Facultatea de Automatică și Calculatoare Cluj-Napoca

Documentație

Sistem de analiza a unor activitati

Student: Costea Ovidiu-Bogdan

Grupa:30229

Contents

1.Obiectivul temei: 3

2.Analiza problemei ,modelare,cazuri de utilizare 3

2.1Analiza problemei 3

2.2. Modelarea 3

2.3.Cazuri de utilizare 3

3.Proiectare 4

3.1. Structuri de date 4

3.2. Clase 5

3.3Algoritmi folosiți 5

3.4. Pachete 5

3.5. Interfețe 5

3.6.Interfața cu utilizatorul ..... **Error! Bookmark not defined.**

4. Implementare și Testare 6

5.Rezultat 6

6.Concluzie 6

7.Bibliografie 6

1.Obiectivul temei:

Principalul scop al acestei teme este realizarea unei aplicati care sa analizeze rezultatele unor activitati .Implementarea se bazeaza pe utilizarea expresiilor lambda si a streamurilor.Proiectul a fost realizat in limbajul de programare Java ,iar ca mediu de programare am folosit IDE-ul Eclipse.

Proiectul pe care deja l-am implementat ofera posibilitatea utilizari atat de catre clienti oferind posibilitatea adaugari drepturilor de administrator, activitatile pe care acestia le pot executa vor fi foarte bine conturate

2.Analiza problemei ,modelare,cazuri de utilizare

2.1Analiza problemei

Dupa o scurta analiza a cerintei obiectivele care trebuie indeplinite sunt foarte bine conturate. Astfel aplicatia trebuie sa permita preluarea activitatilor dintr-un fisier, pentru fiecare activitate se va specifica pe langa data de inceput, data de sfarsit si tipul activitati. Dupa citirea acestor date sin fisier vom putea analiza cu usurinta informatiile dorite.

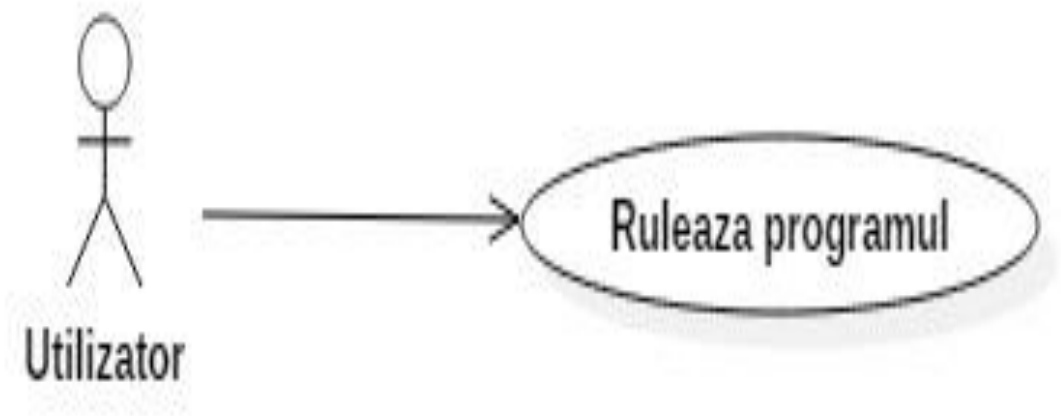
2.2. Modelarea

Pentru implementarea acestui analizator de activitati am avut nevoie de o singura clasa care retinea atat data de inceput cat si data de sfarsit a acitivitatii. Pe langa acesta clasa am mai avut folosit clasa App care se ocupa cu efectuarea operatiile care trebuie sa fie realizate asupra datelor preluate din fisier. Deoarece acest sistem nu necesita o interfata grafica nu mai avem nevoie de alte clase.

2.3.Cazuri de utilizare

Utilizarea aplicatie este simpla. Odata cu deschiderea aplicatiei vom putea putea vedea anumite rezultate in consola aplicatiei. Prentu vizualizarea celorlalte rezultate va trebui sa deschidem fisierele denumite sugestiv in functie de problema pe care o rezolva. Spre deosebire de celelalte proiect acum va trebui sa introducem detele de intrare intr-un fisier numit ” Activity ” care are extensia ” .txt ”. Datele introduse trebuie sa respecte un anumit format astfel zilele , lunile vor trebui sa respecte semnificatea pe care acestea o au in calendar. Anii cu singuranta vor avea exact 4 cifre iar distanta exacta dintre data de start si cea de sfarsit trebuie sa fie de exact doua taburi. Activitate va urma formatul unui String si va respecta la randul ei distanta de doua taburi fata de data de sfarsit a activitati .

