Time Tracker

102759 Software Design

September 11, 2021

1 Introduction

According to the Wikipedia, "*time-tracking software is a category of computer software that allows its users to record time spent on tasks or projects. The software is used in many industries, including those who employee freelancers and hourly workers. It is also used by professionals who bill their customers by the hour. These include lawyers, freelancers and accountants*"[1]. The bare functionality of an automatic time tracker application is

1. The user enters into the application one or more names of tasks on which he/she will eventually work in the future.
2. Each time he/she starts or stops working on a task, communicates it to the application so that it records the starting and ending times of intervals of uninterrupted dedication to the task. All time tracking apps give you a running clock that you launch when you start a task, and that you can pause or stop when you finish.
3. It is possible to query the application about the total time spent on a task or group of tasks in a certain period like last week, this month or all the time until now (for instance to bill a customer), and also when was that (the list of intervals within the period)

The main benefits of a time tracker are[2]

- Accurate billing: having exact data on how much time has been spent for the project helps to avoid the common problem of under/over-billing customers. Moreover, time trackers may generate the invoices.
- Traceability: time-track data can be quickly reviewed and summarized in reports. This way, it can easily be used for justification of costs or time expenses because we have registered when we did what, and even who in time trackers for teams.
- Improved estimation: by knowing how much time took a task it's possible to make future estimations of duration and cost more accurate when a task similar to one already completed comes up.

[1] `https://en.wikipedia.org/wiki/Time-tracking_software`
[2] `https://www.actitime.com/time-tracking/time-tracking-software-essay/`

Examples of popular time trackers as of 2021 are Toggl —“best free one-person time tracking app”—, Clockify —“the only truly free time tracker for teams”—, tmetric —“simple and accurate work time-tracker”—, trackingtime —integrates with Asana, Trello, Jira—, Timely —interface based on a calendar view—. These and others are listed and compared here[3].

We recommend you to try at least one of them to get a better idea of what a time tracker does, because we are going to build our own time tracker from scratch. Actually, it won’t be a fully fledged one because our goal is to learn to design and implement object oriented software and user interfaces. Instead, we are going to build a minimum viable product (MVP) that implements the core functionality of any time tracker and offers it through a user interface. For the sake of simplicity, it will be a personal, single user time tracker, not for a team. The difference being that there are not users sharing, collaborating in one or more tasks. However, extending our time tracker to a multiuser one wouldn’t be difficult to do once finished.

We have divided the whole project into three steps or milestones. In the first two you will build the basic functionality of the application. It has to be implemented in Java because has very convenient libraries for us, nicely supports the object orientation paradigm and it’s one of the most widespread programming languages nowadays. The third milestone is different in that we are going to build the user interface. While we could do this with some Java library also, we will develop it with the Flutter, which technically is library of Dart, a different object oriented programming language. The advantage is that you can build a cross-platform user interface, that is, for IOS, Android and web, with a single code base. What we have developed in the first and second milestones we will wrap it under a simple web service (with Java sockets) with which the Flutter user interface communicates.

2 Development tools

We will need several plugins to be installed into the development environment (IntelliJ) and rely on some libraries. We recommend you to do this at the moment they are needed, that is, not all of them from the very beginning. As mentioned, the timeline of the project has three parts or milestones, each with its own deliverable and assessment.

First milestone

IntelliJ IDEA will be our IDE (integrated development environment). Install the “community edition” which is free and doesn’t ask you to register. It is better to configure now an aspect of the code it generates automatically, in order to be compatible with another tool we will be using later, Checkstyle. Go to `File` → `Settings` → `Editor` → `Code style` → `Java` and change `Indent`, `Tab size` and `Continuation` to 2, 2 and 4, respectively.

