

**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

PROCESSING SENSOR DATA OF DAILY LIVING ACTIVITIES

Documentație

Dorofte Andrei

Grupa 302210

Cuprins

1. Obiectivul temei

2. Analiza problemei, modelare, scenarii, cazuri de utilizare

3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

4. Implementare

5. Rezultate

6. Concluzii

7. Bibliografie

1. **Obiectivul temei**

Proiectati si implementati o aplicatie de simulare care sa analizeze comportamentul unei persoane, in functie de datele inregistrate de catre senzorii amplasati in casa acesteia.

O astfel de aplicatie se poate observa si in viata reala, dezvoltandu-se astfel termenul de „smart house”.

Se da un jurnal, in care sunt trecute, pe fiecare rand, sub forma de tuple, dupa cum urmeaza: data si ora de inceput a activitatii, data si ora de sfarsit a activitatii si numele acesteia. Aplicatia trebuie sa analizeze jurnalul si sa realizeze urmatoarele cerinte:

* Task 1: Sa salveze intr-o strctura de tip lista, activitatile citite din fisier, cu ajutorul stream-urilor si expresiilor lambda. Asupra listei create se vor aplica operatii specifice stream-urilor si expresiilor lambda, pentru a spori viteza de procesare asupra datelor.
* Task 2: Sa numerre cate zile distincte apar de-a lungul perioadei de monitorizare, cu ajutorul, cu ajutorul stream-urilor si expresiilor lambda.
* Task 3: Sa numere de cate ori s-a realizat o activitate de-a lungul perioadei de monitorizare, cu ajutorul stream-urilor si expresiilor lambda.
* Task 4: Sa numere de cate ori s-a realizat o activitate de-a lungul fiecarei zile de monitorizare, cu ajutorul stream-urilor si expresiilor lambda.
* Task 5: Sa calculeze durata toala a fiecarei activitati de-a lungul perioadei de monitorizare, cu ajutorul stream-urilor si expresiilor lambda.
* Task 6: Sa afiseze intr-o structura de tip lista, activitatile ale caror durata a fost mai mica de 5 min, in peste 90% din perioada de monitorizare, cu ajutorul stream-urilor si expresiilor lambda.

|  |
| --- |
| **Obiective secundare** |
| * Crearea claselor necesare |
| * Definirea metodelor de baza pentru fiecare clasa |
| * Crearea interfetelor necesare si legarea de proiect |
| * Implementarea operatiilor fiecarui task, cu stream-uri si expresii lambda |
| * Realizarea algoritmilor care prelucreaza operatiile |
| * Realizarea scrierii in fisier |

Pentru a efectua aceste operatii eficient din punct de vedere al timpului de executie si al codului scris, se vor folosii stream-uri si expresii lambda. Stream-urile se pot aplica asupra colectiilor de date. Astfel, stream-urile reprezinta o alta modalitate rapida de a procesa colectiile de obiecte.

Pentru a putea opera asupra datelor din fisier vom citi cu ajutorul stream-urilor fiecare linie, si o vom parsa si imparti, cu ajutorul metodei split si a regex-ului „\t\t” (prin care sunt impartite datele de pe fiecare linie), corespunzator astfel incat sa putem completa atributele unui nou obiect al clasei MonitoredData, care are start\_time, end\_time (ambele de tip LocalDateTime) si activity\_label (de tip String). Am folosit tipul LocalDateTime pentru a putea avea libertatea de a alege dintr-un numar mai generos de metode de lucru cu timpul, intrucat majoritatea cerintelor se rezolva prin calcularea duratei activitatilor. Dupa citirea unei linii metoda reas\_aux va parsa linia si va returna un nou obiect de tip MonitoredData, convertind astfel fiecare linie intr-un obiect, urmand ca mai apoi sa fie colectat intr-o lista de obiecte de tip MonitoredData. Acum ca datele se afla intr-o lista putem sa efectuam operatiile necesare asupra acestora, cu ajutorul stream-urilor.

Activitatile pot fi: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming.

