DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Entwicklung einer Software zur Schaltplanerstellung in der Elektrotechnik

Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Mikka Jenne und Simon Leitl

Abgabedatum 01. Juni 2020

Bearbeitungszeitraum	12 Wochen
Matrikelnummer	2062885, 7068806
Kurs	tinf17b4
Ausbildungsfirma	cjt Systemsoftware AG Karlsruhe
Gutachter der Studienakademie	Prof. Dr. Kai Becher

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeitmit dem Thema: „Entwicklung einer Software zur Schaltplanerstellung in der Elektrotechnik“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

_____ _____

Ort Datum Unterschrift

Zusammenfassung

Computer sind heutzutage allgegenwärtig.
Kaum ein Unternehmen arbeitet ohne computergesteuerte Unterstützung, sei es das Schreiben einer Rechnung in einem mittelständischen Unternehmen, das Verwalten von einzelnen Arbeitsprozessen oder die Unterstützung bei einzelnen Arbeitsschritten. Speziell im Elektrotechnischen Bereich gibt es heute noch Prozesse die hinsichtlich der physischen und digitalen Welt stark voneinander getrennt sind, z.B. das Zeichnen von Schaltplänen und die digitale Erfassung von Messwerten, die über Programme verwaltet werden können.
Das Ziel dieser Arbeit ist es, die physische Welt durch die Möglichkeit der digitalen Zeichnung von Gebäudeschaltplänen zu erweitern. Für den Gebrauch designed, wird auf eine Verbesserung sowie auf die Transparenz von solchen Schaltplänen gezielt, um so weitere essentielle Informationen zu Gebäuden zu erhalten. Da die händische Zeichnung meist aufwändig, kostenintensiv ist und keiner Norm entspricht, soll durch die Digitalisierung dieses Prozesses Abhilfe geschaffen werden. Durch Nutzung neuester Technologien im Bereich Desktopanwendungen und des Einsatzes einer einheitlichen Zeichenstruktur sollte diese Applikation für den Otto Normalverbraucher leicht nutzbar sein.

Inhaltsverzeichnis

Abbildungsverzeichnis

Liste der Code-Beispiele

Abkürzungsverzeichnis

WPF Windows Presentation Foundation 10
XAML Extensible Application Markup Language 11
UX User-Excperience, dt. Nutzererfahrung und Nutzererlebnis 20

Kapitel 1

Einleitung

In diesem Teil der Arbeit wird auf die Motivation des Themas eingegangen. Die Aufgabenstellung genau erläutert und sowohl die Ziele als auch der Aufbau der Arbeit dargelegt.

1.1 Motivation

Die herkömmlichen Schaltpläne von Gebäuden, Schaltschränken oder Maschinen werden aus der Historie heraus per Hand auf normales Papier gezeichnet. Der Elektrotechniker oder gar ein Architekt nimmt Zeit in Anspruch einen solchen Plan detailgetreu und nach Maßstab zu zeichnen. Diese Arbeit ist sehr zeitintensiv und preislich sehr teuer. Zudem können handschriftliche Änderungen das Dokument unübersichtlich machen, bzw. müsste bei jeder Änderung ein neuer Plan ausgefertigt werden.
Aus diesen Gründen wird eine Zeichnung meistens vernachlässigt, wobei es für Zukünftige Arbeiten, z.B. Sanierungen, Ausbauten o.ä. essentiell ist. Um diesem fatalen Fehler, den Plan zu vermeiden, entgegenzuwirken wurden Ideen und Vorschläge gesammelt, wie dieser Prozess deutlich einfacher und ressourcenschonender vonstattengehen könnte.

Diese Problemstellung der realen Welt führte dazu diese Aufgabe in Angriff zu nehmen und diesen Prozess zu modernisieren. Mit dem Gedanken und der Intension der Digitalisierung in der Elektrotechnik wurden Überlegungen getätigt diese Modernisierung umzusetzen.

Ein motivierender Aspekt dieses Vorgehens ist, dass die auf Papier gezeichneten Dokumente verloren gehen oder physische Schäden erleiden können und so unbrauchbar werden. Durch die Digitalisierung des Verfahrens sind diese Aspekte ausgeschlossen und sind deutlich vielseitiger.

Im Zeitalter der Industrie 4.0, bei der der Schwerpunkt auf Digitalisierung liegt, war es leicht eine Idee zu generieren die standardmäßige Schaltpläne in dem Zeichnungs- und Ausarbeitungsprozess verbessert und ein bekanntest Problem löst. Sowohl in zeitlicher und kostspieliger Hinsicht als auch die Übersichtlichkeit solcher Dokumente kann enorm gesteigert werden. Veränderungen keine neue Zeichnung anzufertigen, sondern können in der bestehenden Datei schnell und einfach ausgebessert werden.

1.2 Aufgabenstellung

Es soll ein Konzept für eine Software erstellt werden, welches erlaubt Schaltpläne digital zu zeichnen, anzuzeigen, flexibel zu ändern und alle wichtigen Informationen zu Verfügung zu stellen.
Dieses ausgearbeitete Konzept soll einen modularen Ansatz verfolgen, um in Zukunft beliebig erweiterbar zu sein und die Integration von neuen Features zu gewährleisten. Nach Erstellung des Konzepts und der definierten Software-Architektur soll der Grundbaustein der Applikation gelegt und gefestigt werden, indem die Grundfunktion implementiert und getestet werden.
Die Grundfunktionen werden in folgendem kurz aufgelistet, um einen groben Überblick zu verschaffen.

- Startmenü - Übersicht
  - Erstellen eines Schaltplans
  - Öffnen einer vorhandenen Schaltplan-Datei
  - Weiterarbeiten an einer Schaltplan-Datei
- Editor
  - Grundgerüst Zeichen (Grundriss, Türen, etc.)
  - Leitungen Zeichen (Größe, Stärke der Leitung)
  - Mögliche Komponenten einfügen (Steckdose, Lichtschalter, etc.)
  - Anzeige von Informationen zu Komponenten und Leitungen
  - Legende zum Verständnis und zur Übersicht der einzelnen Zeichenkomponenten

Die obenstehenden Punkte sehen die Grundfunktionen vor und werden in drei große Teilbereiche, wie zu erkennen, abgegrenzt. Das Design und der allgemeine Aufbau wird in folgenden Kapiteln genauer erläutert und anhand von Bildern, Mockups und Diagrammen dargestellt.

Der Hauptaugenmerk dieser Aufgabe, bzw. der Problemstellung liegt darin die Möglichkeit zu schaffen einen Schaltplan rentabel und einfach zu digitalisieren, um den Einsatz von Schaltplanzeichnungen erneut zu verbreiten beziehungsweise zu modernisieren.

1.3 Aufbau der Arbeit

Nach den oben genannten einleitenden Informationen widmet sich das Kapitel 2 den essentiellen und wichtigsten Grundlagen dieser Arbeit. Zu Anfang werden allgemein gültige Grundlagen zur Digitalisierung in der Gebäudetechnik (2.1) offenbart, um Kontexte zur Arbeit im Allgemeinen zu verstehen, gefolgt von einer Einführung in die modulare Software Architektur (2.2) zum Ende des Kapitels.

Anschließend auf Kapitel 2 wird in Kapitel 3 auf die verwendeten Technologien und Tools eingegangen. Mit Windows Presentation Foundation (3.1) wird erläutert was es damit auf sich hat und welche Programmiersprache diese Technologie sich zu eigen macht. In (3.2) wird das MVVM-Pattern genauestens erklärt, was es bedeutet, wie und warum es angewendet wird. Das Windows eigene System.Drawing, welches zum Zeichnen der Schaltpläne verwendet wird in (3.3) aufgeführt.

Nach den verwendeten Technologien geht es in Kapitel 4 um den eigentlichen Systementwurf, zum einen um das Architekturkonzept (4.1) und zum anderen um das Softwarekonzept (4.2). Dabei wird auch detailliert auf das Aussehen eingegangen und die graphischen Benutzeroberflächen, GUIs, anhand mehreren Mockups dargelegt.

In Kapitel 5 wird genauestens auf die praktische Umsetzung und Implementierung der vier definierten Use Cases eingegangen, die im einzelnen kurz chronologisch aufgeführt werden.

- Startmenü (5.1)
- Toolbox (5.2)
- Zeichenfläche (5.3)

Die letzten zwei Kapitel, Ereignis (6) und Ausblick (7), runden die Dokumentation ab und schließen die Arbeit. Das Ergebnis wird hierbei analysiert und Verbesserungsvorschläge angemerkt. Der Ausblick gibt Aufschluss darüber welche möglichen Erweiterungsmöglichkeiten es gibt und wie die Zukunft dieser Arbeit möglicherweise aussehen könnte.

Kapitel 2

Grundlagen

Kapitel 3

Technologien

In diesem Kapitel werden alle für das Verständnis notwendigen und verwendeten Technologien aufgeführt und genauestens erläutert.
Die durch die Programmiersprache *C#*, von Microsoft, zur Verfügung gestellte Technologie *WPF* wird in folgendem Auskunft über allgemeine Informationen, Funktionsweise, Einsatzgebiete und deren untergeordneten Technologien konkretisiert.
Mit den vorab getätigten Überlegungen zu Zeichentechnologien die kompatibel mit *C#*, bzw. *WPF* waren kam man zu der von Microsoft entwickelten Zeichentool System.Drawing, die ebenso präzisiert wird.

3.1 WPF (Mikka Jenne)

Windows Presentation Foundation (WPF) in Visual Studio bietet Entwicklern ein einheitliches Programmiermodell zum Erstellen von Line-of-Business-Desktopanwendungen unter Windows. [Microsoft 2020]
Das Grafik-Framework WPF, sowie die Entwicklungsumgebung Visual Studio sind beides Produktionen des Unternehmens Microsoft. Von Microsoft wird es als das Framework zur Erstellung von Desktop-Clientanwendungen für Windows mit visuell herausragenden Benutzerflächen umworben. Der Kern von *WPF* ist eine auflösungsunabhängige und vektorbasierte Rendering-Engine, die die Leistungsfähigkeit moderner Grafikhardware nutzt. [WPF Kern] Dieser Kern wird durch *WPF* um mehrere Features zur Entwicklung von Anwendungen erweitert. Zu den Erweiterungen des *WPF*-Kerns zählen unteranderem die Beschreibungssprache Extensible Application Markup Language *XAML*, die in der nächsten Sektion konkretisiert wird, diverse Steuerelemente, Datenbindungen, 2D- und 3D-Grafik, Animationen, Vorlagen, Dokumente, Texte und Typographie.
Die Anlehnung von *WPF* an *.NET*-Typen ist sehr stark, da *WPF* eine Teilmenge von *.NET* ist. Die Typen von *.NET* sind zum größten Teil im Namespace System.Windows, welcher im Nachgang noch genauer aufgezeigt wird, wiederzufinden.
Das Grafik-Framework kann in zweierlei Programmiersprachen implementiert werden, zum einen unter der Nutzung von *C#* und zum anderen kann Visual Basic verwendet werden.
Microsoft entwickelte viele Technologien und Frameworks, die die themenübergreifende Nutzung möglich machen, bzw. die Einfachheit besitzen durch die vorhandenen Grundlagen viele dieser Technologien nutzen zu können ohne neues Wissen aneignen zu müssen, da die Grundlagen identisch sind. Zu diesen Technologien gehören *ASP.NET* und Windows Forms als Vorgänger von *WPF*.

3.1.1 XAML - Extensible Application Markup Language

Allgemein ist Extensible Application Markup Language (XAML) eine Beschreibungssprache zur Gestaltung grafischer Benutzeroberflächen, entwickelt von Microsoft. Die neue deklarative Sprache wurde für das Framework *.Net 3.0* in Windows Presentation Foundation *WPF* für *WPF*-Windows-Anwendungen entwickelt. [Wikipedia 2020]
Die Extensible Application Markup Language ist eine XML-basierende Sprache, die verwendet wird, um Benutzeroberflächen, Verhaltensweisen, Animationen, grafische Elemente etc. zu definieren.

Window - Fenster

Ein Code-Template eines standardmäßigen Fensters *Window* sieht wie folgt aus und wird so als Basis-XAML zur Verfügung gestellt.

```
1 <Window x:Class="WpfApplication1.Window1"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4     Title="Window1" Height="300" Width="300">
5     <Grid>

7     </Grid>
8 </Window>
```

Code-Beispiel 3.1: Window-Template

Der oben aufgeführte Ausschnitt zeigt eine *XAML*-Datei, die ein vorgefertigtes Template von Microsoft darstellt. Bei Erstellung einer neuen Datei, bzw. einer Benutzeroberfläche, wird dieses Template zur Verfügung gestellt. Es beinhaltet zu Anfang alle relevanten Informationen über Größe des Fensters und dessen Design. Ein Grid, welches ebenso in dem Template vorhanden ist, erfüllt die Funktion eines Koordinators. Das bedeutet, dass auf der Oberfläche ein Raster angelegt wird, welches die Positionen verschiedener Komponenten und Elementen koordiniert. So kann allen Elementen ein fester Platz, an dem es angeordnet sein soll, zugewiesen werden. Ebenso umfasst ein *Window* einen hypothetischen Container, bzw. ein Grundgerüst, welches als Repräsentation verschiedener Seiten *Page* verwendet wird. Die Basisklasse stellt einen Standardrahmen, sowie eine Titelzeile, einem Maximierungs-, Minimierungs- und Schließknopf zur Verfügung.

