru genera numarul de elemente pe care noua colectie generata il are.

4.2.2.

Cerinta:

Determine a map of type <String, Integer> that maps to each distinct action type the number of occurrences in the log. Write the resulting map into a text file.

Implementare:

```
Map<String,Integer> distinctActionsMapInteger =  rf.getMonitoredDatas().stream()
        .collect(Collectors.groupingBy(
            MonitoredData::getActivityLabel,
            Collectors.reducing(0,e -> 1, Integer::sum)
        ));
```

- metoda *getMonitoredDatas()* returneaza structura de date de tip lista de obiecte MonitoredData, adica lista cu activitati
- metoda *stream()* transforma lista de obiecte de tip MonitoredData intr-un stream asupra caruia se pot executa operatii agregate.
- metoda *collect()* are rolul de colecta si genera o noua colectie in functie de specificarile pe care la are ca parametrii.
- Collectors.groupingBy(
  MonitoredData::getActivityLabel,
  Collectors.reducing(0,e -> 1, Integer::sum))

este o metoda prezenta in interfata *Collectors* ce ne permite sa cream o strctura noue de tip Map in functie de parametrii specificatii. In cazul de fata noua structura va fi un HashMap ce va avea ca si cheie numele activitatii, dat prin apelarea metodei prin referinta *MonitoredData::getActivityLabel* si ca valoare suma tuturor aparitiilor activitatilor distincte calculate cu metoda *reducing*().

4.2.3.

Cerinta:

Generates a data structure of type Map<Integer, Map<String, Integer>> that contains the activity count for each day of the log (task number 2 applied for each day of the log) and writes the result in a text file.

Implementare:

```
Map<Integer, Map<String, Integer>> activitiesEachDayMap =
        rf.getMonitoredDatas().stream()
          .collect(Collectors.groupingBy(
                e -> e.getStartTime().getDayOfMonth(),
                Collectors.groupingBy(
                      MonitoredData::getActivityLabel,
                      Collectors.reducing(0, e -> 1, Integer::sum)
                )));
```

- metoda *getMonitoredDatas()* returneaza structura de date de tip lista de obiecte MonitoredData, adica lista cu activitati
- metoda *stream()* transforma lista de obiecte de tip MonitoredData intr-un stream asupra caruia se pot executa operatii agregate.
- metoda *collect()* are rolul de colecta si genera o noua colectie in functie de specificarile pe care la ia ca parametrii. Vom exemplifica in continuare ce parametrii are metoda si colectie se genereaza
- prima metoda *goupingBy()* este folosita pentru a obtine o structura de tip Map ce are ca si cheie ziua din luna in care se petrec acitivitatile, returnate prin apelarea metodelor *getStartTime().getDayOfMounth()* si ca valoare o noua structura de date generata tot cu ajutorul unei metode *groupingBy(),* din cadrul interfetei Collectors pe care o exemplificam in continuare.
- a doua metoda *groupingBy()* este la fel ca cea descrisa la cerinta anterioara, adica genereaza o structura de date de tip Map ce are ca si cheie numele activitatii si ca valoare numarul tuturor aparitiilor activitatilor. Cheia este generata prin apelarea unei metode prin referinta, si anume *MonitoredData::getActivityLabel,* iar valoarea unui element din HashMap este dat de metoda *reducing(0, e -> 1, Integer::sum),* care calculeaza suma tuturor aparitiilor.

4.2.4.

Cerinta:

Determine a data structure of the form Map<String, DateTime> that maps for each activity the total duration computed over the monitoring period. Filter the activities with total duration larger than 10 hours. Write the result in a text file.

Implementare:

```
Map<String,Hours> totalDurationOfActivitiesMap = rf.getMonitoredDatas().stream()
    .collect(Collectors.groupingBy(
          MonitoredData::getActivityLabel,
          Collectors.reducing(
                Seconds.ZERO,
                e -> {
                    return Seconds.secondsBetween(e.getStartTime(),e.getEndTime());
                },
                (t1,t2) -> {
                    return t1.plus(t2);
                }
          )
    )).entrySet().stream()
    .filter(e -> e.getValue().compareTo(Seconds.seconds(36000)) > 0)
    .collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue().toStandardHours()));
```

