



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	2
1.1	Ist-Situation	2
1.2	Zielsetzung	2
1.3	Aufgabenstellung	2
2	Verwendete Technologien	3
2.1	Entität Framework	3
2.1.1	Linq	3
2.2	UnitOfWork Pattern	4
2.3	WPF	5
2.4	MVVM	6
2.5	MVVM-Light	7
2.6	Microsoft Office Interop	9
2.7	WPF-Toolkit	10
2.8	MessageBird	10
3	Administrationsprogramm und Website	12
3.1	Administration	12
3.1.1	Kundenverwaltung	12
3.1.2	Auftragsverwaltung	15
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	29
3.2	Datenmodell	37
3.3	Projektarchitektur	37
4	Selbstevaluation	38
5	Zusammenfassung	39
A	Additional Information	43

Kapitel 1

Einleitung

1.1 Ist-Situation

1.2 Zielsetzung

1.3 Aufgabenstellung

Kapitel 2

Verwendete Technologien

2.1 Entität Framework

[noa17] (Referenz msdn) Dabei handelt es sich um ein Framework zur Erstellung von objektrelationalen Abbildungen (ORM – object relational mapping) auf .NET-Objektstrukturen. Dieses Framework wurde von Microsoft entwickelt.

Zum Modellieren einer Datenbank gibt es zwei Ansätze, nämlich Code First und Model First. Zur Erstellung der verwendeten Datenbank wurde die erste Methode gewählt, nämlich Code First. Dazu werden die Klassen im Programm zuerst definiert und darauf basierend wird vom DbContext die Datenbank erzeugt. Als Alternative bietet sich Model First an, wobei zuerst die Entity-Klassen mit Hilfe eines grafischen Tools modelliert werden und darauf basierend ein Datenbankschema erstellt wird. Der wohl wichtigste Bestandteil des Entity Frameworks ist eine Klasse, die von DbContext abgeleitet wird. Diese enthält DbSet aller Datenbankklassen und gibt mittels des Connectionstrings an, welche Datenbankverbindung verwendet werden soll.

Ein DbSet ist eine Klasse, die die entsprechenden Methoden für Entitätstypen anbietet.

2.1.1 Linq

[?] Referenz (docs.microsoft.com) 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 von 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 öfters 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 Kontaktlinsen aufträge 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"));
```

2.2 UnitOfWork Pattern

[?] (Referenz dofactory, c-sharpcorner) 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 der Datenbank sondern mit der UnitOfWork.

Genauer gesagt, muss der Programmierer zuerst das Repository Pattern implementieren, um das UnitOfWork Pattern umzusetzen. 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, in dem 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, welches die Standardmethoden wie Get, Insert oder Delete vorschreibt.
- **IUnitOfWork**: Ein Interface, welches die Definition 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 vom 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 auch gearbeitet.

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

```
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.

2.3 WPF

[?] (Referenz heise.de) 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

Element. Hier ist ein einfaches Beispiel eines XAML-Codes und dessen Erscheinen aufgeführt:

```
<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>
```

