ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Fraunhofer
IZFP

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Kommunikationsinformatik
der Fakultät für Ingenieurwissenschaften

Entwicklung einer Benutzerschnittstelle für die Bedienung von Prüfhardware zur Lichtmastenprüfung

vorgelegt von
Tobias Müller

betreut und begutachtet von
Prof. Dr. Helmut G. Folz

Saarbrücken, 30.09.2016

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 30.09.2016

Tobias Müller

Noch eine Tasse (Kaffee) und ich springe auf Warp.

Captain Kathryn Janeway

Danksagung

An dieser Stelle möchte ich einigen Personen danken.

Zuerst möchte ich Prof. Dr. Helmut Folz für die Betreuung und Begutachtung meiner Bachelorarbeit danken. Auch hätte ich wohl ohne ihn nie die Softwareentwicklung für mich entdeckt. Seine persönliche Ratschläge haben mich immer vorangebracht und mir des öfteren aus scheinbar unmöglichen Situationen geholfen.

Dann möchte ich meinen Arbeitskollegen vom Fraunhofer IZFP danken. Christoph Weingard hat mir einen Kindheitswunsch erfüllt, indem er mich für das Institut angeworben hat. Er schafft es, dass ich selbst nach dieser harten Zeit der Thesis noch lachen kann. Michael Ganster, Betreuer meiner Bachelorarbeit und Projektverantwortlicher, hat mich fachlich und menschlich unterstützt. Auf ihn konnte ich mich verlassen, wenn ich nicht weiterkam.

Dann möchte ich meinen Eltern, Klaus Dieter Müller und Susanne Forster, für die große Unterstützung danken. Ohne sie wäre dieses Studium nicht möglich gewesen. Es war nicht immer einfach, aber ihr hattet immer Verständnis für mich.

Abschließend möchte ich meiner Freundin und Liebe meines Lebens danken. Katharina, du bist das Beste, was mir im Leben passieren konnte. Ich weiß nicht, wo ich ohne dich heute wäre.
Diese Arbeit ist unserer gemeinsamen Zukunft gewidmet.

Tobias Müller,
30.09.2016

Inhaltsverzeichnis

1 Einführung

LimaTest steht für Lichtmastentest und ist eine Kundenanwendung mit langer Tradition. Die Anwendung befindet sich bereits seit zwei Jahrzehnten im täglichen Einsatz, um Lichtmasten in ganz Deutschland auf Korrosion und Beschädigungen unterhalb des Erdeintritts zu prüfen.
Die Software wurde im Lauf der Jahre immer wieder angepasst an neue Gegebenheiten und es war bereits 2008 eine Neuentwicklung geplant. Diese wurde jedoch nicht durchgeführt.
Nun ist eine neue Generation von Prüfhardware geplant, welche jedoch nicht in die bestehende LimaTest-Software integriert werden kann. Daher ergab sich eine erneute Gelegenheit, eine moderne Software zu entwickeln, die zudem leicht erweiterbar ist.

1.1 Technischer Hintergrund

Mithilfe der LimaTest-Software wird die Prüfhardware gesteuert und parametriert. Die Prüfung erfolgt durch elektromagnetischen Ultraschall (EMUS), dessen Ergebnis die Prüfhardware auswertet und über eine Netzwerkverbindung an die LimaTest-Software sendet. Die empfangenen Messdaten, im folgendem auch Ascan genannt, werden dem Anwender visuell dargestellt. In der Visualisierung kann der Anwender, oder auch Prüfer genannt, Korrosion und Beschädigungen aufgrund höherer Energiewerte an den entsprechenden Stellen erkennen. Liegt ein Defekt vor, wird ein Austausch des Masten geplant.

1.2 Ausgangssituation

Es wurde bereits mit der Neuentwicklung der LimaTest-Software begonnen. Die komplette Ansteuerung der Prüfhardware wurde bereits durch Jan Oswald im Rahmen seiner Bachelor-Thesis [Osw16] implementiert. Außerdem wurde eine Bibliothek erstellt, welche das grundlegende Aussehen der Anwendung und der darin enthaltenen Steuerelemente definieren wird.

