SWE2-TourPlanner Protokoll

Table of Contents

Einleitung

Ziel der Aufgabenstellung Tourplanner ist es grob gesagt gewesen, eine Desktop-Applikation für das Verwalten von Touren zu erstellen. Benutzer sollen Touren erstellen, bearbeiten und auch wieder löschen können. Des Weiteren sollen sie zu den Touren auch sogenannte Tour Logs abspeichern können, welche unter anderem in Berichten ausgewertet werden.

Das Github-Repository ist hier zu finden: https://github.com/kretmatt/SWE2-Tourplanner

In diesem Protokoll wird in den nächsten Kapiteln näher auf folgende Punkte eingegangen:

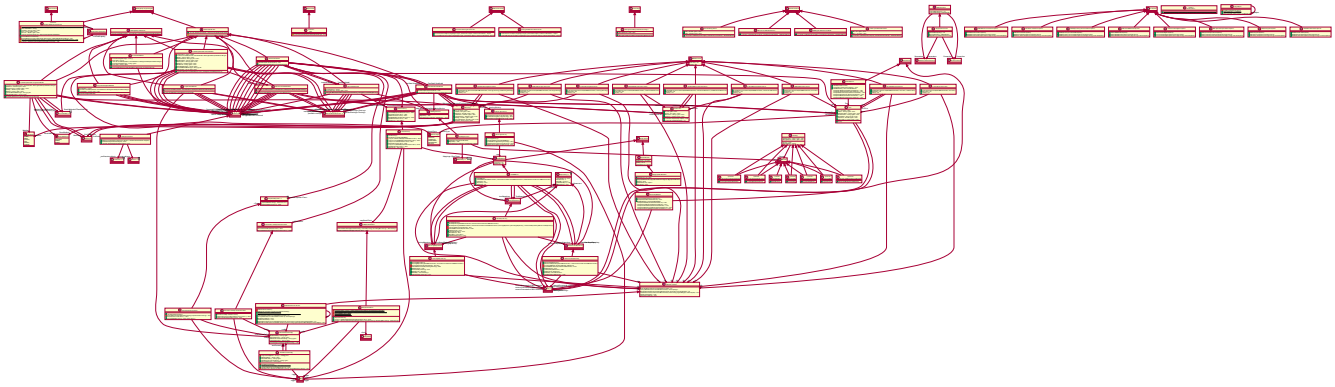
- App Architecture
- Unique Feature / Reusable Component
- Unit Test Design
- Lessons Learned
- Benötigte Zeit

Design / App Architecture

Im Grunde genommen ist die Anwendung in 4 verschiedene Projekte aufgeteilt worden:

1. DataAccessLayer – Class Library für Zugriff zum DataStore. Ist zuständig für den geregelten Zugriff auf den DataStore. Übernimmt Daten vom BusinessLogicLayer und speichert diese in der Datenbank ab. Greift lediglich auf Common zu.
2. BusinessLogicLayer – Class Library für die BusinessLogic. Ist unter anderem für den Anstoß der PDF-Generation der verschiedenen Berichte, für Export / Import von JSON Files und Erstellung von Touren und TourLogs zuständig. Greift auf Common und DataAccessLayer zu.
3. SWE2-TourPlanner – WPF Projekt, welches lediglich auf Common und BusinessLogicLayer zugreift. Ist für die Präsentation der Daten und das Handling des User-Inputs zuständig. Greift auf Common und BusinessLogicLayer zu.
4. Common – Common enthält Klassen, welche von verschiedenen Schichten der Applikation verwendet werden. Darunter fallen unter anderem die LogHelper-Klasse, die TourPlannerConfig-Klasse und Entity-Klassen der Anwendung. Greift auf kein anderes Projekt zu, wird aber so ziemlich von jedem anderen Projekt / Layer verwendet.

Obwohl in den einzelnen Projekten zwar nicht so extrem viele Klassen und Abhängigkeiten untereinander entstehen, ist ein Klassendiagramm mit allen Klassen ziemlich überwältigend / sehr schwer lesbar.



https://github.com/kretmatt/SWE2-Tourplanner/blob/main/Diagrams/CombinedClassDiagram.svg

Für bessere Lesbarkeit / Verständlichkeit werden deswegen in den folgenden Unterkapiteln die einzelnen Layers genauer beschrieben und separate Diagramme (mit Links zu den Diagrammen im Github Repository für leichteres Zoomen) abgebildet.

