Tema 5

Colectare date

Proiect realizat de

Bursuc Andrei

Seria A, Grupa 30224

Contents

[**1.** **Cerinte functionale** 3](#_Toc483065177)

[**2.** **Obiectivul temei** 3](#_Toc483065178)

[**3.** **Analiza problemei, modelare, scenarii, cazuri de utilizare** 3](#_Toc483065179)

[**4.** **Proiectare** 4](#_Toc483065180)

[I. Decizii de proiectare 4](#_Toc483065181)

[II. Diagrama UML 5](#_Toc483065182)

[III. Proiectarea claselor și relațiile dintre ele 5](#_Toc483065183)

[**5.** **Concluzii si dezvoltari ulterioare** 9](#_Toc483065184)

# **Cerinte functionale**

A smart house features a set of sensors that may be used to record the behavior of a person living in the house. The historical log of the person’s activity is stored as tuples (startTime, endTime, activityLabel), where startTime and endTime represent the date and time when each activity has started and ended while the activity label represents the type of activity performed by the person: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming. The attached log file Activities.txt contains a set of activity records over a certain period of time. Define a class MonitoredData having startTime, endTime and activityLabel as instance variables and read the input file data into the data structure monitoredData of type List. Using stream processing techniques and lambda expressions introduced by Java 8, write the following set of short programs for processing the monitoredData.

1. Count the distinct days that appear in the monitoring data.

2. 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.

3. 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.

4. 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.

5. 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.

# **Obiectivul temei**

Acest proiect urmareste realizarea unei aplicatii care ajuta la organizarea datelor colectate de senzorii unei case inteligente. Datele pe care noi trebuie sa le extragem din fisierul cu intregistrarile actiunilor locatarului casei sunt: numarul de zile distincte de-a lungul carora s-a efectuat colectarea datelor, numarul total de aparitii al fiecarei activitati in fisier, numarul de aparitii al fiecarei activitati pentru fiecare zi in parte, durata totala a fiecarei activitati in parte si activitatile care in 90% din cazuri dureaza mai putin de 5 minute.

# **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Problema pe care o ridica aceasta aplicatie este de modul in care putem colecta datele cat mai eficient, scriind cat mai putine linii de cod. Pentru a rezolva aceasta problema vom folosi expresiile lambda din standardul Java 8 ce ne permit sa scriem mult mai putin cod pentru a parcurge colectii si colecta date din ele.

# **Proiectare**

## Decizii de proiectare

Pentru realizarea aplicatiei am folosit expresiile lambda introduse in limbaj odata cu standardul Java 8, dar si API-ul stream introdus in acelasi standard.

O funcţie lambda (funcţie anonimă) este o funcţie definită şi apelată fără a fi legată de un identificator. Funcţiile lambda sunt o formă de funcţii ,,incuibate” (nested functions) în sensul că permit accesul la variabilele din domeniul funcţiei în care sunt conţinute.

Funcţiile anonime au fost introduse de către Alonzo Church în anul 1936, în teoria sa despre calculele [lambda](http://en.wikipedia.org/wiki/Lambda_calculus).

În limbajele de programare, funcţiile anonime sunt implementate din anul 1958 ca parte a limbajului Lisp. În unele limbajele orientate pe obiect, precum Java, apar concepte similare, precum clasele anonime. Abia în versiunea 8 a limbajului Java sunt adăugate şi funcţiile anonime. Alte limbaje, precum C#, JavaScript, Perl, Python, Ruby ofereau demult suport pentru acest concept.

Lambda expresiile ne permit să creăm instanţe ale claselor cu o singură metodă într-un mod mult mai compact.

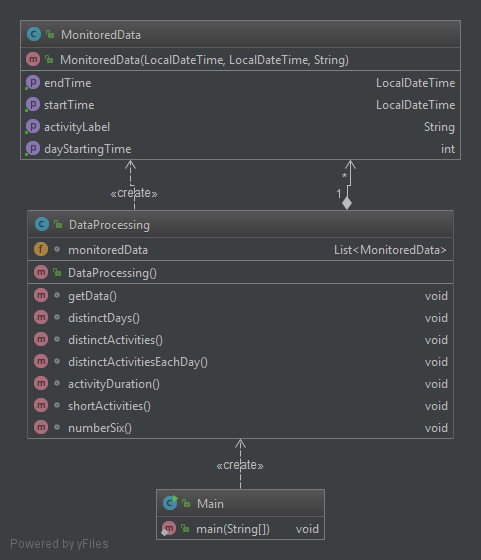
O *lambda expresie* constă:

* dintr-o listă de parametri formali, separaţi prin virgulă şi cuprinşi eventual între paranteze rotunde,
* săgeata direcţională ->,
* un body ce constă dintr-o expresie sau un bloc de instrucţiuni.

O interfaţă funcţională (*functional interface*) este orice interfaţă ce conţine doar o metodă abstractă. Din această cauză putem omite numele metodei atunci când implementăm interfaţa şi putem elimina folosirea claselor anonime. În locul lor vom avea *lambda expresii*. O interfaţă funcţională este anotată cu @FunctionalInterface.

