DOCUMENTATIE TEMA 5

ACTIVITATILE ZILNICE ALE UNEI PERSOANE

Nume si prenume: Fagadar Ionela-Catalina

Grupa: 30226

Profesor Laborator: Marin Oana

Contents

[**1.Obiectivul temei** 5](#_Toc8218281)

[**2.Analiza problemei, modelare, scenarii, cazuri de utilizare** 5](#_Toc8218282)

[**3.Proiectare** 6](#_Toc8218283)

[**3.1. Decizii de proiectare** 6](#_Toc8218284)

[**3.2. Diagrame UML** 7](#_Toc8218285)

[**3.3. Structuri de date** 8](#_Toc8218287)

[**3.4. Proiectare clase** 8](#_Toc8218288)

[**3.5. Proiectare interfete** 9](#_Toc8218289)

[**3.6. Relatii** 9](#_Toc8218290)

[**3.7. Pachete** 9](#_Toc8218291)

[**3.8. Algoritmi** 9](#_Toc8218292)

[**3.8.0. Algoritmul de deserializare** 9](#_Toc8218293)

[**3.8.1. Algoritmul de producere a facturii** 10](#_Toc8218294)

[**3.8.2. Algoritmul de interogare a bazei de date** 10](#_Toc8218295)

[**4. Implementare** 11](#_Toc8218296)

[**4.1. Clasa MenuItem – clasa Abstracta** 11](#_Toc8218297)

[**4.2. Clasa BaseProduct** 11](#_Toc8218298)

[**4.3. Clasa CompositeProduct** 11](#_Toc8218299)

[**4.4. Clasa Order** 11](#_Toc8218300)

[**4.5. Clasa Restaurant** 12](#_Toc8218301)

[**4.6. Clasa RestaurantSerializator** 12](#_Toc8218302)

[**4.7. Clasa Controller, View, AdministratorGraphicalUserInterface, BaseGUI, CompositeGUI, ChefGraphicalUserInterface, WaiterGraphicalUserInterface, ViewJTable** 12](#_Toc8218303)

[**4.8. Clasa Main** 12](#_Toc8218304)

[**5.Rezultate** 12](#_Toc8218305)

[**6.Concluzii si dezvoltari ulterioare** 12](#_Toc8218306)

[**7.Bibliografie** 13](#_Toc8218307)

# **1****.Obiectivul temei**

Considera sarcina de analizare a comportamentului unei persoane printr-un set de senzori. Istoricul de activitati ale persoanei e stocat a tuple(timpul de inceput, timpul de sfarsit, activitatea), unde timpul de inceput si timpul de sfarsit reprezinta data si timpul cand fiecare activitate a inceput sau s-a terminat, in timp ce activitatea reprezsinta tipul de activitate desfasurata de catre persoana: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming. Datele sunt efectuate pe mai multe zile, ca intrari in fiesierul Activities.txt.

|  |  |  |
| --- | --- | --- |
| Obiectiv | Descriere | Capitol |
| dezvoltarea de use case-uri si scenarii |  | 2 |
| alegerea structurilor de date |  | 3 |
| impartirea pe clase |  | 3 |
| dezvoltarea algoritmilor |  | 3 |
| implementarea solutiei |  | 4 |
| testare |  | 5 |

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

Programul este conceput astfel incat sa se extraga datele corect din fisierul Activities.txt, dupa care, sa se faca operatii pe acesta, putand sa se afle catre zile apar in fisierul de monitorizare a datelor, de cate ori a aparut fiecare activitate pe intreaga perioada de monitorizare, de cate ori a aparut fiecare activitate pentru fiecare zi pe intreaga perioada de monitorizare, pentru fiecare activitate aparuta in fisierul de monitorizare a datelor, durata fiecarei activitati aparute, pentru fiecare activitate, durata totata pe toata durata de monitorizare, cat si activitatile ce ocupa 90% din intregul fisier de monitorizare a datelor, cu o durata mai mica decat 5 minute.

