BACHELORARBEIT

im Studiengang Informatik

Line of Business Applikationen auf WinRT

Ausgeführt von: Felix Wagner
Personenkennzeichen: 1010257048

Begutachter: Dipl.-Ing. (FH) Arthur Michael Zaczek

Wien, 19. Mai 2013

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht."

Ort, Datum

Unterschrift

Kurzfassung

Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text ...

Schlagwörter: Schlagwort 1, Schlagwort 2, Schlagwort 3, Schlagwort 4, Schlagwort 5

Abstract

Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text ...

Keywords: Keyword 1, Keyword 2, Keyword 3, Keyword 4, Keyword 5

Danksagung

Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text ...

Inhaltsverzeichnis

1. Einleitung

Der Verkauf von klassischen Desktop PCs ist vor allem in der letzten Monaten stark gesunken, und der Trend nach unten hält weiter an. Laut einer Studie des IT Marktforschungunternehmens International Data Corporation (IDC) gingen die Verkäufe der klassischen Rechenmaschinen im zweiten Quartal 2013 im Vergleich zum selben Quartal des letzten Jahres im IMEA-Raum (Europa, Mittlerer Osten und Afrika) um 20,2 Prozent zurück.
Dies ist allerdings nicht darauf zurückzuführen, dass die Menschen technologieverdrossener werden, tatsächlich ist die Anzahl der Geräte pro Person gewachsen: im Jahr 2012 sind die Ausgaben für technische Geräte um 15 Prozent gestiegen, und auch für 2013 wird eine ähnlich hohe Zahl erwartet.
Erklären lässt sich diese Diskrepanz vor allem anhand der stark erhöhten Verkaufszahlen von Tabletcomputern. Einfache Aufgaben im Konsumentenbereich wie Internet surfen oder Emails verfassen, die man damals an einem PC durchgeführt hat, werden heute lieber über die viel mobileren Geräte durchgeführt.
Klassische Businessanwendungen und aufwändigere Programme werden allerdings derzeit immer noch zum Großteil auf den gewohnten stationären Geräten verwendet. Wenn Produktivität im Vordergrund steht, ist die übliche Eingabe mit Maus und Tastatur der neuen Inputstrategie mit Touch und Gesten noch weit überlegen. [1]

In den nächsten Jahren könnte sich jedoch auch diese Tatsache durch die Optimierung von Line of Business (LOB) Applikationen auf mobile Geräte wie Tablets den veränderten Bedingungen anpassen.
Microsoft hat den Trend erkannt, diesen in der Entwicklung seines neuen Betriebssystemes berücksichtigt, und zwei Versionen auf den Markt gebracht: Windows 8 und Windows RT. Die erste davon, Windows RT, ist nur auf ARM-Prozessoren verwendbar, auf denen die aktuellen Windows Tablets aufbauen. Die Oberfläche wurde dafür im Vergleich zum Vorgänger Windows 7 komplett überarbeitet, Abwärtskompatibilität zu klassischen Desktopapplikationen fehlt.
Die zweite Variante versucht die Vorteile von beiden Welten - PC und Tablet - zu vereinen. Gestartet wird auf die selbe Oberfläche wie bei der auf ARM basierenden Version, es kann allerdings bei Bedarf jederzeit auf den Desktop gewechselt werden und auch die gewohnten Anwendungen von vorherigen Windowsversionen sind auf dieser Version voll lauffähig. Wie lange es diesen klassischen Desktop noch geben wird, ist allerdings ungewiss und ist stark von der Akzeptanz des neuen Designs der Kunden abhängig. Microsoft wagt hier den ersten Schritt und bietet in seiner Datenverarbeitungssoftware Office „für die Fingereingabe optimierte Desktopversionen von Word, PowerPoint, Excel und OneNote“an, wobei diese teilweise noch nicht die volle Funktionalität inkludieren. [2]

Herausforderungen gibt es dabei einige, vor allem in der Bedienung durch Touch-Eingabe. Klassische Controls, ursprünglich für Mauszeiger gedacht, müssen überarbeitet und für die Eingabe mit den Fingern optimiert werden. Diese doch beträchtlichen Änderungen werden nicht von heute auf morgen durchführbar sein, und wenn man den Windows Store (der Marktplatz

für Anwendungen auf der neuen Plattform) nach LOB Anwendungen durchsucht, wird man im Moment noch kaum fündig.
Ob und wie sich diese also auf mobilen Plattformen durchsetzen werden, wird in den nächsten Monaten und Jahren spannend zu beobachten sein.

Diese Bachelorarbeit fokussiert sich vor allem auf zwei Bereiche bei der Entwicklung von Line of Business Applikationen: auf das Layout und auf die vorhandenen Application Programming Interfaces (APIs) von WinRT Applikationen, in die sich Entwickler einhängen können.

WinRT - die neu entwickelte Windows Runtime - ist hierbei keinesfalls mit Windows RT zu verwechseln. Während Windows RT wie bereits erwähnt die auf ARM basierende Version des Betriebssystems Windows bezeichnet, ist WinRT die neue Runtime, auf der die touch-optimierten Apps aufsetzen und sowohl auf Windows RT als auch auf Windows 8 Geräten verwendet werden können. Windows 8 hingegen ist der offizielle Produktname für die x86 und x64 Versionen von der aktuellen Windows-Version. Im zweiten Kapitel wird anhand von Grafiken eine Erklärung gegeben, in welcher Beziehung die neuen APIs zu den bisherigen von .NET stehen. [3]

Im Bereich Layout, der das dritte Kapitel einnimmt, wird die Oberflächenbeschreibungssprache Extensible Application Markup Language (XAML) beschrieben, bevor auf verschiedene Eigenheiten von WinRT Applikationen eingegangen wird. Unter anderem erwähnt werden hier verschiedene Controls, der Splash Screen und die unterschiedliche Darstellung von Apps im Portrait oder Landscape Mode.

Ziel des vierten Kapitels ist es, dem Leser einen Überblick über die verfügbaren Schnittstellen und die so genannten Charms zu geben, die Microsoft mit den neuen Betriebssystemversionen eingeführt hat. Kurz gesagt bezeichnet diese eine neue applikationsübergreifende Art von Menüleiste.

Sschlussendlich wird die erklärte Theorie anhand eines einfachen Beispiels in die Praxis umgesetzt.

Prinzipiell ist es möglich, mit verschiedensten Programmiersprachen eine Applikation für WinRT Geräte zu entwickeln. Diese Bachelorarbeit konzentriert sich jedoch rein auf die Entwicklung mit C#.

2. Von .NET zu WinRT

Seitdem erstmals über WinRT, die neue Windows Runtime, berichtet wurde, gibt es Diskussionen über die unglückliche Namensgebung und die realistischerweise gegebene Verwechslungsgefahr zu Windows RT, die in der Einleitung erwähnt wurde.
Für weitere Ärgernisse sorgt eine von Microsoft erstellte Grafik, welche unter anderem die Unterschiede zu .NET-Applikationen darstellen soll. Dieses Kapitel gibt einen kurzen Überblick über dieses Thema, um Unklarheiten gleich zu Beginn dieser Arbeit möglichst aus dem Weg zu schaffen. Außerdem wird kurz auf die Hardwareanforderungen von Windows RT und Windows 8 Geräten eingegangen.

2.1. Entwicklung auf WinRT

Die eben erwähnte und von Microsoft herausgebene in 2.1 ersichtliche Grafik hat vor allem in den ersten Monaten nach der Ankündigung von WinRT für große Irritationen gesorgt.

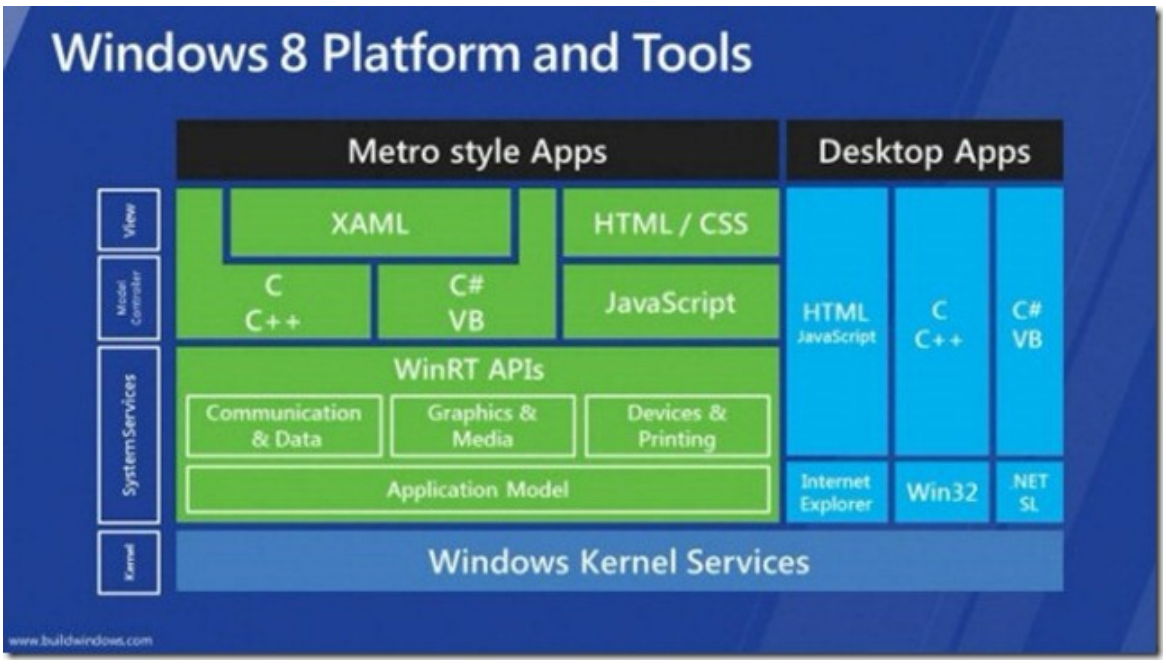

Abbildung 2.1.: WinRT Architektur, Grafik von Microsoft

Einer der Hauptkritikpunkte war dabei, dass nicht klar herauszulesen war, in welcher Beziehung die .NET API zur WinRT API steht, und ob die langjährig verwendete .NET API nun komplett ersetzt wird. [4]

Tatsächlich läuft die neue COM basierte Windows Runtime nur parallel zum API-Set von .NET. Man kann weiterhin in C# programmieren und die Standard-APIs auch wie gewohnt aufrufen, im Hintergrund mapped der Compiler die Aufrufe allerdings und leitet sie an entsprechende WinRT Funktionen weiter.

