MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Processing Sensor Data Of Daily Living Activities

Fundamental Programming Techniques

Horvath Andrea Anett

Faculty of Automation and Computer Science
Computer Science Department
2'nd year, Group 30424

Contents

I. Objectives of the proposed laboratory work

The main objective of the proposed laboratory work was to design, implement and test a Java application that allows analyzing the behavior of a person recorded by a set of sensors installed in the house and accessing the results in files.

As secondary objectives, the following can be considered:

- developing the application in accordance with the functional programming principles
- working with lambda expressions
- getting familiar with stream processing
- drawing the UML diagrams and writing the code based on them (forward engineering)
- using appropriate data structures for the desired approach
- creating .txt files
- determining all the use-cases and assuring that the program takes into account all the possibilities
- getting familiar with JavaDoc files
- running the project as a .jar file

II. Analysis of the laboratory work, modeling, scenarios and use-cases

Analysis

The Processing Sensor Data Of Daily Living Activities application was developed to implement 6 tasks consisting in processing and interpreting data provided by a set of sensors installed in the house and stored in a .txt file. The historical log of the person's activity is stored as tuples: start time, end time, activity label. The types of recorded activities are: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare time/TV, Grooming.

The tasks that are to be implemented are:

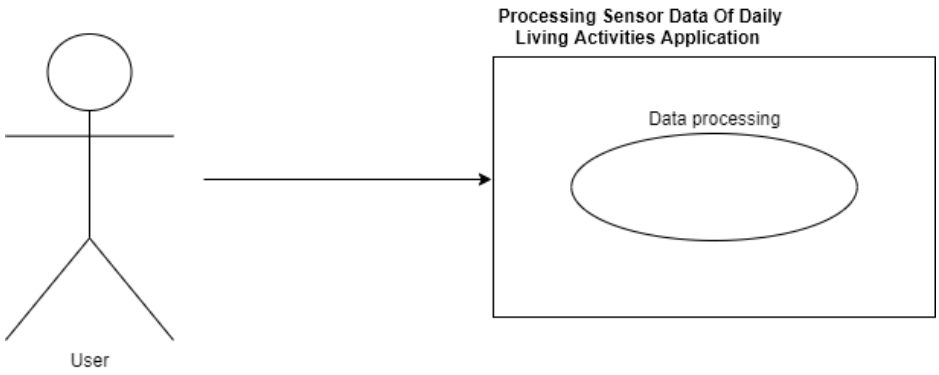
- Task 1: read the data from the .txt file;
- Task 2: count the distinct days that appear in the monitoring data;
- Task 3: count how many times each activity has appeared over the entire monitoring period;
- Task 4: count for how many times each activity has appeared for each day over the monitoring period;
- Task 5: for each activity compute the entire duration over the monitoring period;
- Task 6: filter the activities that have more than 90% of the monitoring records with duration less than 5 minutes.

Modeling

The application will run as a .jar file. The Processing Sensor Data Of Daily Living Activities app will retrieve the data from an "Activities.txt" file. The recorded data will be stored in a list containing objects of type MonitoredData. Objects of this class have as attributes the label of the activity stored as a String and the start and the end time of each activity stored as Date. The program implements the required tasks using streams and lambda expressions and creates a .txt file with the result of each task.

Scenarios and use-cases

The only use-cases is sketched in the next diagram and detailed below.



In the following use-case the primary actor is considered to be the user.

Use-case: *Data Processing*

Main success scenario:

1. The user creates the file with the data named "Activities.txt" in the project's folder.
2. The user opens Command Prompt.
3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command to run the application.
5. The application verifies the existence of the input file.
6. The application executes the 6 tasks and creates a .txt file with the output of each task.

Alternative sequence A:

1. The user creates the file with the data with the wrong name.
2. The user opens Command Prompt.
3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command to run the application.
5. The application verifies the existence of the input file and an exception is thrown and the program exits.

Alternative sequence B:

1. The user creates the file with the data with the correct name in a wrong place.
2. The user opens Command Prompt.
3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command to run the application.
5. The application verifies the existence of the input file and an exception is thrown and the program exits.

Alternative sequence C:

1. The user creates the file with the data with the correct name and in the right place.
2. The user opens Command Prompt.
3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command to run the application.
5. A system error occurs and the files cannot be opened.
6. An exception is thrown and the program exits.

