SWE2- Bericht

Dieser Bericht dient als Kompensation, weil es dazu gekommen ist, dass ich nicht einmal die Lösung in einer Besprechung präsentiert habe. Er beinhaltet die Präsentation meines Projektes in schriftlicher Form und die wesentlichen Lernziele des Übungsprojekts.

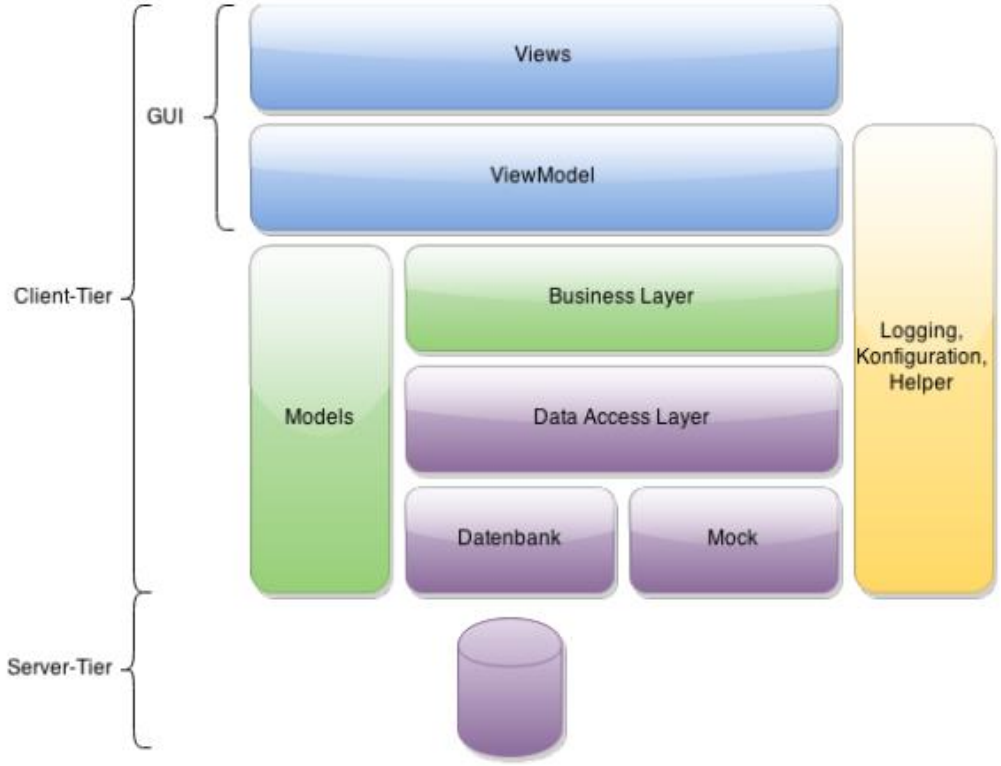
Die Aufgabenstellung ist es eine Fotodatenbank mit Hilfe einer GUI (WPF/JAVAFX) zu realisieren. Ich habe mich dazu entschieden, das Windows Presentation Foundation (WPF) Framework unter .NET zu wählen und programmierte daher mit C#. Dabei mussten wir das Entwurfsmuster Model-View-ViewModel (MVVM) implementieren, um eine 3-Layer Architektur bei der Übung umzusetzen.

MVVM (Model-View-ViewModel)

Das MVVM Design Pattern ist an dem MVC-Muster angelehnt, da es die funktionale Trennung von Model und View auch nutzt. Mit Hilfe von Datenbindung wird die Trennung von UI und Programmcode erst ermöglicht. Es besteht aus den drei folgenden Komponenten:

- Model - die Daten, die dargestellt werden sollen
- View - die Darstellung an sich
- ViewModel - Vermittler zwischen Model & View

Aufbau der Übung



Das MVVM Pattern benützt Datenbindungsmechanismen, um eine lose Kopplung zwischen den Komponenten zu erreichen.

