Documentation

SS23-SWEN2-Tourplanner

Leonard Struck, if21b248
Bernhard Utz, if21b241

Table of Contents

1. Technical steps and decisions

We started our project with .NET MAUI since all team members were using MacOS and WPF was not compatible. However, we encountered difficulties in learning .NET MAUI as our classes had focused solely on WPF. In this phase we used a lot of time watching tutorials and reading documentation. After a lot of research we realized that .NET MAUI was not yet suitable for production, especially for desktop applications, we had to find an alternative solution.
To overcome this challenge, we made the decision to install Windows on our laptops and switch to WPF. At that point we had to start again learning the differences between .NET MAUI and WPF, which we did by revisiting many sections of our class and reading documentations.
Our first step was to implement the basic features, such as adding tours and tour logs, and ensuring that the data was properly stored in the database. This laid the foundation for the functionality of our application.
To effectively manage our tasks and stay organized, we created issues to track our progress and have a clear overview of the work that needed to be done. Categorizing these issues based on their importance helped us prioritize features, considering the time we had already invested in exploring .NET MAUI.
We adopted an iterative development approach that involved each team member assigning himself to an issue, implementing the corresponding feature in a dedicated branch, creating a pull request, and requesting a review from another team member. This collaborative review process ensured code quality and facilitated knowledge sharing within the team.
By following this workflow, we were able to make fast progress and accomplish a lot of work in our project.