1.3 Problembeschreibung

Für die Neuentwicklung gibt es mehrere Gründe. Zunächst ist es kaum noch möglich, die Software zu warten. Außerdem ist die Implementierung der Ansteuerung von derzeit geplanter Prüfhardware nicht umsetzbar, da die aktuell im Einsatz befindlichen Prüfhardware-Revisionen zu stark in der LimaTest-Software verankert sind. Eine Refaktorisierung der Software würde vom Aufwand einer kompletten Neuentwicklung gleich kommen.

Wie bereits beschrieben, wurde die Hardwareschnittstelle zu den im Einsatz befindlichen Prüfhardware-Revisionen von Herrn Oswald entwickelt. Nun fehlt eine Schnittstelle für den Anwender, um diese Klassenbibliothek verwenden zu können. Jedoch muss in der neuen Software davon ausgegangen werden, dass neue Hardware-Klassen dazukommen, wenn die geplante Prüfhardware implementiert ist.

1.4 Zielsetzungen

Das Ziel dieser Arbeit ist es, eine grafische Benutzerschnittstelle zu entwickeln, um die Bedienung der Prüfhardware zu ermöglichen. Dabei soll die Bedienung nach aktuellen Konzepten erfolgen. Die durch die Prüfhardware generierten Messdaten sollen dem Anwender direkt visualisiert werden, damit dieser bereits während der Messung Fehlstellen am Prüfkörper erkennen kann.
Der Kunde hat folgende Anforderungen an die neue Software:

- Nach einer Messungen sollen die Messdaten in Bild- und Rohformat abgespeichert werden.
- Mit der Alt-Software erfolgte Messungen müssen nicht mit der neuen Software geöffnet werden können.
- Mit der neuen Software erfolgte Messungen sollen jedoch wieder geladen werden können.
- Der normale Anwender soll nur bestimmte Prüfparameter verändern dürfen.

Durch das Institut werden ebenfalls Anforderungen an die neue Software gestellt. Dies lauten folgendermaßen:

- Für einen internationalen Vertrieb der Software soll eine Sprachumschaltung möglich sein.
- Es soll eine Benutzerschnittstelle integriert werden, welche dem IZFP-Service vorbehalten ist. Darüber sollen spezielle Funktionen der Prüfhardware ausgeführt werden können.
- Bestimmte Prüfparametersätze sollen zum Zweck der Instandhaltung für den normalen Anwender gesperrt sein.

1.5 Abgrenzung

In dieser Arbeit wird nicht auf die Funktionsweise von elektromagnetischem Ultraschall eingegangen. Wenn jedoch entsprechende Begrifflichkeiten Verwendung finden, werden diese an entsprechender Stelle erklärt.
Es wird zudem nicht auf die Evaluierung von anfallenden Daten eingegangen. Auch ist keine automatisierte Auswertung geplant.
Die Hardware-Klassenbibliothek wird ebenfalls nicht genauer betrachtet. Die interne Struktur, das Senden und Empfangen von Daten, sowie deren Verarbeitung ist ein Bestandteil der Bachelor-Thesis von Jan Oswald.

2 Grundlagen

In diesem Kapitel wird auf das verwendete Framework Windows Presentation Foundation und einige seiner Techniken eingegangen. Das Entwurfsmuster Model-View-ViewModel, welches oft im Kontext des Frameworks genannt wird, wird ebenfalls kurz behandelt.

2.1 Windows Presentation Foundation

Das Framework Windows Presentation Foundation, kurz WPF, ist eine Technologie von Microsoft zur Entwicklung von Desktop-Anwendungen. Es wurde 2006 veröffentlicht und enthält eine „gänzlich neu eingeführte Bibliothek von Klassen, die zur Gestaltung von Oberflächen und zur Integration von Multimedia-Komponenten und Animationen dient. Sie vereint die Vorteile von DirectX, Windows Forms, Adobe Flash, HTML und CSS.“ [The12, S. 13]

Anders als beim Vorgänger Windows Forms, ist eine komplette Trennung von Darstellung und Programmcode möglich. Dazu bietet WPF mit der Extensible Application Markup Language (XAML) eine Sprache zur Beschreibung der Darstellung, beginnend beim Layout der Steuerelemente bis hin zu komplexen Animationen. Um eine lose Kopplung zwischen Darstellung und Programmlogik zu erreichen, gibt es verschiedene Techniken, wie beispielsweise die Datenbindung, Kommandos und Konverter. Trotz allem ist manchmal ein Eingriff in die Code-Behind-Datei notwendig, beispielsweise zum Erstellen von Screenshots der aktuellen Ansicht.