VORTEILE

- Die „Business“-Logik kann unabhängig von der Darstellung bearbeitet werden.
- Die Testbarkeit verbessert sich, da die ViewModel die UI-Logiken enthalten und unabhängig von der View instanziiert werden können. Hierdurch sind keine (in der Regel manuellen) UI-Tests nötig. Stattdessen genügen codebasierte Modultests des ViewModel.
- Die Menge von Glue Code zwischen Model und View wird durch den Verzicht von Controller-Instanzen gegenüber dem MVC-Muster reduziert.
- Views können von reinen UI-Designern gestaltet werden, während Entwickler die Models und ViewModels implementieren können.
- Es können unterschiedliche Views erstellt werden, die alle an dasselbe ViewModel gebunden werden, ohne dass dort Änderungen an der Programmierung vorgenommen werden müssen.

NACHTEILE

- Der generische Datenbindungsmechanismus erspart die Implementierung von verschiedenen Controllern. Allerdings ist dieser Mechanismus zur Umsetzung des MVVM-Musters zwingend erforderlich.

MODEL

Das Model ist nur für den Dateninhalt zuständig. Das Model bildet die Struktur der Anwendungsdomäne ab und kennt außer dem BusinessLayer niemanden.
In der ersten Übung habe ich alle notwendigen Model Komponenten der Fotodatenbank erstellt.

- PictureModel
- IPTCModel
- EXIFModel
- PhotographerModel
- CameraModel

```
class PictureModel : IPictureModel
{
    public PictureModel() {}
    public PictureModel(string filename)
    {
        FileName = filename;
    }

    /// <summary>
    /// Database primary key
    /// </summary>
    public int ID { get; set; }

    /// <summary>
    /// Filename of the picture, without path.
    /// </summary>
    public string FileName { get; set; }

    /// <summary>
    /// IPTC information
    /// </summary>
    public IIPTCModel IPTC { get; set; } = new IPTCModel();

    /// <summary>
    /// EXIF information
    /// </summary>
    public IEXIFModel EXIF { get; set; } = new EXIFModel();

    /// <summary>
    /// The camera (optional)
    /// </summary>
    public ICameraModel Camera { get; set; } = new CameraModel();
}
```

Abbildung 1

Auf der *Abbildung 1* kann man sehr gut erkennen, dass die Model Komponente aus Properties, also *nur aus Daten, besteht*, die der Benutzer im späteren Verlauf auch manipuliert. Die View-Logik wird hier komplett vernachlässigt.

VIEWMODEL

Die ViewModel-Komponente enthält die UI-Logik und verbindet das Model mit der View. Sie transformiert ein Model in eine anzeigbare Form und berücksichtigt dabei die Gegebenheiten auf einer UI. Außerdem stellt sie weitere Eigenschaften & Methoden, die für die Anzeige benötigt werden zur Verfügung.

In der zweiten Übung musste ich alle notwendigen ViewModel-Klassen erstellen.

- CameraListViewModel
- CameraViewModel
- EXIFViewModel
- IPTCViewModel
- MainWindowViewModel
- PhotographerListViewModel
- PictureListViewModel
- PictureViewModel
- SearchViewModel

Auf der *Abbildung 2* kann man gut erkennen, dass sich die Logik für die UI hier befindet. Außerdem sieht man, dass die Verbindung zum Model durch den BusinessLayer hergestellt wird.

```
class PhotographerListViewModel : ViewModel, IPhotographerListViewModel
{
    private BusinessLayer _bl;

    public PhotographerListViewModel()
    {
        _bl = new BusinessLayer(DALFactory.CreateDAL());
        List<PhotographerModel> PMList = (List<PhotographerModel>)_bl.GetPhotographers();
        List<IPhotographerViewModel> PVMList = new List<IPhotographerViewModel>();

        foreach (var item in PMList)
        {
            PVMList.Add(new PhotographerViewModel(item));
        }

        List = PVMList;
        CurrentPhotographer = List.First();
    }
    /// <summary>
    /// List of all PhotographerViewModel
    /// </summary>
    public IEnumerable<IPhotographerViewModel> List { get; set; }

    /// <summary>
    /// The currently selected PhotographerViewModel
    /// </summary>
    public IPhotographerViewModel CurrentPhotographer
    {
        get => photographer;
        set
        {
            if (photographer != value)
            {
                photographer = value;
                //MessageBox.Show("new List");
                OnPropertyChanged("CurrentPhotographer");
            }
        }
    }

    private IPhotographerViewModel photographer;
}
```