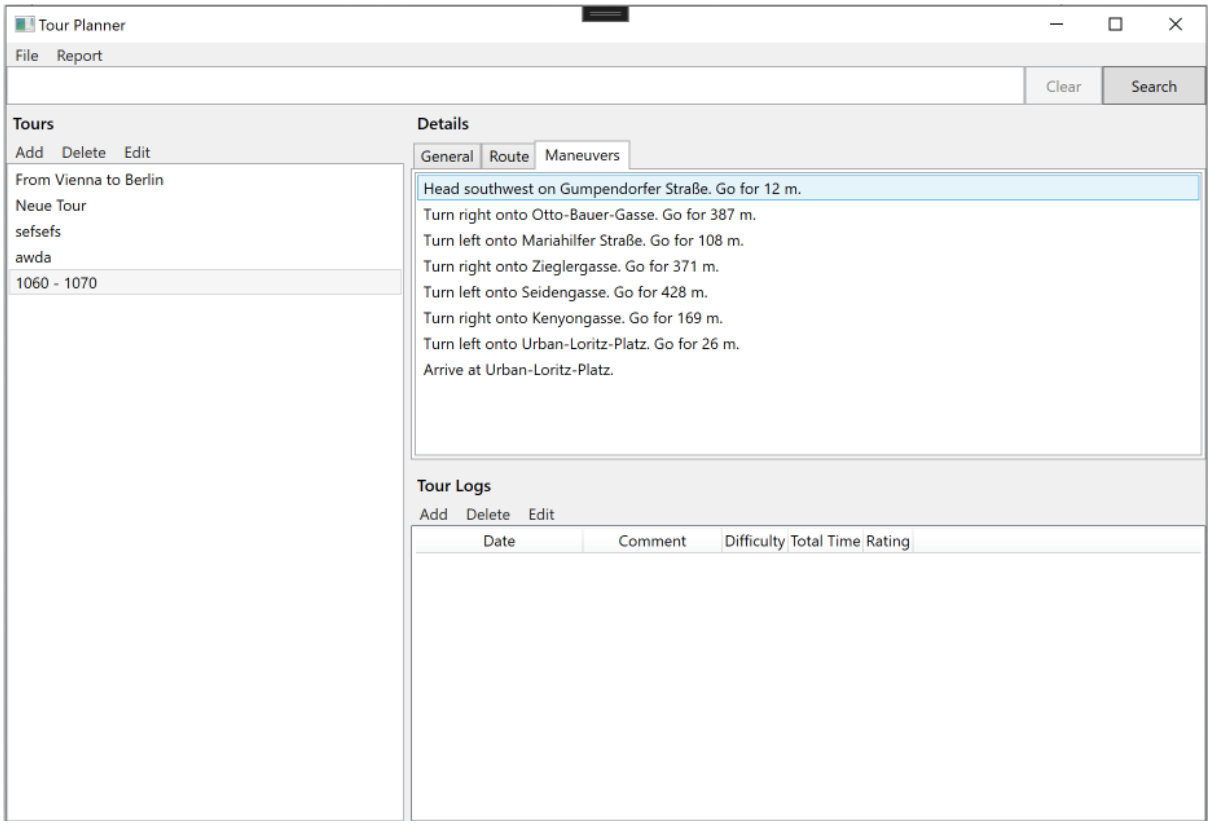
1.1. Libraries

Extended.Wpf.Toolkit: We chose to include the Extended.Wpf.Toolkit library because it provides a datetime user control and a busy-indicator, which greatly enhances the user experience of our application. By utilizing this library, we can provide a more intuitive and user-friendly interface for our users when dealing with date and time-related tasks as well as to indicate when our application is busy.

itext7: We opted to include the itext7 library due to its powerful features for working with PDF documents. This library offers a comprehensive set of functionalities, including creating, editing, and manipulating PDF files programmatically.

Moq: We made the choice to include the Moq library as our mocking framework for unit testing purposes. Moq allows us to create mock objects and define their behavior, enabling us to isolate and test individual components of our solution without dependencies on external systems or resources. By using Moq, we can simulate various scenarios, control inputs and outputs, and verify the interactions between different parts of our codebase. This leads to more reliable and maintainable unit tests, which ultimately contributes to the overall stability and quality of our software.

