TourPlanner – Protokoll

# Architektur

Mein TourPlanner-Projekt ist in 8 Unterprojekte aufgeteilt, sprich 8 Assemblys. Kernbestandteil des Projekts ist die „TourPlanner.Core“ Assembly. Diese enthält die projektübergreifend verwendeten Models und definiert Interfaces, die von Klassen in weiteren Assemblys dann implementiert werden können. Zudem stellt sie auch sogenannte „DataManager“ bereit, welche vom Consumer verwendet werden können, um den Zugriff auf die spezifischen Datenzugriffsklassen weiter zu abstrahieren. DataManager stellen somit effektiv den Business Layer dar. Des weiteren enthält die Assembly auch noch Klassen zur Speicherung von Konfigurationen und interne Implementierungen vom Änderungserkennungsmechanismus.

Die weiteren Assemblys stellen Implementierungen der in TourPlanner.Core spezifizierten Interfaces dar. (TourPlanner.Converters.Json 🡪 IDataConverter; TourPlanner.DataProviders.MapQuest 🡪 IDirectionsProvider, IMapImageProvider; TourPlanner.DB.InMemory, TourPlanner.DB.Postgres 🡪 IDatabaseClient, IDatabaseClientFactory; TourPlanner.Reporting.PDF 🡪 IReportGenerator). TourPlanner.Tests beinhaltet die Unit Tests. TourPlanner.GUI beinhaltet die WPF-Anwendung mit der Benutzerschnittstelle.

Dies ermöglicht besonders viel Flexibilität, wenn es darum geht, einzelne Komponenten auszutauschen. Alle Interfaces wurden so gestaltet, das sie auch einfach für weitere Implementierungen wiederverwendet werden können (z.B. IDataConverter verlangt deshalb auch einen Anzeigenamen und eine präferierte Dateiendung).

Diese ganzen Projekte müssen jedoch an einer Stelle zusammenlaufen. Dies wäre normalerweise bei der Dependency Injection. Da ich bei diesem Projekt jedoch kein DI Framework verwendet habe, habe ich stattdessen die „DependencyInitializer“-Klasse im TourPlanner.GUI-Projekt, welche mir die Instanzen der konkreten Implementierungen (meist Factorys) anlegt.

# Design Patterns

In diesem Projekt habe ich mehrere Design Patterns verwendet. Einerseits natürlich das „Command“ Pattern, welches Teil von MVVM ist. Hierfür habe ich auch noch eine „RelayCommand“-Klasse erstellt, um einfache Befehle ohne eigene Klasse umsetzen zu können. (Zudem habe ich auch noch ein NuGet-Package geladen, welches mir eine asynchrone Variante davon bereitstellt, da meine gesamte Anwendung komplett asynchron läuft.)

Ich habe auch das „Factory Method“ Pattern beim Anlegen von DataManagern in TourPlanner.Core.DataManagers.DataManager eingesetzt.

An mehreren Stellen habe ich aber die umfangreichere Variante davon eingesetzt, das „Abstract Factory“ Pattern: IDatabaseClientFactory, IDataProviderFactory und implementierende Klassen.

# Unit Testing

Da meiner Meinung nach der komplizierteste Teil des Projekts der Änderungserkennungsmechanismus ist habe ich beschlossen, an dieser Stelle ausführlich zu testen. Ich wurde zudem auch von WPF genötigt, dies zu tun, da ich aufgrund von kleinen Bugs in meinen früheren Implementierungsversuchen recht schnell Exceptions erhalten habe, dass etwas mit der im DataGrid hinterlegten Liste nicht stimme. Diese Exceptions waren übrigens überraschend detailreich.

Aus diesem Grund teste ich lediglich die „ChangeTrackingCollection“ und „ChangeTrackingList“ Klassen. Über das gesamte Projekt hinweg waren diese durchwegs problembehaftet aber gleichzeitig auch eine absolut wichtige Funktionalität, da ich in beim Senden an die Datenbank hierauf vertraue. (IDatabaseClient hat nur eine Read und eine Write Methode; letztere macht Nutzen von der ChangeTrackingCollection.)

# Unique Feature(s)

Als “unique Feature” könnte ich mehrere Dinge nennen:

* Der Änderungserkennungsmechanismus, der es mir ermöglicht, den Benutzer vor dem Schließen der Anwendung bei ausstehenden Änderungen zu warnen. (Leider nicht allzu bemerkenswert für den Aufwand…)
* Der Dark Mode, welcher während der Laufzeit aktiviert und deaktiviert werden kann. (Selbst während den Ladezeiten per Tastaturshortcut!)
* Das Anzeigen der einzelnen „Steps“ (Maneuvers) der Strecke inklusive passenden Icons. (Auch von MapQuest Directions API erhalten)

# Challenges

## Änderungserkennungsmechanismus

Dieser Teil wurde in den vorherigen Sektionen schon mehrfach erwähnt. Der Änderungserkennungsmechanismus (change tracking mechanism) erkennt, wenn der Benutzer Objekte geändert hat und leitet beim Synchronisieren mit der Datenbank auch nur diese weiter.

Dies basiert auf „System.ComponentModel.IChangeTracking“, einem Interface, das ähnlich zu INotifyPropertyChanged funktioniert, jedoch mit dem Unterschied, dass man die Abfrage nach Änderungen manuell durchführt (sprich, beim Speichern oder Schließen des Programms). Alle Models mit mutierbaren Eigenschaften (LogEntry und Tour) implementieren dieses Interface, wodurch ihr Zustand nachverfolgt werden kann. Jedoch brauchte ich zusätzlich noch eine Möglichkeit, um neue und entfernte Einträge festzuhalten. Daher habe ich das Interface IChangeTrackingCollection erstellt und eine interne Implementierung hinzugefügt. (Der DataManager wrappt die von der Datenbank erhaltene Collection automatisch in der ChangeTrackingCollection.)

