Protocol – Tour Planner

Planning and Architecture

User Interface and MVVM

We started the development of this application by creating the UI first. For this, we developed a rough overview of the app itself in the Scene Builder and then modified some minor things in the FXML-Markup "by hand". After that, we started to implement the respective Controller- and ViewModel-Classes that deal with user actions like pressing a button or other forms of input. In addition to that, we used the MVVM pattern for this section and therefore divided the whole logic of this part into different packages:

- *views*
- *viewmodels*
- *models*

The *views* package consists of the Controller classes and they are the direct interface to the FXML files. In the *viewmodels* package are the ViewModel classes that deal with the specific properties, data binding stuff etc. Package *models* consists of just the classes *Tour* and *TourLog* which specify how these objects are structured and what attributes they have.

Business Layer

After this layer was done, we continued with the Business Layer. This one consists of three sub-packages and two classes. The packages are:

- *events*
- *mapquestapi*
- *reports*

Package *events* deals with the Observer pattern that we also implemented in this application. It is very useful since it enforces better communication between different UI components. Package *mapquestapi* deals with the HTTP requests for tour information like distance or estimated duration and also in order to retrieve a map of the tour in form of an image which is saved locally. And lastly, *reports* implements the report generation library that is needed to generate single-tour and statistical summary reports.

Our business layer also contains the classes *AppLogic* and *ConfigurationManager*. AppLogic was implemented using the Factory pattern which promotes loose-coupling (→ *IAppLogic* and *AppLogicFactory*). *ConfigurationManager* is a helper class that retrieves certain information like the database credentials from the config file that is stored in the *resources* section of the project.

Data Access Layer

For the database-oriented side of the project we implemented our DAL the following way:

- *common*
- *dao*
- *fileaccess*
- *postgres*

When looking at the *common* package it can be seen that this layer is also built using the factory pattern. It consists of a *DALFactory* which is used by the sub-classes to choose which type of object needs to be created. *IDatabase* and *IFileAccess* are just simple interfaces which declare the must-have methods that the inheriting classes need to implement. The *dao* package also only consists of interfaces for the data access objects of tours and tour logs. The class *FileAccess* in the package *fileaccess* then implements the interface from the *common* package. In *common* the interfaces of *IDatabase*, *ITourDAO* and *ITourLogDAO* are implemented.

Functionality

Regarding the functionality of the application it can be explained very easily with an example. Assuming a user starts the program and he/she is presented with the main view of the Tour Planner. If the user now clicks the button for creating a new tour a method in the Controller is bound to that button by the *onAction* attribute. Then, another method in the *MainViewModel* is called which opens a new window where the user has the opportunity to put in the data of the new tour. This data is then transformed into an instance of the class *Tour* and handed over to *AppLogic* which then submits this data to either TourDAO or TourLogDAO (depending on the task of course). From here on, this data is then transformed into SQL queries with the corresponding parameters and finally handed over to the *Database* class which then lastly performs the requested operations on the database.

UI → Controller → ViewModel → AppLogic → DAO → Database

Unit Tests

For the code testing aspect of the project we ended up with 23 unit tests and an estimated code coverage of about 70 to 80%. We prevailingly tested the *AppLogic* as well as *TourDAO* and *TourLogDAO* since these classes contain the most important and critical actions/operations of the application.

Our unit tests mostly consist of testing the CRUD/SQL operations regarding tours and tour logs. For these we used dummy objects and checked if the methods return the expected values.

Besides that, we also made sure to test the HTTP requests to the MapQuest API because this part of the project plays an important role in retrieving information about the tour and the tour map.

Lastly, we also tested the *ConfigurationManager* in order to check the correctness of retrieving data from the config file.

Failures of the project

The following features were unfortunately not implemented due to timewise reasons:

- Full-text search
- Import/Export of tour data

Mandatory Unique Feature

Our mandatory unique feature consists of a WebView window that can be accessed under the Help tab and it shows the GitHub repository of this project.

Time Spent

- Planning: ~20 hours

- Implementation
    - UI and MVVM: 11 hours
    - BL: 14 hours
    - DAL: 10 hours
    - Reporting: 5 hours
    - Logging: 2 hours
    - Testing: 3 hours

- Protocol: 1 hour

Total: ~66 hours

Technical Information

Libraries

For this project we used various libraries. This is due to the fact that the task specification required some features (e.g. logging, report generation) which were not offered/supported by the basic Java modules.

We used:

- **iText7** (report generation)
- **Jackson Databind/Core/Annotations** (JSON serialization)
- **log4j** (logging)
- **sl4j-api & slf4j-nop** (additionally required for iText7)
- **postgresql-42.3.1** (database functionality)
- **JUnit** (unit testing)
- **javafx.scene.web** (for mandatory unique feature)

SQL File of Database

https://ufile.io/k1fm27vt

Link to Git

https://github.com/zpc912/tourplanner_swen2