



Diplomarbeit

Höhere Technische Bundeslehranstalt Leonding
Abteilung für Informatik

AWO - Administration and Website for Opticians

Eingereicht von: **Eva Pürmayr, 5AHIF**
Danijal Orascanin, 5AHIF
Datum: **April 4, 2018**
Betreuer: **Michael Bucek**
Projektpartner: **Augenoptik Aigner**

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Leonding, April 4, 2018

Eva Pürmayr, Danijal Orascanin

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Gedanken, die aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Leonding, am 4. April 2018

Eva Pürmayr, Danijal Orascanin

Zusammenfassung

Abstract

Inhaltsverzeichnis

1	Einleitung	3
1.1	Ist-Situation	3
1.2	Zielsetzung	3
1.3	Aufgabenstellung	3
2	Verwendete Technologien	4
2.1	Entity Framework	4
2.1.1	Linq	6
2.2	UnitOfWork-Pattern	6
2.3	WPF	8
2.4	MVVM	9
2.5	MVVM-Light	10
2.6	Microsoft Office Interop	12
2.7	WPF-Toolkit	13
2.8	MessageBird	13
3	Implementierung	15
3.1	Administration	15
3.1.1	Kundenverwaltung	15
3.1.2	Auftragsverwaltung	18
3.1.3	Lieferantenverwaltung	22
3.1.4	Verwaltung der lagernden Brillenfassungen	22
3.1.5	E-Mail und SMS	23
3.1.6	Statistiken	27
3.1.7	Sonstiges	28
3.1.8	Filtern und Sortieren	31
3.2	Datenmodell	39
3.3	Projektarchitektur	41
4	Selbstevaluation	44
4.1	Probleme bei der Entwicklung	44
4.1.1	Event-Handling mit MVVM-Light	44
4.1.2	UnitOfWork-Instanzen	45

Kapitel 1

Einleitung

1.1 Ist-Situation

1.2 Zielsetzung

1.3 Aufgabenstellung

Kapitel 2

Verwendete Technologien

2.1 Entity Framework

Zum Nutzen einer Datenbank in Programmen gibt es das von Microsoft entwickelte Entity Framework. Dabei handelt es sich um ein Framework zur Erstellung von objektrelationalen Abbildungen auf .NET-Objektstrukturen. Das wird auch als ORM (object relational mapping) bezeichnet. In anderen Worten ermöglicht es Programmieren in der Applikation mit Objekten zu arbeiten, anstatt sich auf die wirkliche Datenbank mit ihren Tabellen und den Zugriff darauf konzentrieren zu müssen. Der Grund der Notwendigkeit eines solchen Frameworks ist, dass objektorientierte Programmiersprachen wie .NET Daten in Objekten speichern und relationale Datenbanken Daten in Tabellen ablegen. Diese beiden Arten der Speicherung sind grundlegend verschieden, weshalb ein Framework benötigt wird um diese miteinander verwenden zu können.

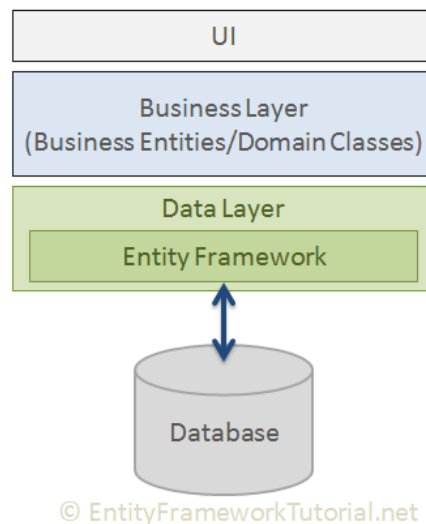


Abbildung 2.1: Einsatz des Entity Frameworks in einer Applikation

Wie man in der obenstehenden Grafik erkennen kann, verbindet das Entity Framework die Objekte im Programm mit der Datenbank. Dabei kann es Daten von Properties in Objekten in die Datenbank speichern und umgekehrt Daten von der Datenbank in Objekte verwandeln.

Um eine Datenbank zu verwenden, gibt es drei Möglichkeiten:

- Code First: Dabei werden die benötigten Klassen im Programm zuerst definiert und darauf basierend wird vom DbContext die Datenbank erzeugt. Diese Methode wurde gewählt um die Datenbank dieser Arbeit zu erstellen.
- Model First: Im Gegensatz zu Code First werden hier zuerst die Entitäten der Datenbank mit Hilfe eines grafischen Tools modelliert und darauf basierend wird ein Datenbankschema erstellt.
- Database First: Es wird eine bestehende Datenbank verwendet.

Zum Verwenden der Datenbank wird im Programm eine Klasse erzeugt, die von der Klasse "DbContext" abgeleitet wird. Diese Klasse enthält DbSet aller Entitätsklassen und gibt mittels des Connectionstrings an, welche Datenbankverbindung verwendet werden soll. Das ist sehr nützlich, weil damit die Datenbank sehr einfach ausgetauscht werden kann. Ein DbSet ist eine Klasse, die die entsprechenden Methoden für Entitätstypen anbietet (Zum Beispiel 'Add' oder 'Remove').

Beispielsweise werden so alle Kunden der Datenbank verwaltet:

```
public DbSet<Customer> Customers { get; set; }
```

Mit dieser Property kann nun im Programm gearbeitet werden. Das heißt, dass alles was in diesem DbSet geändert wird, mit der Methode "SaveChanges" (aufgerufen vom DbContext) auch in die entsprechende Tabelle in der Datenbank übertragen wird. Wenn zum Beispiel ein Objekt Customer an das DbSet Customers angehängt wird, und dann "SaveChanges" aufgerufen wird, wird auch in der Datenbank ein neuer Datensatz erstellt. Somit lässt das Entity Framework zu, dass Insert-, Update- oder Delete-Statements auf der Datenbank ausgeführt werden. Zum Holen von Daten aus der Datenbank kann der Programmierer die Abfragesprache Linq benutzen. Dabei übersetzt das Entity Framework das jeweilige Linq-Statement in die Abfragesprache der verwendeten Datenbank (z.B. SQL). Über sogenannte "DataAnnotations" können mittels des Entity Frameworks auch Bedingungen festgelegt werden, welche gewisse Spalten von Entitäten in der Datenbank erfüllen müssen. Zum Beispiel wird mittels der Data-Annotation "MaxLength" festgelegt, dass der Name eines Ortes nicht länger als 100 Zeichen sein darf:

```
[MaxLength(100)]  
public string TownName { get; set; }
```

Quellen: [Wik17a], [Tut18], [Wik16], [Mic18a], [Mic18b]

2.1.1 Linq

Linq steht abgekürzt für Language Integrated Query und ermöglicht den Zugriff auf Daten aus einem Programm. Mit dieser Abfragesprache kann der Benutzer auf lokale Listen im Programm zugreifen, auf Daten aus der Datenbank, auf XML-Inhalte und vieles mehr. Im Falle dieser Arbeit wird aber nur Linq to Objects und Linq to Entities verwendet. Linq to Entities ermöglicht dem Programmierer im Code direkt Abfragen an das konzeptionelle Modell vom Entity Framework zu stellen. Genauer gesagt werden diese Abfragen dann in Befehlsstrukturen umgewandelt und dann gegen den Objektkontext ausgeführt.

Linq hat die Eigenschaft, dass die gegebenen Ausdrücke nicht bei ihrer Definition ausgeführt werden, sondern erst wenn der Wert tatsächlich gebraucht wird (Lazy Evaluation). Das hat zum Vorteil, dass Abfragen öfter verwendet werden können. Falls der Benutzer das nicht möchte, muss er vortäuschen, die Ergebnismenge sofort zu benötigen (zum Beispiel kann nach der Query ein `.ToList()` angehängt werden).

Mit der nachfolgenden Codezeile werden beispielsweise alle Kontaktlinsenaufräge gewählt, die schon bezahlt wurden.

```
List<Order> paidContactLenses = this.ContactLenses.Where(c => c.PaymentState == "Bezahlt").ToList();
```

Ein anderes Beispiel wäre alle Kunden zu zählen, deren Vorname mit 'E' beginnt:

```
int customersStartingFirstNameWithE = this.Customers.Count(c => c.FirstName.StartsWith("E"));
```

Quellen: [Wik18], [Mic18c]

2.2 UnitOfWork-Pattern

Das UnitOfWork-Pattern ist eines von vielen Design Patterns in .NET. Design Patterns sind allgemeine Lösungen für Software Design Probleme, die immer wieder vorkommen. Das UnitOfWork-Pattern beschreibt einen Weg der Projektarchitektur um mit Datenbanken arbeiten zu können. Es verwaltet Transaktionen, führt Updates geregelt durch und schafft damit Concurrency-Probleme aus der Welt. Dadurch arbeitet man im Code nicht direkt mit den DbSets sondern mit der Klasse UnitOfWork.

Genauer gesagt, muss der Programmierer zuerst das Repository-Pattern implementieren, um das UnitOfWork-Pattern umzusetzen zu können. Dabei geht es nur darum, für jede Entität eine Klasse zu erschaffen (Repository), die alle Operationen für diese Entität beinhaltet. In einem Repository für die Klasse Kunde sollten zum Beispiel die CRUD Methoden (Create, Read, Update, Delete) enthalten sein. In dieser Arbeit wurde das Pattern umgesetzt, indem eine generische Klasse "GenericRepository" für alle Entitäten gestaltet wurde (siehe Data-Access-Layer).

Mit dem Repository-Pattern alleine (ohne UnitOfWork-Pattern), enthält jedes Repository einen eigenen DbContext, welche nicht aufeinander abgestimmt sind. Das würde

allerdings zu Problemen führen, besonders wenn zwei verschiedene Repositories eingesetzt werden und beide gleichzeitig Transaktionen abschließen. Jedes Repository hätte dann seine eigene Version von eventuell geänderten Datensätzen, sich vielleicht unterscheiden würden. Das würde letztendlich zur Datenbankinkonsistenz führen.

Um dieses Problem zu vermeiden, wird das UnitOfWork-Pattern eingesetzt. Dabei wird eine Klasse "UnitOfWork" erstellt, die eine Instanz von allen Repositories enthält und einen zentralen DbContext, der an die einzelnen Repositories weitergegeben wird. Damit können nun Datenbankänderungen, in denen mehrere Repositories benötigt werden, gesammelt in einer Transaktion auf einem zentralen DbContext ausgeführt werden.

Data-Access-Layer: In dieser Arbeit wurde das Repository Pattern und das UnitOfWork-Pattern mit folgenden Klassen implementiert:

- EntityObject: Dies ist eine Klasse, von der später alle Entitäten ableiten. Sie gibt den Entitäten eine Id und einen Timestamp, um später Concurrency-Probleme zu lösen.
- IGenericRepository: Ein Interface für alle Repositories, welches die Standardmethoden wie Get, Insert oder Delete vorschreibt.
- IUnitOfWork: Ein Interface, welches die Definitionen für alle IGenericRepositories, die Save-Methode sowie andere Methodenköpfe, die später selbst implementiert werden, enthält.
- GenericRepository: Eine Klasse, die für jede Entität erstellt wird und die von IGenericRepository ableitet. Sie enthält den Context sowie das DbSet der gewünschten Entität. Zusätzlich implementiert sie alle Standardmethoden (Get, GetById, Insert, Update, Delete, Count...).
- UnitOfWork: Eine Klasse, die von IUnitOfWork ableitet und alle Methoden implementiert. Mit dieser Klasse wird später im Programm gearbeitet.

Zum Beispiel wird mit diesem Befehl ein Kunde mittels der Id gesucht. Das globale, private Feld "uow" wird zuvor mittels Dependency Injection im Konstruktor initialisiert.

```
Customer c = uow.CustomerRepository.GetById(id);
```

Es wird also zuerst auf die UnitOfWork zugegriffen und von dort aus auf das spezielle Repository. CustomerRepository ist vom Typ GenericRepository <Customer> und damit kann man auf die im GenericRepository definierten Methoden (hier GetById) zugreifen. Ein anderes Beispiel wäre das Einfügen eines neuen Datensatzes in die Datenbank. Dazu muss unbedingt die Save-Methode danach aufgerufen werden, denn sonst werden die Änderungen nicht in die Datenbank übertragen.

```
uow.CustomerRepository.Insert(this.Customer);  
uow.Save();
```

Die Klasse "UnitOfWork" beinhaltet auch eine Methode namens "Dispose" welche auf jeden Fall aufgerufen werden muss um den DbContext zu schließen.

Quellen: [Cod18], [dof18], [csh18]

2.3 WPF

WPF (Windows Presentation Foundation) ist eine von Microsoft entwickelte Klassenbibliothek zur Erstellung von grafischen Oberflächen. Mit WPF werden häufig Desktopanwendungen erstellt, allerdings gibt es auch die Möglichkeit 3D-Grafiken, Dokumente oder Videos zu erstellen. Als Vorgängerversion kann man Windows Forms bezeichnen, denn WPF beinhaltet weitaus mehr Möglichkeiten des Designs als Windows Forms. Um eine WPF-Anwendung erstellen zu können, benötigt man die Definitionssprache XAML. Dies ist abgekürzt und steht für Extensible Application Markup Language. Diese Sprache basiert auf XML und beinhaltet zusätzlich WPF-spezifische Elemente.

Beispiel eines XAML-Codes und dessen Erscheinen:

```
<Window x:Class="OpticianMgr.Wpf.Pages.AddCountryWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:OpticianMgr.Wpf.Pages"
        mc:Ignorable="d"
        Title="Neues Land" Height="179.589" Width="576.437">
    <StackPanel Style="{StaticResource StackPanelBackground}">
        <Label Style="{StaticResource HeadingStyle}">Neues Land</Label>
    </StackPanel>
</Window>
```

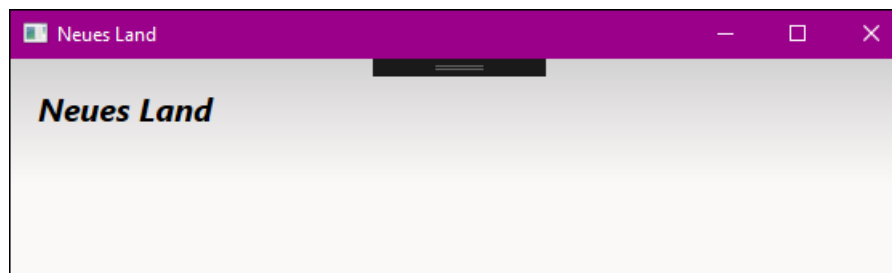


Abbildung 2.2: Einfaches WPF-Fenster

Dennoch hat WPF auch manche Nachteile:

- WPF benötigt ein jüngeres Betriebssystem als Windows XP.