Diagrama USE-CASE:



3.Proiectare

Aplicatia a fost dezvoltată printr-o abordare top-down , sistemul fiind proiectat ca un tot, apoi acesta a fost divizat in subsisteme. Clasele au fost realizate intr-un mod abstract, acestea asteptandu-se la un rezultat specific fara a furniza instructiuni de returnare a acelui rezultat. Dupa ce un model acceptabil a fost atins metodele au putut fi definite.

Motivul utilizarii acestei abordari a fost faptul că oferă un foarte bun nivel de abstractizare, acest lucru imbunatățind viteza de dezvoltare a aplicației, un cod mai ușor de întreținut și mai pușin predispus la erori.

3.1. Structuri de date

Structura principala de date pe care s-a bazat acest proiect este Map<K,V> care mapeaza valoarea de tipul K in locul calculat dupa hash codul indicat de cheia K, in cateva cuvinte ne permite sa retinem valori egale sau diferite la locuri indicate de chei care trebuie sa fie strict diferite. La fel cum HashMap<K,V> functioneaza acelasi principiu este folosit si in cazul Map<K,V>.In acest proiect s-a mai folosit si structura Stream<T> care reprezinta o secventa sau un set de elemente de tipu T asupra carora se pot efectua operatii de agregare(operatiile de agregare sunt :filter, map, limit, reduce, find care permit lucrul cu expresii lambda).

3.2. Clase

Clasa MonitoredData ne ofera posibilitatea de a salva si a efectua anumite operatii asupra activitatiilor care au fost preluate din fisierul ” Activity ”. Acesta clasa are atribute un timp de start, un timp de finish si tipul activitati.

Pe langa clasa MonitoredData am mai creat inca o clasa App care are ca principal rol pornirea aplicatiei. Pe langa operatiile de citire din cadrul fisierul au mai fost implemenntate problemele propriuzise ale aplicatiei. Dupa stocarea datelor din fisier intr-o list de obiecte de tipul MonitoredData incepe rezolvarea propriuzisa a problemelor.

Diagrama UML de clase :

3.3Algoritmi folosiți

In dezvoltarea acestei aplicatii nu am avut nevoie de algoritmi care sa se fie eficienti din punct de vedere al timpului si al memorie ,datorita lucrului cu tabele de dispersie care deja implementeaza metode foarte eficiete din acest punct de vedere. Singurele implementari care au fost facute au fost verificare datelor de intrare pentru creearea liste de activitati.

3.4. Pachete

Datorita dimensiuni foarte mici a acestui proiect am optat pentru structurarea claselor intr-un singur proiect astfel toate clasele se gasesc in pachetul tema5.tema5.

3.5. Interfețe

Nu am avut nevoie de nicio interfata pentru implementarea acestei aplicatii.

4. Implementare și Testare

Clasa App implementeaza citirea datelor din fisier dar si rezolvarea fiecarei cerinte. Principalele rezultate rezultate in urma operatiilor sunt stocate intr-o structura de date de tipul Map<K,V> care retine infromatiile si datele pe care dorim sa aplicam cerintele problemei. Dupa creearea listei de tipul MonitoredData se va parcurge lista cu ajutorul unui stream care ne va oferi posibilitatea sa efectuam anumite operatii de agregare.

Anumite metode au fost folosite in dezvoltarea acestui proiect astfel functia filter ne permite simpla filtrare a tuturor obiectelor retinute de stream . In acest mod nu mai avem nevoie sa parcurgem toata lista si sa facem operatiile necesare.In anumite se cerea gruparea activitatiilor sau a zilelor si aflarea anumitor rezultate pe baza acestora. Pentru gruparea dupa un anumit criteriu am folosit functia groupingBy(). De asemenea pentru contorizarea zilelor care indeplinesc anumite conditii am folosit functia summingInt(element->1) cu specificarea ca la fiecare pas sontorul se va amplifica su cate o unitate.

5.Rezultat

Rezultatele se obtin doar pe baza datelor de intrare preluate din fisier de activitati. In cazul in care se dorete testarea acestei aplicatii va fi nevoie o atentie sporita si multe calcule.

Acet sistem are un scop foarte bine conturat aflarea unor rezultate care sunt rezultatul unor date de intrare. Rezultatele afisate de fiecare cerinta pot fi vizibile in consola dar si in fisierele care poarta nume sugestive in functie de cerinta pe care o implementeaza.

