Dominik Neuwirth
Vincent Bernhart-Straberger

Tour Planner Protocol

Git: https://github.com/Neuwik/SWE_TourPlanner

Architecture description:

App Overview

The TourPlanner uses a responsive WPF for the Graphical User Interface (GUI). The app uses a View model to handle the UI logic, data binding and command handling.

Our application has a layer-based architecture with UI, Business, and Data Access layers. The UI layer handles user interactions and presentation. The Business layer holds the logic and rules of the application. The Data Access layer interacts with our PostgreSQL database using O/R mapping. The O/R mapping is done with Repositories and the EFCore DbContext and DbSets.

For tracking and debugging, we integrate log4net as shown in the moodle example (https://git.technikum-wien.at/swen/swen2/cs/log4net).

We also use iText7 to generate pdf reports of tours.

To calculate the routes of the tours we used the external service https://openrouteservice.org. To display the route in WPF we used WebView2 and https://www.openstreetmap.org/, as shown in the moodle example (https://git.technikum-wien.at/swen/swen2/cs/wpf-webview2-leafletmap)

Configuration management is implemented with a JSON file, containing parameters such as the database connection string, file paths and the API-Key for OpenRouteService. The Config allows modification without need to change the code and rebuilding the project.

We also implemented UnitTests with NUnit and EFCore.InMemory. This allows us to check if the logic is functional without needing to run a separate test database.