- Es kommt zu Leistungsproblemen, wenn mehrere Fenster im Einsatz sind und außerdem hat WPF einen hohen RAM-Bedarf.

Quellen: [IV18], [Sch18]

2.4 MVVM

MVVM ist eine Abkürzung und steht für Model-View-ViewModel. Dies ist ein Entwurfsmuster, wie man Projekte designen kann und dient zur Trennung der Logik und der Darstellung der Benutzerschnittstelle. Es ist speziell geeignet für WPF.

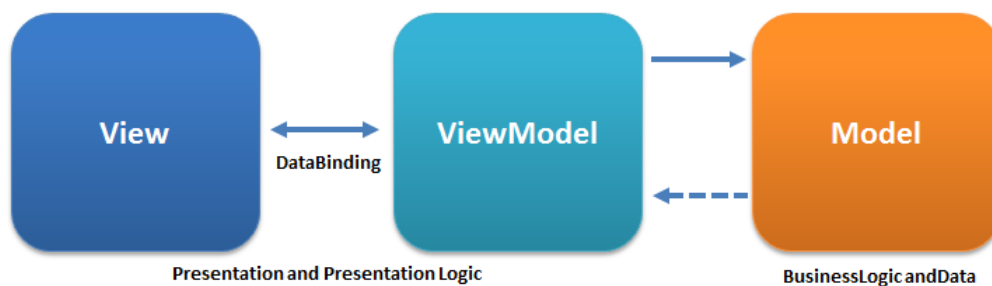


Abbildung 2.3: MVVM-Konzept

In der obenstehenden Grafik kann man leicht erkennen, wie MVVM funktioniert. Die View kommuniziert ausschließlich mit dem ViewModel und zwar über DataBindings. Das ViewModel beinhaltet die Logik und verändert gegebenenfalls das Model (Datenbank).

DataBindings von dem ViewModel zur View funktionieren mit Hilfe des Events "PropertyChangedEventHandler".

```
public event PropertyChangedEventHandler PropertyChanged;
```

Mit der Methode "Invoke" wird das Event ausgelöst. Es benachrichtigt die View, dass sich der Wert der Property mit dem Namen "propertyName" geändert hat. Im ersten Parameter wird der Sender des Events übermittelt. In diesem Fall ist der Sender das ViewModel (this), in dem das Event ausgelöst wird.

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
```

Im Gegenzug kann die View folgendermaßen auf eine Property im ViewModel binden:

```
<ListView ItemsSource="{Binding CustomersView, Mode=TwoWay,  
    UpdateSourceTrigger=PropertyChanged}">
```

Vorteile von MVVM:

- Logik sowie Darstellung können unabhängig voneinander bearbeitet werden. Dadurch kann die Darstellung von Designern und die Logik von Entwicklern erstellt werden.
- Durch die Trennung ergibt sich eine bessere Testbarkeit der Logik.

Quellen: [Wik17b]

2.5 MVVM-Light

MVVM-Light ist ein Framework, welches dazu dient, den Aufwand der Implementierung des MVVM-Musters zu verringern. Neben MVVM-Light gibt es auch noch andere solcher Frameworks, wie zum Beispiel Prism.

Zur Verwendung müssen die Packages MVVMLight (Version 5.3.0) und MVVMLightLibs (Version 5.3.0) über den NuGet-Package-Manager installiert werden.

Beim Einbinden von MVVM-Light werden zwei wesentliche Klassen erstellt

- MainViewModel: Das ViewModel der Hauptseite, abgeleitet von der Klasse "ViewModelBase".
- ViewModelLocator: Beinhaltet statische Referenzen für alle anderen ViewModels. Außerdem bietet diese Klasse einen einfachen IOC-Container.

Alle ViewModels, die später erstellt werden, sollten von der Klasse "ViewModelBase" abgeleitet werden. Damit hat der Programmierer unter anderem die Möglichkeit die Methode "RaisePropertyChanged" zu verwenden. Diese ermöglicht dem ViewModel die View zu benachrichtigen, dass sich Werte von Properties verändert haben und somit kann die View die entsprechenden Daten aktualisieren.

Hier werden die Werte der Kundenübersicht vom ViewModel aus aktualisiert.

```
this.RaisePropertyChanged(() => this.CustomersView);
```

MVVM-Light bietet die vorgefertigte Klasse "RelayCommand", welche die Implementierung des Interfaces "ICommand" für MVVM-Light darstellt. Der Konstruktor der Klasse hat zwei Parameter. Der erste beschreibt die Aktivität, die ausgeführt werden soll, sobald das RelayCommand aufgerufen wird (Lambda-Expression oder Delegate für Methode). Der zweite Parameter ist optional und erwartet ein Delegate für eine Methode oder eine Lambda-Expression, die ein "bool" zurückgibt. Dieser beschreibt, ob das RelayCommand zur Zeit ausgeführt werden darf oder nicht.

Im nachfolgenden Beispiel wird ein RelayCommand aufgerufen, sobald ein Button gedrückt wird.

In der View wird der Name des RelayCommand im Button angegeben.

```
<Button Command="{Binding DeleteFilter}">Filter zuruecksetzen</Button>
```

Im ViewModel wird das gewünschte RelayCommand so initialisiert:

```
DeleteFilter = new RelayCommand(DeleteF);
```

DeleteF ist dabei die Methode, die ausgeführt wird, sobald der Button gedrückt wird. Ein weiteres Feature von MVVM-Light heißt "EventToCommand". Dabei können alle beliebigen Events aus der View an das ViewModel in RelayCommand weitergegeben werden und dort bearbeitet werden. Zusätzlich können auch die Parameter des Events in das ViewModel übertragen werden. Das geschieht, indem der Programmierer "PassEventArgsToCommand" auf "true" setzt. Zur Implementierung müssen in der View zwei Namespaces deklariert werden:

```
xmlns:i="clr-namespace:System.Windows.Interactivity;  
assembly=System.Windows.Interactivity"  
xmlns:cmd="clr-namespace:GalaSoft.MvvmLight.Command;  
assembly=GalaSoft.MvvmLight.Platform"
```

Im nachfolgenden Codeabschnitt wird das Event "Loaded" vom RelayCommand "Initialized" abonniert.

```
<i:Interaction.Triggers>  
  <i:EventTrigger EventName="Loaded">  
    <cmd:EventToCommand Command="{Binding Initialized}"  
      PassEventArgsToCommand="True"></cmd:EventToCommand>  
  </i:EventTrigger>  
</i:Interaction.Triggers>
```

Im ViewModel kann dann auf das Event und dessen Parameter reagiert werden:

```
public ICommand Initialized { get; set; }  
  
public CustomerViewModel()  
{  
    Initialized = new RelayCommand<RoutedEventArgs>(Init);  
}  
private void Init(RoutedEventArgs p)  
{  
    //do something  
}
```

Eine weiteres Feature von MVVM-Light ist der Messenger. Diese Klasse erlaubt den Austausch von Nachrichten zwischen zwei ViewModels. Die ViewModels müssen dabei keine spezielle Verbindung zueinander haben.

Quellen: [dot18a], [dot18b], [Mic18d]

2.6 Microsoft Office Interop

Um aus einem Programm Word-Dokumente zu erstellen, gibt es das Assembly Microsoft.Office.Interop.Word auf welches eine Referenz hinzugefügt wurde. Damit kann der Benutzer im Programm Word-Dokumente bearbeiten, oder Informationen aus diesen herauslesen. Zur Nutzung benötigt der Computer allerdings eine gültige Microsoft Office Lizenz. Außerdem lädt Interop das Word-Dokument im Hintergrund, wenn es bearbeitet wird, was den ganzen Vorgang etwas langsam macht. Dennoch bietet sich der Gebrauch von Interop an, weil es die einfachste Variante ist, auf Word-Dokumente zuzugreifen. Im folgenden Abschnitt wird erklärt wie aus einer Vorlage ein individuelles Word-Dokument erstellt werden kann.

```
Application wordApp = new Application();
Document wordDoc = new Document();

Object oMissing = System.Reflection.Missing.Value;
wordDoc = wordApp.Documents.Add(ref oTemplatePath, ref oMissing, ref oMissing,
    ref oMissing);
foreach (Field myMergeField in wordDoc.Fields)
{
    Range rngFieldCode = myMergeField.Code;
    String fieldText = rngFieldCode.Text;

    //only mergefields should be edited
    if (fieldText.StartsWith(" MERGEFIELD"))
    {
        myMergeField.Select();
        wordApp.Selection.TypeText("Test");
    }
}
wordDoc.SaveAs(completePath);
wordDoc.Close();
wordApp.Quit();
```

Über den String "oTemplatePath" wird der Pfad des gewünschten Templates übergeben. Danach werden alle MergeFields der Vorlage (diese kann man beim Erstellen der Vorlage einfügen: Einfügen -> Schnellbaustein -> Mergefield) mit der Zeichenkette "Test" ersetzt. Im Endeffekt simuliert Interop einen Klick auf das Feld und mit der Methode TypeText("Test") wird der Text eingefügt. Zum Abschluss wird das Dokument unter einem angegeben Pfad abgespeichert (completePath) und die geöffnete Vorlage sowie die Applikation geschlossen.

Quellen: [Gem18]

2.7 WPF-Toolkit

Zur grafischen Darstellung der Statistiken wurde ebenfalls ein eigenes Assembly installiert: `System.Windows.Controls.DataVisualization.Toolkit` (Version 4.0.0). Dieses Assembly kann einfach über den NuGetPackage-Manager heruntergeladen werden. Es ermöglicht die Veranschaulichung verschiedener Diagramme, wie zum Beispiel Linien-, Torten oder Balkendiagrammen. Zur Verwendung muss der Benutzer den Namespace in der View definieren.

```
xmlns:toolkitCharting="clr-namespace:System.Windows.Controls.DataVisualization.Charting;assembly=System.Windows.Controls.DataVisualization.Toolkit"
```

Im folgenden Code wird ein Liniendiagramm mit einer Linie erstellt:

```
<toolkitCharting:Chart Title="{Binding Title}">
    <toolkitCharting:LineSeries Title="{Binding LineTitle}"
        DependentValueBinding="{Binding Value}"
        IndependentValueBinding="{Binding Key}" ItemsSource="{Binding
            DataValues}"/>
</toolkitCharting:Chart>
```

In der Property "Title" wird der Titel des Diagramms übergeben und "LineTitle" beschriftet die Linie mit dem gewünschten Text. "DataValues" (Typ "ObservableCollection<KeyValuePair<string, int>>") beinhaltet die Daten, welche im Diagramm dargestellt werden sollen. "DependentValueBinding="{Binding Value}" beschreibt, dass die abhängigen Werte des Diagramms jeweils vom "Value" bezogen werden. In "DataValues" ist das der int.

Quellen: [sha17]

2.8 MessageBird

MessageBird ist ein Unternehmen, welches einen Online-Dienst anbietet mit dem Unternehmen oder auch einzelne Personen, kostenpflichtig SMS aus einem Programm versenden können. Dazu meldet sich der Benutzer auf der Website "<https://www.messagebird.com>" an und sucht sich das passende Angebot. Danach kann man sein Guthaben aufladen und bekommt im Gegenzug einen AccessKey, über den man Nachrichten versenden kann. Dabei kann jeder Benutzer von MessageBird mehrere AccessKeys haben, beispielsweise einen für Test-Nachrichten, die nicht wirklich versendet werden, oder einen AccessKey, mit dem dann echte SMS versendet werden. Für Nachrichten, die nach Österreich gesendet werden, muss der Benutzer derzeit 4,6 Cent (Stand März 2018) bezahlen. Sobald die SMS versendet wurde, wird der Betrag automatisch vom Guthaben des Benutzers abgezogen. Wenn das Guthaben aufgebraucht ist, versendet MessageBird keine SMS mehr und der Benutzer kann sein Guthaben gegebenenfalls wieder aufladen. Auf der Website hat man einen Überblick über die versendeten SMS, verschiedene angelegte Nummern und vieles mehr.

Über den NuGet-Manager kann man das Package "MessageBird" installieren und im Code dann so verwenden:

```
IProxyConfigurationInjector proxyConfigurationInjector = null; // for no web
    proxies, or web proxies not requiring authentication

Client client = Client.CreateDefault(AccessKey, proxyConfigurationInjector);