PlantUML is a plugin for IntelliJ. This is to make UML class diagrams that document our design. Open IntelliJ and in the launch screen go to `Configure` → `Settings`. Or create/open a project and then `File` → `Settings`. From there, select `Plugins`,

[3] https://zapier.com/blog/best-time-tracking-apps/

look for `PlantUML integration` and install it. Once installed, we can create a class diagram by right-clicking on the project, `New` → `PlantUML File`, selecting `Class` and writing a name for our new PlantUML File.

org.json We will need this library to have JSON objects in Java and convert them into strings, and the other way around. In the first milestone this will be our way to implement persistence (save and restore the timetracker data). In the third milestone the Java webserver and the Flutter user interface will exchange JSON strings. To add this library to your IntelliJ Java project follow these steps: `File` → `Project structure` → `Libraries` → `+` → `From Maven...`, write `org.json` and click on search. Install the most recent stable version.

Second milestone

Checkstyle-IDEA (CSi) is another plugin for IntelliJ, install it like before. Once done go to `File` → `Settings` → `Tools` → `Checkstyle` and check `Treat errors as warnings` and `Google checks` (less demanding than Sun checks). To see the result, load some Java file and select `Analyze` → `Inspect code...`. Two new tabs `Inspection results` and `CheckStyle` will appear at the bottom.

SLF4J and Logback are two libraries we will need to perform logging. Search `org.slf4j:slf4j-ext` and select last stable version (not alpha or beta). Do the same with `ch.qos.logback:logback-core` and `ch.qos.logback:logback-classic`.

Third milestone

Flutter is "*Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase*". Follow installation instructions of the *Get started* section. This includes installing Android SDK (maybe by installing Android Studio, an IDE that we won't be using though). Do not install Flutter for Android Studio but for IntelliJ. Also, you'll need to install an Android virtual machine (AVD) within IntelliJ that emulates some specific mobile phone on which will run our Flutter app. Launch the AVD Manager and select the Pixel 2 device for instance. Disable front and back camera emulation because we won't need it.

3 Milestone 1

3.1 Hierarchy of projects and tasks

Until now we have used the words task and project interchangeably but they won't be the same for us. A task is something for which the user will count the time spent and also when has it been. For instance, if an undergraduate student has a task named *practicum* he wants to record all the intervals when he's been working on that, like yesterday from 21:00-23:00 and today from 17:00-18:30. Like in other time trackers, in ours users will be able to organize their tasks in other ways that a long list if they wish, that is, build a hierarchy. A project may contain tasks but also other projects. Mind the verb *may*, it means a project has zero or more tasks and zero or more other projects. The moment it is created, obviously it has no “children”, and same for tasks and intervals.

We are interested in recording the initial and final date and time of projects and tasks. “date-time” refers to a time instant: year, month, day, hour, minute and second. The initial date-time of a task is the first instant the user started working on it (not when it was created), and final date-time last moment he/she did. For a project it’s the same with respect to its descendent tasks. Also, we want to record the duration, the total time spent on something. For tasks it is equal to the sum of the durations of its intervals. For a project it is the sum of the durations of its children, in other words, of all its descendent intervals. Note that except for intervals, duration is not the difference between the initial and final date-time (quantized to some time count unit like n seconds). As we see in the example of appendix A, we have decided that projects and tasks have a name and intervals don’t.

3.2 Counting time

This refers to, obviously, count the time for a task the user has decided to do or to work on, until he/she wants to stop doing so. It seems a simple thing to implement like in listing 1.