Data Access Layer

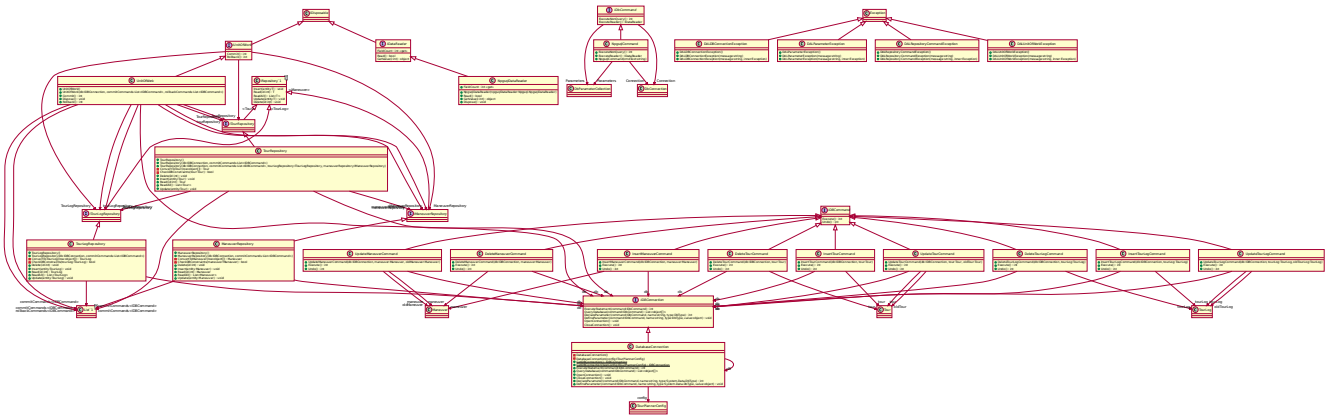
Wie bereits schon zuvor erwähnt, ist der DataAccessLayer für den geregelten Zugriff auf den DataStore zuständig. Der Zugriff auf die Daten wird neben der Datenbankverbindung über 3 primäre Teile geregelt:

1. Commands – Jedes Insert / Update / Delete Statement um Daten in der Datenbank abzuspeichern ist in einem eigenen Command gekapselt. Diese Commands wiederrum stellen zwei essentielle Methoden zur Verfügung – Execute und Undo. Sollte sozusagen ein bereits ausgeführter Command rückgängig gemacht werden, ist dies über die Undo-Methode möglich. Es entsteht dadurch zwar um einiges mehr Code, dadurch hat man aber die Möglichkeit leicht Datenkonsistenz bei fehlgeschlagenen Transaktionen wiederherzustellen.
2. Repositories – Mit den Repositories ist es möglich Daten aus der Datenbank zu lesen und Commands zu erstellen. Zurzeit ist es in meinem Projekt so geregelt, dass man mit Repositories im Grunde genommen lediglich Daten lesen kann. Commands werden zwar erstellt, diese werden aber nicht sofort ausgeführt. Dies ist nämlich Aufgabe des letzten und eigentlich wichtigsten Teils des DataAccessLayer-Projekts: dem UnitOfWork.
3. UnitOfWork – Eine UnitOfWork (UOW) repräsentiert eine Transaktion, welche über von UOW bereitgestellten Repositories mit Commands befüllt werden kann. Erst wenn die Commit Methode aufgerufen wird, werden alle registrierten Commands auf einmal ausgeführt. Falls etwas schief läuft, kann man mittels der Rollback-Methode die Transaktion rückgängig machen und Datenkonsistenz wiederherstellen.