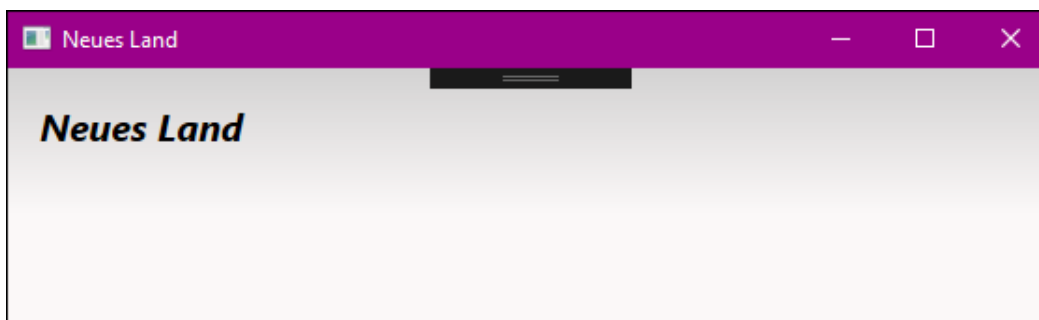


Abbildung 2.1: Einfaches WPF-Fenster

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

2.4 MVVM

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

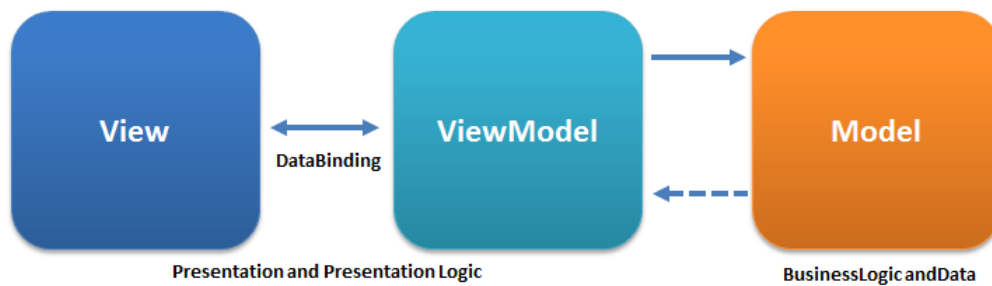


Abbildung 2.2: MVVM-Konzept

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. Das hat wiederum den Vorteil, dass die Darstellung von Designern und die Logik von Entwicklern erstellt werden kann.
- Durch die Trennung ergibt sich eine bessere Testbarkeit der Logik.

2.5 MVVM-Light

(Referenz dotnetcurry, dotnetpattern, msdn) 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 ViewModelBase.
- ViewModelLocator: Beinhaltet statische Referenzen für alle anderen ViewModels. Außerdem bietet die Klasse einen einfachen IOC-Container an.

Alle ViewModels, die später erstellt werden, sollten von 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" an, 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 RelayCommands 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 weitergegeben werden und dort bearbeitet werden. Dazu müssen zusätzlich werden 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 "MouseDoubleClick" vom RelayCommand "OpenCustomer" abonniert.

```
<i:Interaction.Triggers>
  <i:EventTrigger EventName="MouseDoubleClick">
    <cmd:EventToCommand Command="{Binding OpenCustomer}"></cmd:EventToCommand>
  </i:EventTrigger>
</i:Interaction.Triggers>
```

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.

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 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.(Referenz Gembox) 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 Vor-

lage 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.

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.

2.8 MessageBird

(Referenz Website einbinden, [messagebird.com](https://www.messagebird.com)) MessageBird ist ein Unternehmen, welches einen Online-Dienst anbietet mit dem Unternehmen oder auch einzelne Personen, kostenpflichtig SMS aus einem Programm aus zu 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 Key, mit dem dann echte SMS versendet wer-

den. Für Nachrichten, die nach Österreich gesendet werden, muss der Benutzer derzeit 4,6 Cent (Stand März 2018) zahlen. 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 versendete 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.

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. 6 Cent gekostet und es wäre ein monatlicher Betrag zur Benutzung einer Nummer angefallen. Aus Kostengründen und aus Benutzerfreundlichkeit ist die Wahl des Anbieters auf MessageBird gefallen.

Kapitel 3

Administrationsprogramm und Website

3.1 Administration

3.1.1 Kundenverwaltung

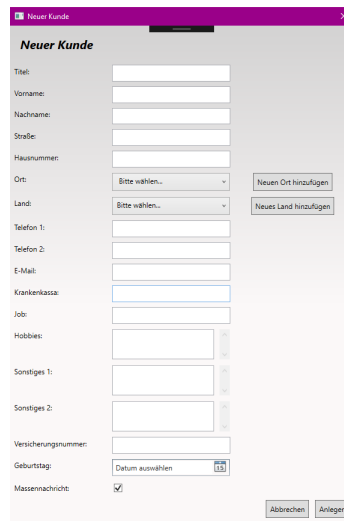
Um dem Benutzer eine kompakte Übersicht über seine Kunden zu geben, gibt es die Kundenverwaltung, bei der alle Kunden in einer Liste dargestellt werden. Angezeigt wird, die Id, Titel, Vorname, Nachname, Straße, PLZ, Ort, Telefon1 und das Land des Kunden. Diese Liste kann man filtern und sortieren (siehe Kapitel 'Filtern und Sortieren'). 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.