6.Concluzie

Proiectul pot spune dupa parerea mea este un foarte bun exemplu care ne ofera oportunitatea de a ne dezvolta cunostiile asupra expresiilor lambda , dar si a lucrului cu streamuri. Pe langa aceste avantaje ne ajuta sa ne dezvoltam gandirea in implementarea proiectelor si rezolvarea problemelor majorare care apar in dezvoltarea si depanarea anumitor probleme.

7.Bibliografie

https://stack overf low.co m

https://www.y out ube.co m /

htt ps:// www.tut oria lspoint. Co m/

Notiuni teoretice:

`public interface` **`Stream<T>`**
`extends BaseStream<T,Stream<T>>`
A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using `Stream` and `IntStream`:

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

In this example, `widgets` is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

In addition to `Stream`, which is a stream of object references, there are primitive specializations for `IntStream`, `LongStream`, and `DoubleStream`, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

To perform a computation, stream operations are composed into a *stream pipeline*. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more *intermediate operations* (which transform a stream into another stream, such as `filter(Predicate)`), and a *terminal operation* (which produces a result or side-effect, such as `count()` or `forEach(Consumer)`). Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source. However, if the provided stream operations do not offer the desired functionality, the `BaseStream.iterator()` and `BaseStream.spliterator()` operations can be used to perform a controlled traversal.

A stream pipeline, like the "widgets" example above, can be viewed as a *query* on the stream source. Unless the source was explicitly designed for concurrent modification (such as a `ConcurrentHashMap`), unpredictable or erroneous behavior may result from modifying the stream source while it is being queried.

Most stream operations accept parameters that describe user-specified behavior, such as the lambda expression `w -> w.getWeight()` passed to `mapToInt` in the example above. To preserve correct behavior, these *behavioral parameters*:

- must be non-interfering (they do not modify the stream source); and
- in most cases must be stateless (their result should not depend on any state that might change during execution of the stream pipeline).

Such parameters are always instances of a functional interface such as `Function`, and are often lambda expressions or method references. Unless otherwise specified these parameters must be *non-null*.

A stream should be operated on (invoking an intermediate or terminal stream operation) only once. This rules out, for example, "forked" streams, where the same source feeds two or more pipelines, or multiple traversals of the same stream. A stream implementation may throw `IllegalStateException` if it detects that the stream is being reused. However, since some stream operations may return their receiver rather than a new stream object, it may not be possible to detect reuse in all cases.

Streams have a `BaseStream.close()` method and implement `AutoCloseable`, but nearly all stream instances do not actually need to be closed after use. Generally, only streams whose source is an IO channel (such as those returned by `Files.lines(Path, Charset)`) will require closing. Most streams are backed by collections, arrays, or generating functions, which require no special resource management. (If a stream does require closing, it can be declared as a resource in a `try`-with-resources statement.)

Stream pipelines may execute either sequentially or in parallel. This execution mode is a property of the stream. Streams are created with an initial choice of sequential or parallel execution. (For example, `Collection.stream()` creates a sequential stream, and `Collection.parallelStream()` creates a parallel one.) This choice of execution mode may be modified by the `BaseStream.sequential()` or `BaseStream.parallel()` methods, and may be queried with the `BaseStream.isParallel()` method.

Map:

<R> Stream<R> map(Function<? super T,? extends R> mapper)
Returns a stream consisting of the results of applying the given function to the elements of this stream.

This is an intermediate operation.

Type Parameters:

R - The element type of the new stream

Parameters:

mapper - a non-interfering, stateless function to apply to each element

Returns:

the new stream

Filter:

Stream<T> filter(Predicate<? super T> predicate)
Returns a stream consisting of the elements of this stream that match the given predicate.

This is an intermediate operation.

Parameters:

predicate - a non-interfering, stateless predicate to apply to each element to determine if it should be included

Returns:

the new stream

Distinct:

Stream<T> distinct()
Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.

For ordered streams, the selection of distinct elements is stable (for duplicated elements, the element appearing first in the encounter order is preserved.) For unordered streams, no stability guarantees are made.

This is a stateful intermediate operation.

API Note:

Preserving stability for distinct() in parallel pipelines is relatively expensive (requires that the operation act as a full barrier, with substantial buffering overhead), and stability is often not needed. Using an unordered stream source (such as generate(Supplier)) or removing the ordering constraint with BaseStream.unordered() may result in significantly more efficient execution for distinct() in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with distinct() in parallel pipelines, switching to sequential execution with BaseStream.sequential() may improve performance.

Returns:

the new stream