Eine große Stärke von WPF ist die Anpassungsfähigkeit der visuellen Objekte. Das Aussehen eines jeden Steuerelementes kann vollständig verändert werden, um beispielsweise dem Corporate Design gerecht zu werden. Dies wird durch sogenannte Styles bewerkstelligt. Sie sind typgebunden und besitzen eine Vererbungshierarchie, die parallel zur Hierarchie der Steuerelemente verläuft [Weg13].

Auch ist es ohne weiteres möglich, das Aussehen von eigenen Klassen, die im speziellen keine visuellen Objekte sein müssen, zu definieren. Dazu gibt es DataTemplates. Erstellt man beispielsweise eine Liste von Objekten, für die ein Aussehen definiert wurde, werden diese entsprechend angezeigt. Ansonsten erscheint standardmäßig der vollständige Name der Objektklassen als Ausgabe.

2.1.1 Die Arten der Bindungen

Zu den interessantesten Techniken von WPF gehören die verschiedenen Möglichkeiten der Zuweisung von Daten [Küh12].

2.1.1.1 Datenbindung

Bei einem sogenannten DataBinding wird auf eine Eigenschaft des aktuellen DataContext verwiesen (Listing 2.1). Der DataContext kann ein beliebiges Objekt einer Klasse sein, welches öffentliche Attribute bereitstellt.

```
<TextBox Text="{Binding Name}"/>
```

Listing 2.1: Beispiel: DataBinding

Zusätzlich gibt es noch weitere Einstellmöglichkeit, um beispielsweise den Zeitpunkt des Übertragens an den DataContext zu bestimmen.

2.1.1.2 Statisches und dynamisches Binden von Ressourcen

Um auf eine durch XAML-Code definierte Ressource zugreifen zu können, gibt es zwei Möglichkeiten: statisches und dynamisches Binden.

Beim statischen Binden mittels StaticResources wird der Zugriff auf eine Ressource bereits zur Compile-Zeit aufgelöst. Dazu werden die Elternobjekte nach der Ressource durchsucht. Wird die entsprechende Ressource nicht gefunden, bricht das Kompilieren mit einem Fehler ab [vgl. Weg13, S. 146].

Beim dynamischen Binden mittels DynamicResources wird die Ressource erst zur Laufzeit gesucht und gesetzt. Dies sollte ausschließlich dann verwendet werden, wenn erst zur Laufzeit die Ressource bekannt ist oder sie sich währenddessen verändert [vgl. Weg13, S. 146f].

Damit Texte ausgetauscht werden können, müssen diese in WPF an ein String-Objekt innerhalb eines sogenannten ResourceDictionary gebunden werden. Diese werden in XAML geschrieben. Die String-Objekte besitzen einen Schlüssel und den Textinhalt. Die WPF-Textelemente, wie Label oder TextBox, verwenden diese Schlüssel zum Laden der Textinhalte, indem sie statische oder dynamische Bindung verwenden.

2.1.2 DataTemplates

Mithilfe eines DataTemplates erstellt man eine Vorlage für einen bestimmten Datentyp [vgl. The12, S. 237]. Ein Objekt dieses Datentyps kann in einem beliebigen WPF-Element, zum Beispiel einer ListBox oder einer ContentControl, verwendet werden.

```
<DataTemplate DataType={x:Type viewModels:UserLoginViewModel}">
 <views:UserLoginView />
</DataTemplate>
```

Listing 2.2: Beispiel: DataTemplate

Im Listing 2.2 wird die Verwendung von DataTemplates demonstriert. Es wird eine Klasse als Typ eingetragen und als Darstellung wird ein UserControl übergeben. Es ist möglich, das Aussehen direkt innerhalb eines DataTemplates zu definieren.

2.1.3 Konverter

Die Datenbindung in WPF verfügt über mehrere Eigenschaften, wie dem Festlegen von String-Formaten. Es kann zudem ein Konverter-Objekt übergeben werden, welches die gebundene Eigenschaft vor dem eigentlichen Anzeigen verändert. Umgekehrt findet bei einer Veränderung der Eigenschaft durch den Benutzer eine Rückkonvertierung statt.