Page - Seite

Neben dem Template für ein Fenster gibt es anderweitige Templates, um das Erscheinungsbild von Oberflächen zu gestalten. Zum Beispiel gibt es, wie im Absatz darüber erwähnt, die sogenannten Pages, die im Template sichtlich das Markup *Window* durch *Page* ersetzen. Der Funktionsunterschied is jedoch enorm, wogegen das Window ein Grundgerüst mitliefert, zeigt die Seite nur die Oberfläche ohne Rahmen, die verwendet werden kann.

```
<Page x:Class="WpfApplication1.Page1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Page1" Height="300" Width="300">
    <Grid>

    </Grid>
</Page>
```

Code-Beispiel 3.2: Page-Template

UserControl - Benutzersteuerelement

Eine dritte From der Repräsentation ist das sogenannte *UserControl* (dt. Benutzerkontrolle), welches im Prinzip eine Mischung aus einem Window und einer Seite ist. Der Unterschied zwischen den einzelnen Elementen ist lediglich das Markup, bzw. die Namensgebung *UserControl*.

```
<UserControl x:Class="WpfApplication1.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UserControl1" Height="300" Width="300">
    <Grid>

    </Grid>
</UserControl>
```

Code-Beispiel 3.3: UserControl-Template

Grid

Ein Grid ist ein Rasterelement in *WPF* und teilt eine Seite oder ein Fenster in eine vorab definierte Anzahl von Zeilen und Spalten. Das Raster ist ein sehr leistungsfähiges und nützliches Layout in WPF. Damit können untergeordnete Elemente in Zellen angeordnet werden, die durch Zeilen und Spalten definiert sind. Wenn ein neues XAML-Dokument erstellt wird, fügt Visual Studio automatisch ein Raster als ersten Container innerhalb des Fensterelements hinzu. Dieses vorgegebene Template kann durch *Grid.ColumnDefinitions* in Spalten und durch *Grid.RowDefinitions* in Zeilen unterteilt werden, wie dem folgenden Code-Ausschnitt zu entnehmen ist. [WPF-Tutorial 2020]

```
<Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
</Grid.RowDefinitions>
```

Code-Beispiel 3.4: Grid-Template

3.1.2 System.Windows

System.Windows ist ein in WPF deklarierter Namespace. Eine Bibliothek die viele essentiellen Methoden, verschiedene Klassen und WPF-Basiselementklassen zur Verfügung stellt. [Window 2020] die das WPF-Eigenschaftensystem und die zugehörige Ereignislogik unterstützen, sowie andere Typen bereit, die häufig vom WPF-Kern und -Framework benötigt werden. Die API, engl. Application Programming Interface, stellt zum Beispiel eine Drag & Drop - Funktion bereit, die Hilfsmethoden und Felder für die Einleitung von Drag & Drop-Vorgängen bietet. Zusätzlich beinhaltet diese Schnittstelle eine Methode zum Starten eines solchen Vorgangs und verwaltet und überwacht das Hinzufügen und Entfernen von Drag & Drop-bezogenen Ereignishandlern.
Darüberhinaus gibt es viele weitere Methoden die über diesen Namespace zur Verfügung gestellt werden.
Ein weiterer Namespace der in der Entwicklung verwendet wurde ist der sogenannte *System.Collections*-Namespace, der in folgendem kurz beschrieben und mit einem Beispiel dargelegt wird.

3.1.3 System.Collections

Der *System.Collections*-Namespace enthält Schnittstellen und Klassen, die verschiedene Auflistungen von Objekten definieren, z.B. *Listen*, *Warteschlangen*, *Bitarrays*, *Hashtabellen* und Wörterbücher. [Collection 2020] Ein bekannter Parameter dieser Schnittstelle ist die *ArrayList*, eine Anreihung von Feldern, die nach Anforderungen, bzw. nach Gebrauch beliebig erweitert werden kann und Datenstrukturen der gleichen Form speichert.
Zur Implementierung der Hauptanwendung, das Zeichnen der Schaltpläne, werden die Schnittstellen des *System.Drawings*-Namespaces verwendet.

3.1.4 System.Drawing

Der *System.Drawing*-Namespace ist für die Darstellung von bestimmten Textformaten, Anzeigen von Graphiken durch *Bitmaps* oder *Icons* und ermöglicht den Zugriff auf die grundlegensten Grafikfunktionen. [Drawing 2020] Durch weitere Namepsaces können diese Funktionen erweitert, bzw. verfeinert werden. Es werden über diesen Namespace auch Methoden zur händischen Zeichnung von Graphiken angeboten.

3.2 MVVM (Simon Leitl)

Model-View-ViewModel ist ein Architekturmuster welches Entwicklern als Vorlage dient und hilft grafische Oberflächen standardisiert und strukturiert zu entwickeln. MVVM wird als Weiterentwicklung bekannter Architekturmuster wie MVC betrachtet. MVVM ist 2005 im Zuge der Entwicklung von WPF entstanden und ist mittlerweile bestandteil von Universal Apps und Webframeworks. MVVM trennt die Benutzeroberflächen von der Logik. Dabei wird die Logik in sogenannten ViewModel Klassen dargestellt. Die Benutzeroberfläche bindet sich an die Properties der ViewModel Klassen und erhält fast keinen prozeduralen Code. Setzt man das MVVM Pattern richtig um, enthalten die ViewModel Klassen keine Benutzeroberflächen Elemente. Dies ist ein großer Vorteil für das durchführen von Unit Tests. Die ViewModel Klasse selbst stellt öffentliche Properties und Commands zur Verfügung, an welche sich die View binden kann. Die Ebenen sind in 3.1 zu sehen.

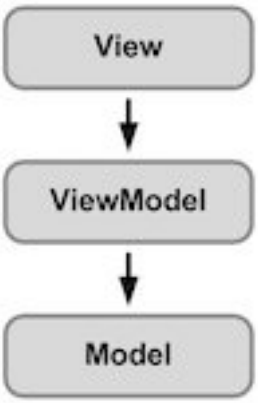

Abbildung 3.1: MVVM Pattern

Die View Ebene wird zum Darstellen genutzt, die ViewModel Ebene enthält die Logik durch Commands und zuletzt das Model, welches die Daten enthält.
Die Kommunikation zwischen der Benutzeroberfläche und der Logik wird durch sogenannte Bindings durchgeführt. Dabei werden Bestandteile der Benutzeroberfläche an das ViewModel oder bestimmte Commands des ViewModel, also der Logik gebindet. Über das Binding verläuft dann die Kommunikation zwischen den Komponenten. Dabei muss das Binding in der XAML Datei der View, einem Objekt zugewiesen werden, wie in Code-Beispiel 3.5 zu sehen ist. Die Zuweisung ist der Name einer Funktion, die im ViewModel vorhanden ist.

```
1     <TextBlock Margin="3" Text="{Binding␣ViewModelNameList}"/>
```

Code-Beispiel 3.5: Binding in XAML

Die Funktion wird dabei dem TextBlock aus dem Code-Beispiel 3.5 als Quelle zugewiesen. Über diese Funktion werden nun die Daten, die innerhalb des Textblocks auf der Oberfläche angezeigt werden, bezogen. Um die Daten im ViewModel entsprechend zur Verfügung zu stellen, wird eine Funktion benötigt. Diese Funktion besitzt einen Getter Methode, um die Daten über einen return auszugeben. Im Code-Beispiel 3.6 wird diese Funktion dargestellt. Sie enthält eine Liste in der die Namen aller VieModels gespeichert ist.

```
1 public ObservableCollection<String> ViewModelNameList
2 {
3     get { return _viewModelNameList; }
4 }
```

Code-Beispiel 3.6: Binding im ViewModel

Die Funktion gibt die Namen eines ViewModel als String zurück. Dieser String wird dann innerhalb der TextBox in 3.5 abgebildet.
Alle ViewModel die Funktionen für Bindings enthalten erben von der ViewModelBase Klasse. Dadurch wird die, in der ViewModelBase enthaltene *INotifyPropertyChanged* Funktion implementiert. Diese ist Notwendig um Anzeigen auf der Oberfläche während der Laufzeit der Anwendung zu ändern und zu aktualisieren. Durch die *INotifyPropertyChanged* erkennt die

Anwendung zur Laufzeit Änderungen an der Oberfläche oder der Logik. Findet eine Änderung statt so ist die Funktion dafür verantwortlich, alle Teilnehmer des Bindings zu informieren und die Änderungen zu aktualisieren. [ENTWICKLER 2010]

3.2.1 MVVM Light Framework

MVVM Light ist ein Framework, das dazu entwickelt wurde MVVM Architekturen in Windows Universal, WPF, Silverlight, Xamarin.iOS, Xamarin.Android und Xamarin.Forms zu entwickeln. Dabei hilft das Framework bei der Erstellung der Strukturen innerhalb eines Projekts. Somit haben die Entwickler eine Basis, auf die sie sich bei der Umsetzung beziehen können. Dadurch erhält die Anwendung eine einfache und saubere Wartbarkeit sowie eine einfache Erweiterbarkeit. MVVM Light hilft dabei, die View vom Model zu trennen. Ein weiterer Vorteil des Frameworks sind die verbesserten Testmöglichkeiketen für das entwickelte Projekt. Es erstellt testbare Anwendungen welche über eine dünnere Benutzeroberfläche verfügen. [MVVMLIGHT 2020]

Kapitel 4

Systementwurf

Dieses Kapitel widmet sich dem Systementwurf mit Bezug auf die in Kapitel 1 dargelegte Aufgabenstellung. Dabei wird die Softwarekonzeption dargestellt und Gründe für die Wahl der Frameworks genannt.

4.1 Architekturkonzept (Simon Leitl)

Um Schaltpläne in der Gebäudetechnik, durch die Software zu realisieren, muss ein geeignetes Konzept, sowie ein erster Entwurf erstellt werden. Dieser wird als Basis für die künftigen Weiterentwicklungen verwendet. Die Anwendung muss in der Lage sein, auf Benutzerbedienungen zu reagieren, Skizzen zeichnen zu können und die Daten passend zu verändern. Wird ein Schaltplan gezeichnet, soll dieser gespeichert und bei der nächsten Ausführung wieder geöffnet werden können. Das ermöglicht die mehrmalige Anwendung des Systems für den Nutzer.
Die entstehende Software soll für eine fortführende Entwicklung geeignet sein. Dafür wird ein modularer Aufbau vorgesehen. Dieser soll das einfache und schnelle Hinzufügen von Bestandteilen ermöglichen. Des weiteren besteht die Software aus verschiedenen Bestandteilen, die oft einzeln behandelt werden. Durch einen modularen Aufbau wird gewährleistet, dass die Module unabhängig voneinander bearbeitet und verändert werden können. Ein weiterer Faktor der Systemarchitektur ist das Verhältnis zwischen FrontEnd und BackEnd. Also dem View und der Logik. Für eine parallele Entwicklung bietet sich die Entkopplung zwischen der Benutzeroberfläche und der Logik an. Dies hat außerdem den Vorteil, dass bei späteren Änderungen der Benutzeroberfläche die Logik nicht angepasst werden muss. Für die entstehende Software wird sowohl eine Benutzeroberfläche, sowie eine Datenhaltung und systemweite Verarbeitungsprozesse benötigt. Da als Basissystem *C#* und das WPF Framework benutzt werden, wird für die Architektur das Model-View-ViewModel Pattern ausgewählt. Dieses begünstigt außerdem das Testen der Entwicklung. Durch MVVM können UI und BackEnd unabhängig voneinander getestet werden. Die Software wurde zunächst in zwei große Bausteine geteilt.

- Startmenu
- Editor

Neben den zwei Bausteinen, verfügt die Anwendung noch über zwei Ordnerstrukturen. Eine für das Speichern von Projekten, die andere für die Komponenten, die in das System geladen werden. Die Anwendung im gesamten Überblick ist in Abbildung 4.1 sichtbar.