Abbildung 2

VIEW

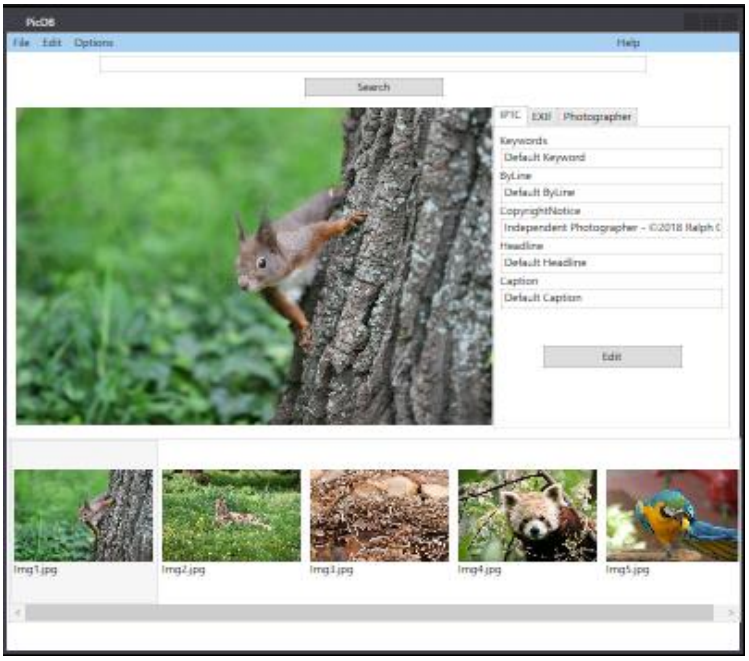


Abbildung 3

Dieser Bereich beschreibt alle angezeigten Elemente der grafischen Benutzeröberfläche. Die Daten, die die View anzeigt, werden aus dem ViewModel dargestellt und nie aus dem Model selbst. Durch Datenbindung ist die View einfach austauschbar und ihr Code dahinter gering. Wir haben zwischen WPF und JavaFX wählen dürfen, und da ich in C# programmiere, habe ich die WPF-Variante gewählt. Mit Visual Studio kann die WPF entweder mit Hilfe ihrer Benutzeroberfläche oder durch XAML-Code realisiert werden. XAML bedeutet Extensible Application Markup Language und im XAML wird das Aussehen definiert und im Code die Logik. Es kann auch das Aussehen im Code definiert werden und das Praktische mit MVVM ist, dass der „Code-Behind“ nicht notwendig zu implementieren ist. Außerdem bevorzuge ich es die GUI mit dem XAML-Code zu bearbeiten, da es ein wenig mehr Überblick über alle Elemente verschafft. Bei XAML werden Controls (UI-Elemente), wie z.B. TextBlock, TextBox, ComboBox, Button, usw., mit Hilfe von Layouts

positioniert. Zu Layouts zählen zum Beispiel Border, Canvas, DockPanel, Expander, Grid, GroupBox, ScrollViewer, StackPanel.

Die UI-Elemente können mittels DataBinding verknüpft werden. Was ist DataBinding denn überhaupt? Es ist ein Prozess, welcher eine Verbindung zwischen der Applikations-UI und der Geschäftslogik herstellt. Wenn das Binden richtig implementiert wurde und die Daten eine Benachrichtigung bereitstellen, werden sich die an die Daten gebundenen Elemente automatisch bei jeder Änderung des Werts der Daten anpassen. Es kann aber auch bedeuten, dass Daten auf der UI sich automatisch aktualisieren, wenn sie geändert werden. Da gibt es nun drei Varianten: Daten werden mit Controls gebunden, Controls werden mit Daten gebunden oder Controls werden mit Controls gebunden. Dies erfolgt normalerweise deklarativ im XAML, ist aber auch im Code möglich. Benachrichtigungen erfolgen normalerweise über die INotifyPropertyChanged- oder DependencyProperties-Klasse.

Bevor man Daten binden kann, muss man den entsprechenden DataContext setzen. In meinem Projekt habe ich den DataContext im MaindWindow.xaml.cs auf eine MainWindowViewModel-Instanz gelegt.