Die WinRT APIs sind dabei in WinMD-Files definiert, die dasselbe Dateiformat (ECMA-335) besitzen, wie die .NET Framework Assemblies. Trotz dieses gemeinsamen Formates ähneln sich die Metadaten-Files nur und sind nicht ident, was unter anderem den Grund hat, dass die WinRT und .NET komplett unterschiedliche Datentypen besitzen, welche ebenfalls bei Bedarf aufeinander gemapped werden müssen.

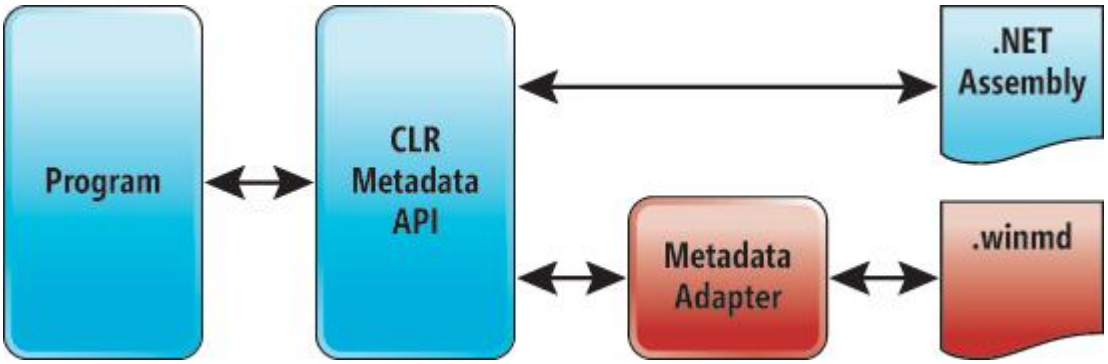

Abbildung 2.2.: Metadata Adapter

Damit das funktioniert, müssen Compiler auch WinMD Files lesen können, was standardmäßig natürlich nicht der Fall ist, da die Compiler im Normalfall vor der Entwicklung von WinRT eingesetzt wurden. Aus diesem Grund wurde der Metadata Adapter eingeführt, wie in 2.2 ersichtlich.
Die Common Language Runtime (CLR) Metadata API wird von verschiedenen Compilern, wie auch von Visual Studio verwendet, um .NET Metadata Files auszulesen. Der Metadata Adapter hat nun die Aufgabe, die WinMD-Files für die CLR Metadata API zu übersetzen, damit diese gelesen werden können, als ob sie selbst .NET Assemblies wären. [5]

Aus Gründen des beschränkten Umfangs dieser Arbeit wird der interessierte Leser an dieser Stelle jedoch auf die angegebenen Quellen verwiesen. Die APIs von WinRT finden an späterer Stelle in dieser Arbeit noch einmal detailliert Erwähnung.

2.2. Neue Hardware und Tablets

WinRT Applikationen laufen immer noch auf bisherigen Desktopcomputern und Laptops (solange auf diesen Windows 8 oder Windows RT installiert ist), erstmalig sind aber Anwendungen, die für ein Windows-Betriebssystem geschrieben wurden, auch auf Tablets ausführbar.
Diese Geräte werden gewöhnlich ohne Maus und Tastatur bedient, die Toucheingabe muss genügen. Um trotzdem eine ausreichende Bedienbarkeit zu gewährleisten, hat Microsoft Mindest-Hardwareanforderungen festgeschrieben, die ein Hersteller erfüllen muss, um ein Gerät mit einem der neuen Betriebssysteme auf den Markt bringen zu dürfen.

Diese sind in drei PDF-Dokumenten nachlesbar, die insgesamt 1258 Seiten umfassen. Um ein Beispiel zu nennen: User verwenden gewöhnlich die Tastenkombination Ctrl + Alt + Del am Startscreen, um sich anschließend anmelden zu können. Bei Geräten, die keine Tastatur mit sich führen, muss diese Funktionalität ebenfalls gewährleistet bleiben. Aus diesem Grund müssen solche Geräte eine Windowstaste und einen Power-Knopf besitzen, welche, in Kombination gedrückt, die selbe Funktion auslösen.

Die Windows-Taste spielt also vermehrt eine wichtige Rolle für Microsofts neue Usability-Strategie und ist neben dem Power-Knopf und den Lautstärkereglern auch die einzige physische Taste, die jedes Gerät implementiert haben muss. [6], S.89f

Näher wird in dieser Bachelorarbeit jedoch nicht auf die Hardwareanforderungen eingegangen, der interessierte Leser kann die detaillierten Informationen aber über die von Microsoft zur Verfügung gestellten PDFs abfragen.

3. Layout

Im Rahmen dieser Bachelorarbeit wird eine einfache Line of Business Applikation namens *Practitioner Mobile* erstellt. Auf diese wird vereinzeilt bereits in diesem Kapitel zu praktischen Demonstrationszwecken eingegangen. Der genaue Zweck dieser Applikation wird in einem späteren Kapitel erläutert, in diesem stehen hingegen Basics im Bereich des Layouts einer typischen Windows 8 / Windows RT Applikation im Vordergrund.

3.1. XAML

Die Extensible Application Markup Language (XAML) ist kein neues Konzept, das für WinRT entwickelt wurde. Vielmehr existiert diese bereits seit .NET 3.0 und wurde erstmalig für die Benutzeroberflächenerstellung bei Windows Presentation Foundation (WPF) Anwendungen eingesetzt. Grund für die Einführung war der Wunsch einer besseren Trennung von User Interface (UI) Design und Logik, wobei die Oberfläche prinzipiell immer noch rein programmatisch erstellt werden kann.

XAML basiert, wie der Name bereits suggeriert, auf der Extensible Markup Language (XML), wobei den Elementen Klassen im Code zugeordnet werden. Attribute werden als Properties und Events interpretiert. Um verschiedene Klassen dabei unterscheiden und kategorisieren zu können, werden diese in Namespaces eingeteilt, die im XAML Code deklariert werden müssen. [7], S.141ff

Zwei Namespaces sind dabei besonders hervorhebenswert:

- *http://schemas.microsoft.com/winfx/2006/xaml/presentation*:
  In diesem Namespace finden sich die meisten der Standard UI Elemente. Elemente in diesem Namespace müssen als einzige beim Einfügen in XAML Code kein Kürzel angeben, um sich von anderen Namespaces zu unterscheiden.

  Beispiele: Grid, Button, TextBlock.

- *http://schemas.microsoft.com/winfx/2006/xaml*:
  Das ist der Namespace für XAML spezifische Elemente, in welchem spezielle XAML-Parser Direktiven definiert werden.
  Diesen wird üblicherweise ein x vorgestellt.
  Klassische Controls, wie man sie unter einem UI Namespace eigentlich erwarten würde, findet man hier nicht.

  Beispiele: Class, Name, Key.

Natürlich können neben den Standard Controls auch eigene je nach Bedarf ergänzt werden. Diese sind dann in einem eigenen zu definierenden Namespace zu finden. [8], S.29f

Ein aus Visual Studio frisch erzeugtes Windows 8 / Windows RT Projekt in XAML Code sieht folgendermaßen aus:

Listing 3.1: XAML einer neu erzeugten Windows 8 Applikation

```
<Page
    x:Class="PractitionerMobile.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:CustomNamespace"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    </Grid>
</Page>
```

Die beiden zuvor erwähnten Namespaces sind an dieser Stelle wiederzufinden.
Wie man anhand des *local* Namespaces sieht, können eigene Namespaces ganz simpel durch das Voranstellen einer *using* Direktive hinzugefügt werden. Dieser eigene Namespace muss natürlich auch im Code Behind vorhanden sein.
CustomNamespace ist dabei der Namespace der Klasse oder Klassen, in welchen eigene Funktionen definiert sind. Die Rolle des *Grid* Elements wird später in dieser Arbeit erklärt.

Näher wird an dieser Stelle aus Platzgründen nicht auf die Grundlagen von XAML eingegangen, diese werden zum weiteren Verständnis dieser Bachelorarbeit vorrausgesetzt und können bei Bedarf in den erwähnten Quellen selbst erarbeitet werden.

3.2. Struktur einer Applikation

Jedes XAML File besitzt auch ein Code Behind File, in welchem der Großteil der Logik ausprogrammiert wird. So wird die in 3.1 erwähnte Trennung dieser vom UI Design realisiert.

In den Files App.xaml und dessen Code Behind File App.cs können Elemente definiert werden, die seitenunabhängig zur Verfügung stehen sollen. In letzterem sind auch globale Events wie das Starten oder das Beenden einer Applikation abfangbar.

Bilder finden sich im Folder *Assets*, Styles im Folder *Common* wieder.

Der klassische Aufbau eines frisch generierten Solution ist in Abbildung 3.1 ersichtlich.