Listing 1: How to *not* count the time

```java
Task t = new Task("study");
Scanner in = new Scanner(System.in);
t.start(); // set initial and final dates to now, duration to zero
while (true) {
  try {
    Thread.sleep(2000); // wait 2 seconds, the time count unit
    System.out.println("2␣seconds␣have␣passed");
    t.addSeconds(2); // update final date and duration
    System.out.println("do␣you␣want␣to␣stop␣task␣?");
    String s = in.nextLine();
    // wait until user writes something and presses return
    if (s.equals("yes")) {
      t.stop();
      break;
    }
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

The problem of this code is it’s not *responsive*: during 2 seconds the application stops listening to commands from the user (whether to stop or not) and can not do anything else than to wait. A similar thing happens when reading from console with `nextLine`. At these moments the application can not even print anything because it’s waiting for something to happen. This code simulates the user interaction by sending the start and stop messages to the task, but in the final time tracker there will be a user interface that we want to listen to user commands *always* and show the updated durations and final dates of running tasks and projects *always* too (actually every second or two seconds for instance), not just when the user is done with some job. The user wants to know at any time, constantly, how long has he/she been doing some task/project, active or not. The solution is to have a thread running in parallel, one dedicated to count the time and the other to the input/output to console or, later, to exchange messages with the graphical user interface.

3.3 Persistence

Persistence refers to saving user data just before leaving the application (at least) and loading it when execution starts. The format we have chosen for that is JSON[4,5]. The idea is to represent the tasks, projects and intervals as a JSON object that can be converted to a string and then saved to a file. The inverse process is also easy to do thanks to a method to parse this string (see section 3.7).

There are other ways to implement persistence but this one is quite simple (though not as simple as serialization) and, this is the true reason, the one we will need when developing the user interface.

3.4 Comments

Code must have sensible comments. Do not assume the reader has read this handout or knows anything about the project, but imagine he/she is the first time hears about it and has to understand the code to perform changes.

Each class should have a header comment explaining its purpose, how does it fit into the design, whether it is part of some design patter and why. Don't state the evident like "Class Task is abstract", "Class Task represents a task", instead explain what is a task and how is it different from a project, what role plays in a certain design pattern etc. Comments should explain also code difficult to understand, not obvious because the way it is (uses not common features of Java or some library) or its purpose. Any tricky detail has to be explained. On the other hand, classes, attributes, methods, variables, constants will have sensible names that reduce the amount of comments needed. Do not write Javadoc comments, we'll consider it another sign of copied code.

3.5 Test code

In order to test your code you need to simulate the actions of the user, which at the moment are just three: start a task, wait for some time and stop it. Probably the first idea you have to do it is to implement a menu based command line interface that prints messages like

```
What do you want to do ?
  1. Create task
  2. Count time
```

[4]https://www.baeldung.com/java-org-json

[5]https://stackabuse.com/reading-and-writing-json-in-java/

Listing 2: How to write the test code

```
package core;

public class Main {
  private static void simpleTest() {
    Task t0 = new Task("study");
    t0.start();
    wait(12);
    t0.stop()
  }
  private static void notSoSimpleTest() { ... }
  public static void main(String[] args) {
    simpleTest();
    // notSoSimpleTest();
  }
}
```

