Activity Manager

1. Objectives

The aim of this application is to use Java 8 Stream objects to easily process a given list of activities. Several independent tasks must be achieved:

- Read the input file
- Count the number of days that are logged
- Count how many times each activity appears in the entire log and respectively for each day
- Compute the duration for each activity instance and respectively the total duration for each activity
- Filter the activities that have 90% of the monitoring records with less than 5 minutes of duration

2. Problem Analysis

This chapter aims to describe the requirements of the problem as well as how the end-users will interact with the application, together with any assumptions made about the input of such end-users.

2.1. Functional Requirements

The finished application must provide the following functionality:

- Provide an basic interactive interface to the target users, allowing them to request any of the operations described in chapter 1

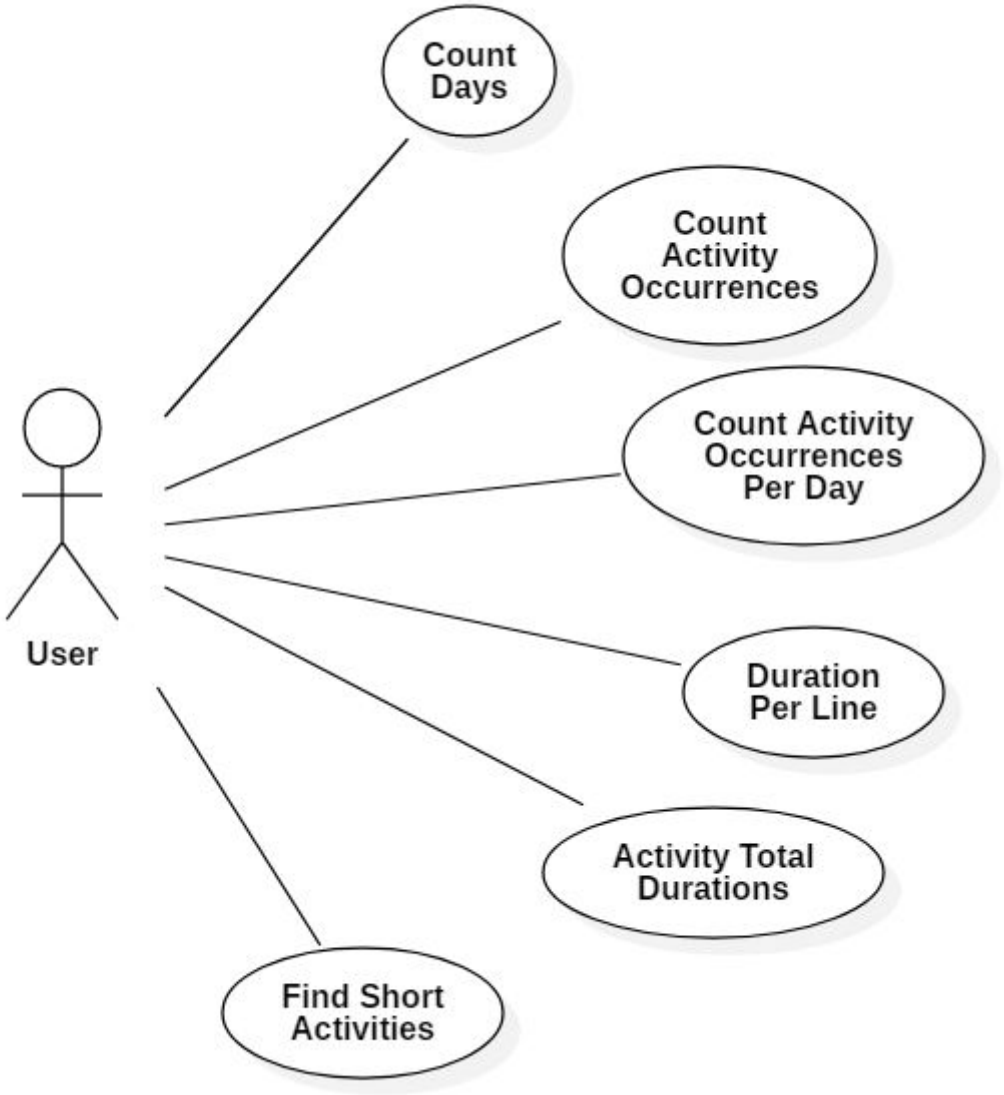
2.2. Assumptions

We have not made any assumptions about the input users provide, because the user is not required to give any data, but merely select the desired functionality of our application.

We did, however, assume that we will be provided with a file named *Activities.txt* that follows the specifications provided in the laboratory (namely that it will encode one activity per line, and that an activity is defined by a start time, and end time and a label / name). We also assume that this file has a correct structure and that there are no contradictory data in the dataset. We believe it is also acceptable to assume that there will be no continuous activity spanning more than 24 hours.