The screenshot shows a web form titled "Neuer Kunde" (New Customer). The form contains the following fields and controls:

- Titel:** Text input field.
- Vorname:** Text input field.
- Nachname:** Text input field.
- Straße:** Text input field.
- Hausnummer:** Text input field.
- Ort:** Dropdown menu with "Bitte wählen..." and a button "Neuen Ort hinzufügen" (Add new location).
- Land:** Dropdown menu with "Bitte wählen..." and a button "Neues Land hinzufügen" (Add new country).
- Telefon 1:** Text input field.
- Telefon 2:** Text input field.
- E-Mail:** Text input field.
- Krankenkasse:** Text input field.
- Job:** Text input field.
- Hobbies:** Text input field.
- Sonstiges 1:** Text input field.
- Sonstiges 2:** Text input field.
- Versicherungsnummer:** Text input field.
- Geburtsdag:** Date picker with "Datum auswählen" (Select date) and a calendar icon.
- Massenachrichte:** Checkmark.
- Buttons:** "Abbrechen" (Cancel) and "Anlegen" (Create) at the bottom right.

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.

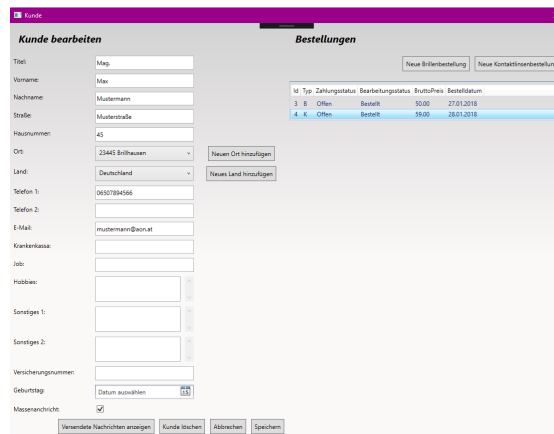


Abbildung 3.3: Screenshot der Kundendetails

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 Filtern und Sortieren). 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 macht werden, weil sie ja immer sofort in die "UnitOfWork" übertragen werden würden.

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

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 auf zurück auf der Startseite ist, wurde der Vorname aber trotzdem auf 'Maxi' geändert. Das würde passieren, wenn zuvor keine lokale Kopie erstellt worden wäre und somit der alte Vorname überschrieben worden wäre.

Eine andere Möglichkeit wäre, für jeden Datenbankzugriff eine neue Instanz von "UnitOfWork" in einem Using-Block zu benutzen. Dann müsste man nur eine Referenz auf einen Kunden erstellen und könnte trotzdem den vorherigen Zustand wiederherstellen.

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

Dennoch hat auch diese Vorgehensweise Nachteile, denn dann müsste wirklich jeder einzelner Datenbankzugriff von einem Using-Block umgeben sein und das würde den Code wesentlich verlängern und unverständlicher machen.

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 eine Brillenfassung und der Kontaktlinsenauftrag nicht. Außerdem hat ein Brillenauftrag einen Brillentyp und ein Kontaktlinsenauftrag einen Kontaktlinsentyp. Damit kann der Benutzer 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. 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						
Id	Nachname	Bearbeitungsstatus	Zahlungsstatus	Bruttopreis	Bestelldatum	Glasytbezeichnung	Id	Nachname	Bearbeitungsstatus	Zahlungsstatus	Bruttopreis	Bestelldatum	Kontaktlinsentypbeschre
1	Test	Bestellt	Bezahlt	0.00			2	Pürmayr	Bestellt	Offen	0.00		
3	Mustermann	Bestellt	Bezahlt	396.45	27.01.2018	Bildschirmbrille	4	Mustermann	Bestellt	Offen	59.00	28.01.2018	

Abbildung 3.4: Screenshot der Auftragsverwaltung

Wenn der Benutzer doppelt auf einen Auftrag klickt, erscheint entweder ein Detailfenster eines Brillenauftrags 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.

