**DOCUMENTATIE TEMA 5**

**PROCESSING SENSOR DATA OF DAILY LIVING ACTIVITIES**

**Tudor Colceriu**

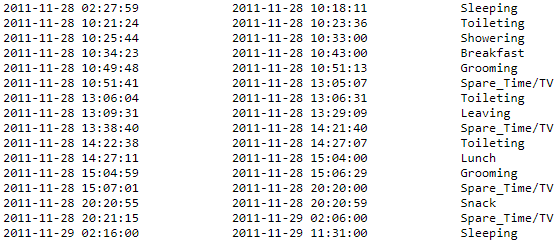
**Grupa 30227**

**Profesor Laborator: Dan Mitrea**

# Cerinte Functionale

Pentru aceasta tema se doreste gandirea, implementarea si rularea (testarea) unei aplicatii ce ar analiza comportamentul unei persoane, pe o anumita perioada de timp, monitorizat prin senzori instalati in propria casa. Datele furnizate de acesti senzori, pentru respectiva persoana, sunt stocate intr-un fisier text (.txt) numit „Activities.txt”. In acest fisier tipul informatiilor este asemanator cu cel al unui tabel, stocat prin randuri si coloane. Pentru un fisier de acest tip vom avea urmatoarele informatii prezente in el: timpul de inceput al activitatii (start\_time), timpul cand activitatea s-a terminat (end\_time) si eticheta activitatii (activity\_label). Timpul de start si de final pentru fieacare activitate in parte il reprezinta un sir de caractere (strig), ce reprezinta data si ora la care acea activitate a inceput / s-a terminat, dupa caz. Formatul acestui string este: ***an-luna-zi ora:minut:secunda***. Pentru fiecare activitate in parte se vor retine aceste 2 informatii.

Activity label este reprezentat tot de un string, de data asta semnificand numele activitatii pe care persoana monitorizata a facut-o. Activitatile posibile intr-un astfel de fisier sunt: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming.

Un fisier de acest tip are ca date de intrare un format de genul:

----- -

------ ----- ----- --------- ----- ---------- ------ ------- -- ------------- -------------

# Obiective

## Obiectiv Principal:

Obiectivul principal pe care aceasta cerinta il urmareste este procesarea informatiilor primite in fisierul initial de activitati („Activities.txt”), iar pe baza lor realizarea anumitor cerinte (task-uri), ce convertesc si proceseaza aceste date de intrare.

Fiecare task va trebuie sa genereze la finalizarea acestuia un fisier in format text (.txt) in care sa scrie datele procesate si analizate, in functie de cerintele fiecarui task in parte.

Obiectivul principal, in acest caz, ar fi respectarea task-urilor, prin modul lor de proceasare a informatiei, cat si prin datele de iesire pe care fiecare clasa trebuie sa le genereze, in urma procesarii acestor date de intrare. Pentru ca acest lucru sa fie posibil, fiecare task are cate o cerinta separata:

TASK1 : se cere realizarea unei clase java ce are numele MonitoredData. Aceasta clasa trebuie sa contine 3 campuri pentru date: start time, end time si activity, acestea fiind primite si procesate ca string-uri. Apoi, datele din fisierul de activitati („Activities.txt”) trebuie citite, linie cu linie, iar fiecare linie trebuie divizata in 3 parti: start\_time, end\_time si activity\_label, cu ajutorul carora se va crea o lista de obiecte instanta a clasei MonitoredData.

TASK2 : pentru toate datele de intrare, se cere numararea tuturor zilelor distincte ce apar in perioada de monitorizare a activitatilor persoanei

TASK3 : se cere numararea fiecarei aparitii pentru activitatile distincte intalnite in fisierul de input. Asadar, pentru fiecare activitate (trebuie parcurs fisierul si delimitate activitatile) se va crea o tabela de dispersie (vezi 4.1 Structuri de date) in care aceasta este cheie, iar pentru fiecare cheie va trebui salvat un numar intreg, ce ii reprezinta numarul de apritii, in acest caz, pentru fiecare activitate distincta in parte.