WPF – Windows Presentation Foundation

Windows Presentation Foundation ist ein Grafik-Framework und Fenstersystem des .NET Frakeworks. Es bietet Entwicklern ein einheitliches Programmiermodell zum Erstellen von Desktop Line-of-Business-Anwendungen unter Windows. Die WPF-Entwicklungsplattform unterstützt eine breite Palette an Anwendungsentwicklungsfeatures, darunter ein Anwendungsmodell, Ressourcen, Steuerelemente, Grafik, Layout, Datenbindung, Dokumente und Sicherheit. Es ist eine Teilmenge von .NET Framework, wenn man also bisher Anwendungen mit dem .NET Framework mithilfe von ASP.NET oder Windows Forms erstellt hat, sollte einem die Programmiererfahrung nicht unbekannt sein. WPF verwendet XAML (Extensible Application Markup Language), um ein deklaratives Modell für die Anwendungsprogrammierung bereitzustellen. Alternativ kann auch JavaFX verwendet werden, ich arbeite allerdings gerne mit C#.

Wie WPF in meinem Projekt verwendet wurde, habe ich bereits weiter oben erklärt.

DataAccessLayer

Die DataAccessLayer-Klasse ist dafür da, dass die Verbindung zur gegebenen Datenbank hergestellt wird. Weiters enthält die Klasse die gewollten Funktionen mit der man die Datenbank gewünscht manipuliert. Bei diesem Projekt war es unsere Aufgabe neben dem „richtigen“ DAL auch einen MockDAL anzufertigen. Da wir die Models, ViewModels und den BusinessLayer schon vor der Datenbankverbindung implementiert haben, war es von Vorteil einen MockDAL anzufertigen, um die erstellten Komponenten des MVVM-Models durch die gegebenen Unit-Tests zu überprüfen.

Ich habe neben dem MockDAL und dem „richtigen“ DAL noch eine Factory implementiert, die es mir vereinfacht, zwischen den jeweiligen DALs auszuwählen.

Wie auch im letzten Jahre habe ich mich dazu entschieden eine MSSQL Datenbank zu verwenden. Man stellt eine Verbindung zur Datenbank her. Dazu verwendet man folgenden Befehl:

```
private readonly string _connectionstring = $"Server={Settings.Default.Servername}; Database=PicDB; Integrated Security=True;";
public SqlConnection Connection => new SqlConnection(_connectionstring);
```

Abbildung 4

Durch diese Verbindung hat man den Zugriff zur Datenbank ermöglicht. Dies ist besonders wichtig, da jedes Foto verschiedene Metadaten hat (wie beispielsweise Name, ISOValue, Fotograf usw.), um diese nicht wirklich zu editieren, mussten wir das Editieren der Metadaten simulieren.

Dabei haben wir verschiedene Funktionen verwendet wie zum Beispiel GetPicture, GetPhotographer, GetPictures, DeletePictures, DeletePhotographer, usw. Einige Funktionen benötigten dafür das Erstellen einer Stored Procedure, um Fehler zu vermeiden.

```
try
{
    SqlConnection connection = Connection;
    connection.Open();

    SqlCommand command = new SqlCommand("SelectAllPhotographers", connection) { CommandType = CommandType.StoredProcedure };
    command.ExecuteNonQuery();

    List<PhotographerModel> ResultList = new List<PhotographerModel>();
    SqlDataReader reader = command.ExecuteReader();

    while (reader.Read())
    {
        PhotographerModel result = new PhotographerModel();

        result.ID = reader.GetInt32(0);
        result.FirstName = reader.GetString(1);
        result.LastName = reader.GetString(2);
        result.BirthDay = reader.GetDateTime(3);
        result.Notes = reader.GetString(4);

        ResultList.Add(result);
    }
    connection.Close();
    return ResultList;
}
```

Abbildung 5

Wie man an der oberen Abbildung sieht verwende ich die Verbindung zur Datenbank und greife in diesem Fall auf den Stored Procedure mit dem Namen „SelectAllPhotographers“ zu und führe diesen Befehl aus. Mithilfe des SQLDataReader Objekts kann ich auf die gewünschten Daten zugreifen und sie anschließend wieder zurückgeben.

