**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

**DEPARTAMENTUL CALCULATOARE**

**Tema 5**

la disciplina

**Tehnici de programare**

**Titlu**

***„Analyzing a person's behavior”***

Nume si prenume : Trif Gheorghe Andrei

Grupa: 30225

An academic : 2018-2019

**1.Obiectivul temei**

Propuneti, proiectati si implementati un sistem de monitorizare al ativitatilor facute de o anumita persoana.

|  |  |  |
| --- | --- | --- |
| Obiective secundare | Descriere | Capitolul |
| Dezvoltare de use case-urii si scenarii in legatura cu intelegera temei propuse. | Se va dezvolta un comportament pe care programul va incerca sa le urmeze. | 1 |
| Analiza problemei si modelarea acesteia | Se va prezenta proiectarea OOP a problemei ,diagrame UML ,pachete si structuri de date folosite. | 2 |
| Dezvoltarea algoritmilor | Se va perzenta operatile alese  si descrierea lor. | 3 |
| Impelmentarea solutiei | Se va prezenta codul sursa aferent proiectari. | 3 |
| Testare aplicatiei | Prezentarea ca de cazuri favorabile de rulare si modul de utilizare ,dar si anumite defecte. | 4 |
| Dezvoltare ulterioara | Noi ideii de a inbunatati proiectul existent. | 5 |

Scopul acestui proiect este de a implementa un sistem de monitorizare al activitatilor unui unei persoane si de a descrie modul prin care acest sistem de monitorizare poate fi creat si manipulat cu ajutorul paradigmelor OOP .

Activitatile sunt de forma ora de start ora de final si nume , astfel monitorizarea unei persoane facunduse foarte amanuntit.Perioada de monitorizare intinzandu-se pe o durata foarte lunga ( minim 10 zile ) . Acest sistem ne ajuta sa ne gestionam timpul mai eficient deoarece pe o perioada intinsa putem observa unde pierdem timp si ce activitati ocupa o buna majoritate a timpului nostru . Astfel cu ajutorul paradigmelor oop putem impelementa un sistem foarte precis si care sa ne permita sa observam comportamentul persoanei .

Fiecare activitate va fi detaliata avand ora si data de start precum ora si data de final astfel cunoscandu-se o durata foarte precisa pentru fiecare activitate .

2.Analiza problemei si modelarea acestei

2.1.Analiza problemei

În programare oactivitate poate fi privita ca o combinatie de informatie care are atat data si ora de start cat si de final .

Aceste tipuri de informatie pot fi stocate foarte usor si incapsulate impreuna cu ajutorul expresiilor lamda si a streamurilor , care sunt indicat petntru acest aspect de lucru cu o varietate foarte mare de informatie precum si la filtrarea lor .

Expresiile Lambda exprimă în esență cazuri de interfețe funcționale (o interfață cu o singură metodă abstractă se numește interfață funcțională. Un exemplu este java.lang.Runnable). expresiile lambda implementează singura funcție abstractă și, prin urmare, implementează interfețe funcționale .

Expresiile Lambda sunt adăugate în Java 8 și oferă funcționalități de mai jos :

1. Activate pentru a trata funcționalitatea ca argument de metodă sau cod ca date.

2. O funcție care poate fi creată fără să aparțină vreunei clase.

3. O expresie lambda poate fi transmisă ca și cum ar fi un obiect și executată la cerere.

Introdus în Java 8, API-ul Stream este folosit pentru a procesa colecții de obiecte. Un flux este o secvență de obiecte care acceptă diverse metode care pot fi configurate pentru a produce rezultatul dorit.

Caracteristicile fluxului Java sunt:

1. Un flux nu este o structură de date, ci o intrare din colecții , arhiteuri sau canale .
2. Fluxurile nu modifică structura de date originală, ci furnizează rezultatul numai conform metodelor configurate.
3. Fiecare operație intermediară este executată leneș și, prin urmare, returnează un flux, prin urmare pot fi realizate diferite operații intermediare. Operațiile terminale marchează sfârșitul fluxului și returnează rezultatul.

Prin analiza problemei, ne referim la un prim set abstract de operații și proprietăți prin care încercăm să depistăm eventualele însușiri și comportamente ale proceselor necunoscute. Programarea orientată pe obiect ne oferă aici un avantaj clar, tocmai fiindcă ea permite să tratăm problema de la un nivel superiror, fără a mai fi constrâși, într-o așa măsură, de caracteristicile tehnice.

Un model al informaţiilor este o reprezentare abstractă, formală a entităţilor cu ajutorul căreia se precizează proprietăţile, relaţiile şi operaţiile ce pot fi efectuate asupra acestora. Entităţile care se modelează pot proveni din lumea reală, cum ar fi dispozitivele existente în cadrul unei reţele, sau pot să fie le însele abstracte, cum ar fi entităţile utilizate în cadrul sistemelor de facturare.