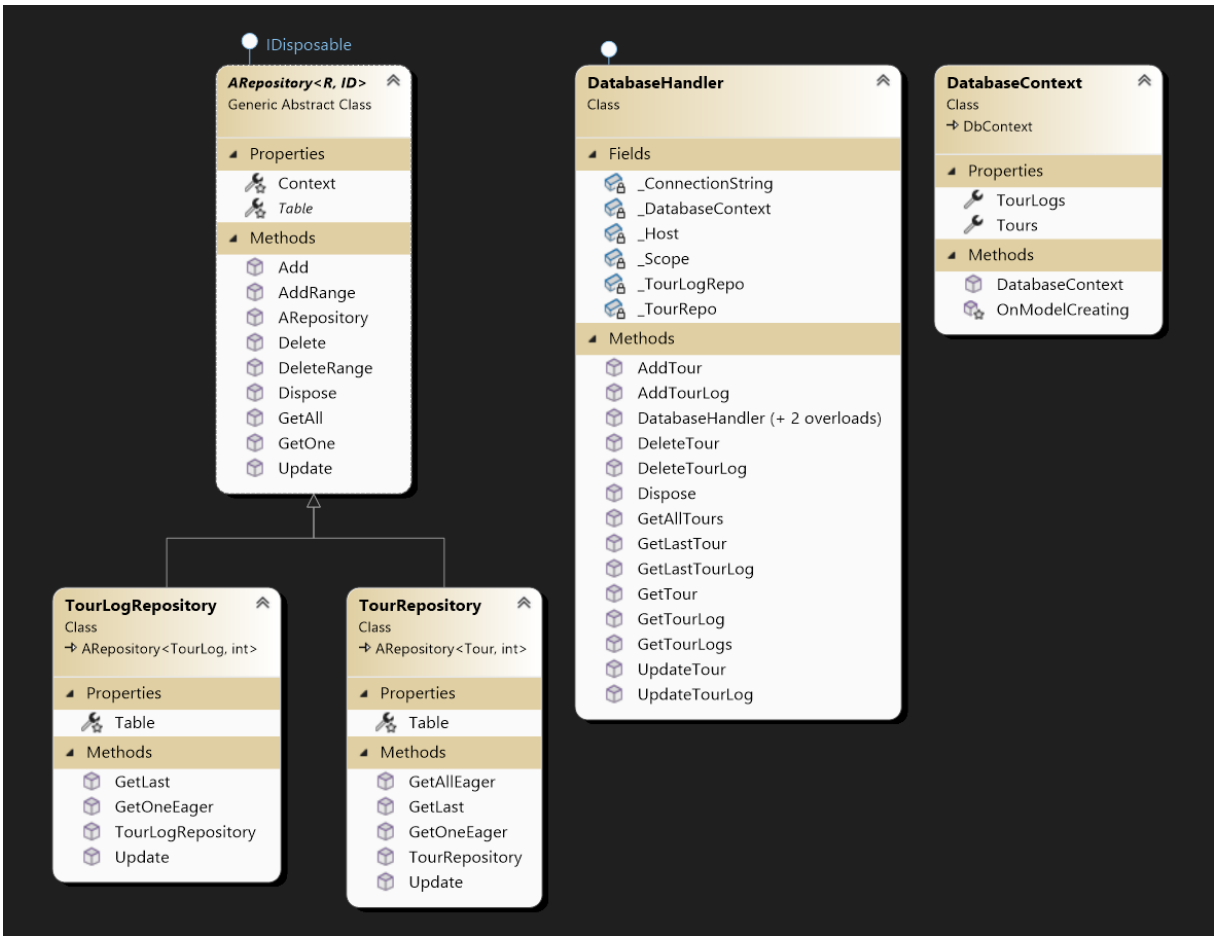
Layers

Data Access Layer:

DatabaseContext: This class implements the database context using Entity Framework's DbContext. It contains DbSet properties representing the entities in the database schema.

Repositories: These classes provide an abstraction layer over the database context, offering methods for CRUD operations on entities. Each repository corresponds to a DbSet in the context. The Repositories also implement IDisposable to dispose of the referenced DbContext.

DatabaseHandler: Acts as a central access point to the database functionality. It uses IHost to create a connection to the Docker PostgreSQL database. This class instantiates and manages repositories, providing a simplified and central interface for the Business Layer to interact with the database. This class also implements IDisposable to dispose of the referenced DbContext