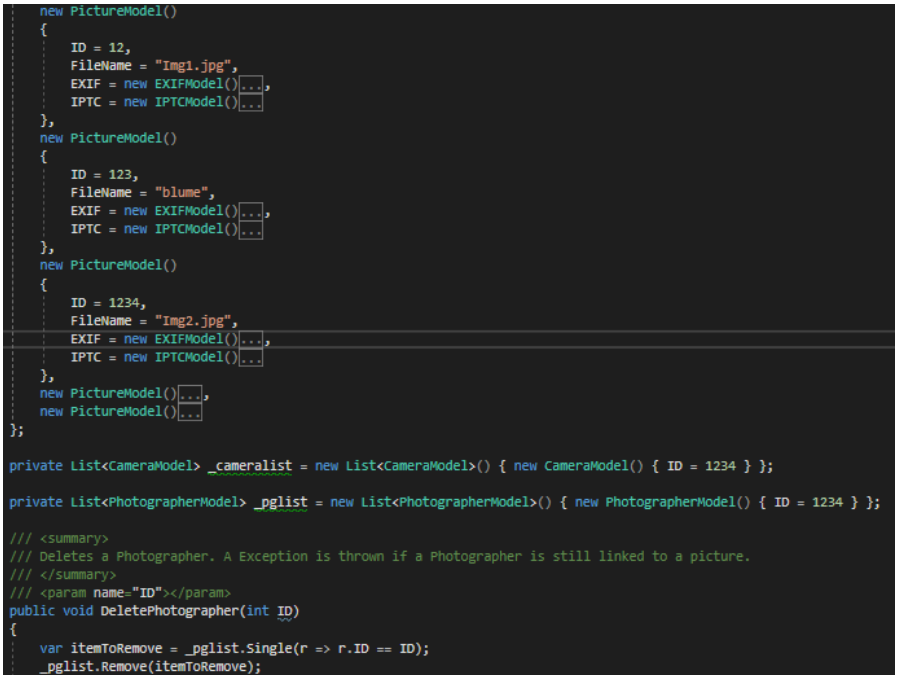
BUSINESSLAYER

Die BusinessLayer-Klasse ist die Kommunikation zwischen DataAccessLayer und ViewModel. Diese Klasse wird im Konstruktor ein DAL als Parameter übergeben, da wir zwischen MockDAL und DAL unterscheiden müssen. Der BusinessLayer ist zuständig für alle Geschäftsregeln und daher werden diese Regeln alle in dieser Klasse implementiert. Der BusinessLayer ist weder zuständig für die GUI noch für den Datenzugriff.

Wie bereits erwähnt ist das BusinessLayer essentiell für die Verbindung zwischen DataAccessLayer und Viewmodel. Es verwendet die Funktionen, welche vom DataAccessLayer bereitgestellt werden, das bedeutet, dass das BusinessLayer daher keienn direkten Zugriff auf die Datenbank hat. Außerdem bringt es den Vorteil, dass man den Businesslayer mit einem MockDAL füttern kann, um die Funktionen des Viewmodels zu testen. Diesen Vorteil konnten wir auch beim Projekt nutzen.

UNITTEST – ÜBUNGEN

In der ersten Übung habe ich alle nötigen Klassen und die Struktur des MVVM-Models erstellt. Dazu gehören alle Models (CameraModel.cs, EXIFModel.cs, IPTCModel.cs, PhotographerModel.cs, PictureModel.cs) und auch alle ViewModels, zu jedem Model habe ich auch ein ViewModel angelegt, beispielsweise: CameraViewModel. Bei den ViewModels kamen nur ein paar Klassen dazu, die für die Auflistung von Objekten zuständig sind, wie z.B. PictureListViewModel. Neben dem erstellten BusinessLayer mussten wir einen Mockup-DataAccessLayer erstellen, um die bereits erstellten Klassen durch die Unittests, schon vor der eigentlichen Verbindung mit einer Datenbank, testen zu können. Nachdem wir die Struktur bereitgestellt hatten, habe ich die Klassen mit deklarierten und initialisierten Objekten gefüllt, sodass Model mit ViewModel verbunden werden. Die Models beinhalten dabei reine Informationen, die



von Logik befreit sind. Die Viewmodels hingegen beinhalten die Logik beziehungsweise Funktionen, die von der View benötigt werden. Um die Funktionen auf Richtigkeit zu testen haben wir wie bereits erwähnt einen MockDAL erstellt.