Ca urmare a folosirii API-ului stream operaţiile efectuate pe o colecţie pot fi destul de complexe şi anume: filtrarea după un predicat de selecţie, maparea obiectului filtrat, respectiv executarea unei acţiuni pe fiecare obiect mapat.

## Diagrama UML



## Proiectarea claselor și relațiile dintre ele

**Clasa MonitoredData**

private LocalDateTime **startTime**;  
private LocalDateTime **endTime**;  
private String **activityLabel**;

Aceasta clasa este definita de variabilele startTime, endTime de tip LocalDateTime si variabila activityLabel de tip String. Ele sunt mulate pe modul in care casa inteligenta a stocat datele in fisierul .txt.

public **MonitoredData**(LocalDateTime startTime, LocalDateTime endTime, String activityLabel) {  
 this.**startTime** = startTime;  
 this.**endTime** = endTime;  
 this.**activityLabel** = activityLabel;  
}

Constructorul primeste ca argumente variabile de tipul variabilelor si nu face decat sa transmita argumentele la variabilele ce definesc obiectul. Restul metodelor sunt gettere.

public LocalDateTime getStartTime() {  
 return **startTime**;  
}  
  
public LocalDateTime getEndTime() {  
 return **endTime**;  
}  
  
public String getActivityLabel() {  
 return **activityLabel**;  
}  
  
public int getDayStartingTime(){  
 return this.**startTime**.getDayOfMonth();  
}

De mentionat fiind faptul ca getterul „getDayStartingTime” preia ziua din luna (zilele sunt numerotate de la 1 la 31) din atributul startTime.

**Clasa DataProcessing**

List<MonitoredData> **monitoredData**;

Singurul atribut al acestei clase este lista de obiecte de tipul MonitoredData in care practic colectam toate datele din fisier.

public **DataProcessing**()  
{  
 **monitoredData**=new ArrayList<MonitoredData>();  
}

Constructorul joaca acelasi rol ca si in cazul clasei MonitoredData.

void getData()  
{  
 LocalDateTime **startTime**,**endTime**;  
  
 String **activity**;  
 MonitoredData **monitoredDataField**;  
 try {  
 Scanner **input**=new Scanner(new File("D:\\Downloads Chrome\\TP2017\_30224\_Bursuc\_Andrei\_Tema\_5\\src\\main\\java\\Activities.txt"));  
 while(**input**.hasNextLine())  
 {  
 String **line**=**input**.nextLine();  
 String[] **fields**=**line**.split("\t\t");  
 DateTimeFormatter **inputFormat** =DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  
 **startTime**=LocalDateTime.parse(**fields**[0],**inputFormat**);  
 **endTime**=LocalDateTime.parse(**fields**[1],**inputFormat**);  
 **activity**=**fields**[2];  
 **monitoredDataField**=new MonitoredData(**startTime**,**endTime**,**activity**);  
 **monitoredData**.add(**monitoredDataField**);  
   
 }  
  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
}

Metoda getData citeste datele din fisierul „Activities.txt”, citirea se face cu ajutorul unui obiect de tip Scanner intr-o bucla de tip while care impune o conditie ce permite ca citirea sa se faca atata timp cat mai exista linii cu text. Dupa linia o salvam intr-un obiect de tip String pe care il impartim in mai multe string-uri cu comanda split. Primele doua stringuri obtinute le salvam in obiecte de tip LocalDateTime folosind metoda „.parse” specifica acestei clase, iar ultimul intr-un obiect de tip String. Obiectele noi obtinute le transmitem ca argumente constructor-ului unui obiect de tip MonitoredData pe care il adaugam in lista monitoredData in care stocam toate datele din fisier.

void distinctDays()  
{  
 long **distinct**=**monitoredData**.stream()  
 .map(x->x.getStartTime().getDayOfMonth())  
 .distinct()  
 .count();  
 //dates.forEach(date -> System.out.println("Data "+date));  
 System.***out***.println("Distinct days: "+**distinct**);  
}

Cu ajutorul acestei metode numaram toate zilele distincte in care casa inteligenta a stocat date despre activitatea locatarului, acest lucru a fost folosit utilizand API-ul stream si expresii lambda ce ne-au permis sa scriem mai putine linii de cod decat am fi scris daca am fi folosit elemente ce nu erau prezente in JAVA 8. Cu ajutorul „.map” impunem transformarea datelor din MonitoredData in Integer ce reprezinta zilele lunii, dupa metodele „.distinct” preia doar zilele diferite si „.count” le numara.

void distinctActivities()  
{  
 //String activity;  
 Map<String,Long> **distinctAction**=**monitoredData**.stream().collect(  
 Collectors.groupingBy(MonitoredData::getActivityLabel,Collectors.counting()));  
  
 **distinctAction**.forEach((activity,count)->System.***out***.println("Activitate: "+activity+"\t\tAparitie: "+count));  
  
}

In aceasta metoda folosindu-ne de metoda GrupingBy din clasa Collectors numaram aparitiile activitatilor in cazul acestui fisier, acest lucru fiind mult mai greu de realizat in Java 7.