Im DataAccessLayer-Projekt werden sowohl Interfaces, als auch konkrete Implementationen bereitgestellt. Falls man also einen Zugriff auf einen anderen DataStore (z.B.: andere Datenbank)

irgendwann mal hinzufügen möchte, müsste man lediglich sich an die Schnittstellen halten. Natürlich müsste man auch zusätzlich im BusinessLogicLayer Code so anpassen, dass tatsächlich Instanzen anderer Klassen verwendet werden, aber eins ist klar: der DataAccessLayer kann jederzeit erweitert werden.

Die Verbindung zur Datenbank wurde als **Singleton** realisiert, um zu verhindern, dass mehrere Datenbankverbindungen gleichzeitig existieren und somit hier der geregelte Zugriff auf den DataStore verhindert wird. Des Weiteren wurde bei den IDBCommand-Klassen das **Command-Pattern** umgesetzt, um letztendlich das **UnitOfWork-Pattern** (+ **Repository Pattern**) zu befolgen.

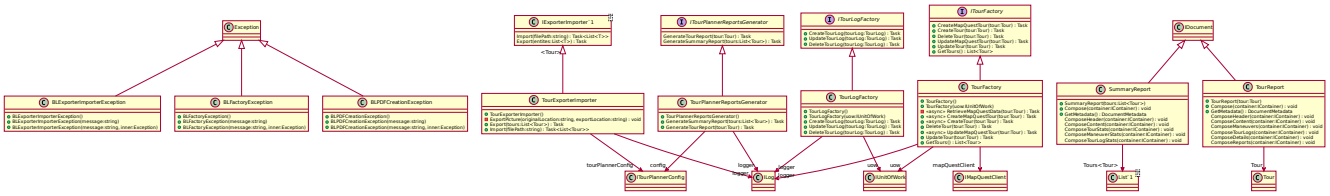


https://github.com/kretmatt/SWE2-Tourplanner/blob/main/Diagrams/DataAccessLayer.svg

Business Logic Layer

Der Business Logic Layer ist zwar verglichen zu den anderen Schichten ein bisschen mager ausgefallen, ist aber dennoch essentiell für die Anwendung. Ich habe mich dazu entschieden, Import / Export und PDF Generierung in diese Schicht zu verlegen, da ein gewisses Ausmaß an BusinessLogic hinter diesen Teilen der Anwendung stecken. Man hätte diese Teile zwar auch in den DataAccessLayer auslagern können, dies wäre aber meiner Meinung nach nicht wirklich passend. Der DataAccesLayer ist nämlich für den geregelten Zugriff auf den DataStore und nicht für PDF-Erstellung, Import und Export (Laden und Serialisierung von JSON Files / Deserialisieren und Export von JSON Files) von Daten zuständig.

Der Business Logic Layer ist hier eigentlich nur eine Art Weiterleitung an den DataAccessLayer, welche aber auch zusätzlich Daten von MapQuest bezieht und auf andere BusinessLogic achtet.

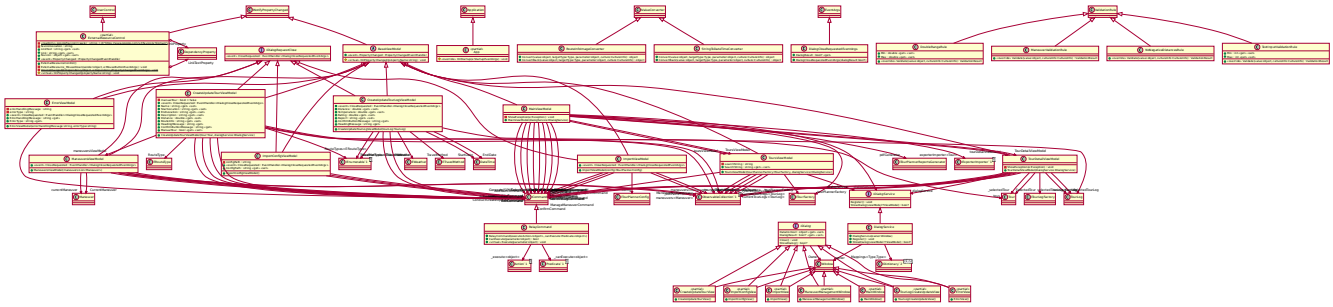


https://github.com/kretmatt/SWE2-Tourplanner/blob/main/Diagrams/BusinessLogicLayer.svg