Brillenbestellung ändern

Kunde: Max Mustermann
 Glasyt: Bildschirmbrille
 Glasytsonstiges:
 Brillenfassung: 123 GUESS
 Doktor: Dr. med. Barbara Huber
 Zahlungsdatum: 29.01.2018
 Bestelldatum: 27.01.2018
 Zahlungsstatus: Bezahlt
 Bearbeitungsstatus: In Werkstatt
 Sonstiges:

Linker Glaspreis: 50.00
 Rechter Glaspreis: 55
 Preis von Sonstigem: 3
 Krankenkassageld: 12
 Selbstbehalt: 3
 Rabatt: 7
 Brillenfassung: 300.45 €
 Brutto: 396.45 €
 Mehrwertsteuer: 65.74 €

Berechnen

sph cyl Achse Prima PD/NTH FWS/ Ink HSA

R
L
R
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 das alte Dokument wirklich überschrieben 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 so aussehen:



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

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 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 Arten von Feldern in Word, jedoch sind in der Vorlage alle zu bearbeitenden Felder sogenannte Mergefields). Dann wird die gesuchte Property (Code 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 so aussehen könnte:

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

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 (siehe Kapitel 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 3.8: 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 3.9: Screenshot der lagernden Brillenfassungen

3.1.5 E-Mail und SMS

Massennachrichten

Um regelmäßige Info- und Werbenachrichten auszusenden, bietet das Programm die Möglichkeit E-Mails oder SMS an alle Kunden zu versenden. Falls ein Kunde diese Nachrichten nicht mehr erhalten möchte, kann das der Benutzer bei dem einzelnen Kunden 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 Adresse des Kunden verwendet. Falls keine E-Mail-Adresse angegeben wurde, wird eine entsprechende Fehlermeldung angezeigt.

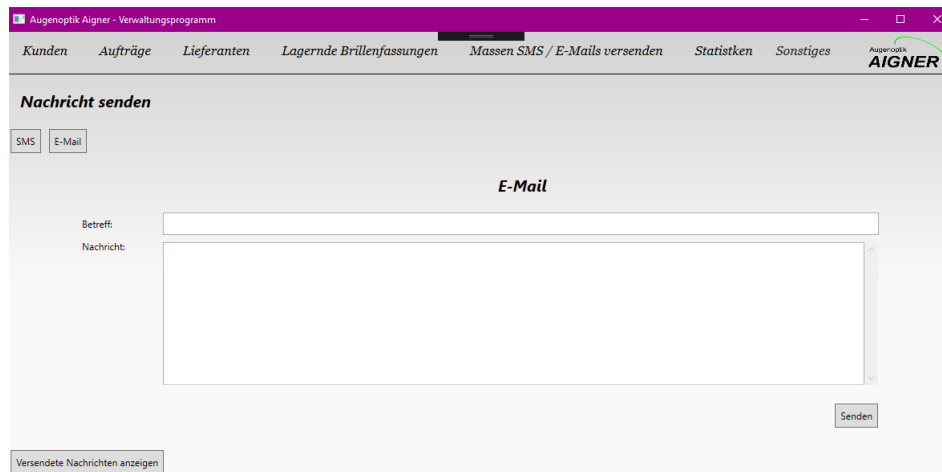


Abbildung 3.10: Screenshot der Massen E-Mails

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.augenoptikaigner@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.augenoptikaigner@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.