TASK4 : pentru fiecare zi a perioadei de monitorizare, trebuie sa numaram de cate ori, persoana in cauza, a realizat fiecare activitate, insa acest lucru trebuie delimitat pe zile de monitorizare. Asadar, o structura (vezi 4.1) de tip tablela de dispersie (Map) o sa fie utilizata unde: cheia pentru dispersia datelor o reprezinta fiecare zi in parte, iar datele pentru zile sunt, exact ca la task-ul precedent, o tabela unde se tine cheie activitatea si apoi numarul aparitii pentru ea.

TASK5 : pentru fiecare activitate trebuie determinata durata totala a realizarii ei, pentru intreaga perioada de monitorizare. Acest lucru o sa fie salvat intr-o noua structura de tip tabela de dispersie, unde ca si cheie se va salva activitatea, iar datele pentru fiecare activitate in parte (deoarece cheile unei tabele sunt unice) o sa fie obiecte de tip LocalTime, ce reprezinta intreaga durata a activitatii, pentru toata perioada de monitorizare.

TASK6 : activitatile transmise trebuie parsate si filtrate astfel incat: pentru activitatile care au mai mult de 90% din timpii de realizare sub durata de 5 minute, ele trebuie colectate intr-o structura de tip List<String> (vezi 4.1 pentru informatii aditionale), ea continand doar numele activitatilor (distincte -> fieacare activitate trebuie sa apara o singura data).

## Obiective Secundare:

Pentru realizarea task-urilor, cat si a parsarii input-ului din fisierul text, sau a scrierii rezultatelor pentru fiecare task in parte, va fi necesar sa se foloseasca stream-uri si expresii lambda (pentru detalii -> sectiunea de implementare a proiectului, cati si cea de structuri de date folosite).

Pe langa rularea si compilarea normala a programului implementat, folosind IDE-uri, acesta trebuie sa permita si rularea dintr-un terminal de comanda, cu ajutorul unui fisier .jar, folosind comanda:

**java -jar PT2020\_Group\_LastName\_FirstName\_Assignment\_5.jar**

Pentru proiectarea solutiei cerute trebuie respectate si paradigmele proiectarii orientate pe obiecte. Acest lucru este intarit de respectarea modelelor arhitecturale, fapt ce doreste organizarea proiectului in mai multe pachete (o organizare te tip Layered), astfel incat fiecare pachet contine anumite clase ce au ca scop si functionalitate un obiectiv comun (de exemplu, pentru un pachet de procesare de date, clasele din acesta sunt cele responsabile pentru scrierea in fisiere si citirea din acestea).

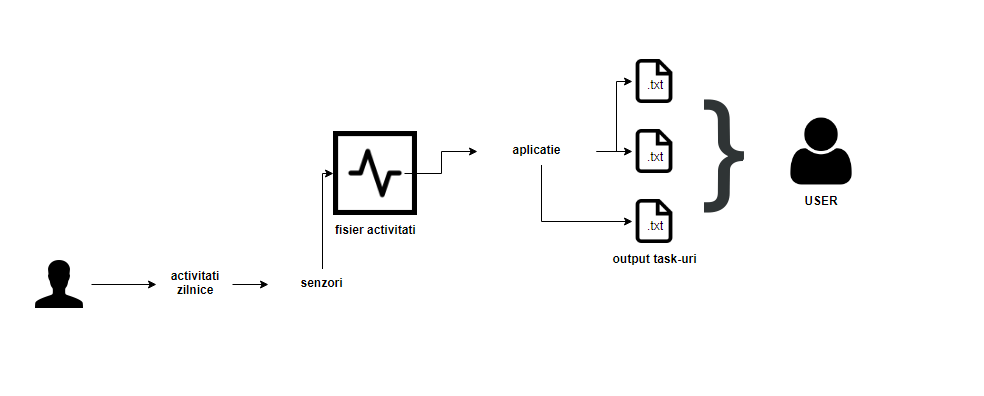
Leagat de implementarea in sine, clasele trebuie sa aiba dimensiuni sub 300 de linii de cod, iar metodele nu trebuie sa depaseasca 30 de linii, fapt mentionat in specificatiile minime ale temei.