III. Design

1. Development approach

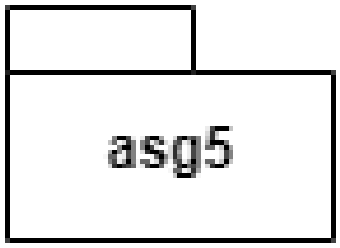
The queues simulator application is all included in one package, the asg5 package. Since the object oriented paradigm imposed in case of this task only four classes which did not needed a further classification into packages, only the asg5 package was used.

Thus, the only existing package includes the following classes:

- MonitoredData: contains the model that stores the provided data in tuples;
- Monitor: contains the implementation of the tasks;
- Writer: contains the methods for creating the output .txt files for each task;
- Main: contains only the main method which calls the tasks from Monitor.

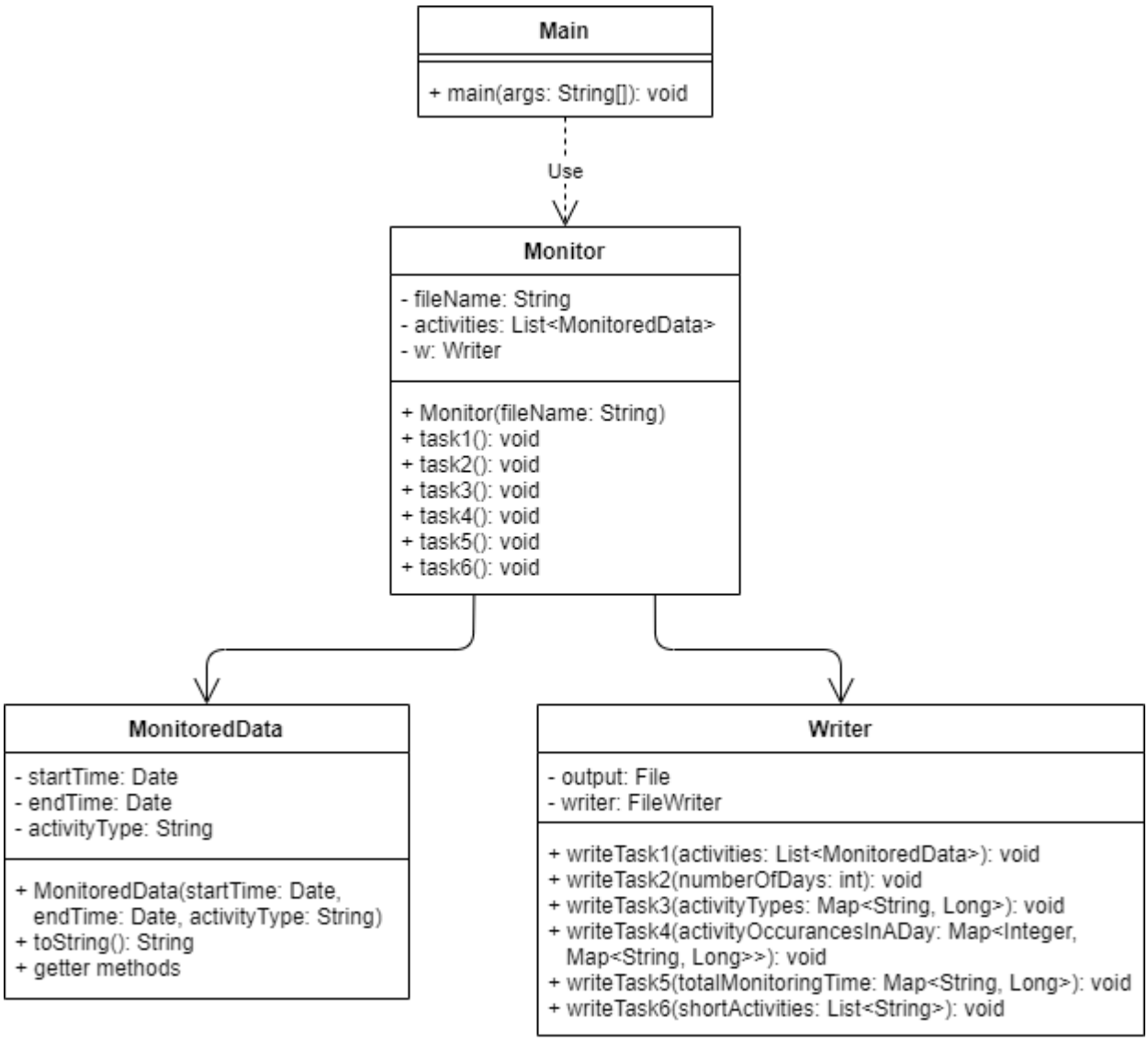
2. Structure and UML Diagrams

2.a. Package Diagram



The content of the package is shown in the class diagram.