# **3.Proiectare**

## **3.1. Decizii de proiectare**

Am ales sa proiectez acest proiect ajutandu-ma de 2 clase, ordonate pe 2 pachete, cu numele: Main si MonitoredData. Main este clasa ce este folosita pentru rularea aplicatiei, iar MonitoredData este folosita pentru operatiile pe datele ce sunt monitorizate. Clasa Main contine doar apelul metodelor din clasa MonitoredData, cat si constructorul pentru crearea unui obiect de tipul MonitoredData. Clasa MonitoredData contine 3 campuri: startTime, care reprezinta timpul de inceput al activitatii, endTime, care reprezinta timpul de final al activitatii, si un camp numit activity, care reprezinta activitatea ce s-a desfasurat de la startTime pana la endTime. Aceasta clasa contine metode ce ajuta la interogarea pe anumite criterii a datelor din fisierul Activities.txt.

## **3.2. Diagrame UML**

Diagrama de clase a acestui proiect:

Diagrama de pachete a acestui proiect:

## **3.3. Structuri de date**

Se folosesc doua structuri de date in acest proiect: Lista si HashTable. Lista este o structura de date folosita pentru preluarea activitatilor din fisierul dat, denumit „Activities.txt”, iar HashMap este structura de date folosita pentru a stoca fiecare activitate cu numarul de apartitii ale fiecarei activitati.

## **3.4. Proiectare clase**

Se folosesc doua clase ce apartin de pachetele cu acelasi nume: Main si MonitoredData. Clasa Main contine metoda main ce ajuta la pornirea aplicatiei. Clasa MonitoredData contine metode ce ajuta la interogarea datelor din fisierul „Activities.txt”. Metode ca: readActivities pentru a citi date din fisier si a le stoca intr-o lista, iar celelalte metode sunt folosite pentru interogarea datelor din fisierul „Activities.txt”, cat si pentru a indeplini cerintele cerute in tema acestei lucrari.

## **3.5. Proiectare interfete**

Nu se foloseste nicio interfata in acest proiect.

## **3.6. Relatii**

## **3.7. Pachete**

Proiectul e structurat in 2 pachete: Main si MonitoredData, care contine fiecare cate o clasa, cu acelasi nume: Main si MonitoredData.

## **3.8. Algoritmi**

Algoritmii utilizati pentru acest proiect sunt algoritmii utilizati pentru citirea activitatilor din fisier.

## **3.8.1. Algoritmul de citire a activitatilor**

Acest Algoritm citeste date dintr-un fisier, cu ajutorul streamurilor. Se citesc toate datele, folosind stream-uri, apoi se parseaza liniile din fisier obtinute, memorandu-se datele intr-o alta lista.

**public** List<MonitoredData> readActivities() {

String fileName = "Activities.txt";

List<String> list = **new** ArrayList<>();

List<MonitoredData> mdList = **new** ArrayList<MonitoredData>();

**try** (Stream<String> stream = Files.*lines*(Paths.*get*(fileName))) { // convert it into a List

list = stream.collect(Collectors.*toList*());

} **catch** (IOException e) {

e.printStackTrace();

}

List<String> listaParsata = **new** ArrayList<>();

**for**(**int** i=0;i<list.size();i++) {

listaParsata = Stream.*of*(list.get(i).split(" "))

.map(elem -> **new** String(elem))

.collect(Collectors.*toList*());

MonitoredData md;

**try** {

md = **new** MonitoredData(dateFormat.parse(listaParsata.get(0)),dateFormat.parse(listaParsata.get(1)),listaParsata.get(2));

mdList.add(md);

} **catch** (ParseException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**return** mdList;

}

# **4. Implementare**

## **4.1. Clasa Main – pachetul Main**

Metoda main ce porneste aplicatia.

## **4.2. Clasa MonitoredData – pachetul MonitoredData**

Atribute: startTime, endTime, activity.

