FH University of Applied Sciences TECHNIKUM WIEN

Informatik, BA
Software Engineering 2
Practical Part
Tour Planner
Desktop Application
C# (WPF)

Tobias Meindl
Lukas Varga

Table of contents

1. Technical Steps

Designs

MVVM Pattern:
The interface between our WPF UI and the business logic is realized via the MVVM pattern. This separates the xaml markup from the logic behind it and thus makes it easily replaceable.

The mediator pattern:
In order to reduce coupling between the different view models we make use of the mediator pattern. The view models that interact with the UI get administered by a mediator who bundles all of the logic behind the UI and also talks to the business layer. So if an action on one view results in another action in another view the view talks to the mediator (Main View Model in our case) and the mediator then tells the other view model it needs to change.

Command and Observer:
In order to realize the communication from the UI to the view model in turn to the main view model we make use of the command and the observer pattern. A command is executed by the UI which in turn prompts a change event and triggers the observer (in our case the Main View Model).

Failures

In the PL we tried to handle all ViewModel actions for the MainViewModel in one file, but after adding more and more code the whole file became unable to read properly which took too much time and led to error-prone programing.

Validating input has been hard to adapt in the late stage of the process because of implementing multiple features which also led to trouble with generating reports for specific tours resulting in non-pdf files.

Using the wpf for this project caused us to miss a classic program file which uses a console to write information and "execute" the program. Therefore, no debugging or displaying error messages is possible in the classic style. We were advised to use integration tests for fixing errors and debugging, but the effort was in no proportion to the outcome that we had to look for other ways of displaying errors.

Selected Solutions

Database:
For our postgres database we choose to run it in a docker container. We wrote a docker file that sets the image up to our needs. The image is also saved on dockerhub to make sharing more easy. We then wrote a .sql and two .bat files. In the sql file we specified the way we want our database to look like. In the first .bat files are instructions to pull the docker image and run it. In the second .bat file the sql file is copied and executed in the container to update the database.

MainViewModel:
To get a hold on the amount of code lines we used the partial class for the first time, where you can split one class into multiple files with full access to all properties and methods. This way we got 8 files for the MainViewModel but this helped in the process of orientation and quicker coding.

Validating:
Especially for the specific tour report we needed to use a regex validation, because the name of a report is combined with the tour name, which is for the user free to choose, and can cause problems if characters are used which are forbidden for a file to include in the name. Therefore we scan for specific characters and replace them with a "_" to avoid broken files because of incorrect naming schemes.

Debugging:
The system offers a popup called MessageBox which has a method .Show to display text messages that can be used to display custom messages or even error messages in full extent. This was also used to display messages for validating input (if the user uses incorrect data or to confirm certain actions like generating reports and so forth).