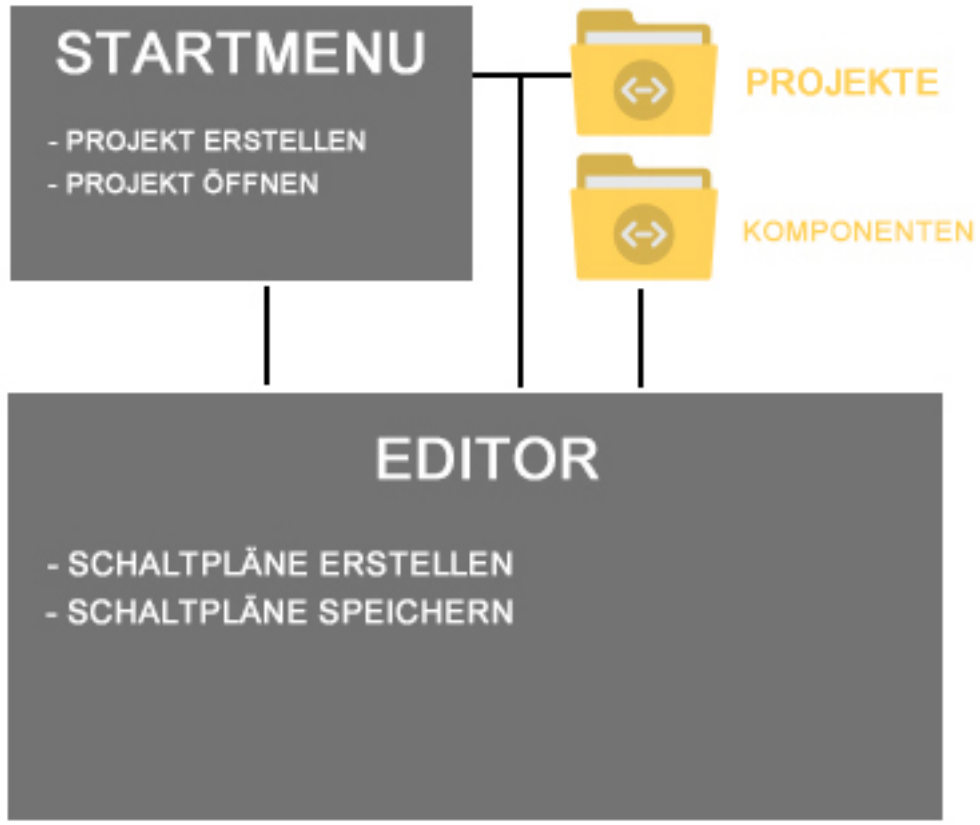

Abbildung 4.1: Software-Architektur

4.1.1 Startmenü

Der erste Baustein ist der Einstieg der Anwendung für den Nutzer. Hier wird die Möglichkeit geboten, neue Projekte anzulegen oder bestehende Projekte zu öffnen. Dieser soll immer beim Start der Anwendung erscheinen. Dem Nutzer sollen die Optionen über große Schaltflächen auf der Benutzeroberfläche zur Verfügung stehen. Das Startmenu zählt daher im ganzen als ein Modul der Software. Der Aufbau des Moduls ist in Abbildung 4.2 genauer sichtbar.

Die Klasse *MainWindow* ist eine XAML Klasse und stellt somit eine Oberfläche dar. Sie bündelt die einzelnen Oberflächen des Startmenu. Beim Start der Anwendung öffnet das *MainWindow* die erste Benutzeroberfläche der Software. Die Klasse selbst bindet die *StartMenuView.XAML* ein. Diese ist, wie in Abbildung 4.2 zu entnehmen, als UserControl angelegt. Dies fördert insbesondere den modularen Aufbau der Software. Wird in der zukünftigen Weiterentwicklung etwas an der Startmenu-Komponente geändert, so muss nicht das komplette Window verändert werden, sondern ausschließlich der UserControl des Startmenu. Nach MVVM Prinzip benötigt das Startmenu-Modul auch eine oder mehrere ViewModel-Klassen, welche die Logik enthalten. Das Startmenu enthält vier ViewModel-Klassen:

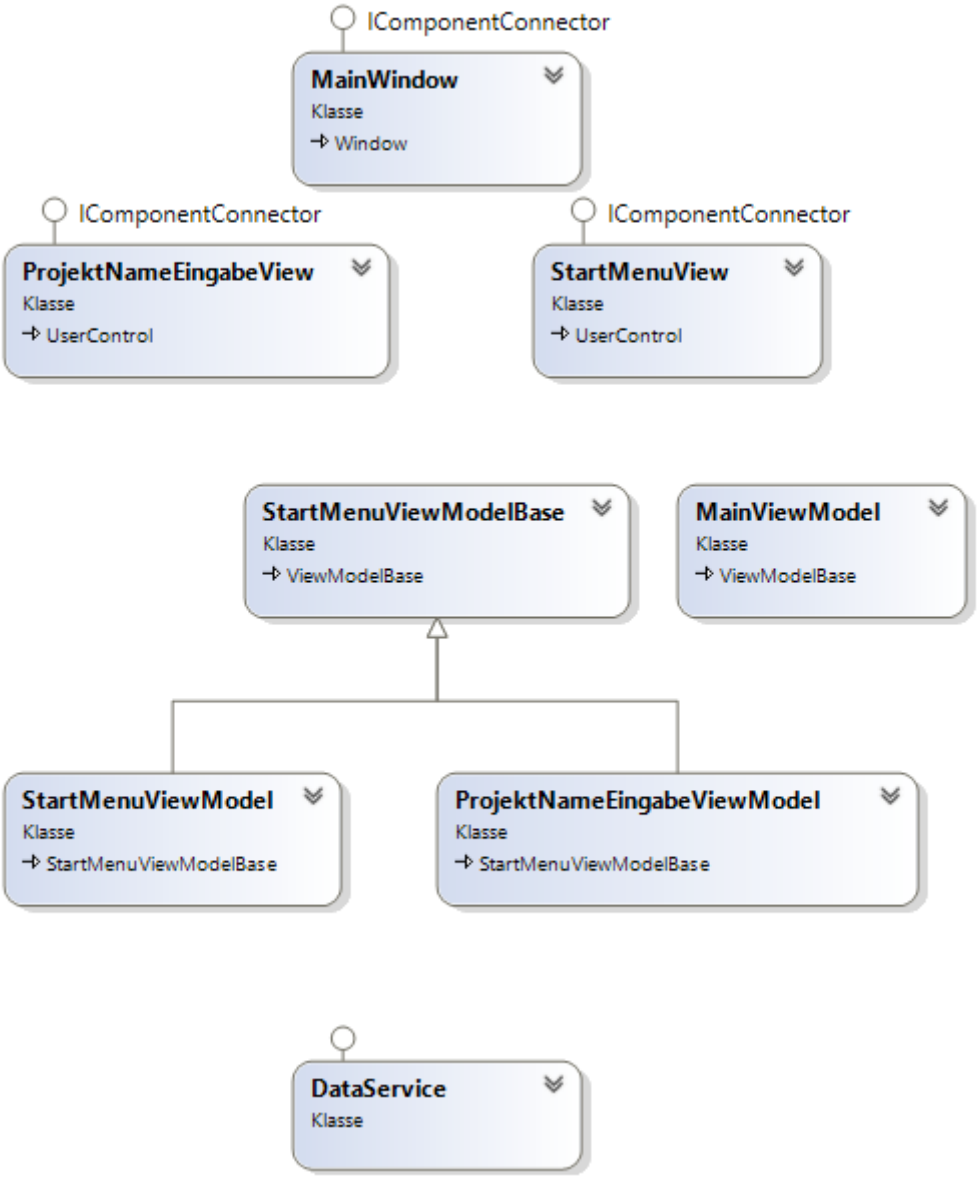

Abbildung 4.2: Klassendiagramm-Startmenü

- *StartMenuViewModelBase*
- *MainViewModel*
- *StartMenuViewModel*
- *ProjektNameEingabeViewModel*

StartMenuViewModelBase und *MainViewModel* erben von der globalen ViewModelBase Klasse, welche bereits in Kapitel 3.2 beschrieben wurden. Sie sind der zentrale Anhaltspunkt für die Logik und beinhalten die Funktionen, welche von der Benutzeroberfläche direkt gebindet werden. Neben den ViewModels, enthält das StartMenu natürlich auch noch ein Model. Dieses ist in

Abbildung 4.2 als *DataService* dargestellt. Die DataService Klasse kümmert sich um die Bereitstellung der Daten. Hier insbesondere um die Bereitstellung der vorhandenen Projektdateien, die aufgerufen und geladen werden können.
Die zweite Funktion des Startmenus beinhaltet das Erstellen eines neuen Projekts. Dafür dient das ProjektNameEingabeView UserControl. Hier hat der Benutzer die Möglichkeit, einen Namen für das neu angelegte Projekt zu definieren. Das Projekt wird dann dementsprechend, mit dem vom Benutzer definierten Namen, in einer Ordner-Struktur gespeichert. Mehr zum Ablauf der Speicherung, sowie der Ordner-Struktur ist in Kapitel **??** beschrieben. Um das Erzeugen der Datei des neuen Projekts durchzuführen, ist das *ProjektNameEingabeViewModel* zuständig.

4.1.2 Editor

Der Editor enthält die eigentlichen Kernfunktionen der Anwendung. Hier sollen die Schaltpläne entstehen. Er erscheint nach dem im Startmenu ein neues Projekt angelegt, oder ein bestehendes Projekt ausgewählt wurde. Der Editor stellt eine Zeichenfläche bereit, in der es ermöglicht wird Schaltpläne zu skizzieren. Dabei werden durch verschiedene Werkzeugleisten, einzelne Komponenten zur Auswahl gestellt. Diese sind Notwendig um einen Schaltplan zu erstellen. Das Editor Fenster selbst, besteht aus einzelnen UserControls, um künftige Änderungen an der Oberfläche zu gewährleisten, ohne das diese Änderungen an den restlichen Elementen auslösen. Auch im Editor wird wieder nach dem Schema des MVVM Patterns vorgegangen. Dabei wird für jeden einzelne Element der Oberfläche, welche in UserControls angelegt werden, eigene ViewModels sowie Models erstellt. Beispielsweise enthält der Editor ein UserControl, der Werkzeuge zum Zeichnen der Leitungen bereitstellt. Nach MVVM wurden hier die folgenden Klassen erzeugt:

- *LeitungenToolsView*
- *LeitungenToolsViewModel*
- *LeitungenToolsModel*

Der Editor umfasst Alle User Controls des Editors werden in Kapitel 4.2 genauer beschrieben.

4.1.3 Programmier Richtlinien

Um eine Übersichtlichkeit innerhalb des Projekts zu schaffen, damit auch künftig Entwickler, Änderungen an der Anwendung vornehmen können, wurden zu Beginn einige Richtlinien für die Programmierung und den Programmierstil festgelegt.

- Klassen, Variablen und Funktionen sollen aussagekräftige Namen erhalten. Wenn Möglich sollen die Namen den Inhalt bzw die Funktion der Klasse, Variable und Funktion beschreiben.
- Klassen, Variablen und Funktionen die unterschiedliche Inhalte und Funktionen beinhalten, dürfen keinen ähnlichen Namen enthalten. Die Namen sollen für die Entwickler eindeutig unterscheidbar sein.
- Aufgrund der Benutzung des MVVM Patterns, sind für Bestandteile der Anwendung meist drei Klassen enthalten. *View, ViewModel, Model.* Die zugehörigen Klassen erhalten den gleichen Namen und unterscheiden sich lediglich in der Endung. Z.b. *StartMenuView, StartMenuViewModel, StartMenuModel.*

Diese Richtlinien dienen als Leitfaden für Entwickler. Sie sind an das Konzept Clean-Code angelehnt.

4.1.4 Mockups (Mikka Jenne)

Bei der Entwicklung des allgemeinen Architekturkonzepts wurden parallel dazu auch erste grafische Entwürfe für die Darstellung der Benutzeroberflächen erstellt. Dabei wurden User-Experience und Usability Faktoren berücksichtigt. [Gruenderszene 2020]
Die User-Excperience, dt. Nutzererfahrung und Nutzererlebnis (UX) umfasst alle Berührpunkte eines Nutzers mit einem Produkt, Dienstleistung oder Service. Sie spiegelt Erfahrungen sowie auch Empfindungen und Gefühle einer Person während der Benutzung eines Produktes wieder. [Müller 2020]
Usability (dt. Gebrauchstauglichkeit), ist ein Teil der User-Experience der speziell auf die exakte Gebrauchstauglichkeit der Anwendung, Dienstleistung o.ä. abzielt. Dabei zu beachtende Faktoren sind unter anderem der Aufgabe, der Problemlösung angemessen und z.B. verständliche Anweisungen.
Die für uns wichtigsten Faktoren der User-Experience und Usability wurden aus einer Menge bestehender Merkmale selektiert und während des Designs berücksichtigt. Diese Aspekte werden in folgender Auflistung erläutert. [Degruyter 2020]

- Effizienz
  - Der Nutzer kann seine Ziele mit minimalem zeitlichem und physischem Aufwand erreichen. Das Produkt reagiert schnell auf Nutzereingaben. Das Design der Nutzerschnittstelle macht keine unnötigen Arbeitsschritte erforderlich. Quelle: ISO 9241 (Teil 11) bzw. (Teil 110 – Aufgabenangemessenheit).
- Intuitive Bedienung
  - Kann der Nutzer die Benutzerschnittstelle mit seinen vorhandenen Fähigkeiten unmittelbar und ohne jegliche Einarbeitung oder Anleitung durch andere bedienen? Quelle: Mohs et al. (2006).
- Vollständigkeit
  - Das Produkt bietet dem Nutzer alles, was er oder sie erwartet. Es fühlt sich vollständig an, auch wenn es nicht tatsächlich alles im konkreten Nutzungskontext Notwendige bieten sollte. Der Faktor Vollständigkeit bezieht sich somit mehr auf die wahrgenommene Vollständigkeit und weniger auf die Summe der aufgabenangemessenen Nutzungskontexte.