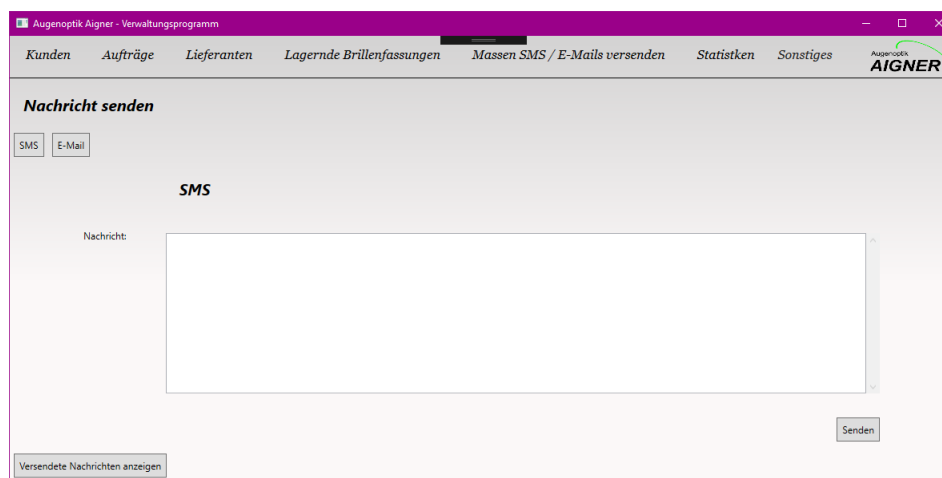


Abbildung 3.11: Screenshot der Massen SMS

Diese Nachricht wird dann an alle Kunden gesendet, außer jene, bei denen eingetragen 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.

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. Sollte dies nicht der Grund sein, warum eine Nachricht gesendet werden soll, kann der Benutzer die Nachricht natürlich auch verändern.

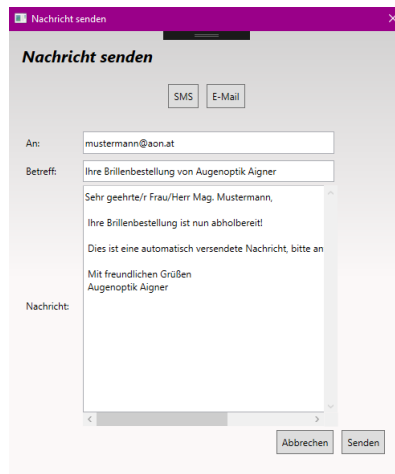


Abbildung 3.12: Screenshot der einzelnen Nachricht

Versendete Nachrichten

Außerdem ist es möglich, alle Nachrichten, die vom System aus 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();
```

Versendete Nachrichten						
Datum	Empfänger	Betreff	Nachricht	Auftrags Id	Typ	
04.03.2018 22:56:08	Pürmayr Mustermann evapuermayr@gmail.com 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 3.13: Screenshot der versendeten Nachrichten

3.1.6 Statistiken

Unter dem Menüpunkt "Statistiken" erhält der Benutzer eine Übersicht über alle verkauften Brillen und Kontaktlinsen. Dazu wird ein Liniendiagramm der verkauften Brillen/Kontaktlinsen von diesem Jahr und dem Jahr davor angezeigt. Damit ein Brillen/-Kontaktlinsenauftrag in der Statistik mitberücksichtigt wird, muss ein Zahlungsdatum angegeben werden und der Bezahlungsstatus muss auf „Bezahlt“ gesetzt werden.

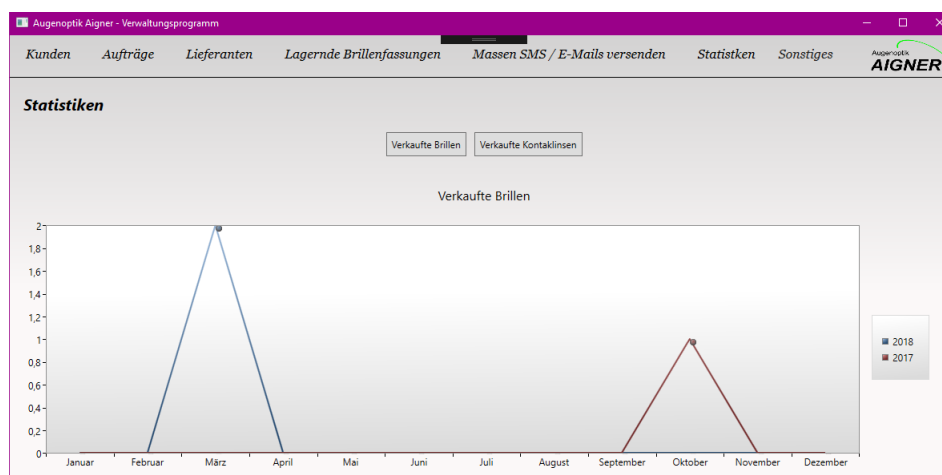


Abbildung 3.14: Screenshot der Statistiken

Technischer Hintergrund

Zur Darstellung wurde das Wpf-Toolkit verwendet (siehe Kapitel 2.7). Um ein Liniendiagramm zu erzeugen:

```
<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>
```

Dabei sind "NewValues" und "OldValues" vom Typ:

```
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 vier Unterpunkte:

- Orte bearbeiten
- Länder bearbeiten
- Brillentypen bearbeiten
- Kontaktlinsentypen bearbeiten

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

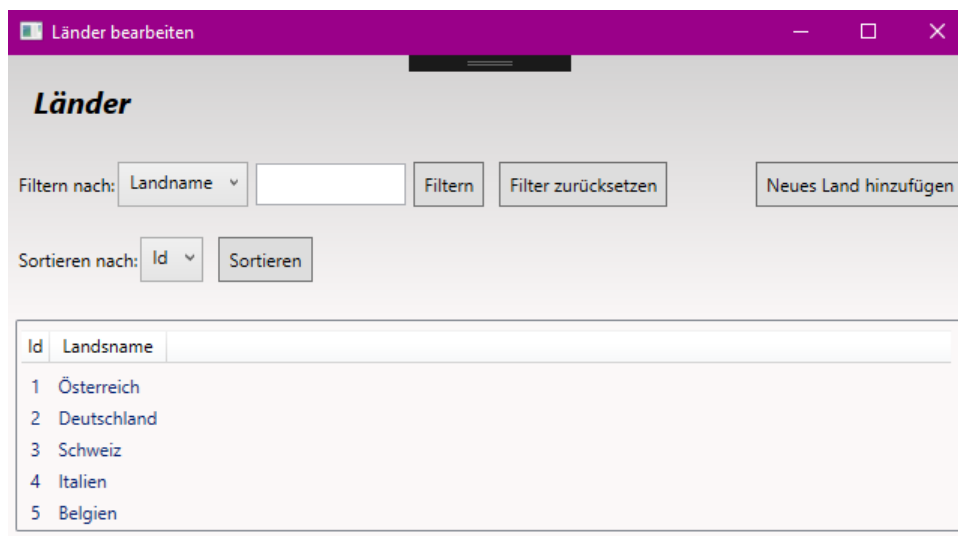


Abbildung 3.15: Screenshot der Länder

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.

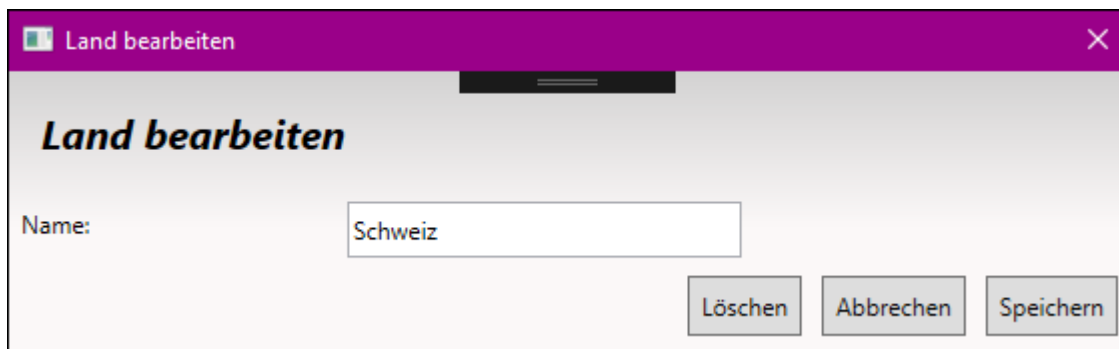


Abbildung 3.16: Land bearbeiten

3.1.8 Filtern und Sortieren

Filtern

Auf allen Hauptseiten der Applikation (Kunden, Brillen- und Kontaktlinsenaufträge, Lieferanten, Lagernde Brillenfassungen) sowie auf den Seiten unter dem Menüpunkt „Son-

stiges“ (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“ 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.