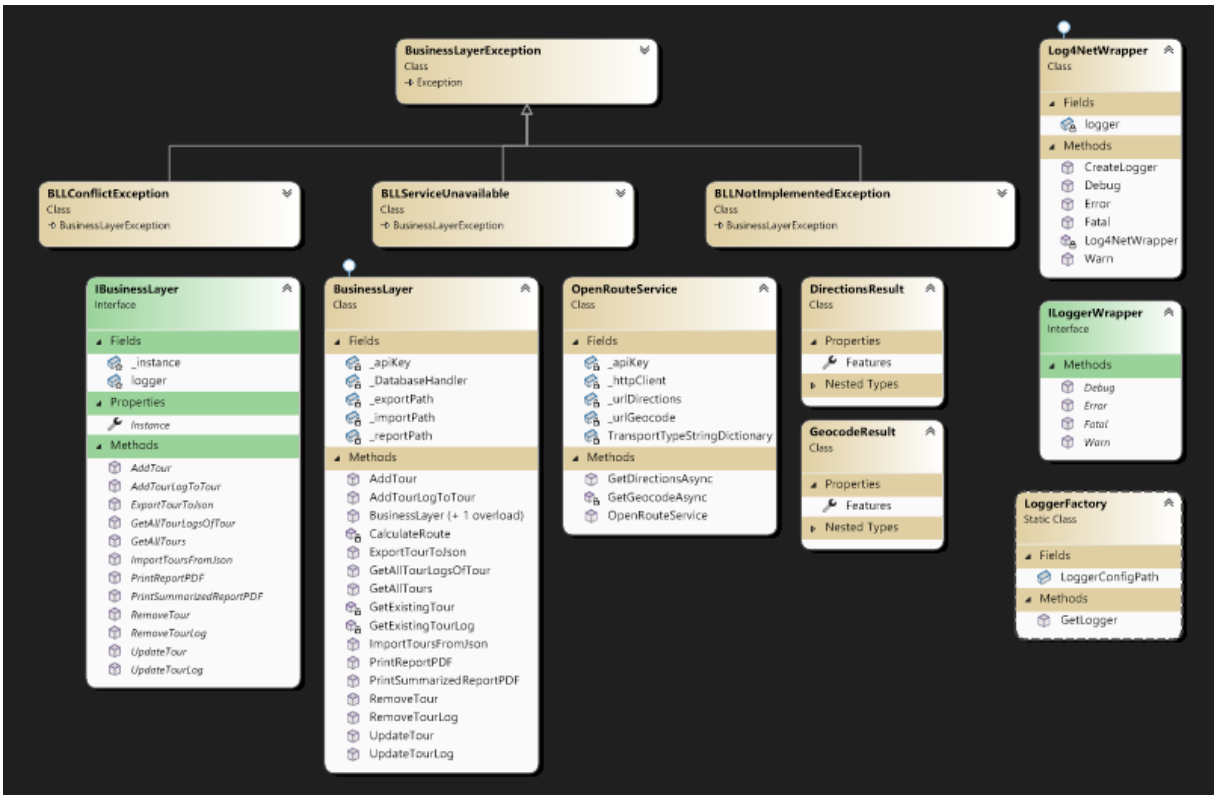


Business Layer:

IBusinessLayer: An interface defining the contract for the Business Layer. It contains method signatures representing the operations that the Business Layer can perform. It is implemented as a singleton to ensure a single instance throughout the application's lifecycle.

BusinessLayer: Implements the IBusinessLayer interface. It serves as the core logic layer of the application, handling interactions between the UI Layer and the Data Access Layer. It holds references to the DatabaseHandler and coordinates various helper classes and functions.

Helper Classes and Functions: These include functionalities such as logging, routing, PDF generation, JSON importing/exporting, and value calculations.



View Layer:

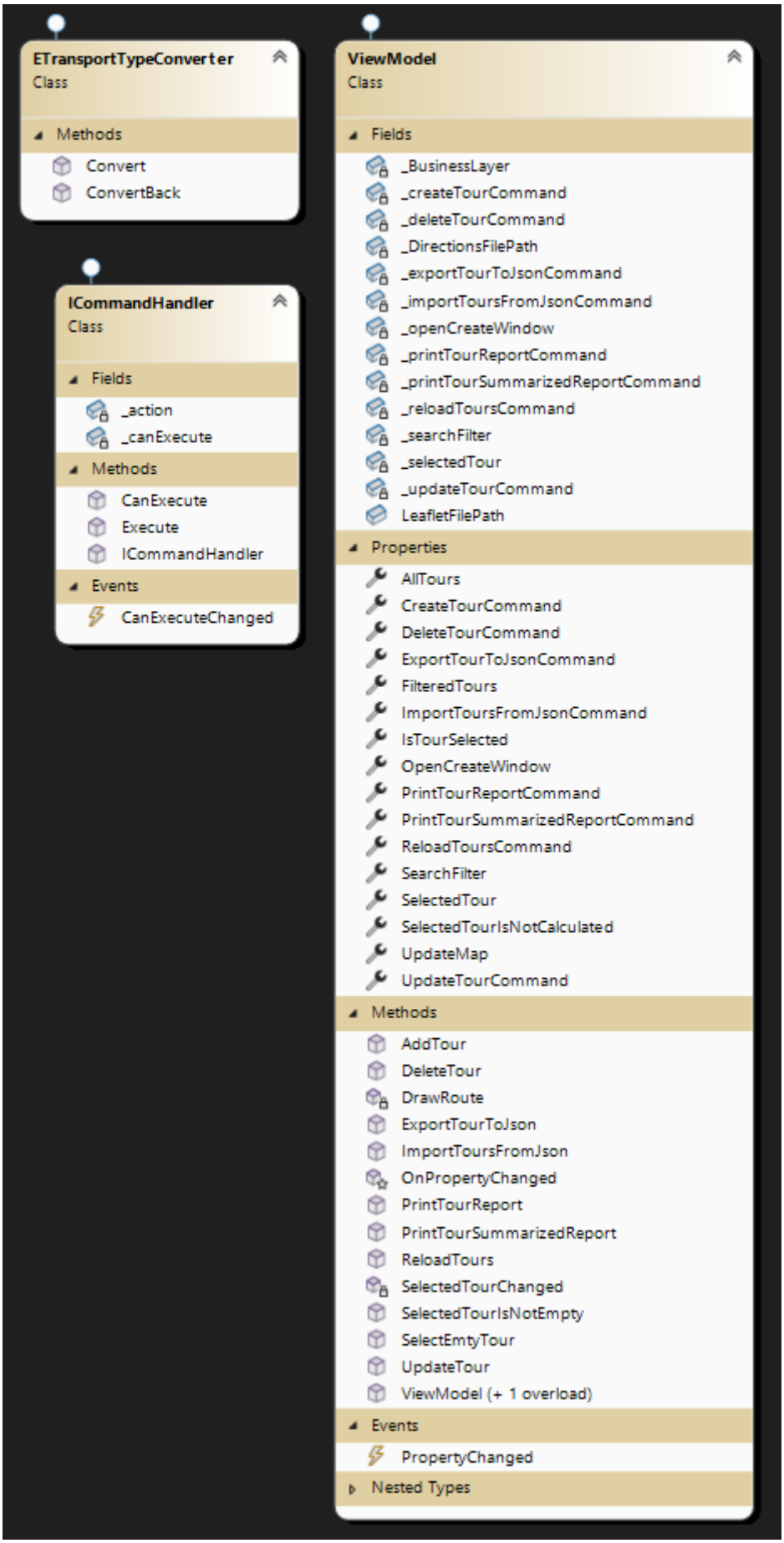
WPF View: Represents the visual presentation of the application's user interface. It includes XAML markup defining the layout and appearance of UI elements.

