SEMINARARBEIT

im Studiengang Informatik/Computer Science

Lehrveranstaltung Software Engineering 2 Labor

TourPlanner

Ausgeführt von: Kai Lukasser, Paul Steindl

Personenkennzeichen: 2010257232, 51918753

BegutachterIn: DI Wolfgang Berger

Wien, 10.07.202

Inhaltsverzeichnis

[1.1 Description 3](#_Toc108306640)

[1.2 Tracked time 3](#_Toc108306641)

[1.3 GitHub Link 3](#_Toc108306642)

[2.1 Design 3](#_Toc108306643)

[2.2 Failures 3](#_Toc108306644)

[2.3 Selected solutions 3](#_Toc108306645)

[3.1 Tourplanner 3](#_Toc108306646)

[3.2 Tourplanner.BusinessLayer 3](#_Toc108306647)

[3.3 Tourplanner.DataAccessLayer 3](#_Toc108306648)

[3.4 Tourplanner.Models 3](#_Toc108306649)

[3.5 Tourplanner.Shared 3](#_Toc108306650)

[3.6 Tourplanner.Shared.Log4Net 3](#_Toc108306651)

[3.7 Tourplanner.UnitTests 3](#_Toc108306652)

[4.1 Via UML use case diagram 4](#_Toc108306653)

[4.2 Via Wireframes 4](#_Toc108306654)

[4.3 Via UML Class Diagram 4](#_Toc108306655)

[4.4 Via sequence diagram for full-text search 4](#_Toc108306656)

# Overview

## Description

Develop an application based on the GUI frameworks C# / WPF or Java / JavaFX. The user creates (bike-, hike-, running- or vacation-) tours in advance and manages the logs and statistical data of accomplished tours.

## Tracked time

About 70 hours each

## GitHub Link

https://github.com/PaulSteindl/Tourplanner

# Technical steps and desicions

## Design patterns

### MVVM

We implemented our project based on the MVVM-pattern. So, we have a decoupling of Model and View through the different ViewModels. Therefore, it makes the ViewModel reusable. It exposes properties and commands for binding in view.

### SOLID

We followed the SOLID principle. Therefore, we have:

* A separation of concerns
* No repetition of code
* We kept everything super simple
* No code what we don’t need or use
* Each unit has limited knowledge about other units

### Layered Architecture

Using logical layers in which components are grouped logically and communicate with each other and with other applications and clients, we have an overall structure for our applications. The structure is described in the following sites.

## Failures

We had one big failure in the frontend. Namely, we implemented the code not based on the MVVM-pattern. At first, we created all the different windows in the MainWindowViewModel. Furthermore, did we implement all buttons in there. In conclusion, we had to change the whole frontend code based on the MVVM-pattern in the last few days. The MainWindowViewModel is our main window, where everything is shown. It consists of the TourListViewModel which represents all the tours which are created or imported and also of the TourInformationViewModel which represents the information to one single tour and also the logs. The TourManagerViewModel is for adding or import a new tour. Furthermore, there is the ICloseWindow which is responsible for closing a window.

# App architecture

## Tourplanner

The Tourplanner project is our main project. It is responsible for everything and starts the application. Furthermore, it represents the frontend based on the MVVM-pattern for decoupling Model and view through ViewModel.

## Tourplanner.BusinessLayer

Our Tourplanner.BusinessLayer project implements the core functionality of the system and also encapsulates the relevant business logic. It consists of different interfaces that other callers can use.

### Calculate Attributes

Is responsible for calculating Tour attributes and the average of these attributes. Also, it formats the time and errorMessages into the correct format. It has access to the ILogDAO since the Tourlogs a relevant for some Attributes.

### Check Input

For checking the user input. Has two private methods and two public methods. The private ones check the Input and the public ones collect those Inputs for either Tour or Tourlog.

### Import Export

Is separated into two classes, one for importing the other one for exporting.

Import has access to the IFileDAO and ITourManager. It has a single responsibility process the data it gets from a file and convert it into a new Tour. It gains the data through IFileDAO and converts them through ITourManager.

Export has access to ITourDAO and IFileDAO. It is responsible for transforming a Tour into a json string. After that it transfers the string to IFileDAO and which puts the information into a File.

### Map Quest API

MapQuestApi consist of three classes and interfaces

IMapQuestConfiguration, this interface holds the main information it gains through the appsettings.config file. It saves the DirectionUrl, MapUrl and most importantly the MapQuest Key.