2. Application Features

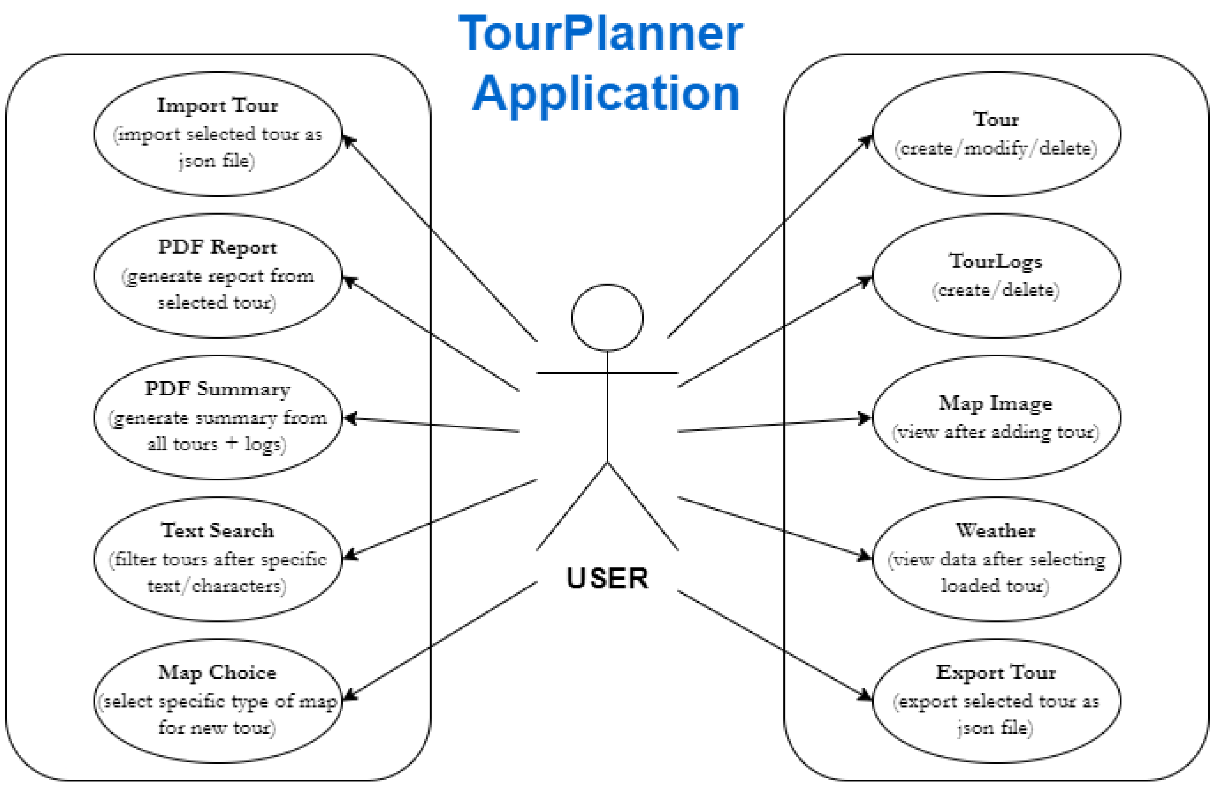
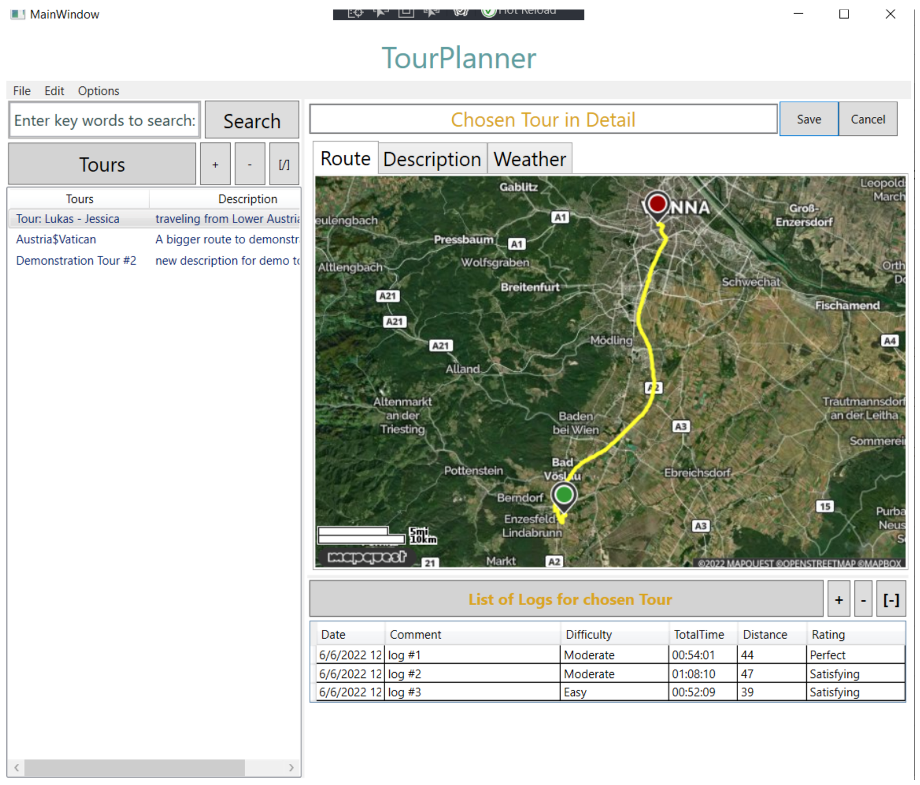

TourPlanner Application
Import Tour (import selected tour as json file)
PDF Report (generate report from selected tour)
PDF Summary (generate summary from all tours + logs)
Text Search (filter tours after specific text/characters)
Map Choice (select specific type of map for new tour)
USER
Tour (create/modify/delete)
TourLogs (create/delete)
Map Image (view after adding tour)
Weather (view data after selecting loaded tour)
Export Tour (export selected tour as json file)

Figure 2.1 shows the Use-Case-Diagram to display the application features available for every user (because there is no login system to differ users)