void distinctActivitiesEachDay() {  
 try {  
 PrintStream **out** = new PrintStream(new FileOutputStream("distinctActivitiesEachDay.txt"));  
 Map<Integer, Map<String, Long>> **distinctAction** = **monitoredData**.stream()  
 .collect(Collectors.groupingBy(MonitoredData::getDayStartingTime, Collectors.groupingBy(MonitoredData::getActivityLabel, Collectors.counting())));  
  
 **distinctAction**.forEach((activity, count) -> count.forEach((activity2, count2) -> out.println("Zi: " + activity + " Activitate: " + activity2 + "\t\tAparitie: " + count2)));  
 }catch (IOException e) {  
 e.printStackTrace();  
 }  
 //distinctAction.forEach((day,activity)->);  
}

In cadrul acestei metode particularizam cazul studiat in metoda de mai sus si numaram aparitiile fiecarei activitati pentru fiecare zi in parte, pentru a realiza acest lucru folosim de data asta doua metode GroupingBy imbricate.

void activityDuration()  
{  
 try {  
 PrintStream **out** = new PrintStream(new FileOutputStream("activityDuration.txt"));  
 Map<String,Long> **activityDuration**=**monitoredData**.stream()  
 //.filter(i->ChronoUnit.MINUTES.between(i.getStartTime(),i.getEndTime())>600)  
 .collect(Collectors.toMap(i->i.getActivityLabel(),i->ChronoUnit.***MINUTES***.between(i.getStartTime(),i.getEndTime()),(val1,val2)->val1+val2))  
 .entrySet().stream()  
 .filter(i->i.getValue()>=60\*10)  
 .collect(Collectors.toMap(i->i.getKey(),i->i.getValue()));  
 //Map<String,Long> filteredActivityDuration=activityDuration.entrySet().stream()  
  
 **activityDuration**.forEach((activity,duration)->out.println("Activitate: "+activity+"\t\tDuratie: "+duration));  
  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
}

In cazul acestei metode calculam durata totala a fiecarei activitati de-a lungul perioadei de timp in care casa inteligenta a colectat date despre locatar, dupa stocam intr-un fisier doar acele activitati care au avut o durata totala mai mare de 10 ore. Realizam acest lucru folosind metoda .collect careia ii furnizam drept argumente doua expresii lambda cu care preluam numele activitatii si durata activitatii ( se face diferenta intre endTime si startTime), dar si un argument ce specifica faptul ca atunci cand avem doua elemente cu aceeasi cheie valorile lor se aduna. Dupa parcurgem elementele noi obtinute si le pastram doar pe acele care dureaza mai mult de 10 ore, folosind un filter. In final colectam datele folosind din nou un collect.

void shortActivities()  
{  
 try{  
 PrintStream **out** = new PrintStream(new FileOutputStream("shortActivities.txt"));  
  
 Map<String,Long> **shortTime**=**monitoredData**.stream()  
 .filter(i->ChronoUnit.***SECONDS***.between(i.getStartTime(),i.getEndTime())<5\*60)  
 .collect(Collectors.toMap(i->i.getActivityLabel(),i->ChronoUnit.***SECONDS***.between(i.getStartTime(),i.getEndTime()),(val1,val2)->val1+val2));  
  
 Map<String,Long> **totalTime**=**monitoredData**.stream()  
 .collect(Collectors.toMap(i->i.getActivityLabel(),i->ChronoUnit.***SECONDS***.between(i.getStartTime(),i.getEndTime()),(val1,val2)->val1+val2));  
  
 List<String> **activities**=**shortTime**.entrySet().stream()  
 .filter(i->totalTime.get(i.getKey())-i.getValue()<0.1\*totalTime.get(i.getKey()))  
 .map(i->i.getKey())  
 .collect(Collectors.toList());  
   
 **activities**.forEach(i-> out.println("Activity: "+i));  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
}

Ultima metoda consta in colectarea acelor activitati care in 90% din cazuri dureaza mai putein de 5 minute. Pentru a realiza asta ne folosim de trei structuri, in prima adunam pentru toate activitatile toate acele perioade de timp mai scurte de 5 minute, dupa calculam durata totala de timp pentru toate activitatile. Si in final realizam filtrarea activitatilor care in 90% din cazuri dureaza mai putin de 5 minute, acest lucru il realizam aplicand un filtru care verifica daca diferenta totala dintre durata totala si durata activitatilor scurte este mai mica de 10% din durata totala.

# **Concluzii si dezvoltari ulterioare**

Aceasta aplicatie poate fi folosita in scopuri de cercetare pentru a filtra datele colectate de mai multi senzori fara a fi nevoie sa parcurgi manual toate elementele din fisier.

In urma realizarii acestui proiect am reusit sa aflu mai multe despre expresii lambda si API-ul stream, notiuni introduse odata cu Java 8. Prin aprofundarea acestor notiuni am reusit sa invat metode mai usoare prin care pot sa colectez date dintr-o colectie, dar si sa programez mai rapid si sa scriu cod mai usor de inteles de alti programatori.

# **Bibliografie**

1. <http://docs.oracle.com/javase/>
2. <http://stackoverflow.com/>
3. http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html