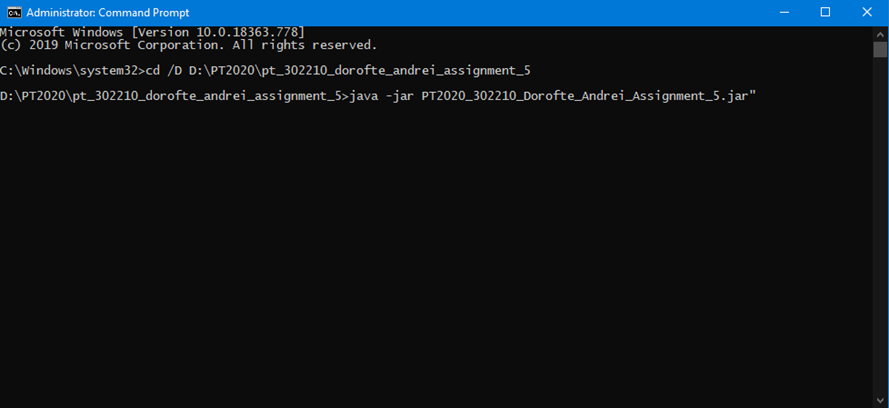
1. **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Cu totii poate ne dorim sa traim intr-o casa „smart”, in care multe dintre activitatile pe care le facem zilnic, sunt automatizate. Acest lucru presupune, de asemenea, crearea unui jurnal de monitorizare al fiecarei activitati pe care o facem. Acest lucru ne poate ajuta sa ne organizam mult mai bine timpul, prin observarea duratei fiecarei activitati pe care o facem.

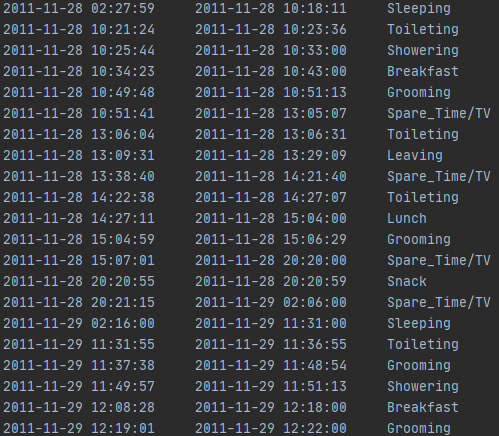
Acest proiect are rolul de a simula operatii care au loc asupra unui jurnal de monitorizare, creat de-a lungul mai multor zile. Sistemul trebuie sa realizeze cele 6 task-uri mentionate mai sus, aplicatia fiind rulata din linia de comanda.

**Datele de intrare:**

Utilizatorul ruleaza aplicatia din Command Prompt, cu ajutorul comenzii „java -jar PT2020\_302210\_Dorofte\_Andrei\_Assignment\_5.jar” si se asigura ca fisierul Activities.txt este valid:



Activities.txt:



**Datele de iesire:**

Utilizatorul verifica, in acelasi folder cu aplicatia, cele 6 fisiere text create, cu rezultatele aferente fiecarui task.

Application

Task 1

Male icon

Male icon

Task 2

Task 3

User

Text files for every task

User

Task 4

Task 5

Task 6

Caz de utilizare:

1. Utilizatorul ruleaza aplicatia dintr-o fereastra command prompt, deschisa din folder-ul in care se afla aplicatia, utilizand comanda „java -jar App.jar” si se asigura ca fisierul Activities.txt este valid.
2. Se asteapta terminarea rularii aplicatiei.
3. Se verifica fisierele text cu rezultatele fiecarui task.
4. **Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfață utilizator)**

Unified Modeling Language sau UML, pe scurt, este un limbaj standard pentru descrierea de modele si specificatii pentru software. UML a fost la bază dezvoltat pentru reprezentarea complexității programelor orientate pe obiect, al căror fundament este structurarea programelor pe clase, și instanțele acestora ( numite și obiecte ). Cu toate acestea, datorită eficienței și clarității în reprezentarea unor elemente abstracte, UML este utilizat dincolo de domeniul IT.

**Diagrama UML:**

A picture containing screenshot

Description automatically generated

Am încercat sa modularizez cerința cat de mult posibil, astfel rezultând clasele din diagrama de mai sus: clasa Main, interfata FileWorker si clasa MonitoredData. Clasele au fost proiectate intuitiv, pentru a fi ușor de citit si de înțeles.

Am creat doar pachetul App, deoarece am considerat ca este suficient pentru cele 3 clase rezultate.

În Java, o structura de date este un mod de stocare a informațiilor și datelor, în așa fel încât ele să poată fi folosite în mod eficient. Deseori o alegere bine făcută a structurii de date va permite și implementarea unui algoritm eficient. Structura de date aleasă este derivată de multe ori dintr-un tip de date. Din cauză că structurile de date au o importanță atât de mare, multe dintre ele sunt incluse în bibliotecile standard ale multor limbaje de programare și medii de dezvoltare, cum ar fi Java Collections Framework .