```
  3. Stop task
```

Don't. Instead, simply hardcode the user interaction like in listing 2. Put each test you want to run in a separate function of a `Main` class with a `main` method. And put all the classes of this first milestone into a package named `core`.

3.6 Debugging

Surely you will write a lot of sentences to print the value of variables during execution, to trace it make sure it works as expected, or to find errors. This is good and better than using the debugger, for several reasons we will see. Once the code works well, these messages become annoying and you want to get rid of them. At this point, don't remove them! Simply comment them out because in the next milestone we are going to transform the `System.out.println` sentences into something else called *log messages*.

3.7 Hints

Date and time types. Do NOT use classes `Date` and `Calendar`. The first is deprecated and also is what we used in former courses so it may be a sign of copied code. Instead, use classes of the `java.time` library introduced in Java 8. Specifically, use `LocalDateTime` and `Duration`, they have very convenient methods for us. In order to format dates and times, see also class `DateTimeFormatter`. To learn how to use these classes read [6,7].

Threads. Other classes interesting to know are `java.util.Timer` and `java.util.TimerTask`. Their joint use allows to create a thread that executes some function periodically by

[6] https://www.baeldung.com/java-8-date-time-intro

[7] https://www.w3schools.com/java/java_date.asp

means of the timer's method `scheduleAtFixedRate`. As for usage examples, see for instance [8,9].

Observer pattern. Do NOT use classes `java.beans.PropertyChangeListener`, `java.beans.PropertyChangeSupport` for the same reasons as above. Instead use other classes in `java.util` that do the same. Even though they are deprecated, they are simpler to use, are well known and the reasons for its deprecation are not a problem for us.

`org.json` library. As for persistence and JSON format, use classes `JSONObject`, `JSONArray` and `JSONTokener`. A simple way to write a file in Java is with class `FileWriter`[10]. To know how to read a JSON string in a file to reconstruct objects, see this example[11]. Note that to save a null Java object in JSON we have `JSONObject.NULL`. Finally, to parse a date time JSON string read[12,13,14].

3.8 Grades

E Some of these cases: can not create a tree of tasks and projects, can not count and display dates and duration for tasks, projects and intervals "live", that is, while one or more tasks are active, no class diagram or not in plantUML format, problems of low cohesion, high coupling or bad responsibility assignment, the design does not make use of the two or three design patterns you have to discover.

D One can create the hierarchy of projects and tasks of appendix A and count the time but dates and durations are not consistent. Or it is not possible to see the right dates and durations until a task stops. Or durations and times do not propagate upward from intervals/tasks to antecesor projects. In sum, there is some problem with the basic functionality of representation and counting time.

C The hierarchy of projects, tasks and intervals of appendix A is well constructed, and there is code to demonstrate that you successfully pass the test of appendix B.

B C plus you have implemented persistence by saving the tree in JSON format to a text file. There is code to test that what you reconstruct from the file is the same you had before saving, for example by printing the JSON objects. Besides persistence, converting a subtree to JSON format is something we will need in the third milestone, so we recommend you to implement it now because then it won't count to get a higher mark.

[8] https://www.baeldung.com/java-timer-and-timertask

[9] https://alvinalexander.com/source-code/java/java-timertask-timer-and-scheduleatfixedrate-example

[10] https://www.w3schools.com/java/java_files_create.asp

[11] https://kodejava.org/how-do-i-read-json-file-using-json-java-org-json-library/

[12] http://tutorials.jenkov.com/java-date-time/parsing-formatting-dates.html

[13] https://stackoverflow.com/questions/22463062/how-to-parse-format-dates-with-localdatetime-java-8

[14] http://tutorials.jenkov.com/java-internationalization/simpledateformat.html#parsing-dates

A There are sufficient and good comments according to what is explained above.

To deliver this milestone (and also the next two) go to the home folder of the project, create a zip file with name `milestone1.zip` including

- Source code and configuration files, if any (like logback.xml)
- Results of the tests in the appendices as text files
- Plantuml files plus PNG images of the class diagrams
- JSON files
- A text file `authors.txt` with name and NIU of students responsible of this deliverable.

By adding your NIU into the authors file you state your authorship and contribution comparable to the other members of the team. Moreover, explicitly state who did what, in detail. Saying "we all did all" does not cut it.

Just one of the members uploads the zip file to Campus Virtual.

4 Milestone 2

4.1 Style

