Tourplanner Dokumentation

Design

Das Programm ist in GUI Layer, Business Layer und Data Access Layer aufgeteilt. Außerdem wurden die Models auf einen eigenen Layer ausgelagert. Eine saubere Aufteilung der der Funktionalitäten auf ihre zugehörigen Layer ist mir leider aus Zeitmangel nicht gelungen, es finden sich also ab und an Funktionen welche eigentlich in den Businesslayer gehören, sowohl im GUI Layer als auch im DAL. Hier erkannte ich auch meine erste „Lesson learned“ des Projekts:

> Nimm dir (vor allem als Laie) Zeit, dich vor dem coden mit der Architektur zu beschäftigen. Trenne klar ab welche Funktionen gebraucht werden. Oftmals funktioniert anfangs auch unsauberes coden, bis es das nicht mehr tut.

Für den Datenaustausch und die Navigation wurde das MVVM Pattern verwendet. Hierbei gelang mir eine sauberere Abgrenzung und der Code Behind blieb so gut wie leer. Das war mitunter genügender Recherche und vorhandener Erfahrung zu verdanken. Im Vergleich zur Layer-Abgrenzung sind bei MVVM die Grenzen was wohin gehört klarer definiert und daher leichter verständlich.

GUI Layer

Bei der Gestaltung der Nutzeroberfläche verfolgte ich einen eher minimalistischen Ansatz und versuchte Simplizität und Funktion in den Vordergrund zu stellen.
Der Aufbau enthält eine Mainview mit einem Bereich für das Darstellen von Content. Dieser Content wird über User Controls gesteuert, welche mit Bindings an Buttons gebunden sind (über ICommands). Als Grundlage der Viewmodels wurde eine baseVM erstellt welche das Interface INotifyPropertyChanged implementiert. Dadurch konnten innerhalb der Views Elemente in Runtime aktualisiert werden.

Business Layer

Da ich, wie bereits erwähnt, den Durchblick bezüglich Layer Aufteilung nicht von Anfang an hatte ist der Businesslayer eher leerer als es mir recht ist. Zumindest gegen Ende des Projekts wurden alle Funktionen über das Factory Pattern im Businesslayer angesiedelt. Ein Refactoring der Layer wäre definititv mein nächster Schritt gewesen, hätte ich mehr Zeit in das Projekt investiert.

Datenbank

Als Datenbank wurde wie empfohlen PostgreSQL als DBMS verwendet und über pgadmin4 verwaltet. Als Library für den Zugriff auf die Datenbank wurde das open source Tool Npgsql verwendet.

Beim Schreiben von Queries welche User Input enthielten wurde stets mit Parametern (@p) gearbeitet. Das brachte mit sich, dass SQL Injections praktisch ausschließbar wurden, da die Parameter separat an PostgreSQL geschickt werden und nie als SQL interpretiert werden.

Library Decisions

- Npgsql: Empfehlung Angabe
- Log4net: Empfehlung Angabe
- Newtonsoft.Json: Hier hatte ich bereits Erfahung mit der Verwendung der Library, daher fiel die Entscheidung leicht.
- iText7: Nach ein paar frustrierenden Anläufen mit Crystal Reports hat mir ein Kollege iText ans Herz gelegt. iText hat eine weitaus ausführlichere Dokumentation als vergleichbare Libraries.

Unit Test Design

Hier findet sich wohl die größte Lehre, die ich aus dem Projekt gezogen habe:

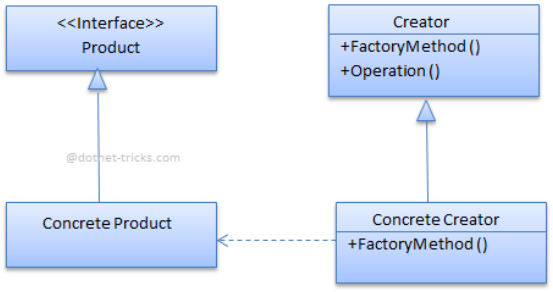
> Manches lässt sich nicht testen, wenn es nicht test-freundlich entwickelt wurde. Überlege dir im Vorfeld, ob du eine Klasse testen willst und wie du das machen kannst. Am besten schreibe den Test simultan mit der Klasse/Methode.

Ich beging den Anfängerfehler erst am Ende des Projekts meine Testklassen zu schreiben und musste schnell feststellen, dass mein Code an manchen Stellen unglaublich schwer zu Testen ist. So konnte ich zum Beispiel die .config Datei nicht testen, da Nunit nicht auf diese zugreifen kann. Hätte ich schon während des Codens getestet, hätte ich wahrscheinlich einen ConfigController erstellt, um den Zugriff zu erleichtern.

Generell wurde versucht jeden Layer mit Tests abzudecken. Alle Datenbankfunktionen (AddTour, EditTour, etc.) wurden abgedeckt. Auch im Businesslayer konnte weitestgehend alle wichtigsten Funktionen abgedeckt werden. Wie bereits erwähnt entpuppte sich das Testen im GUI Layer als herausfordernd da weder Config noch Viewmodels darauf ausgelegt waren getestet zu werden.

Factory Pattern

Beim Factory Pattern wird ein Interface für das Erstellen von Objekten erstellt, die Entscheidung welche Objekte instanziiert wird auf die Sub-Classes übertragen. Objekte werden also nur dann erstellt, wenn sie wirklich gebraucht werden. Der Projekterstellungsprozess wird sozusagen abstrahiert. Im Rahmen meines Projekts habe ich die Suchfunktion über dieses Pattern geschrieben.



Factory Method Pattern

Unique Feature

Bei der Erstellung eines Logs, lässt sich das Fortbewegungsmittel bestimmen. Je nach Art (Fahrrad, Auto, etc), wird dann geschätzter Spritverbrauch bzw. Energieverbrauch in Joule berechnet und als Parameter im Log gespeichert.

Zeit & Git

Das Git ist einsehbar unter: https://github.com/D0omsNinthCat/tourplanner

Es wurden für das Projekt in etwa 65 Stunden über 10 Tage gebraucht. Der Fortschritt des Projekts wurde regelmäßig über Git aktualisiert. Details finden sich hier.

Bonus Lesson Learned

Ließ dir die Microsoft Naming Guidelines nicht erst am Ende durch. Ein System für die Benennung von Variablen/Klassen wird umso wichtiger, je größer das Projekt wird.