# Analiza problemei

## Use case-uri si scenarii:

Interactiunea dintre utilizator si acest program este putin diferita de cea normala, in care, spre exemplu, am avea o fereastra grafica, cu butoane sau zone de text, unde utilizatorul poate sa introduca datele dorite si sa primeasca un feedback vizual imediat din partea aplicatiei implementate. Pentru acest proiect, utilizatorul se va folosi exclusiv de fisierul de activitati, in care a fost monitorizata persoana de test prin intermediul senzorilor.

Desigur, cat timp formatul prezentat pentru citirea si parsarea datelor este valid si respectat de utilizator, acesta poate introduce si propriul fisiser de activitati. Feedback-ul primit de la aplicatie consta in mare in raspunsul fiecarui task implementat la datele de intrate, acela fiind prin intermediul respectivelor fisiere .txt unde se vor scrie datele.



# Proiectare (etape de proiectare):

## Structuri de date:

Structurile de date folosite pentru aceasta aplicatie au ca scop modelarea datelor primite ca intrare, pentru ca datele si formatul de iesire al lor sa fie cel dorit initial, specificat in pct. 1.

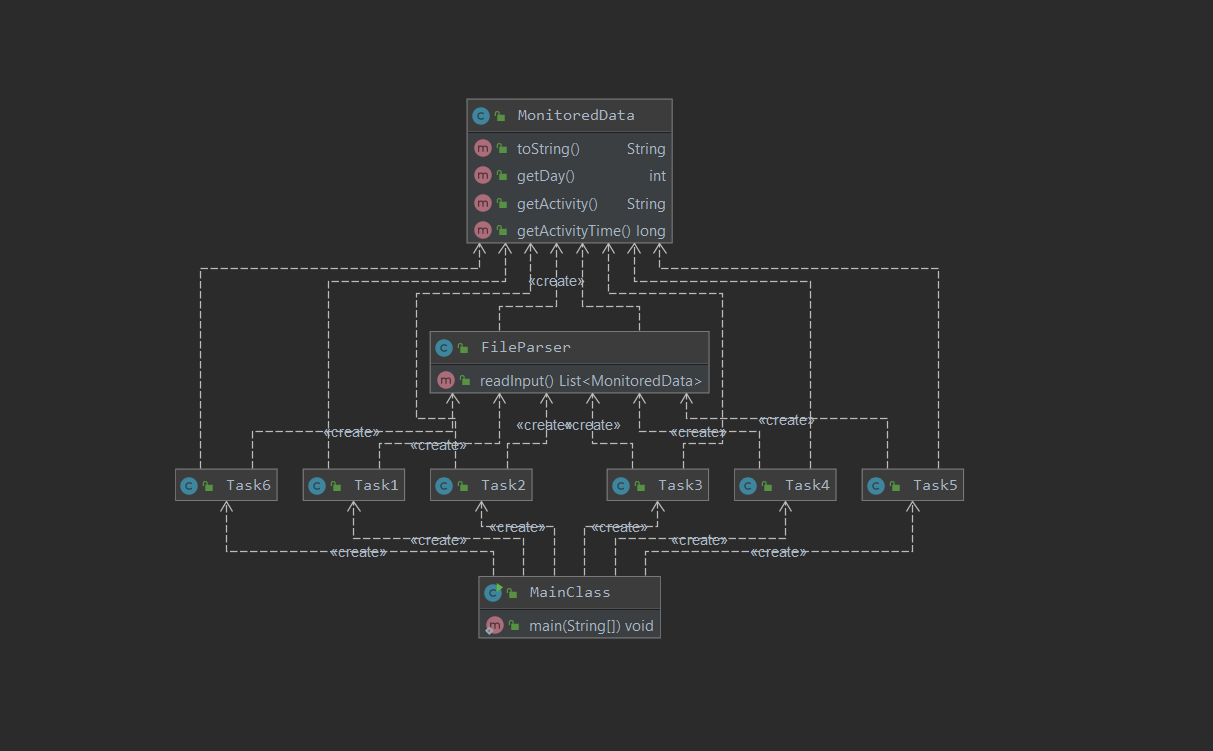
Map:

* Se foloseste in task-ul cu numarul 3 o structura de tip Map<String,Integer>. Acest lucru permite, din informatiile cu fisierul txt, sa numaram pentru fiecare activitate numarul de aparitii pe care aceasta le are. Folosind o structura de tip map, unde cheia fiecarul element din aceasta o constituie numele activitatii, putem asigura faptul ca o activitate nu va aparea de 2 ori, datorita unicitatii cheii intr-o tabela de dispersie de acest tip (proprietate a structurii Map).
* Pentru al doilea task se foloseste o structura noua de tip Map, ce contine si o structura asemanatoare cu tabela precedenta: Map<Integer,Map<String,Integer>>.
* Pentru task-ul cu numarul 5 se foloseste Map<String,LocalTime> pentru a determina timpii pentru fiecare tip de activitate in parte

List:

* Pentru ultimul task se foloseste o structura de tip lista, pentru a colecta toate informatiile necesare, transmise in cerinta functionala a task-ului (la pct. 5 -> implementare, o sa fie preciazate aceste lucruri), ele urmand, ca la orice task, sa fie transmise utilizatorului prin fisiere text.

## Diagrama de clase (UML):



# Implementare:

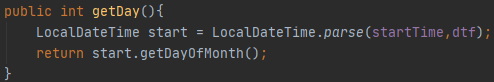
Pentru implemetare, proiectul a fost impartit in pachete, conform modelului arhitectural Layer. Asadar, proiectul este impartit in 4 pachete: BusinessLayer, Executer, FileProcessor, TaskLogic. Fiecare pachet contine clase ce ajuta la realizarea finala a proiectului.

**BusinessLayer**

Contine clasa MonitoredData, ce se doreste a fi generata la task-ul 1. Aceasta are ca variabile cele 2 date de inceput si respectiv final pentru activitati, cat si numele activitatii respective:

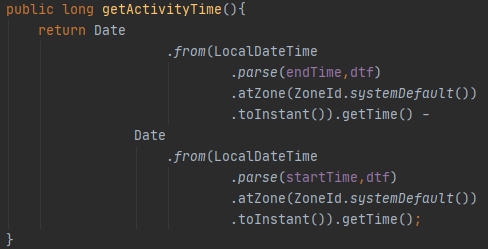


Metoda getDay() returneaza ziua in care activitatea este desfasurata, folosind un format, transmis printr-un obiect de tip DateTimeFormatter.



Pattern-ul pentru formatare este:

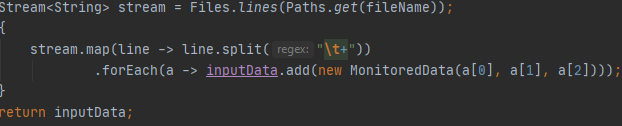


Metoda getActivityTime, ce foloseste expresii de tip lambda, returneaza , pentru activitatea curenta, timpul total al acesteia, rezultatul fiind transmis printr-un numar de tip long, fiind returnat in realitate timpul in milisecunde.

Acest lucru este realizat scazand din timpul de final al activitatii, timpul de start al acesteia (dupa ce este de altefel convertit, cu aceeasi regula de parsare).

**FileProcessor**

Contine clasa FileParser, care, asa cum ii spune si numele, se ocupa de parsarea fisierelor. In aceasta clasa se citeste input-ul primit din fisierul „Activities.txt” si se transmite mai departe o lista cu elementele din acesta, ce va putea fi folosita ulterior de restul task-urilor.



**TaskLogic**

Acest pachet contine toate clasele responsabile pentru task-uri. Decizia de implementare a fost ca fiecare task sa aiba o clasa proprie asociata, in care sa se rezolve cerinta primita (vezi 2.1 Obiective principale pentru detalii).

Fiecare clasa de tip Task va avea asociat un nume de fisier.txt si o lista pe care o primeste din clasa de parsare de fisier, conform exemplului de mai jos:



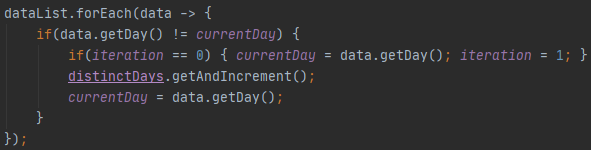
1. Task 1:

Deoarece la acest task trebuie doar sa iteram prin fisierul de intrare si sa parsam datele, aceasta clasa are mai mult rol de generearea output-ului. Deoarece in momentul in care parsam fisierul, generam si obiecte de tip MonitoredData, aici nu vom mai face acest lucru, ci doar iteram prin lista de obiecte si le afisam in fisierul corespunzator: task\_1.txt .