Starting now almost all the Java code has to follow the style enforced by Checkstyle, except for Javadoc rules. Occasionally you can deviate from the style, say you remove 90-95% of the issues raised by Checkstyle.

Checkstyle helps us to write the code in a certain style, but complementary to it are the "lint" type tools that check for source code *possible* bugs. Do `Analize` → `Inspect Code...` and solve the issues in the category `Java`. Not all of them, only those that you think are real improvements.

4.2 Logging

Add the file `logback.xml` to the project, put it into folder `out/production/timetraker/`. Then transform the `System.out.println` sentences you have written in the first milestone, and maybe in this second also, into proper logging messages. Remember that each class must declare a `Logger` object. We are asking yo to do the following:

- Write messages of Trace, Debug, Info and Warning levels and check they appear or not when you change the level or the root logger.
- Prepare the file logback.xml so that you can choose, by commenting out and uncomment some lines, whether to : 1) log messages from classes of the first milestones only, 2) from classes of this milestone only, 3) from both.
- Send the messages to the console and also a to an Html file.

4.3 Design by contract

Check class invariants and pre and post-conditions for all constructors and methods with parameters, according to Oracle's guidelines. Adding asserts where it is sensible is a plus, for instance to check postconditions. Class invariant must be implemented as a `private` or `protected boolean invariant()` method that should be called when necessary. Remember that you need to enable assertions for the project with `Run...` → `Edit configurations`, select a configuration and in VM edit box write `-ea`.

4.4 Search by tag

An interesting feature to be added to our time tracker is the possibility to associate zero, one or more tags (keywords) to a project or task, and then being able to search for those having a certain tag (case insensitive). For example, those in table 1. Another feature that some trackers posses is invoice generation. In order to bill customers or for other purposes, we want to know how much time we have spent on a project between two given dates, not all the time. In the future, we will want to yet add other processes like these, for instance search for projects and task with a certain name.

These three examples have something in common: they require to traverse the tree of tasks, projects and intervals and do something different depending on the process *and* the type of node. If we don't design carefully but add each of these functionalities independently, classes will loose cohesion.

Since time for the second milestone is short we will only ask you to design and implement the search by tag feature, but in a way that makes the addition of the two other features easy : no changes in the design besides the addition of some classes, project, task and interval classes remain unchanged.

Put the new classes of this new functionality into a separate package, not in `core`.

4.5 Grades

E D, C not done or doesn't reach a minimum level of quality

D Coding style plus comments, which now are compulsory: you have updated the comments, that is, written good quality comments for the new code since the first milestone.

C D plus search by tag feature. There is code to test this feature with the activities and tags of table 1.

B C plus Logging messages of Trace, Debug, Info and Warning levels. Configuration file allows to choose whether to generate logs from the first and/or second milestone classes. Messages are sent to the console and to an HTML file.

A B plus design by contract for Project and Task classes (check class invariants plus pre and post-conditions for all suitable methods).

5 Milestone 3: the user interface

Almost all commercial time trackers share two features. First is they offer versions of the same application for desktop, web, and mobile, both IOS and Android. To avoid having to reprogram the user interface for each platform, we'll use Flutter, a cross-platform library released by Google. What we have developed in the first and second milestones we will now make it accessible through a web server interface in Java made with sockets (we'll provide the code). That is, our user interface is a mobile application that sends requests to a server that keeps the tree of projects, tasks and intervals and know how to count their time, do search by tag etc. The requests are in REST protocol. To each request the web server builds an answer that reaches the user interface with information to update itself.

Second feature is that user data are not saved locally but in some backend repository. In our case this will be the JSON file for persistence that now is in the server, so this is already done.

5.1 Web server

We provide a very simple web server implemented with Java sockets so that it doesn't need external libraries. It is a simple web server built in Java with "server sockets" so we adopt a standard protocol for the format of a request known as Rest or Restful API. The user interface sends requests in this protocol to the webserver. In practice this means that the request is an http string like