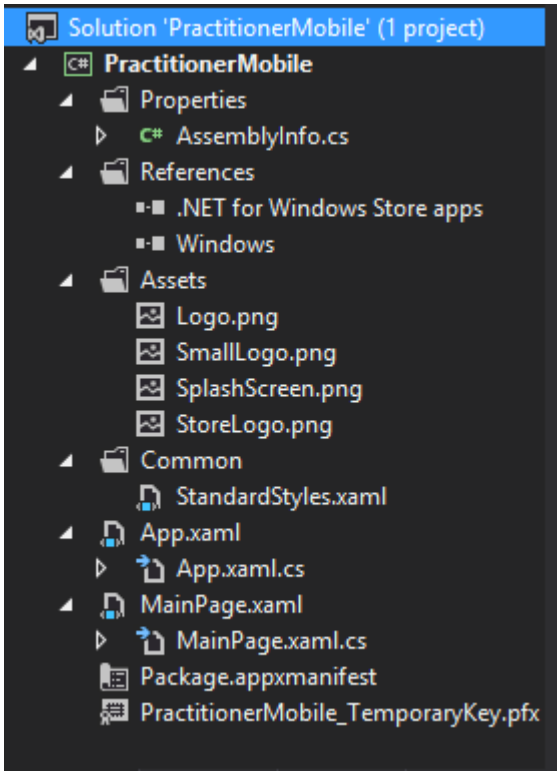

Abbildung 3.1.: File Tree

Wählt man beim Erstellen des Projekts kein leeres Projekt aus, sondern eines der anderen verfügbaren Templates, wird etwas mehr (XAML) Code automatisch generiert. Wählen kann man zwischen einer Grid App oder einer Split App, die bereits Listen von Elementen beinhalten und auch die Navigation zu einer Detailseite ist bereits vorimplementiert. In der Practitioner Mobile Applikation wird von diesen Templates jedoch kein Gebrauch gemacht, alle Elemente können auch später bei Bedarf ergänzt werden.

3.3. Portrait, Landscape und Snapped View

Die Tatsache, dass sich bei Tablets nicht nur die Bildschirmgröße, sondern auch die Orientierung ändern kann, stellt Entwickler vor eine zusätzliche Herausforderung. Außerdem hat Microsoft in seinen neuen Betriebssystemen den Snapped View integriert, welcher es ermöglicht, zwei Anwendungen gleichzeitig am Bildschirm anzuzeigen, normalerweise nimmt eine Anwendung immer den gesamten Bildschirm ein. Dieser Snapped View eröffnet viele neue Möglichkeiten, man kann in etwa einen Texteditor und ein Wörterbuch parallel laufen lassen, um beim Schreiben eines fremdsprachlichen Textes ein Wort schnell und einfach nachschlagen zu können.
Abbildung 3.2 visualisiert dieses Beispiel anhand zweier bereits im Windows Store verfügbaren Apps: Microsofts OneNote Notizbuch, und dem Übersetzungstool Leo. In den Snapped View kann nur im Landscape Mode gewechselt werden und ist verpflichtend für jede Applikation.

Um Entwicklern die Arbeit zu erleichtern, bestimmt Microsoft eine Mindestbreite für die verschiedenen Viewstates. Für die beiden Fullscreen Views (Portrait und Landscape) wird so eine

Mindestauflösung von 1024x768 Pixel garantiert. Um den Snapped View zu unterstützen, muss das Gerät bereits eine Breite von 1366 Pixel aufweisen - 320 Pixel extra für die gesnappte App und 22 Pixel für den Splitter zwischen den beiden Applikationen. [8], S.65ff

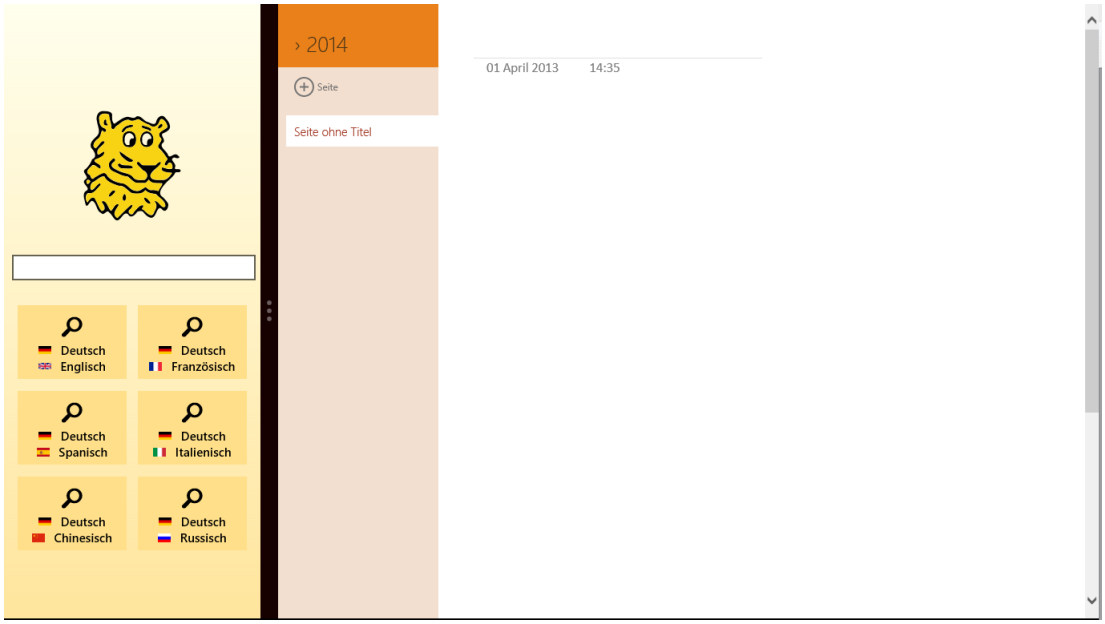

Abbildung 3.2.: Snapped View

3.4. Splashscreen

Beim Öffnen der Applikation wird ein Startbildschirm, der so genannte Splash Screen, angezeigt. Dieser ist statisch und kann durch ein eigenes Bild ersetzt werden, in dem man das File *SplashScreen.png*, ersichtlich in Abbildung 3.1, durch die gewünschte Grafik ersetzt.
Die Zeit, in welcher der Splashscreen gezeigt wird, kann der Entwickler dazu nutzen, um Vorbereitungsarbeiten für seine Applikation zu treffen. Diese dürfen die Zeitspanne von fünf Sekunden nicht überschreiten. [8], S.10f

3.5. Controls

Wie jedes XML basierte Dokument besitzt auch ein XAML File ein Root-Element, bei einer WinRT Seite ist dies *<Page>*. Innerhalb dieses Elements finden sich dann verschachtelt, wie in einem XML Dokument, verschiedenste weitere Elemente.

3.5.1. Panels

Ein Panel ist ein Wrapper-Element - ein so genanntes Content Control -, das für das Basis-Layout seiner Kindelemente verantwortlich ist.
Die am häufigsten verwendetsten Panels sind:

- *Canvas*:
  Einfaches, dafür performantes Panel.
- *StackPanel*:
  Optimal, um Elemente horizontal oder vertikal aneinanderzufügen.
- *Grid*: Eines der am häufigsten verwendeten und mächtigsten Panels, mit einer Tabelle vergleichbar.
  Elemente können in zuvor definierte Zeilen und Spalten eingefügt und somit präzise positioniert werden.

[8], S.71ff

3.5.2. Aufzählungen

Im Gegensatz zu den eben in 3.5.1 vorgestellten Panels, besitzen so genannte Items Controls, zu denen Aufzählungscontrols gehören, zusätzliche Properties: *SelectedIndex*, *SelectedItem* und *SelectedValue*. Diese machen es möglich, in Aufzählungen bestimmte Elemente auszuwählen, und somit in weiterer Folge nach Elementen zu suchen, zu aktualiseren, zu löschen oder neue Elemente hinzuzufügen.
Diese Properties werden dabei automatisch synchronisiert, der Programmierer muss sich also in etwa nicht darum kümmern, den Index beim Ändern einer Aufzählung zu verändern.

Als übergeordnetes Element besitzen Items Controls ebenfalls Panels, die meisten dieser besitzen aber zusätzlich eine Art von Virtualisierung, um Performanceoptimierungen zu erreichen. Besonders bei einer großen Anzahl von Elementen in diesen Listen ist eine deutliche Effizienzsteigerung bemerkbar.

Die bekanntesten Aufzählungscontrols sind:

- *ComboBox*: Die ComboBox ist ein Control, die es erlaubt, aus mehreren verfügbaren Elementen genau eines aus einer Drop Down Liste auszuwählen. Sie nimmt dabei nicht viel Platz am UI weg, da sie im geschlossenen Zustand nur das aktuell gewählte Element anzeigt und wird aus diesem Grund gerne verwendet. Neue Elemente zu einer bestehenden ComboBox kann der User allerdings vom UI aus nicht.

- *ListBox*: Die ListBox ähnelt der CoboBox, mit dem Unterschied, dass alle Elemente am UI angezeigt werden, und dass man - je nach Konfiugration - auch mehrere Elemente auf einmal auswählen kann.

- *ListView*: Die ListView ist der ListBox sehr ähnlich und stellt sozusagen eine für Touch optimierte Variante derselben dar.

- *GridView*: Das GridView hat wiederum viele Gemeinsamkeiten mit der ListView. Unterschiede findet man nur in der Darstellung der Elemente: während die ListView standardmäßig vertikal scrollt, orientiert sich die GridView sich an der Horizontalen. Außerdem wird jeden Element so viel Platz gegeben, wie dem größten Element in der Aufzählung. Alle anderen Elemente passen ihre Darstellung also diesem an.