1. Task 2:

Pentru acest task trebuie sa numaram zilele diferite ce apar in fisier. Asadar, vom itera prin colectia transmisa de calsa de parsare si, cu ajutorul unei variabile auxiliare verificam daca am dat peste o zi diferita. Deoarece stim ca in fisierul de intrare nu putem avea intr-o ordine intamplatoare perioadele, ci doar cronologic (nu puteam avea luna x ziua 3, luna x ziua 4 apoi luna x ziua 2), vom folosi o variabila pentru a verifica daca, la fiecare citire de linie noua, ziua s-a schimbat. Daca da, variabila distinctDays va fi incrementata. Pentru ca folosim expresii lambda, trebuie sa asiguram o buna incrementare (si safe) pentru aceasta variabila, fapt pentru care am facut-o de tip AtomicInteger. Initial, valoarea ei este 0.



Pentru a nu numara ziua de inceput de mai multe ori si a avea grija ca noua zi si fosta zi isi schimba valorile, am folosit variabila iteration pentru a asigura ca prima data cand intram in aceasta bucla se executa corect modificarea zilei curente.

Desigur, ca la fiecare task in parte, rezultatul acestei cerinte o sa fie scrisa in fisierul task\_2.txt (variabila distinctDays).

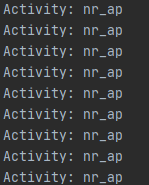
1. Task 3:

La al treilea task se doreste sa numaram de cate ori am intalnit fiecare activitate, pe parcursul intregii perioade de monitorizare.

Acest lucru este realizat folosind o structura de tip Map<String,Integer> unde cheia (String) este reprezentata de numele activitatii, iar valoarea (Integer) de numarul de aparitii. Folosind expresii lambda, am apelat din clasa Collectors metoda summingInt pentru a incrementa valoarea numarului de aparitii, de fiecare data cand dadeam peste o activitate. Faptul ca am folosit Map asigura unicitatea fiecarui element, deoarece intr-o tabela de dispersie, cheile trebuie sa fie unice.



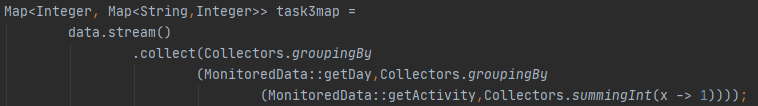
Ca la restul exercitiilor, rezultatul este scris in fisierul task\_3.txt, ce asteapta un output de forma:



1. Task 4:

Acest task este asemanator cu cel precedent (task-ul 3), insa de aceasta data se doreste ca aparitiile sa nu fie numarate in intreaga perioada de monitorizare, ci pentru fiecare zi in pare.

Pentru implementarea ei se foloseste exact acelasi procedeu ca la punctul 3, insa structura Map va avea ca elemente un intreg (Integer) si o alta tabela, care este cea precedenta. Asadar, se numara (utilizant tot summingInt) doar pentru noua cheie, ce reprezinta ziua curenta.

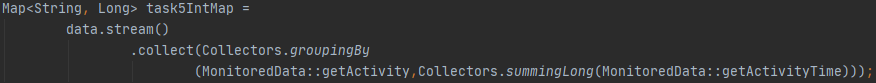


In fisierul text rezultat, vom putea observa un output de forma:

Day <nr\_zi>: {Activitate1=<nr\_aparitii>,..., ActivitateX=<nr\_aparitii>}

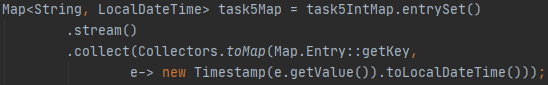
1. Task 5:

Pentru acest task se doreste calculare, pentru fiecare activitate in parte, a timpului total din momentul in care a inceput monitorizarea. Pentru acest lucru se utilizeaza metoda din clasa MonitorData (getActivityTime), ce returneaza pentru activitati timpul lor.