3. UI-Flow

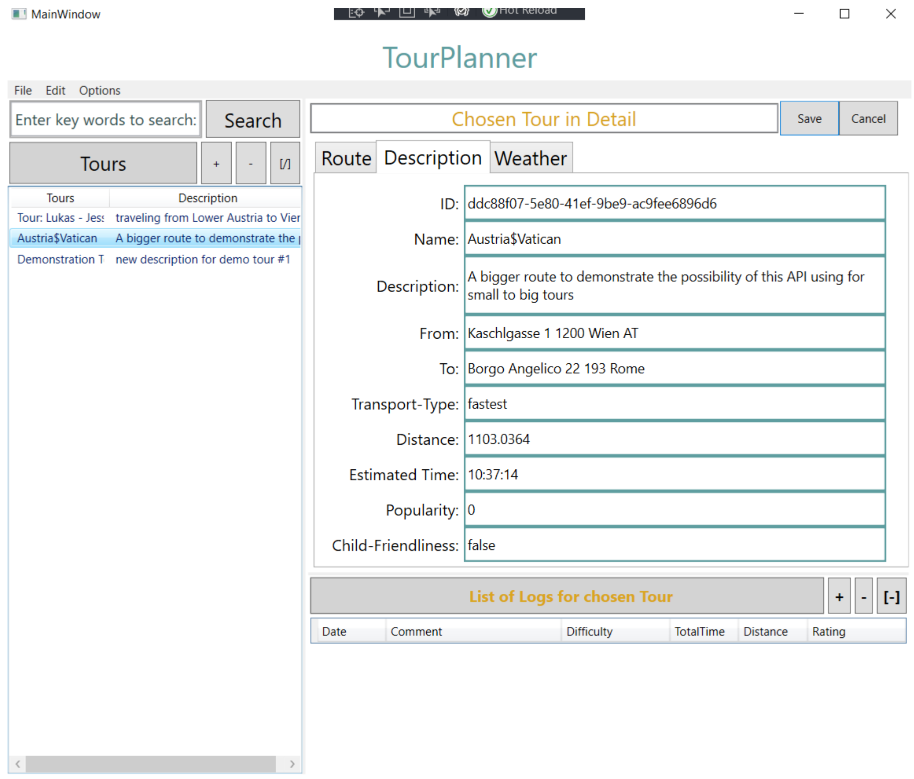
Following figures show the different UI parts to use of the program including Unique Feature.



Tours	Description
Tour: Lukas - Jessica	traveling from Lower Austri
Austria$Vatican	A bigger route to demonstr
Demonstration Tour #2	new description for demo t

Date	Comment	Difficulty	TotalTime	Distance	Rating
6/6/2022 12	log #1	Moderate	00:54:01	44	Perfect
6/6/2022 12	log #2	Moderate	01:08:10	47	Satisfying
6/6/2022 12	log #3	Easy	00:52:09	39	Satisfying

figure 3.1 (shows tour overview with map image)

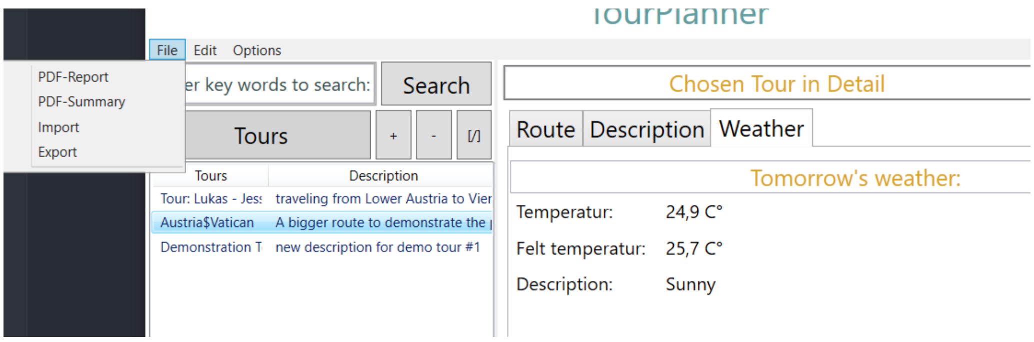


Tours	Description
Tour: Lukas - Jes	traveling from Lower Austria to Vier
Austria$Vatican	A bigger route to demonstrate the
Demonstration T	new description for demo tour #1

ID:	ddc88f07-5e80-41ef-9be9-ac9fee6896d6
Name:	Austria$Vatican
Description:	A bigger route to demonstrate the possibility of this API using for small to big tours
From:	Kaschlgasse 1 1200 Wien AT
To:	Borgo Angelico 22 193 Rome
Transport-Type:	fastest
Distance:	1103.0364
Estimated Time:	10:37:14
Popularity:	0
Child-Friendliness:	false

Date	Comment	Difficulty	TotalTime	Distance	Rating

figure 3.2 (shows tour details from attributes)