SWE2-TourPlanner

SWE2-TourPlanner ist eigentlich der UI-Layer der Anwendung. Im Grunde genommen besteht dieser Layer fast nur aus ViewModels, Views / Windows, einigen ValidationRules und zwei Convertern. Die Commands, auf welche im XAML-Code gebindet wird, befinden sich in den jeweiligen ViewModels. Man hätte sie zwar als eigene konkrete Commands implementieren können, diese wären aber sowieso sehr stark mit den ViewModels gekoppelt gewesen. Deswegen habe ich sofort RelayCommand eingesetzt.

Um die Kopplung zwischen ViewModels und Views so weit wie möglich zu senken, habe ich einen DialogService implementiert, welcher das Öffnen / Schließen und Behandeln von Dialogen übernimmt. Hierbei habe ich mich sehr stark an ein Youtube Video gehalten, welches mir geholfen hat die Kopplung bei diesem Aspekt der Anwendung zu senken: https://youtu.be/OqKaV4d4PXg



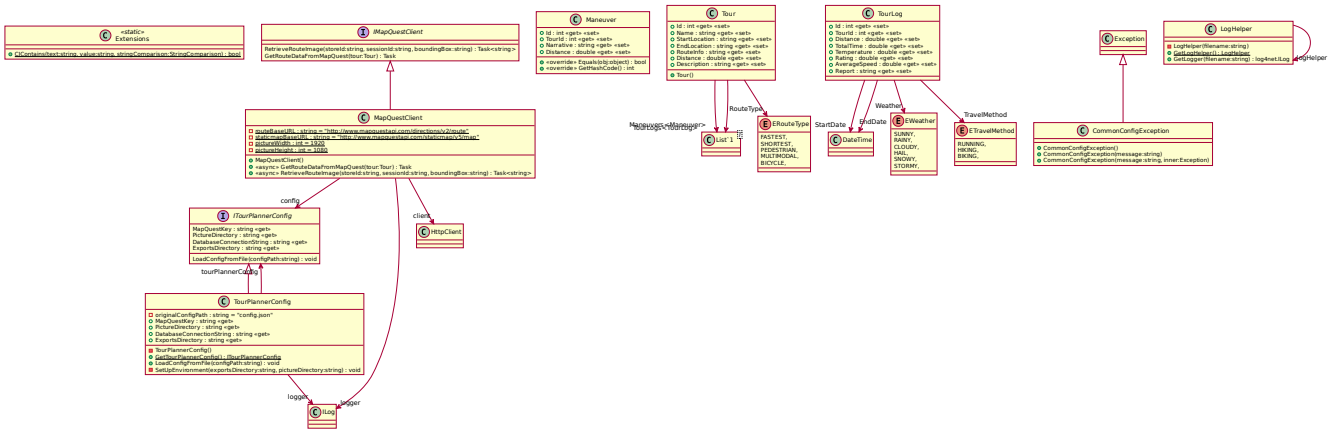
https://github.com/kretmatt/SWE2-Tourplanner/blob/main/Diagrams/UILayer.svg

Common

Im Common Projekt habe ich von unterschiedlichen Layern verwendete Klassen gruppiert, damit nur aufeinanderfolgende Layer kommunizieren und z.B.: der GUI-Layer nicht auf den DataAccessLayer für die Entity-Klassen des Projekts zugreifen muss. Folgende Teiler der Applikation habe sich hier angesammelt:

- Entity-Klassen – Da Tour, TourLog und Maneuver in beinahe allen Layern verwendet werden, habe ich sie in das Common-Projekt ausgelagert.
- Enums – Enums für Wetter, RouteType und TravelMethod werden ebenfalls in mehreren Projekten verwendet.
- LogHelper – LogHelper ist lediglich eine Klasse, welche eine Ilog Instanz für eine andere Klasse konfiguriert. Deswegen befindet sich LogHelper im Common-Projekt, da die Klasse beinahe am öftesten in unterschiedlichen Projekten verwendet wird. Ist übrigens ein **Singleton**, damit nicht mehrere unterschiedliche Konfigurationen für Log4Net während der Ausführung im „Umlauf“ sind.