Für einen Konverter wird das Interface IValueConverter aus Listing 2.3 implementiert. Dieses legt zwei Methoden fest:

Convert wird auf dem Weg vom gebundenen Objekt zur View verwendet. Der Anwender sieht das Ergebnis der Konvertierung.

ConvertBack wird aufgerufen, sobald die View das gebundene Objekt verändert.

```
public interface IValueConverter
{
  object Convert(object value, Type targetType, object parameter, CultureInfo culture);
  object ConvertBack(object value, Type targetType, object parameter, CultureInfo
      culture);
}
```

Listing 2.3: Das Interface IValueConverter

Beide Methoden besitzen die selben Parameter:

value ist der Wert der gebundenen Eigenschaft. Er wird als Object-Typ übergeben, die Basisklasse in C#.

targetType ist der Typ der gebundenen Eigenschaft.

parameter enthält weitere durch die Bindung festgelegte Parameter und ist optional.

culture sollte eigentlich die aktuellen Kulturinformationen enthalten. Dies ist leider nicht der Fall, sondern es wird immer die englisch-amerikanischen Informationen übergeben. Dabei handelt es sich um einen Bug in WPF.

2.2 Entwurfsmuster

Ein Entwurfsmuster beschreibt ein Problem, welches immer wieder auftaucht und schildert dann den Kern der Lösung in einer Art und Weise, dass man diese Lösung immer wieder anwenden kann, ohne diese zwei mal auf die gleiche Weise zu implementieren [Gam+95].

2.2.1 Model-View-ViewModel

Das Model-View-ViewModel-Muster, kurz MVVM, ist ein Architekturmuster für Benutzeroberflächen. Es wurde von Mircosoft für WPF und Silverlight entwickelt, findet aber auch bei JavaFX [Mau15] und Javascript [1] Anwendung. In diesem Fall wird nur auf die Implementierung in WPF und C# eingegangen.

2.2.1.1 Zweck

Ziel ist die Entkopplung der Daten von ihrer Darstellung, indem man eine Zwischenschicht einführt, die bei Datenänderungen die Darstellung und das Datenobjekt aktualisiert und zudem die Darstellungslogik enthält. So bleibt das Datenobjekt frei von Darstellungseigenschaften und die Darstellung kann leicht ausgetauscht werden.

2.2.1.2 Struktur

Die Abbildung 2.1 zeigt das Muster in seiner einfachsten Form. Die View und das ViewModel sind über Data- und Command-Binding miteinander verknüpft. Das ViewModel verwendet das Model und verändert es. Das Model löst gegebenenfalls Ereignisse aus, die das ViewModel abonniert hat, um Änderungen an seinen Daten zu signalisieren.

[1] z.B. Knockout.js (`http://knockoutjs.com`)

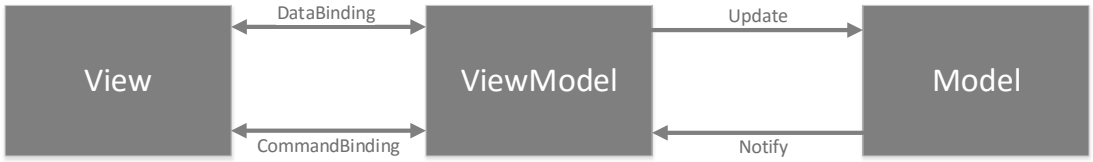

Abbildung 2.1: Das Model-View-ViewModel-Muster

2.2.1.3 Teilnehmer

In diesem Entwurfsmuster sind grundsätzlich drei Objekte involviert:

Model ist ein Objekt einer Datenklasse. Sie beinhaltet die Business-Logik und kann ebenfalls die Validierung enthalten.

View definiert die Darstellung. Sie wird meist in einer XML-ähnlichen Sprache beschrieben.

ViewModel ist das Bindeglied zwischen dem Model und der View. Es besitzt keinerlei Kenntnisse über die View, implementiert jedoch deren Logik. Durch die Implementierung des Beobachtermusters gibt es Änderungen am Model an die View weiter. Umkehrt erhält das ViewModel Eingaben durch die View. In WPF wird dazu die Datenbindung verwendet, ansonsten würde dies durch eine weitere Implementierung des Beobachtermusters geschehen.