[8], S. 207ff

3.5.3. TextBlock

Das TextBlock wird dazu verwendet, um Text im UI anzuzeigen. Dabei stehen dem Control viele verschiedene Formatierungsmöglichkeiten als Properties zur Verfügung. Verändern kann man so unter anderem die Schriftart, die Schriftgröße, die Farbe des Textes oder dessen Formatierung.
Dabei muss innerhalb eines TextBlocks nicht der ganze Textstring gleich formatiert werden. Man kann diesen durchaus in einzelne Bereiche, so genannte *Run*-Blöcke aufteilen und getrennt voneinander bearbeiten:

Listing 3.2: Run-Blöcke in einem TextBlock

```
<TextBlock>
        <Run FontSize="10">Dieser Text</Run>
        <Run FontSize="20">kann seperat</Run>
        <Run FontSize="30">formatiert werden.</Run>
</TextBlock>
```

Setzt man das Property *IsTextSelectionEnabled* auf *true*, kann der User Text in TextBlocks auswählen und anschließend zum Beispiel in die Zwischenablage kopieren. [8], S. 227ff

3.5.4. TextBox

Das TextBox Control wird an den Stellen im UI verwendet, in denen der User Text eingeben können soll. Bekommt dieses Control den Fokus, öffnet sich die Software-Tastatur automatisch (sofern es sich um ein Touch-fähiges Gerät handelt). Rechtschreibprüfung unterstützt das Control von Haus aus, genauso wie das Anbieten von Vorschlägen für die Eingabe. [8], S. 240ff

3.6. Semantic Zoom

Semantic Zoom wird die Möglichkeit genannt, aus einer großen Anzahl von Elementen - wie etwa in Aufzählungen - hinauszuzoomen, um einen besseren Überblick über die verfügbaren Elemente zu bekommen, diese zu gruppieren, oder ähnliches. Windows verwendet diese Funktionalität bereits am Startbildschirm, auf welchem alle verfügbaren Apps angezeigt werden.

Wie bei den meisten Möglichkeiten, die WinRT Applikationen beherrschen, macht diese Option nicht überall Sinn. Gerade aber wenn es darum geht, sich in einer Vielzahl von Elementen zurecht zu finden, kann diese aber sehr hilfreich sein. [8], S.223ff

3.7. Entwicklung von eigenen Controls

Wie auch schon bei WPF, stehen mit XAML dem Entwickler eine große Anzahl von Möglichkeiten für das Erstellen von eigenen Controls zur Verfügung.

3.7.1. Dependency Properties

Dependency Properties zählen zu den wichtigsten Elementen in XAML. Von einem normalen Property unterscheiden sie sich dabei nur durch XAML-spezifische Zusatzfunktionalitäten: Änderungsnotifikationen an gebundene Elemente, Wertevererbung an Kindelemente und die Unterstützung für mehrere Provider. Man erklärt die Motivation für Dependency Properties und die Funktion der automatischen Änderungsnotifikation am Besten an einem einfachen Beispiel.

Dafür wird eine Art Basis-Button Control gebaut, welches - wie der Standard-Control Button - erkennen soll, ob er sich gerade in gedrücktem oder ungedrücktem Zustand befindet. Es wird eine neue Klasse namens *BaseButton* erstellt, die wie Button auch von *ContentControl* ableitet, um möglichst viele Gemeinsamkeiten zu behalten. Unser Button soll ein neues Property namens *IsPressed* bekommen, anhand dem er erkennen kann, ob er aktiviert oder deaktivert wurde. Ein Klick auf ihn soll dabei seinen Zustand ändern.

Abbildung 3.3.: Button nicht gedrückt

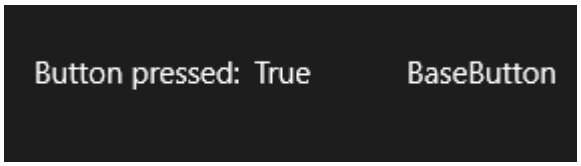

Abbildung 3.4.: Button gedrückt

Der entsprechende XAML Code, gekürzt auf die wichtigsten Elemente, sieht folgendermaßen aus:

Listing 3.3: XAML eines neu erstellten Controls

```
<Page
 [...]
xmlns:pc="using:PractitionerMobile.Controls">
 [...]
<TextBlock Text="Button pressed:" />
<TextBlock Text="{Binding ElementName=baseButton, Path=IsPressed}" />
<pc:ButtonBase Name="baseButton" Content="BaseButton"
        Tapped="baseButton_Tapped" />
 [...]
</Page>
```

Wie in Listing 3.3 zu erkennen ist, wird unsere Klasse *ButtonBase* nicht einfach so eingefügt, sondern bekommt einen Namespace vorangestellt. Dieser muss im XAML File innerhalb des Page-Elements ebenfalls deklariert werden.

Der Text des zweiten TextBlock Elements bindet sich an unseren gerade erstellten Button und zeigt den Wert dessen IsPressed Value an, welcher sich bei jedem Klick auf *BaseButton* ändert. Die Grundlagen des XAML Databindings sind nicht Gegenstand dieser Arbeit, vielmehr sollen die Möglichkeiten aufgezeigt werden, die Entwicklern mit XAML und C# zur Verfügung stehen.

Schlussendlich muss noch der Code Behind implementiert werden, welcher so aussieht, wie in Listing 3.4 dargestellt.

Listing 3.4: Für Dependency Properties benötigter Code

MainPage.xaml.cs:

```
using CustomBaseButton = PractitionerMobile.Controls.ButtonBase;
[...]
private void baseButton_Tapped(object sender, TappedRoutedEventArgs e)
{
        CustomBaseButton button = (sender as CustomBaseButton);

        if(button.IsPressed)
        {
            button.SetValue(CustomBaseButton.IsPressedProperty, false);
        }
        else
        {
            button.SetValue(CustomBaseButton.IsPressedProperty, true);
        }
 }
[...]
```

ButtonBase.cs:

```
[...]
public class ButtonBase : ContentControl
{
    public bool IsPressed
    {
        get { return (bool)GetValue(IsPressedProperty); }
        set { SetValue(IsPressedProperty, value); }
    }

    public static readonly DependencyProperty IsPressedProperty =
        DependencyProperty.Register("IsPressed", typeof(bool),
                typeof(ButtonBase), new PropertyMetadata(false,
```

```
                                OnIsPressedChanged));

    private static void OnIsPressedChanged(DependencyObject d,
                                           DependencyPropertyChangedEventArgs e)
    {
        Debug.WriteLine("button pressed: " + e.NewValue);
    }
}
[...]
```

Im File *MainPage.xaml.cs* wird mittels der SetValue-Methode der Wert des in *ButtonBase.cs* definierten Dependency Properties umgesetzt.
Der .NET Property Wrapper *public bool IsPressed* ist dabei rein optional, ebenso wie der Property Changed-Callback *OnIsPressedChanged*, in dem man zusätzliche Logik verpacken könnte. [8], S. 102f

3.7.2. AttachedProperties

Attached Properties unterscheiden sich von Dependency Properties in der Funktionsweise und Implementierung nur gering, und zwar hauptsächlich dadurch, dass diese

> [...] nicht auf Objekten der Klasse gesetzt [werden], die das *DependencyProperty*-Feld und damit den *Schlüssel* zum eigentlichen Wert besitzt, sondern auf Objekten anderer Klassen.

Somit kann man in etwa von einem Button-Control aus, das ein TextBox-Control beinhaltet, direkt auf dieses zugreifen und dessen Wert verändern. Näher wird auf Attached Properties aufgrund der großen Ähnlichkeit zu Dependency Properties nicht eingegangen. [7], S.413

3.7.3. Property Wertevererbung und mehrere Provider

Wird auf einem Parent-Element ein Property, zum Beispiel die Schriftgröße eines Textes, auf einen bestimmten Wert gesetzt, kann es die Eigenschaft auch auf seine Kindelemente übertragen. Ob das geschieht oder nicht, hängt jedoch von einigen Faktoren ab. Setzt das Kindelement seinerseits einen Wert für dasselbe Property, wird der Wert des Parents überschrieben.

Auch sonst gibt es eine definierte Prioritätsliste, welche genau angibt, welcher Wert für ein Property nun tatsächlich verwendet werden soll. Wenn in etwa gerade eine Animation durchgeführt wird, hat der in dieser gesetzte Wert immer Vorrang. Die genaue Prioritenliste sieht wie folgt aus:

1. Aktive Animationen
2. Lokale Werte
3. Template Properties
4. Style-Setters
5. Property Wertevererbung

6. Default-Wert

Über Styles und Templates wird an späterer Stelle dieser Arbeit noch geschrieben. [8], S. 104ff

3.7.4. Styles

Angenommen, man will das Aussehen eines Buttons verändern. In diesem Fall setzt man im XAML-File die entsprechenden Properties und bekommt so einen Button, der genauso gerendert wird, wie man es möchte.
Wenn man nun aber nicht nur eines Controls ändern möchte, sondern eine ganze Reihe von einem bestimmten Control, wird es mühsam und unübersichtlich: man müsste bei jedem einzelnen die selben Properties setzen, was das XAML-File unnötig aufbläht.
An dieser Stelle kommen Styles ins Spiel.

Styles machen es möglich, mehrere Eigenschaften auf einmal zu gruppieren und auf alle Elemente eines Typs einfach anzuwenden. Definiert werden Styles direkt auf Controls, sodass alle in der Hierarchie darunterliegenden Kindelemente dieses Typs besagte Eigenschaften nutzen können. [8], S. 410ff

In Listing 3.5 ist dargestellt, wie ein Style für TextBlock-Elemente innerhalb der definierten ListView definiert wird, der die *FontWeight* auf *Bold*, die *Foreground*-Farbe auf *Red*, und ein *Padding* von links auf 20 setzt.
Der Style wird dazu als Resource der ListView definiert und in den gewünschten TextBlock-Elementen mit *Style="{StaticResource ListViewTextBlockStyle}"* angewendet.

Listing 3.5: Definition eines Styles

```
<ListView Name="elementPanel">
    <ListView.Resources>
        <Style x:Key="ListViewTextBlockStyle" TargetType="TextBlock" >
            <Setter Property="FontWeight" Value="Bold" />
            <Setter Property="Padding" Value="20,0,0,0" />
            <Setter Property="Foreground" Value="Red" />
         </Style>
    </ListView.Resources>
    <ListViewItem>
        <TextBlock Text="TextBlock 1"
          Style="{StaticResource ListViewTextBlockStyle}" />
    </ListViewItem>
    <ListViewItem>
        <TextBlock Text="TextBlock 2" />
    </ListViewItem>
</ListView>
```

Nur das erste TextBlock-Control bekommt den Style zugewiesen, der zweite behält somit seine Default-Formatierung, wie in Abbildung 3.5 gezeigt wird. Lässt man *x:Key* und *x:Name* weg, verzichtet also darauf, dem Style einen Identifier zu geben, wird der Style auf alle Elemente des definierten Typs angewendet.

