

TourPlanner Protokoll

Burak Konca

BIF4 4A1 – SWE2

Inhalt

[Technische Schritte 3](#_Toc73547853)

[Design 3](#_Toc73547854)

[Layer 3](#_Toc73547855)

[Architectural Decisions 10](#_Toc73547856)

[UX Decisions 10](#_Toc73547857)

[Library Decisions 10](#_Toc73547858)

[Fehlschläge 10](#_Toc73547859)

[Lösungsansätze 11](#_Toc73547860)

[Unit Tests 12](#_Toc73547861)

[Unique Feature 13](#_Toc73547862)

[Tracked time 14](#_Toc73547863)

[Git Repository link 14](#_Toc73547864)

[Abbildungsverzeichnis 15](#_Toc73547865)

Protokoll

# Technische Schritte

## Design

Wie in der untenstehenden Abbildung zu sehen ist, verfügt die TourPlanner Projektmappe über sieben Projekte.

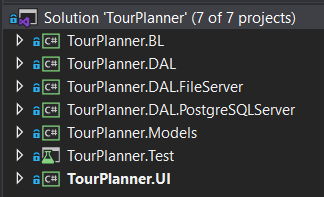


Abbildung : Projektmappe

Unter Anderem ist der TourPlanner gegliedert in einen Business Layer (BL), Data Access Layer (DAL), der Grafischen Benutzeroberfläche (GUI), den einzelnen Modellen und einem Testprojekt zum Schreiben von Unit Tests etc.

### Layer

#### Business Layer

Der Business Layer ist der Teil des Programms, der die realen Geschäftsregeln codiert, die bestimmen, wie Daten erstellt, gespeichert und geändert werden können. Es schreibt vor, wie Geschäftsobjekte miteinander interagieren, und erzwingt die Routen und Methoden, mit denen auf Geschäftsobjekte zugegriffen und diese aktualisiert werden.

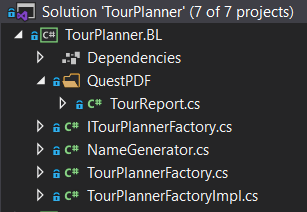


Abbildung 2: BL Projekt

In diesem Projekt habe ich beschlossen das **Factory Pattern** anzuwenden. Die Factory-Pattern-Methode definiert dabei ein Interface für die Erstellung von Objekten und delegiert die Objekterstellung an die Sub-Klassen. Ziel ist, dass der Client nichts von der Objekt-Instanziierung mitbekommt und über eine gemeinsame Schnittstelle auf die Objekte zugreift.

Daher beinhaltet der Business Layer bei mir folgende drei Klassen, die dem Factory Pattern angehören:

* **ITourPlannerFactory**: Schnittstelle zur Definition der Struktur der Funktionalitäten/Methoden in Bezug auf die Touren und Tourlogs. Diese Klasse definiert die Strukturen für folgende
* **TourPlannerFactory**: Stellt durch den Einsatz des **Singleton** Patterns nur eine einzige Instanz der Factory zur Verfügung auf, die dann zugegriffen werden kann.
* **TourPlannerFactoryImpl**: Enthält die Implementierung der im Interface ITourPlannerFactory festgelegten Methodenstrukturen.

Des Weiteren gehören noch zwei weitere Klassen dem Business an und zwar eine Klasse **NameGenerator**, welche für Testzwecke verwendet wurde und um den Aufwand zum Erstellen von Touren und TourLogs zu minimieren. Und einer Klasse **TourReport**, welche für die Erstellung eines PDF Tour Reports dient, die Infos zu einer bestimmten Tour, einem Bild der Route und der dazugehörigen TourLogs beinhaltet.

#### Data Access Layer

Ein Data Access Layer (DAL) ist eine Schicht eines Computerprogramms, die einen vereinfachten Zugriff auf Daten bereitstellt, die in irgendeiner Art von Dauerspeicher gespeichert sind, wie beispielsweise einer Datenbank oder einem lokalen Filesystem.