Constructori, Getter, Setter, metoda toString(), readActivities(), howManyDays( List<MonitoredData>), howManyActivities( List<MonitoredData>), howManyActivitiesPerDay( List<MonitoredData>), duration( List<MonitoredData>), durataTotala( List<MonitoredData>), ex7( List<MonitoredData>).

Stream-urile sunt folosite in aceasta clasa cu precadere. Acestea sunt folosire pentru a procesa colectii de obiecte. Un stream este o secvente de obiecte ce suporta variatii de metode ce pot fi alaturate pentru a produce rezultatul dorit.

Avantajele acestora sunt:

- un Stream nu e o strucura de data, in schimb, mosteneste metode de la Collections, Arrays, si I/O Channels

- stream-urile nu produc schimbari la structura de date originala, ele doar produce rezultatul prin metodele pipeline.

- fiecare operatie intermediara e executa tarziu si returneaza un stream ca rezultat, prin variatii intermediare de operatii care pot fi pipelined. Operatiile terminale marcheaza sfarsitul streamurilor si returneaza rezultatul.

Operatii intermediare:

- map: metoda e folosita pentru a mapa itemii in colectiile altor obiecte in concordanta cu Predicatul pasat in argumentul functiei.

- filter: metoda filter e folosita pentru a selecta elementele cu ajutorul Predicatului pasat ca argument.

- sorted: metoda sorted e folosita pentru a sorta streamul.

Operatii terminale:

- collect: metoda collect este folosita pentru a returna rezultatul prin intermediul operatiilor intermediare efectuate pe stream.

- forEach: metoda forEach e folosita pentru a itera fiecare element din stre

- reduce: metoda reduce e folosita pentru a reduce elementele stream-ului la o singura valoare.

Un stream e consistenta dintr-o sursa urmarita de zerouri, sau mai multe metode intermediare combinate impreuna (pipelined) si o metoda terminala pentru a procesa obiecgtele obtinute de la sursa prin intermediul metodelor descrise.

Un stream e folosit pentru a compune elemente prin metodele de pipeline fara a altera originala valoare a acestui obiect.

Generarea stream-urilor se face prin doua metode:

* Stream() – returneaza o stream secvential considerand colectia ca sursa.
* parallelStream() – returneaza un stream paralel considerand stream-ul ca sursa.