MessageBird.Objects.Message message = client.SendMessage("OptikAigner",
    this.Message, new[] { Convert.ToInt64(this.To) });
```

"AccessKey" im zweiten Befehl ist der String, den man von der Website bekommt, über den abgerechnet wird. "OptikAigner" wird als Sendernamen angegeben. Der zweite Parameter des letzten Befehls, gibt den Text der Nachricht an. Der letzte Parameter ist ein "long"-Array mit einem Wert, nämlich der Telefonnummer, an die die SMS versendet werden soll.

Andere Möglichkeiten

Im Vergleich zum Versand von E-Mails, gibt es keine Möglichkeit kostenfrei und ohne Anbieter SMS zu versenden. Als Alternative hätte sich das Unternehmen "Twilio" angeboten, allerdings hätte dort eine SMS ca. 9 Cent gekostet. Aus Kostengründen und aus Benutzerfreundlichkeit ist die Wahl des Anbieters auf MessageBird gefallen.

Quellen: [Mes18b], [Mes18a], [twi18]

Kapitel 3

Implementierung

3.1 Administration

3.1.1 Kundenverwaltung

Um dem Benutzer eine kompakte Übersicht über seine Kunden zu geben, gibt es die Kundenverwaltung, bei der alle Kunden in einer Liste dargestellt werden. Angezeigt wird die Id, Titel, Vorname, Nachname, Straße, PLZ, Ort, Telefon-1 und das Land des Kunden. Diese Liste kann man filtern und sortieren (siehe Kapitel 3.1.8). Zusätzlich zu dem normalen Filter, kann man bei der Kundenverwaltung ebenso gelöschte Kunden ein- oder ausblenden. Der Grund dafür wird später noch näher erklärt.

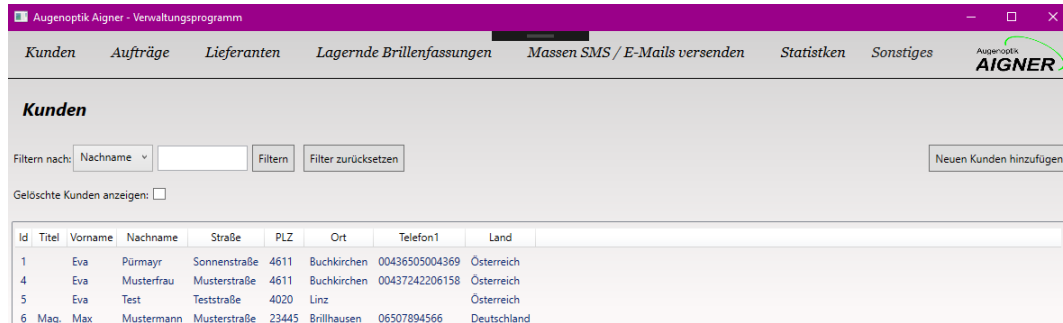


Abbildung 3.1: Screenshot der Kundenverwaltung

Beim Klick des Buttons "Neuen Kunden hinzufügen" erscheint ein neues Fenster, welches einen neuen Kunden erstellt. Hier kann der Benutzer weitere Daten eingeben, wie beispielsweise Hobbies, Job oder den Geburtstag. Außerdem kann der Benutzer den Ort und das Land aus einer Drop-Down-List auswählen. Falls der Ort bzw. das Land noch nicht vorhanden ist, kann der Benutzer auf den danebenliegenden Knopf drücken und einen neuen Ort/Land anlegen.

Neuer Kunde

Titel:

Vorname:

Nachname:

Straße:

Hausnummer:

Ort:

Land:

Telefon 1:

Telefon 2:

E-Mail:

Krankenkassa:

Job:

Hobbies:

Sonstiges 1:

Sonstiges 2:

Versicherungsnummer:

Geburtsdag:

Massennachricht: ☒

Abbildung 3.2: Screenshot Neuen Kunden anlegen

Falls der Benutzer nach dem Anlegen des Kunden noch Änderungen vornehmen möchte, kann er dies auf der Startseite durch einen Doppelklick auf den gewünschten Kunden erledigen. Dadurch erscheint ein neues Fenster, auf welchem der Benutzer alle Daten des Kunden bearbeiten kann und zusätzlich alle Bestellungen des Kunden sieht. Außerdem besteht hier die Möglichkeit den Kunden zu löschen. Damit ist allerdings aus datenbanktechnischen Gründen gemeint, den Kunden nicht mehr bearbeitbar zu machen. Der Benutzer kann keinen Kunden wirklich löschen. Der Grund dafür ist, dass jede Bestellung in der Datenbank auf einen Kunden verweisen muss und wenn ein Kunde bereits mehrere Bestellungen getätigt hat und der Benutzer danach den Kunden löschen möchte, würden alle seine Bestellungen mitgelöscht werden.

Kunde bearbeiten

Titel: Mag.
 Vorname: Max
 Nachname: Mustermann
 Straße: Musterstraße
 Hausnummer: 45
 Ort: 23445 Brillhausen
 Land: Deutschland
 Telefon 1: 06507894566
 Telefon 2:
 E-Mail: mustermann789@gmail.com
 Krankenkassa:
 Job: Angestellter
 Hobbies: Radfahren
 Sonstiges 1:
 Sonstiges 2:
 Versicherungsnummer:
 Geburtstag: 06.07.1996
 Massenanschrift: ☒

Bestellungen

Neue Brillenbestellung | Neue Kontaktlinsenbestellung

Id	Typ	Zahlungsstatus	Bearbeitungsstatus	BruttoPreis	Bestelldatum
3	B	Bezahlte	Bestellt	396.45	27.01.2018
4	K	Offen	Bestellt	59.00	28.01.2018
5	B	Bezahlte	Bestellt	380.45	08.03.2018

Versendete Nachrichten anzeigen | Kunde löschen | Abbrechen | Speichern

Abbildung 3.3: Screenshot der Kundendetails

Technischer Hintergrund: Damit die Kunden auf der Startseite angezeigt werden können, müssen sie zuerst aus der Datenbank in eine ObservableCollection vom Typ Customer geladen werden. Danach wird auf Basis der Datensätze eine ICollectionView erstellt, welche die Daten dann anzeigt. Im Vergleich zur ObservableCollection bietet die ICollectionView beim Anzeigen viele Vorteile (siehe Kapitel 3.1.8). Um einen Kunden zu bearbeiten, muss das Objekt zuerst lokal kopiert werden.

```
private Customer CopyCustomer(Customer item)
{
    Customer customer = new Customer();
    GenericRepository<Customer>.CopyProperties(customer, item);
    if (item.Town_Id != null)
    {
        Town town = new Town(); //Referenced town must be copied as well
        GenericRepository<Town>.CopyProperties(town,
            uow.TownRepository.GetById(item.Town_Id));
        customer.Town = town;
    }
    if (item.Country_Id != null)
    {
        Country country = new Country();
        GenericRepository<Country>.CopyProperties(country,
            uow.CountryRepository.GetById(item.Country_Id));
    }
}
```

```

        customer.Country = country;
    }
    return customer;
}

```

Der Grund dafür ist, dass immer dieselbe Instanz von "UnitOfWork" verwendet wird. Wenn nur eine Referenz auf das Objekt erstellt werden würde, könnten die Änderungen nie rückgängig gemacht werden, weil sie ja immer sofort in die UnitOfWork-Instanz übertragen werden würden (siehe Kapitel 4.1.2).

```
Customer cus = uow.CustomerRepository.GetById(1);
```

Damit ein Kunde gelöscht werden kann, enthält die Klasse Kunde eine Property namens 'Deleted', welche angibt, ob der Kunde gelöscht wurde. Wenn diese auf 'true' gesetzt wird, kann der Benutzer den Kunden durch die Checkbox auf der Startseite ausblenden. Falls er dies nicht tut, wird der Kunde auf der Startseite angezeigt, allerdings erscheint eine Fehlermeldung, wenn der Benutzer versucht die Detailseite des Kunden zu öffnen. Dadurch ist es auch unmöglich neue Bestellungen für diesen Kunden anzulegen oder die Daten des Kunden zu bearbeiten. Die Bestellungen des gelöschten Kunden werden trotzdem normal angezeigt.

3.1.2 Auftragsverwaltung

Grundsätzlich gibt es zwei Arten von Aufträgen: Brillen- und Kontaktlinsenaufträge. Eigentlich haben beide Arten dieselben Eigenschaften, nur der Brillenauftrag verweist auf eine Brillenfassung und der Kontaktlinsenauftrag nicht. Außerdem hat ein Brillenauftrag einen Brillentyp und ein Kontaktlinsenauftrag einen Kontaktlinsentyp. Damit kann der Benutzer bei Brillenaufträgen beispielsweise unterscheiden, ob es sich um eine Fern- oder Nahbrille handelt. Generell wird streng zwischen Brillen- und Kontaktlinsenaufträgen unterschieden, deshalb werden unter dem Menüpunkt "Aufträge" auch zwei verschiedene Listen angezeigt. Jeder dieser Aufträge beinhaltet nur eine Brillen- oder Kontaktlinsenbestellung. Wie gewohnt kann man diese Listen wieder filtern und sortieren. Allerdings kann der Benutzer von dieser Sicht aus keine neuen Aufträge erfassen, dazu muss er in der Kundenverwaltung zuerst einen Kunden auswählen.

Brillen								Kontaktlinsen							
Filtern nach:		Kunde		Filtern	Filter zurücksetzen			Filtern nach:		Kunde		Filtern	Filter zurücksetzen		
Id	Nachname	Bearbeitungsstatus	Zahlungsstatus	Bruttopreis	Bestelldatum	Glastypbezeichnung		Id	Nachname	Bearbeitungsstatus	Zahlungsstatus	Bruttopreis	Bestelldatum	Kontaktlinsentypbeschre	
3	Mustermann	Bestellt	Bezahlt	396.45	27.01.2018	Bildschirmbrille	Dr.	2	Pürmayr	Bestellt	Offen	78.00			
5	Mustermann	Bestellt	Bezahlt	380.45	08.03.2018	Sportbrille		4	Mustermann	Bestellt	Offen	59.00	28.01.2018		
6	Musterfrau	Bestellt	Offen	156.00											

Abbildung 3.4: Screenshot der Auftragsverwaltung

Wenn der Benutzer doppelt auf einen Auftrag klickt, erscheint entweder ein Detailfenster eines Brillen- oder Kontaktlinsenauftrags. Im rechten Bereich des Fensters kann der Benutzer die einzelnen Preise der Komponenten angeben. Das Programm rechnet alle Preise brutto und gibt zum Schluss die darin enthaltene Mehrwertsteuer an. Es wird nach folgender Formel gerechnet:

Zuerst werden linker und rechter Glaspreis, Preis von Sonstigem und wenn vorhanden der Verkaufspreis der Brillenfassung addiert. Davon wird das Krankenkassageld abgezogen, der Selbstbehalt hinzugezählt und der Rabatt (dieser wird in Euro angegeben) wieder subtrahiert. Zwanzig Prozent davon sind schlussendlich die Mehrwertsteuer.

Im unteren Bereich des Fensters kann der Benutzer Details zur Glasverarbeitung angeben. Wenn der Benutzer den Bearbeitungsstatus auf "Abgeholt" setzt, wird auch der Status der Brillenfassung in der Brillenfassungsverwaltung auf "Verkauft" gesetzt.

Der Benutzer hat die Möglichkeit zu jedem Auftrag einen Augenarzt zu vermerken. Von diesem Arzt wird allerdings nur der Name gespeichert. Der Benutzer kann nur aus einer Liste von Ärzten auswählen, welche er selbst mittels dem Button 'Neuer Doktor' angelegt hat.

Brillenbestellung ändern

Kunde: Max Mustermann

Glastyp: Bildschirmbrille

Glastypsonstiges:

Brillenfassung: 123 GUESS

Doktor: Dr. med. Barbara Hube Neuer Doktor

Zahlungsdatum: 29.01.2018

Bestelldatum: 27.01.2018

Zahlungstatus: Beahlt

Bearbeitungsstatus: In Werkstatt

Sonstiges:

Linker Glaspreis: 50.00

Rechter Glaspreis: 55

Preis von Sonstigem: 3

Krankenkassageld: 12

Selbstbehalt: 5

Rabatt: 7

Brillenfassung: 300.45 €

Brutto: 394.45 €

Mehrwertsteuer: 65.74 €

Berechnen

	sph	cyl	Achse	Prisma	PD/NTH	
F	R					FWS/
	L					Ink
N	R					HSA
	L					

Auftragsbestätigung erstellen Rechnung erstellen Nachricht senden Bestellung löschen Abbrechen Speichern

Abbildung 3.5: Screenshot eines Brillenauftrags

Dokumente erstellen

Unter den Angaben der Details zu den Gläsern, kann der Benutzer eine Auftragsbestätigung oder eine Rechnung erstellen. Die Dokumente werden als Word-Dokumente in einem Ordner abgespeichert. Das hat den Vorteil, dass der Benutzer selbst entscheiden kann, ob er noch etwas nachträglich ändern will oder das generierte Dokument gleich ausdruckt oder versendet. Gleichzeitig hat das Benutzen eines Word-Dokuments auch einen großen Nachteil, denn wenn der Benutzer nachträglich etwas ändert, wird diese Information nicht in das System weitergeleitet und somit könnten die erstellten Dokumente und der Auftrag selbst nicht dieselben Informationen beinhalten. Außerdem wird in der Datenbank immer nur der Pfad der zuletzt erstellten Rechnung/Auftragsbestätigung gespeichert. Das bedeutet, dass der Benutzer auch mehrere Dokumente zum selben Auftrag erstellen kann. Diese könnten sich auch voneinander unterscheiden, worüber das Programm ebenfalls keine Kontrolle hat. In diesem Fall wird nur ein Hinweis mit dem abgespeicherten Dokumentnamen angezeigt und gefragt, ob wirklich ein neues Dokument erstellt werden soll.

Damit das automatische Erstellen der Word-Dokumente funktioniert, muss der Benutzer eine Word-Vorlage erstellen, die Felder enthält (MergeFields), welche das Programm ersetzen kann. Der Dokumentname der generierten Datei besteht immer aus der Bestell-Id, dem Dokumenttyp (Rechnung oder Auftragsbestätigung), dem Nachnamen des Kunden und dem aktuellen Datum.

Eine generierte Auftragsbestätigung könnte aussehen, wie Abbildung-1 im Anhang. TODO

Technischer Hintergrund:

Zur Erstellung von Word-Dokumenten wird Interop verwendet (siehe Kapitel 2.6). Die Methode "CreateDocument" benutzt eine vom Benutzer erstellte Wordvorlage und ersetzt die darin enthaltenen Mergefields. Der erste Parameter "orderId" gibt die Id der gewählten Bestellung an. Mit Hilfe dieser Nummer können aus der Datenbank die restlichen Daten geladen werden. Der Parameter "oTemplatePath" beschreibt den Pfad der Wordvorlage und der String "path" den Namen, unter dem das Dokument abgespeichert werden soll. Der Code unterhalb ist nur ein Ausschnitt der Methode.