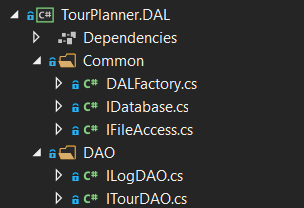


Abbildung : DAL Projekt

In meiner Anwendung beschloss ich zwei Zugriffsmöglichkeiten für Daten bereitzustellen. Und zwar über eine Datenbank und einem lokalen Filesystem.

Das Ganze läuft abstrahiert ab, indem die beiden Zugriffsmöglichkeiten ihre Gemeinsamkeiten von entsprechenden Interfaces **IDatabase** für den Datenbankzugriff und **IFileAccess** für den Filesystemzugriff erben und wie sie zum Einsatz kommen wird in der Klasse **DALFactory** klar festgelegt. Die DALFactory entscheidet anhand einer Konfigurationsdatei, welche Zugriffsart erfolgen soll.

Des Weiteren muss die Struktur der Methoden der Data Access Objects (DAO) festgelegt werden. Dies erfolgt in den entsprechenden Klassen **ILogDAO** und **ITourDAO**.

##### FileServer

Eine der beiden Unterteilungen des Data Access Layers ist der FileServer. Das FileServer Projekt ist für Handlen des Zugriffs auf das lokale FileSystem zuständig.

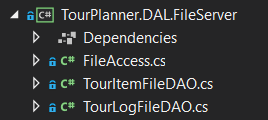


Abbildung : FileServer Projekt

Das Projekt besteht aus drei Klassen. Einer Klasse **FileAccess**, welche für das Erzeugen von Textdateien für Touren und TourLogs zuständig ist, als auch das Erstellen des Routen Bildes durch **integration der MapQuest API.** Des Weiteren gibt es die Klasse **TourItemFileDAO**, welche für das Interagieren von Touren mit dem FileSystem zuständig ist, um zum Beispiel Touren von einer Datei einzulesen, etc.

Natürlich müssen dieselben Funktionalitäten auch für die TourLogs bereitgestellt werden. Dies wird dann in der **TourLogFileDAO** Klasse bereitgestellt.

##### PostgreSQLServer

Die Zweite Unterteilung des Data Access Layers ist der PostgreSQLServer. Das PostgreSQLServer. Das PostgreSQLServer Projekt ist für Handlen des Zugriffs auf die Datenbank zuständig.

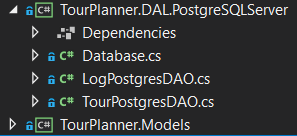


Abbildung : PostgreSQLServer Projekt

Das Projekt besteht aus drei Klassen. Einer Klasse **Database**, welche für das Aufbauen der Datenbankverbindung und erstellen und durchführen von SQL Statements zuständig ist. Die Klassen **LogPostgresDAO** und **TourPostgresDAO** sind für die Datenbankabfragen bezüglich Touren und TourLogs zuständig.

#### Models

In dem Projekt Models werden die einzelnen Modelle der Objekte abgespeichert.

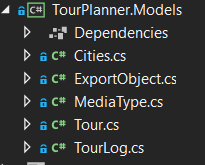


Abbildung 6: Models Projekt

Hier habe ich zunächst eine Klasse **Cities** wurde für Komfortzwecke entwickelt. Sie beinhaltet ein Enum von Orten, das in der weiteren Abfolge der Anwendung zur schnellen Erstellung von zufälligen Touren verwendet wurde.

Die Klasse **ExportObject** ist im Grunde eine Klasse, die ein Objekt erzeugt, welches aus einer Tour und einer Liste von ihren zugehörigen TourLogs besteht. Diese Klasse wird, wie der Name schon sagt für das Exportieren einer Tour mit ihren TourLogs in eine Datei verwendet.

Die Klasse **MediaType** ist eine Helper class, das nur aus einem Enum besteht, welches im FileAccess seine Verwendung findet.