ViewModel: Acts as an intermediary between the View and the Business Layer. It handles data binding and command binding between the UI elements and the underlying data and operations. It references the IBusinessLayer.Instance to access and manipulate data.

ICommandHandler: Defines command objects that encapsulate user actions and operations triggered from the UI.

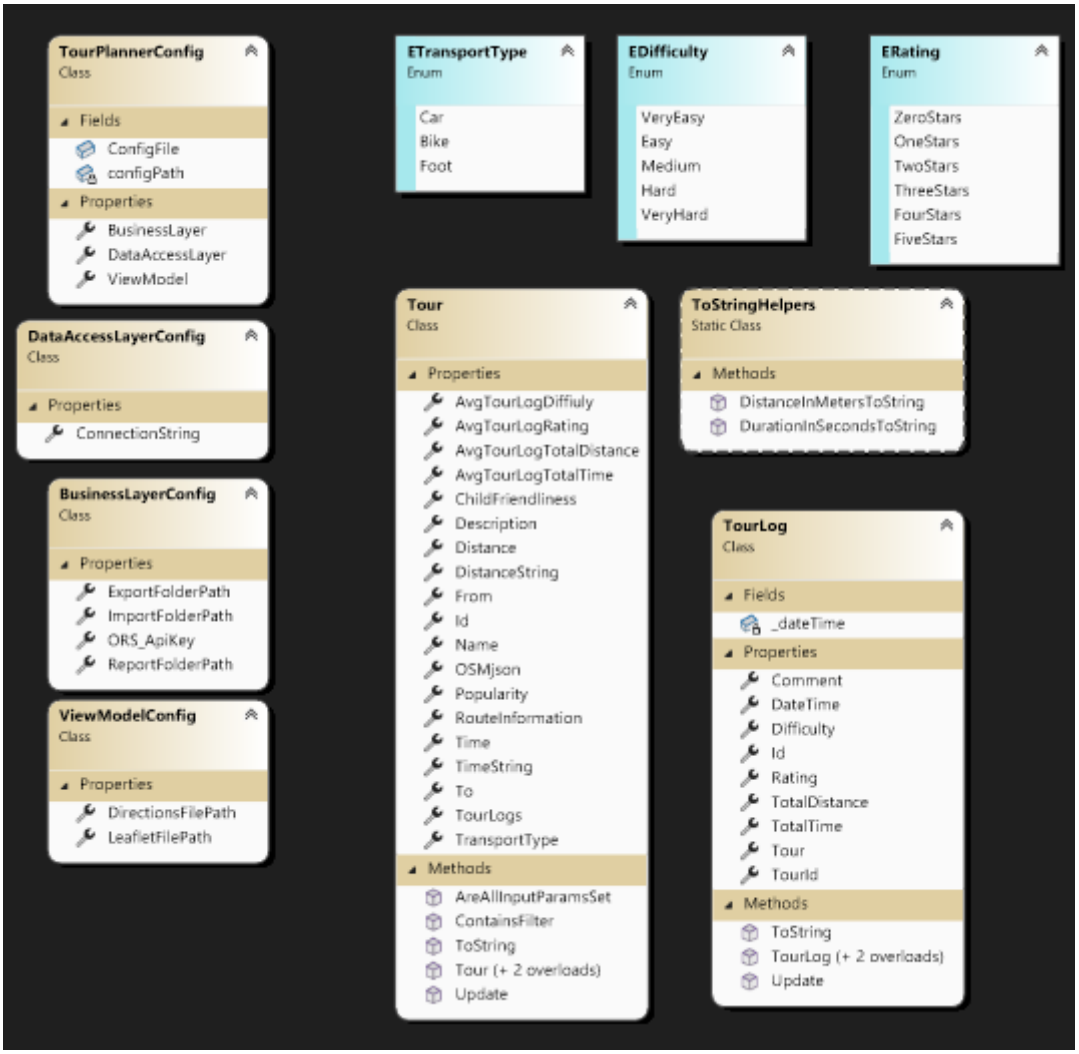
Converters for Enum: These classes convert between enum values and their corresponding representations in the UI, for integration of enum data types into the View Layer.