```
http://localhost:8080/get_tree?1
```

where 1 is the id of the subtree root, that is, the activity from which to obtain the children. This means that each activity and interval must have a unique identifier and it has to be provided by some mechanism that ensures the uniqueness of the ids.

The answer of the webserver to the interface is a string in JSON format, like

```
{"name":"p1",
 "id":1,
 "class":"project",
 "initialDate":"2021-09-16 23:18:26",
 "finalDate":"2021-09-16 23:18:46",
 "duration":40,
 "activities":[
    {"name":"p2",
     "id":2,
     "class":"project",
     "duration":34,
     "initialDate":"2021-09-16 23:18:26",
     "finalDate":"2021-09-16 23:18:46"},
    {"name":"t3",
     "id":5,
     "class":"task",
     "active":true,
     "initialDate":"2021-09-16 23:18:40",
     "finalDate":"2021-09-16 23:18:46",
     "duration":6,
     "intervals":[],}
```

Listing 3: Base URL of our REST Api

```
final http.Client client = http.Client();
const String baseUrl = "http://10.0.2.2:8080";
//...
Future<void> start(int id) async {
  String uri = "$baseUrl/start?$id";
  final response = await client.get(uri);
  //...


   ]
}
```

The user interface decodes this string and shows the necessary information to the user.

The webserver listens to the the address `localhost:8080` for requests in the Rest API protocol. We also provide code supporting two other requests, namely, one to start a task and to stop it, which do not reply with an answer (actually, the answer string is empty or blank):