Die **Tour** Klasse und die **TourLog** Klasse gehören zu den Core Models der Applikation. Daher macht es auch Sinn hier nochmal kurz ihre Eigenschaften zu veranschaulichen:

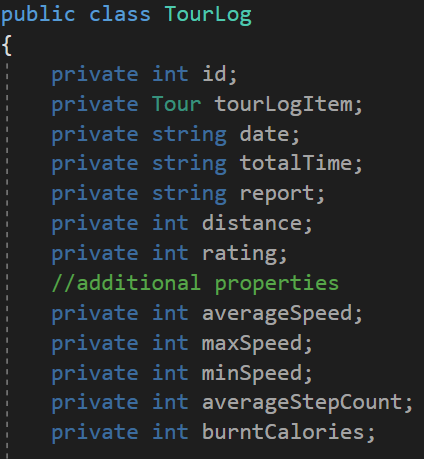
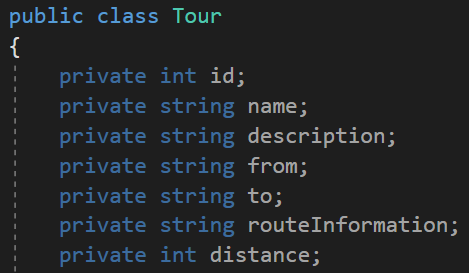


Abbildung : Tour Klasse

Abbildung : TourLog Klasse

#### GUI

Das GUI Projekt ist für die Bereitstellung der grafischen Benutzeroberfläche verantwortlich. Bei der Herangehensweise der Entwicklung der GUI wurde das **Model View ViewModel (MVVM)** Entwurfsmuster angewandt. Daher erfolgte die Unterteilung des Projektes in ViewModels und Views und den Models selbst, welche sich aber außerhalb der GUI im Model Projekt befinden.

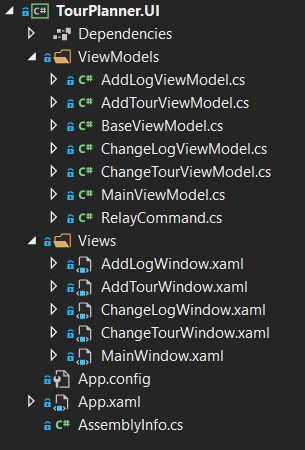


Abbildung 9: UI Projekt

Das **Model** ist hierbei die Datenzugriffsschicht für die Inhalte, die dem Benutzer angezeigt und von ihm manipuliert werden. Diese wurden im Model Project bereits erläutert.

Die **Views** sind alle durch die grafische Benutzeroberfläche (GUI) angezeigten Elemente. Im TourPlanner kamen bei mir folgende Views zum Einsatz:

