Contents

[Tour Planner Architecture 1](#_Toc104043676)

[Presentation-layer 2](#_Toc104043677)

[Business-layer 3](#_Toc104043678)

[Data-access-layer 5](#_Toc104043679)

[Model-layer 7](#_Toc104043680)

[Common layer 7](#_Toc104043681)

[Use Case Diagram 8](#_Toc104043682)

[Wireframes 8](#_Toc104043683)

[Sequence diagram of text search 11](#_Toc104043684)

[Unit testing decisions 11](#_Toc104043685)

[Time tracking 12](#_Toc104043686)

[Git link 12](#_Toc104043687)

[Design patterns 12](#_Toc104043688)

[Unit of work 12](#_Toc104043689)

[Repository 12](#_Toc104043690)

[Factory 13](#_Toc104043691)

[MVVM 13](#_Toc104043692)

[Unique feature 13](#_Toc104043693)

# Tour Planner Architecture

Text

Description automatically generated

Picture 1: Tour Planner Project view

The architecture of the Tour Planner App is layered, listed from the highest to the lowest layer tier, with a Presentation-, Business-, and Data-access-layer. Each layer uses either its own code or the code of the layer immediately below. There is also a Model- and Common-layer which are used by all other layers. Each layer is its own project, as seen in Picture 1.

The class diagram files from where the following screenshots have been taken, can all be found in the tour planner solution.

We used following additional NuGet packages: itext7, log4net, Newtonsoft.Json, Npgsql and NUnit.

## Presentation-layer

Graphical user interface

Description automatically generated

Picture 2: View and View Models

The presentation layer contains is a WPF-Application containing different views which all have abstract representations in the form of view models. The main view model contains instances of all other view models and implements the required logic by calling functions of Business-layer classes with the Controller Factory. An UML representation can be seen in Picture 2. The App class sets the instances of the main view model and sets the same instances as data-context in the main window.

Text

Description automatically generated with medium confidence

Picture 3: Edit Tour Log Dialog

In addition, the PL contains a window dialog for editing tour logs. The UML representation can be seen in Picture 3.

Graphical user interface, text, application, chat or text message

Description automatically generated

Picture 4: PL Helper Classes

Finally, there are some helper classes, one of which is an implementation of ICommand and the other is used to convert png files to Bitmaps (Picture 4).

## Business-layer

Graphical user interface, text, application, chat or text message

Description automatically generated

Picture 5: Controller Factory

The core functionality of the BL can be accessed by using the controller factory (Picture 5) to generate instances of controllers (Picture 6). The factories handle dependency injection into the controllers.

Graphical user interface, text, application, chat or text message

Description automatically generated

Picture 6: Controller

The controllers expose the functionality of the BL to the PL. They implement the IDisposable interface, so the unmanaged resources of the BL and DAL can be released properly. The controllers utilize several other classes which can be seen below (Picture 7).

Graphical user interface, application

Description automatically generated

Picture 7: Classes used by controller

## Data-access-layer

Graphical user interface, text, application, chat or text message

Description automatically generated

Picture 8: Unit of work and repository

The DAL provides an implementation of the repository pattern in combination with the unit of work pattern (Picture 8). There is also a mock implementation of the unit of work interface which uses an in-memory DB to allow for easier unit testing.

Application

Description automatically generated with medium confidence

Picture 9: DB context

The context classes (Picture 9) provide methods to load data from various data sources into memory, depending on the implementation.

Graphical user interface, text, application, chat or text message

Description automatically generated

Picture 10: In memory mock DB and ORM

For the production DB an ORM is used and for testing purposes the DBMock class is utilized. Both classed provide CRUD operations to manipulate data. See Picture 10.

Text

Description automatically generated with medium confidence

Picture 11: DAL exceptions

The DAL also defines its own exceptions (Picture 11) to not make other layers dependent on the custom exceptions of frameworks used by the DAL.

## Model-layer

Graphical user interface, application

Description automatically generated

Picture 12: Models

The model-layer contains an interface which specifies a contract of required properties for models which you want to be able to be used by the DAL. Besides a model for tours and tour logs there is also another model for the relation of tour-to-tour logs, which extends the tour class and adds a list of all linked tour logs for that tour. This class is used for the import and export functionality of the tour planner app. An overview of all the properties can be seen in Picture 12.

## Common layer

Graphical user interface, text, application, chat or text message

Description automatically generated

Picture 13: Classes of common layer

The common layer contains classes which are needed by all other layers. It contains access to classes which handle config files and a logging framework, in this case Log4Net. The classes can be seen in Picture 13.

# Use Case Diagram

Diagram

Description automatically generated

Picture 14: Tour planner use case diagram

# Wireframes

The following images display the different views of the tour planner. To make it more descriptive, some example data has been added.

Graphical user interface, application, Word

Description automatically generated

Picture 15: Tour planner main view

Graphical user interface, application

Description automatically generated

Picture 16: Image view of tour

Graphical user interface, application

Description automatically generated

Picture 17: View of menu bar

Graphical user interface, text, application

Description automatically generated

Picture 18: View of edit tour log dialog window

# Sequence diagram of text search

Diagram

Description automatically generated

Picture 19: Sequence diagram of text search

# Unit testing decisions

To allow true unit tests and not integration tests, external dependencies have been mocked out, which includes both the DB and any API calls.

The most critical code tested is the mock DB, because it is also part of the unique feature which is described below.

# Time tracking

We tracked the time spent with the project in a text file (Picture 20: Time tracking) which you can see below this paragraph. Since most of the project was coded in pair-programming we did not concern ourselves with individual workloads.

Text

Description automatically generated

Picture 20: Time tracking

# Git link

<https://github.com/MichaelDusk2361/TourPlanner>

# Design patterns

## Unit of work

The unit of work pattern has been implemented in the DAL and is used to group CRUD operations into a single transaction (unit of work), so that all operations either fail or pass.

## Repository

Repositories encapsulate the logic required to access data sources. In our case the data source is loaded via a data-context class which provides methods to retrieve tables from a DB. Then the data is manipulated with the repository CRUD operations. Then the unit of work class starts a transaction and tells the context to save all changes.

The repository used in the tour planner is generic and can be used with any model that inherits from the IEntity interface.

## Factory

A factory is a class that creates other classes. Factories can therefore also handle dependency injection, as can be seen in Picture 21. This is very convenient, because if you require a different class to be injected, only one line of code needs to be changed in the entire program. Alternatively, a new implementation of the factory interface can be created.

Text

Description automatically generated

Picture 21: Factory implementation

## MVVM

This pattern has been covered during the lecture throughout and does not require further elaboration.

# Unique feature

The unique feature is, that in case of a connection failure to the DB during startup, the app still starts but without a DB. All features work as intended and routes can be still planned, data exported and imported, and so on. To persist any work the user has done, the OnExit function of the app has been overridden, and will export all data to a json file as soon as the user closes the application.