Anhand der aufgelisteten Faktoren und der Umschreibung des zu lösenden Problems der Schaltplanzeichnung sind die aufgeführten Abbildung 4.3 und Abbildung 4.4 entstanden, die es galt in der Implementierung möglichst abbildungsgetreu wiederzugeben.

Startmenü Mockup

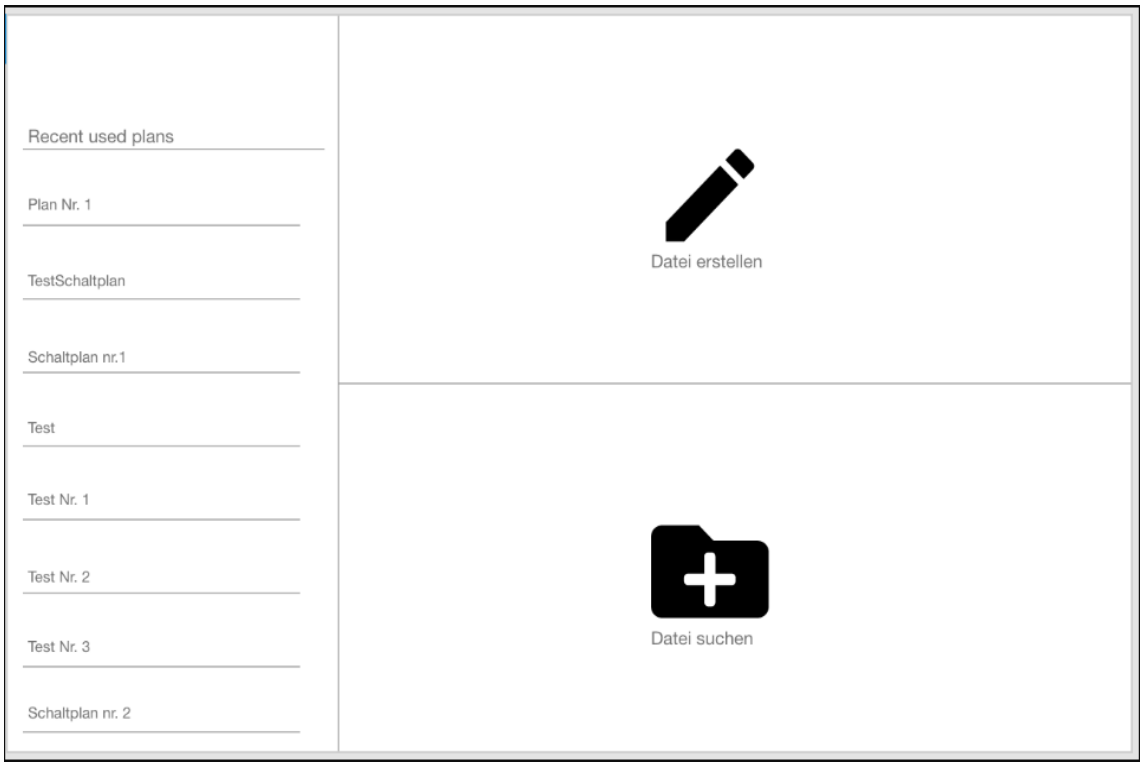

Abbildung 4.3: Startmenü Mockup

Die Benutzeroberfläche des Startmenüs ist einfach und selbsterklärend aufgebaut, sodass ohne große Worte der Umgang und der weitere Verlauf des Programms selbsterklärend ist. Die Oberfläche ist sichtlich in drei Rubriken eingeteilt, womit die wichtigsten Starteigenschaften repräsentiert werden.

Über die in Abbildung 4.3 dargestellte Auflistung, getitelt mit *Recent used plans*, werden alle erstellten Dokumente als Schnellzugriff zur Verfügung gestellt. Jede einzelne Datei kann über diesen Menüpunkt aufgerufen und im Editor geöffnet werden. Zudem bietet diese Auflistung eine schnelle Übersicht über alle erstellten Dokumente.

Das rechte obere Drittel des Bildes stellt den Service zum Erstellen einer neuen Datei zur Verfügung. Durch klicken des Bildes, bzw. der Fläche wird das eigentliche Programm angestoßen, welches in dem Kapitel 5 Implementierung ausführlich erläutert und dargelegt wird.

Das rechte untere Drittel weißt darauf hin einen Ordner öffnen zu können, damit vorhandene Dateien, z.B. aus anderen Ordnern, geöffnet und geladen werden können. Die genaue Vorgehensweise, bzw. die implementierten Funktionen werden ebenso in Kapitel 5 genauestens dargestellt.

Ein simples Layout zum Start der Applikation, während die Editoransicht, der eigentliche Kern der Anwendung, deutlich mehr Funktionen und Möglichkeiten bietet. Um die Abbildung 4.4 besser deuten zu können, wird nach Aufführung des Bildes eine Erklärung und eine kurze Begründung, wieso das Layout so gewählt wurde, stattfinden.

Editor Mockup

Abbildung 4.4: Editor Mockup

Wie in der Abbildung zu sehen ist, wird zentral eine weiße Fläche dargestellt. Diese Fläche repräsentiert ein Canvas, welches für die Zeichnung der Schaltpläne geeignet ist.
Das Canvas in WPF ist das grundlegendste Layout-Fenster für das Zeichnen von Objekten. Durch explizite Koordinaten können Elemente und Objekte in diesem Fenster positioniert werden. Die Koordinaten können relativ zu jeder Seite des Bedienfeldes im Canvas-Bereich angegeben werden.

Die Oberfläche ist aufgeteilt in drei Segmente. Dies sind die Zeichenfläche als primäre Funktion, die zur Verfügung stehenden Komponenten zum Zeichnen angeordnet um das Canvas herum und die allgemeinen Einstellungen und Optionen als Reiterflächen im linken oberen Bereich.

Das entwickelte Design ist stark an andere bekannte Softwaredesigns, wie z.B. Microsoft Word, angelehnt, da dies ein gewohntes Format ist und der Nutzer ungefähr, weiß wie das Programm zu benutzen ist, um schnell auf einen Fortschritt oder ein Ergebnis zu kommen. Trotz der Anlehnung an MS Word weicht es doch vom Design ab und kann damit nicht direkt und eindeutig verglichen werden. Vergleichbar sind z.B. die symbolisierten Flächen zum Auswählen verschiedenster Tools und Komponenten, die Einstellungsrubriken und die eigentliche Designfläche.

4.2 Softwarekonzept (Simon Leitl)

Im folgenden werden die einzelnen Bestandteile des in Kapitel 4.1.2 beschriebenes Editor, genauer erläutert.

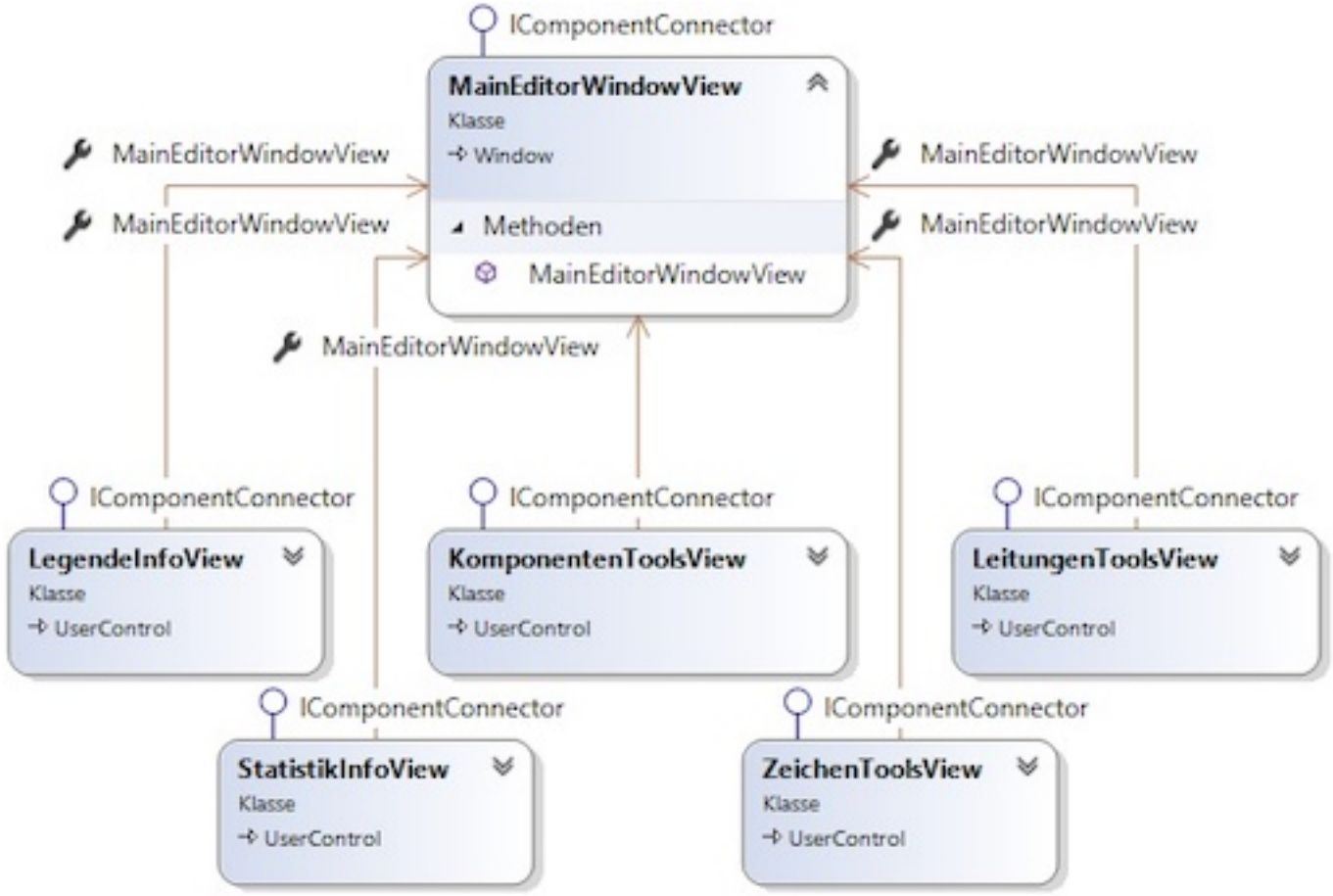

Abbildung 4.5: Editor-viewKlassendiagramm

Alle Bestandteile werden als UserControls in das MainEditorWindow eingebunden. Somit repräsentiert ein Fenster die verschiedenen Funktionen. Für die Implementierung sind diese jedoch abgekoppelte Bestandteile. Jedes UserControl kann einzeln und selbständig programmiert und gestaltet werden. Daher werden diese im Klassendiagramm (Abb. 4.5) als einzelne Klassen aufgeführt, mit Zuordnung zum MainEditorWindow.

4.2.1 Zeichenfläche

Die Zeichenfläche ist der größte Bestandteil des Editors. Hier hat der Nutzer die Möglichkeit einen Schaltplan zu zeichnen. Dabei wählt er zunächst in den Werkzeugleisten ein Werkzeug aus. Wählt er beispielsweise ein Werkzeug zum zeichnen des Grundrisses aus den ZeichenTools aus, hat er die Möglichkeit innerhalb der Zeichenfläche einen Grundriss zu zeichnen.

4.2.2 KomponentenTools

Die KomponentenTools sind eine Werkzeugleiste, welche dem Benutzer die für einen Schaltplan benötigten elektrischen Komponenten zur Verfügung stellt. Solche sind zum Beispiel eine Steckdose oder eine Leuchte. Die Komponenten verfügen über einen Namen sowie ein Symbol, um sie eindeutig zu identifizieren. Das Symbol einer Komponente wird später im Schaltplan angezeigt, sofern es benutzt wird.

4.2.3 LeitungenTools

Die LeitungenTools sind eine weitere Werkzeugleiste, welche dem Benutzer die für einen Schaltplan benötigten Leitungen zur Verfügung stellen. Hierbei gibts es folgende Leitungen zur Auswahl.

- 3 x 1,5mm
- 3 x 2,5mm
- 5 x 1,5mm
- 5 x 2,5mm

Alle Leitungen werden durch unterschiedliche Farben und Formen gekennzeichnet, damit sie im Schaltplan eindeutig und schnell erkannt werden.

4.2.4 ZeichenTools

Die ZeichenTools sind eine weitere Werkzeugleiste im Editor, welche zum Zeichnen der Gebäudeumrisse dienen. Sie bieten verschiedene Darstellungen und Formen um Gebäude zeichnerisch korrekt darzustellen.

4.2.5 LegendeInfo

Die LegendeInfo ist ein Bestandteil des Editors, welcher eine Legende zum Schaltplan darstellt. Die Legende dient dazu den Schaltplan zu lesen und zu verstehen. Sie stellt die Bedeutung von Symbolen dar. So kann der Schaltplan auch schnell von dritten, die ihn nicht erstellt haben, überblickt werden.

4.2.6 StatistikInfo

Die StatistikInfo ist ein Bestandteil des Editors, welcher eine Übersicht über die verwendeten Werkzeuge und Komponenten darstellt. Die Statistik soll dem Benutzer einen Überlick verschaffen, wie viele Werkzeuge bzw. Komponenten er in seiner Zeichnungen benutzt hat. Hat er beispielsweise vier Steckdosen in der Zeichnung angebracht, so zeigt ihm die Statistik an, dass vier Steckdosen verwendet wurden. Dies erspart dem Benutzer das abzählen innerhalb der Skizze.