|  |  |
| --- | --- |
| boolean allMatch(Predicate<? super T> predicate) | Returneaza toate elementele stream-ului ce indeplinesc contditia predicatul dat ca argument. Daca stream-ul e gol, atunci este returnata valoarea true, si predicatul nu e evaluat. |
| boolean anyMatch(Predicate<? super T> predicate) | Retruneaza orice element din stream ce indeplineste conditia data de predicat. Daca stream-ul e gol, atunci se returneaza o valoare false si predicatul nu e evaluat. |
| static <T> Stream.Builder<T> builder() | Returneaza un constructor pentru stream. |
| <R,A> R collect(Collector<? super T,A,R> collector) | Produce o operatie de reducere mutabila pe elementele stream-ului, folosind un collector. Un collector incapsuleaza functiile foloste ca argumente pentru a colecta( Supplier, BiConsumer, BiConsumer), dand voie sa se refoloseasca strategiile colectiei si compozitiile operatilor de colectare, ca gruparea pe multe nibele sau partitionarea. |
| <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) | Produce o operatie de reducere mutabila pe elementele stream-ului. O reducere mutabila e una in care valoarea de reducere e mutata intr-un container rezultat, ca un ArrayList, si elementele sunt incorporate prin actualizarea starii rezultatului, in loc sa se inlocuiasca. |
| static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) | Creează un flux concatenat ale cărui elemente sunt toate elementele primului flux, urmate de toate elementele celui de-al doilea flux. Fluxul rezultat este comandat în cazul în care ambele fluxuri de intrare sunt comandate, și paralel în cazul în care oricare dintre fluxurile de intrare este paralel. Când fluxul rezultat este închis, se invocă rutine apropiate pentru ambele fluxuri de intrare. |
| long count() | Se returnează numărul de elemente în acest flux. Acesta este un caz special de reducere. |
| Stream<T> distinct() | Returnează un flux constând din elemente distincte (în conformitate cu Object. egal (Object)) din acest flux. |
| static <T> Stream<T> empty() | Returnează un flux secvențial gol. |
| Stream<T> filter(Predicate<? super T> predicate) | Acesta returnează un flux constând din elementele acestui flux care se potrivesc cu predicatul dat. |
| Optional<T> findAny() | Returnează un opțional care descrie un element al streamului sau un gol opțional dacă fluxul este gol. |
| Optional<T> findFirst() | Returnează un Optional care descrie primul element al acestui flux sau un empty Optional dacă fluxul este gol. Dacă fluxul nu are ordine de întâlnire, atunci orice element poate fi returnat. |
| <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | Returnează un flux constând din rezultatele înlocuirii fiecărui element al acestui flux cu conținutul unui flux mapat produs prin aplicarea funcției de cartografiere furnizată fiecărui element. Fiecare flux mapat este închis după ce conținutul său a fost plasat în acest flux. (Dacă un flux mapat este nul este utilizat un flux gol, în schimb.) |
| DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) | Returnează un DoubleStream constând din rezultatele înlocuirii fiecărui element al acestui flux cu conținutul unui flux mapat produs prin aplicarea funcției de cartografiere furnizată fiecărui element. Fiecare flux mapat este închis după ce conținutul său a fost plasat în acest flux. (Dacă un flux mapat este nul este utilizat un flux gol, în schimb.) |
| IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper) | Returnează un IntStream constând din rezultatele înlocuirii fiecărui element al acestui flux cu conținutul unui flux mapat produs prin aplicarea funcției de cartografiere furnizată fiecărui element. Fiecare flux mapat este închis după ce conținutul său a fost plasat în acest flux. (Dacă un flux mapat este nul este utilizat un flux gol, în schimb.) |
| LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper) | Returnează un LongStream constând din rezultatele înlocuirii fiecărui element al acestui flux cu conținutul unui flux mapat produs prin aplicarea funcției de cartografiere furnizată fiecărui element. Fiecare flux mapat este închis după ce conținutul său a fost plasat în acest flux. (Dacă un flux mapat este nul este utilizat un flux gol, în schimb.) |
| void forEach(Consumer<? super T> action) | Se efectuează o acțiune pentru fiecare element al acestui flux. |
| void forEachOrdered(Consumer<? super T> action) | Efectuează o acțiune pentru fiecare element al acestui flux, în ordinea întâlniți a fluxului dacă fluxul are o ordine de întâlnire definită. |
| static <T> Stream<T> generate(Supplier<T> s) | Returnează un flux infinit secvențial neordonat unde fiecare element este generat de furnizorul furnizat. Acest lucru este potrivit pentru generarea fluxurilor constante, fluxuri de elemente aleatorii, etc. |