- (I)TourPlannerConfig – Repräsentiert sozusagen die Konfiguration der Anwendung. Damit nicht unterschiedliche Konfigurationen gleichzeitig verwendet werden, ist TourPlannerConfig ein **Singleton**. Es wird aber eine Methode angeboten, mit der jederzeit eine andere Konfiguration für die gesamte Anwendung geladen werden kann.
- IMapQuestClient – Im Nachhinein betrachtet könnte der MapQuestClient auch im Business Logic Layer platziert werden. Dennoch habe ich ihn sicherheitshalber in das Common-Projekt eingefügt, falls er jemals in anderen Projekten zu Einsatz kommen sollte.
- Extension Method(s) – Ich habe ursprünglich geplant gehabt, alle erstellten Extension Methods im Common Projekt unterzubringen, damit ich sie leicht in den unterschiedlichen Layern der Anwendung einsetzen kann. Letztendlich habe ich aber nur eine Extension Method erstellt die wiederum nur im SWE2-TourPlanner (GUI) Projekt verwendet wird.



https://github.com/kretmatt/SWE2-Tourplanner/blob/main/Diagrams/common.svg

Eingesetzte Design Patterns

Folgende Design Patterns sind in der Anwendung umgesetzt worden:

- Singleton Pattern – DatabaseConnection, TourPlannerConfig,
- Command Pattern – Sämtliche konkreten Implementierungen von IDBCommand
- UnitOfWork Pattern – UnitOfWork
- Repository Pattern – Alle Repositories im Data Access Layer

Eingesetzte Libraries

Folgende Libraries / Nuget Packages sind im Laufe der Implementierung eingesetzt worden:

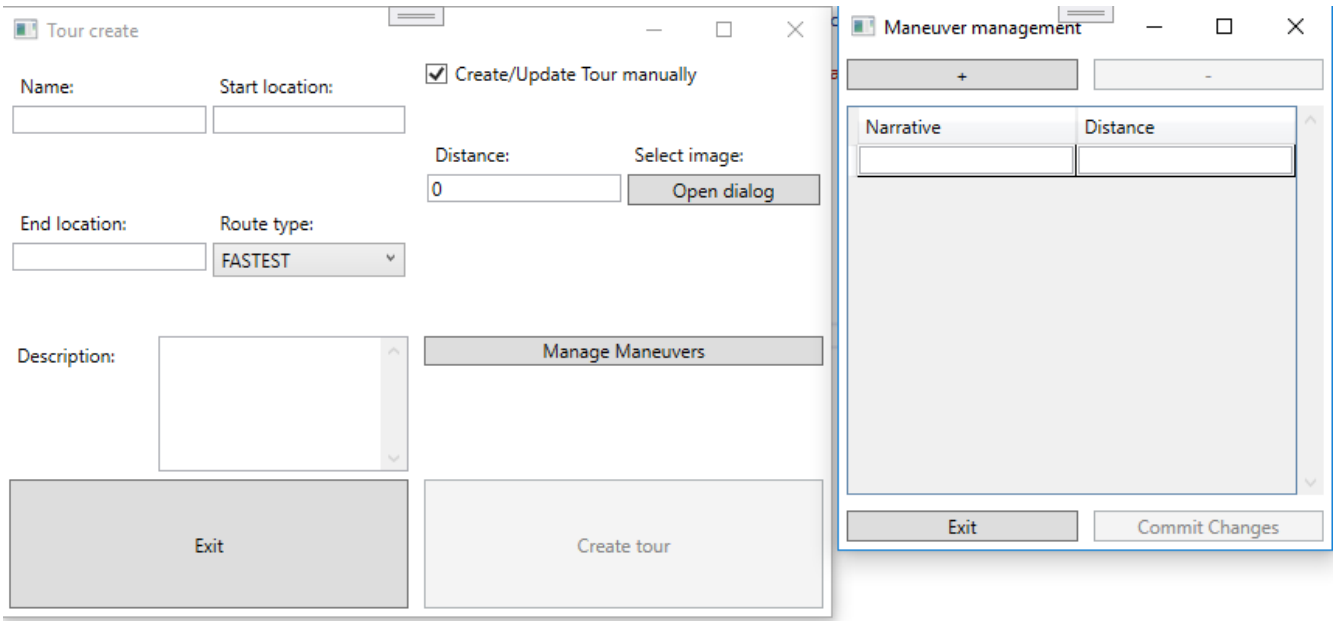
- **Moq** – Für das Testen des DataAccessLayers habe ich mich sehr stark auf das Framework Moq verlassen. Oftmals habe ich hierbei bei den Tests nicht Werte überprüft, sondern vielmehr ob / wie oft Methoden aufgerufen worden sind.
- **QuestPDF** – Für die Erstellung der PDF Berichte (Summary und Tour Report) habe ich QuestPDF verwendet.
- **Newtonsoft** – Für Serialisierung / Deserialisierung von JSON Daten habe ich Newtonsoft verwendet.
- **Nunit** – Für Unit Tests ist das Testing Framework Nunit eingesetzt worden.
- **Npgsql** – Für das Ausführen von PLSQL-Statements habe ich das Npgsql Package verwendet. Hierbei habe ich aber mittels eigenen Interfaces nur die wichtigsten Properties / Methoden meiner Anwendung bereitgestellt.
- **Log4Net** – Für das Logging von Fehler, Warnungen und sonstigen Informationen ist Log4Net eingesetzt worden.
- **Microsoft.Extensions.Configuration(.Abstractions, .Json, .FileExtensions)** – Für das Laden der JSON-Konfigurationsdatei ist Microsoft.Extensions.Configuration verwendet worden.