Dominik Neuwirth
Vincent Bernhart-Straberger



Config and Model Classes

The app has separate model classes and classes used for the config management, that every Layer has access to.



use cases:

plan tours :-)

Library decisions:

Microsoft.EntityFrameworkCore

Entity Framework Core (EFCore) was chosen to handle all the database interactions in the application. It simplifies the process of querying and updating the PostgreSQL database by allowing us to work with .NET objects. This ORM saves a lot of time and effort since we don't have to write complex SQL queries manually. EFCore also supports PostgreSQL as demanded.

Npgsql.EntityFrameworkCore.PostgreSQL

Npgsql.EntityFrameworkCore.PostgreSQL is the EFCore provider for PostgreSQL that I used to connect the application to the PostgreSQL database. It allows EFCore to perform CRUD operations on the PostgreSQL Database.

Microsoft.Extensions.Hosting

We used Microsoft.Extensions.Hosting to manage the lifecycle of the application. This package makes it easy to configure services and handle startup and shutdown processes cleanly. It ensures that dependencies are injected properly, making the code more modular and easier to maintain. It simplifies creating the connection to the docker a lot, because migrations are not needed.

Microsoft.Web.WebView2

Microsoft.Web.WebView2 is used to display OpenStreetMap within the WPF application. This component allows embedding web content, such as maps, using the Chromium engine, making it possible to render dynamic and interactive maps.

Log4Net (Microsoft.Extensions.Logging.Log4Net.AspNetCore)

Log4Net was integrated for logging purposes, using the Microsoft.Extensions.Logging.Log4Net.AspNetCore package for easy integration with the .NET logging system. It helps me track the application's behaviour by logging important events, errors, and warnings. Also, this package was mandatory for the project.

itext7 + itext7.bouncy-castle-fips-adapter

We used iText7 in the application to generate PDF reports for the tours that users create. The library made it easy to format and create PDF documents. The Bouncy-Castle-FIPS-Adapter is included to provide cryptographic support and security for potential sensitive data (Visual Studio said we need this).