2.2.1.4 Konsequenzen

Möchte man das Model-View-ViewModel-Muster konsequent einhalten, kommt es zu einigen Einschränkungen:

- Die Code-Behind-Datei der View sollte nicht bearbeitet werden. Verwendet man diese, erhöht sich gegebenenfalls die Kopplung zum ViewModel. Wenn die View Logik enthält, lässt sich diese nicht mehr einfach austauschen.
- Das ViewModel darf die visuellen Objekte der Darstellung nicht kennen. Sobald dieses von den visuellen Objekten weiß, ist die Darstellung nicht mehr vollständig entkoppelt, da eine Abhängigkeit zwischen View und ViewModel entsteht.

2.2.1.5 Implementierung

Das Model ist als Datenklasse implementiert. Die Klasse kann das INotifyPropertyChanged-Interface implementieren, wenn Models untereinander kommunizieren müssen und Änderung nicht nur über das entsprechende ViewModel geschieht. Eine weitere Möglichkeit ist es, das ViewModel als eine Fassade zu betrachten, welche den Zugriff auf meine Menge von Models handhabt. So muss kein spezifisches Model implementiert werden.
Das ViewModel oder seine Basisklasse müssen für das MVVM das INotifyPropertyChanged-Interface implementieren (Listing 2.4), da die View das Ereignis PropertyChanged abonniert. Nur so kann die View ihre Darstellung aktualisieren und auf veränderte Daten reagieren. Das ViewModel besitzt eine Referenz auf das ensprechende Model.

```
public class ViewModelBase : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  public void RaisePropertyChanged(string propertyName)
  {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

Listing 2.4: Eine typische Implementierung des ViewModels

Die View wird in der Beschreibungssprache XAML erstellt und spiegelt den Zustand und die Inhalte des ViewModels wider [Weg13]. Die in der View definierten Steuerelemente können durch Datenbindung (DataBinding) an Eigenschaften des ViewModels gebunden werden. Damit die Datenbindung funktioniert, muss zuvor die View das ViewModel als Kontext übernehmen (DataContext). Dafür gibt es zwei Varianten. Entweder definiert man den Kontext per DataTemplate, wie in Listing 2.2 oder man trägt den DataContext direkt im XAML oder in der Code-Behind-Datei ein.

```
<UserControl ...>
 <UserControl.DataContext>
  <viewModel:UserLoginViewModel />
 </UserControl.DataContext>
 ...
</UserControl>
```

Listing 2.5: Beispiel: DataContext in XAML

2.2.1.6 Beispiel

Um die Funktionsweise des Model-View-ViewModel zu zeigen, folgt nun ein einfaches Beispiel. Das Beispielprogramm soll eine eingegebene Zahl in das Model übertragen.
Das Datenmodel (Listing 2.6) enthält eine Integer-Eigenschaft mit ihren Get- und Set-Methoden.

```
public class NumberModel
{
  public int Number { get; set; }

  public NumberModel(int Number)
  {
    this.Number = Number;
  }
}
```

Listing 2.6: Beispiel: Model

Das ViewModel (Listing 2.7) spiegelt das Datenmodel gegenüber der Darstellung wieder und besitzt daher ebenfalls eine Integer-Eigenschaft. Dieses gibt in der Get-Methode jedoch den Wert aus dem Model zurück. In der Set-Methode ändert das ViewModel den Wert des Models und löst mithilfe der Methode RaisePropertyChanged das Ereignis PropertyChanged aus. Als Übergabeparameter wird der Name der Eigenschaft als String übergeben.
In der Methode RaisePropertyChanged wird mithilfe des „?“-Operators zunächst geprüft,

ob das Ereignis überhaupt abonniert wurde. Wenn ja, wird das Ereignis ausgelöst. Als Übergabeparameter wird immer die aktuelle Instanz des ViewModels und der Eigenschaftsname übergeben.