Unique Feature & Reusable Component

Teil der Aufgabenstellung ist es gewesen, zusätzlich zu den vorgegebenen Features auch eine einzigartige Funktionalität in der Anwendung einzubauen. Des Weiteren musste auch eine wiederverwendbare UI-Komponente erstellt werden, die auch in anderen WPF-Projekten eingesetzt werden kann. In diesem Kapitel werden ebendiese Teile der Anwendung näher beschrieben.

Unique Feature / Bonus Features

Im Grunde genommen sind in meiner Anwendung mehrere Unique Features eingebaut worden. Zum Beispiel werden neben den Tour Daten auch die notwendigen Manöver, um ans Ziel zu kommen, von der MapQuest API bezogen und in der lokalen Datenbank gespeichert. Des Weiteren kann auch jederzeit eine andere Config-Datei während der Ausführung des Programms geladen werden. Hierbei muss lediglich auf das Format und die Korrektheit der angegebenen Daten geachtet werden. Diese Features können aber als Bonus Features bezeichnet werden. Das eigentliche Unique Feature ist nämlich die manuelle Erstellung / Bearbeitung von Touren.



Sobald man eine Tour manuell erstellt, muss man sich selbstständig um die Manöver, das Routen-Bild und die Distanz kümmern. Hierbei darf die zusammengerechnete Distanz aller Manöver nicht von der angegeben insgesamten Distanz abweichen. Ansonsten ist nämlich die Erstellung einer manuellen Tour nicht möglich. Wenn alle benötigten Daten (bei MapQuest Tour lediglich Name, Start, Ende, Route Type und Description; bei manueller Tour zusätzlich auch Manöver, Distanz und Bild-Pfad) angegeben sind, kann eine Tour erstellt werden. Bei einer manuellen Tour werden die Daten sofort in die Datenbank eingefügt / in der Datenbank geupdated, wohingegen bei einer MapQuest Tour Daten zuvor noch von der MapQuest API bezogen werden.