Abbildung 3.5.: Styles

Ein Control kann einen durch Styles gesetzten Property-Wert auch überschreiben, indem es einfach einen anderen Wert für diesen setzt.

Schon beim Anlegen einer neuer WinRT-Solution wird das File *Common/StandardStyles.xaml* erstellt. In diesem File finden sich einige Styles bereits vordefiniert, die nur noch im XAML eingebunden werden müssen. Styles können dabei auch voneinander erben, was wieder einiges an Schreibarbeit erspart und den Code übersichtlicher macht.

WinRT-Applikationen können unter drei verschiedenen Themes laufen: *Default* (entspricht schwarz), *Light* (entspricht weiß) und *High Contrast* (entspricht weißem Text auf schwarzem Hintergrund).
Schlussendlich ist es auch möglich, für jedes dieser Themes einen anderen Style zu definieren. [8], S. 413ff

3.7.5. Templates

Die Einführung von XAML hat es ermöglicht, die Darstellung eines Objekts im UI vollständig von dessen Logik zu trennen. Somit kann in etwa die Darstellung eines Buttons nach Belieben verändert werden, ohne dass sich an der Funktionalität etwas ändert.
Das vorherige Unterkapitel 3.7.4 hat gezeigt, wie einfach es ist, mittels Styles das Erscheinungsbilds eines Controls zu verändern. Dies hat sich aber auf einfache Properties beschränkt, will man das Design von Grund auf neu schreiben, bedient man sich am besten Templates.

Es gibt mehrere verschiedene Arten von Templates, drei davon erben von der Klasse *FrameworkTemplate*: *ControlTemplate*, *DataTemplate* und *ItemsPanelTemplate*.

ControlTemplate

ControlTemplates ermöglichen es, das Aussehen eines Controls vollkommen nach Belieben neu zu gestalten. Ein Button muss somit nicht mehr rechteckig sein, sondern kann durchaus auch als Kreis oder Ellipse dargestellt werden. Die Funktionalität bleibt dabei erhalten, er verfügt zum Beispiel immer noch über sein *OnClick*-Event.

ControlTemplates kann man direkt im XAML definieren, aber auch wie Styles in ein Resources-File auslagern. Ebenfalls möglich ist es, in der Definition eines Styles (zum Beispiel im File *StandardStyles.xaml*) ein Template zu inkludieren.

Ein Problem taucht aber dennoch auf: wenn man das Design eines Controls, zum Beispiel eines Buttons, neu schreibt, woher weiß dieser, an welcher Stelle er sein Content-Property anzeigen soll? Angenommen, er würde als Content den String *Mein Button* und als Design einen Kreis statt ein Rechteck gesetzt haben - der String würde nie angezeigt werden.
Um dieses Problem zu lösen, muss man in die Template-Definition des Buttons ein Control definieren, welches diese Aufgabe übernimmt. Das könnte eine TextBox sein, generischer ist allerdings eine ViewBox - der Content beschränkt sich dann nicht nur auf Text, sondern könnte ebensogut ein Bild oder ein sonstiges Element sein. Das Template könnte dann so aussehen, wie in Listing 3.6.

Listing 3.6: Definition eines Templates. Modifiziert von [8], S.421

```
<Button Content="Mein Button">
   <Button.Template>
      <ControlTemplate TargetType="Button">
         [...]
         <ViewBox Width="100" Height="100">
            <ContentControl Margin="20"
                  Content="{TemplateBinding Content}" />
         </ViewBox>
         [...]
      </ControlTemplate>
   </Button.Template>
</Button>
```

Durch das explizite Setzen des Contents auf ein Control kann das Problem also gelöst werden. XAML bedient sich hierbei an einem Konzept, das als *DataBinding* bekannt ist und bereits im Abschnitt 3.7.1 verwendet wurde. Die Funktionsweise dieses Konzepts wird in dieser Bachelorarbeit nicht detailliert erläutert. [8], S. 418ff

Visual States

Bisher wurde erklärt, wie das Aussehen eines Controls verändert werden kann. Ein solches kann aber auch unterschiedliche Zustände besitzen, bei einem Button würde so in etwa zwischen gedrückt und nicht gedrückt unterschieden werden. Wenn dieser je nach Zustand eine andere Darstellung besitzen soll, kommen Visual States ins Spiel.

Zustände, die ein Control annehmen kann, werden in so genannte *State Groups* gruppiert, die einander ausschließen. Definieren kann man solche hierarchisch direkt unter dem gewünschten Control-Root-Element mit dem Attached Property *VisualStateManager.VisualStateGroups*, welche wiederum *VisualStateGroup*-Elemente und abermals eine hierarchische Ebene darunter *VisualStates* enthalten.
Das *x:Name* Attribut einer VisualStateGroup entspricht dabei einer möglichen Zustandsgruppe (zum Beispiel *CommonStates*), das eines VisualStates einem möglichen Zustand (zum Beispiel *Focused*). Welche Zustände es gibt, ist individuell pro Control verschieden und als Attribut auf der Klasse im Code Behind definiert.

Übergänge zwischen verschiedenen Visual States werden gerne durch Animationen durch-

geführt, für die man das *Storyboard*-Element verwendet. *Visual Transitions* erleichtern dabei viele Animationsvorgänge, indem man nur angeben muss, welches Property sich von welchem auf welchen Wert ändern soll. Die Animation wird dann berechnet und automatisch durchgeführt. Auch gewünschte Verzögerungen können dabei angegeben und somit die Schnelligkeit der Transformation kontrolliert werden.
Auf Animationen wird in dieser Bachelorarbeit aus Platzmangel jedoch nicht näher eingegangen. [8], S.428ff

DataTemplate

Oftmals steht im Vorhinein noch nicht fest, welche Elemente eine Aufzählung tatsächlich beinhalten wird, da diese erst später programmatisch und durch Data Binding hinzugefügt werden. Wenn es die Elemente aber im XAML noch nicht gibt, kann man sie auch nicht stylen oder Properties setzen.

Um mehrere Objekte innerhalb einer Aufzählung gleichzeitig zu formatieren, werden aus diesem Grund Data Templates eingesetzt. Diese werden hierarchisch direkt unter dem gewünschten Aufzählungscontrol definiert. [8], S.447

Für eine ListView würde ein DataTemplate in etwa so aussehen, wie in Listing 3.7 dargestellt.

Listing 3.7: Data Template

```
<ListView ItemsSource="{Binding Patients}>
   <ListView.Resources>
      <Style x:Key="ListViewTextBlockStyle" TargetType="TextBlock" >
         <Setter Property="FontWeight" Value="Bold" />
         <Setter Property="Padding" Value="20,0,0,0" />
      </Style>
   </ListView.Resources>

   <ListView.ItemTemplate>
      <DataTemplate>
          <TextBlock Text="{Binding Path=Name}"
                Style="{StaticResource ListViewTextBlockStyle}" />
      </DataTemplate>
   </ListView.ItemTemplate>
</ListView>
```

Die ListView bekommt hier ein *DataTemplate* gesetzt, in dem für *TextBlock*-Elemente ein neuer Style gesetzt wird. Dieser ist in diesem Codebeispiel nicht direkt gesetzt, sondern als Style innerhalb der ListView referenziert. Styles wurden im Unterkapitel 3.7.4 erklärt. Der Content des TextBlock-Elements ist das *Name*-Property eines Elements, das sich in eine Liste namens *Patients* im CodeBehind befindet.

ItemsPanelTemplate

Mit einem ItemsPanelTemplate können Items innerhalb eines Controls einfach gestaltet werden. Im Gegensatz zum im Unterkapitel 3.7.5 vorgestellten Data Template, das sich auf Daten innerhalb

einer Liste bezieht, werden hier nicht deren Properties verändert, sondern die des Listenelements selbst. Will man in etwa erreichen, dass sich jedes Listenelement innerhalb eines VirtualizingStackPanels befindet, das eine horizontale Orientierung hat, so kann man dies einfach durch in Listing 3.8 dargestelltes XAML erreichen.

Listing 3.8: ItemsPanelTemplate, aus [8], S.210

```
<ListBox>
    <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
</ListBox>
```

[8], S.210f

3.8. Frameworks

TODO: mention frameworks as syncfusion

3.9. Custom Control für Practitioner Mobile

Für die Anwendung Practitioner Mobile wurde ein eigenes Control entwickelt, das es erlaubt, für einen bestimmten Patienten einen Ordinationseintrag hinzuzufügen. Das Control ist in Abbildung 3.6 ersichtlich. Der vollständige Code dazu befindet sich im Anhang unter A und B.

Die folgenden Schritte beziehen sich alle auf ein englische Version der Entwicklungsumgebung Visual Studio 2012 Ultimate.

Zuerst fügt man ein neues Item zur Solution hinzu, indem man mit der rechten Maustaste auf einen Ordner des Solution Explorers klickt und *Add -> New Item* auswählt. Aus der vorgeschlagenen Liste sucht man den Eintrag *Templated Control* und bestätigt, was Visual Studio eine Klasse anlegen lässt, die von *Control* ableitet und einen Konstruktor besitzt, der das Property *DefaultStyleKey* setzt.
Außerdem wird, falls dieses noch nicht existiert, das File *Themes / Generic.xaml* angelegt und ein Eintrag für das eben erstellte Custom Control hinzugefügt. In dieser XAML-Datei kann ein Template für das neue Control erstellt werden. Standardmäßig werden für das *Border*-Property die drei Attribute *Background*, *BorderBrush* und *BorderThickness* gesetzt und die Struktur für ein eigenes Template aufgebaut. Auf diesem kann man anschließend aufsetzen und sein eigenes Template definieren.

Das Template des Custom Controls von Practitioner Mobile besteht aus einem Grid als Wrapper-Element, das ein Image-, ein ComboBox-, drei Textblockelemente und einen OK- und Cancel-Button beinhält. Der Pfad des Bildelements und die Elemente der ComboBox werden durch ein *TemplateBinding* gesetzt, welches auf je ein DependencyProperty verweisen, die im File des Custom Controls definiert sind. Wie DependencyProperties erzeugt werden, wurde in Abschnitt 3.7.1 erklärt.