```
http://localhost:8080/start?5
http://localhost:8080/stop?5
```

where 5 is the identifier of some task to be started/stopped. The webserver is the façade or interface to the Time Tracker you have built in Java in the two first milestones, that holds the projects, tasks intervals etc. as objects and the logic for counting time etc.

5.2 Client-server communication

To test the Flutter interface, you have first to run the webserver and set some communication mechanism. The simplest one is run the client into the Android emulator, that is, a virtual machine you have installed previously. Open the AVD manager by `Help` → `Find action...` and search AVD. The webserver listens to the localhost at address 10.0.2.2, port 8080, so in the client you have to send the REST requests to this address, like in listing 3.

This is fine and the simplest way to try the client. However, optionally you can "go real" and run the app in your mobile, see appendix C.

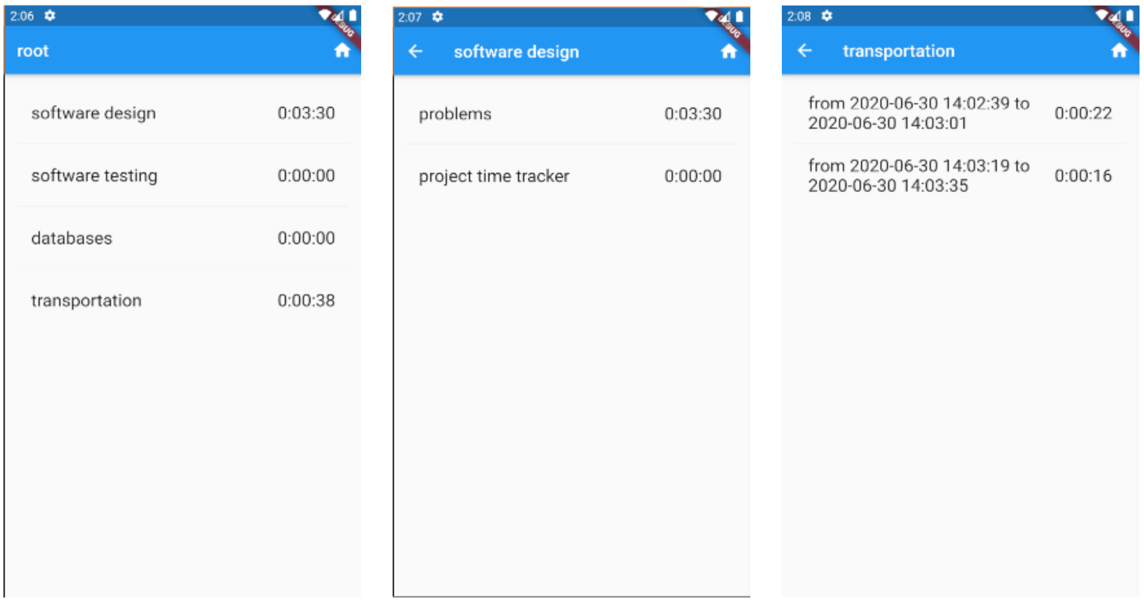
5.3 Flutter client

To design the user interface ideally one has to start doing user research, to know who are the users and what do they need. Next is building some prototypes to gain more knowledge and validate or invalidate our designs through user testing sessions. As time passes our designs/prototypes become more complete and maybe high-level also. At the end, we already know which user interface we have to build, that is, to program. Suppose we are at this point now.

The implementation we intend to do has two sources of difficulty: 1) Flutter (and Dart, the programming language on which it is based) is probably new to you, 2) maybe also the client-webserver architecture and how make the two parts talk to each other through a JSON protocol and REST. Since time is short, we provide you the source code of a webserver and a Flutter interface, so you can learn by example. The webserver part most probably you'll want to keep it unchanged, apart from adapt it to the code you have developed in the previous milestones. The Flutter part definitely needs to change because the user interface lacks many things and others should be done differently.

The Flutter user interface we provide is shown in figure 1 implements the following behaviour:

- upon launching it asks the server for the list of root children (tasks and projects) and displays it in the home page
- single click to one element of this list enters into it and we see its children, which are more tasks/projects if it's a project, or intervals if a task
- the up button (←) brings you back to the previous "screen" without leaving the application the home icon to the first screen
- long press on a task starts counting the time (and creates a new interval for it) if it wasn't active, stops it if already active
- the project/tasks and the intervals screens are updated every 6 seconds



software design	0:03:30
software testing	0:00:00
databases	0:00:00
transportation	0:00:38

problems	0:03:30
project time tracker	0:00:00

from 2020-06-30 14:02:39 to 2020-06-30 14:03:01	0:00:22
from 2020-06-30 14:03:19 to 2020-06-30 14:03:35	0:00:16

Figure 1: Simple user interface showing activities (left, middle) and intervals of a task (right).

There are many possible improvements. You don't have to stick to this interface, if you wish you can discard it and make yours from scratch. Here is a list of wrong things in ours:

1. What is a project or a task is unknown.
2. The title “root” is misleading/weird for the end user.
3. No way to add a new project or task.
4. Long-press to start/stop a task is wrong because this action is to select an item (in Android).
5. One can not see the details of a project/task, that is, initial and final dates. But if shown along with the name, probably it would be difficult to read.
6. Intervals info is difficult to read, for instance to check whether we were working on a task at a certain time. Also, what/if an interval is active.
7. No way to search by tag. Even if you have not implemented this feature, you have to build the interface for it (that returns a wrong, fake result).
8. No way to localize the interface (essentially, messages and formats of time and date).
9. Surely there’s a better way to show the intervals of a task than a simple list. Even tasks and projects.
10. The order of items in the lists of projects/tasks and intervals is the right one ?
11. Add filter by tag
12. Obtain the total time within a given time window and also the amount to bill, given a certain cost per hour in some currency.
13. A mechanism like list the tasks most recently accessed is a useful shortcut.

5.4 Grades

E Problems 1, 2 or 3 have not been addressed

D Solve the basic problems 1, 2, 3

C D plus Fins a good alternative to 4, 5, 6

B C plus solve 7-10

A B plus 11-13 and any other substantial improvement or new feature that you have previously discussed with your teaching assistant. Look for ideas in the commercial time trackers mentioned in page 2.

A Sample tree

You can build the following tree of projects and tasks in order to then pass the tests of each milestone:

- at the top level, projects *software design*, *software testing*, *databases* and task *transportation*
- under *software design*, projects *problems* and *project time tracker*
- under *problems*, tasks *first list* and *second list*
- under *project time tracker*, tasks *read handout* and *first milestone*

To test the search by tag in the second milestone, associate the tags of table 1 to them (literally, mind upper and lowercase):

Project, Task	Tags
software design	java, flutter
software testing	c++, Java, python
databases	SQL, python, C++
first list	java
second list	Dart
first milestone	Java, IntelliJ

Table 1: Tags

B Test of counting time

Set the clock period to 2 seconds, which is the time unit we will use but should be configurable. We will make date-times consistent with durations by doing this: each time the user (the code that simulates it) starts a task and later receives a signal from the clock, since the time unit is 2 seconds we will consider that already 2 seconds have passed for this task. Consequently we have to set the new interval duration to 2 seconds, the initial date-time to 2 seconds before the first signal received, and the final date-time to the clock time.

Write code to simulate the following user actions that, if you insert the corresponding print sentences in the right places to show only those nodes that change each time, should produce the next output:

1. start task *transportation*, wait 4 seconds and then stop it
2. wait 2 seconds
3. start task *first list*, wait 6 seconds
4. start task *second list* and wait 4 seconds
5. stop *first list*
6. wait 2 seconds and then stop *second list*
7. wait 2 seconds
8. start *transportation*, wait 4 seconds and then stop it

To simulate the user waits for some time use `Thread.sleep()` within a try-catch block.