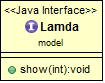
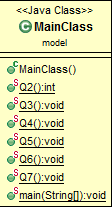
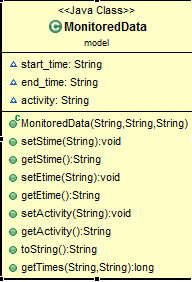
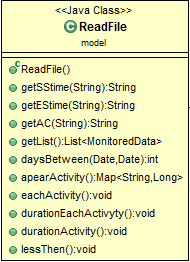
Scopul principal care se ascunde în spatele acestui concept este acela de a formaliza descrierea domeniului unei probleme fără a avea în vedere constrângerile prin care descrierea respectivă este afectată la momentul transformării acesteia în cadrul implementării .

Modelarea este un aspect deosebit de important, deoarece trebuie luate în considerare aspecte ce trebuie avute în vedere în cazul eventualelor modificări ulterioare, fără a afecta în mod semnificativ utilizarea produsului. Modelarea permite compatibilizarea cu modelele anterioare şi prevede soluţii pentru extensiile viitoare.

Colecțiile și fluxurile, având în același timp niște asemănări superficiale, au obiective diferite. Colecțiile se referă în primul rând la gestionarea eficientă și accesul la elementele lor. Dimpotrivă, fluxurile nu oferă un mijloc de acces direct sau de manipulare a elementelor lor și sunt în schimb preocupate de descrierea declarativă a operațiilor computaționale care vor fi efectuate în totalitate pe acea sursă. În consecință, fluxurile diferă de colecții în mai multe moduri:

1. Fără depozitare. Fluxurile nu au stocare pentru valori; acestea conțin valori dintr-o sursă (care ar putea fi o structură de date, o funcție de generare, un canal de intrare / ieșire, etc) printr-o conductă de etape de calcul.
2. Funcțional în natură. Operațiile pe un flux produc un rezultat, dar nu modifică sursa de date subiacentă.
3. Multe operațiuni de flux, cum ar fi filtrarea, cartografierea, sortarea sau eliminarea duplicat) pot fi implementate leneș. Acest lucru facilitează executarea eficientă a unei singure treceri a conductelor întregi, precum și facilitarea implementării eficiente a operațiunilor de scurtcircuit.
4. Există multe probleme care pot fi exprimate ca fluxuri infinite, permițând clienților să consume valori până când sunt satisfăcuți.

2.2.Proiectarea claselor

Sistemul de analiza al activitatilor este alcatuit doar din 3 clase :

Clasa Main aici sunt activate metodele de la un anumit task , functioneaza ca un meniu unde alegem ce task sa folosim.

Clasa MonetoredData este clasa de baza a unei activitati aici ia nastere sub forma de obiect o anumita activitate de forma start\_time, end\_time si numele activitati.

Clasa ReadFile in aceasta clasa se construieste lista tuturor activitatilor si se rezolva fiecare task intr-o anumita metoda .Citirea activitatilor se face dintrun fisier text in care activitatile sunt ordonate.

3.Impementarea Claselor

3.1. Clasa Main

Aceasta clasa are roul de a rula metoda fiecurui task in parte .

Lamda obj=(i)->{System.***out***.println("In monitor apar "+i+" zile!!");};

**int** i=*Q2*();

obj.show(i);

Q3();

Q4();

Q5();

Q6();

Q7();

In principal este o calasa banala care ajuta doar la rularea aplicatiei.

3.2.Clasa MonitoredData

Aceasta clasa ajuta la definirea unei activitati ca obiect aici ia nastere propriu zis fiecare activitate cu ajutorul constructorului.

String start\_time;

String end\_time;

String activity;

**public** MonitoredData(String t1,String t2,String act) {

**this**.start\_time=t1;

**this**.end\_time=t2;

**this**.activity=act;

}

O activitate este compusa din timp de incepre de terminare si numele acesteia.

3.3.Clasa ReadFile

In aceasta clasa se citeste dintun fisier fiecare activitate si se creaza o lista cu fiecare astfel putem face respectivele tascuri.

**public** ReadFile() {

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

String fileName = "activity.txt";

//read file into stream, try-with-resources

**try** (Stream<String> stream = Files.*lines*(Paths.*get*(fileName))) {

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

list.forEach(n->{

MonitoredData m=**new** MonitoredData(getSStime(n),getEStime(n),getAC(n));

list2.add(m);

}

);

} **catch** (IOException e) {

e.printStackTrace();

}

}

In aceasta clasa sunt toate metodele care efectueaza operatii asupra listei create precum si metodele care rezolova tascurile .Spre exemplu :