```
private static bool CreateDocument(int orderId, Object oTemplatePath, string
    path)
{
    Application wordApp = new Application();
    Document wordDoc = new Document();
    try
    {
        Order order;
        Customer cus;
        //Ausgeschnitten: Hier werden Order und Customer Werte aus der Datenbank
        zugewiesen
        Object oMissing = System.Reflection.Missing.Value;
        wordDoc = wordApp.Documents.Add(ref oTemplatePath, ref oMissing, ref
            oMissing, ref oMissing);
```

```

foreach (Field myMergeField in wordDoc.Fields)
{
    Range rngFieldCode = myMergeField.Code;
    String fieldText = rngFieldCode.Text;

    //Nur Mergefields sollte bearbeitet werden
    if (fieldText.StartsWith(" MERGEFIELD"))
    {
        string translatedFieldName;
        //Ausgeschnitten: Hier wird der Name der Property aus dem fieldText
        //herausgeholt und auf Englisch uebersetzt
        string value = String.Empty;
        //Ausgeschnitten: Die Property wird in den Klassen gesucht und der
        //Wert gespeichert z.B: Properties der Klasse Customer:
        if (typeof(Customer).GetProperty(translatedFieldName) != null)
        {
            value =
                cus.GetType().GetProperty(translatedFieldName).GetValue(cus,
                null)?.ToString();
        }
        myMergeField.Select();
        wordApp.Selection.TypeText(value);
    }
}
int idx = oTemplatePath.ToString().LastIndexOf("\\");
string p = oTemplatePath.ToString().Substring(0, idx + 1);
string completePath = p + path + ".docx";
wordDoc.SaveAs(completePath);
wordDoc.Close();
wordApp.Quit();
return true;
}
catch (Exception e)
{
    Console.WriteLine(e);
    wordApp.Quit();
    wordDoc.Close();
    return false;
}
}

```

Zuerst werden eine neue Applikation und ein neues Dokument erstellt. Nachdem der Auftrag und der Kunde aus der Datenbank geladen worden sind, wird die Vorlage für das Dokument geladen. Dabei ist nur der erste Parameter ("oTemplatePath") von Bedeutung. Danach werden in einer Foreach-Schleife alle Felder des Dokumentes durchgegangen. Dabei werden nur jene bearbeitet, bei denen der Text des Feldes mit "MERGEFIELD" beginnt (es gibt auch andere Arten von Feldern in Word, jedoch sind in der Vorlage alle zu bearbeitenden Felder sogenannte Mergefields). Dann wird die gesuchte Property (Co-

de ausgeschnitten) mit Hilfe der Variable "fieldText" ermittelt. "fieldText" muss aber zuerst auf englisch übersetzt werden, da die Namen der Felder in der Vorlage deutsch sind, die Properties der Klassen aber englisch. Der übersetzte Name der Property wird in der Variable "translatedFieldName" gespeichert. Anschließend werden allen Klassen nach dem Propertynamen durchsucht. Wenn die richtige Klasse gefunden wurde, wird der Wert der Property des Objekts in der Variable "val" gespeichert. Danach wird an Stelle des Mergefields in der Vorlage der ermittelten Wert eingetragen. Zum Schluss wird noch der gewünschte Pfad ermittelt und das Dokument unter diesem Pfad gespeichert. Genau die selbe Methode wird benutzt, wenn eine Rechnung erstellt wird, die aussehen könnte wie das zweite Bild im Anhang. TODO

3.1.3 Lieferantenverwaltung

Ebenso wie für die Kunden, gibt es auch eine Verwaltung für die Lieferanten des Benutzers. Auch die Lieferanten lassen sich filtern und sortieren. Lieferanten haben folgende Attribute: Name, Ort, Land, Adresse, FAX, Telefon, E-Mail, Kundennummer (damit ist die Id des Benutzers beim jeweiligen Lieferanten gemeint), Kontaktperson, Produkte und Sonstiges. Einen neuen Lieferant kann man mittels dem Button links oben anlegen und Lieferanten bearbeiten und löschen kann der Benutzer durch einen Doppelklick auf den gewünschten Lieferanten.

Id	Name	PLZ	Ort	Straße	Hausnummer	Land	FAX	Telefon	Email	Kundennummer	Kontaktperson	Produkte
1	Silhouette	4020	Linz	Silhouettestraße	3	Österreich	0049-711-123456	07242 206456	j@gmail.com	K2343	Pürmayr Johann	Sonnenbrillen und normale Brill
2	Luxottica	P-345	ItalienischerOrt	Luxotticaweg	345	Italien	0047/789456232	07242 456789	ialksdjf@aon.at	K-34	Italiano Fernando	Sonnenbrillen
3	Charmant	23445	Brillhausen	Charmantgasse	32	Deutschland	456456464	5465454	charmant@aon.at	K2343	Charmant Fritz	Edle Brillen
4	Silhouette	4020	Linz	Silhouettestraße	3	Österreich	0049-711-123456	07242 206456	j@gmail.com	K2343	Pürmayr Johann	Sonnenbrillen und normale Brill
5	Luxottica	P-345	ItalienischerOrt	Luxotticaweg	345	Italien	0047/789456232	07242 456789	ialksdjf@aon.at	K-34	Italiano Fernando	Sonnenbrillen
6	Charmant	23445	Brillhausen	Charmantgasse	32	Deutschland	456456464	5465454	charmant@aon.at	K2343	Charmant Fritz	Edle Brillen

Abbildung 3.6: Screenshot der Lieferantenverwaltung

3.1.4 Verwaltung der lagernden Brillenfassungen

Genau wie bei der Verwaltung der Kunden und der Lieferanten gibt es auch eine Verwaltung der lagernden Brillenfassungen. Jede Brille die der Optiker verkauft, hat eine eigene Fassung und die wird hier erfasst. Dabei hat jede Brillenfassung folgende Attribute: Modell, Marke, Farbe, Größe, Status (bestellt, lagernd oder verkauft), Einkaufspreis, Einkaufsdatum, Verkaufspreis, Verkaufsdatum und der Lieferant. Die Liste kann wie gewohnt gefiltert und sortiert werden. Um eine neue Brillenfassung zu erfassen kann der Benutzer auf den Button links oben klicken und um eine bestehende Brillenfassung zu bearbeiten oder zu löschen muss der Benutzer einen Doppelklick auf die gewünschte

Brillenfassung tätigen.

Eigentlich würde man erwarten, dass zu den Brillenfassungen auch die Anzahl an lagernden Stück abgespeichert wird. Allerdings wurde dieses Feature nach Absprache mit dem Auftraggeber nicht implementiert, da jede Brillenfassung für jeden Auftrag einzeln bestellt wird. Deswegen wird auch der Status der Brillenfassung gespeichert.

Id	Modell	Marke	Farbe	Größe	Status	Einkaufspreis	Einkaufsdatum	Verkaufspreis	Verkaufsdatum	Lieferant
1	M-789	Joop	schwarz	36	Bestellt	78.50	02.05.2017	100.00	09.08.2017	Luxottica
2	123	GUESS	grau	45	Lagernd	200.00	02.12.2015	300.45	04.09.2016	Charmant
3	789	ZEISS	weiß	7	Lagernd	40.78	05.04.2017	94.00	01.08.2017	Silhouette
4	M-789	Joop	schwarz	36	Bestellt	78.50	02.05.2017	100.00	09.08.2017	Luxottica
5	123	GUESS	grau	45	Lagernd	200.00	02.12.2015	300.45	04.09.2016	Charmant
6	789	ZEISS	weiß	7	Lagernd	40.78	05.04.2017	94.00	01.08.2017	Silhouette

Abbildung 3.7: Screenshot der lagernden Brillenfassungen

3.1.5 E-Mail und SMS

Massennachrichten

Um regelmäßige Info- und Werbenachrichten auszusenden, bietet das Programm die Möglichkeit Nachrichten an alle Kunden zu versenden. Die Nachrichten können entweder als E-Mails oder SMS versendet werden. Für den Fall, dass ein Kunde diese Nachrichten nicht mehr erhalten möchte, kann das der Benutzer bei einzelnen Kunden auf der Detailseite eintragen.

E-Mail

Wenn der Benutzer eine Massenmail versenden möchte, kann er einen Betreff und eine Nachricht eingeben, die nachher an alle Kunden gesendet wird. Als E-Mail-Adresse, wird die abgespeicherte E-Mail-Adresse des Kunden verwendet. Falls keine E-Mail-Adresse angegeben wurde, wird eine entsprechende Fehlermeldung angezeigt.

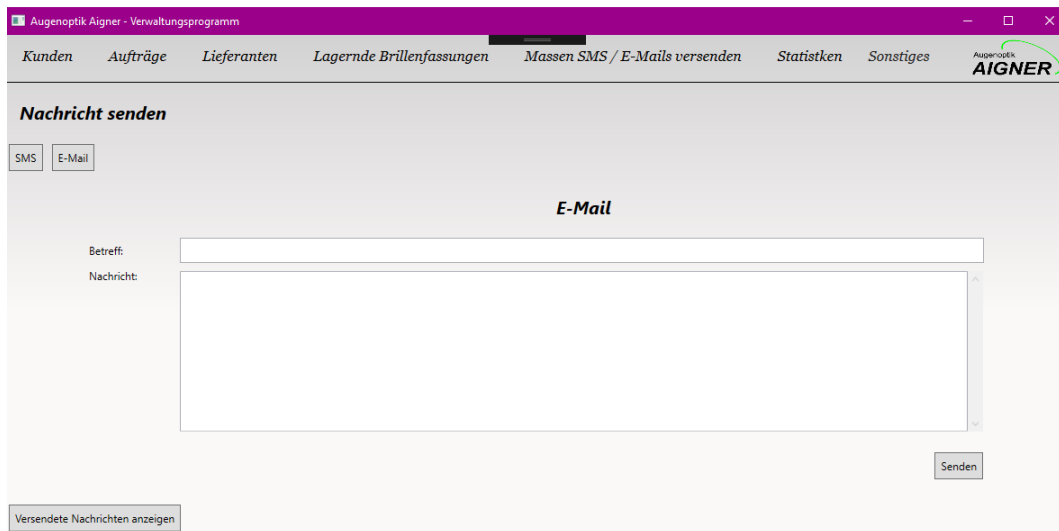


Abbildung 3.8: Screenshot der Massen E-Mails

Technischer Hintergrund:

Es wird für jeden Kunden die gleiche Mail erstellt (Klasse MailMessage vom Namespace System.Net.Mail). Diesem Objekt werden Attribute wie Sender, Empfänger, Betreff, Nachricht usw. gesetzt und mittels eines SMTP-Clients versendet (Klasse SmtpClient ebenfalls vom Namespace System.Net.Mail). Der Smtp-Client bekommt noch Informationen wie Host, Port und natürlich die E-Mail-Adresse, von der die E-Mail weggeschickt werden soll, sowie das Passwort für die E-Mail-Adresse. In diesem Fall wurde eine Gmail-Adresse verwendet, die extra für diesen Zweck erstellt wurde.

```
var message = new MailMessage();
message.To.Add(new MailAddress(item.Email));
message.From = new MailAddress("infodienst.augenoaptikaigner@gmail.com");
message.Subject = this.Subject;
message.Body = this.Message;
this.Recipients.Add(new CustomRecipient() { Customer = item, Address =
    item.Email });

using (var smtp = new SmtpClient())
{
    var credential = new NetworkCredential
    {
        UserName = "infodienst.augenoaptikaigner@gmail.com",
        Password = //not shown here
    };
    smtp.Credentials = credential;
    smtp.Host = "smtp.gmail.com";
    smtp.Port = 587;
```

```

smtp.EnableSsl = true;
await smtp.SendMailAsync(message);
}

```

Danach wird die gesendete Nachricht noch in die Datenbank abgespeichert, damit der Benutzer später einen Überblick über alle gesendeten Nachrichten hat.

SMS

Zum Versenden der SMS wird der SMS-Dienst MessageBird verwendet (siehe Kapitel 2.8). Ähnlich wie beim Versenden einer E-Mail, gibt der Benutzer wieder eine Nachricht ein, allerdings kann er keine Betreff einfügen.

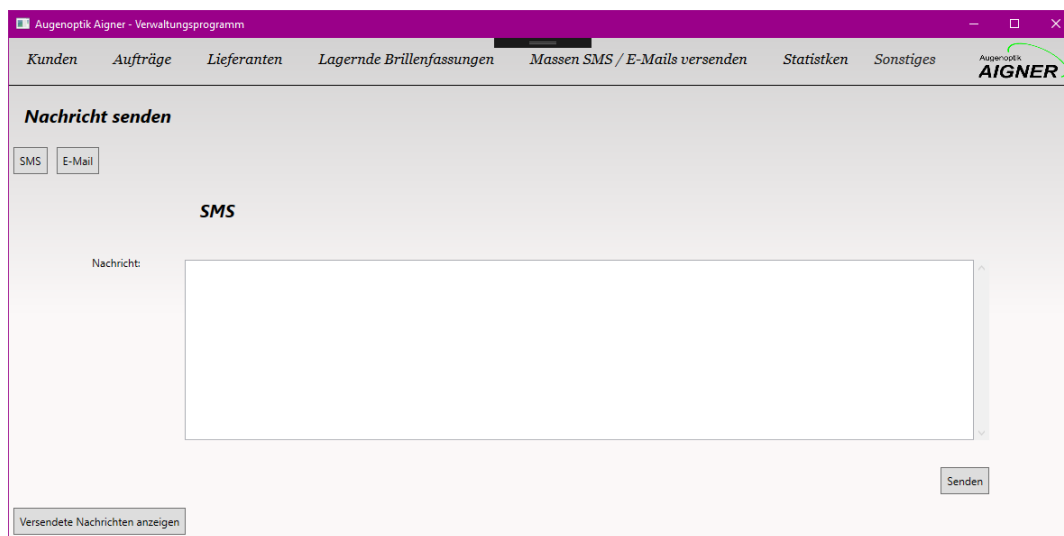


Abbildung 3.9: Screenshot der Massen SMS

Diese Nachricht wird dann an alle Kunden gesendet, außer an jene, bei denen eingetragen worden ist, dass sie keine Massennachricht mehr erhalten wollen. Als Telefonnummer wird standardmäßig die Telefonnummer 1 gewählt, außer diese ist nicht vorhanden, dann wird die Telefonnummer 2 gewählt. Die ausgewählte Nummer sollte eine mobile Telefonnummer (kein Festnetz) sein, sonst kann die Nachricht nicht versendet werden.

Einzelne Nachrichten

Dieselben Vorgänge werden auch verwendet um einzelne Nachrichten zu versenden. Dazu muss der Benutzer auf die Detailseite einer Bestellung klicken und dann auf „Nachricht senden“. Standardmäßig wird ein Text eingefügt, der dem Kunden mitteilt, dass seine Bestellung nun abholbereit ist. Der Standardtext kann vom Benutzer allerdings geändert

werden (siehe Kapitel 3.1.7). Falls der Benutzer jedoch einmal einen ganz anderen Text versenden wollen, kann der Benutzer die Nachricht natürlich auch verändern.

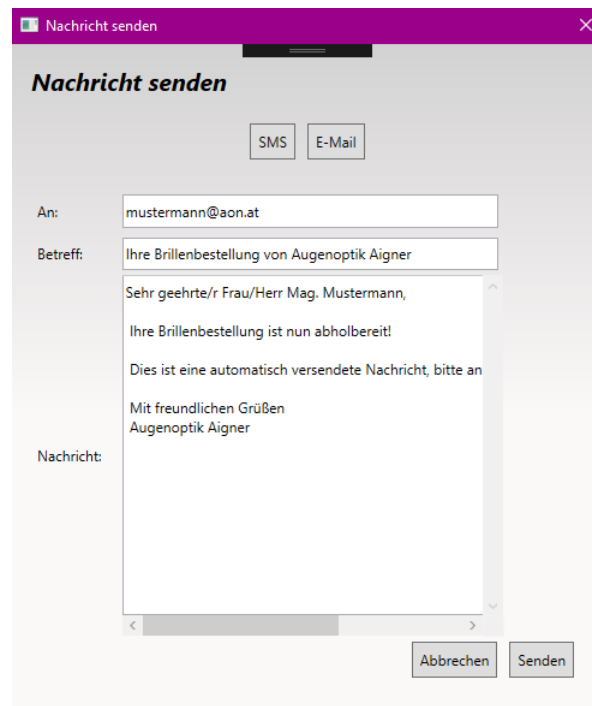


Abbildung 3.10: Screenshot der einzelnen Nachricht

Versendete Nachrichten

Außerdem ist es möglich, alle Nachrichten, die vom System gesendet worden sind, anzuzeigen. Um nur Nachrichten anzuzeigen, die an einen bestimmten Kunden gesendet worden sind, muss der Benutzer auf die Detailseite eines Kunden klicken und dann die „Versendeten Nachrichten“ anzeigen. Falls der Benutzer alle Nachrichten sehen will, die er versendet hat, kann er diese unter dem Menüpunkt „Massen SMS /E-Mails verschicken“ sehen.

Dazu wurden in der Datenbank extra die Tabellen "CustomMessage" und "CustomRecipient" angelegt, um alle Nachrichten und deren Empfänger abspeichern zu können. Hier wird beispielsweise eine Massensms gespeichert:

```
var m = new CustomMessage();  
m.Date = DateTime.Now;  
m.MessageText = this.Message;  
m.MessageType = OpticiatnMgr.Core.Entities.MessageType.SMS;  
m.Recipients = new List<CustomRecipient>();  
var numbers = GetPhoneNumbers();
```

```

for (int i = 0; i < this.Customers.Count; i++)
{
    m.Recipients.Add(new CustomRecipient() { Customer = this.Customers[i],
        Address = numbers[i].ToString() });
}
uow.MessageRepository.Insert(m);
uow.Save();

```

Datum	Empfänger	Betreff	Nachricht	Auftrags Id	Typ
04.03.2018 22:56:08	Pürmayr evapuermayr@gmail.com Mustermann mustermann@aon.at	Test	Das ist ein Test.	0	Email
04.03.2018 22:51:21	Pürmayr 00436505004369		Ihre Kontaktlinsenbestellung ist nun abholbereit!	2	SMS

Abbildung 3.11: Screenshot der versendeten Nachrichten

3.1.6 Statistiken

Unter dem Menüpunkt "Statistiken" erhält der Benutzer eine Übersicht über die Anzahl der verkauften Brillen oder Kontaktlinsen. Das kann er mittels den beiden Buttons im oberen Teil der Seite festlegen. Darunter wird zum Vergleichen der Verkaufszahlen ein Liniendiagramm der verkauften Brillen/Kontaktlinsen von dem aktuellen Jahr und dem Jahr davor angezeigt. Die Werte der Statistik sind in Monate aufgegliedert.

Damit ein Brillen/Kontaktlinsenauftrag in der Statistik mitberücksichtigt wird, muss ein Zahlungsdatum angegeben werden und der Bezahlungsstatus muss auf "Beahlt" gesetzt werden.

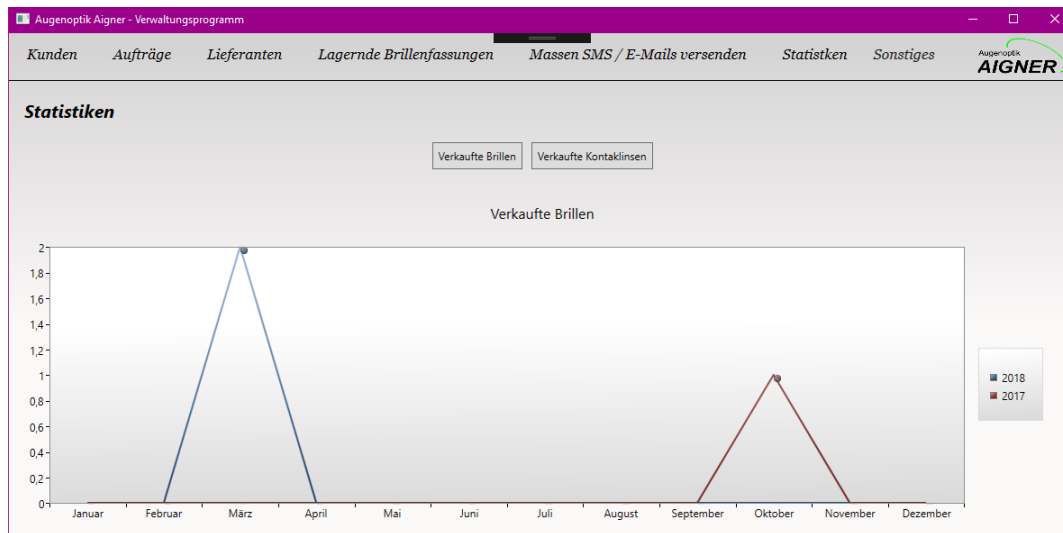


Abbildung 3.12: Screenshot der Statistiken

Technischer Hintergrund

Zur Darstellung wurde das Wpf-Toolkit verwendet (Kapitel 2.7).

Um ein Liniendiagramm zu erzeugen ist folgender Code nötig:

```
<toolkitCharting:Chart Title="{Binding Title}">
  <toolkitCharting:LineSeries Title="{Binding NewYear}"
    DependentValueBinding="{Binding Value}"
    IndependentValueBinding="{Binding Key}" ItemsSource="{Binding
    NewValues}" />
  <toolkitCharting:LineSeries Title="{Binding OldYear}"
    DependentValueBinding="{Binding Value}"
    IndependentValueBinding="{Binding Key}" ItemsSource="{Binding
    OldValues}" />
</toolkitCharting:Chart>
```

”NewValues” und ”OldValues” sind so deklariert:

```
public ObservableCollection<KeyValuePair<string, int>> NewValues { get; set; }
public ObservableCollection<KeyValuePair<string, int>> OldValues { get; set; }
```

Die Daten werden mittels Linq (Kapitel 2.1.1) erfasst.

3.1.7 Sonstiges

Unter dem Menüpunkt ”Sonstiges” erscheinen fünf Unterpunkte:

- Orte bearbeiten

- Länder bearbeiten
- Brillentypen bearbeiten
- Kontaktlinsentypen bearbeiten
- Vorgegebene Texte bearbeiten

Wie die Überschriften schon vermuten lassen, öffnen die vier oberen Buttons jeweils ein eigenes Fenster, welches eine Übersicht über alle vorhandenen Objekte zeigt. Am Beispiel "Länder" wird nun die Verwendung gezeigt:

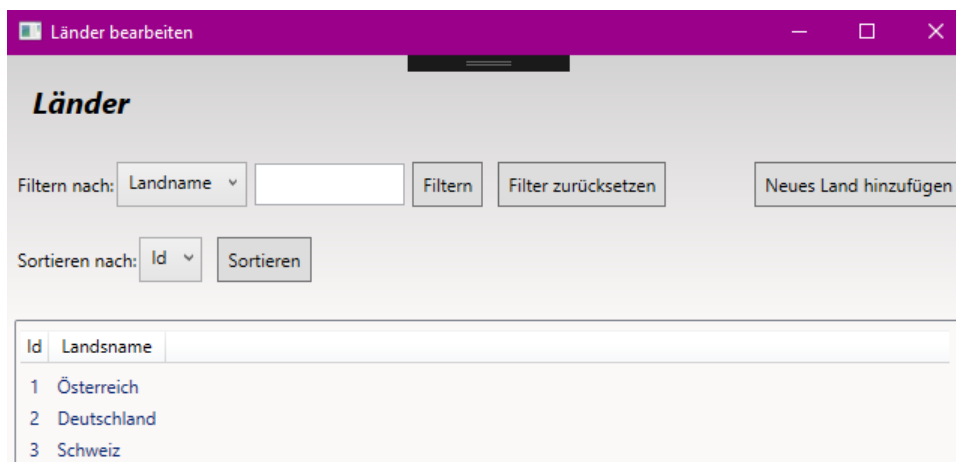


Abbildung 3.13: Screenshot der Länder

Im oberen Bereich können die Länder nach Name oder Id gefiltert werden. Allerdings funktioniert das Sortieren hier anders als bei den Übersichtlisten. Der Benutzer kann auswählen, nach was er gerne sortieren möchte und danach sortiert das Programm aufsteigend nur nach dieser einen Property. Mit einem Doppelklick kann ein Land bearbeitet/gelöscht werden und rechts oben kann ein neues Land hinzugefügt werden.

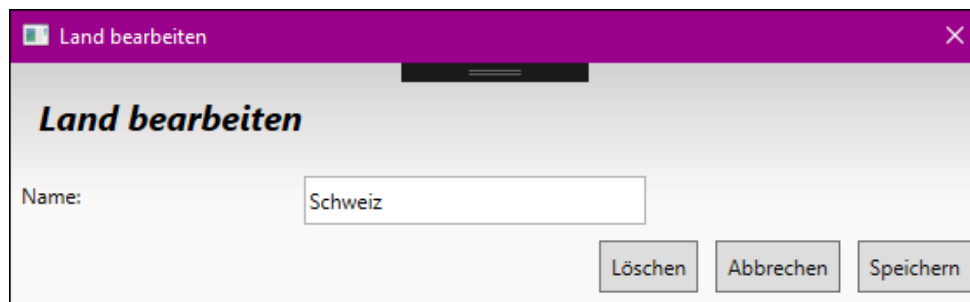


Abbildung 3.14: Land bearbeiten

Vorgegebene Texte bearbeiten

Unter diesem Menüpunkt können Texte, die standardmäßig im Programm vorgeschlagen werden, bearbeitet werden. Dazu gehören sämtliche Texte und bei E-Mails auch Betreffe, die beim Versenden einer einzelnen Nachricht vorgeschlagen werden. Außerdem kann auch der Name bearbeitet werden, der Kunden angezeigt wird, wenn sie eine SMS vom Programm erhalten.

Bei allen Texten außer dem Sendernamen von SMS, hat der Benutzer die Möglichkeit die Nachrichten zu personalisieren. Dazu muss er nur an der Stelle, an dem zum Beispiel der Nachname des Kunden eingefügt werden soll, "{3}" schreiben. Außerdem kann der Benutzer auch Zeilenumbrüche einfügen, damit beispielsweise in einer E-Mail nicht nur eine einzige Zeile Text steht.

Vorgegebene Texte bearbeiten

Vorgegebene Texte

Einzelne E-Mail: Betreff Brillenauftrag: Ihre Brillenbestellung von Augentoptik Aigner

Text Brillenauftrag: Sehr geehrte/r Frau/Herr {1} {3}, {0} {0} Ihre Brillenbestellung ist nun abholbereit! {0} {0}

Betreff Kontaktlinsenauftrag: Ihre Kontaktlinsenauftrag: Ihre Kontaktlinsenauftrag von Augentoptik Aigner

Text Kontaktlinsenauftrag: Sehr geehrte/r Frau/Herr {1} {3}, {0} {0} Ihre Kontaktlinsenauftrag ist nun abholbere

Einzelne SMS: Text Brillenauftrag: Ihre Brillenbestellung ist nun abholbereit!

Text Kontaktlinsenauftrag: Ihre Kontaktlinsenauftrag ist nun abholbereit!

SMS Sender Text OptikAigner

{0} = Zeilenumbruch, {1} = Titel des Kunden, {2} Vorname des Kunden, {3} Nachname des Kunden

Abbrechen Speichern

Abbildung 3.15: Vorgegebene Texte bearbeiten

Technischer Hintergrund:

Um diese Informationen zu speichern, werden sogenannte User-Settings verwendet. Diese sind in den Properties des Projektes unter dem Punkt "Settings" eingetragen und haben die Eigenschaft, dass sie nicht zurückgesetzt werden, sobald das Programm beendet wird. In diesen "Settings" kann der Name der Einstellung, der Typ (in dieser Arbeit wird nur der Typ "string" benötigt), der Geltungsbereich und der Wert eingetragen werden. Der Geltungsbereich kann entweder "Application" oder "User" sein. Der Unterschied besteht darin, dass "Application"-Einstellungen zur Laufzeit nicht verändert

werden können und "User"-Einstellungen schon. Deshalb sind alle Einstellungen dieser Arbeit "User"-Einstellungen.

Im Code des ViewModels sind Properties zu den zu bearbeitenden Texten vorhanden, welche im Getter nur die passende Einstellung zurückliefern und im Setter diesen Text speichern.

Die nachfolgende Property beschreibt den Text im Betreff, der angezeigt wird, wenn eine einzelne E-Mail an einen Kunden mit einem Brillenauftrag versendet wird.

```
public string SingleEmailSubjectGlassesOrder
{
    get
    {
        return Properties.Settings.Default.SingleEmailSubjectGlassesOrder;
    }
    set
    {
        Properties.Settings.Default.SingleEmailSubjectGlassesOrder = value;
    }
}
```

Bevor das Fenster mit "Speichern" geschlossen wird, muss unbedingt folgende Methode aufgerufen werden. Ansonsten werden die Änderungen nicht gespeichert und beim nächsten Programmstart werden wieder die alten Texte angezeigt.

```
Properties.Settings.Default.Save();
```

Falls der Benutzer die Änderungen nicht speichern möchte kann er den "Abbrechen"-Button betätigen. Dann muss allerdings nachfolgende Methode aufgerufen werden, ansonsten bleiben die Änderungen bis zum Schließen des Programms erhalten. Erst beim nächsten Programmstart werden sie wieder zurückgesetzt.

