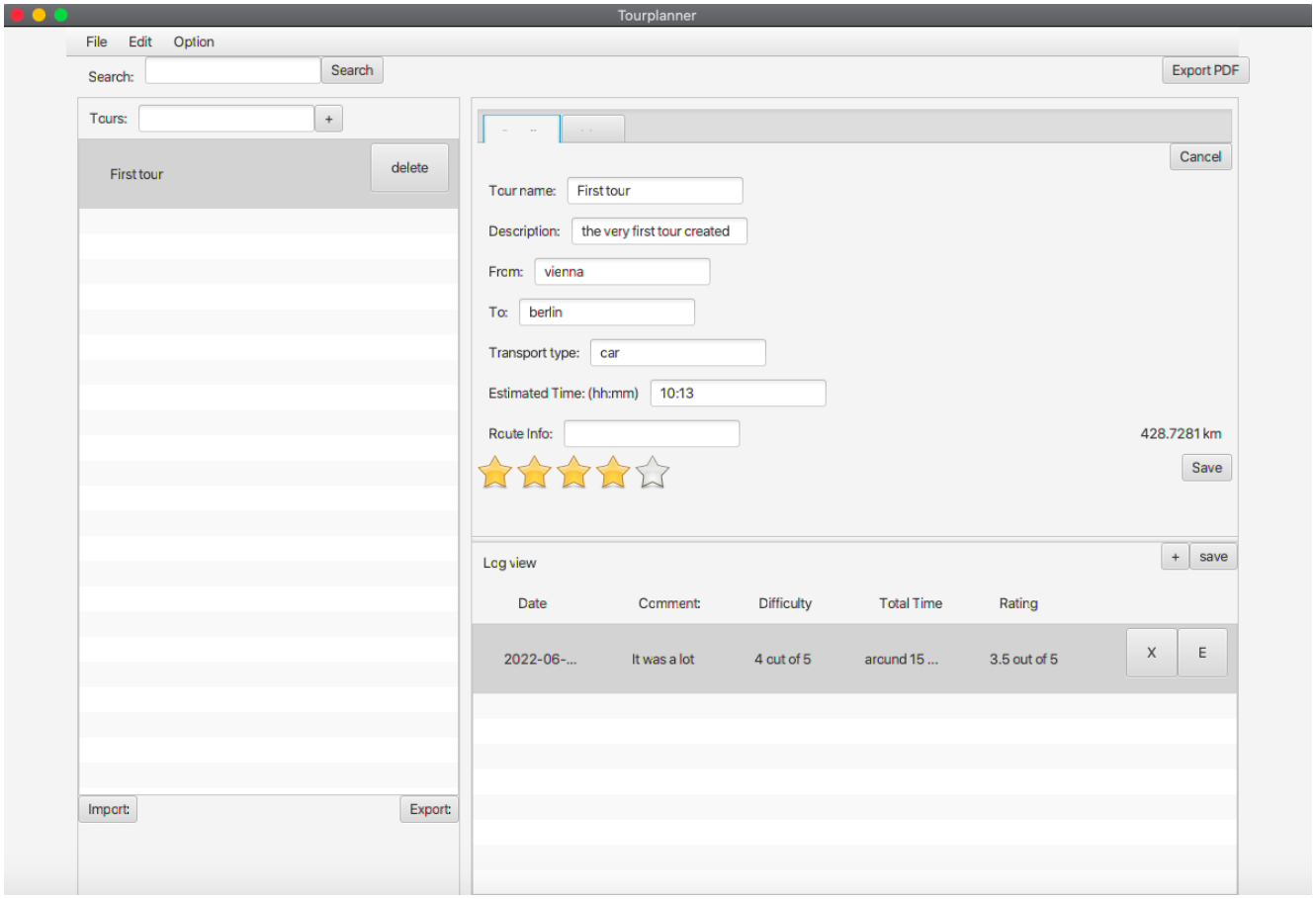
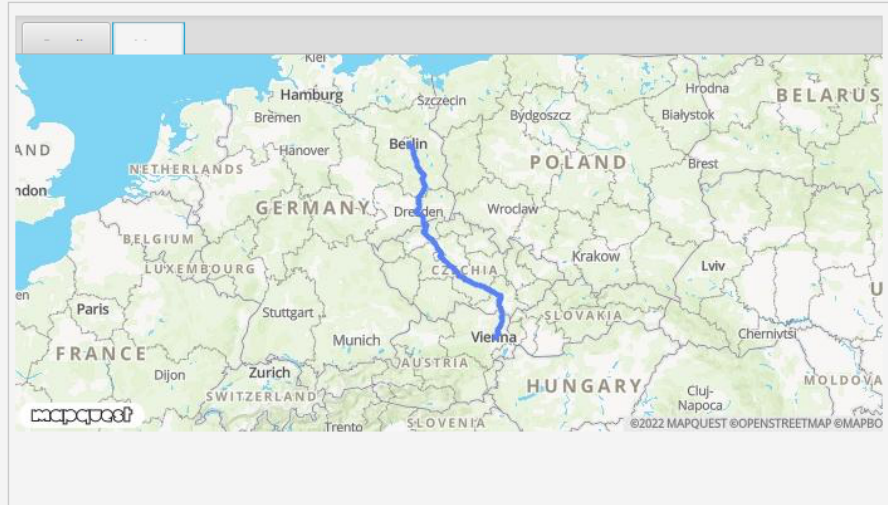
TourPlanner