2.3. Use Cases

The following diagram shows the possible use cases for this application:



Use case: count activity occurrences per day
Actor: user
Preconditions:

- The activity file has been read properly
- The activity file was valid and non-empty

Main success scenario:

1. The user selects that he wants to see the number of times each activity occurs each day
2. The program uses Java 8 Streams to obtain, for each day, a list of pairs of activity name and number of occurrences
3. The program displays the results in the UI
4. The system stores the order

5. The system notifies the chef which dishes need to be prepared

Alternative sequences:

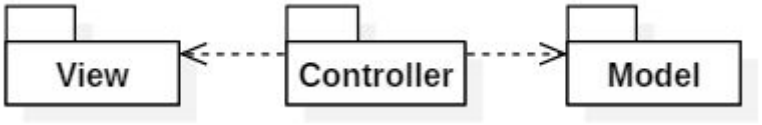
A. None. This use cannot fail, given the preconditions.

3. Design

This chapter deals with how the the problem is approached from an object-oriented perspective, how it will be split into packages, what interfaces and data structures will be used and the algorithms that will be employed.

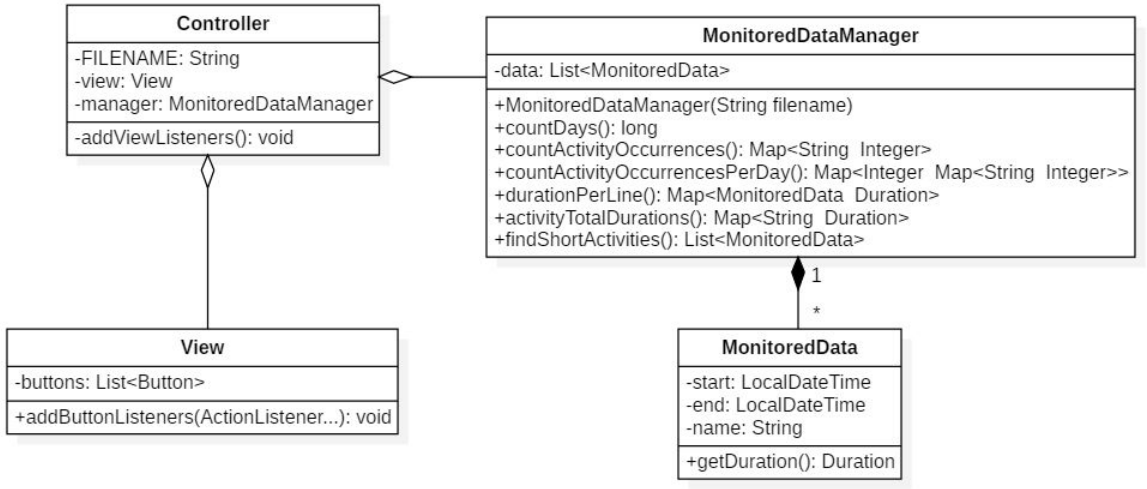
3.1. Package Diagram

The application, due to its extremely small size, will be built on a very simple MVC package structure, as shown in the diagram below:



3.2. Class Diagram

In this chapter, we will discuss what kind of classes we wish to implement and explain our reasoning. Since the application is relatively small, we will include an application-wide class diagram below:



There is nothing hard to understand in the above figure, actually:

- MonitoredData will be the entity mapping an activity
- MonitoredDataManager will be the class performing operations on the list of activities
- View will be a simple view providing the user buttons for input and a text area for output
- Controller will be a connecting class that manages user actions through View and outsources its computations to MonitoredDataManager

4. Implementation

This chapter describes the classes and interfaces in more detail, covering the most important methods and aspects of each. We will focus mostly on the Java 8 part, since Views and Controllers have been presented in past applications, too.

4.1. MonitoredData

The MonitoredData maps on the activity format provided in Chapters 1 and 2.

```
public class MonitoredData {
    private String name;
    private LocalDateTime start;
    private LocalDateTime end;
```

While it is relevant that the class constructor parses String objects directly to obtain its start and end date, we believe that presenting the *getDuration()* method is more relevant, since the presence of this method economises us some time later.

```
/**
 * Computes and returns the duration between the beginning and end of the data.
 * @return A Duration object.
 */
public Duration getDuration() {
    return Duration.between(start, end).abs();
}
```

Notice how we don't *actually* do any computation ourselves, since the Duration class in *java.time* can manage just fine on its own.

4.2. MonitoredDataManager

The most important class, this is where using Streams saves us many lines of code. Despite the novelty of the subject, we believe that Streams are fairly straightforward, so we will only cover the most relevant examples.