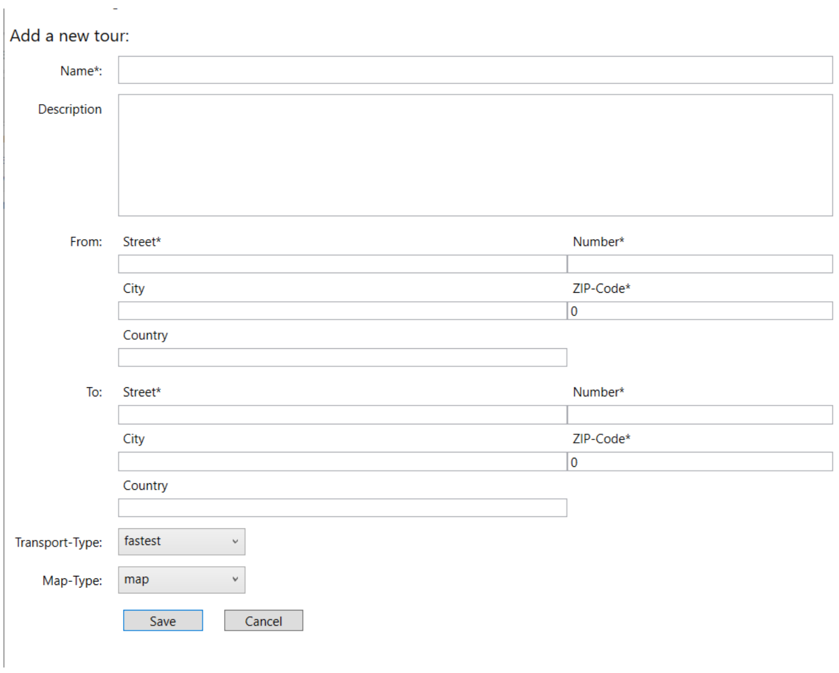


Tours	Description
Tour: Lukas - Jes:	traveling from Lower Austria to Vier
Austria$Vatican	A bigger route to demonstrate the
Demonstration T	new description for demo tour #1

Tomorrow's weather:

Temperatur:	24,9 C°
Felt temperatur:	25,7 C°
Description:	Sunny

figure 3.3 (shows the unique feature calling additional weather data from regarding api and File button for the additional features of report/summary/import/export)


Add a new tour:

Name*:

Description

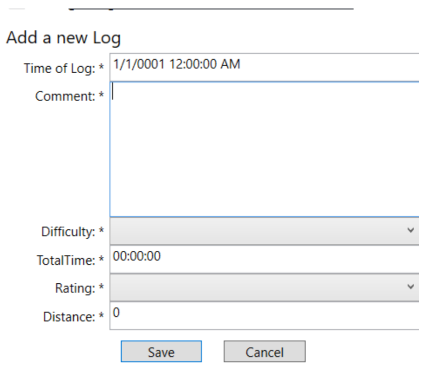
From: Street* Number*
City ZIP-Code* 0
Country

To: Street* Number*
City ZIP-Code* 0
Country

Transport-Type: fastest

Map-Type: map

figure 3.4 (shows the tour adding window with text boxes for free writing, dropdowns to choose an pre set option and save/cancel button)


Add a new Log

Time of Log: * 1/1/0001 12:00:00 AM
Comment: *
Difficulty: *
TotalTime: * 00:00:00
Rating: *
Distance: * 0

figure 3.5 (shows the log adding window with text boxes, dropdowns and save/cancel button for user-input => time of log is auto generated and will compute the correct timestamp in db and for the log view, but does throw the wrong data into the addlog window which we don't know how to change)

4. Software Architecture

Layering

The project uses different layers to structure the code and its communication. For the communication it is meant that each layer uses their own code or the code in the layer below. We differ between Presentation-Layer (PL), Business-Layer (BL) and Data-Access-Layer (DAL). Additionally we used class libraries to add our Models. Originally there was the plan to use multiple class libraries for extending our Business Layer, but ended up adding everything in need to the BL module.

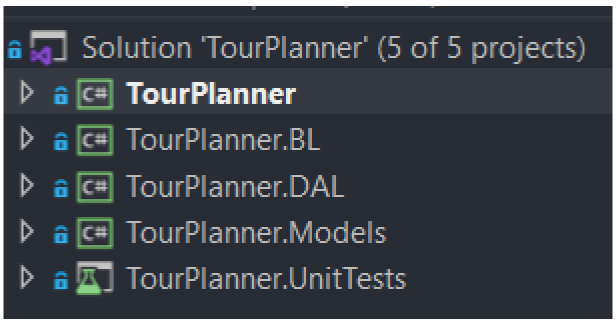

Solution 'TourPlanner' (5 of 5 projects)
TourPlanner
TourPlanner.BL
TourPlanner.DAL
TourPlanner.Models
TourPlanner.UnitTests

figure 4.1 (shows Layering and differing into modules + additional project for testing)

Additional NuGet extensions/libraries in use: text7, log4net, Npgsql, NUnit, Drawing.Common, Newtonsoft.Json