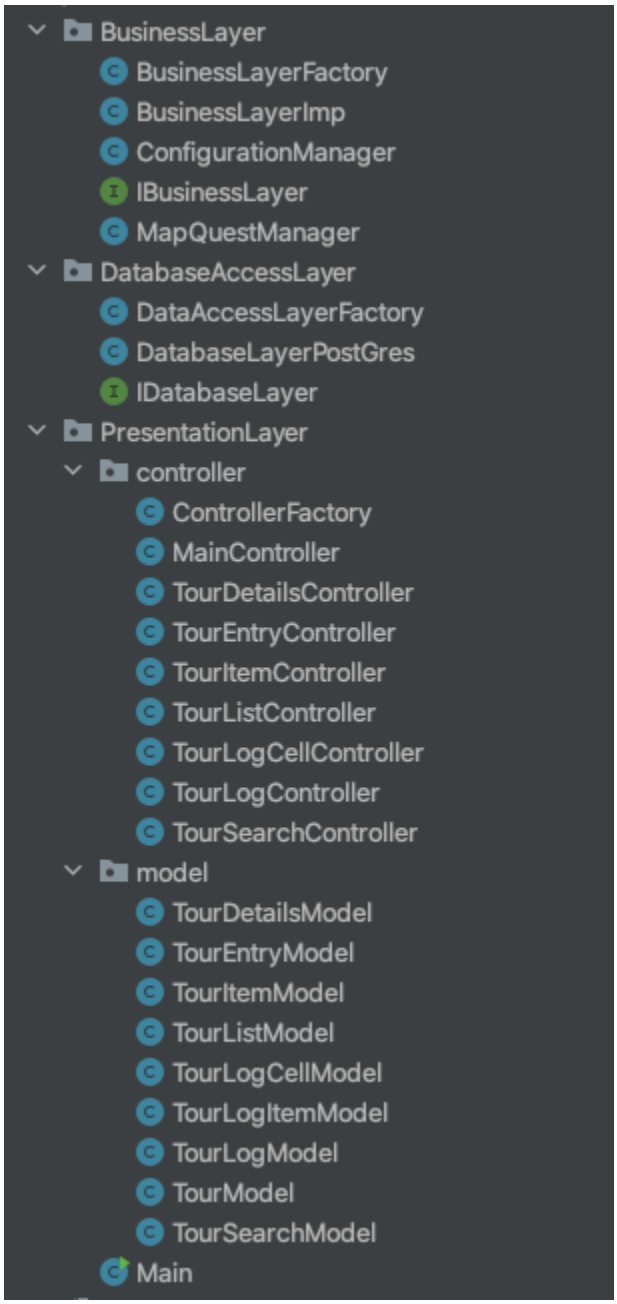




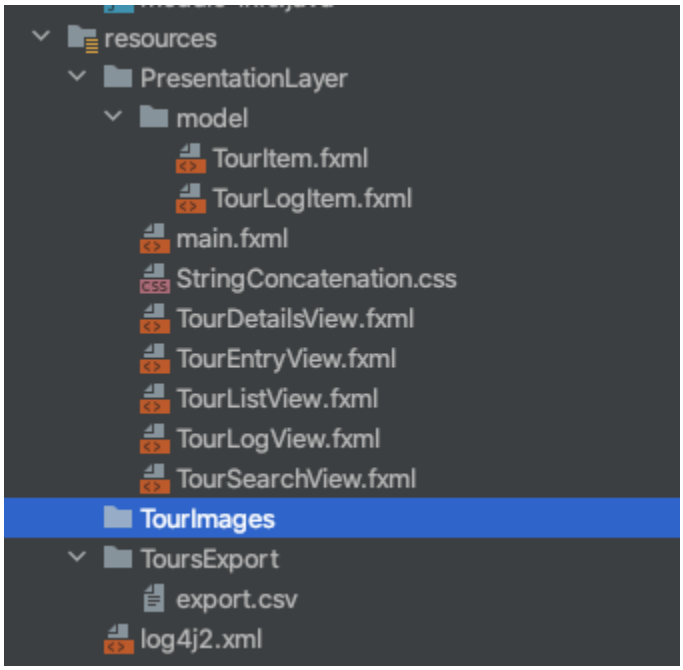
Arber Bajraktari
Ledio Rejzi

Structure

The structure of the project was decided to be as shown in the picture below. Each class is assigned in the Folder which represents the layered architecture. This means that the Classes under the Presentation Layer (Models and controller) are under the PresentationLayer folder. The same goes for BusinessLayer and DataAccessLayer.
The reason for this was because for us it makes more sense to use a separation based on the functionalities that the classes represent and not separate them in packages for each individual functionality (like manageTours, manageLogs etc.).