Note that our methods work in pairs usually: one that produces a list or map of values, followed by another that can convert those into an easy-to-read String. The only exception is the method that computes the duration for each activity, where we simply return a list, because a map would have been superfluous, given that we have the *getDuration()* method. We will not present the methods for converting data to String format, because they are very, very simple.

```
/**
 * Counts the number of distinct days over which the activities extend
 * @return A number of days
 */
public long countDistinctDays() {
    List<LocalDate> dates = new ArrayList<LocalDate>();
    activities.forEach(activ -> {
        dates.add(activ.getStart().toLocalDate());
        dates.add(activ.getEnd().toLocalDate());
    });

    return dates.stream().distinct().count();
}
```

The above method is pretty plain to read: first we create a list in which we put every start and end date we meet. Then, we simply count the distinct values in that list. (Since we've assumed no activity is longer than one day, doing this will guarantee that every date in which an activity happened has been caught)

```java
/**
 * Computes the total duration across all days for each activity.
 * @return A map of names to durations.
 */
public Map<String, Duration> totalDuration() {
    return activities.stream()
    .collect(
            Collectors.toMap(MonitoredData::getName,
                             MonitoredData::getDuration,
                             Duration::plus)
    );
}
```

This method is also plain to read once we know what the call to Collectors does. A Collector is an object that takes a Stream and *collects* it into something - in our case, a map. Luckily for us, the *java.util.stream.Collectors* library happens to have a method that lets us easily get a Collector object that uses our chosen methods.

In plain English, the code above says:

- Turn the activities into a Stream
- Then collect that Stream into a Map, so that:
    - The keys are the names of the activities
    - The values are the durations
    - If we find two activities with the same name, we add their durations together

```java
/**
 * Generates a list of the activities that have over 90% of
 * their occurrences represented by activities of 5 minutes or less.
 * @return A list of strings representing activity names.
 */
private List<String> selectShortActivityNames() {
    Map<String, Integer> occurrences = activities.stream()
    .collect(Collectors.groupingBy(MonitoredData::getName, Collectors.summingInt(e->1)));

    Map<String, Integer> underFiveMinutes = activities.stream()
    .filter(act -> {return act.getDuration().minusMinutes(5).isNegative();}) //Accept only activities of under 5
    .collect(Collectors.groupingBy(MonitoredData::getName, Collectors.summingInt(e->1)));

    return underFiveMinutes.entrySet().stream()
    .filter(elem -> {
        return elem.getValue().intValue()  >=  0.9 * occurrences.get(elem.getKey());
    })
    .map(elem -> {return elem.getKey();})
    .collect(Collectors.toList());

}
```

Finally, the above is the method that filters only the short activities. For this, we get the number of occurrences in total for each ability, then

the number of occurrences that are under 5 minutes in length, then *filter()* so that we only keep those where 90%+ of occurrences are short. Good to know is that *summingInt(e->1)* is basically just a fancy way to say *counting()*, except in this case we collect into an int. Also good to know is that the *toList()* option in the last line provides a list that *does not* need to provide any methods for modification, so if you want to do more than just read from it, it's worth moving its contents to a friendlier data type.

4.3. View

A very simple view, given that there is no input to be had from the user. Attached below is a sample of the elements on the JFrame:

```
public class View {
    private JFrame frame;

    private JButton btnDistinctDays;
    private JButton btnTotalOccurrences;
    private JButton btnDailyOccurrences;
    private JButton btnTotalTime;
    private JButton btnActivityDurations;
    private JButton btnShortInstances;

    private JTextArea labelResult; //We use a JTextArea as a label, simply because it works.
```

The visual result is this:

Nothing complex, but enough to get the job done, no matter how big the dataset.

4.4. Controller

This class mostly deals with user input and setting up button listeners, and outsources most of its logic to its MonitoredDataManager.

```
view.addListenerShortInstances(e -> {
    String result = manager.textForSelectShortActivityNames();
    view.setResultsText(result);
});
```

Above is one of the button listeners we've added; this one is for demanding to see only the very short activities. Notice how the presence of a method in the manager to convert to nice text is very helpful. The Controller's main role is to attach these listeners connecting the manager to the view.

5. Results

Based on extensive test runs of the application, its performance has proven excellent, its interface good and no errors have been noticed. The program has not been unit-tested in any way, but given how small it is, they are not needed.

6. Conclusions

I believe this topic has been a good exercise in working with streams, collectors and maps. It has also been a good chance to unwind, given that it was much simpler than the previous projects.

Overall, I would say that Streams and Lambdas provide major advantages in terms of brevity and code readability, and would suggest to use such streams when possible.

7. Bibliography

1. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html for insight into how to make use of the Collectors library
2. https://www.baeldung.com/java-8-collectors for covering pretty much everything else about collecting streams, which is arguably the hardest part of the task