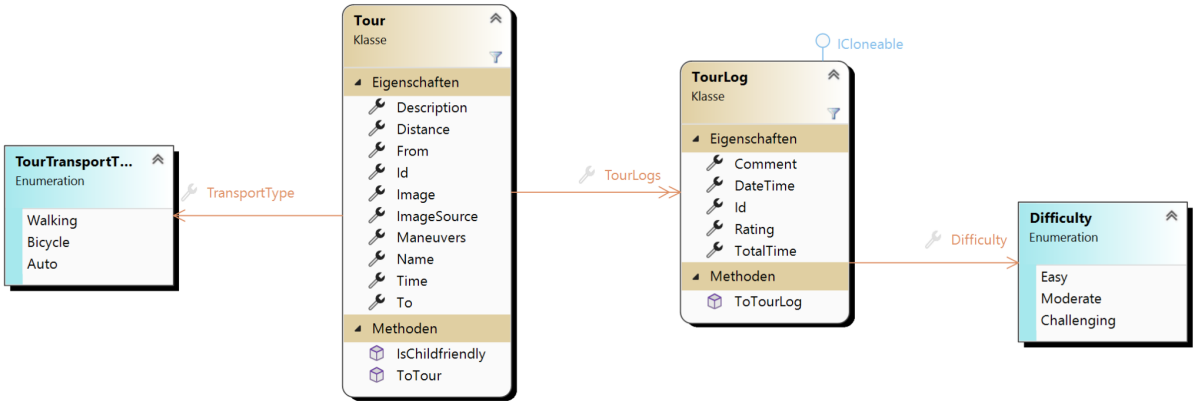
1.2. Unique feature



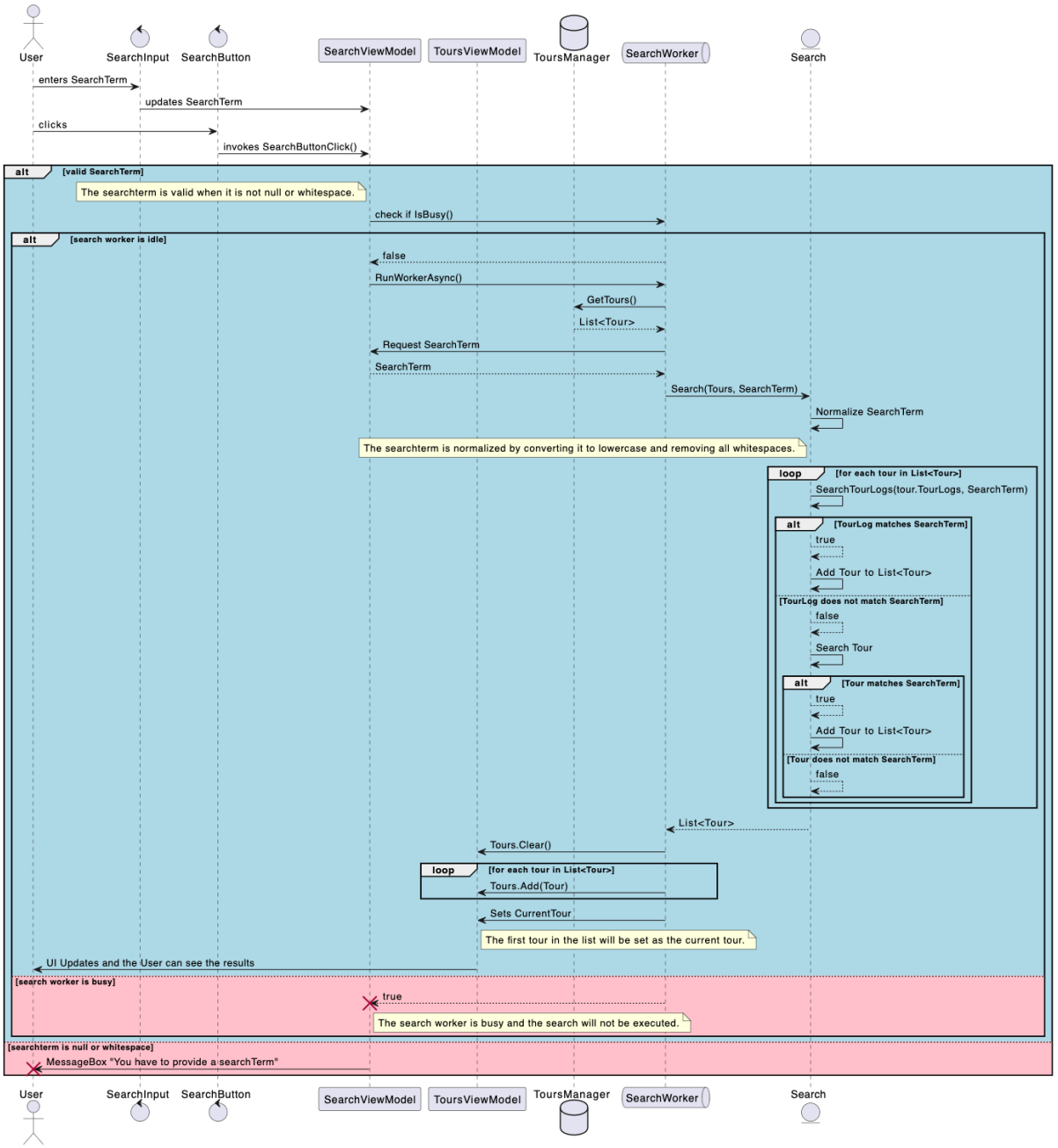
As a unique feature we decided to not only showcase the map with the route but also a list of directions. In a separated tab the User is able to get a list of all the maneuvers they have to take to complete the route. We decided on that feature because we wanted to make the App actually usable and since the map is way too small for longer distances, we thought the map would give a general direction, but the maneuvers would tell the user in more detail where they would have to go.