Directions has two methods each one fetching Data from the MapQuest. Method, one fetches from the DirectionsApi and returns a Route Information object. Method two fetches from the StaticMapApi and returns the byte array, which holds the information for the Tour Map. It has access to IMapQuestConfiguration.

RouteManager combines both methos from Directions and makes sure everything runs smoothly. Also, responsible to transfer the mapArray to IFileDAO. Has access to IFileDAO and IDirections.

### Report

The ReportManager creates both the SummarizeReport and the TourReport. It gets the calculated Attributes from ICalculateAttributes, the tour Information from tourManager and ITourDAO and the log information from ILogDAO. It has also access to IFileDAO since it has to delete a file.

### Text Search

SearchManager is responsible to search for a Text in every single Tour data, which includes Logs. Therefore, it needs access to ITourDAO and ILogDAO to load the data into a searchable object.

### Tour

TourManager is the core for every action you can do with a tour. Therefore, it has access to IRouteManager, TourDAO, ILogDAO, IFileDAO, ITourLogManager, ICheckInput, ICalculateAttributes. It can Load every Tour from the DB and assures that ever Log and Map is loaded aswell. Furthermore, it can check if a MapExists and load the map if needed. It can create, update and delete logs, as well as return AllLogs or Update the attributes from a log after update.

### Tour Log

TourLogManager is the core for every operation you can do with Logs. Since Logs are smaller objects with fewer possible operations it only requires ILogDAO and ICheckInput. It can create, update and delete Logs, as well as returning all Logs by a tour id.

## Tourplanner.DataAccessLayer

Our Tourplanner.DataAccessLayer project provides access to data. For example, it reads files, creates dictionaries and also has some database commands.

### File

FileDAO manages every action taken on the local machine regarding the file explorer. It can save and delete an image, read from Files which is important for the import function and create and delete files.

### Log

LogDAO requires DB access, therefore it needs access to IDatabaseManager. It has 4 core methods, which are Inserting a new Log into the DB, selecting all logs with a tour id from DB,

pdate a Log with a log id and delete a log from the DB through a log id.

### Tour

TourDAO also requires DB access and therefore it needs access to IDatabaseManager. It has 5 core methods, which are inserting a new Tour, selecting a tour by its id, selecting all tours in the DB, updating a tour by its id and deleting a tour by its id.

### Database Manager

The DatabaseManager builds a connection to the DB and also ensures that every Table is created. Since it requires the connection string it has access to IPostgreSqlDAOConfiguration.

### IPostgreSqlDAOConfiguration

IPostgreSqlDAOConfiguration, this interface holds the main information it gains through the appsettings.config file. It saves the connection string.

## Tourplanner.DataAccessLayerInMemory

This project represents an InMemoryDataAccessLayer. Only used for unit tests.

### Log

LogInMemoryDAO has 4 core methods, which are inserting a new Log into a private List, selecting all logs with a tour id from the list, update a Log with a log id and delete a log from the list through a log id.

### Tour

IDatabaseManager has 5 core methods, which are inserting a new Tour, selecting a tour by its id, selecting all tours into the private list, updating a tour by its id and deleting a tour by its id.

## Tourplanner.Models

Keeps the data of the business objects / entities.

## Tourplanner.Shared

We have several log levels between we can choose. It is used for different treatment of messages based on its level of severity.

## Tourplanner.Shared.Log4Net

Core functionality for logging.

## Tourplanner.UnitTests

We implemented over twenty unit tests, but unfortunately we wrote no tests for the ViewModels although it would be of importance.

# Documentation

## Via UML use case diagram

## Via Wireframes

At the beginning of our project, we created this Wireframing. It was just a simple one for getting an idea how it could look like. However, we overestimated ourselves very much with the graphical overview. In the end, it doesn't look anything like that.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, application

Description automatically generated

## Via UML Class Diagram

## Via sequence diagram for full-text search

# Unit Tests

We wrote Unit test for the core functionality in the business layer. It would be better if we also included test for the ViewModel. However, the backend was done before the ViewModel, therefore it was more time efficient to write the Unittest for the BussinesLayer only.

# Unique feature

Unfortunately, we don’t have a unique feature.

# Library Used

* NUnit for Unittesting
* Moq for Mocking in Unittests
* iText7 for PDF creation
* Npgsql for easy DB access
* Log4Net for logging

# Lessons learned

In the end, we learned a lot within this project.

* Working in pairs
* Handling merge conflicts
* Implementing a layered architecture
* MVVM-pattern
* How to handle different failures
* How to split up such a project