Kapitel 5

Implementierung

In diesem Kapitel geht es um die eigentliche Entwicklung des Projekts. Die Recherchen von wichtigen und notwendigen Informationen, die Planungen des generellen Vorgehens, Struktur- sowie Architekturpläne wie die Implementierung vonstatten gehen soll und wie die Planungen im Endeffekt umgesetzt wurden.

5.1 Use Case1: Startmenü

5.1.1 Frontend (Mikka Jenne)

Nach der grundlegenden theoretischen Ausarbeitung der Softwarearchitektur, dem allgemeinen Aufbau und dem Design ging der praktische Abschnitt dieser Arbeit los. Die zu Anfang definierten Use Cases wurden in logischer, aufeinander aufbauender Reihenfolge aufgestellt und in jeweils Front- und Backend Entwicklung aufgeteilt. Die ersten Entwicklungsschritte waren das Implementieren der grafischen Benutzeroberfläche (GUI) des Startmenüs, um daraufhin das Backend passend zu dieser Oberfläche zu erstellen. Somit wurde ersichtlich, welche Funktionen diese erste Benutzeroberfläche umfasst.

Bei der Umsetzung der Startmenu-UI wurde sich an dem MockUp in Abbildung 4.3 orientiert, bzw. an die vorangestellten Überlegungen angelehnt. Dabei wurde darauf geachtet, dass der Unterschied des Designs nur in feineren Details, z.B. den Identifizierungs-Icons von dem eigentlichen Objekt abweicht, um weiteren Umstrukturierungen aus dem weg zu gehen.
Im Vergleich zur Abbildung 4.3 wurden die in Kapitel 4 genannten Anforderungen umgesetzt und so ein anwendungsfreundliches und übersichtliches Startfenster erzeugt.
Das UserControl basierende XAML wurde mit einem Grid, welches in Kapitel 3 näher beschrieben ist, versehen und in drei Bereiche aufgeteilt. Zum einen die Rubrik zum Erstellen von neuen Projekten, die Rubrik zum Öffnen von vorhandenen Projekten und zum anderen die Rubrik zum Anzeigen von vorhandenen Projekten, die sich lokal auf dem Computer befinden.
Auf die Ansicht der Liste der vorhandenen Projekte wird im Nachgang in dem Code-Beispiel 5.1 noch genauer eingegangen.
Um das Aussehen für den Benutzer interessanter und anschaulicher zu gestalten wurde der Hintergrund des Fensters und diverse Maus-Highlighting-Effekte mit Blautönen versehen.
Die genannten Grid-Rubriken für das Erstellen und Öffnen von neuen Projekten wurde mit einem Button implementiert, um das jeweilige Event auslösen zu können und um auf die referenzierten

Seiten zu gelangen. Diese Referenzierungen wurden anschließend im *Backend* realisiert.
Die folgende Abbildung repräsentiert das Aussehen des Startmenüs, wenn die Applikation gestartet, bzw. ausgeführt wird.
Nachdem dieses Objekt erstellt wurde, konnte mit der weiteren Entwicklung fortgefahren werden und unabhängig zu der Backend-Entwicklung dieses Use Cases ein weiteres benötigtes Benutzerfenster, die Eingabe eines Projektnamens für ein neues Projekt, erstellt werden.

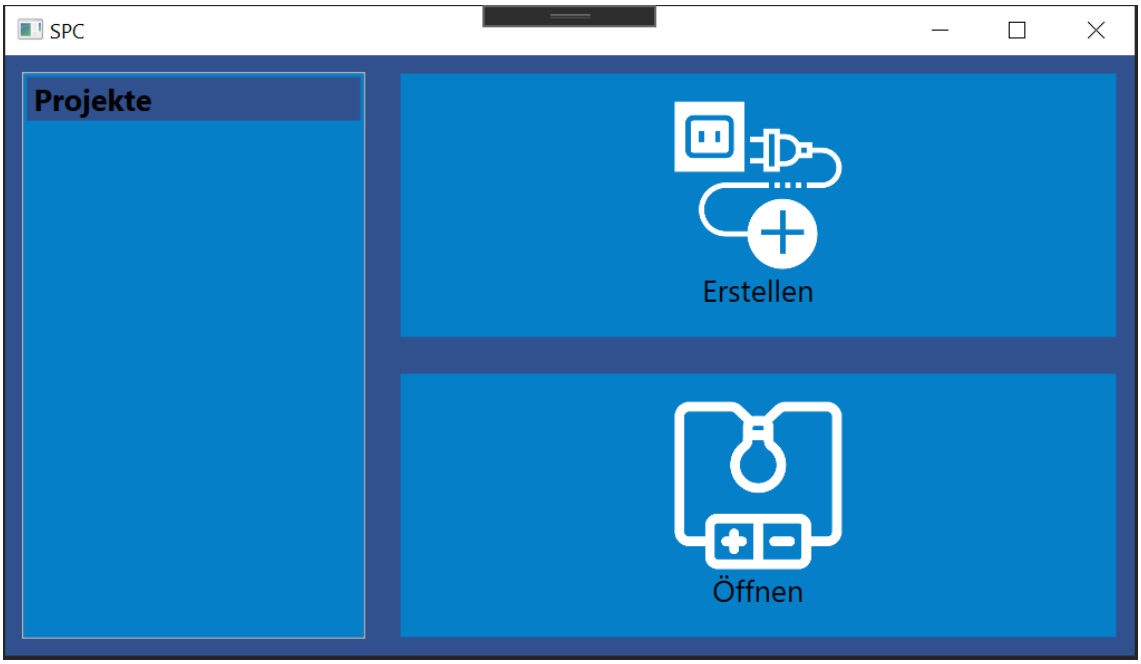

Abbildung 5.1: Startmenü View

Um den Prozess, ein Projekt zu erstellen, fließend übergehen zu lassen wird zwischen der Startmenu-Seite und dem eigentlichen Editor eine zusätzliche Komponente benötigt, die zwischen den Komponenten transferiert und alle benötigten Informationen von dem Benutzer einholt, benötigte Ressourcen generiert und lädt.

Beim betätigen des Erstellen-Buttons wird das UserControl der *ProjektNameEingabeView*-Seite aufgerufen. In diesem Schritt der Applikation kann der Benutzer dem Projekt einen Namen vergeben und die Datei so im Verzeichnis anlegen. Die dazugehörige GUI beinhaltet eine schlichte und einfach gehaltene Eingabemaske, in der lediglich ein Textfeld zur Eingabe des Projektnamens vorhanden ist. Ebenso ist diese mit zwei Buttons versehen die jeweils den Prozess zum einen voranbringen durch den (*WeiterButton*) oder zum anderen auf den vorherigen Stand zurückgehen, indem der (*ZurueckButton*) gedrückt wird.

Abbildung 5.2: Projektname View

Der Code-Ausschnitt zeigt eine vereinfachte From des implementierten UserControls der Eingabe eines Projektnamens. Dabei wurde der *Code behind* weggelassen, da in diesem Fall nur das Frontend-Design betrachtet wird.
Dies kann der folgenden Abbildung entnommen werden.

```
<UserControl
    DataContext="{Binding␣Main,␣Source={StaticResource␣Locator}}">
    <Grid>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <TextBlock Text="Projektname␣eingeben" Grid.ColumnSpan="2"
            FontSize="24"
            TextAlignment="Center"
            HorizontalAlignment="Center" VerticalAlignment="Bottom"/>
            <TextBox Grid.Column="0" x:Name="ProjektNameTextFeld"/>
            <Button Grid.Column="0" x:Name="ZurueckButton"
            Grid.Row="2" Content="Zurueck"/>
            <Button x:Name="WeiterButton" Grid.Row="2"
            Grid.Column="1" Content="Weiter"/>
        </Grid>
    </Grid>
</UserControl>
```

Code-Beispiel 5.1: Projektname-View XAML-Code

Als Transferseite wurde diese vom Design unaufmerksam gehalten, da ein Benutzer in der Regel nicht viel Zeit auf dieser Seite aufwenden muss. Nach dem Ausfüllen des Textblocks und der Bestätigung durch den *WeiterButton* wird der Benutzer direkt auf den eigentlichen Editor weitergeleitet und ist somit in der Hauptansicht der eigentlichen Anwendung.

5.1.2 Backend (Simon Leitl)

Für die Funktionalität des Startmenüs sind mehrere Klassen verantwortlich. Nach MVVM-Prinzip findet die Logik innerhalb der ViewModel-Klassen statt. Speziell für das Startmenü lassen sich die Funktionalitäten in drei Bereiche aufteilen.

- Das erstellen eines neuen Projekts
- Die Anzeige der gespeicherten Projekte im automatisch generierten Projektordner
- Das öffnen eines gespeicherten Projekts

Der Nutzer hat die Möglichkeit ein neues Projekt über die Schaltfläche *Erstellen* zu erstellen. Hierbei muss im Hintergrund des Programms eine neue Datei angelegt werden. Die Datei wird

später Daten enthalten, mit welchen die Schaltpläne gespeichert werden können. Da der Aufwand ein eigenes Dateiformat zu erstellen für diese Arbeit zu hoch wäre, wird das Textdatei (txt) Format verwendet. Dies eignet sich um die Daten der Zeichnung abzuspeichern. Mithilfe von Input/Output Streams ist es so möglich, die Daten in die Textdatei zu schreiben und zu lesen. Um die gespeicherten Projekte an einem gemeinsamen Ort abzulegen wird zunächst beim ersten Ausführen, automatisch ein Ordner erstellt. In diesem werden zukünftig die neu erstellten Projekte, also Textdateien, abgelegt. Um den gesamten Prozess zu starten wird im Startmenu *Erstellen* ausgewählt. Von dort wird man innerhalb des Fensters weitergeleitet auf die Benutzeroberfläche ProjektNameEingabeView, welche die Möglichkeit bietet ein Projektname über das vorhandene Eingabefeld einzugeben. Über den Button *weiter* wird das Projekt erstellt. Hierbei wird innerhalb der Logik ein RelayCommand ausgeführt. Dieser kommuniziert die Interaktion über das, in der XAML festgelegten, Binding. Die Logik wird in der ProjektnameEingabeViewModel Klasse als Methoden abgebildet und ausgeführt. Wird also mit dem Button *weiter* interagiert, so wird die Methode *NewProjektCommand* im folgenden Code LiteraturNewProjektCommand aufgerufen.

```
1     public RelayCommand NewProjektCommand { get; private set; }
```

Code-Beispiel 5.2: NewProjektCommand

Über die enthalten get Methode wird die Interaktion übergeben. Der RelayCommand selbst ruft eine weitere Methode zur Ausführung auf.

```
1     public void CreateNewProject()
2         {
3             if (CheckProjectDirectory() == true)
4             {
5                 CreateProjectFile();
6             }
7             else
8             {
9                 CreateProjectDirectory();
10                CreateProjectFile();
11            }
12        }
```

Code-Beispiel 5.3: CreateNewProject

Die Methode *CreateNewProject* ist der Startpunkt der Ablauffolge um eine neues Projekt und damit eine neue Datei zu erstellen. Es überprüft zunächst ob der Ordner, in dem die Dateien erzeugt werden sollen, vorhanden ist. Dies gelingt über eine if-Abfrage, die den boolschen zurückgegebenen Wert einer Methode auf Richtigkeit überprüft. Bei der Methode handelt es sich um die nachfolgende Methode mit dem Namen *CheckProjectDirectory*. Sie gibt *true/false* als Ausgabe zurück. Der Ordner für gespeicherte Dateien, sollte sich im selben Pfad bzw. Ablageort befinden wie die ausführende kompilierte .exe des Programms.

```
1     public Boolean CheckProjectDirectory()
2     {
3        return Directory.Exists("Savings");
4     }
```

Code-Beispiel 5.4: CheckProjectDirectory

Ist der Rückgabewert *true* wird eine weitere Methode *CreateProjectFile* ausgeführt. Diese enthält Funktionen zum erstellen der Datei. Wird *false* zurückgegeben, ist das Verzeichnis nicht vorhanden und muss mittels der Methode *CreateProjectDirectory* erstellt werden.

```
1     public void CreateProjectDirectory()
2         {
3             Directory.CreateDirectory("Savings");

5         }
```

Code-Beispiel 5.5: CreateProjectDirectory

CreateProjectDirectory nutzt System.IO.Directory für das Erstellen des Verzeichnisses. Das Verzeichnis soll mit dem Namen *Savings* erstellt werden.
Jetzt wurde sichergestellt, dass das Verzeichnis zum speichern der Dateien existiert und verfügbar ist. Im nächsten Schritt wird die Datei erzeugt. Hierfür dient die *CreateProjectFile* Methode. Zunächst wird allerdings der vom Benutzer gewählte Projektname benötigt. Dieser kann über die Benutzereingabe erlangt werden. Hierfür wurde eine weitere Methode implementiert, welche an die View gebindet wurde.

```
1     public string ProjektName
2     {
3         get { return projektName; }
4         set { projektName = value; }
5     }
```

Code-Beispiel 5.6: ProjektName

Sie gibt den Eingabewert des Nutzer zurück und speichert ihn in einer globalen Variable *projektName*. Diese Variable wird im folgenden dazu verwendet die Datei mit dem eingegebenen Namen zu erzeugen.

```
1     public void CreateProjectFile()
2     {
3         String path = "Savings/" + projektName + ".txt";
4         using (FileStream fs = File.Create(path)){}
5         MainEditorWindowView mw = new MainEditorWindowView();
6         mw.Show();
7     }
```

Code-Beispiel 5.7: ProjektName

Im ersten Schritt der Methode wird der Pfad definiert, der benötigt wird um die Datei abzulegen. Hierbei wird der Name des Verzeichnis angegeben sowie der Name des Projekts und die

Dateiendung. Über einen *Filestream* wird dann die Datei, mit dem zuvor festgelegten Pfad erzeugt. Da dies der abschließende Schritt beim Erstellen eines neuen Projektes war, muss das die nächster Benutzeroberfläche geladen werden. Es handelt es sich um den Editor. Eine neue Instanz des MainEditorWindowView wird erstellt und durch den Befehl *Show()* geöffnet.