2. Architecture

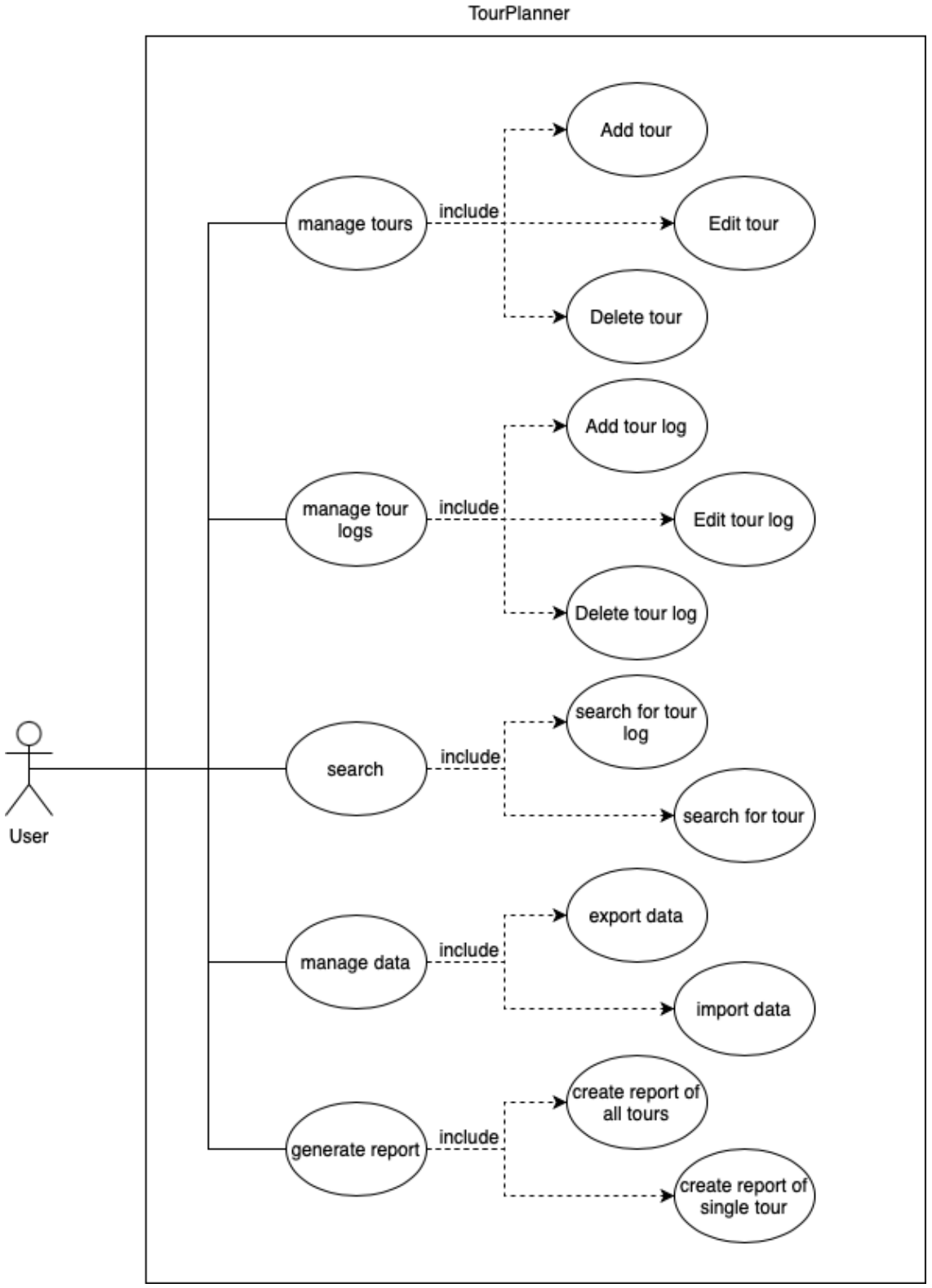
2.1. Model-Classes



2.2. Sequence diagram for full-text search

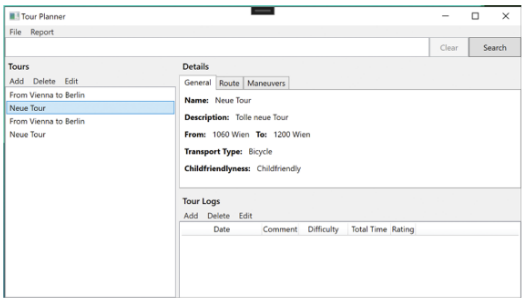


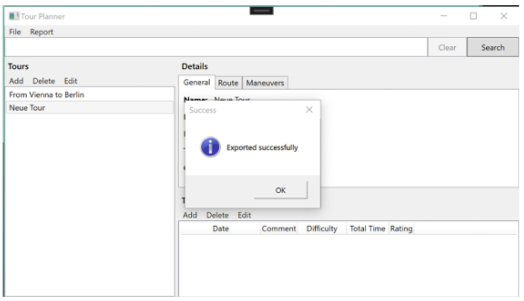
3. Use cases


TourPlanner
Add tour
manage tours
include
Edit tour
Delete tour
Add tour log
manage tour logs
include
Edit tour log
Delete tour log
search for tour log
search
include
search for tour
User
export data
manage data
include
import data
create report of all tours
generate report
include
create report of single tour

4. UI

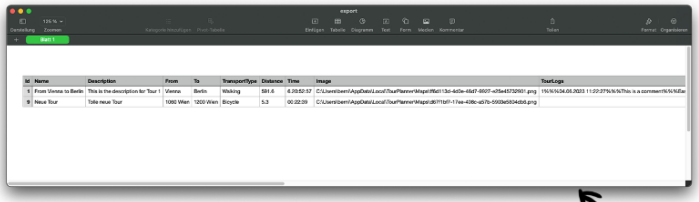

add a tour
delete a tour
...logs
search ...
...tours
View Details
View Route
View Maneuvers
add a log
delete a log