Die Abbildung (Abb. 6) links bringt einen Einblick ins MockDAL. Ich habe drei Listen angelegt, eine List mit Picture Models, eine

mit Camera Models und eine mit Photographer Models. Wie man im Bild erkennen kann habe ich die Picture Model Liste mit Objekten gefüllt und sie mit Informationen bereitgestellt, um eine Datenbank zu simulieren. Die Funktionen des MockDALs entnehmen ihre nötigen Objekte aus den Listen und nicht aus einer Datenbank. Auf diese Art und Weise wurde der MockDAL in unserem Projekt zur Verwendung gebracht.

IMAGEMANAGER

Ich habe die Singleton Klasse ImageManager implementiert, um die Bilder im gewählten Verzeichnis mit simulierten Informationen in der Datenbank zu speichern. Dabei hole ich mir mit der Directory.GetFiles()-Funktion alle Bilder aus dem gewählten Verzeichnis und extrahiere aus dem Rückgabewert (welcher normalerweise der gesamte Pfad des Bildes ist) mit der Funktion Path.GetFileName() nur den Namen aus dem Pfad heraus. Bevor ich das Bild letztendlich in der Datenbank speichere, prüfe ich, ob das Bild schon in der Datenbank ist, um nur neu hinzugefügte Bilder in die Datenbank zu speichern. (Siehe Abbildung 7)

```
public void CreateDefaultPictureModels()
{
    string[] imgFiles = Directory.GetFiles($"{FilePath}", "*.jpg").Select(Path.GetFileName).ToArray();
    DataAccessLayer DAL = new DataAccessLayer();
    int count_dal = DAL.GetPictures(null, null, null, null).Count();
    int count_folder = imgFiles.Count();

    if (count_dal != 0 && count_dal != count_folder)
    {
        foreach (var item in imgFiles)
        {
            PictureModel Default = new PictureModel();
            IEnumerable<IPictureModel> CheckList = DAL.GetPictures(item, null, null, null);

            if (CheckList.Count() == 0)
            {
                Default.FileName = item;
                Default.EXIF.Make = "Default Make";
                Default.EXIF.FNumber = 0;
                Default.EXIF.ExposureTime = 0;
                Default.EXIF.ISOValue = 0;
                Default.EXIF.Flash = false;
                Default.IPTC.Keywords = "Default Keyword";
                Default.IPTC.ByLine = "Default ByLine";
                Default.IPTC.CopyrightNotice = "Independent Photographer - ©2018 Ralph Quidet, all rights reserved";
                Default.IPTC.Headline = "Default Headline";
                Default.IPTC.Caption = "Default Caption";

                DAL.Save(Default);
                ImageList.Add(Default);
                CheckList = null;
            }
```

Abbildung 7

CommandHandler

Neben dem DataBinding-Mechanismus existieren Commands, um zwischen ViewModel und View zu kommunizieren. Deswegen habe ich die RelayCommand-Klasse erstellt, die von ICommand ableitet. Es ist gewöhnlich das ICommand Interface für das MVVM-Model zu nutzen. Man kann es sich so, wie ein Event vorstellen: „Wird ein Button A gedrückt, werden bestimmte Informationen verändert.“ Folgende Commands habe ich in meinem Projekt implementiert:

- SetPhotographer_command
- AllTags_command
- NewReport_command
- EditIPTC_command

```
public class RelayCommand : ICommand
{
    private Action _action;
    private bool _canExecute;

    /// <summary>
    ///
    /// </summary>
    /// <param name="action"></param>
    /// <param name="canExecute"></param>
    public RelayCommand(Action action, bool canExecute)
    {
        _action = action;
        _canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
    {
        return _canExecute;
    }

    /// <summary>
    ///
    /// </summary>
    public event EventHandler CanExecuteChanged;

    /// <summary>
    ///
    /// </summary>
    /// <param name="parameter"></param>
    public void Execute(object parameter)
    {
        _action();
    }
}
```

Abbildung 8

Wie bereits erwähnt habe ich vier Commands erstellt, um beispielsweise einen Bericht zu erstellen. Zur Aktivierung dieser vier Commands war es notwendig die RelayCommand Klasse zu erstellen.

Bei der Klasse wird mithilfe einer Action eine Funktion im Konstruktor weitergegeben. Des Weiteren wird auch ein Bool, wie man dem Bild (Abb. 8) entnehmen kann, „canExecute“ weitergegeben, der angibt, ob die Aktion ausgeführt werden soll oder nicht.