Die zweite Funktion des Startmenü beinhaltet die Anzeige der gespeicherten Projekte. Sie soll einen schnellen Zugriff auf die vorhandenen Projekte gewährleisten. Hierzu werden die Projektdateien in Form einer Liste untereinander ausgegeben. Für die Implementierung dieser Funktion müssen also alle Dateien, die innerhalb des Verzeichnis liegen, abgerufen und auf der Benutzeroberfläche ausgegeben werden. Da es Möglich ist ganze Listen an die View zu binden, werden die Dateinamen in einer Liste gespeichert. Die Implementierung befindet sich in der MainViewModel-Klasse des Startmenü.

```
1         public void showProjekte()
2         {
3             string path = @"\Savings";

5             ProcessDirectory(path);
6             splitString();
7         }
```

Code-Beispiel 5.8: showProjekte

In der Methode *showProjekte* ist zunächst der Pfad für den Ort des Verzeichnis deklariert. Mit diesem wird eine weitere Methode *ProcessDirectory(path)* aufgerufen. Durch diese Methode werden alle Dateien innerhalb des Verzeichnis in eine Liste geschrieben.

```
1     public void ProcessDirectory(string targetDirectory)
2     {
3         // Process the list of files found in the directory.
4         string[] fileEntries = Directory.GetFiles(targetDirectory);

6         foreach (string fileName in fileEntries)
7         filePaths.Add(fileName);
8     }
```

Code-Beispiel 5.9: ProcessDirectory

Über die in System.IO enthalten Methode *Directory.GetFiles(Path)* werden alle Dateien die innerhalb des zuvor definierten Pfad liegen, ausgegeben. Hier werden diese gleich in einen Array *fileEntries* gespeichert. Mithilfe einer Schleife werden die einzelnen Dateinamen in eine weitere Liste *filePaths* geschrieben, welche global definiert wurde. Da die Methode den vollständigen Dateipfad und nicht nur den Namen der Datei ausgibt, muss der *String* zerteilt werden. Dafür wird der String im nachfolgenden Code gesplittet.

```
1     public void splitString()
2         {
3             for (int i = 0; i < filePaths.Count; i++){
4                 string name = filePaths[i];
5                 string[] split = Regex.Split(name, "\\");
6                 _files.Add(split[split.Length-1]);
7                 }
8         }
```

Code-Beispiel 5.10: splitString

Die Liste *filePaths*, welche Global definiert wurde und die Dateipfade beinhaltet wird hier über *Regex.Split()* aufgeteilt. Die Bedingung für die Aufteilung ist der sogenannte Backslash \. Jedesmal wenn innerhalb der Zeichenkette ein Backslash erscheint, wird an dieser Stelle geteilt und als neue Position in die Liste geschrieben. Da der Dateiname der letzte Teil des Pfads ist, kann der letzte Eintrag der Liste als Dateinamen ausgewählt werden. Damit alle Einträge der Liste gesplittet und in die neue Liste *files* eingetragen werden, wird in einer Schleife über die Liste iteriert. *files* wird in der folgenden Methode dazu genutzt um an die View anhand eines Bindings übergeben zu werden.

```
1     public ObservableCollection<string> getFileList
2     {
3         get
4         {
5             return _files;
6         }
7     }
```

Code-Beispiel 5.11: getFileList

Die Methode *getFileList* liefert die Liste mit den Dateinamen als Observablecollection zurück. Das UI Element kann über ein Binding zu dieser Methode die Namen auf der Benutzeroberfläche abbilden. Ein Code-Ausschnitt der Listenansicht-Implementierung soll einen Einblick verschaffen wie die *Starmenu-XAML* in Bezug auf die Liste aufgebaut und Über die in System.IO enthalten Methode *Directory.GetFiles(Path)* werden alle Dateien die innerhalb des zuvor definierten Pfad liegen, ausgegeben. Hier werden diese gleich in einen Array *fileEntries* gespeichert. Mithilfe einer Schleife werden die einzelnen Dateinamen in eine weitere Liste *filePaths* geschrieben, welche global definiert wurde.
Ein Code-Ausschnitt der Listenansicht-Implementierung soll einen Einblick verschaffen wie die *Starmenu-XAML* in Bezug auf die Liste aufgebaut und implementiert ist.

5.2 Use Case 2: Toolbox

Nachdem die Dateien und Objekte zur Erstellung eines Projekts implementiert waren, wurde sich dem Frontend des Editors gewidmet, um die eigentlichen Funktionen bereitzustellen, damit die Implementierung des Editors umgesetzt werden konnte.
Parallel zu dem Design des Startmenüs wurden ebenso die grafische Darstellung des Editors erstellt. Das dazugehörige MockUp wurde in Kapitel 4 Abbildung 4.4 bereits erwähnt und genauestens beschrieben. Da die Überlegungen zum Design im Voraus getätigt und eine visuelle Vorarbeit geleistet wurde, konnte sich bei der Entwicklung lediglich an dem grafisch dargestellten Entwurf orientiert werden.

5.2.1 Frontend (Mikka Jenne)

Das UI-Element des Editors besteht grundlegend aus einem *Window-XAML* und ist in ein Raster aufgeteilt. Um das Raster zu verdeutlichen, wird die Definition dieses Rasters aufgeführt.

```
1 <Grid.ColumnDefinitions>
2     <ColumnDefinition/>
3     <ColumnDefinition/>
4     <ColumnDefinition/>
5     <ColumnDefinition/>
6     <ColumnDefinition/>
7     <ColumnDefinition/>
8     <ColumnDefinition/>
9     <ColumnDefinition/>
10 </Grid.ColumnDefinitions>
11 <Grid.RowDefinitions>
12     <RowDefinition Height="0.3*"/>
13     <RowDefinition/>
14     <RowDefinition/>
15     <RowDefinition/>
16     <RowDefinition/>
17     <RowDefinition/>
18     <RowDefinition/>
19     <RowDefinition/>
20 </Grid.RowDefinitions>
```

Code-Beispiel 5.12: Editor Raster-Layout

Mit dieser Implementierung wird das *Window* in ein Raster aufgeteilt mit Rechtecken gleicher Größe. Durch Verschmelzung einzelner Rechtecke können größere Flächen gebildet werden, um dementsprechend mehr Inhalt darin anzeigen zu können. So kann z.B. ein *UserControl* referenziert werden, welches viel Platz benötigt. In dem Fall des Editors ist das die eigentliche Zeichenfläche.

Das Editor-Element ist als ein separates *Window* angelegt. Es öffnet sich nach dem betätigen des *WeiterButton* des *UserControls-ProjektNameEingabe* ein neues Fenster mit der Editor Ansicht. Auf diesen Schritt wird nicht weiter drauf eingegangen, da diese Aktion über das Backend verwaltet wird.

Der aktuelle Entwurf des Editors ist in folgender Abbildung zu entnehmen. Die Umsetzung ähnelt der Abbildung 4.4 in grundlegenden Struktur.

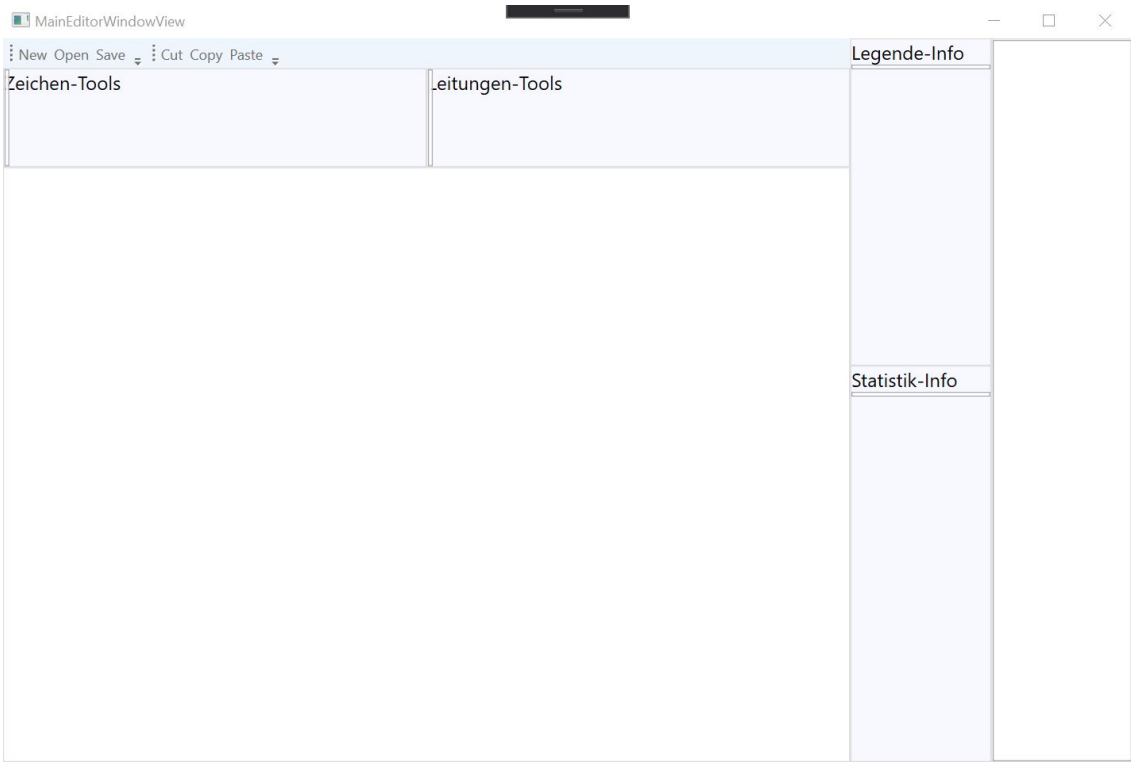

Abbildung 5.3: Editor View

Die *Window*-Datei ist durch eine Anreihung von *Grid-Elementen* strukturiert und in sieben unterschiedlich große Flächen gegliedert. Die Rasterelemente sind für eine jeweilige Komponente als Platzhalter gedacht, bzw. für ein bestimmtes *UserControl* vorgemerkt. Alle Komponenten sind als ein *UserControl* ausgelagert, um die Code-Struktur der Hauptseite zu entlasten und übersichtlicher zu gestalten.
Das *Window-XAML*-Element is so als zentraler Ort für alle *UserControls* bestimmt.

In folgender Abbildung wird die Aufteilung der einzelnen Felder in der *XAML*-Datei veranschaulicht. Um genauer zu verdeutlichen wie die einzelnen *UserControls* in dem Editor-Fenster integriert werden, wird ergänzend zu der Abbildung 5.3 ein Code- Ausschnitt von einer der *UserControls* aufgeführt.

```
<Window
    xmlns:local="clr-namespace:SPC3.SPC.Editor.Views">
    <DockPanel Grid.Column="0" Grid.Row="2"
     Grid.RowSpan="6" Grid.ColumnSpan="6">
        <Grid>
            <local:ZeichenFlacheView x:Name="ZeichenFlaecheView"/>
        </Grid>
    </DockPanel>
</Window>
```

Code-Beispiel 5.13: UserControl-Referenz Code-Auschnitt

Der Verweis durch *local* gibt Informationen darüber in welchem Namespace oder genauer in welchem Ordner des Projekts dieses Objekt zu finden ist, um das Element referenzieren zu können. Durch diesen Aufruf wird dem Hauptfenster eindeutig mitgeteilt welche Klasse, konkret welches *UserControl* in diesem Feld geladen werden soll. Auf diese Weise werden alle *UserControls* in die *MainWindow-XAML*-Datei geladen.

```
<ListView Grid.Column="0" x:Name="viewUsedProjects" Background="#067FC9"
    Margin="10">
    <ListView.ItemTemplate>
        <DataTemplate>
            <WrapPanel>
                <TextBlock Text="{Binding␣ProjektPfad}"></TextBlock>
            </WrapPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
    <ListViewItem Background="#31508E" FontWeight="Bold" FontSize="18">
    Projekte</ListViewItem>
</ListView>
```

Code-Beispiel 5.14: Startmenu-Recent-Projects-Code

Das ListView-Item wird eindeutig einer *Grid.Column* zugeordnet und mit einer *Background-Color* Farbe versehen. Zusätzlich wird dem Element einen Namen vergeben, um bei einer Referenzierung das Objekt eindeutig ansprechen zu können.
Der Ausschnitt zeigt, wie der Aufruf, bzw. das *Binding* eines Objekts, in dem Fall das dynamisch generierte *ListView-Item*, angebunden und auf den Code im Backend verwiesen wird.