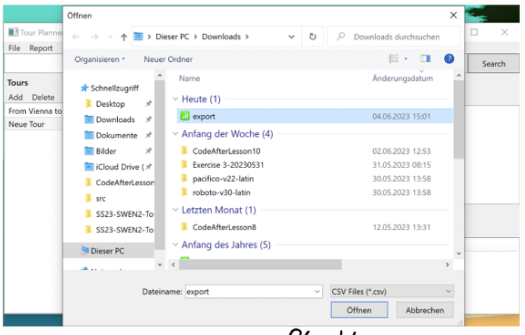




import

export

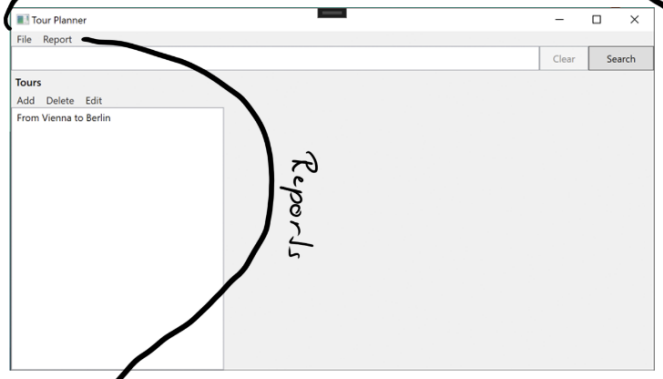




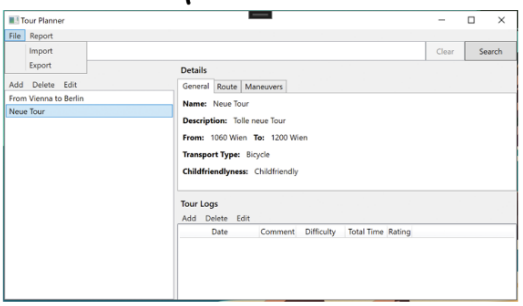
Import/Export

the exported file

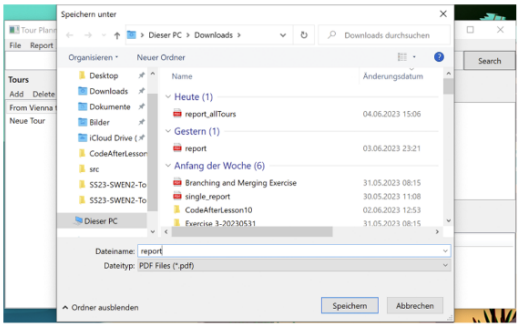
choose filepath



Reports

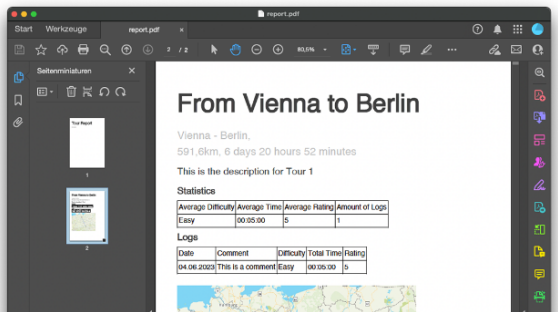


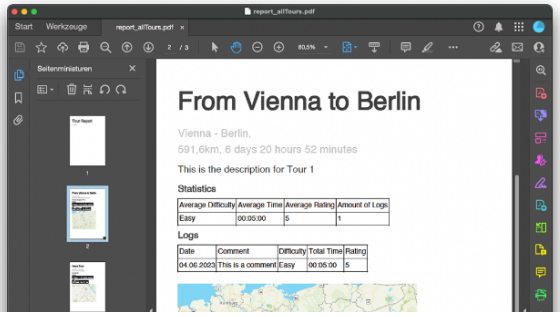
generate Report for...



... a specific tour

... all tours





5. Design Patterns

5.1. Model-View-ViewModel (MVVM)

We used this architectural design pattern in our Solution. MVVM provides a clear separation between the user interface (View), the application logic (ViewModel) and the underlying data (Model), allowing for easier development and maintenance of complex applications.

We used the following key concepts and features of MVVM:

- Data Binding:
  Allows properties of the ViewModel to be bound to visual elements in the View, ensuring that any changes in the ViewModel are automatically reflected in the View and vice versa.
- Commands:
  Actions that can be performed by the user in response to UI events, such as button clicks.
- INotifyPropertyChanged:
  The INotifyPropertyChanged interface allows the Viewmodel to notify the View of property changes, ensuring that the UI is updated whenever data changes. By implementing this interface and raising the PropertyChanged event, the ViewModel informs the View to refresh the corresponding bindings.