Beim zweiten Bild (Abb. 9) wird der RelayCommand benützt und wie man sehen kann wird die private NewReport Funktion als Action und der Boolwert „true“ weitergegeben.

```
private void EditIPTC()
{
    _bl.EditIPTC(Picture);
}

/// <summary>
/// Command to create a report
/// </summary>
public ICommand NewReport_command
{
    get { return _command2 ?? (_command2 = new RelayCommand(() => NewReport(), true)); }
}

private void NewReport()
{
    _bl.NewReport(Picture);
}
```

Abbildung 9

Um die Funktion zu aktivieren muss man im MainWindow einen gewissen Button betätigen. Das sieht man in der Abbildung unterhalb (Abb. 10).

```
<Button Width="150" Height="25" Margin="40" Command="{Binding CurrentPicture.EditIPTC_command}">Edit</Button>
```

Abbildung 10

Zu jedem Command habe ich eine StoredProcedure in die Datenbank hinzugefügt. Beim SetPhotographer_command habe ich es erst in einem falschen ViewModel erstellt und verbrachte einige Zeit damit herauszufinden, woran mein Fehler liegen könnte. Erst wollte ich das Problem mit MultiBinding lösen, schließlich fiel mir jedoch auf, dass ich den Command im MainViewModel erstellen sollte, da ich dort Zugriff auf den aktuellen Fotografen und das aktuelle Bild hatte. - Problem solved –

REPORT PDF

Um eine Zusammenfassung der Bildinformationen zur Verfügung zu stellen habe ich mich für das NuGet Package IronPDF entschieden. Ich habe mich auch mit anderen Packages befasst und versucht diese zu verwenden, allerdings hat sich IronPDF als das unkomplizierteste Package erwiesen.

Dabei habe ich in der ReportPDF Klasse zwei Funktionen geschrieben und zwar: PictureString und AllTags. Die PictureString Methode gibt einen großen String zurück, der einen HTML Code beinhaltet. Dieser String wird dann mithilfe von IronPDF-Funktionen in ein PDF umgewandelt und bei Betätigung des dafür zuständigen Buttons erscheint ein neues PDF-Dokument mit allen Metainformationen des aktuellen Bildes.

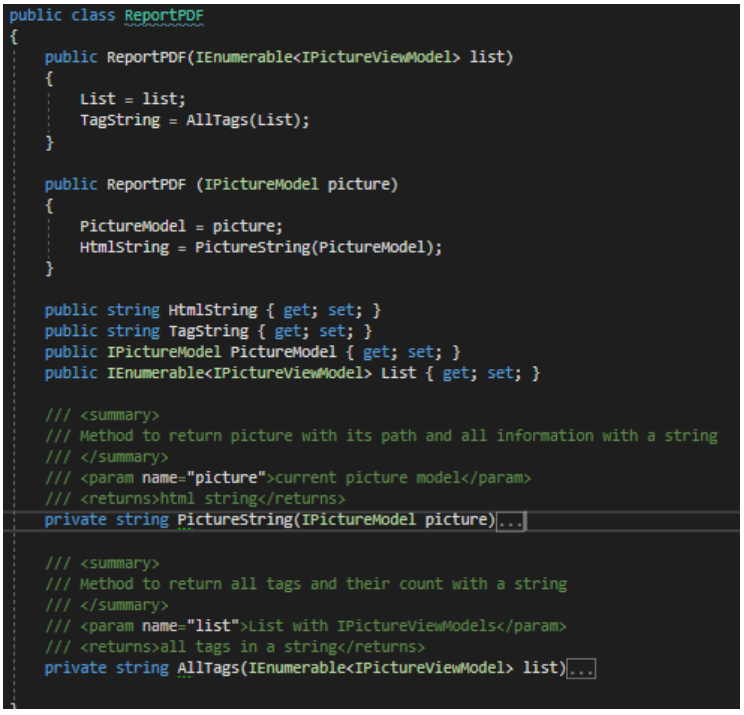


Abbildung 11

Es war neben dem Bildbericht auch die Aufgabe eine PDF-Datei für alle benützten Tags zu erstellen. Deshalb habe ich in dieser Klasse auch die AllTags-Funktion realisiert, die wie die vorherige Funktion einen String zurückgibt. Dieser String beinhaltet alle Tags und deren Anzahl.