* MainWindow

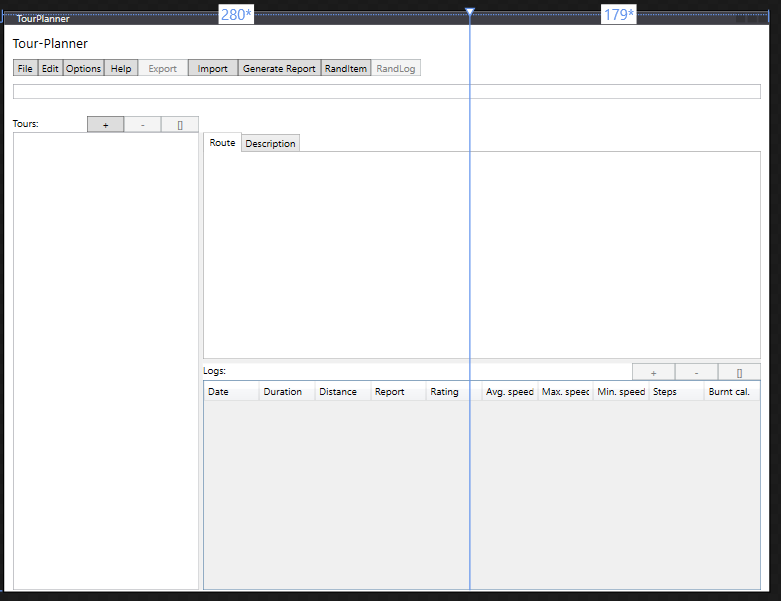


Abbildung 10: MainWindow

* AddLogWindow

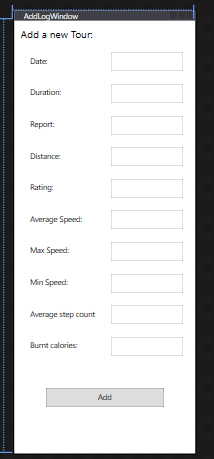


Abbildung 11: AddLogWindow

* AddTourWIndow

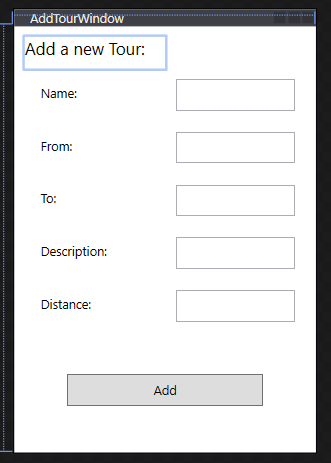


Abbildung 12: AddTourWindow

* ChangeLogWindow

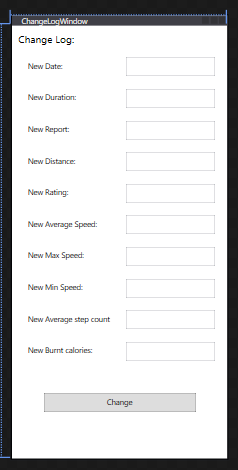


Abbildung 13: ChangeLogWindow

* ChangeTourWindow

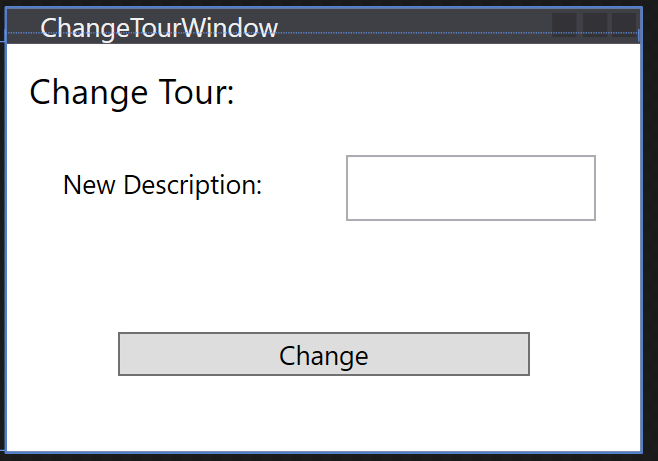


Abbildung 14: ChangeTourWindow

Die **ViewModels** enthalten die UI-Logik (Model der View) und dienen als Bindeglied zwischen den Views und den Models. Bei meinen TourPlanner Projekt kamen ViewModel jeweils für das Hinzufügen von TourLogs, Hinzufügen von Touren, Ändern einer Tour, Ändern eines TourLogs zum Einsatz. Das MainViewModel, darf hierbei auch nicht vergessen werden und das BaseViewModel, welches eine abstrakte Klasse ist, die das RaisePropertyChangedEvent festlegt von der die anderen ViewModels erben. Zuletzt gibt es noch die Klasse RelayCommand, welche das Problem löst, für jeden Befehl eine neue Klasse erstellen zu müssen, da die Befehle sonst nicht über unterschiedliche CanExcecute- oder Execute-Methoden verfügen würden.

##### Config

Das TourPlanner Projekt beinhaltet auch ein Config File, in dem die Konfigurationseinstellungen fürs Logging und die connectionStrings für die DALSQLAssembly und DALFileAssembly festgelegt werden. Bei den connectionStrings werden lediglich für die Datenbank der connectionString festgelegt und für das FileSystem das Verzeichnis.