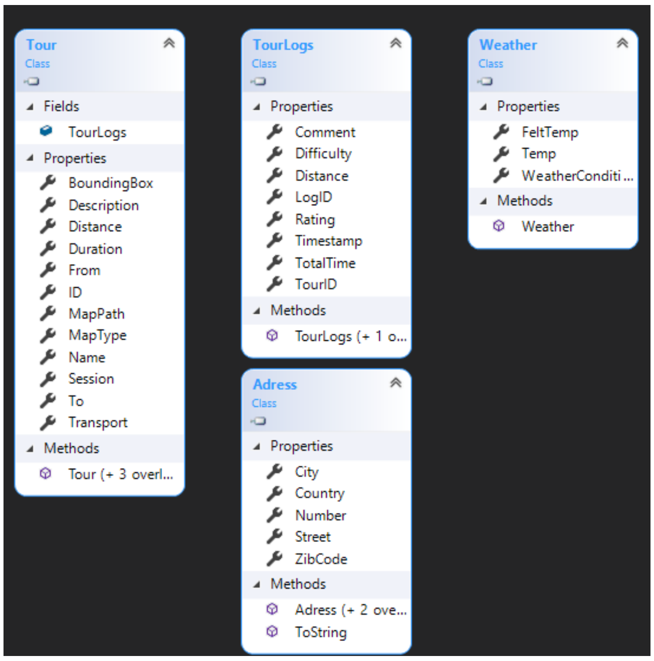
For the PL we used the MVVM pattern and defined our Abstract "BaseViewModel" as parent to inherit for all ViewModels (Main, Sub). The Utility "RelayCommand" helps to use the commands and the different View Windows for the UI have been divided to reusable components.

The BL does the main work for the logic which communicates with the ViewModels in PL and the database interactions in DAL. All the features are layered within as follows: Logging, MapQuestAPI, PDFGeneration, TextSearch, WeatherAPI (unique feature) as well as our Controllers (Services) for Input/Output, the tours, tourlogs and in general.

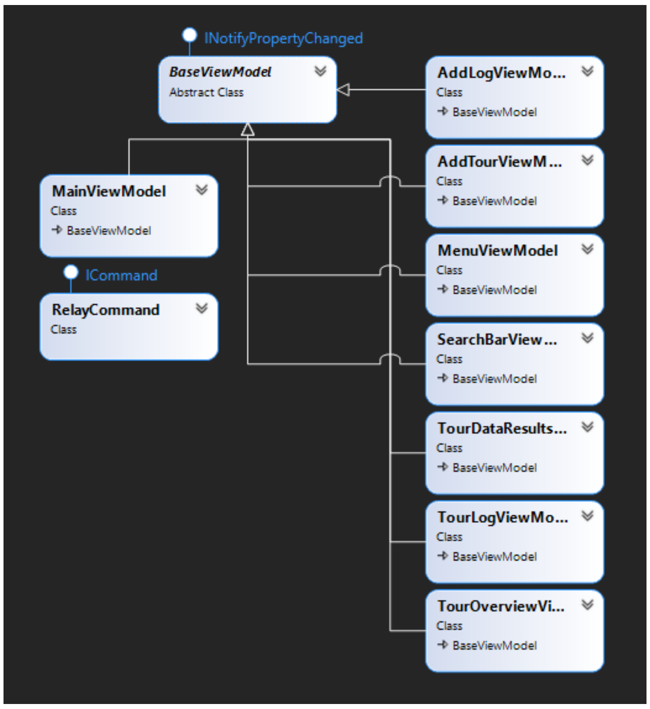
Additionally to the BL we used a class library to list our models which contains the tour and tourlogs model, an customized Address model to manage the addresses for our requests and db storing as well as a Weather model and all our enums.

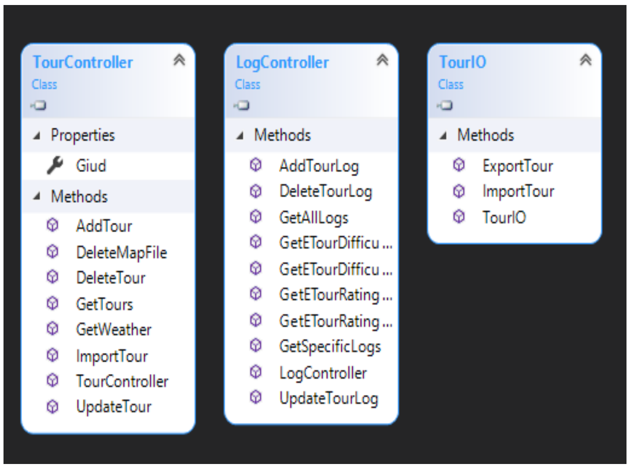
The DAL manages the connection by the DBConnection class, using input from a config file to not display connection data openly. Furthermore there are two classes to manage the database communication for tours and tourlogs, controlled by the Controllers from the BL.