The same goes for the Resources folder, where the same structure ought to be followed.
TourImages is the folder where the image of the map for the tours will be saved.
TourExport is the folder where the Export of the logs will be saved.



Design patterns used

The first design pattern would be the Observable pattern.

Observable Pattern

This is the pattern that we used for the ListView. Instead of notifying the Items of the list for every change whatsoever, we used the observable pattern, where it works with a “subscription”. They are all subscribed, and for every change they will be notified.

Factory

The factory is another pattern used for this project. One part of the project where this was used is the Controller factory. We created the models for each respective factory in a class called ControllerFactory. And there, once the Controller Class is instantiated, the subclasses choose which Controller to instantiate. And for this controller, the Model defined is passed on as a parameter. So, the subclasses decide which type of the Instance is going to be created, in this case which type of Controller.

```
public Object create(Class controllerClass) throws Exception {
    //for each controller, assign the Model
    if (controllerClass == TourEntryController.class) {
        return new TourEntryController(this.tourEntryModel);
    }else if(controllerClass == TourListController.class){
        return new TourListController(this.tourListModel, this.tourDetailsModel, this.tourLogModel);
    }else if(controllerClass == TourLogController.class){
        return new TourLogController(this.tourLogModel);
    }else if(controllerClass == TourDetailsController.class){
        return new TourDetailsController(this.tourDetailsModel);
    }else if(controllerClass == MainController.class){
        return new MainController(this.tourListModel);
    }else if(controllerClass == TourSearchController.class){
        return new TourSearchController(this.tourSearchModel);
    }else {
        throw new Exception("Controller not supported " + controllerClass.getName());
    }
}
```

Arber Bajraktari
Ledio Rejzi

Singleton

Another pattern used is the singleton. What this does is that if we need access to DB for example, instead of creating a DB object in different classes, we use the singleton which allows us to use the same object.

```
private static IDatabaseLayer databaseLayer;

//Singleton for the Database Layer
public static IDatabaseLayer getDatabase(){
    if (databaseLayer == null) {
        databaseLayer = new DatabaseLayerPostGres();
    }
    return databaseLayer;
}
```

We create an object of the DB, and then when we want to have access to the DB from different classes we call **getDatabase()**, which gives us the object from this class or a new object if it is null.
This way, we are always using the same object to access the DB.

Configuration

We used a configuration file to store the following values.

```
PostgresSqlConnectionString=jdbc:postgresql://localhost:5432/postgres
user=postgres
pwd=myPassword
FileAccessStoragePath=src/main/resources/TourImages/
CsvAccessStoragePath=src/main/resources/ToursExport/
MapKey=wrq6qJ05GD4w8ZMjbTsZ25C5matuLpNw
```

This helps a lot if there is a need afterwards to change the MapKey (key of the API) or the path of where the files are stored.

We also used a configuration file to configure the Log4J. This is XML, as it can be seen below, and we specify the level (which is INFO), the name of the file which will be saved and how will be the format of the infos saved.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO" >
    <Appenders>
        <Console name="ConsoleAppender" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>
        <File name="FileAppender" fileName="Logs/Tours-${date:yyyyMMdd}.log" immediateFlush="false" append="true">
            <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="debug">
            <AppenderRef ref="ConsoleAppender" />
            <AppenderRef ref="FileAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

Arber Bajraktari
Ledio Rejzi

Lessons learned

It is better to at least start with a design pattern while the teacher is explaining it in class. We thought it would be better to start with it after we understood it, but it turns out that the best way to do it is “together” with the teacher.

We tried doing the development of the Project with branches, which was the first time we tried that. We are happy to say that that was a success. The only “problems” would be the merge conflicts, which were extremely easy to fix because the tasks which we took over were always different from one another. This was a lesson learned and a helpful experience which we will keep on using.

The other lesson would be to not underestimate how much time something takes. Especially something you know. Also when you know how something is done and in your head it seems like you can finish it really quickly, that is almost never the case. There will always come something up which will prolong your solution. That is why it is better to start easy, so that when you estimate your timing right, then you just have more time for the other task.

Git Link

Something new we used with git this time was the tags. We used the master as the production environment. And for each new version of master, we assigned a new tag.

Link: https://github.com/ArberBajraktari/TourPlanner

Arber Bajraktari
Ledio Rejzi