**public** Map<String, Long> apearActivity() {

**long** count1= list2.stream().filter(s->s.getActivity().equals("Toileting")).count();

**long** count2 = list2.stream().filter(s->s.getActivity().equals("Sleeping ")).count();

**long** count3=list2.stream().filter(s>s.getActivity().equals("Spare\_Time/TV")).count();

**long** count4 = list2.stream().filter(s->s.getActivity().equals("Showering")).count();

**long** count5 = list2.stream().filter(s->s.getActivity().equals("Snack")).count();

**long** count6 = list2.stream().filter(s->s.getActivity().equals("Lunch ")).count();

**long** count7 = list2.stream().filter(s->s.getActivity().equals("Grooming")).count();

**long** count8 = list2.stream().filter(s->s.getActivity().equals("Breakfast")).count();

**long** count9 = list2.stream().filter(s->s.getActivity().equals("Leaving")).count();

Map<String,Long> map = **new** HashMap<>();

map.put("Toileting", count1);

map.put("Sleeping",count2);

map.put("Spare\_Time/TV",count3);

map.put("Showering",count4);

map.put("Snack",count5 );

map.put("Lunch",count6 );

map.put("Grooming",count7);

map.put("Breakfast",count8);

map.put("Leaving",count9);

**return** map;

}

Aici este metoda care mapeaza fiecare activitate in functie de nr de aparitii.Pentru a face acest lucru se foloseste metoda count si metoda filter aplicate pe lista de activitati.

3.4.Java steam metode

Aplicația Stream API ajută să înlocuiască, pentru fiecare și în timp ce buclele. Acesta permite concentrarea asupra logicii operației, dar nu asupra iterației asupra secvenței elementelor. De exemplu:

for (String string : list) {

     if (string.contains("a")) {

        return true;

    }

}

Acest cod poate fi modificat doar cu o singură linie de cod Java 8:

boolean isExist = list.stream().anyMatch(element ->element.contains("a"));

Următorul cod creează un Stream <String> din lista <String>, găsește toate elementele din acest flux care conțin char "d" și creează un flux nou care conține numai elementele filtrate:

Stream<String>stream =

list.stream().filter(element->element.contains("d"));

API-ul Stream oferă un set de instrumente la îndemână pentru a valida elementele unei secvențe în conformitate cu unele predicate. Pentru aceasta, se poate utiliza una dintre următoarele metode: anyMatch (), allMatch (), noneMatch (). Numele lor se explică de la sine. Acestea sunt operații terminale care returnează un boolean.

boolean isValid = list.stream().anyMatch(element -> element.contains("h"));

boolean isValidOne = list.stream().allMatch(element -> element.contains("h")); boolean isValidTwo = list.stream().noneMatch(element ->element.contains("h"));

Reducerea poate fi asigurată și de metoda collect () de tip Stream. Această operație este foarte utilă în cazul conversiei unui flux într-o colecție sau într-o hartă și reprezentând un flux sub forma unui singur șir. Există o clasă de utilitate Colectoare care oferă o soluție pentru aproape toate operațiunile tipice de colectare. Pentru unele sarcini banale, poate fi creat un colector personalizat.

List<String> resultList = list.stream().map(element -> element.toUpperCase()).collect(Collectors.toList());

Metoda Java Stream forEach () este o operație terminală care inițiază iterația internă a elementelor din Stream și aplică un consumator (java.util.function.Consumer) fiecărui element din Stream. Metoda forEach () revine nevalid. Iată un exemplu Java Stream forEach ():

List<String> stringList = new ArrayList<String>();

stringList.add("one");

stringList.add("two");

stringList.add("three");

stringList.add("one");

Stream<String> stream = stringList.stream();

stream.forEach( element -> { System.out.println(element); });

Această metodă ia un mapper ca parametru, pe care îl folosește pentru a face conversia, atunci putem numi metoda sum () pentru a calcula suma elementelor fluxului. Să vedem un exemplu rapid despre modul în care îl putem folosi:

List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

Integer sum = integers.stream().mapToInt(Integer::intValue).sum();

A doua metodă de calculare a sumei unei liste cu numere întregi este prin utilizarea operației terminal colector ():

List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

Integer sum = integers.stream().collect(Collectors.summingInt(Integer::intValue));

4.Testarea aplicatiei

Aceasta aplicatie se testeaza foarte usor doar se actualizeaza fiecare activitate in fisierul text si se alege taskul dorit .

5.Dezvoltarea ulterioara

Din aceasta tema mi-am dat seama de numaratele moduri de a lucra cu steamuri cat si despre importanta modului de lucru cu expresii lamda.

O prima dezvoltare ar fii extinderea domeniului de monitorizare adaugand caracteristici despre activitati.

O alta ulterioara dezvoltare a acestei aplicatii de monitorizare ar putea fii crearea de operatii noi nu doar cele descrise de mine.

Bibliografie

[1].httpshttp://tutorials.jenkov.com/java-functional-programming/streams.html#foreach

[2]. https://www.baeldung.com/java-stream-filter-lambda

[3]. http://www.drdobbs.com/jvm/lambdas-and-streams-in-java-8-libraries/240166818

[4]. https://www.geeksforgeeks.org/stream-in-java/

[5]. https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/