Class Diagram



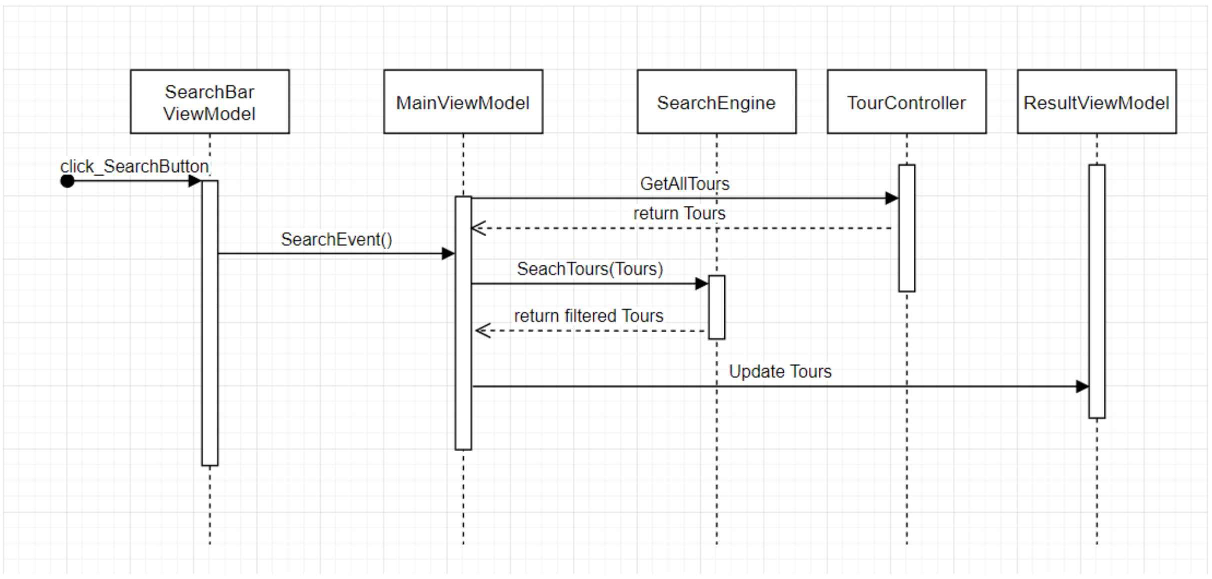
Since there are no complicated inheritance structures we don't feel the need to present every single class as a diagramm. Most of the complexity lies in the interplay of the view models. Most of the BL and DAL make use of simple static functions.





Sequence Diagram

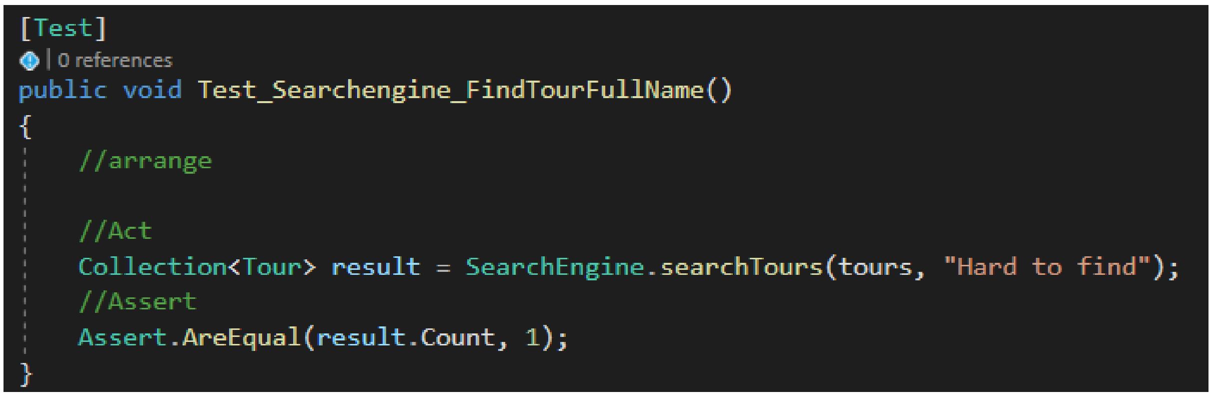
Sequence Diagram of our tour search.



Summary: When the search button is clicked, the View Model raises a Search-Event. Upon observing this event the Main View Model calls the search engine with the data of all tours. The engine returns the filtered tours, with which the Main View Model then updates the result view.