NUnit + Microsoft.EntityFrameworkCore.InMemory

NUnit is the testing framework we chose for writing unit tests for the application. Microsoft.EntityFrameworkCore.InMemory is used in our tests to simulate a database within the testing framework. This allows testing of the DatabaseHandler and the BusinessLayer functionality without needing the external database.

Design Pattern

Singleton Pattern

The Singleton Pattern was used for the Business Layer. This ensured that only one instance of the Business Layer exists. Also, it allows to access all the Business Layer functions by calling the interfaces static instance. So, if the Business Logic needs to be changed than it can be simply swapped out in the interfaces instance getter.

Dominik Neuwirth
Vincent Bernhart-Straberger

Repository Pattern

The Repository Pattern was used for the data access. We implemented a ARepository which works with generic datatypes. For specific management of Tours and TourLogs a TourRepository and a TourLogRepository were implemented respectively. The ARepository also implements the IDisposable interface for disposing of the DbContext.

Lessons learned:

Neuwirth	Bernhart-Straberger
My biggest learning was that documentation from my HTL (3 years ago) was nearly useless for setting up the DbContext. Microsoft changed so much, that most of the methods did not exist anymore. Other than that the project was straight forward. Maybe the API calls were a little bit tricky, because I had to figure out what the response data looked like to write model classes for it. Also, the tilemap is a pain and the interactive map is way easier thanks to the example on Moodle. Same counts for the logging. The Moodle code really helped a lot.	I think my biggest learning achievement was made while worked on the data mapping in the DatabaseHandler, because at the beginning I tried to code the DatabaseHandler like last semester which would have been ~300 additional lines, but after talked to Dominik he told that I should use `DatabaseContext` which was a bit frustrating at the beginning, but after around one hour I got the hang of it. Unfortunately, the implementation of the Database wasn't as smooth as I hoped for which is where Dominik helped a lot, but after we spent 10+ hours we were able to fix the data mapping.

Unit testing decisions:

Tests	Description / Reason for implemation
BusinessLayerTests (6)	
Test_BusinessLayer_AddTour Passed	Is used for verifying that a tour can be successfully added to the business layer, ensuring all properties are correctly set and calculated values (distance, time, etc.) are generated.
Test_BusinessLayer_AddTourLogToTour Passed	Is used for verifying that a tour log can be added to a specific tour in the business layer, ensuring the log's properties are correctly saved and linked to the tour.
Test_BusinessLayer_RemoveTour Passed	Is used for confirming that a tour can be successfully removed from the business layer, ensuring the total count of tours decreases appropriately.
Test_BusinessLayer_RemoveTourLog Passed	Is used for confirming that a tour log can be successfully removed from a tour in the business layer, ensuring the total count of logs decreases appropriately.
Test_BusinessLayer_UpdateTour Passed	Is used for ensuring that an existing tour's properties can be updated in the business layer and the changes are correctly reflected in the database.
Test_BusinessLayer_UpdateTourLog Passed	Is used for ensuring that an existing tour log's properties can be updated in the business layer and the changes are correctly reflected in the database.