```
public class NumberViewModel : INotifyPropertyChanged
{
  private NumberModel model;

  public int Number
  {
    get { return model.Number; }
    set
    {
      model.Number = value;
           RaisePropertyChanged("Number");
    }
  }

  public NumberViewModel()
  {
    this.model = new NumberModel(0);
  }

  public event PropertyChangedEventHandler PropertyChanged;

  public void RaisePropertyChanged(string propertyName)
  {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

Listing 2.7: Beispiel: ViewModel

Die Darstellung (Listing 2.8) verwendet das NumberViewModel als DataContext. Es besitzt zum Anzeigen und Verändern einer Zahl eine TextBox. Dazu wird die bereits erwähnte Datenbindung verwendet.

```
<UserControl x:Class="MVVM_Example.Views.NumberView"
 xmlns:viewModel="clr-namespace:MVVM_Example.ViewModels"
 ...>
 <UserControl.DataContext>
  <viewModel:NumberViewModel />
 </UserControl.DataContext>
 <Grid Margin="10">
  ...
  <TextBox Text="{Binding Number}" HorizontalContentAlignment="Center"
      VerticalContentAlignment="Center"/>
  ...
 </Grid>
</UserControl>
```

Listing 2.8: View

2.3 Visualization Toolkit und IZFPVisualizationLib

Das Visualization Toolkit, kurz VTK, ist eine Open-Source-C++-Bibliothek für 3D-Computergrafik. Ihr Fokus liegt im wissenschaftlichen Bereich [Vtk].

Die Klassenbibliothek IZFPVisualizationLib wird von der Abteilung Zustandsüberwachung und Lebensdauermanagement des Fraunhofer IZFP ständig weiterentwickelt. Sie ermöglicht die Visualisierung von Messdaten. Die Palette an Möglichkeiten reicht von der Darstellung von Graphen, über 2D-Bilder, bis hin zu komplexen 3D-Bildern.

Die IZFPVisualizationLib untergliedert sich in drei Schichten:

VTKBasic greift direkt auf die durch die VTK bereitgestellten Klassen zu. In diesen Klassen wird die eigentliche Funktionalität implementiert, beispielsweise das Zoomen in ein Bild und die Verwendung von einem oder mehreren Cursor innerhalb eines Bildes.

VTKWrap stellt einen Adapter zwischen der nativen VTKBasic und dem Visualization-Provider dar.

VisualizationProvider macht aus den VTK-Objekten WinForms-Elemente, die in einer Windows-Anwendung direkt verwendet werden können. Sie teilt außerdem die Daten von der Ansicht.

Im weiterem wird nur auf die Klassenbibliothek VisualizaionProvider eingegangen, da sie die für die Anwendung benötigten visuellen Objekte enthält.

2.3.1 VisualizaionProvider

Die Bibliothek VisualizaionProvider stellt die verschiedenen VTK-Viewer als Windows Forms-Elemente bereit. Diese erhalten über ihren entsprechenden DataProvider die Daten, welche sie anzeigen sollen. Im folgenden werden kurz die verwendeten VTK-Elemente erklärt, deren genaue Verwendung in der Dokumention [Izf] festgehalten wird.

2.3.1.1 Viewer2D

Wie der Klassenname schon vermuten lässt, dient der Viewer2D zur Darstellung von 2D-Daten. Er kann mit zwei Objekten der Klasse Viewer1D gekoppelt werden, welche anhand der Position des Cursors aktualisiert werden. Durch eine Toolbar können, unabhängig vom hostenden visuellen Element, Einstellungen getätigt werden. Dazu zählt die Hintergrundfarbe, aber auch die Art der Darstellung der Region of Interest (ROI), also eines markierten Bereiches. Auch kann eine Interpolation der Ansicht aktiviert werden, damit die Darstellung natürlicher wirkt.
Im Viewer kann man zoomen und einen Bereich markieren. Dies hat Auswirkung auf die gekoppelten 1D-Viewer. Sie zoomen bzw. markieren ebenfalls in diesen Bereich.

Sowohl 1D- als auch 2D-Viewer besitzen an den Rändern Lineale. Deren Einheit wird durch die Viewer gesetzt und mittels des DataProviders wird der Maßstab festgelegt.

2.3.1.2 Viewer1D

Die Klasse Viewer1D dient zur Visualisierung von eindimensionalen Daten als Liniendiagramm. Sie bietet mit der Toolbar ähnliche Funktionalitäten wie die Klasse Viewer2D.

2.3.1.3 VTKColorTableEditor

Um die Farbgebung der angezeigten Daten eines Viewer2D anzupassen, kann der Viewer einem ColorTableEditor-Objekt übergeben werden. Wird diese angezeigt, kann man aus

einer Liste von Farbpaletten wählen. Darunter fallen lineare, semilogarithmische und logarithmische Darstellungen. Zudem kann man über eine Utility-Klasse eigene Farbpaletten definieren und diese in einen ColorTableEditor injizieren.