#### Test

Das Test Projekt beinhaltet MOCK Klassen und die Unittests, welche im Detail in weiterer Folge dieses Protokolls beschrieben werden.

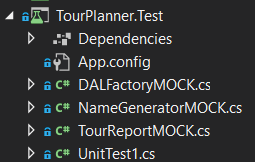


Abbildung 15: TourPlanner Test

### Architectural Decisions

Beim Architektur pattern fiel die Wahl auf das Model View ViewModel (MVVM) pattern, welches eine Variante vom Model View Controler (MVC) ist. MVVM dient zur Trennung von Darstellung und Logik der Benutzerschnittstelle (UI). Des Weiteren kamen Patterns wie das Factory Pattern und Singleton bei meinem TourPlanner zum Einsatz. Das Factory Entwurfsmuster beschreibt, wie ein Objekt durch Aufruf einer Methode anstatt durch direkten Aufruf eines Konstruktors erzeugt wird und das Singleton Pattern stellt sicher, dass von einer Klasse genau ein Objekt existiert.

### UX Decisions

Bezüglich UX wurde der TourPlanner sehr benutzerfreundlich gestaltet. Es verfügt über eine einheitliche GUI, bei der es grundsätzlich für keine Verwirrung oder sonstigen Komplikationen kommen sollte.

### Library Decisions

Folgende Bibliotheken kamen bei meinem TourPlanner Projekt zum Einsatz.

* **Newtonsoft**: Zum Serialisierun und Deserialisiern von Objekten. Wurde aus dem Grund ausgewählt, weil es die am häufigsten verwendete Bibliothek zum JSON Serializen ist
* **Nunit**: Um Unit Tests zu schreiben. Der Grund für die Verwendung war der, dass ich bis jetzt immer Nunit verwendet habe und mich damit wohl fühle.
* **Log4net**: Zum Loggen von diversen Informationen. Wurde im Unterricht behandelt und galt als Pflicht Log4Net zu verwenden in der C# Umgebung
* **QuestPDF**: Zum Generieren von PDF Reports. Wurde verwendet, weil es sehr einfach integrierbar und nutzbar ist.
* MapQuest: Zum Generieren eines Bildes einer Route. Wurde im Unterricht behandelt und galt als Pflicht zu verwenden.

## Fehlschläge

Während der Entwicklung des TourPlanners bin ich auf viele Fehlschläge gestoßen. Darunter würde unter Anderem gehören, dass ich Komplikationen hatte TourLogs auf die PostgreSQL Datenbank zu speichern, aufgrund dessen, dass PostgreSQL Spalten einer Tabelle anders erzeugt, wenn sie groß und Kleinschreibung beinhalten.

Des Weiteren hat das Exportieren von Touren im JSON Format ein wenig für Ärgernisse gesorgt, da ich letztendlich nur die Tour selbst ohne ihre TourLogs serialisieren und in einer JSON Datei abspeichern kann.

Mit dem Anzeigen des Images der Route einer Tour in der GUI hatte ich auch eine Zeit lang Probleme, bis ich dann einen anderen Ansatz probiert habe, und zwar eine Bitmap mit dem angegebenen Pfad zu generieren.

## Lösungsansätze

Wie im in den obigen Fehlschlägen schon erwähnt hatte ich Schwierigkeiten mit der Darstellung der Route mit dem per MapQuest API generierten Bild. Gelöst habe ich das Problem dadurch, dass ich den Pfad des abgespeicherten Images einlese und eine Bitmap daraus generiere.

Den oben beschriebenen Fehler mit der PostgresSQL Datenbank habe ich gelöst, indem ich die Tabelle TourLogs gelöscht hab und die Tabelle nochmal erstellt habe, jedoch diesmal mit allen Spaltennamen klein geschrieben.

Das Problem mit dem Exportieren konnte ich bis zum Schluss leider nicht lösen, also es wird nur die Tour ohne ihre Logs exportiert.