```
Properties.Settings.Default.Reload();
```

3.1.8 Filtern und Sortieren

Filtern

Auf allen Hauptseiten der Applikation (Kunden, Aufträge, Lieferanten, Lagernde Brillenfassungen) sowie auf den Seiten unter dem Menüpunkt „Sonstiges“ (Orte, Länder, Brillen- und Kontaktlinsentypen) ist es möglich die Datensätze zu filtern. Dies passiert immer nach demselben Schema, dennoch ist diese Funktion für jede dieser Seiten einzeln implementiert. Dazu muss der Benutzer das Feld aussuchen, nach welchem er gerne filtern möchte, danach einen Text eingeben und dann auf „Filtern“ klicken oder die Taste „Enter“ drücken. Das Programm gibt nun nur jene Datensätze aus, bei denen das gewünschte Feld den eingegebenen Text enthält. Neben dem „Filtern“-Button befindet sich ein „Filter löschen“-Button, der wieder alle Datensätze zum Vorschein bringt.

Lagernde Brillenfassungen										
Filtern nach: Modell ▾ <input type="text" value="789"/> Filtern Filter zurücksetzen										
Id	Modell	Marke	Farbe	Größe	Status	Einkaufspreis	Einkaufsdatum	Verkaufspreis	Verkaufsdatum	Lieferant
1	M-7899	Joop	schwarz	36	Bestellt	78.50	02.05.2017	100.00	09.08.2017	Luxottica
3	789	ZEISS	weiß	7	Lagernd	40.78	05.04.2017	94.00	01.08.2017	Silhouette

Abbildung 3.16: Screenshot des Filters

Im nachfolgenden Beispiel wird anhand der „Lagernden Brillenfassungen“ erklärt wie der Filter funktioniert. Im ViewModel gibt es ein Feld, welches „PropertiesList“ heißt (vom Typ `ObservableCollection<string>`). In diesem werden alle Felder der jeweiligen Klasse aufgezählt. Davor werden noch Felder, nach denen der Benutzer später nicht filtern sollte, herausgestrichen. Bei Referenzen auf andere Objekte, zum Beispiel bei der Brillenfassung der Lieferant, wird die „Supplier_Id“ entfernt, der String „Supplier“ bleibt allerdings in der Liste. Später beim Übersetzen ins Englische wird überprüft, ob nach einem Fremdschlüssel gefiltert wird. In diesem Fall wird der Name des Fremdschlüssels (hier „Supplier“) zu dem Hauptnamen in der Property umgewandelt („SupplierName“). Das bedeutet, dass wenn der Lieferant als Filterfeld ausgewählt wird, in Wirklichkeit nur nach einem Feld (hier dem Namen des Lieferanten) gefiltert wird.

Nachdem aller Felder ausgewählt wurden, wird jedes Feld ins Deutsche übersetzt. Dies geschieht mittels einem kleinen Wörterbuch (Klasse `ResourceManager`), welches eine Übersetzung für jedes Feld bereithält. Die Wörter, die im `ResourceManager` stehen, müssen selbst eingefügt werden und werden in einem File namens „Resources.resx“ unter den Properties des Projektes abgespeichert. Neben einfachen Wörtern könnten hier auch Bilder, Dateien oder Ähnliches verwaltet werden.

Hier wird die Liste der Felder aus denen der Benutzer später sein „Filterfeld“ auswählen kann erstellt:

```

public ObservableCollection<String> PropertiesList { get; }

private ResourceManager manager = Properties.Resources.ResourceManager;

private ObservableCollection<string> GetAllProperties()
{
    ObservableCollection<string> props = new
        ObservableCollection<string>(typeof(EyeGlassFrame).GetProperties()
        .Select(p => p.Name).ToList());
    ObservableCollection<string> newList = new ObservableCollection<string>();
    props.Remove("Timestamp"); //Shouldnt be able to filter by timestamp
    props.Remove("Supplier_Id"); //Shouldnt be able to filter by supplier_id
    foreach (var item in props)

```

```

{
    var germanItem = manager.GetString(item);
    if (germanItem != null)
        newList.Add(germanItem);
}
return newList;
}

```

Wenn der Benutzer einen Text eingibt und danach „Filtern“ drückt, wird das Feld, das er gewählt hat zuerst mit Hilfe des ResourceManagers ins Englische übersetzt. Danach wird die Methode Filter() aufgerufen, die den passenden Filter setzt, falls der Benutzer einen Text eingegeben hat.

```

public void Filter()
{
    try
    {
        if (!String.IsNullOrEmpty(this.FilterText))
        {
            this.EyeGlassFramesView.Filter = new Predicate<object>(Contains);
        }
        else
            this.EyeGlassFramesView.Filter = null;
    }
    catch (Exception e)
    {
        Console.WriteLine(e.StackTrace);
    }
}

```

Dabei muss man wissen, dass EyeGlassFramesView vom Typ ICollectionView ist. Diese Property wird im Konstruktor aus der Liste der wirklichen Brillenfassungen erzeugt (EyeGlassFrames). Der Typ ICollectionView ist als Anzeigeelement für Listen gedacht, weshalb es auch ein Feld namens „Filter“ gibt (mehr dazu: [Mic17a]). Durch die Methode „Contains“ wird dieser auch gesetzt. Der folgende Code stammt aus dem ViewModel der lagernden Brillenfassungen.

Properties:

```

public ObservableCollection<EyeGlassFrame> EyeGlassFrames { get; set; }
public ICollectionView EyeGlassFramesView { get; set; }
public string TranslatedFilterProperty { get; set; }
public string FilterText { get; set; }

```

Im Konstruktor wird EyeGlassFramesView initialisiert.

```

this.EyeGlassFramesView = CollectionViewSource.DefaultView(EyeGlassFrames);

```

Die Methode Contains gibt zurück, ob das Objekt „f“ dem angegebenen Filter entspricht.

Dazu wird zunächst überprüft, ob die Klasse EyeGlassFrame die Property enthält, nach der der Benutzer filtert. Wenn ja, gibt die Methode zurück, ob in dieser Property die Zeichenkette vorkommt, nach der der Benutzer sucht. Danach überprüft das Programm ob die gesuchte Eigenschaft eine Eigenschaft der Klasse Supplier ist. Das passiert, weil jede lagernde Brillenfassung einen Lieferanten hat. Deswegen kann es sein, dass der Benutzer nach einer Eigenschaft filtert, die gar nicht in der Klasse EyeGlassFrame enthalten ist, sondern nur in der Klasse Supplier. Wenn die Property in keiner Klasse gefunden wird, was nicht vorkommen sollte, wird eine Fehlermeldung zurückgegeben.

```
private bool Contains(object f)
{
    EyeGlassFrame frame = f as EyeGlassFrame;
    if (typeof(EyeGlassFrame).GetProperty(TranslatedFilterProperty) != null)
    {
        return frame.GetType().GetProperty(this.TranslatedFilterProperty)
            .GetValue(frame,
                null)?.ToString().ToUpper().IndexOf(this.FilterText.ToUpper()) >= 0;
    }
    else if (typeof(Supplier).GetProperty(TranslatedFilterProperty) != null)
        //Does the user filter by suppliername?
    {
        return frame.Supplier?.GetType()
            .GetProperty(this.TranslatedFilterProperty)
            .GetValue(frame.Supplier,
                null)?.ToString().ToUpper().IndexOf(this.FilterText.ToUpper()) >= 0;
    }
    else
    {
        MessageBox.Show("Beim Filtern ist ein Fehler aufgetreten!");
        return false;
    }
}
```

Sortieren

Bei den allen Hauptseiten, auf denen Daten angezeigt werden, ist eine dynamische Sortierung implementiert. Diese macht es dem Benutzer möglich, nach drei Spalten gleichzeitig auf- oder absteigend zu sortieren. Dazu muss der Benutzer auf eine beliebige Spaltenüberschrift klicken. Das ist dann die Spalte, nach der zuerst aufsteigend sortiert wird. Drückt der Benutzer erneut auf dieselbe Spaltenüberschrift, werden die Datensätze nach dieser Spalte absteigend sortiert. Wenn der Benutzer nach einer zweiten Spalte sortieren möchte, muss er zusätzlich die Shift-Taste drücken, während er eine weitere Spaltenüberschrift auswählt. Wiederrum muss der Benutzer ein zweites Mal mit der Shift-Taste die gleiche Spaltenüberschrift anklicken, um absteigend zu sortieren. Dasselbe gilt für die dritte Spalte. Um die Sortierung wieder zurücksetzen zu können, kann der Benutzer eine andere Spaltenüberschrift mit einem normalen Klick wieder sortieren.

Im nachfolgenden Bild hat der Benutzer zuerst nach dem Vornamen aufsteigend, dann nach der Postleitzahl absteigend und zum Schluss nach Nachnamen aufsteigend sortiert. Zur besseren Übersichtlichkeit zeigt das Programm einen normalen Pfeil oder einen Pfeil mit einem oder zwei Punkten an, je nachdem in welcher Reihenfolge die Spalten sortiert wurden.




Id	Titel	Vorname 	Nachname 	Straße	PLZ 	Ort
4		Eva	Musterfrau	Musterstraße	4611	Buchkirchen
1		Eva	Pürmayr	Sonnenstraße	4611	Buchkirchen
5		Eva	Test	Teststraße	4020	Linz
6	Mag.	Max	Mustermann	Musterstraße	23445	Brillhausen

Abbildung 3.17: Screenshot des Sortierens

Technischer Hintergrund:

Als Grundlage zur Implementierung war ein Beispiel von Microsoft sehr hilfreich ([Mic17b]).

Es gibt eine Klasse SortManager, die die Sortierung für alle Hauptseiten regelt.

```
public SortManager SortManager { get; set; }
```

Dazu wird im Konstruktor jedes ViewModels ein neuer SortManager initialisiert.

```
SortManager = new SortManager();
```

Zusätzlich werden drei Events von der View abonniert.

Das erste Event "Initialized" wird von der ListView bereitgestellt:

```
<i:Interaction.Triggers>
  <i:EventTrigger EventName="Loaded">
    <cmd:EventToCommand Command="{Binding Initialized}"
                        PassEventArgsToCommand="True" />
  </i:EventTrigger>
</i:Interaction.Triggers>
```

In jedem ListViewHeader werden noch Shift+LeftClick und MouseDown durch Mouse-Bindings abonniert. Als Beispiel wurde die Spalte der Vornamen in der Kundenübersicht gewählt.

```
<GridViewColumn DisplayMemberBinding="{Binding FirstName,
  UpdateSourceTrigger=PropertyChanged}">
  <GridViewColumnHeader Content="Vorname">
    <GridViewColumnHeader.InputBindings>
      <MouseBinding Gesture="Shift+LeftClick" Command="{Binding SortShift}"
        CommandParameter="FirstName" >
```

```

        </MouseBinding>
    </GridViewColumnHeader.InputBindings>
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="MouseDown">
            <cmd:EventToCommand Command="{Binding SortCommand}"
                                PassEventArgsToCommand="True" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</GridViewColumnHeader>
</GridViewColumn>

```

Im ViewModel gibt es die zugehörigen ICommandS:

```

public ICommand SortCommand { get; set; }
public ICommand SortShift { get; set; }
public ICommand Initialized { get; set; }

```

Diese werden im Konstruktor initialisiert:

```

SortCommand = new RelayCommand<RoutedEventArgs>(SortS);
SortShift = new RelayCommand<object>(SortSh);
Initialized = new RelayCommand<RoutedEventArgs>(Init);

```

Im ViewModel werden dann folgende Methoden aufgerufen:

```

private void Init(RoutedEventArgs p)
{
    SortManager.Init(p);
}
//Click without shift key
private void SortS(RoutedEventArgs e)
{
    var tmp = this.CustomersView;
    SortManager.SortNormal(e, ref tmp);
}
//Click with shift
private void SortSh(object p)
{
    var tmp = CustomersView;
    SortManager.SortShift(p, ref tmp);
}

```

In der Methode SortManager.Init(RoutedEventArgs p) werden durch die Variable p alle GridViewColumnHeaders abgespeichert. Der Grund dafür ist, dass bei dem Event SortShift keine EventArgs mitgegeben werden können, weil es sich um ein MouseBinding handelt und nicht um ein normales Event. Dadurch kann die Methode SortShift(object p, ref ICollectionView View) nicht wissen, welche Spaltenüberschrift gedrückt wurde und daher müssen am Anfang einmal alle GridViewColumnHeaders abgespeichert werden.

Wenn die Methode `SortNormal(RoutedEventArgs e, ref ICollectionView View)` aufgerufen wird, wird zunächst überprüft ob die Spaltenüberschrift schon einmal gedrückt wurde (dann soll nämlich die Sortierrichtung geändert werden). Wenn ja, werden die Suchrichtung sowie die Richtung des Pfeils neben der Spaltenüberschrift geändert. Ansonsten werden alle Pfeile neben den Überschriften gelöscht, die Suchrichtung auf aufsteigend gesetzt und ein neuer Pfeil angezeigt.

Ein Auszug der `SortNormal`-Methode:

```
//Same column pressed?
if (SortHeaders.Count == 1 && SortHeaders[0] == columnHeader)
{
    //Change sort direction
    dir = View.SortDescriptions[0].Direction;
    dir = dir == ListSortDirection.Ascending ? ListSortDirection.Descending :
        ListSortDirection.Ascending;
    header = ChangeArrow(columnHeader, dir, 0);
}
else
{
    //Remove arrow from old column header
    if (SortHeaders.Count > 0)
    {
        foreach (var item in SortHeaders)
        {
            item.Column.HeaderTemplate = null;
            item.Column.Width = item.ActualWidth - 20;
        }
    }
    SortHeaders.Clear();
    SortHeaders.Add(columnHeader);
    dir = ListSortDirection.Ascending; //default sort direction is ascending
    header = SetNewArrow(columnHeader, dir, 0);
}
View.SortDescriptions.Clear();
View.SortDescriptions.Add(new SortDescription(header, dir));
```

`SortHeaders` ist eine globale Variable vom Typ `List<GridViewColumnHeader>`, der die Spalten enthält, nach welchen aktuell sortiert wird. Die lokale Variable "dir" bezeichnet die gewünschte Sortierrichtung und ist vom Typ `ListSortDirection`. Der String "header" enthält die Property, auf die der `GridViewColumnHeader` bindet. Diese ist natürlich englisch und stellt wieder ein Übersetzungsproblem dar.

Wie schon weiter oben erwähnt, ist der Typ `ICollectionView` extra für das Darstellen von Listen gemacht, deshalb enthält er auch eine Eigenschaft namens `SortDescriptions`, in welche man beliebig viele `SortDescriptions` einfügen kann und nach welchen die Liste dann automatisch sortiert wird. In den Methoden `ChangeArrow` und `SetNewArrow` wird die Spalte entsprechend breiter gemacht und das passende vorgefertigte Template gesetzt. Dieses enthält selbst designete Bilder von Pfeilen, die unter den Ressourcen des

Projektes abgespeichert sind.

```
column.Column.HeaderTemplate = Application.Current.FindResource("ArrowUp") as
    DataTemplate;
```

In diesem Beispiel wird dem GridViewColumnHeader "column" ein Pfeil, der nach oben ausgerichtet ist, beigelegt.

In der Methode SortShift(object p, ref ICollectionView View) wird mittels dem Parameter "p" der Name der Property übergeben, auf die die Spalte bindet. Dieser wird händisch in der View übergeben (siehe oben) und ist englisch, weshalb er zuerst übersetzt werden muss. Danach wird der passende GridViewColumnHeader nach der Überschrift in der am Anfang angelegten Liste von GridViewColumnHeaders gesucht.