- metoda *getMonitoredDatas()* returneaza structura de date de tip lista de obiecte MonitoredData, adica lista cu activitati
- metoda *stream()* transforma lista de obiecte de tip MonitoredData intr-un stream asupra caruia se pot executa operatii agregate.
- metoda c*ollect()* are rolul de colecta si genera o noua colectie in functie de specificarile pe care la ia ca parametrii. Vom exemplifica in continuare ce parametrii are metoda si colectie se genereaza
- metoda *groupingBy()* genereaza o structura de date de tip HashMap cu elemente ce au ca si cheie numele activitatii si ca si valoare valoarea totala insumata a timpilor care s-au petrecut pentru desfasurarea fiecarei activitati. Numele activitatii este returnat de metoda exprimata prin referinta *MonitoredData::getActivityLabel,* iar calcularea timpului total petrecut in cadrul desfasurarii unei activitati este dat de executia metodei *reducing()* ce are ca si parametrii o functie **identitate** *Seconds.ZERO,* de la care porneste, o functie de **mapare** a elementului cu care se vor face operatiile si un **operator** prin care se specifica exact care este operatia intre cele doua elemente de reducere.
- asupra colectii nou generate se aplica metoda *entrySet()* pentru a obtine toate elementele structurii de tip Map
- se apeleaza din nou metoda *stream()* care transforma lista elementelor intr-un stream asupra caruia se pot executa operatii agregate.
- metoda *filter()* compara toate valorile obtinute in valorile structurii Map anterior obtinute si filtreaza toate elementele care au valoarea timpului total mai mare de 36000 de secunde, adica 10 ore.
- metoda *collect()* colecteaza valorile mai sus filtrate si le mapeaza cu ajutorul metodei *toMap()* la o noua structura de date care contine exprimare valorii timpului in ore, si nu in secunde.

4.2.5

Cerinta:

> Filter the activities that have 90% of the monitoring samples with duration less than 5 minutes, collect the results in a List<String> containing only the distinct activity names and write the result in a text file.

Implementare:

```
List<String> lessThanFiveNinetyPercent =
        rf.getMonitoredDatas().stream()
            .collect(Collectors.groupingBy(
                    MonitoredData::getActivityLabel,
                    Collectors.groupingBy(
                            e -> {
                                if(isLessThanFive(e)) return "less";
                                else return "more";
                            }
                    )
            ))
            .entrySet().stream()
                .filter(e -> isMoreThanNinetyPercent(e.getValue()))
                .map(e -> e.getKey())
                .collect(Collectors.toList());
```

- metoda *getMonitoredDatas()* returneaza structura de date de tip lista de obiecte MonitoredData, adica lista cu activitati
- metoda *stream()* transforma lista de obiecte de tip MonitoredData intr-un stream asupra caruia se pot executa operatii agregate.
- metoda c*ollect()* are rolul de colecta si genera o noua colectie in functie de specificarile pe care la ia ca parametrii. Vom exemplifica in continuare ce parametrii are metoda si colectie se genereaza
- metoda *groupingBy()* genereaza o structura de date de tip HashMap cu elemente ce au ca si cheie numele activitatii si ca si valoare o alta structura de date de tip HashMap generata prin apealrea din nou a metodei *groupingBy()*. Numele activitatii este returnat de metoda exprimata prin referinta *MonitoredData::getActivityLabel,* iar noua structura o vom exemplifica la urmatorul punct
- a doua metoda *groupingBy()* imbricata are rolul de a forma o structura HashMap ce va contine o lista cu elementele ce reprezinta activitati ce sunt desfasurate in mai putin de 5 minute si o lista cu elementele ce reprezinta activitatile desfasurate in mai mult de 5 minute. In acest sens prin folosirea unei expresii lamba care returneaza un srting “less” pentru activitatile mai scurte de 5 minute si un string “more” in caz contrar. Asadar se formeaza o structura HashMap ce are ca si cheie un String (“less”, respectiv “more”) si ca valoare contine liste cu activitatile corespunzatoare (o lista de obiecte MonitoredData)
- asupra noii colectii obtinute se apeleaza metoda *entrySet()* pentru a obtine toate elementele structurii de tip Map.
- metoda *stream()* se apeleaza din nou asupra elementelor noii colectii pentru a genera un stream asupra caruia se pot efectua operatii agregate de procesare a datelor.
- metoda *filter()* este apelata pentru a filtra toate doar elementele care in 90% sau mai mult din cazuri au timpul de monitorizare de sub 5 minute. Pentru acest lucru cu ajutorul unei expresii lambda se apeleaza o functie ce ca si parametru o structura HashMap <String, List <MonitoredData>> ce contine cele doua liste cu activitati ce au cheile “more” si “less” anterior generate. Aceasta functie calculeaza daca numarul de activitati desfasurate in mai putin de 5 minute reprezinta cel putin 90% din totalul de activitati si returneaza o valoare booleana in functie de rezultatul calculat.
- metoda *map()* se foloseste pentru a extrage informatia dorita si a forma o noua colectie cu aceasta. In caul de fata, conform cerintei, trebuie sa extragem doar numele activitatii, iar acesta se regaseste in cheia structurii HashMap
- metoda *collect()* este folosita ca si operatie terminala pentru a genera colectia obtinuta in urma filtrari si a maparii sub forma unei liste, datorita apelarii metodei *toList()*.