# Unit Tests

In diesem Projekt wurde TDD betrieben und insgesamt 20 Unit Tests erstellt, von denen aber drei zuletzt nicht im Code implementiert wurden.

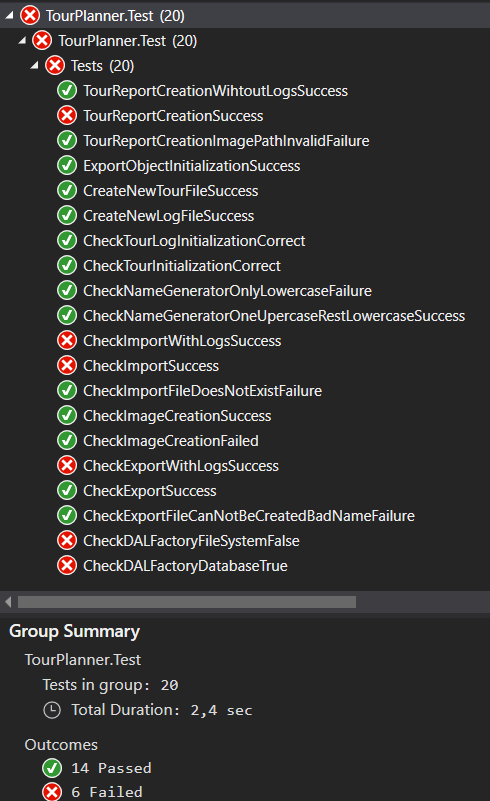


Abbildung 16: Unit Tests

Es wurden insbesondere Unit Tests gewählt, die auf kritischen Code bezogen sind, unter anderem für:

* Korrekte Initialisierungen der Modell Objekte. Kritisch, weil, die Initialisierung problemlos erfolgen muss, damit alles weiteren Schritten richtig verlaufen können.
* Generieren der Bilder von den Tour Routen via MapQuest API. Dieser Code ist kritisch, weil, durch falsche Angaben der Ortsinformationen wie zb einem Rechtschreibfehler etc, können Fehler bei der Durchführung auftreten und das Bild falsch generiert werden.
* Generieren der Tour Reports via QuestPDF API.
* Erstellen von Touren und TourLogs in Textdateien im Falle der Benützung des Filesystems als Zugriffsmöglichkeit. Kritisch, weil, wenn nicht valide Parameter übergeben werden, das zum Nichterstellen der Datei führen kann etc.
* Erstellen von zufälligen Namen durch den NameGenerator
* Feststellen der DALFactory welche Zugriffsmöglichkeit im Einsatz ist anhand der Config Datei.
* Exportieren und Importieren der Touren und TourLogs in/von JSON Dateien. Kritisch, weil, wenn nicht valide Parameter übergeben werden, das zum Nichterstellen beziehungsweise falschem Erstellen der JSON Datei führen kann oder schlechte Dateinamen zu Abstürzen und Fehlern führen kann etc.

# Unique Feature

Als Unique Feature für das TourPlanner Projekt habe ich beschlossen mithilfe von Document Annotations eine Website mit Doxygen zu erstellen, die dann einen Überblick der Klassen, Methoden, Struktur und Funktionalitäten gewährt.