5.2.2 Backend (Simon Leitl)

Die Toolbox-Werkzeug beeinhalten, wie bereits innerhalb dieser Arbeit erwähnt, Funktionen und Komponten die nötig sind um den Schaltplan zu zeichnen. Der Aufbau der Elemente innerhalb der Toolboxen wird im folgenden Abschnitt exemplarisch anhand der Elektronischen Komponenten erläutert.

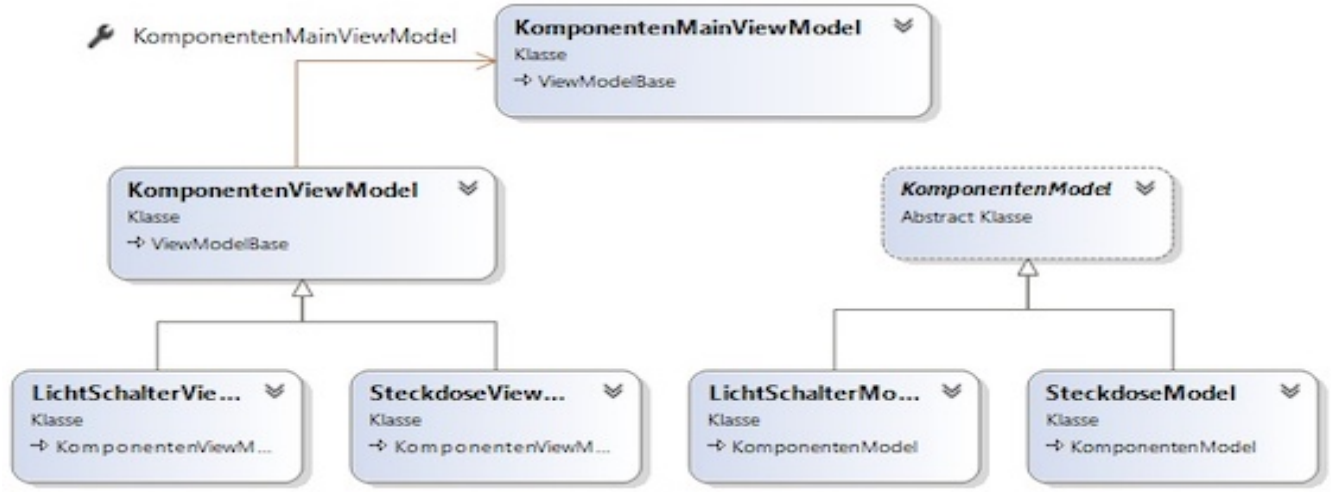

Abbildung 5.4: Klassendiagramm Toolbox

Das oberhalb abgebildetet Klassendiagramm gibt die Klassen, welche die Komponenten repräsentieren wieder. Auch hier lassen sich wieder Models und ViewModels gemäß MVVM wiederfinden. Dabei enthalten die Models die Struktur einer Komponente. In diesem Fall den Namen der Komponenten, sowie Eigenschaften des Symbols. Die ViewModels dienen dazu die Struktur der KomponentenModels abzubilden und mit einer Logik zu versehen. Außerdem werden die einzelnen Komponenten später über die ViewModels aufgerufen, da nicht direkt mit den Models kommuniziert wird. Denn diese sollen nur die Datenstruktur festlegen. In diesem Beispiel werden zwei Komponenten behandelt. Ein Lichtschalter und eine Steckdose sollen zur Verfügung gestellt werden. Diese sollen später in den Schaltplan integriebar sein. Beide Klassen, LichtSchalterModel und SteckdoseModel erben von der abstrakten Klasse KomponentenModel. Die abstrakte Klasse gibt die allgemeine Struktur für die Models der Komponenten vor. Wenn eine weitere Komponente hinzugefügt werden soll, muss jediglich ein Model erstellt werden, das von der abstrakten KomponentenModel Klasse erbt. Die Methoden und Eigenschaften sind dann aufgrund der Vererbung vorgegeben. Somit gelingt eine einfache Erweiterung.

```
1     abstract class KomponentenModel
2     {
3         public KomponentenModel(){}

5         public String komponentenName;
6         private String komponentenBeschreibung;
7         private String symbolPfad;

9         public String KomponentenName
10        {
11            get -> komponentenName;
12            set -> komponentenBeschreibung = value;
13        }


16        public String KomponentenBeschreibung
17        {
18            get -> komponentenBeschreibung;
19            set -> komponentenBeschreibung = value;
20        }

22        public String symbolPfad
23        {
24            get -> symbolpfad;
25            set -> symbolPfad = value;
26        }
27    }
```

Code-Beispiel 5.15: KomponentenModel

Der konkrete Aufbau für eine KomponentenModel Klasse kann dem Code-Beispiel 5.15 entnommen werden. Es werden zunächst die Eigenschaften als drei globale Felder in Form von *String* deklariert. Sie enthalten den Namen, eine Beschreibung und den Pfad zur Bilddatei der Komponente. Über öffentliche Getter- und Settermethoden ist es Möglich die Eigenschaften zu setzen und zu lesen. Es wird allerdings nicht empfohlen die Eigenschaften innerhalb des Systems zu ändern. Eine Änderung wird nur innerhalb der Modelklasse empfohlen. Die Methode wird jedoch implementiert um sie in möglichen Fällen bereitzustellen.

```
1     class LichtSchalterModel : KomponentenModel
2     {
3       private string komponentenName = "Lichtschalter";
4       private string symbolPfad = @"\KomponentenPictures\lichtschalter.png";

6       public string getKomponentenName()
7       {
8           return komponentenName;
9       }

11      public string getSymbolPfad()
12      {
13          return symbolPfad;
14      }
15    }
```

Code-Beispiel 5.16: KomponentenModel

Im Code-Beispiel 5.16 ist die konkrete Ausführung einer KomponentenModel Klasse dargestellt. Hier wird der Name, sowie der Symbolpfad festgelegt. Mithilfe von Gettermethoden werden diese zurückgegeben und bereitgestellt, sodass sie von den ViewModels im benutzt und implementiert werden können.

```
1     public class LichtSchalterViewModel : KomponentenViewModel
2     {
3         LichtSchalterModel lichtSchalterModel = new LichtSchalterModel();
4         public LichtSchalterViewModel()
5         {
6             Name = lichtSchalterModel.getKomponentenName();
7             Symbol = lichtSchalterModel.getSymbolPfad();
8         }
9     }
```

Code-Beispiel 5.17: LichtSchalterViewModel

Parallel zur Model Klasse einer Komponente, in diesem Fall der Steckdose, gibt es eine ViewModelKlasse der Komponente. Diese trägt den Klassennamen LichtSchalterViewModel. Sie implementiert, die im nachfolgenden Code-Beispiel 5.18 beschriebene KomponentenViewModel Klasse. Um die Daten der Model Klasse zu nutzen bildet die LichtSchalterViewModel Klasse eine Instanz ihrer zugehörigen Model Klasse. Mit dieser Instanz werden die Daten anhand der definierten Gettermethoden abgerufen.

```
1     public class KomponentenViewModel : ViewModelBase
2     {
3         public string _name;
4         public string _symbol;


7         public string Name
8         {
9             get { return _name;}
10            set { Set(() => Name, ref _name, value); }
11        }

13        public string Symbol
14        {
15            get { return _symbol; }
16            set { Set(() => Symbol, ref _symbol, value); }
17        }

19        public KomponentenMainViewModel KomponentenMainViewModel
20        {
21            get => default(KomponentenMainViewModel);
22            set{}
23        }
```

Code-Beispiel 5.18: KomponentenViewModel

Betrachtet man nun die KomponentenViewModel Klasse, welche in Code-Beispiel 5.18 dargestellt wird, lässt sich eine Ähnlichkeit zu den Models erkennen. Diese ViewModel Klasse stellt die Methoden zur Verfügung um den Namen sowie den Pfad des Symbols, einer Komponenten, aus dessen ViewModelKlasse zu lesen. Diese Methode hätten prinzipiell auch im ViewModel der Komponenten definiert werden können. Sprich das LichtSchalterViewModel hätte selbst diese Getter- und Settermethoden implementieren können. Allerdings wäre das dann bei jedem weiteren ViewModel einer Komponente nötig. Um duplizierten Code einzusparen wurden diese Methoden ausgelagert in die KomponentenViewModel Klasse im Code-Beispiel 5.18. Sie kann einfach in allen weiteren ViewModelKlassen der Komponenten implementiert werden und so die Namen und Symbolpfade abrufen.

Jetzt wo geklärt ist, wie die Daten aus den Models gewonnen werden können, fehlt noch die eigentliche Logik. Im nächsten Schritt sollen die Komponenten mit ihren Namen und Symbolen in einer Liste auf der Benutzeroberfläche bereitgestellt werden. Dazu sollen die einzelnen ViewModels in eine Liste geladen werden. Hierfür ist die KomponentenMainViewModel Klasse vorhanden. Sie vereint die gemeinsame Logik, so wie den gemeinsamen Gebrauch der einzelnen ViewModelKlassen in einem. Außerdem werden ihre Methoden an die Benutzeroberfläche gebindet. Um in MVVM ViewModels global im Projekt bereitzustellen gibt es eine ViewModelLocator Klasse. Sie wird verwendet um die ViewModels zu lokalisieren und innerhalb des gesamten Projekts bereitzustellen. Dies hat zum Vorteil, dass die ViewModel Klassen nicht jedesmal bei Gebrauch einzeln instanziert werden müssen. Neben der Backend Implementierung nutzen auch die XAML Dateien des Frontends den ViewModelLocator um die Quelle ihrer Bindings festzulegen.

```
1     public KomponentenMainViewModel KomponentenMainViewModel
2     {
3      get
4      {
5       return ServiceLocator.Current.GetInstance<KomponentenMainViewModel>();
6      }
7     }
```

Code-Beispiel 5.19: viewModelLocator

Eine Instanzierung der KomponentenViewModel Klasse wird in Code-Beispiel 5.19 abgebildet. Es wird eine Instanz der Klasse zurückgegeben, welche dann global im Projekt verwendet werden kann. Im nächsten Code-Beispiel 5.20 kommt diese Instanz zum Einsatz. Hier wird eine ObservableCollection verwendet, die alle Instanzen der KomponentenViewModel Klasse bündelt und in einer Liste ausgibt.

```
1     public ObservableCollection<KomponentenViewModel> ViewModelList
2     {
3         get
4         {
5             return _viewModelList;
6         }
7     }
```

Code-Beispiel 5.20: viewModelList

Die zurückgegebene ViewModelList enthält nun die definierten ViewModels. Über diese ist es möglich die Daten der einzelnen Komponenten zu lesen. Um nun die Namen letztendlich auf der Benutzeroberfläche anzuzeigen, wird eine weitere Methode erstellt welche von der XAML gebindet werden kann. Sie enthält eine Liste vom Typ *String* mit den Namen aller definierten Komponenten. Sie wird im Code-Beispiel 5.21 dargestellt.

```
1     public List<string> ViewModelNameList
2             {
3                 get { return _viewModelNameList; }
4             }
```

Code-Beispiel 5.21: viewModelNameList

Die ViewModelNameList Methode vom Typ *List<String>* gibt die Liste mit allen Namen der Komponenten zurück.
Damit die Benutzeroberfläche die Komponenten in einer Liste wiederverwendet und darstellt, wurde die von WPF bereitgestellte ListBox Methode verwendet. Sie gibt die Daten, die sie vom Binding erhält in einer Liste aus.

```
1 <ListBox ItemsSource="{Binding␣ViewModelList}"
2 ItemTemplate="{StaticResource␣komponenten}"/>
```

Code-Beispiel 5.22: KomponentenToolsView XAML-Code

Als ItemsSource enthält die Listbox die ViewModelList. Dadurch erkennt sie wie viele Komponenten vorhanden sind und wie die Daten eines zugehörigen strukturiert sind. Um nun festzulegen

wie die Komponenten innerhalb der Liste angezeigt werden sollen, muss ein Custom Template definiert werden. Diese soll beschreiben, dass zuerst das Symbol und darunter der Name dargestellt werden soll.

```
1 <DataTemplate x:Key="komponenten">
2     <StackPanel>
3         <Image Source="{Binding␣ViewModelSymbolList}"/>
4                 <TextBlock Text="{Binding␣ViewModelNameList}"/>
5         </StackPanel>
6 </DataTemplate>
```

Code-Beispiel 5.23: KomponentenToolsView XAML-Code

Hierfür wird ein DataTemplate in den globalen Ressourcen der XAML Datei gebildet. Das Bild wird ales Image ausgegeben und erhält seinen Pfad über das Binden der ViewModelSymbolList Methode. Der Name wird in einem Textblock dargestellt und wird über die ViewModelNameList Methode abgerufen.