Reusable Component

Der Prozess der Erstellung einer wiederverwendbaren UI Komponente hat erstaunlicherweise länger als gedacht gedauert. An sich ist die Erstellung eine User- oder Custom Component nicht sonderlich kompliziert, aber es hat von meiner Seite Probleme bei der Ideenfindung gegeben. Ursprünglich habe ich mich dafür entschieden, eine User Component für die Temperatur (Slider für Celsius, Fahrenheit und Kelvin die miteinander verbunden sind) zu erstellen, bis ich auf die umständliche Verlinkung einer Website in WPF gestoßen bin. Es gibt nämlich kein Element, welches für die Verlinkung von Websites gedacht ist. Natürlich besteht die Möglichkeit Buttons zu verwenden und Commands die Aufgabe zu übergeben, Links aufzurufen, aber dies hat meiner Meinung nach ziemlich umständlich gewirkt. Deswegen habe ich letztendlich die sogenannte ExternalResourceControl erstellt. ExternalResourceControl ist eine User Control und soll die einfache Verlinkung von Websites ermöglichen. Diese User Control hat zwei grundlegende Properties, welche mittels Binding gesetzt werden können:

- Link – Der Link zur Website, welche aufgerufen werden soll.
- LinkText - Der eigentliche Text, welcher in der User Control angezeigt werden soll.

Beide Properties sind Dependency Properties. Fall sich aber der Link ändert, gibt es auch Auswirkungen auf eine dritte Property, welche im User Control dargestellt wird: dem Favicon. Jedes mal wenn sich der Link ändert, wird auch ein neuer String / Link für das Favicon gebildet. An sich wäre es ziemlich schwer, Links zu den Favicons jeder möglichen Website zu generieren. Jedoch gibt es von Google eine „Favicon-API“, mit der leicht Favicons in die Anwendung integriert werden können.

Folgendermaßen können Favicons von beliebigen Websites über die API eingebunden werden:

https:/www.google.com/s2/favicons?domain	**=**	***{Link}***
Base-URL der Favicon-API		Link der Website, von der ein Favicon gesucht wird

Beispielsweiße kann man auch vom Moodle der FH Technikum Wien das Favicon beziehen:

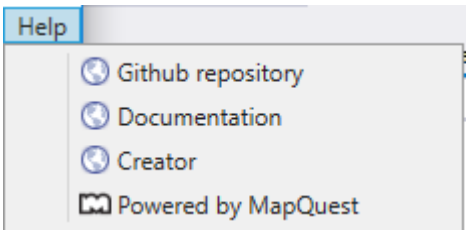
https://www.google.com/s2/favicons?domain=https://moodle.technikum-wien.at

An sich funktioniert die API mit sehr vielen verschiedenen Websites, aber aus unerklärlichen Gründen kann für Github kein Favicon gefunden werden. Falls hier kein Favicon gefunden werden kann, wird einfach ein Default Icon zurückgeliefert:

https://www.google.com/s2/favicons?domain=https://github.com/

Besonders ist hier zu erwähnen, dass außer den standardmäßig verfügbaren Klassen kein selbst programmierter Code in die Control eingebunden worden ist. User Controls sollen nämlich möglichst eigenständig und unabhängig sein, damit sie auch in verschiedenen Anwendungen eingesetzt werden können. Deswegen ist auch ein normales Click-Event für die User Control verwendet worden. Normalerweise entsteht hiermit eine ziemlich hohe Kopplung, diese ist jedoch für die hohe Eigenständigkeit der User Control eingegangen worden.

In einer Anwendung kann die wiederverwendbare UI Komponente folgendermaßen ausschauen:



Unit Test Design

Grundsätzlich liegt der Hauptfokus der Unit Tests bei mir beim Data Access Layer. Da ich mit UnitOfWork etwas neues ausprobiert habe, mit dem ich ursprünglich auch einige Denkprobleme / Logikfehler gehabt habe, wollte ich hierbei sicher gehen, dass der Ablauf der Klassen im Data Access Layer richtig abläuft. Deshalb habe ich auch mehr mit Mocks als mit Asserts gearbeitet. Anstatt Werte zu vergleichen habe ich hierbei eher überprüft, ob die richtigen Methoden aufgerufen worden sind. Auch die Anzahl der Aufrufe habe ich getestet.

Beim Business Logic Layer habe ich nicht besonders viel getestet. Export / Import und PDF Erstellung wollte ich nicht testen, da ich im Grunde genommen testen würden, ob die Packages / Frameworks funktionieren, und nicht ob mein eigener Code funktioniert. Deswegen habe ich beim Business Logic Layer getestet, ob die Factories richtig funktionieren.