Beide Funktionen werden von dem BusinessLayer benützt und können von dort von ViewModels verwendet werden.

DAS FERTIGE PROGRAMM

Wenn die Applikation über das Exe-File gestartet wird, öffnet sich der Picture-Viewer. Das Programm entnimmt aus dem gewählten Verzeichnis alle die Filenamen aller JPG-Daten, die noch nicht in der Datenbank gespeichert wurden und fügt sie der Datenbank hinzu. Mein Picture-Viewer erstellt für jedes Bild ein Objekt in der Datenbank und generiert dem Bild Default-Informationen. Ich habe das manipulieren der EXIF- und IPTC-Daten simuliert und extrahiere nicht die echten Metainformationen aus den Bildern heraus.

Die Bilder erscheinen im Programm auf der unteren Seite als Thumbnail-Liste. Das aktuelle Bild wird etwas größer mit ihren EXIC- und IPTC Daten, die rechts vom Bild sind, angezeigt. Ich habe rechts vom Bild drei TabItems hinzugefügt, die IPTC – EXIF und Photographers anzeigen.
Im IPTC Tab können die Daten mit dem Edit Button auch in der Datenbank geändert werden, während die Textblöcke im EXIF-Tab nur gelesen werden können.

Klickt man auf den Photographer Tab kann man oben auf der ComboBox alle Fotografen auflisten lassen. Die Informationen des ausgewählten Fotografen wird dann unter der ComboBox angezeigt. Die Daten des Fotografen können, wie bei den IPTC-Daten mit einem Edit-Button geändert werden.
Unter den Informationen ist noch ersichtlich welcher Fotograf das Bild geschossen hatte. Ist dem Bild ein Fotograf zugeordnet, erscheint „Filename.jpg (by Vor- und Nachname). Ist dem Bild kein Fotograf zugeordnet, erscheint auch kein Name. Wenn der User sich dazu entschließt, dem aktuellen Bild einen Fotografen zuzuordnen, muss er oben auf der ComboBox den gewünschten Fotografen aussuchen und anschließend auf den „Set Photographer to Picture“-Button klicken. Dann wird dem aktuellen Bild den aktuellen Fotografen als Fotograf hinzugefügt.

Auf dem oberen Teil des Programms habe ich auch eine Suchfunktion implementiert. Dazu muss man die gewünschte Suche in die Suchleiste eintippen und dann den „Search“-Button betätigen. Die Suchergebnisse werden unten auf der Thumbnail-Liste angezeigt.
Links oben auf dem Picture-Viewer ist noch das „File“ Menu ersichtlich. Dort ist es dem User möglich, entweder vom aktuellen Bild einen PDF-Bericht zu erstellen oder einen Bericht mit allen benützten Bilder-Tags, mit ihrer Anzahl zu erstellen. Beide PDF-Berichte werden bei Betätigung der Funktion als PDF-Bericht im deploy Ordner gespeichert und anschließend vom Standard-PDF-Viewer geöffnet.

Was würde ich das nächste Mal anders machen

Ich habe wieder mal den Zeitaufwand des Projekts unterschätzt. Das eigentliche Problem liegt jedoch im Unverständnis des MVVM Models. Wenn ich das Projekt noch einmal würde, würde ich das Projekt in dieser Reihenfolge machen: Models -> View (UI Elemente) -> ViewModels / BusinessLayer -> MockDAL -> DAL -> Commands. Ich war etwas im Chaos und arbeitete mal hier und mal dort ohne die Materie wirklich zu verstehen. Jetzt am Ende würde ich sagen, dass ich die Struktur am Anfang des Projekts wirklich verstehen müsste, um effizienter arbeiten zu können.

Außerdem habe ich bei dem Versuch das Extrahieren der echten Metainformationen auch ein wenig Zeit verschwendet. Ich würde dennoch aus reinem Interesse das Extrahieren der EXIF- & IPTC-Daten realisieren und vervollständigen. Mich stört es ein wenig, dass ich bei meiner Code-Konvention nicht sehr konsistent war/bin. Ich möchte in Zukunft in meiner ausgewählten Konvention konsistent werden.