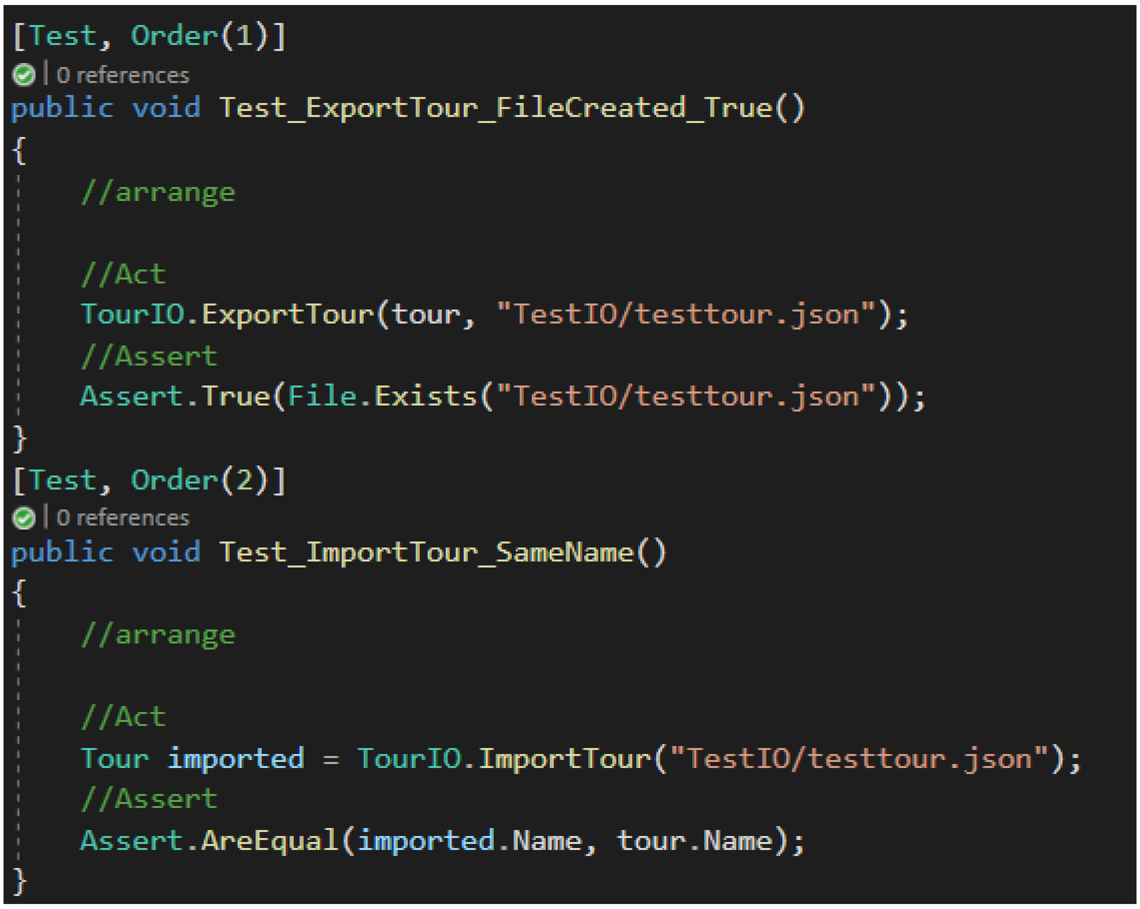
5. Unit-Tests

The framework used was NUnit. Since the majority of the code in this project is code for the UI, which in our opinion was very hard to test, our main focus when writing unit tests was on the business layer. We focused on the helper functions that for example calculate values of our tours, the import and export function as well as the search engine.



```
[Test]
| 0 references
public void Test_Searchengine_FindTourFullName()
{
    //arrange

    //Act
    Collection<Tour> result = SearchEngine.searchTours(tours, "Hard to find");
    //Assert
    Assert.AreEqual(result.Count, 1);
}
```



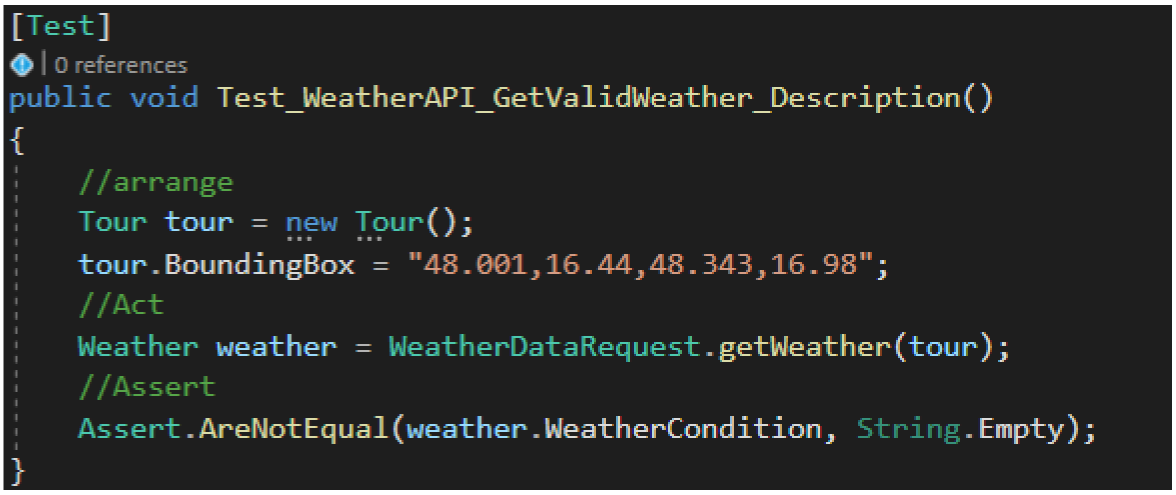
```
[Test, Order(1)]
| 0 references
public void Test_ExportTour_FileCreated_True()
{
    //arrange

    //Act
    TourIO.ExportTour(tour, "TestIO/testtour.json");
    //Assert
    Assert.True(File.Exists("TestIO/testtour.json"));
}
[Test, Order(2)]
| 0 references
public void Test_ImportTour_SameName()
{
    //arrange

    //Act
    Tour imported = TourIO.ImportTour("TestIO/testtour.json");
    //Assert
    Assert.AreEqual(imported.Name, tour.Name);
}
```

To test the import/export we exported a tour, checked if the file was created, imported it again and compared if the data was still the same as prior to the export. We also made use of the [OneTimeSetUp] tag to create a folder for the exported file.