Intr-o prima faza (imaginea de sus) se genereaza o tabela ce contine ca si chei numele activitatii (String) si ca valori timpii pentru ficare activitate (Long). Pentru a insuma pentru fiecare aparitie a activitatii timpii, se foloseste metoda summingLong, apelata cu parametru metoda getActivityTime din clasa MonitorData.

Apoi, elementele din aceasta tabela sunt copiate in alta, unde timpul in milisecunde generat anterior este transformat in variabila de tip LocalDateTime, pentru o afisare mai exacta, dar si pentru respectarea cerintei functionale, ce preciza ca se doreste un astfel de obiect transmis ca output.



Toate elementele din aceasta colectie (task5Map) sunt in forma finala si afisate in fisierul text corespunzator, al carui output este de forma (la afisare, elementele sunt scrise dupa un format):

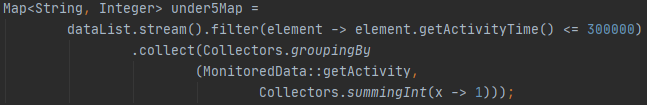
Activitate: <ore>:<minute>:<secunde>

1. Task 6:

Pentru acest task se doreste afisarea acelor activitati care au peste 90% din timpii de desfasurare mai mici de 5 minute. Pentru acest lucru s-au folosit 2 colectii intermediare, si o a treia colectie, finala, care respecta modul de transmitere al datelor, dorit de cerinta functionala (o structura de tip List<String>).

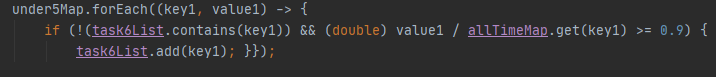


allTimeMap este in esenta structura folosita la task-ul 3, refacuta, ce tine minte pentru fiecare activitate numarul total de apraritii, pe durata de monitorizare.

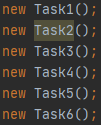


Under5Map, asa cum sugereaza si numele, este o structura ce retine pentru fiecare activitate numarul de aparitii (la fel ca si ceea de mai sus, respectiv ce a de la task3), insa de data aceasta incrementarea aparitiilor se face doar in cazul in care timpul activitatii este mai mic de 5 minute (sau egal) -> 300 000 fiind echivalentul in milisecunde.

Avand aceste doua colectii intermediare, urmeaza sa se itereze prin ele, simultan, iar numarul de aparitii din colectia de sub 5 minute este impartit la cel din colectia de peste 5 minute. Daca acest rezultat este mai mare (sau egal) de 0.9, fapt ce inseamna peste 90% din timpi sub 5 minute pentru activitatea respectiva, si nu a fost parcursa inca activitatea, ea este adaugata in lista finala (initializata la inceputul clasei) -> task6List.



In urma calculelor efectuate, in task6List avem toate activitatile ce respecta cerinta prezentata task-ului si urmeaza sa fie scrise in fisierul text corespunzator (task\_6.txt).

**Executer**

Acest pachet contine doar clasa MainClass, clasa cu metoda main, in care sunt instantiate doar task-urile, nefiind atribuite obiectelor, ele neavand metode ce trebuie apelate din exterior, totul realizandu-se prin constructori.

# Concluzii si dezvoltare ulterioara:

Acest proiect as putea sa spun ca a fost pe atat de provocator, cat a fost interesant. A constituit de altfel prima tangenta cu expresiile de tip lambda si stream, folosite pentru un proiect de dimensiune si complexitate mai mare, fapt ce pot spune ca mi-a pus anumite dificultati, nu majore insa. Pentru partea de dezvoltare ulterioara, cu siguranta o asezate mai eficienta, mai buna a claselor putea fi realizata, cat si comprimarea unor clase si metode din toate cele utilizate, pentru a avea cat mai putine linii de cod scris, si eventual cat mai multe expresii lambda.

# Bibliografie:

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html> pentru partea de expresii lambda

<https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html> pentru procesarea datelor cu streas

<https://howtodoinjava.com/java/date-time/java-date-examples> pentru toate obiectele si datele de tip Data, LocalTime, etc.

<https://stackoverflow.com/>, <https://www.geeksforgeeks.org/> pentru orice alte nelamuriri si probleme de proiectare aparute pe parcurs