5.2. Dependency Injection

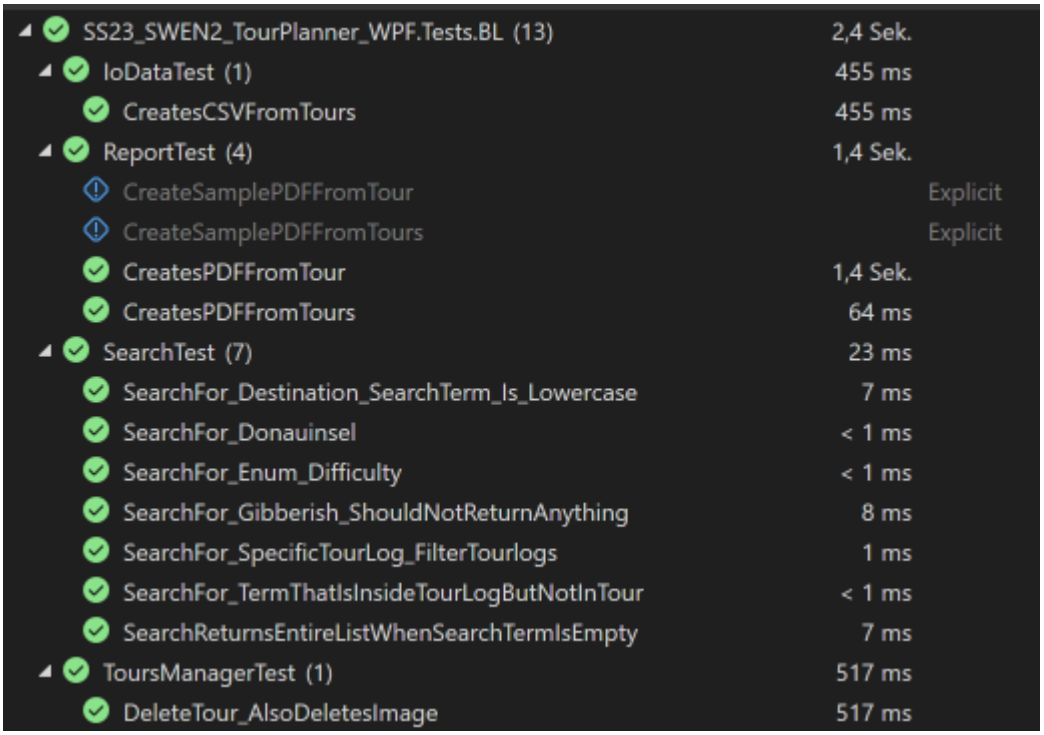
We used Dependency Injection (DI) in our Solution. DI is a pattern where the dependencies required by a class are provided externally, rather than being initialized within the class itself. This results in a better separation of concerns and promotes the concept of “inversion of control”) (IoC).
A rather important advantage of DI we benefited from in our Solution is that we are able to write unit tests for each of our components. The dependencies can be easily mocked or replaced with test doubles, enabling isolated testing of each component.
In our solution we used both Constructor Injection, and the Resolving of Dependencies using the DI Container (in our case a ServiceCollection).
When adding our services to the ServiceCollection we also made sure to choose a suitable lifetime of the service. The DataManager for example is registered as a singleton. This means that only one instance of the DataManager will be created and shared throughout the application. This is very useful to keep the amount of connections to the database to a minimum.
By externalizing the dependencies we were able to introduce new features or modify existing ones without having to introduce major changes to the codebase.

One example where we use DI is a MessageBoxService that we use throughout the application. The MessageBoxService exposes the built-in MessageBox Class in WPF, but allows us to mock the service inside Unit Tests.

6. Unit Tests

To test our program we implemented 22 Unit Tests including two explicit tests. To get a better overview we separated them into the layers where the tested functions are located.

6.1. BL



Inside of the Business Logic we tested the three main functions being exporting the tours, creating a report and searching for tours. In addition to that we tested that when a tour is deleted, the corresponding file of the displayed map is also deleted.

6.1.1. IoDataTest

In the “IoDataTest” we tested the exporting function. We set our focus here on if a file is created when the tours should be exported.

6.1.2. ReportTest

Like in the “IoDataTest” our focus in the “ReportTest” was to check if a file was created. Since in our program there are two options, one to create a Report of a single Tour and one to create a Report of all the tours, we tested both of these functions.
Apart from those tests we also implemented two explicit Tests. Those only run when specifically called and create a sample Report for the developer to look at.

6.1.3. SearchTest

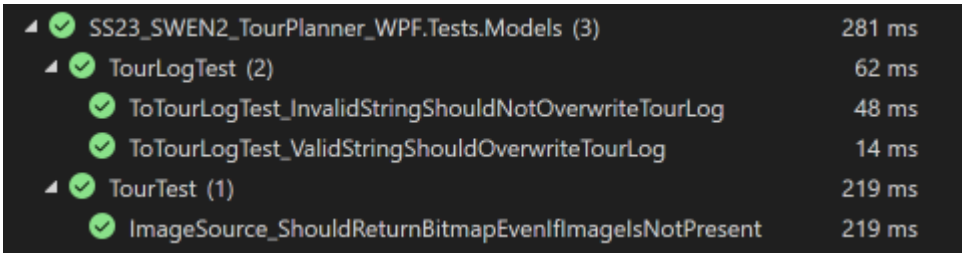
Inside of the BL Tests we set our main focus on testing the search functionality. Here we test what happens on different user inputs and if the search function gives us the wanted results

back. Here we tested searching for a tour, a comment in the logs, difficulty and what happens on no input or gibberish.