```
var columnHeader = AllHeaders.Where(h => String.Equals(h.Content.ToString(),
    germanColumnName)).ToList().FirstOrDefault();
```

Dann wird wieder überprüft, ob dieselbe Spalte zweimal hintereinander ausgewählt wurde, sodass dann die Sortierrichtung gewechselt werden kann. Ansonsten wird überprüft ob schon drei Spalten ausgewählt wurden und wenn nicht wird eine neue Spalte zu den Sortierspalten hinzugefügt. Auszug der SortShift-Methode:

```
if (View.SortDescriptions.Count >= 1)
{
    ListSortDirection dir;
    int index = View.SortDescriptions.Count - 1;
    //Change sorting direction
    if (View.SortDescriptions.Count == index + 1 &&
        View.SortDescriptions[index].PropertyName == columnName)
    {
        dir = View.SortDescriptions[index].Direction;
        dir = dir == ListSortDirection.Ascending ? ListSortDirection.Descending :
            ListSortDirection.Ascending;
        View.SortDescriptions.RemoveAt(index);
        View.SortDescriptions.Insert(index, new SortDescription(columnName, dir));
        ChangeArrow(columnHeader, dir, index);
        SortHeaders.Add(columnHeader);
    }
    else if (View.SortDescriptions.Count(s => s.PropertyName == columnName) ==
        0)
    {
        if (View.SortDescriptions.Count >= 3)
        {
            MessageBox.Show("Sie koennen maximal nach drei Spalten sortieren!",
                "Hinweis", MessageBoxButton.OK, MessageBoxImage.Exclamation);
            return;
        }
        dir = ListSortDirection.Ascending;
        SetNewArrow(columnHeader, dir, index+1);
    }
}
```

```

View.SortDescriptions.Add(new SortDescription(columnName, dir));
SortHeaders.Add(columnHeader);
}

```

3.2 Datenmodell

Wie auch der Rest der Diplomarbeit, wurde auch das Datenmodell des Administrationsprogramms und der Website strikt getrennt. Dabei gehören die Tabellen TODO .. der Website an und der Rest der Tabellen dem Administrationsprogramm.

Die Tabelle, die die meisten Felder enthält ist Tabelle der Aufträge (Order). In einem Datensatz dieser Tabelle kann entweder ein Brillenauftrag oder ein Kontaktlinsenauftrag enthalten sein, was mit dem Attribut 'OrderType' gekennzeichnet ist. Falls es sich um einen Brillenauftrag handelt nimmt 'OrderType' den Wert 'B' an, ansonsten 'K'. Je nach dem um welchen Auftrag es sich handelt, kann ein Glas- bzw. Kontaktlinsentyp mit dem Auftrag verknüpft werden. In jeder der beiden Tabellen wird nur der Name des Glastyps/Kontaktlinsentyps gespeichert. Mit dem Glastyp kann beispielsweise angegeben werden, ob es sich um eine Nah- oder Fernbrille handelt. Werte in der Tabelle Kontaktlinsentyp könnten 'Weich 6 Monate' oder 'Formstabil 12 Monate' sein.

Falls der Auftrag ein Brillenauftrag ist, kann zusätzlich auch eine Brillenfassung vermerkt werden. In der Tabelle werden auch sämtliche Werte der Gläser gespeichert, welche die meisten Attribute der Tabelle in Anspruch nehmen. Unter dem Attribut 'BillPath' bzw. 'OrderConfirmationPath' wird jeweils der Pfad zur letzten erstellten Rechnung/-Auftragsbestätigung gespeichert. Falls der Benutzer den Auftrag mit einem Augenarzt ('Doctor') in Verbindung bringen möchte, ist das ebenfalls möglich.

Jeder Auftrag muss einem Kunden zugewiesen werden. Die Tabelle Kunde enthält alle Grunddaten des Kunden, wobei Ort und Land in eigene Tabellen ausgelagert sind. Zusätzlich ist auch vermerkt, ob der Kunde Massennachrichten erhalten soll ('Newsletter') und ob der Kunde gelöscht wurde ('Deleted', siehe Kapitel 3.1.1).

Weiters gibt es noch die Tabellen Lieferant ('Supplier'), in der alle Informationen über einen Lieferanten gespeichert werden, wobei Ort und Land wieder in eigene Tabellen ausgelagert sind, und die Tabelle der Brillenfassungen ('EyeGlassFrame'). Diese enthält alle Informationen zu einer lagernden Brillenfassung.

Damit alle versendeten Nachrichten nach dem Versenden noch einmal angezeigt werden können, gibt es die Tabellen 'CustomMessage' und 'CustomRecipient'. In der Tabelle 'CustomMessage' wird das Datum sowie die Uhrzeit der Nachricht, der Text, der Typ der Nachricht (SMS oder E-Mail), der Betreff (falls es sich um eine E-Mail handelt) und die Id des Auftrages (falls es eine einzelne Nachricht ist) gespeichert. Eine solche Nachricht kann beliebig viele Empfänger haben. Die Empfänger der Nachricht müssen gesichert werden, da sonst im Nachhinein unklar ist, wer die Nachricht erhalten hat und an welche Telefonnummer bzw. E-Mail Adresse die Mitteilung versendet worden ist. Diese könnte ja seit dem Versenden der Nachricht verändert worden sein. Jeder der gespeicherten Empfänger muss einem Kunden in der Datenbank zugeordnet sein.

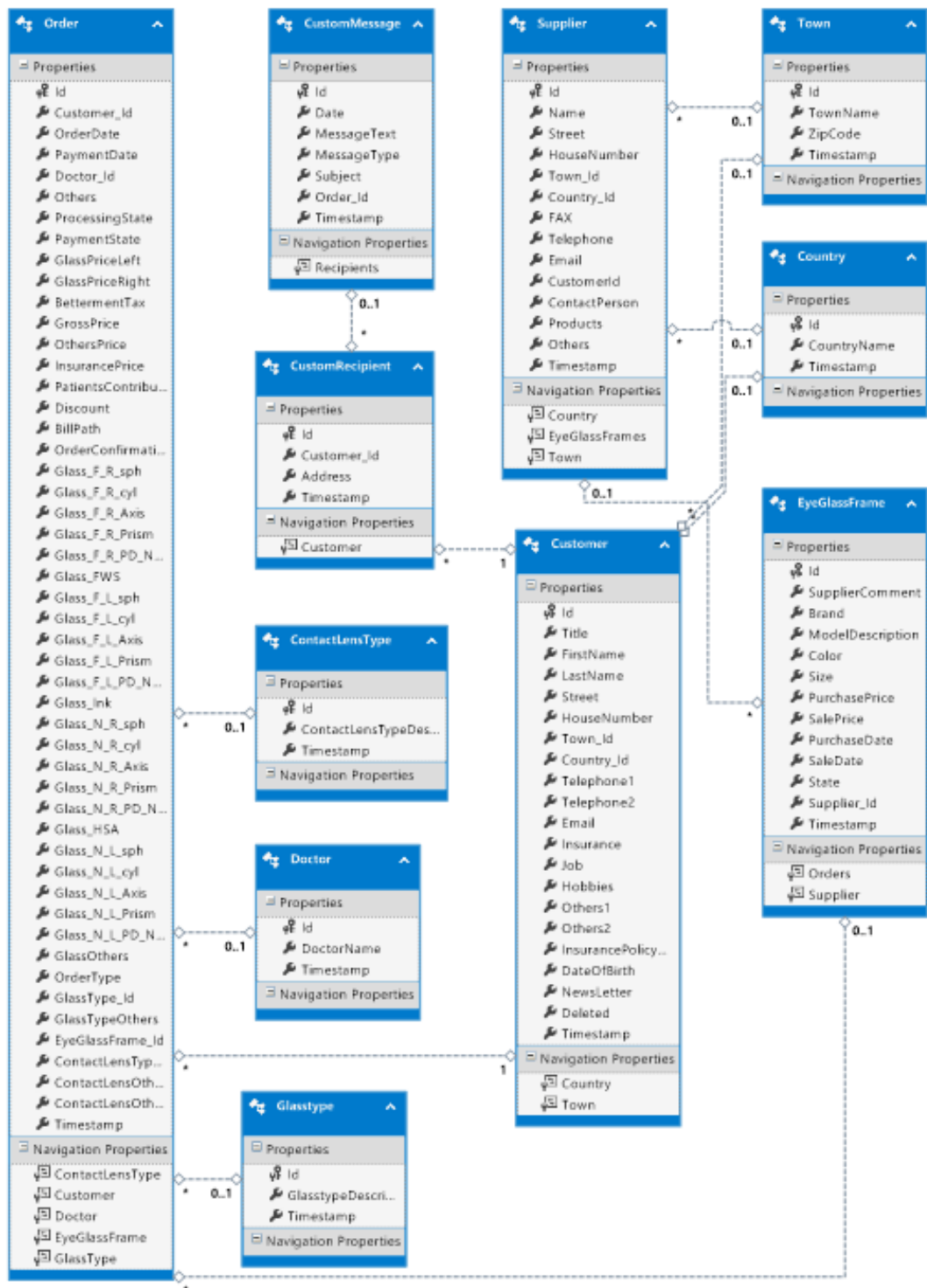


Abbildung 3.18: Datenmodell

3.3 Projektarchitektur

Um die Diplomarbeit so übersichtlich wie möglich zu halten, wurden fünf Projekte kombiniert.

- 'OpticianMgr.FillDb': Es handelt sich um eine Konsolenapplikation, welche den `ConnectionString` zur Datenbank enthält. Beim Start dieses Projektes wird der Inhalt der gesamten Datenbank gelöscht und mit Testdatensätzen gefüllt. Diese Daten werden zuvor in CSV-Files abgespeichert. Für jede Tabelle in der Datenbank gibt es ein CSV-File, welches eben solche selbst geschriebenen Testdatensätze beinhaltet. Ein Testdatensatz für einen Kunden könnte beispielsweise so aussehen: "Mag.;Susi;Sonne;Sonnenstraße;3;4020;Österreich;0650789456123;07242456123789;susi@gmail.com;Gebietskrankenkassa;Polizistin;Fußball;Arbeitet in Wels;Hat vormittags keine Zeit;7789031098;03.10.1998"

Die einzelnen Werte der Attribute werden durch Strichpunkte getrennt. Die Reihenfolge der Werte muss für alle Datensätze dieselbe sein, deshalb wird sie am Beginn des Files angeführt.

- 'OpticianMgr.Core': Dieses Projekt enthält alle Entitäten der Datenbank, einige Interfaces sowie eine Klasse, die das Einlesen und Einfügen der Testdatensätze in die Datenbank übernimmt.

In dem Unterordner "Entities" sind die Entitäten der Datenbank sowie die Klasse "EntityObject" enthalten. Wie im Kapitel 2.2 schon beschrieben, erben alle Entitäten von dieser Basisklasse, sodass jede Entität eine Id und einen Timestamp besitzt. Die Klasse "Town" wurde beispielsweise so implementiert:

```
namespace OpticianMgr.Core.Entities
{
    public class Town : EntityObject
    {
        [Required(ErrorMessage = "Bitte geben Sie einen Ortsnamen an!"),
        MaxLength(100, ErrorMessage = "Der Name des Ortes ist zu
        lange!")]
        public string TownName { get; set; }
        [Required(ErrorMessage = "Bitte geben Sie eine Postleitzahl an!"),
        MaxLength(15, ErrorMessage = "Die Postleitzahl ist zu lange!")]
        public string ZipCode { get; set; }
    }
}
```

Dieser Code gibt an, dass jeder Ort obligatorisch eine Bezeichnung und eine Postleitzahl haben muss. Außerdem darf die Ortsbezeichnung maximal 100 Zeichen haben und die Postleitzahl nur 15. Das ist für Fälle außerhalb von Österreich gedacht.

Zusätzlich beinhaltet dieses Projekt auch noch die Interfaces "IGenericRepository" und "IUnitOfWork". Deren Aufgaben wurden allerdings schon im Kapitel 2.2

beschrieben.

Zuletzt enthält das 'Core'-Projekt auch noch eine Klasse namens 'OpticianController'. Diese implementiert wiederum eine Methode 'FillDatabaseFromCsv()', welche die Testdatensätze der CSV-Files einliest, per LINQ konvertiert und in der Datenbank abspeichert. Um beispielsweise Testdatensätze für Orte einzulesen wird folgende Methode verwendet:

```
private List<Town> GetTowns()
{
    return GetStringMatrix("TestOrte.csv").Select(o =>
        new Town()
        {
            TownName = o[1],
            ZipCode = o[0]
        }
    ).ToList();
}
```

Die Methode 'GetStringMatrix' sucht im Projektverzeichnis nach einem File mit dem übergebenen Namen und gibt ein zweidimensionales String-Array mit den Daten zurück. Mittels Linq wird jede Zeile dieses Arrays dann in einen Ort verwandelt, dessen Postleitzahl die erste Spalte des Arrays ist und die Bezeichnung die Zweite.

- 'OpticianMgr.Persistence': Dieses Projekt beinhaltet die Klassen 'ApplicationDbContext', 'GenericRepository' und 'UnitOfWork'. Die beiden letzteren sind wesentlich Bestandteile des Data-Acces-Layers des UnitOfWork-Patterns und deshalb im Kapitel 2.2 gut beschrieben. Die Klasse 'ApplicationDbContext' beinhaltet alle DbSet's der Entitäten und den ConnectionString (Kapitel 2.1).
- 'OpticianMgr.Wpf': Diese Projekt führt das Administrationsprogramm der Diplomarbeit aus. Zu den dazu notwendigen Klassen zählen in erster Linie alle Views, also alle Fenster, die der Benutzer sieht, sobald er das Programm startet. Zu jeder dieser Views wurde ein ViewModel erstellt, welches den logischen Code zur View enthält. Neben den vielen ViewModels gehören dem Projekt auch alle Ressourcen und Einstellungen an, welche zur Ausführung der WPF-Applikation notwendig sind (siehe Kapitel 3.1.7 Vorgegebene Texte bearbeiten - User-Settings, Kapitel 3.1.8 Filtern-'ResourceManager', Kapitel 3.1.8 Sortieren - Bilder der Pfeile). Weiters enthält das Projekt Klassen wie den 'VieModelLocator', welcher Referenzen zu allen ViewModels zur Verfügung stellt (Kapitel 2.5). Außerdem gibt es noch den 'SortManager', der die Sortierung der Daten übernimmt (Kapitel 3.1.8) und den 'FileCreator', welcher zur Erstellung von Word-Dokumenten notwendig ist (Kapitel 3.1.2).

Um neue Fenster anzuzeigen und wieder zu schließen, wurden die Klasse 'WindowService' sowie die Interfaces 'IRequestClose' und 'IWindowService' benötigt.

Der WindowService enthält für jedes Fenster, das man aus dem Programm öffnen können soll, eine Methode, welche dann wiederum die generische Methode 'Show-Window' aufruft.