Stream-urile reprezinta o multime de obiecte venite dintr-o sursa, care suporta operatii de agregare cum ar fi: map, filter, forEach, sorted. Exista doua tipuri de Stream: normal si parelel. Pentru generarea unor rezultate mai detaliate se poate folosi Collectors, cu ajutorul caruia rezultatele pot fi grupate. In acest proiect am utilizat structuri de date de tip Map<String, Integer> (unde String este cheia, si Integer este valoarea asociata cheii) si Map<Integer, Map<String, Integer>> (unde Integer este cheia, si Map<String, Integer> este valoarea asociata cheii).

Am ales utilizara Map-ului deoarece este o caracteristica foarte importanta pentru Stream-uri si se lucreaza foarte eficient cu aceasta structura. De asemenea, am folosit List<MonitoredData> in care am salvat toate activitatiile. Am folosit aceasta structura la generarea map-urilor, deoarece avand toate activitatiile intr-o lista a fost mai usor de filtrat dupa cerintele cerute.

Date fiind cerintele, expuse mai jos, se va implemenata aceasta aplicatie:

* Se defineste o clasa cu 3 campuri: start\_time, end\_time, si activity\_label. Datele se citesc dintr-un fisier "Activities.txt" folosind stream-uri si facandu-se split fiecarei linii, incepand cu prima.
* Se creeaza o lista de obiecte de tip MonitoredData.
* Se numara cate zile diferite apar pe intreaga durata a monitorizarii.
* Se numara de cate ori apare fiecare activitate pe intreaga durata a monitorizarii, si se returneaza un map de tipul <String, Integer> reprezentand maparea a fiecarei activitati cu numarul de aparitii a acesteia.
* Se numara de cate ori apare fiecare activitate pe durata fiecarei zile de pe parcusul intregii monitorizarii.
* Pentru fiecare linie, se mapeaza durata activitatii.
* Pentru fiecare activitate calculeaza intreaga durata de pe parcursul intregii monitorizarii.
* Se filtreaza activitatiile care au peste 90% aparitii care au durata mai mica decat 5 minute.
* Pentru toate aceste cerinte este ceruta folosirea stream-urilor si a expresiilor lambda.

1. **Implementare**

**Clasa Main**

Aceasta clasa implementeaza interfata FileWorker si creeaza o lista de tip MonitoredData in care se stocheaza datele citite cu ajutorul metodei read() din interfata. Se creeaza apoi un obiect de tip MonitoredData si se apeleaza functiile ce rezolva task-urile cerute.

**Clasa MonitoredData**

Aceasta clasa este nucleul aplicatiei si include metodele ce rezolva fiecare dintre task-urile cerute. Contine 3 atribute private: start\_time, de tip LocalDateTime, end\_time, de tip LocalDateTime si activity\_label, de tip String. Astfel, aceasta clasa modeleaza un obiect de tip MonitoredData, ce va fi ulterior extras de catre metoda read, din interfata FileWorker, din fisierul text Activities.txt.

Astfel, clasa include metoda getDistinctDays(List<MonitoredData> timeline), ce rezolva a doua cerinta, ce presupune calcularea numarului de zile distincte ce apar in perioada de monitorizare, metoda nrActivities(List<MonitoredData> timeline), ce rozolva cerinta 3, ce presupune calcularea numarului de aparitii a fiecarei activitati de-a lungul perioadei de monitorizare.

Metoda activityPerDay(List<MonitoredData> timeline) rezolva cea de-a patra cerinta, ce presupune calcularea numarului de aparitii a fiecarei activitati in fiecare zi din perioada de monitorizare. Am folosit metoda getDayOfYear() ce apartine de LocalDateTime, deoarece orice alta metoda (ex. getDayOfMonth), ar fi condus la mici erori, in sensul ca ziua de 1 a lunii ar fi fost adunata si pentru luna a 11-a si pentru luna a 12-a. Astfel, cu metoda folosita, se diferentiaza usor aceste cazuri de coincidenta.