DatabaseHandlerTests (11)	
Test_DatabaseHandler_AddTour Passed	Is used for verifying that a tour can be successfully added to the database and all properties are correctly saved.
Test_DatabaseHandler_AddTourLog Passed	Is used for checking that a tour log can be added to the database and all properties are correctly saved.
Test_DatabaseHandler_DeleteTour Passed	Is used for verifying that a tour can be deleted from the database and ensuring it no longer exists when fetched.
Test_DatabaseHandler_DeleteTourLog Passed	Is used for verifying that a tour log can be deleted from the database and ensuring it no longer exists when fetched.
Test_DatabaseHandler_GetAllTours Passed	Is used for confirming that multiple tours can be added and retrieved from the database, verifying the count matches the expected number.
Test_DatabaseHandler_GetLastTour Passed	Is used for checking that the last added tour can be retrieved correctly and its properties match the input data.
Test_DatabaseHandler_GetTour Passed	Is used for validating that a specific tour can be retrieved by its ID and its properties are as expected.
Test_DatabaseHandler_GetTourLogs Passed	Is used for confirming that multiple tour logs can be added for a tour and retrieved from the database, verifying the count matches the expected number.
Test_DatabaseHandler_InMemoryDatabase Passed	Is used for ensuring that the DatabaseHandler object is correctly instantiated and not null when using an in-memory database.
Test_DatabaseHandler_UpdateTour Passed	Is used for ensuring that an existing tour's properties can be updated and the changes are correctly saved in the database.
Test_DatabaseHandler_UpdateTourLog Passed	Is used for ensuring that an existing tour log's property can be updated and the changes are correctly saved in the database.
TourLogTests (5)	
Test_TourLog_CopyConstructor Passed	Is used for confirming that the copy constructor accurately duplicates all properties from the original TourLog instance.
Test_TourLog_DateTimeSetterUTC Passed	Is used for ensuring that the DateTime setter converts local time to UTC correctly.
Test_TourLog_DefaultConstructor Passed	Is used for verifying that the default constructor initializes all properties correctly, including setting the current UTC time for the DateTime property.
Test_TourLog_ParameterizedConstructor Passed	Is used for ensuring that the parameterized constructor correctly sets all properties to the provided values.
Test_TourLog_Update Passed	Is used for checking that the Update method correctly updates the properties of a TourLog instance with values from another instance.
TourTests (13)	
Test_Tour_AreAllInputParamsSet Passed	Is used for verifying that the AreAllInputParamsSet method correctly identifies when all necessary properties are set.
Test_Tour_AvgTourLogDifficulty_CheckCorrectCalculation Passed	Is used for confirming that the AvgTourLogDifficulty property correctly calculates the average difficulty of all associated tour logs.
Test_Tour_AvgTourLogRating_CheckCorrectCalculation Passed	Is used for confirming that the AvgTourLogRating property correctly calculates the average rating of all associated tour logs.

Test_Tour_AvgTourLogTotalDistance_CheckCorrectCalculation Passed	Is used for verifying that the AvgTourLogTotalDistance property correctly calculates the average total distance of all associated tour logs.
Test_Tour_AvgTourLogTotalTime_CheckCorrectCalculation Passed	Is used for ensuring that the AvgTourLogTotalTime property correctly calculates the average total time of all associated tour logs.
Test_Tour_ChildFriendliness_ CheckIfVeryEasyRouteHasChildFriendlinessVeryEasy Passed	Is used for ensuring that a route with a very easy difficulty and other conducive parameters results in a ChildFriendliness rating of very easy.
Test_Tour_ChildFriendliness_ CheckIfVeryHardRouteHasChildFriendlinessVeryHard Passed	Is used for confirming that a route with a very hard difficulty and other challenging parameters results in a ChildFriendliness rating of very hard.
Test_Tour_ContainsFilter Passed	Is used for ensuring that the ContainsFilter method accurately determines whether the tour's name or description contains a given string.
Test_Tour_CopyConstructor Passed	Is used for confirming that the copy constructor accurately duplicates all properties from the original Tour instance, including a deep copy of the TourLogs list.
Test_Tour_DefaultConstructor Passed	Is used for verifying that the default constructor initializes all properties correctly, including the TourLogs list being instantiated and empty.
Test_Tour_ParameterizedConstructor Passed	Is used for ensuring that the parameterized constructor correctly sets all properties to the provided values.
Test_Tour_Popularity_CheckIfIncreasesWithTourLogCount Passed	Is used for verifying that the Popularity property increases as the number of associated tour logs increases.
Test_Tour_Update Passed	Is used for checking that the Update method correctly updates the properties of a Tour instance with values from another instance.