6.1.4. TourManagerTest

In the “TourManagerTest” we are testing the tourManager and check if it also deletes the image of the map when a tour is deleted.

6.2. Models



The Tests of the Models mostly test if helper functions do what they are supposed to do.

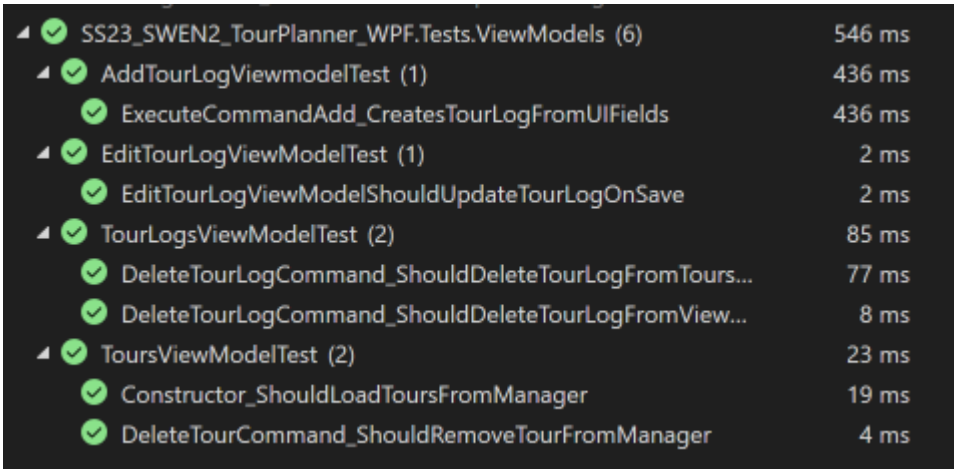
6.2.1. TourLogTest

In the “TourLogTest” the functionality of the .ToTourLog(string) is tested. If a valid string is passed the TourLog that called the function should be set to the TourLog that is represented by the string. This is tested with a valid and an invalid string.

6.2.2. TourTest

The “TourTest” checks if a tour returns the placeholder Bitmap if the image of the map is not set.

6.3. ViewModels



When testing the ViewModels we primarily set our focus on checking the functionality of adding and deleting tours and tour logs in their ViewModels.

6.3.1. AddTourLogViewModelTest

This test checks if a TourLog that is created via the AddTourLogViewModel triggers the EventHandler and that the TourLog is set correctly.

6.3.2. EditTourLogViewModelTest

This test checks if a TourLog is correctly edited by the EditTourLogViewModel.

6.3.3. TourLogsViewModelTest

The DeleteTourLogCommand is tested to verify that it removes a tour log from the view model's TourLogs collection and, in a separate test, if it also removes it from the ITourLogManager.

6.3.4. TourViewModelTest

In the “TourViewModelTest” we check if the tours are correctly loaded from the IToursManager as well as if the DeleteTourCommand correctly deletes a Tour from the IToursManager.

7. Tracked Time

We encountered a setback during the project because we initially implemented the TourPlanner with .NET Maui, a framework that proved to be unstable.
Consequently, we decided to rebuild the application using WPF, resulting in an additional 40 hours of development time.

.NET MAUI

- Total Commits: 29
- Hours: ~40 hours

WPF

- Total Commits: 195
- Hours: ~160 hours

8. Git

https://github.com/leonardstruck/SS23-SWEN2-TourPlanner-WPF.git