Metoda totalDuration(List<MonitoredData> timeline) rezolva cerinta a cincea, ce presupune calcularea duratei totale a fiecarei activitati de-a lungul perioadei de monitorizare. Pentru a calcula durata unei activitati am folosit transformarea in secunde, deoarece tranformarea in minute ar fi condus la mici erori, urmand ca in metoda de afisare din interfata FileWorker sa transform secundele in ore, repectiv minute, pentru o citire mai eficienta.

Metoda activityDuration(List<MonitoredData> timeline) rezolva ultima cerinta, ce presupune afisarea activitatilor ale caror durata a fost mai mica de 5 min, in peste 90% din perioada de monitorizare

De asemenea, sunt prezente si metodele care apeleaza metoda de scriere in fisier din interfata FileWorker, in functie de task-ul rezolvat.

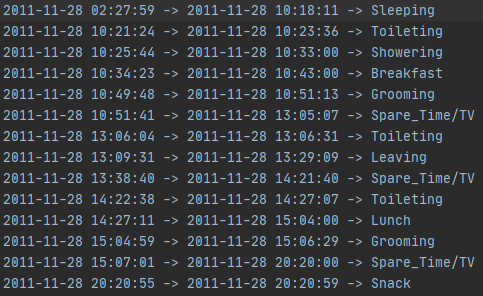
**Interfata FileWorker**

In aceasta interfata se afla metoda read si read\_aux, cu ajutorul carora se realizeaza citirea din fisier si construirea listei de obiecte de tip MonitoredData, dar si metoda de scriere in fisier in functie de cerinta. Pentru a identifica despre care cerinta este vorba, am folosit o variabila i ca parametru, ce indica numarul task-ului ce a fost rezolvat si urmeaza a fi scris in fisier. Pentru scrierea in fisier, am folosit BufferedWriter. Pentru parcurgerea structurilor de date de tip lista, Map<String, Integer> si Map<String, Map<String, Integer>>, am folosit forEach.

1. **Rezultate**

Pentru cazul de testare am testat aplicatia pentru una dintre zilele din perioada de monitorizare. Am calculat pe hartie rezultatul fiecarei cerinte, utilizand fisierul text de intrare, dupa care am comparat rezultatul cu cel scris in fisierul text aferent cerintei.

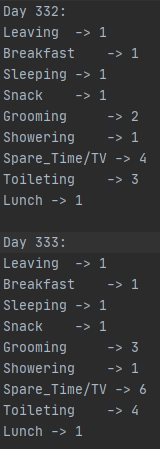
Rezultatele din etapa de testare arata ca aplicatia este modelata corect, iar algoritmii de lucru pentru fiecare cerinta implementata, sunt functionali, si genereaza rezultatele asteptate.

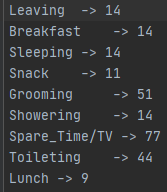


Task 1:

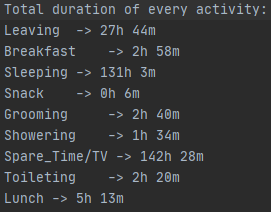
Task 2:



Task 3: Task 4:



Task 5:





Task 6:

1. **Concluzii**

In concluzie, sunt de părere ca acest proiect m-a ajutat sa îmi aprofundez cunoștințele in tot ce înseamnă limbajul Java, implementarea paradigmelor OOP, si dezvoltarea unui algoritm eficient, care sa satisfacă obiectivul unui proiect.

Aceasta tema simuleaza situatii din viata de zi cu zi, intrucat, oricat de bine ar suna termenul de „smart home”, nu ne-am fi gandit, probabil, ce presupune acest lucru in background.

Chiar daca simularea este corecta, ea nu poate prevedea toate situatiile neasteptate din realitate. In aceasta tema am invatat sa lucrez cu Stram-uri si expresii lambda. De asemenea, am aprofundat lucrul cu structuri de tip Map si List, dar si sa generez un document de tip text utilizand limbajul Java. De asemenea, am aprofundat generarea si configurarea unui fisier .jar si rularea din linia de comanda.

In ceea ce privește dezvoltarea ulterioara a programului:

* Se poate imbunatati performanta algoritmilor
* Se pot implementa operatii de analiza a activitatilor pe care le face fiecare persoana care locuieste in casa. (de exemplu o familie)
* Se poate implementa monitorizarea consumului de curent, apa, etc.

1. **Bibliografie**

* <https://stackoverflow.com/>
* <http://docs.oracle.com/>
* <https://www.geeksforgeeks.org/>
* youtube.com - tutoriale