2.3.1.4 DataProvider2D und DataProvider1D

Die DataProvider-Klassen dienen der Entkopplung von Anzeige und Datenhaltung. Den Viewer-Klassen wird bei der Konstruktion eine Instanz der passenden DataProvider-Klasse übergeben. Mithilfe dieser Instanz kann dann völlig unabhängig von der verwendeten Viewer-Klasse Daten gesetzt werden. Die DataProvider-Klassen unterscheiden sich grundsätzlich nur von der Anzahl an definierten Achsen. Ein DataProvider1D besitzt daher lediglich eine Anzahl an Werten, ein DataProvider2D besitzt zwei Achsen, uws. Durch das Setzen der sogenannten StepSize, also der Schrittweite, kann der Maßstab der Daten gesetzt werden.

3 Konzeption

Vor der Implementierung wird aufgrund der gesetzten Ziele die Anforderungsspezifikation formuliert. Aufgrund dieser wird ein Entwurf erstellt.

3.1 Anforderungsspezifikation

Die Forderungen aus Abschnitt 1.4 werden nun ausformuliert. Für eine moderne Benutzeroberfläche werden das Frameworks Windows Presentation Foundation und die Programmiersprache C# [Mic] verwendet. Es werden durch das Institut entwickelte WPF-Bibliotheken verwendet, die zum Einem das Aussehen der meisten visuellen Objekte abdecken und zum anderen das Implementieren des Model-View-ViewModel-Entwurfsmusters vereinfachen. Zur Visualisierung der Messdaten wird die institutseigene Weiterentwicklung des Visualization Toolkits verwendet. Die Prüfparametersätze werden im XML-Format abgespeichert, um sie einfacher austauschen und zwischen den Computern des Kunden verteilen zu können. Die Anforderungen vom Kunden werden folgendermaßen verwirklicht:

- Nach einer Messung werden die Messdaten automatisch im JPG- und XML-Format gespeichert. Um die Messdaten eindeutig zuordnen zu können, werden die Dateinamen die Auftrags- und Identifikationsnummer des geprüften Lichtmasten enthalten. Zudem wird dem Prüfer die Möglichkeit eingeräumt, Daten zum Lichtmasten, wie Wandstärke und Türhöhe, abzuspeichern.
- Die gespeicherten Messungen können mit dieser Software wieder geöffnet werden. Dabei werden alle Parameter wiederhergestellt, damit die Messung besser nachvollziehbar ist. Der Name des Prüfers wird ebenfalls abgespeichert.
- Mit der neuen Software können Messungen, die durch die Alt-Software erstellt wurden, nicht geöffnet werden.
- Durch ein Rechtesystem werden die Parameter festgelegt, die der Prüfer verändern darf.
- Die Software wird eine einfache Benutzerverwaltung enthalten. Ein Benutzer kann eine der folgenden drei Rollen haben:

  Prüfer bezeichnet den normalen Anwender. Dieser kann freigegebene Parameter verändern und Messungen durchführen.

  Administrator kann, wie der Prüfer, Messungen durchführen. Er kann alle Parameter verändern und für den Prüfer mittels des Rechtesystems freigeben. Er kann zudem die Benutzerverwaltung verwenden, um Prüfer anzulegen und wieder zu löschen. Der Administrator wird in der Software integriert und der Zugriff auf ihn wird durch ein Passwort gesichert.

  Service spiegelt den Zugriff für das Personal des Fraunhofer IZFP wider. Er ist ebenfalls durch ein Passwort geschützt und in der Software integriert. Zu seiner Funktion mehr in Anforderungen durch das Institut.

- Da die Software hauptsächlich im Außeneinsatz verwendet werden soll und somit auf Notebooks installiert wird, ist die Software als Vollbildanwendung ausgelegt.