2.b. Class Diagram



IV. Implementation

a. Class MonitoredData

Instance variables: *startTime*: Date, *endTime*: Date, *activityType*: String.

The variables represent the attributes of an activity recorded by the sensors. Each tuple in the activity log .txt file is structured to have these three attributes: start time, end time and the label of the activity.

The class contains the constructor, and getters in order to have access to the private instance variables (all of them) since their only purpose is to provide a model for the activity.

The class also overrides the method *toString* which provides the possibility of displaying the content of the historical log of the person's activity.

b. Class Monitor

Instance variables: *fileName*: String, *activities*: List<MonitoredData>, *w*: Writer.

The constructor of the class receives as parameter a String representing the name of the file.

Besides the void methods implementing the required tasks, the class contains three functions.

```
static Function<String, Date> convStringToDate = date -> {
        try {
            return new SimpleDateFormat("yyyy-MM-dd hh:mm:ss").parse(date);
        } catch (Exception e) {
        }
        return null;
    };
```

The *convStringToDate* function converts a String to a Date and is used to process the input from the Activities.txt file with the log of the person's activity.

```
static Function<Date, Date> getSimpleDateFormat = date -> {
        SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
        try {
            return formatter.parse(formatter.format(date));
        } catch (Exception e) {
        }
        return null;
    };
```

The *getSimpleDateFormat* function is used for the second task and formats a date of form "yyyy-MM-dd hh:mm:ss" to a date of form "dd/MM/yyyy". Converting all the dates in the log to this form, the number of different days can be determined (the hours, minutes and seconds have to be ignored in order to retrieve the different days, so by formatting the date in this way, all of them will have the same time in the day - the current hour, minutes, seconds).

```
static BiFunction<Date, Date, Long> getTimeInterval = (dateBeginning, dateEnd) ->
Math.abs(dateEnd.getTime() - dateBeginning.getTime()) / 1000;
```

The *getTimeInterval* bifunction receives two input dates and computes the time interval between them in seconds. The .getTime() returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by the date on which is applied. Computing the difference between the dateEnd.getTime() and the dateBeginning.getTime() we obtain the time interval between them in milliseconds and by dividing by 1000 we get the result in seconds.

The other 6 methods in this class are implementing the required 6 methods.

```
public void task1() {
        Stream<String> stream;
        try {
            stream = Files.lines(Paths.get(fileName));
            activities = stream.map(line -> line.split("[\t]+"))
                    .map(attribute -> new MonitoredData(convStringToDate.apply(attribute[0]),
convStringToDate.apply(attribute[1]), attribute[2]))
                    .collect(Collectors.toList());
        } catch (Exception e) {
        }

        w.writeTask1(activities);
    }
```

This method reads the data from the Activities.txt file using streams and splits each line in 3 parts: start time, end time and activity label, and creates a list of objects of type *MonitoredData*. In the end, the corresponding method from the *Writer* class is called.

```
public void task2() {
        List<Date> days = new ArrayList<Date>();
        activities.forEach(item -> days.add(getSimpleDateFormat.apply(item.getStartTime())));
        int numberOfDays = (int) days.stream().distinct().count();

        w.writeTask2(numberOfDays);
    }
```

This method creates a list containing the start days from the log in the form dd/MM/yyyy, this way not taking into account the hour differences. The number of distinct days in the list is counted and sent as parameter to the method corresponding to task2 from *Writer* class.

```
public void task3() {
        Map<String, Long> activityTypes = activities.stream()
                .collect(Collectors.groupingBy(MonitoredData::getActivityType,
Collectors.counting()));

        w.writeTask3(activityTypes);
    }
```

This method implements task 3 and constructs a structure of type Map<String, Long> representing the mapping of each distinct activity to the number of occurrences in the log; therefore, the key of the map will represent a String corresponding to the activity name, and the value will represent a Long corresponding to the number of times the activity has appeared over the monitoring period. In the end the method *writeTask3* from class *Writer* will be called with the obtained map as a parameter.

```
public void task4() {
        Map<Integer, Map<String, Long>> activityOccurancesInADay = activities.stream()
                .collect(Collectors.groupingBy(d -> d.getStartTime().getDate(),
                        Collectors.groupingBy(MonitoredData::getActivityType,
                                Collectors.counting())));

        w.writeTask4(activityOccurancesInADay);
    }
```