Für die ComboBox wird in *OrdinationMask.cs* eine Standardliste definiert, indem beim Registrieren des DependencyProperties dem Parameter *PropertyMetadata* eine bereits befüllte Liste übergeben wird. XAML-Files, welche das Custom Control implementieren, können diese durch das explizite Setzen des *ItemsSource* Attributes überschreiben.

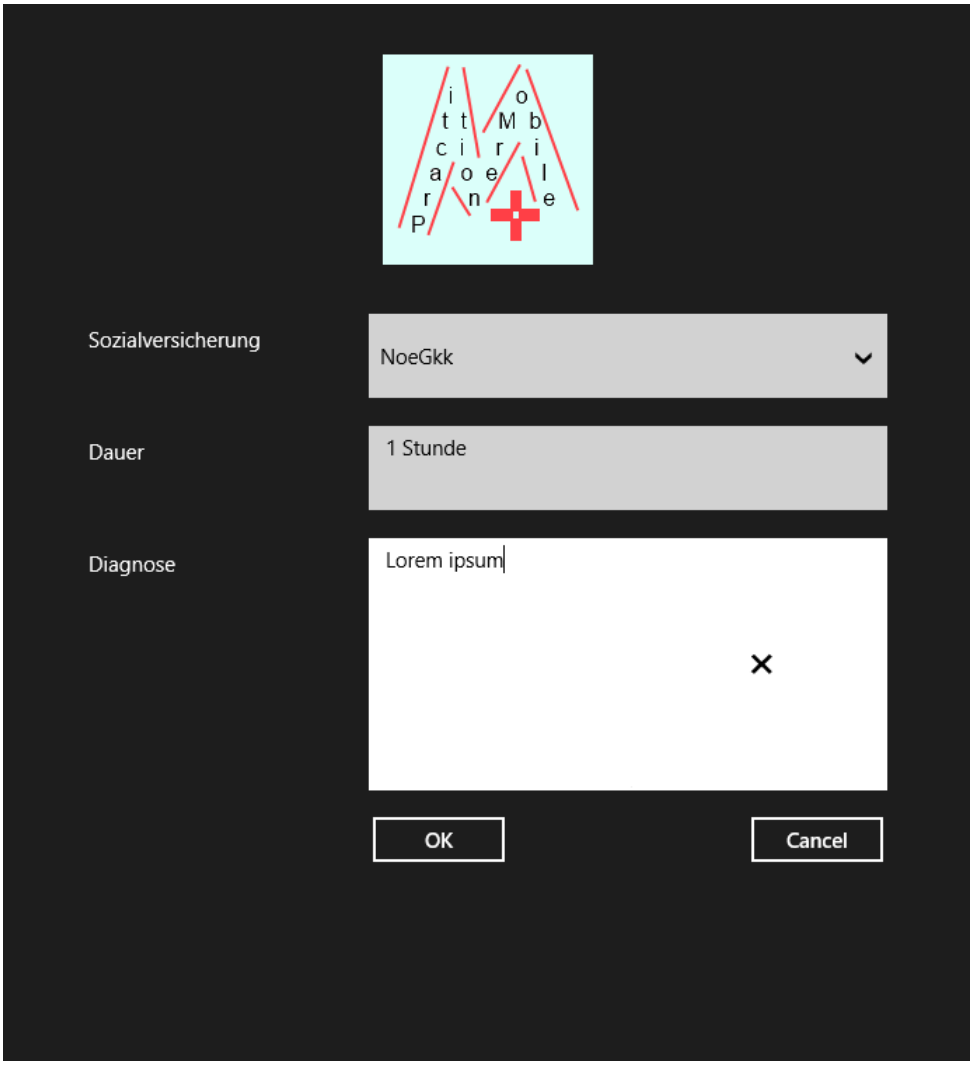

Abbildung 3.6.: Custom Control

Neben den Dependency Properties werden auch zwei *TappedEventHandler* deklariert, die für die Logik beim Betätigen des *OK*- oder *Cancel*-Buttons verantwortlich sind.

Das fertige Control kann man im *ToolExplorer* von Visual Studio wiederfinden. Von dort muss es nur noch in ein XAML-File gezogen werden und ist anschließend vollständig einsatzbereit.

4. Schnittstellen und Charms

Neben den ursprünglichen, auf WinRT umgemappten API-Aufrufen gibt es auch ein Set von komplett neuen APIs. Viele davon sind in der Charm-Bar zusammengefasst, welche mit den neuen Windows-Versionen eingeführt wurde. Die Charm-Bar ist eine Art Menüleiste, die auf der rechten Seite des Bildschirms durch eine Wischgeste oder dadurch erscheint, dass man die Maus in die rechte obere Ecke des Bildschirms bewegt. Hier finden sich appspezifische Einstellungen und Funktionen. Die neuen Möglichkeiten, die Entwicklern dadurch offenstehen, werden in diesem Kapitel erläutert. Auf Abbildung 4.1 ist diese neue Art der Menüleiste auf der rechten Seite der Online-Wörterbuchsapplikation *LEO* erkennbar.

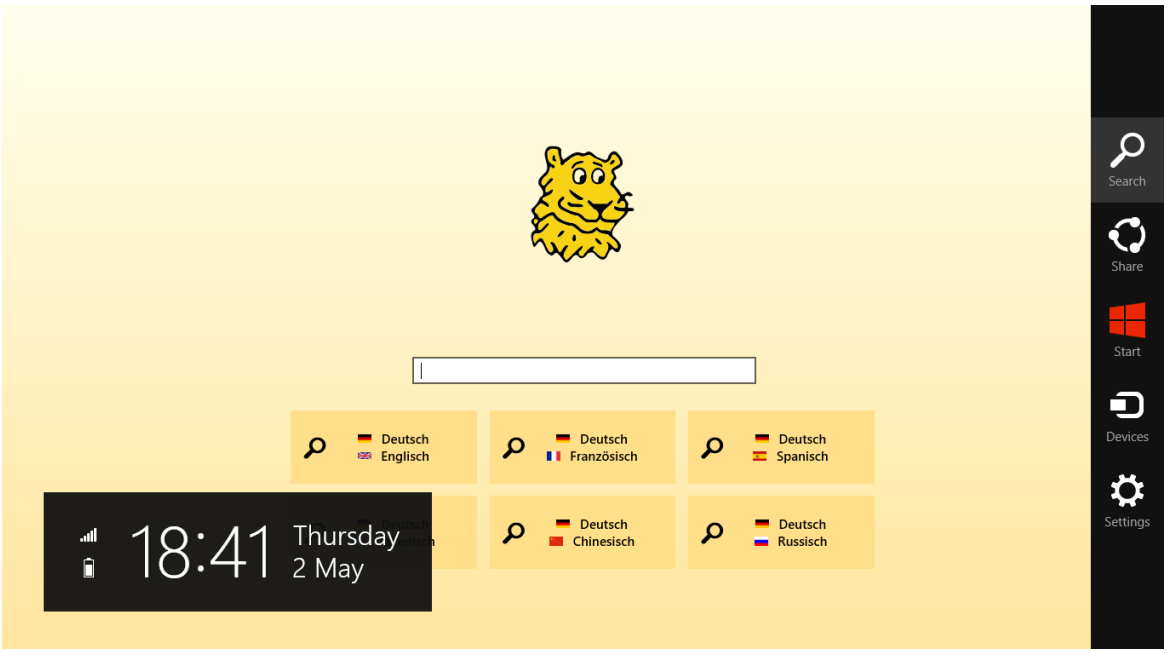

Abbildung 4.1.: Die neue Charms-Bar

4.1. Suchen

Die Suche in WinRT Applikationen wurde im Vergleich zu den vorhergehenden Windows Versionen gründlich überarbeitet und mit neuen Funktionen ausgestattet. So ist es nun nicht mehr nur möglich, nach Dateien und Ordnern zu suchen. Zusätzlich kann man nun einen Suchbegriff eingeben, eine WinRT App auswählen und innerhalb einer App (welche diese Funktionalität explizit unterstützen muss) nach diesem Begriff suchen. So wird es unter anderem möglich, den

Namen einer Stadt einzugeben, die Wetter-App auszuwählen, und direkt auf die Vorhersage an diesem Ort zuzugreifen. Wichtig dabei zu erwähnen ist, dass diese Applikation nicht laufen muss, nicht einmal im Hintergrund!
Die Suchresultate werden anschließend innerhalb der gewählten Applikation auf einer gesonderten Seite angezeigt, von wo aus man weitere Funktionen der Anwendung aufrufen kann.

Ein zusätzliches Feature der Suche ist das Anbieten von Suchvorschlägen beim Eintippen des Suchbegriffs. Auch die Logik hierhinter muss in der Applikation ausprogrammiert sein, kann dann aber eine weitere Usability-Verbesserung darstellen. Die Vorschläge werden ebenfalls in der Charms-Bar, direkt unter der Suchleiste angezeigt. [8], S.477ff

Auch die später vorgestellte Applikation Practitioner Mobile verwendet die erwähnten Funktionalitäten. Demonstriert werden diese in Kapitel 5.

4.2. Teilen

Die Teilen-Funktionalität ermöglicht es auf einfache Weise, Inhalte einer App mit anderen Applikationen zu teilen. Anstatt in etwa einen Text innerhalb einer Anwendung zu kopieren, das Mailprogramm zu öffnen, den Text dort einzufügen und anschließend zu versenden, könnte man dasselbe Resultat einfacher erzielen, indem man die Teilen-Funktion der Charm-Bar öffnet, ein Programm auswählt, das sich selbst als möglicher Empfänger klassifiziert (wie zum Beispiel das Mailprogramm), und direkt aus der gerade offenen Anwendung die Mail mit den gewünschten Inhalt verschickt.