Jedoch machte sich ein weiteres Problem auf: Das DataGrid von WPF kann nicht mit Collections umgehen und benötigt unbedingt eine Liste. (Wahrscheinlich, da die Position der Einträge vorhersehbar sein muss, was bei meiner ChangeTrackingCollection nicht der Fall war.) Daher habe ich auch noch eine IChangeTrackingList definiert und dafür eine weitere Implementierung bereitgestellt. Da WPF allerdings etwas älter ist unterstützt es IList<T> nicht, sondern nur IList, was den Entwicklungsaufwand zusätzlich etwas erhöhte. Die Entwicklung der ChangeTrackingList war jedoch grundsätzlich erheblich aufwendiger als die der ChangeTrackingCollection, da ich alle Einträge in einer einzigen Liste speichern musste, um die Position zu bewahren. (Bei der ChangeTrackingCollection hatte ich es einfacher, da ich drei separate Listen verwenden konnte, weil die Positionierung egal war.) Der schwierigste Punkt hierbei war, sicherzustellen, dass die Indizes richtig umgerechnet wurden, da bereits gelöschte Einträge bis zum nächsten Speichervorgang immer noch in der Liste vorhanden waren aber ignoriert werden mussten. (Außer für den Fall, dass ein bereits gelöschter Eintrag wieder hinzugefügt wird, was die Änderung rückgängig macht.)

## Dark Mode

Der Dark Mode war vergleichsweise einfach umzusetzen. Ich konnte hierbei auf ein NuGet-Package zurückgreifen, welches mir einen großen Teil der Arbeit abgenommen hat, ich musste lediglich die entsprechenden Ressourcen hinzufügen und ein paar kleine Änderungen am XAML-Code machen und die neuen Styles haben direkt funktioniert. Für den Wechsel zwischen Light und Dark Mode muss dann nur mehr der ResourceLocator des Packages notifiziert werden.

Das NuGet-Package (“AdonisUI”) kommt zudem auch noch mit einer eigenen MessageBox-Implementierung, welche ebenfalls den Dark Mode unterstützt, sodass die Anwendung auch hierbei einheitlich wirkt. Außerdem sind auch unterschiedliche Ladeanimationen enthalten, eine davon habe ich im Ladescreen verwendet.

## Komplette asynchrone Codeausführung

Das gesamte Programm ist durchgängig asynchron. Da ich aus vergangenen Projekten bereits weiß, dass es wirklich unangenehm ist, synchronen und asynchronen Code zu mischen habe ich darauf geachtet, dass in diesem Projekt überall, wo es möglich ist, asynchroner Code verwendet wird.

Bei einigen Interfaces habe ich Methoden als Task<…> oder ValueTask<…> definiert, obwohl ich wusste, dass die dahinterliegende Implementierung nur synchronen Code beinhielt. Dies habe ich deshalb gemacht, weil ich wusste, dass es in Zukunft möglich sein könnte, dass diese Implementierungen Nutzen von Asynchronität machen. Somit könnten diese neuen Implementierungen nun asynchronen Code nutzen, ohne, dass ich das Interface dazu abändern muss.

Genau dies ist auch passiert. „IReportGenerator.GenerateTourReport“ hat ursprünglich komplett synchron gearbeitet, aber als ich schlussendlich das Laden des Routenbildes hinzugefügt habe, habe ich einfach asynchronen Code hinzufügen können, ohne andere Teile des Programms verändern zu müssen. Sollte QuestPDF in Zukunft auch einen asynchronen Speichervorgang unterstützen, wäre es somit auch kein Problem, diesen aufzurufen.

## Manuelle Dependency Injection

Ich habe in diesem Projekt kein DI-Framework verwendet und musste daher einige Schritte manuell durchführen. Grundsätzlich habe ich zwar schon ein bisschen Erfahrung mit Microsoft.Extensions.DependencyInjection aus diversen ASP.NET Core Projekten, jedoch wollte ich es dieses Mal selbst machen. Dies hat zwischenzeitlich dazu geführt, dass ich vom ViewModel abgeleitet habe, um alle Dependencys in der Unterklasse anzulegen. Mittlerweile habe ich das Anlegen der Abhängigkeiten in die Klasse „DependencyInitializer“ verschoben. Diese legt mir, natürlich komplett asynchron, die Abhängigkeiten an. Dies kann entweder „normal“ gemacht werden oder auch zum Testen mit einer InMemory-Datenbank.

## Darstellung von Werten passend zur Region des Nutzers

Eine Sache, die mich an WPF etwas störte, war, dass Bindings immer die CurrentCulture des Threads ignoriert haben und ich die Formatierung manuell korrigieren musste. Ich habe dann jedoch herausgefunden, dass sich dies über eine kleine Anpassung beim Anwendungsstart beheben lässt (siehe App.xaml.cs).

An manchen Stellen habe ich dennoch manuelle Anzeigeformatierungen vornehmen müssen, um die Darstellung natürlicher wirken zu lassen. (Z.B. bei der Darstellung von Prozentwerten als 'XX %' über den 'P0' Formatstring, welcher sich aber von double.Parse nicht einlesen lies und wiederrum einen eigenen Converter brauchte.)

# Summary

Ungefähre aufgewendete Zeit: 90h (nur Coding und Debugging, laut Zeittracker Plugin)