Unique feature description:

The Unique feature of our application can be found in the “Detailed Information” tab. This tab displays a detailed description of the rout containing a step-by-step description of where and when to turn in which direction. This description is also printed into the report PDF (only the normal report not the summarized report)

Time measurement

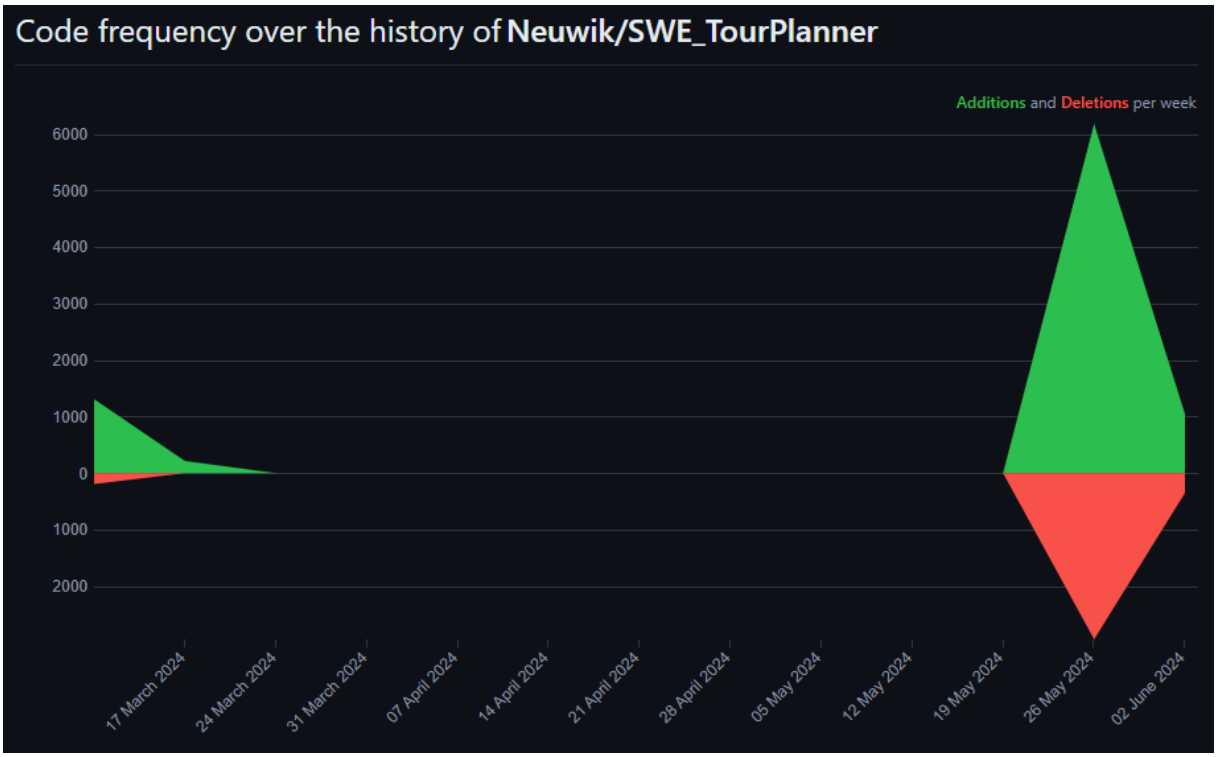
Neuwirth:

Date	Time (h)	Description
Bis 31.03.2024	?*	ViewModel, Databinding (CRUD), ICommands
30.05.2024	5	Business Layer basics, OpenRouteService, OpenStreetMap
31.05.2024	12	Business Layer + DatabaseHandler + ViewModel connected, Logging

01.06.2024	10	Config File, PDF Reporting, Json export/import, Searchbar
02.06.2024	3	Protocol, Final Touches

Bernhart-Straberger:

Date	Time (h)	Description
Bis 31.03.2024	?*	Model Classes, WPF User-Interface, User-Input validation, UnitTest
29.05.2024	2	DatabaseHandler basics
30.05.2024	10+	DbContext and Repos
31.05.2024	12	Docker creation, Docker connection with EFCore, UnitTests
01.06.2024	6	UnitTests, Protocol
02.06.2024	3	UnitTests, Protocol, Final Touches



(* ? weil wir uns nicht mehr erinnern können wer wie viel Zeit für die Zwischenabgabe gemacht hat.)