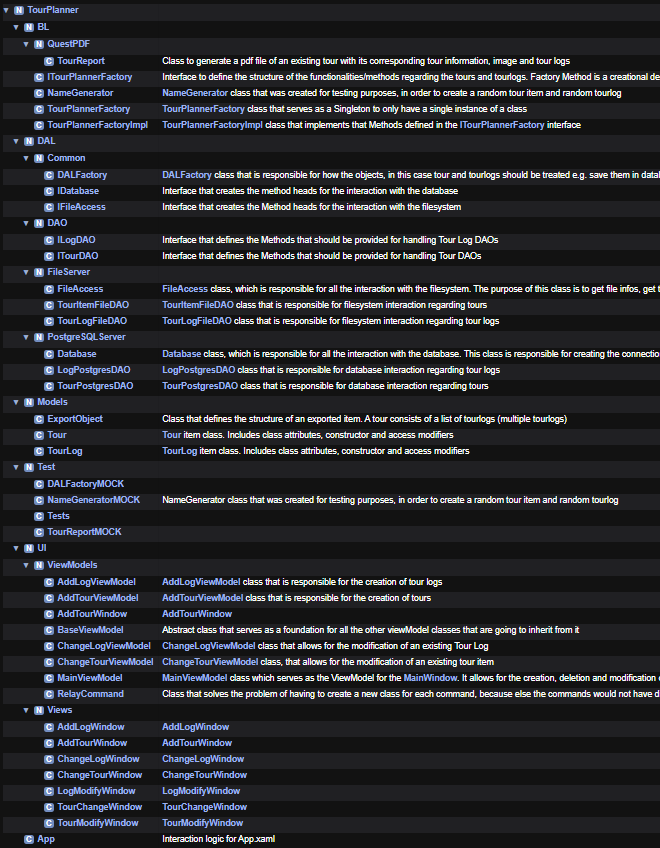


Abbildung 17: Doxygen Projekt Struktur

# Tracked time

Das gesamte TourPlanner Projekt hat mich insgesamt einen Aufwand von ungefähr 70 Stunden gekostet, da ich mehrmals beim mergen bzw commiten Schwierigkeiten hatte beziehungsweise auch korrupte Dateien hatte, die mir die Solution zerschossen haben.

Es ist unter anderem sehr viel Zeit in das Erstellen der Struktur geflossen, Da zunächst einmal das gesamte Prinzip mit dem DAL verstanden werden musste. Danach fiel das Entwickeln der anderen Komponente nicht allzu schwer, bis auf die GUI, die mich auch viel Zeit und Nerven gekostet hat, insbesondere die einzelnen Elemente richtig zu binden und mit den ViewModels und miteinander zu verlinken.

# Git Repository link

In diesem Abschnitt wird der Link zum GitHub Repository zur Verfügung gestellt. Ich würde aber gerne anmerken, dass ich das Projekt insgesamt drei Mal neu geschrieben hab und das mein dritter Versuch war, daher ist auf diesem GitHub Repository nicht alle Commits zu sehen, sondern erst ab dem Stand wo ich das drei-Stündige Tutorial fertig befolgt hatte.

<https://github.com/Avokaci/TourPlannerrr>

# Abbildungsverzeichnis

[Abbildung 1: Projektmappe 3](#_Toc73547737)

[Abbildung 2: BL Projekt 3](#_Toc73547738)

[Abbildung 3: DAL Projekt 4](#_Toc73547739)

[Abbildung 4: FileServer Projekt 5](#_Toc73547740)

[Abbildung 5: PostgreSQLServer Projekt 5](#_Toc73547741)

[Abbildung 6: Models Projekt 5](#_Toc73547742)

[Abbildung 7: Tour Klasse 6](file:///C:\Users\burak\OneDrive\Desktop\TourPlannerrr\TourPlanner_Protocol.docx#_Toc73547743)

[Abbildung 8: TourLog Klasse 6](file:///C:\Users\burak\OneDrive\Desktop\TourPlannerrr\TourPlanner_Protocol.docx#_Toc73547744)

[Abbildung 9: UI Projekt 6](#_Toc73547745)

[Abbildung 10: MainWindow 7](#_Toc73547746)

[Abbildung 11: AddLogWindow 8](#_Toc73547747)

[Abbildung 12: AddTourWindow 8](#_Toc73547748)

[Abbildung 13: ChangeLogWindow 9](#_Toc73547749)

[Abbildung 14: ChangeTourWindow 9](#_Toc73547750)

[Abbildung 15: TourPlanner Test 10](#_Toc73547751)

[Abbildung 16: Unit Tests 12](#_Toc73547752)

[Abbildung 17: Doxygen Projekt Struktur 13](#_Toc73547753)