Dieser ist dabei keinesfalls auf Text beschränkt, auch andere Objekte, von Bildern, spezielle Dokumente oder Binärdaten können geteilt werden. Wie der Inhalt dann weiter genutzt wird, entscheidet die App, die diesen entgegennimmt, programmatisch.
Zwei Properties, die beim Teilvorgang gesetzt werden, sind dabei besonders hervorzuheben: *Title* und *Description*. Diese werden in der Charm-Bar über der Liste der verfügbaren Apps angezeigt und entsprechen somit einer Art Vorschau, welche Daten geteilt werden. [8], S.486ff
Diese Funktion wird ebenfalls in Kapitel 5 demonstriert.

4.3. Geräte

Die Funktion *Geräte* ist der *Teilen* Funktionalität im Prinzip sehr ähnlich. Auch hier können Inhalte geteilt werden, diesmal allerdings nicht nur mit anderen Apps, sondern sogar mit anderen Geräten. Diese müssen allerdings Windows-zertifiziert sein, darunter fallen Geräte wie in etwa Drucker, Lautsprecher oder Ähnliches. Eine Quelle von so geteiltem Inhalt zu werden ist dagegen deutlich einfacher und mit ein wenig Code können Inhalte an einen Drucker, an Microsoft OneNote oder den Microsoft XPS Document Writer gesendet werden. [8], S.492ff

Wie einfach diese Art des Teilens ist, wird in Kapitel 5 gezeigt.

4.4. Einstellungen

Wie auch die zuvor erwähnten Funktionen, sollen auch die Applikationseinstellungen an einem zentralen Punkt auffindbar sein. Der letzte Eintrag in der Charms-Bar ist genau dafür reserviert. Bis zu acht eigene Anpassungen lassen sich so zentral an einer Stelle durchführen.

Beim Klicken auf einen Link innerhalb dieses Einstellungs-Tabs kann im Prinzip jeder beliebige Code ausgeführt werden. Konvention ist es allerdings, im XAML einen neuen Bereich zu definieren, der dem Settings-Screen sehr ähnlich sieht und in welchem man genauere Anpassungen durchführen kann. Der Name des Controls, das für diesen Zweck häufig verwendet wird, ist Popup. Dieses besitzt bereits von Haus aus Animationen und einen Zurück-Button, dessen Funktionalität man allerdings programmatisch selbst implementieren muss. [8], S. 503f

WinRT bietet außerdem noch eine zusätzliche Funktionalität an - Einstellungen können nun auch zwischen unterschiedlichen Geräten, auf denen der User mit derselben Live-ID angemeldet ist, synchronisiert werden. Microsoft nennt diese Funktion *Roaming Settings*. Die Menge der Daten, die synchron gehalten werden können, ist allerdings sehr begrenzt - lediglich 100 Kilobytes werden derzeit von Microsoft erlaubt. Aus diesem Grund sollten nur wichtige Informationen geroamt werden, wie in etwa ein Spielstand, an welcher Stelle man die App verlassen hat, oder eben Einstellungen.
Außerdem gibt es keine Garantie dafür, dass die Einstellungen sofort nach Änderungen synchronisiert werden, so kann es durchaus durch Netzwerkprobleme zu Verzögerungen kommen, und außerdem kann der User das Roaming-Verhalten seines Gerätes auch komplett deaktivieren. In diesem Fall steht dem Entwickler die Roaming-Funktion nicht zur Verfügung und muss sich um eine andere Art der Persistierung kümmern. [8], S. 463

4.5. Lokale Daten und Datenbankanbindung

Da - wie im vorherigen Unterkapitel 4.4 erwähnt - nur 100 Kilobyte geroamt werden dürfen, muss man sich als Entwickler eine andere Art der Datensicherung suchen, falls man mehr Informationen speichern will.
Eine Möglichkeit ist es, sogenannte *App Files* anzulegen. Diese sind reguläre Files am normalen Filesystem, auf die andere WinRT Apps jedoch keinen Zugriff haben, da diese in einem speziellen abgetrennten Bereich abgelegt werden. Begrenzungen in der Größe der Dokumente gibt es lokal nicht, die Unterordner-Hierarchie darf lediglich eine Tiefe von 32 nicht überschreiten.

Datenbankanbindung an SQL Server wird derzeit nicht unterstützt - falls eine Datenbank benötigt wird, kann man aber auf File-basierte Datenbanken wie SQLite zurückgreifen.

Auch temporäre Dateien können bei Bedarf sehr einfach erstellt werden. Diese existieren selten länger als eine App-Session lang. [8], S.464ff

4.6. App Bars

Die AppBars sind eine besondere Art von Control und bestehen aus Balken, die sich vom oberen oder unteren Rand der Applikation über die normale Views legen. Standardmäßig sind diese Balken

ausgeblendet, durch einen rechten Mausklick oder eine entsprechende Geste werden sie erst ein-, beziehungsweise wieder ausgeblendet. Inhaltlich sollten diese Bars Buttons enthalten, die zusätzliche Optionen anbieten, sich aber nicht direkt auf der View befinden sollen, weil sie dort zu viel Platz einnehmen oder einfach nicht dorthin passen. Gruppiert werden können diese Buttons in den bereits in 3.5.1 vorgestellten Panels. [8], S.196ff

4.7. Live Tiles

Defaultmäßig wird eine App am Windows 8 oder Windows RT Startbildschirm mit seinem statischen Logo angezeigt. Neu ist allerdings die Unterstützung für sogenannte Live Tiles. Diese bieten einem die Möglichkeit, dynamisch ständig neue Informationen direkt am Startbildschirm anzuzeigen. Auch sekundäre Tiles können erstellt werden, die dem User einen Shortcut auf einen bestimmten Bereich einer App ermöglichen.

Wie ein Live Tile am Bildschim angezeigt wird, wird durch ein Template bestimmt, welches durch einen XML String definiert ist. Es können entweder auf verschiedene lokal gesicherte Informationen zurückgegriffen, oder aber in periodischen Abständen neue Daten aus dem Internet abgefragt werden. Wie Push Notifications dazu beitragen können, wird in 4.9 erklärt.

Über ein Live Tile können auch *Badges* gelegt werden. Diese werden als Overlay dargestellt, die zusätzliche Informationen, wie in etwa die Anzahl der neu eingegangenen Emails bei einer Mailapp, darstellen kann. [8], S.539ff

In Abbildung 4.2 sieht man das Live Tile der *Bing Wetter* Applikation, die regelmäßig aktuelle Daten aus dem Internet abfragt.

Abbildung 4.2.: Live Tile

4.8. Toast Notifications

Toast Notifications sind vergleichbar mit einer Message Box: sie erscheinen im rechten oberen Rand des Bildschirms und machen den User auf etwas Bestimmtes aufmerksam. Ignoriert dieser die Nachricht, verschwindet sie automatisch nach einer bestimmten Zeitspanne.
Toast Notifications werden analog zu Live Tiles durch XML und Templates definiert, angezeigt werden sie entweder periodisch, oder programmatisch im Code. [8], S.552ff

4.9. Push Notifications

Um in periodischen Abständen aktuelle Informationen aus dem Internet abzufragen, oder um von seinen eigenen Servern Nachrichten an Benutzer einer Applikation senden zu können, kann man das Windows Push Notification Service (WNS) nutzen.

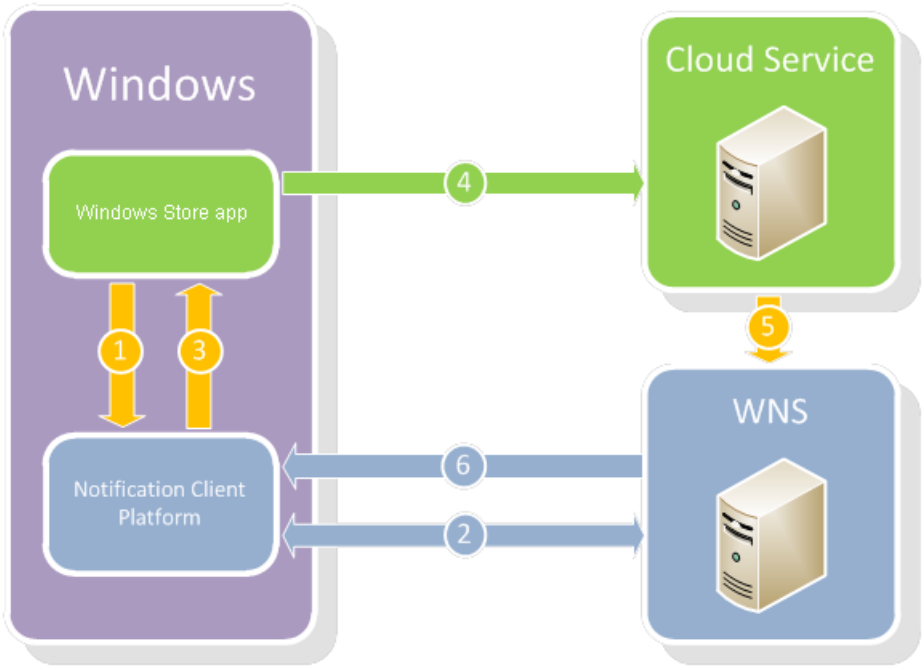

Abbildung 4.3.: Nutzung des Push Notification Service

Dieses funktioniert, wie auch in Abbildung 4.3 visualisiert wird, folgendermaßen:

- Die Applikation sendet eine Push Notification Request an die Notification Client Plattform und fragt um einen Kommunikationskanal an.
- Die Notification Client Plattform leitet diese Anfrage an das WNS weiter, welches diesen im Erfolgsfall als Uniform Resource Identifier (URI) zurückliefert.
- Diese URI wird jetzt wieder an die aufrufende WinRT App returniert. Sie kann an den eigenen Cloud-Dienst weitergeleitet werden und ab sofort als Kommunikationskanal zwischen Cloud Service und Applikation dienen.
- Sobald der Cloud Service eine Update-Nachricht zur Verfügung hat, benachrichtig dieses den WNS über die erhaltene URI und eine HTTP POST Request. Dieses liefert die erhaltene Nachricht anschließend an das referenzierte Gerät aus.