The task 4 creates a structure of type Map<Integer, Map<String, Long>> that contains the activity count for each day of the log; therefore the key of the map will represent an Integer corresponding to the number of the monitored day, and the value will represent a map: in this map the key which is a String corresponds to the name of the activity, and the value which is an Integer corresponds to the number of times that activity has appeared within the day. In the end the method *writeTask4* from class *Writer* will be called with the obtained map as a parameter.

```
public void task5() {
        Map<String, Long> totalMonitoringTime = activities.stream()
                .collect(Collectors.groupingBy(MonitoredData::getActivityType,
                        Collectors.summingLong(d -> getTimeInterval.apply(d.getStartTime(),
d.getEndTime()))));

        w.writeTask5(totalMonitoringTime);
    }
```

The task 5 creates a structure of type Map<String, Long> in which the key of the map will represent a String corresponding to the activity name, and the value will represent a Long corresponding to the entire duration of the activity over the monitoring period in seconds. For each type of activity, the time intervals are summed up. In the end the method *writeTask5* from class *Writer* will be called with the map as parameter.

```
public void task6() {
        Map<String, Long> activityOccurances = activities.stream()
                .collect(Collectors.groupingBy(MonitoredData::getActivityType,
Collectors.counting()));
        Map<String, Long> activitiesShortTimings = activities.stream()
                .filter(activity -> getTimeInterval.apply(activity.getStartTime(),
activity.getEndTime()) < 300)
                .collect(Collectors.groupingBy(MonitoredData::getActivityType,
Collectors.counting()));

        Function<String, Long> getValueFromActivitiesShortTimings = activity -> {
            for (Map.Entry<String, Long> entry : activitiesShortTimings.entrySet()) {
                if (entry.getKey().equals(activity)) {
                    return entry.getValue();
                }
            }
            return Long.valueOf(0);
        };

        List<String> shortActivities = activityOccurances.entrySet().stream()
                .filter(item -> getValueFromActivitiesShortTimings.apply(item.getKey()) /
item.getValue() > 0.9)
                .map(item -> item.getKey())
                .collect(Collectors.toList());

        w.writeTask6(shortActivities);
    }
```

In this method two maps are created. First one maps just like in task 3 the activities to their number of occurrences. The second one maps the activities to the number of occurrences where the time interval is less than 5 minutes. After this, a list with the activities that have more than 90% of the monitoring records with duration less than 5 minutes is created.

c. Class Writer

Instance variables: *output*: File, *writer*: FileWriter.

In this class are 6 methods, each one creating a .txt file with the output of each task.

V. Results

For testing the Processing Sensor Data Of Daily Living Activities application, the program was tested on the provided input file, which is included in the project's directory. The output is visible in the 6 generated txt files.

```
Number of different days over the monitoring periond: 14
```

```
Number of times each activity appeared over the monitoring period:

Breakfast: 14
Grooming: 51
Toileting: 44
Sleeping: 14
Leaving: 14
Spare_Time/TV: 77
Snack: 11
Showering: 14
Lunch: 9
```

```
Number of times each activity has appeared for each day over the monitoring period:

Day 1
Breakfast: 1
Toileting: 2
Grooming: 3
Sleeping: 1
Leaving: 1
Spare_Time/TV: 6
Showering: 1
Lunch: 1

Day 2
Breakfast: 1
Toileting: 3
Grooming: 4
Sleeping: 1
Spare_Time/TV: 7
Snack: 1
Showering: 1
Lunch: 1

Day 3
Breakfast: 1
Grooming: 3
Toileting: 2
Sleeping: 1
Leaving: 1
Spare_Time/TV: 4
Showering: 1
```

- continues with the other days.

```
The entire duration of each activity over the monitoring period:

Breakfast: 14:58:8
Grooming: 26:35:50
Toileting: 14:14:14
Sleeping: 116:27:39
Leaving: 39:44:44
Spare_Time/TV: 194:40:33
Snack: 0:6:1
Showering: 1:34:9
Lunch: 5:13:31
```

```
Activities that have more than 90% of the monitoring records with duration less than 5 minutes:

Snack
```

VI. Conclusions

The Processing Sensor Data Of Daily Living Activities application fulfills all the requirements of the laboratory work.

For future improvements the following can be considered:

- More interpretations for the provided data;
- User interface for visualizing the results.

VII. Bibliography

1. http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_4/

2. https://www.javatpoint.com/

3. https://amigoscode.com/courses/java-functional-programming