In diesem gesamten Verfahren können nun weitere Komponenten hinzugefügt werden. Dafür muss eine Model Klasse mit der Beschreibung der Komponente, sowie eine zugehörige ViewModel Klasse angelegt werden. Die restlichen Strukturen sind vorhanden. So ermöglicht es eine modulare und einfache Erweiterbarkeit für die gesamte Softwarearchitektur. Bei der Toolbox LeitungenTools wurde nach dem selben Prinzip vorgegangen.

5.3 Use Case 3: Zeichenfläche (Mikka Jenne)

5.3.1 Frontend

Mit der Fertigstellung der graphischen Benutzeroberfläche des Editors konnte sich an die Arbeit der noch ausstehenden Use Cases gemacht werden. Darunter fällt z.B. der obig aufgeführt UC 2, dieser beinhaltet alle relevanten Komponenten die zum Zeichnen benötigt werden, bspw. Symbol und Funktion von Lichtschaltern oder Steckdosen o.ä. und der Use Case 3, welcher in diesem Kapitel aufgeführt ist.
Explizit handelt es sich bei diesem Anwendungsfall um das individuelle Zeichnen in einem vorgegebenen Bereich. In diesem Fall ist dieser begrenzte Bereich ein *Canvas*. Dieser ist in Abbildung 5.4, der Editor-Ansicht, im linken unteren Eck zu entnehmen. Eine große markante Fläche, die am meisten Platz beansprucht. Ein Canvas, dt. Leinwand, repräsentiert in WPF eine Fläche die explizit als eine Leinwand zum Zeichnen vorgegeben ist und viele Funktionen dazu bereitstellt. Der Code-Ausschnitt 5.13 zeigt das Canvas, welches als fester Parameter in der XAML vorhanden ist. Alle weiteren Änderungen auf dem Canvas werden dynamisch erstellt und nachgeladen. Die Funktionsweise und was es mit dem *Interaction.Triggers* auf sich hat wird im Backend genauestens erläutert.

```
<UserControl>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Border Background="GhostWhite"
          BorderBrush="Gainsboro" BorderThickness="1">
            <Canvas Grid.Row="1" Background="White">
                <i:Interaction.Triggers>
                    <i:EventTrigger EventName="MouseLeftButtonUp">
                        <command:EventToCommand Command="{Binding
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣GetKooradinatesClickOnCanvas}"
                          PassEventArgsToCommand="True"/>
                    </i:EventTrigger>
                </i:Interaction.Triggers>
            </Canvas>
        </Border>
    </Grid>
</UserControl>
```

Code-Beispiel 5.24: UserControl-Referenz Code-Ausschnitt

5.3.2 Backend

Der *Interaction.Triggers* ist die Schnittstelle zu der Funktion im Backend, die die Methode zum Zeichnen beinhaltet und aufruft. Mit dem *EventToCommand* wird eine Funktion gebindet die in der *ViewModel-Class* implementiert ist. Diese Methode ist als *ICommand*-Methode initialisiert und implementiert zu Anfang das Event mit dem der *MouseClick* aufgefangen wird. Dieses MouseClick-Event wird dann auf einen Command gecastet und so an das Backend übergeben.

```
1 public ICommand GetKooradinatesClickOnCanvas
2 {
3     get { return new RelayCommand<EventArgs>(MouseClick_ToDraw); }
4 }
```

Code-Beispiel 5.25: ICommand-OnClick Code-Ausschnitt

Bei einem MouseClick wird wie oben erwähnt die Funktion *GetKoordinatesClickOnCanvas* aufgerufen, die einen Command returnt, welcher ebenso eine Methode beinhaltet in der final der Befehl zum Zeichnen ausgeführt werden soll.

```
1  private void MouseClick_ToDraw(EventArgs args)
2  {
3      MouseEventArgs e = (MouseEventArgs) args;
4
5      if(e.LeftButton == MouseButtonState.Released) //
6      {
7          if (TestArrayPosition(mousePosition) == true)
8          {
9              var position = new Point();
10             position = e.GetPosition(e.Device.Target);
11             mousePosition[0] = position;
12         }
13     }
14 }
```

Code-Beispiel 5.26: MouseClick-Funktion Code-Ausschnitt

In diesem Ausschnitt wird das Event zu einem *MouseEventArgs* gecastet, um bei dem MouseClick die Koordinaten in dem Canvas bestimmen zu können. Dabei wird die jeweilige Position gespeichert. Nach erfolgreicher Überprüfung und Speicherung des zweiten Punkts werden die gespeicherten Koordinaten einer Methode des Namespaces System.Drawing übergeben. Die Funktion des System.Drawings nimmt die Koordinaten und wandelt diese zu Punkten um, um sie anschließend der *DrawLine()-Methode* zu übergeben. Diese Methode ist ebenso Bestandteil des Namespaces *System.Drawing*. Dieser Ausdruck bekommt alle relevanten Parameter übergeben, um schlussendlich eine Linie zeichnen zu können.

Nach dem die Applikation gestartet wurde und das Editor-Fenster erscheint, kann der Benutzer prinzipiell anfangen Linien in das Canvas zu zeichnen. Mit diesen Linien werden Grundriss oder Leitungen definiert.

Bei der Aktion des ersten MouseClick werden die ersten Koordinaten abgerufen und in ein Array gespeichert. Bei dem zweiten Klick wird überprüft, ob in dem Array schon Werte vorhanden sind und ob die Koordinaten des zweiten Klicks eventuell identisch zu den Koordinaten des ersten Klicks. Falls dieser Fall nicht zutrifft, werden die im Array gespeicherten Koordinaten dem *Line*-Parameter übergeben, damit ein *Line*-Objekt erstellt werden kann. Dieser Parameter wird generiert und beinhaltet auch alle zutreffenden Werte, nur kann die Linie aktuell nicht auf der Benutzeroberfläche angezeigt werden. Prinzipiell sollte der ICommand-Befehl die grafische Benutzeroberfläche bei einem *OnPropertyChanged*-Aufruf den jeweiligen

Wert ändern. Aus derzeit unerklärlichen Gründen wird die generierte Linie nicht auf das Canvas in der Benutzeroberfläche übertragen. Der Compiler kompiliert alles nach Plan und zeigt keine Exception an, bzw. terminiert die Applikation nicht durch einen Error.

Aus zeitlichen Gründen war es im Rahmen dieser Arbeit nicht möglich diesen Fehler genauer zu analysieren und die fehlerhafte Darstellung zu berichtigen. Eine Erklärung für dieses Verhalten der Applikation, bzw. des Compilers ist nicht vorhanden und müsste erst mit einer detaillierten Recherche analysiert und berichtigt werden.

Kapitel 6

Fazit und Ausblick

6.1 Fazit

Das Ziel dieser Arbeit war die Grundlage und Architekturkonzeption einer Software zur Erstellung von Schaltplänen zu erarbeiten. Dafür wurden in den ersten Schritten die genauen Anforderungen an das System festgelegt. Darauf basierend wurden mit C#, .NET und WPF die Technologien ausgewählt. Um eine Modularität zu gewährleisten, damit auch künfigt an der Software weiterentwickelt werden kann, musste ein geeignetes Entwurfsmuster gefunden werden, nach dem die Architektur aufgebaut wird. Mit MVVM wurde ein passendes Muster, dass vorallem für WPF und C# besonders geeignet ist gefunden. Bevor es in die konkrete Umsetzung ging, wurde die Architektur anhand von Use Cases und Klassendiagrammen modeliert. Anhand dieser Überlegungen konnte dann die Architektur implementiert werden. Eine große Rolle für die Implementierung auf Basis des MVVM Musters hat dabei das MVVM Light Framework gespielt.

Aufgrund der hohen Entkopplung zwischen den einzelnen Modulen und speziell der Benutzeroberfläche und Logik, wird eine einfachere Wartbarkeit sowie Erweiterbarkeit des Projekts gewährleistet. Aber auch die Möglichkeit die einzelnen Bestandteile zu testen, ist durch die hohe Entkopplung möglich. Neben den Vorteilen bringt die Moldularität auch Nachteile bei der Entwicklung. So traten immer wieder Schwierigkeiten und komplexe Probleme auf, insbesondere in der Verbindung zwischen der Benutzeroberfläche und der Logik. In einzelnen Fällen mussten Synchronisierungen zwischen Benutzeroberfläche und Logik durch Hilfsklassen unterstützt werden. Auch für die Anzeige bestimmter Elemente mussten Converter Klassen geschrieben werden. Änderungen in der Projektstruktur haben teilweise zu komplexen Verschiebungen, vorallem in den Namensräumen innherlab der Visual Studio IDE geführt. Daher wird nicht empfohlen die Struktur in der künftigen Entwicklung groß zu ändern. In dieser Arbeit wurde eine fundierte Grundlage geschaffen, auf der eine Weiterentwicklung stattfinden kann.

Im gesamten betrachtet bietet das Ergebnis dieser Arbeit einen klaren und leicht verständlichen Systemaufbau und die Grundlage für Weiterentwicklungen. Hält man sich an das vorgegebene Muster sowie die bisherig implementierte Struktur des Projekts, gelingt das Hinzufügen weiterer Funktionen einfach und erfolgreich.

6.2 Ausblick

Bisher bietet die Software die Möglichkeit neue Projekte anzulegen. Dabei erzeugt das System automatisch eine Projektdatei. Nachdem ein Projekt angelegt wurde, öffnet sich der Editor. Er stellt die Eigentliche Funktionalität des Programms dar. Der Editor hat die grafische Struktur für alle nötigen Funktionen implementiert. Eine Struktur der Komponenten für die Toolbox der Elektro-Komponenten liegt ebenfalls vor und kann beliebig erweitert werden. Auch die Zeichenfläche ist verfügbar. Sie erkennt die Mausposition und registriert einen Klick. Dabei wird die Position gespeichert und anhand der zwei Positionen, eine Linie gezeichnet. Die Linie existiert im Programmcode, wird allerdings bislang noch nicht an der Benutzeroberfläche angezeigt.
In weiteren Entwicklungen können die Funktionalitäten erweitert werden. Auf Basis des Systems kann ein Speicherverfahren implementiert werden, dass die Zeichnungen innerhalb der vorhandenen Text-Datei abspeichert. Die Möglichkeit Schaltpläne zu zeichnen kann ebenfalls Funktional erweitert werden. So ist das langfristige Ziel vollständige 2D Schaltpläne zu entwerfen, die in einem fertigen Format (z.B. PDF) exportiert werden können. Eine weitere Konzeption sieht vor, die Schaltpläne in einer 3D Sicht abzubilden um festzustellen, wo genau innerhalb eines Gebäudes bestimmte Komponenten angebracht werden können. Abschließend lässt sich sagen, dass die Grundlage und vorallem eine grundlegene Architektur innerhalb dieser Arbeit erstellt und entwickelt wurden. Diese können als ausreichende Basis für zukünftige Entwicklungen genutzt werden um ein vollständiges, für den Anwender bestimmtes Programm zu erstellen.

Literatur

Collection, Microsoft [Apr. 2020]. *Website Microsoft Nampespace Collection*. https://docs.microsoft.com/de-de/dotnet/api/system.collections?view=netcore-3.1 [siehe S. 13].

Degruyter [März 2020]. *Website Degruyter*. https://www.degruyter.com/downloadpdf/books/9783110443882/9783110443882-005/9783110443882-005.pdf[3] [siehe S. 20].

Drawing, Microsoft [Apr. 2020]. *Website Microsoft Nampespace Drawing*. https://docs.microsoft.com/de-de/dotnet/api/system.drawing?view=netcore-3.1 [siehe S. 13].

entwickler [Sep. 2010]. *Website entwickler.de*. https://entwickler.de/online/tipps-und-tricks-zum-mvvm-pattern-153374.html [siehe S. 15].

Gruenderszene [März 2020]. *Website Gruenderszene*. https://www.gruenderszene.de/lexikon/begriffe/user-experience?interstitial [siehe S. 20].

Microsoft [Apr. 2020]. *Website Microsoft Docs*. https://docs.microsoft.com/de-de/dotnet/framework/wpf/getting-started/introduction-to-wpf-in-vs [siehe S. 10].

MVVMLight [Apr. 2020]. *Website des MVVM Light Frameworks*. http://mvvmlight.net [siehe S. 15].

Müller, Maurice [März 2020]. *Foliensatz Advanced Software Engineering Usability und UX*. https://drive.google.com/drive/folders/1a0zCxcIvym1Z7twfZTUCSwgaN49Plho2 [siehe S. 20].

Wikipedia [Apr. 2020]. *Website Gruenderszene*. https://de.wikipedia.org/wiki/Windows_Presentation_Foundation [siehe S. 11].

Window, Microsoft [Apr. 2020]. *Website Microsoft Nampespace Window*. https://docs.microsoft.com/de-de/dotnet/api/system.windows?view=netcore-3.1 [siehe S. 13].

WPF-Tutorial [Apr. 2020]. *Website WPF-Tutorial*. https://www.wpf-tutorial.com/de/1/uber-wpf/was-ist-wpf/ [siehe S. 12].