Damit eine Applikation überhaupt Push Notifications empfangen kann, muss diese für diesen Dienst registriert sein. Die Authentifikation geschieht dann über das OAuth 2.0 Protokoll. [9]

4.10. Sensoren

Tablets bieten Unterstützung für eine ganze Reihe von zusätzlicher Hardware an, die stationäre Geräte nicht besitzen und für die diese auch wenig sinnvoll wäre. Darunter können sich folgende Sensoren befinden: ein Beschleunigungsmesser, ein Gyrometer, ein Abweichungsmesser, ein Kompass, ein Lichtsensor, ein Orientierungssensor, ein Ortungssensor oder ein NFC-Chip. [8], S.529ff Die Anzahl der Möglichkeiten ist also so groß wie noch nie. Es macht oftmals keinen Sinn, Unterstützung für alle der oben genannten Sensoren anzubieten, aber je nach Applikation sind diese zusätzlichen Optionen eine echte Bereicherung.

5. Praktisches Beispiel anhand einer einfachen Line Of Business Applikation

Infos über Motivation für Applikation, Highlights der Anwendung präsentieren, im vorherigen Abschnitt erstelle Custom Control implementieren, Verwendung von Schnittstellen ...
- Suche implementieren für Patienten / Medis
- Share: Kontakt wählen, als Patient hinzufügen (mit App teilen) bzw. Info per Mail teilen (Graph an Patient) - thumbnail on list?

Literaturverzeichnis

[1] "EMEA PC Shipments Contracted 20.2% in 1Q13 as Consumers Continue to Divert Spending to Tablets While Commercial Spending Remains Constrained, says IDC," Press Release, IDC, 04 2013, http://www.idc.com/getdoc.jsp?containerId=prUK24077313 [Zugang am 20.04.2013].

[2] "Windows RT: Häufig gestellte Fragen," Website, Microsoft Windows, 2013, http://windows.microsoft.com/de-at/windows/windows-rt-faq [Zugang am 20.04.2013].

[3] B. LeBlanc, "Announcing the Windows 8 Editions," Blog, Microsoft Windows, 04 2012, http://blogs.windows.com/windows/b/bloggingwindows/archive/2012/04/16/announcing-the-windows-8-editions.aspx [Zugang am 20.04.2013].

[4] D. Seven, "A bad picture is worth a thousand long discussions." Blog, 09 2011, http://dougseven.com/2011/09/15/a-bad-picture-is-worth-a-thousand-long-discussions/ [Zugang am 21.04.2013].

[5] S. Farkas, "," Website, 2012, http://msdn.microsoft.com/en-us/magazine/jj651569.aspx [Zugang am 24.04.2013].

[6] "Windows Hardware Certification Requirements: Client and Server Systems," PDF, 09 2012, http://msdn.microsoft.com/library/windows/hardware/hh748188 [Zugang am 23.04.2013].

[7] T. C. Huber, *Windows Presentation Foundation - Das umfassende Handbuch*, 2nd ed. Galileo Press, Bonn, 2010.

[8] A. Nathan, *Windows 8 Apps with XAML and C# UNLEASHED*, 1st ed. Sams Publishing, Indiana, USA, 2012.

[9] "Push notification overviw (Windows Store Apps)," Website, 10 2012, http://http://msdn.microsoft.com/en-us/library/windows/apps/hh913756.aspx [Zugang am 04.05.2013].

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CLR	Common Language Runtime
IDC	International Data Corporation
IMEA	Europa, Mittlerer Osten und Afrika
LOB	Line Of Business
PC	Personal Computer
UI	User Interface
URI	Uniform Resource Identifier
WinRT	Windows Runtime
WNS	Windows Push Notification Service
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

A. Generic.xaml

Listing A.1: Generic.xaml

```
<ResourceDictionary
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:pc="using:PractitionerMobile.Controls">

 <Style TargetType="pc:OrdinationMask">
     <Setter Property="Template">
         <Setter.Value>
             <ControlTemplate TargetType="pc:OrdinationMask">
                 <Border
                     Background="{TemplateBinding Background}"
                     BorderBrush="{TemplateBinding BorderBrush}"
                     BorderThickness=
                        "{TemplateBinding BorderThickness}"
                     Width="600"
                     >
                     <Grid Name="OrdinationContentWrapper">
                         <Grid.ColumnDefinitions>
                             <ColumnDefinition Width="200" />
                             <ColumnDefinition Width="370" />
                         </Grid.ColumnDefinitions>
                         <Grid.RowDefinitions>
                             <RowDefinition Height="200" />
                             <RowDefinition Height="80" />
                             <RowDefinition Height="80" />
                             <RowDefinition Height="200" />
                             <RowDefinition Height="50" />
                         </Grid.RowDefinitions>

                         <Image Grid.ColumnSpan="2" Grid.Row="0"
                        Grid.Column="0"
                        Source="{TemplateBinding ImagePath}"
                        Stretch="Uniform" Width="150"
                        Height="150" />

                         <TextBlock Grid.Row="1" Grid.Column="0"
                        Text="Sozialversicherung" Margin="0,20,0,0"
                        FontSize="15" />
                         <ComboBox Grid.Row="1" Grid.Column="1"
```

```
                                Name="SocialInsurance"
                                ItemsSource=
                                "{TemplateBinding SocialInsurances}"
                                Height="60" />

                                 <TextBlock Grid.Row="2" Grid.Column="0"
                                Margin="0,20,0,0" Text="Dauer"
                                FontSize="15" />
                                 <TextBox Grid.Row="2"
                                Grid.Column="1"
                                Height="60" Name="Duration" />

                                 <TextBlock Grid.Row="3" Grid.Column="0"
                                Margin="0,20,0,0" Text="Diagnose"
                                FontSize="15" />
                                 <TextBox Grid.Row="3" Grid.Column="1"
                                  Height="180" Name="Diagnosis" />

                                 <Button x:Name="Ok" Grid.Row="4"
                                Grid.Column="1" Content="OK"
                                HorizontalAlignment="Left"
                                Width="100" Tag="ok" />
                                 <Button x:Name="Cancel" Grid.Row="4"
                                Grid.Column="1"
                                Content="Cancel"
                                HorizontalAlignment="Right"
                                Width="100" Tag="cancel" />
                            </Grid>
                        </Border>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </ResourceDictionary>
```

B. OrdinationMask.cs

Listing B.1: OrdinationMask.cs

```
using System.Collections.Generic;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;

// The Templated Control item template is documented at
// http://go.microsoft.com/fwlink/?LinkId=234235
namespace PractitionerMobile.Controls
{
    /// <summary>
    /// This class contains Practitioner Mobiles' custom control -
    /// a control, in which data of an ordination can be entered.
    /// </summary>
    [TemplatePart(Name = "Ok", Type = typeof(Button))]
    [TemplatePart(Name = "Cancel", Type = typeof(Button))]
    [TemplatePart(Name = "SocialInsurance",
        Type = typeof(ComboBox))]
    [TemplatePart(Name = "Duration", Type = typeof(TextBox))]
    [TemplatePart(Name = "Diagnosis", Type = typeof(TextBox))]
    public sealed class OrdinationMask : Control
    {
        public OrdinationMask()
        {
            this.DefaultStyleKey = typeof(OrdinationMask);
        }

        #region Image
        public ImageSource ImagePath
        {
            get { return (ImageSource)GetValue(ImagePathProperty); }
            set { SetValue(ImagePathProperty, value); }
        }

        // Using a DependencyProperty as the backing store
        // for MyProperty. This enables animation, styling,
        // binding, etc...
        public static readonly DependencyProperty
        ImagePathProperty =
```

```
            DependencyProperty.Register("ImagePath",
                    typeof(ImageSource),
                    typeof(OrdinationMask),
                   new PropertyMetadata(null));
#endregion

#region Social Insurances
private static List<string> _defaultSocialInsurances =
        new List<string>() { "NoeGkk", "Wgkk", "BVA" };

public static readonly DependencyProperty
SocialInsurancesProperty =
      DependencyProperty.Register("SocialInsurances",
      typeof(IEnumerable<string>),
      typeof(OrdinationMask),
      new PropertyMetadata(_defaultSocialInsurances));

public IEnumerable<string> SocialInsurances
{
    get { return (IEnumerable<string>)
        GetValue(SocialInsurancesProperty); }
    set { SetValue(SocialInsurancesProperty, value); }
}
#endregion

#region Button logic
public event TappedEventHandler OnOkButtonHit;
public event TappedEventHandler OnCancelButtonHit;

/// <summary>
/// Calls custom OK event.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
 private void okButton_Tapped(object sender,
        TappedRoutedEventArgs e)
 {
       if(OnOkButtonHit != null)
       {
           OnOkButtonHit(this, null);
       }

      this.ClearFields();
 }

/// <summary>
/// Calls custom event and clears fields.
```

```
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void cancelButton_Tapped(object sender,
                TappedRoutedEventArgs e)
        {
            if(OnCancelButtonHit != null)
            {
                OnCancelButtonHit(this, null);
            }

            this.ClearFields();
        }
        #endregion

        /// <summary>
        /// Sets events to buttons.
        /// </summary>
        protected override void OnApplyTemplate()
        {
            base.OnApplyTemplate();

            Button okButton = (Button)GetTemplateChild("Ok");
            Button cancelButton= (Button)GetTemplateChild("Cancel");

            okButton.Tapped += okButton_Tapped;
            cancelButton.Tapped += cancelButton_Tapped;
        }

        /// <summary>
        /// Clears all fields of the control.
        /// </summary>
        private void ClearFields()
        {
            ComboBox socialInsurance =
                (ComboBox)GetTemplateChild("SocialInsurance");
            TextBox duration = (TextBox)GetTemplateChild("Duration");
            TextBox diagnosis = (TextBox)GetTemplateChild("Diagnosis");

            socialInsurance.SelectedIndex = -1;
            duration.Text = string.Empty;
            diagnosis.Text = string.Empty;
        }
    }
}
```