```
                                initial date          final date        duration
start test
transportation starts
interval:                    2021-09-22 16:04:56    2021-09-22 16:04:58    2
activity:    transportation  2021-09-22 16:04:56    2021-09-22 16:04:58    2
activity:    root            2021-09-22 16:04:56    2021-09-22 16:04:58    2
interval:                    2021-09-22 16:04:56    2021-09-22 16:05:00    4
activity:    transportation  2021-09-22 16:04:56    2021-09-22 16:05:00    4
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:00    4
interval:                    2021-09-22 16:04:56    2021-09-22 16:05:02    6
activity:    transportation  2021-09-22 16:04:56    2021-09-22 16:05:02    6
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:02    6
transportation stops
first list starts
interval:                    2021-09-22 16:05:04    2021-09-22 16:05:06    2
activity:    first list      2021-09-22 16:05:04    2021-09-22 16:05:06    2
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:06    2
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:06    2
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:06    8
interval:                    2021-09-22 16:05:04    2021-09-22 16:05:08    4
activity:    first list      2021-09-22 16:05:04    2021-09-22 16:05:08    4
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:08    4
```

```
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:08    4
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:08    10
interval:                    2021-09-22 16:05:04    2021-09-22 16:05:10    6
activity:    first list      2021-09-22 16:05:04    2021-09-22 16:05:10    6
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:10    6
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:10    6
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:10    12
second list starts
interval:                    2021-09-22 16:05:10    2021-09-22 16:05:12    2
activity:    second list     2021-09-22 16:05:10    2021-09-22 16:05:12    2
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:12    8
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:12    8
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:12    14
interval:                    2021-09-22 16:05:04    2021-09-22 16:05:12    8
activity:    first list      2021-09-22 16:05:04    2021-09-22 16:05:12    8
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:12    10
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:12    10
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:12    16
interval:                    2021-09-22 16:05:10    2021-09-22 16:05:14    4
activity:    second list     2021-09-22 16:05:10    2021-09-22 16:05:14    4
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:14    12
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:14    12
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:14    18
interval:                    2021-09-22 16:05:04    2021-09-22 16:05:14    10
activity:    first list      2021-09-22 16:05:04    2021-09-22 16:05:14    10
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:14    14
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:14    14
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:14    20
first list stops
interval:                    2021-09-22 16:05:10    2021-09-22 16:05:16    6
activity:    second list     2021-09-22 16:05:10    2021-09-22 16:05:16    6
activity:    problems        2021-09-22 16:05:04    2021-09-22 16:05:16    16
activity:    software design 2021-09-22 16:05:04    2021-09-22 16:05:16    16
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:16    22
second list stops
transportation starts
interval:                    2021-09-22 16:05:18    2021-09-22 16:05:20    2
activity:    transportation  2021-09-22 16:04:56    2021-09-22 16:05:20    8
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:20    24
interval:                    2021-09-22 16:05:18    2021-09-22 16:05:22    4
activity:    transportation  2021-09-22 16:04:56    2021-09-22 16:05:22    10
activity:    root            2021-09-22 16:04:56    2021-09-22 16:05:22    26
transportation stops
end of test
```

C Tunneling

To run the Flutter interface in a mobile we need a third party tunneling application, a program that redirects URL requests to a real internet address to the localhost. `ngrok` is program that connects localhost:8080 to some temporal URL that `ngrok.com` provides you for free: run `ngrok http 8080` and set the address it prints as the base URL:

```
const String baseUrl = "http://f51fb19f.ngrok.io";
```

This address is valid only once, next time you run the client you will need to obtain a new one. In Linux I've installed ngrok with `sudo npm install ngrok -g`. You can download ngrok for Linux, Windows and Mac from `https://ngrok.com/`. More on this here.