```
public void ShowWindow<TViewModel>(TViewModel viewModel, Window
    newWindow) where TViewModel : IRequestClose
{
    EventHandler<EventArgs> closeHandler = null;
    closeHandler = (sender, e) =>
    {
        viewModel.CloseRequested -= closeHandler;
        newWindow.Close();
    };
    viewModel.CloseRequested += closeHandler;
    newWindow.DataContext = viewModel;
    newWindow.ShowDialog();
}
```

Wichtig ist dabei, dass das ViewModel vom Interface IRequestClose ableitet, damit das ViewModel das Event CloseRequested enthält. Dieses dient dazu, dass das ViewModel, welches ein weiteres Fenster öffnet, auf das Schließen des neuen Fensters reagieren kann.

- 'OpticianMgr.Web': TODO

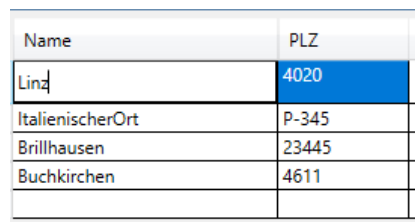
Kapitel 4

Selbstevaluation

4.1 Probleme bei der Entwicklung

4.1.1 Event-Handling mit MVVM-Light

Als Alternative zu den vielen einzelnen Fenstern für das Anlegen und Bearbeiten von Daten, wäre am Anfang eigentlich geplant gewesen, Datagrids zu benutzen. Ein Datagrid ist vergleichbar mit einer Excel-Tabelle. Das bedeutet, dass es vordefinierte Spalten gibt und die Daten in einer Tabelle darunter angezeigt werden.



Name	PLZ
Linz	4020
ItalienischerOrt	P-345
Brillhausen	23445
Buchkirchen	4611

Abbildung 4.1: Screenshot eines DataGrids

Während des Entwickelns des Programmes mit DataGrids sind allerdings einige Fragen aufgekommen. Die wichtigste war: Wann werden die Daten überprüft und in die Datenbank gespeichert? Die sinnvollste Antwort auf diese Frage wäre gewesen: Nachdem der Benutzer das Feld verlässt. Allerdings wäre dazu das Abonnieren eines Events aus der View ('RowEditEnding') mittels EventToCommand von MVVM-Light notwendig gewesen, doch das war auch nach unzähligen Lösungsversuchen aus unerklärlichen Gründen nicht möglich. Grundsätzlich gibt es zwei Möglichkeiten, um Events aus der View abzufangen:

- Events mittels Mouse- und KeyBindings auf Commands in das ViewModel übertragen. Für Events für Maus und Tastatur ist das eine sehr praktische Lösungsmöglichkeit, allerdings können damit keine anderen Events behandelt werden.

- Alle Events mittels MVVM-Light und der EventToCommand-Funktion ins View-Model übertragen (Kapitel 2.5).

Das 'RowEditEnding'-Event fällt leider in die zweite Kategorie, weshalb es nicht möglich war dieses Event zu behandeln. Als Alternative wurde versucht, die Daten erst nach dem Drücken der Enter-Taste in die Datenbank zu speichern. Das wäre möglich gewesen, weil dann auf ein Event der Tastatur mittels KeyBinding reagiert werden hätte können. Doch auch diese Variante war nicht optimal, denn sobald der Benutzer einmal vergessen hätte die Enter-Taste zu betätigen, wären die Daten nicht gesichert gewesen. Aus diesem Grund wurde dann ein ganz anderer Ansatz gewählt, nämlich der der ListViews. Damit konnte das Problem des Event-Handlings umgangen werden und diese Variante ist auch benutzerfreundlicher. So öffnet sich nun zum Anlegen sowie zum Bearbeiten eines Datensatzes ein neues Fenster.

In der späteren Entwicklungsphase haben sich die Probleme mit MVVM-Light und EventToCommand dann aus unerfindlichen Gründen gelöst und somit konnten alle Events verwendet werden. Das war insbesondere zum Sortieren wichtig (Kapitel 3.1.8). Dennoch blieb man aber bei der Variante der neuen Fenster für das Bearbeiten und Anlegen von Daten, da es sich als viel benutzerfreundlicher erwies.

4.1.2 UnitOfWork-Instanzen

In der frühen Entwicklungsphase der Diplomarbeit traten Probleme mit dem Speichern der Daten in der Datenbank auf. Auf unerklärliche Weise wurden veränderte Daten schon vor dem eigentlichen Speichern in der UnitOfWork-Instanz geändert und somit konnten Veränderungen nie rückgängig gemacht werden.

Beispiel: Der Benutzer öffnet einen Kunden und möchte den Vornamen von 'Max' auf 'Maxi' ändern. Bevor er die Änderung speichert, beschließt er allerdings, dass er die Änderung doch nicht vornehmen möchte und klickt statt 'Speichern' auf 'Abbrechen'. Wenn er nun zurück auf der Startseite ist, wurde der Vorname aber trotzdem auf 'Maxi' geändert.

Später stellte sich heraus, dass der Grund dafür war, dass im Programm nur eine UnitOfWork-Instanz verwendet wurde und immer nur mit Referenzen von Objekten in der UnitOfWork-Instanz gearbeitet wurde. Lösungen für das Problem wären gewesen für jeden Datenbankzugriff eine neue UnitOfWork-Instanz zu erzeugen oder alle Daten, die veränderbar sein sollten, vorher lokal kopiert. Schlussendlich wurde entschieden weiterhin nur eine UnitOfWork-Instanz zu verwenden und alle veränderbaren Daten vorher lokal zu kopieren. Die andere Möglichkeit, bei der für jeden Datenbankzugriff eine neue UnitOfWork-Instanz erzeugt worden wäre, hätte den Code um einiges verlängert und unübersichtlicher gemacht. Dazu hätte für jeden Zugriff ein Using-Block verwendet werden müssen:

```
using(UnitOfWork uow = new UnitOfWork())
{
    Customer cus = uow.CustomerRepository.GetById(1);
}
```

Literaturverzeichnis

- [Cod18] CODEPROJECT: *Unit of Work Design Pattern - CodeProject*. <https://www.codeproject.com/Articles/581487/Unit-of-Work-Design-Pattern>. Version: Februar 2018
- [csh18] CSHARPCORNER: *Unit of Work in Repository Pattern*. <https://www.c-sharpcorner.com/uploadfile/b1df45/unit-of-work-in-repository-pattern/>. Version: März 2018
- [dof18] DOFACTORY: *.NET Design Patterns in C# and VB.NET - Gang of Four (GOF) - doFactory.com*. <http://www.dofactory.com/net/design-patterns>. Version: März 2018
- [dot18a] DOTNETCURRY: *Using MVVM Light in WPF for Model-View-ViewModel implementation*. <http://www.dotnetcurry.com/wpf/1037/mvvm-light-wpf-model-view-viewmodel>. Version: März 2018
- [dot18b] DOTNETPATTERN: *MVVM Light Messenger*. <http://dotnetpattern.com/mvvm-light-messenger>. Version: März 2018
- [Gem18] GEMBOX: *Microsoft Office Interop (Word Automation) in C# - GemBox.Document*. [/document/articles/c-sharp-microsoft-office-interop-word-automation](http://document/articles/c-sharp-microsoft-office-interop-word-automation). Version: Februar 2018
- [IV18] IT-VISIONS: *Was ist WPF? - WPF im Vergleich zu Windows Forms*. <https://www.it-visions.de/lserver/artikeldetails.aspx?b=3718>. Version: Februar 2018
- [Mes18a] MESSAGEBIRD: *Preise*. <https://www.messagebird.com/de-de/tarife>. Version: März 2018
- [Mes18b] MESSAGEBIRD: *SMS*. <https://www.messagebird.com/de-de/sms>. Version: Februar 2018
- [Mic17a] MICROSOFT: *CollectionView.Filter-Eigenschaft (System.Windows.Data)*. [https://msdn.microsoft.com/de-de/library/system.windows.data.collectionview.filter\(v=vs.100\).aspx](https://msdn.microsoft.com/de-de/library/system.windows.data.collectionview.filter(v=vs.100).aspx). Version: Oktober 2017

- [Mic17b] MICROSOFT: *Windows Sorting a WPF ListView by clicking on the header (2) - Sort Direction Indicators sample in C#, XAML for Visual Studio 2012*. <https://code.msdn.microsoft.com/windowsdesktop/Sorting-a-WPF-ListView-by-5769086f>. Version: September 2017
- [Mic18a] MICROSOFT: *Entity Framework Connections and Models*. [https://msdn.microsoft.com/en-us/library/jj592674\(v=vs.113\).aspx](https://msdn.microsoft.com/en-us/library/jj592674(v=vs.113).aspx). Version: März 2018
- [Mic18b] MICROSOFT: *Entity Framework Database First*. [https://msdn.microsoft.com/en-us/library/jj206878\(v=vs.113\).aspx](https://msdn.microsoft.com/en-us/library/jj206878(v=vs.113).aspx). Version: März 2018
- [Mic18c] MICROSOFT: *LINQ to Entities*. <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/linq-to-entities>. Version: März 2018
- [Mic18d] MICROSOFT: *MVVM - Commands, RelayCommand and EventToCommand*. <https://msdn.microsoft.com/en-us/magazine/dn237302.aspx>. Version: März 2018
- [Sch18] SCHWICHTENBERG, Holger: *Vor- und Nachteile*. <https://www.heise.de/developer/artikel/Vor-und-Nachteile-228136.html>. Version: März 2018
- [sha17] SHARPCORNER C: *Charting in WPF*. <http://www.c-sharpcorner.com/uploadfile/mahesh/charting-in-wpf/>. Version: August 2017
- [Tut18] TUTORIAL, EntityFrameWork: *What is Entity Framework?* <http://www.entityframeworktutorial.net/what-is-entityframework.aspx>. Version: März 2018
- [twi18] *SMS Pricing for Text Messaging - Twilio*. <https://www.twilio.com/sms/pricing/at>. Version: April 2018
- [Wik16] WIKIPEDIA: *Objektrelationale Abbildung*. https://de.wikipedia.org/w/index.php?title=Objektrelationale_Abbildung&oldid=161037938. Version: Dezember 2016. – Page Version ID: 161037938
- [Wik17a] WIKIPEDIA: *Entity Framework*. https://de.wikipedia.org/w/index.php?title=Entity_Framework&oldid=170788827. Version: November 2017. – Page Version ID: 170788827
- [Wik17b] WIKIPEDIA: *Model View ViewModel*. https://de.wikipedia.org/w/index.php?title=Model_View_ViewModel&oldid=168867092. Version: September 2017. – Page Version ID: 168867092
- [Wik18] WIKIPEDIA: *LINQ*. <https://de.wikipedia.org/w/index.php?title=LINQ&oldid=173859405>. Version: Februar 2018. – Page Version ID: 173859405

Abbildungsverzeichnis

2.1	Einsatz des Entity Frameworks in einer Applikation	4
2.2	Einfaches WPF-Fenster	8
2.3	MVVM-Konzept	9
3.1	Screenshot der Kundenverwaltung	15
3.2	Screenshot Neuen Kunden anlegen	16
3.3	Screenshot der Kundendetails	17
3.4	Screenshot der Auftragsverwaltung	18
3.5	Screenshot eines Brillenauftrags	19
3.6	Screenshot der Lieferantenverwaltung	22
3.7	Screenshot der lagernden Brillenfassungen	23
3.8	Screenshot der Massen E-Mails	24
3.9	Screenshot der Massen SMS	25
3.10	Screenshot der einzelnen Nachricht	26
3.11	Screenshot der versendeten Nachrichten	27
3.12	Screenshot der Statistiken	28
3.13	Screenshot der Länder	29
3.14	Land bearbeiten	29
3.15	Vorgegebene Texte bearbeiten	30
3.16	Screenshot des Filters	32
3.17	Screenshot des Sortierens	35
3.18	Datenmodell	40
4.1	Screenshot eines DataGrids	44
A.1	Generierte Auftragsbestätigung	51
A.2	Generierte Rechnung	52

Tabellenverzeichnis

Anhang A

Generierte Dokumente des Verwaltungsprogrammes



Max Mustermann
Musterstraße 45
23445 Brillhausen

11.03.2018

Auftragsbestätigung

Ihre Bestellung/Ihr Auftrag vom 08.03.2018

Sehr geehrter Kunde,

vielen Dank für Ihren Auftrag (Nr. 5 , Sportbrille). Wir haben den Auftrag erhalten und werden uns so bald als möglich darum kümmern.

Leistung	Preis inkl. MwSt.
Glas links	50,00 EUR
Glas rechts	50,00 EUR
Brillenfassung	300,45 EUR
Sonstiges	5,00 EUR
Versicherungsgeld	-0,00EUR
Selbstbehalt	0,00 EUR
Rabatt	-0,00 EUR

Gesamtbetrag: 405,45EUR , davon MwSt: 67,57EUR

Bei Rückfragen stehen wir selbstverständlich jederzeit gerne zur Verfügung.

Mit freundlichen Grüßen

Augenoptik Aigner

Augenoptik Aigner
Kaiser Josef Platz 3
4600 Wels
Österreich
Tel.: 07242 22 43 84
E-Mail: office@augenoptik-aigner.at

Sparkasse OÖ
IBAN: AT64 2032 0321 0007 1003
BIC: ASPK AT2L XXX

USt-ID: ATU 63 84 12 03
Geschäftsführer:
Wolfgang Aigner

Abbildung A.1: Generierte Auftragsbestätigung

Max Mustermann
Musterstraße 45
23445 Brillhausen



Rechnung

Rechnung Nr. 3

Kunden-Nr. 6

01.02.2018

Leistung	Preis inkl. MwSt.
Glas links	50,00 EUR
Glas rechts	55,00 EUR
Brillenfassung	300,45 EUR
Sonstiges	3,00 EUR
Versicherungsgeld	-10,00EUR
Selbstbehalt	7,00 EUR
Rabatt	-9,00 EUR

Gesamtbetrag: 396,45EUR
davon MwSt: 66,07EUR

Doktorname: Dr. med. Barbara Huber
Typ: Bildschirmbrille

Augenoptik Aigner
Kaiser Josef Platz 3
4600 Wels
Österreich
Tel.: 07242 22 43 84
E-Mail: office@augenoptik-aigner.at

Sparkasse OÖ
IBAN: AT64 2032 0321 0007 1003
BIC: ASPK AT2L XXX

USt-ID: ATU 63 84 12 03
Geschäftsführer:
Wolfgang Aigner

Abbildung A.2: Generierte Rechnung