We also wrote some integration tests for the access to the weather API.

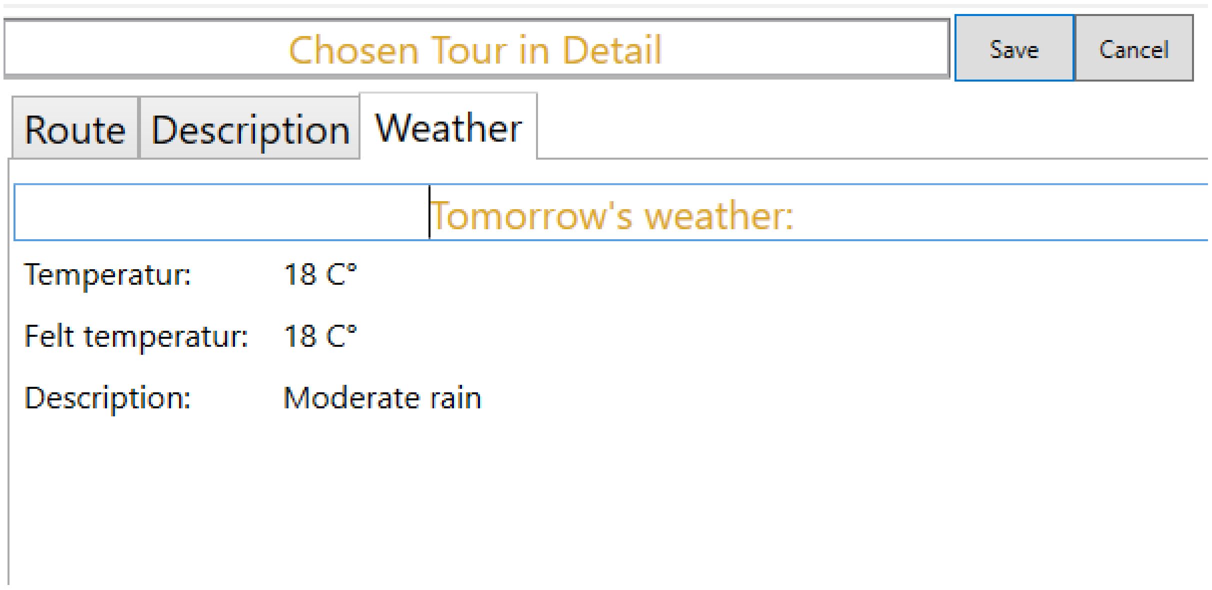


```
[Test]
0 references
public void Test_WeatherAPI_GetValidWeather_Description()
{
    //arrange
    Tour tour = new Tour();
    tour.BoundingBox = "48.001,16.44,48.343,16.98";
    //Act
    Weather weather = WeatherDataRequest.getWeather(tour);
    //Assert
    Assert.AreNotEqual(weather.WeatherCondition, String.Empty);
}
```

One problem we encountered was testing the logging. We didn't know a way to read the log file while the program was running since it is reserved by the logger. So we just tested the creation of the log file but not its content.
end.

6. Unique Feature

For our unique feature we choose to add a weather function to our tour. The function displays the current weather and the temperature at the start coordinates. For this we use https://www.weatherapi.com/ which is free to use. We just make a request with the coordinates and the key which is saved in the config file. The result is then parsed into a weather object and displayed accordingly.



Temperatur: 18 C°
Felt temperatur: 18 C°
Description: Moderate rain

7. Timetable

Full content of timetable is available to access here:
https://docs.google.com/spreadsheets/d/1xSppqy2dRaRBgpASiDI4jcNdfcQUHKwk_M99GS_QDdo/edit#gid=1859198073

or see the short summary here:

Tobias Meindl

DAL:	22:30:00
Testing:	02:40:00
Documentation:	05:00:00
Features:	24:15:00
User-Interface:	10:45:00
Total time effort:	65:10:00

Lukas Varga

DAL:	05:56:00
Testing:	02:45:00
Logic:	38:11:00
Patterns:	04:45:00
Report:	08:22:00
Config:	01:43:00
Documentation:	04:40:00
User-Interface:	15:51:00
Total time effort:	82:13:00

8. Git-Link

Link to reach the GitHub Repository storing the project TourPlanner for lecture Software-Engineering 2 under supervision of lecturer Bernhard Wallisch:
https://github.com/yqni13/TourPlanner

9. Additional computed values

Popularity of a tour will be shown as the number of logs a tour has.
Child-Friendliness only shows "true" if the average of all regarding tourlogs have a low difficulty, remain under 01:00:00 duration and a distance under 30 km.