Die Tests des Common-Projekts sind letztendlich darauf hinausgelaufen, ob die Singletons korrekt funktionieren, ob richtige Ilog Instanzen zurückgeliefert werden und ob das Laden von Konfigurationsdateien einigermaßen korrekt abläuft.

Im UI Abschnitt der Anwendung habe ich die ValidationRules und die Converter überprüft. Hier wäre zugegebenermaßen einiges mehr möglich gewesen. Unter anderem hätte man die einzelnen Commands der ViewModels und den DialogService zusätzlich testen können.

Lessons Learned

Während der Implementierung / Erfüllung der Aufgabenstellung habe ich folgende Dinge gelernt:

- Kommentare von Anfang an schreiben dauert nicht so lange – Ursprünglich habe ich beim DataAccessLayer nebenbei alle Klassen mit Documentation Comments versehen. Je weiter ich beim Projekt gekommen bin, desto weniger habe ich kommentiert, wodurch ich gegen Ende hin einiges nachholen durfte. Somit habe ich auf schmerzhafte Art und Weise gelernt, dass man bereits während der Implementierung Kommentare schreiben sollte.
- Exception Handling mit Tasks – Besonders viel Zeit habe ich beim Exception Handling bei Tasks benötigt. An sich hat der Code gepasst, eine Einstellung bei Visual Studio hat aber dafür gesorgt, dass meine eigens definierten Exceptions zu einer Unterbrechung während der Ausführung der Anwendung geführt haben.
- Anforderungen sollten noch besser definiert / besser geplant werden – Verglichen zum Projekt im letzten Semester habe ich dieses Semester eine viel bessere Vorangehensweise gehabt. Ich habe mich vom DataAccessLayer nach oben bis zur Oberfläche hochgearbeitet. Dadurch habe ich mir zwar schon einiges an Zeit sparen können, ich habe aber immer wieder mal zu den unteren Schichten zurückkehren müssen, weil ich ein paar Kleinigkeiten nicht beachtet habe.

Deswegen werde ich bei kommenden Projekten ein bisschen mehr Zeit bei der Planung einplanen.

Benötigte Zeit

Während des Projekts habe ich zwar die Zeit gemessen, aber nicht für jeden einzelnen Schritt separat. Für folgende Abschnitte kann ich aber die Zeit relativ genau angeben:

- Data Access Layer: 25h
  - Ein Großteil der Zeit ist für das Konzept / die Idee benötigt worden. Die eigentliche Umsetzung hat zwar nicht solange gedauert, aber die Behebung einiger kleiner Denkfehler hat umso mehr Zeit in Anspruch genommen.
- Business Logic Layer: 10h
  - Der Business Logic Layer hat erstaunlicherweise nicht viel Zeit in Anspruch genommen. PDF Erstellung hat am meisten Zeit benötigt, dann der Export / Import von Daten und die Factories sind am schnellsten erledigt gewesen.
- UI Layer: 25h
  - Es ist besonders viel Zeit bei den ViewModels benötigt worden, da diese auch ein Großteil dieses Layers sind. Die anderen Klassen / Oberfläche sind schnell erledigt gewesen, das Beheben von Binding Errors und die Reusable Component haben den Rest der Zeit in Anspruch genommen.
- Common Projekt: 5h
  - Da die Inhalte des Common-Projekts nicht besonders komplex gewesen sind, ist hier auch dementsprechend wenig Zeit benötigt worden.
- Unit Tests: 7h
  - Die Unit Tests sind auch ziemlich schnell erledigt gewesen. Ein Großteil der Zeit ist hierbei für das Testen des Data Access Layers benötigt worden. Die anderen Tests sind vergleichsweise ziemlich schnell erledigt gewesen.
- Dokumentation + Source Code Documentation: 9h
  - Die Dokumentation ist ziemlich schnell erledigt gewesen, aber das Kommentieren des Source Codes hat aufgrund bereits genannter Gründe umso länger gedauert.

Insgesamt habe ich somit für das gesamte Projekt 81 Stunden benötigt, also der ungefähre Arbeitsaufwand der Lehrveranstaltung für zuhause.