Id	Name	PLZ	Ort	Straße	Hausnummer	Land	FAX	Telefon
1	Silhouette	4020	Linz	Silhouettestraße	3	Österreich	0049-711-123456	07242 206456
2	Luxottica	P-345	Italienischer Ort	Luxotticaweg	345	Italien	0047/789456232	07242 456789
3	Charmant	23445	Brillhausen	Charmantgasse	32	Deutschland	456456464	5465454

Abbildung 3.17: 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 in Englisch 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 in Deutsch 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 auf Englisch 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. Durch die Methode „Contains“ wird dieser auch gesetzt. Der folgende Code stammt aus dem ViewModel der lagernden Brillenfassung. 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 keiner dieser Fälle zutrifft, 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 Spaltenübersicht, 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 die 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, dann nach Ort und zum Schluss nach Nachnamen sortiert. Zur besseren Übersichtlichkeit zeigt das Programm einen normalen Pfeil oder einen Pfeil mit einem oder zwei Punkten, 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 3.18: Screenshot des Sortierens

Zur Implementierung gibt es eine Klasse SortManager, die die Sortierung für alle Hauptseiten regelt.

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

Dazu wird im Konstruktor ein neuer SortManager initialisiert.

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

Zusätzlich werden drei Events von der View abonniert. Dieses Event 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 abonniert.

```
<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 GridViewColumnHeader 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 gesetzt. 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);
    //default sort direction is ascending
    dir = ListSortDirection.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 SortDescriptions, in welche man beliebig viele SortDescriptions einfügen kann und nach welchen die Liste automatisch sortiert wird. In den Methoden ChangeArrow und SetNewArrow wird die Spalte

entsprechend breiter gemacht und das passende vorgefertigte Template gesetzt.

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

In diesem Beispiel wird dem GridViewColumnHeader column ein Pfeil der nach oben ausgerichtet ist beigegefügt. In der Methode SortShift(object p, ref ICollectionView View) wird mittels dem Parameter p der Name der Property übergeben, an die sich 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 den am Anfang angelegten 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

3.3 Projektarchitektur

Kapitel 4

Selbstevaluation

Kapitel 5

Zusammenfassung

Literaturverzeichnis

[noa17] Entity Framework, November 2017. Page Version ID: 170788827.
URL: https://de.wikipedia.org/w/index.php?title=Entity_Framework&oldid=170788827.

Abbildungsverzeichnis

2.1	Einfaches WPF-Fenster	6
2.2	MVVM-Konzept	7
3.1	Screenshot der Kundenverwaltung	12
3.2	Screenshot Neuen Kunden anlegen	13
3.3	Screenshot der Kundendetails	14
3.4	Screenshot der Auftragsverwaltung	16
3.5	Screenshot eines Brillenauftrags	16
3.6	Generierte Auftragsbestätigung	18
3.7	Generierte Rechnung	21
3.8	Screenshot der Lieferantenverwaltung	22
3.9	Screenshot der lagernden Brillenfassungen	23
3.10	Screenshot der Massen E-Mails	24
3.11	Screenshot der Massen SMS	25
3.12	Screenshot der einzelnen Nachricht	26
3.13	Screenshot der versendeten Nachrichten	27
3.14	Screenshot der Statistiken	27
3.15	Screenshot der Länder	29
3.16	Land bearbeiten	29
3.17	Screenshot des Filters	30
3.18	Screenshot des Sortierens	33

Tabellenverzeichnis

Anhang A

Additional Information