| static <T> Stream<T> iterate(T seed,UnaryOperator<T> f) | Acesta returnează un infinit secvențial ordonat Stream produs prin aplicarea iterativă a unei funcții f la un element inițial semințe, producătoare de un stream constând din seed, f (seed), f (f (seed)), etc. |
| Stream<T> limit(long maxSize) | Se returnează un flux constând din elementele acestui flux, trunchiate pentru a fi nu mai mult de maxSize în lungime. |
| <R> Stream<R> map(Function<? super T,? extends R> mapper) | Acesta returnează un flux constând din rezultatele aplicării funcției date la elementele acestui flux. |
| DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) | Acesta returnează un DoubleStream constând din rezultatele aplicării funcției date la elementele acestui flux. |
| IntStream mapToInt(ToIntFunction<? super T> mapper) | Acesta returnează un IntStream constând din rezultatele aplicării funcției date la elementele acestui flux. |
| LongStream mapToLong(ToLongFunction<? super T> mapper) | Acesta returnează un LongStream constând din rezultatele aplicării funcției date la elementele acestui flux. |
| Optional<T> max(Comparator<? super T> comparator) | Acesta returnează elementul maxim al acestui flux în conformitate cu comparator furnizate. Acesta este un caz special de reducere. |
| Optional<T> min(Comparator<? super T> comparator) | Acesta returnează elementul minim al acestui flux în conformitate cu comparator furnizate. Acesta este un caz special de reducere. |
| boolean noneMatch(Predicate<? super T> predicate) | Se returnează elemente din acest flux se potrivesc Predicatul furnizat. Dacă fluxul este gol atunci True este returnat și predicatul nu este evaluată. |
| @SafeVarargs static <T> Stream<T> of(T... values) | Returnează un flux ordonat secvențial ale cărui elemente sunt valorile specificate. |
| static <T> Stream<T> of(T t) | Returnează un flux secvențial care conține un singur element. |
| Stream<T> peek(Consumer<? super T> action) | Acesta returnează un flux constând din elementele acestui flux, în plus, efectuarea acțiunii furnizate pe fiecare element ca elemente sunt consumate din fluxul rezultat. |
| Optional<T> reduce(BinaryOperator<T> accumulator) | Se efectuează o reducere a elementelor acestui flux, folosind o funcție de acumulare asociativă și returnează un opțional care descrie valoarea redusă, dacă este cazul. |
| T reduce(T identity, BinaryOperator<T> accumulator) | Acesta efectuează o reducere a elementelor acestui flux, folosind valoarea de identitate furnizată și o funcție de acumulare asociativă și returnează valoarea redusă. |
| <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) | Acesta efectuează o reducere a elementelor acestui flux, folosind identitatea furnizată, acumularea și combinarea funcțiilor. |
| Stream<T> skip(long n) | Acesta returnează un flux constând din elementele rămase ale acestui flux după aruncarea în prim-elemente n a fluxului. Dacă acest flux conține mai puțin de n elemente, atunci va fi returnat un flux gol. |
| Stream<T> sorted() | Acesta returnează un flux constând din elementele acestui flux, sortate în funcție de ordinea naturală. Dacă elementele acestui flux nu sunt comparabile, o Java. lang. ClassCastException pot fi aruncate atunci când se execută operațiunea terminal. |
| Stream<T> sorted(Comparator<? super T> comparator) | Acesta returnează un flux constând din elementele acestui flux, sortate în conformitate cu comparator furnizate. |
| Object[] toArray() | Returnează o matrice care conține elementele acestui flux. |
| <A> A[] toArray(IntFunction<A[]> generator) | Se returnează o matrice care conține elementele acestui flux, folosind funcția generator furnizat pentru a aloca matrice returnate, precum și orice matrice suplimentare care ar putea fi necesare pentru o execuție partiționat sau pentru redimensionare. |

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. Pentru a înţelege modul în care se lucrează cu lambda expresii am construit un mic exemplu prin care am creat colecţii de obiecte sortate după diverse criterii. Implementarea interfeţei Comparator a fost făcută într-o clasă anonimă, folosind lambda expresii. Implementarea cu lambda expresii a fost posibilă pentru că în versiunea 8 Comparator este anotată cu @FunctionalInterface.

# **5.Rezultate**

Aceasta aplicatie descrie o solutie optima de a prelua datele personale ale unei persoane, furnizate de senzori.

# **6.Concluzii si dezvoltari ulterioare**

In concluzie, consider ca acest program preia datele furnizate de un senzor.

# **7.Bibliografie**

* <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW5_Tema5/HW5_Tema5.pdf>
* https://www.mkyong.com/java8/java-8-stream-read-a-file-line-by-line/