Die durch das Institut gestellte Anforderungen werden folgendermaßen implementiert:

- Die Sprachumschaltung wird während der Programmlaufzeit möglich sein. Zudem können Sprachdateien ausgetauscht werden, ohne die Software entsprechend konfigurieren zu müssen, da die Software diese beim Programmstart erkennt und einbindet.
- Der Kalibrierungsmodus wird integriert und kann über das Service-Benutzerprofil verwendet werden.
- Ein als Service angemeldeter Benutzer wird Prüfparametersätze erstellen können, welche nur ihm zur Verfügung stehen, da sie der Instandhaltung dienen. Dazu wird in der XML-Datei des Parametersatzes ein bestimmter Eintrag gesetzt.

3.2 Ausgangspunkt

Vor Beginn dieser Arbeit wurden bereits verschiedene Schritte eingeleitet, um die Software rechtzeitig ausliefern zu können. Zum Einen wurde von Jan Oswald eine Klassenbibliothek zur Ansteuerung der derzeitig im Einsatz befindlichen Prüfhardware-Revisionen geschaffen. Sie wurde in nativen C++ entwickelt und hat verschiedene Aufgaben. Sie bildet den internen Zustand der Prüfhardware ab, sorgt für das Parametrieren und den Empfang der durch die Hardware generierten Messdaten. Die Kommunikation zwischen der Bibliothek und der Hardware findet über TCP/IP statt. Nach außen bietet die Klassenbibliothek eine einfache Schnittstelle an, um Messungen durchzuführen.

Außerdem wurde bereits vor Beginn dieser Arbeit damit begonnen, verschiedene Komponenten der neuen Software zu erstellen, da eine vollständige Entwicklung einer im täglichen Gebrauch befindlichen Software den Rahmen einer Bachelor-Thesis überschreiten würde. Dazu gehören unter anderem verschiedene visuelle Objekte, beispielsweise Schaltflächen und Texteingabefelder, die im Corporate Design gehalten sind. Diese sind in Form einer Bibliothek implementiert. Zudem wurden für die Umsetzung des Model-View-ViewModel-Musters benötigte Basisklassen verwirklicht. Dieses Entwurfsmuster wird in Abschnitt 2.2.1 näher betrachtet.

3.3 Vorgehensweise

Da mit der Entwicklung bereits vor dem eigentlichen Zeitraum der Bachelorarbeit begonnen wurde, wurde sich für eine iterative Vorgehensweise entschieden. Der erste Entwurf liefert zwar in relativ kurzer Zeit erste Ergebnisse, jedoch fehlen die meisten geplanten Funktionalitäten. Diese sollen im Laufe der Entwicklung implementiert werden. Ständige Refaktorisierung sorgt dafür, dass die Struktur erhalten bleibt und Zuständigkeiten auf neue Klassen verteilt werden.

3.4 Erster Entwurf

Für den Entwurf wurden nächst einige grundsätzliche Überlegungen gemacht. Die Vollbildanwendung benötigt einen Rahmen, in dem sie läuft. Dieser sollte einen Kopf- und

eine Statuszeile besitzen. In der Kopfzeile wird das Logo des Fraunhofer IZFP, der Name der Software und die aktuelle Uhrzeit bzw. das Datum dargestellt. Dazu sollte sie eine Möglichkeit bieten, die Software zu beenden. Da die Kopfzeile keinerlei Geschäftslogik enthält, wird keine Model-Klasse implementiert. Im späteren Verlauf wurde die Sprachumschaltung in dieser Klasse integriert.

In der Statuszeile sollen Informationen zum aktuellen Geschehen innerhalb der Software gezeigt werden. Auch die Statuszeile wird ohne eine Model-Klasse implementiert.

Die Hauptansicht, welche den Rahmen bildet, implementiert ihre Geschäftslogik durch das Aufteilen der Zuständigkeiten auf die verschiedenen Unteransichten. Innerhalb des Rahmens kann nun zwischen Ansichten gewechselt werden: Dem Anmeldefenster (UserLoginView) und den zwei Prüfansichten (LimaTestControlView und LimaTestServiceView) für das Durchführen von Messungen und der des Kalibrierungsmodus. Die Prüfansichten teilen sich eine Klasse zum Ansteuern der Hardware.

Dabei entstand folgender Aufbau, der in Abbildung 3.1 dargestellt wird. In der Abbildung sind alle geplanten Ansichten und auch der Eintrittspunkt jeder WPF-Anwendung, die Klasse App, zu sehen. Im weiteren Verlauf werden diese aufgrund der Übersicht nicht mehr gezeigt.

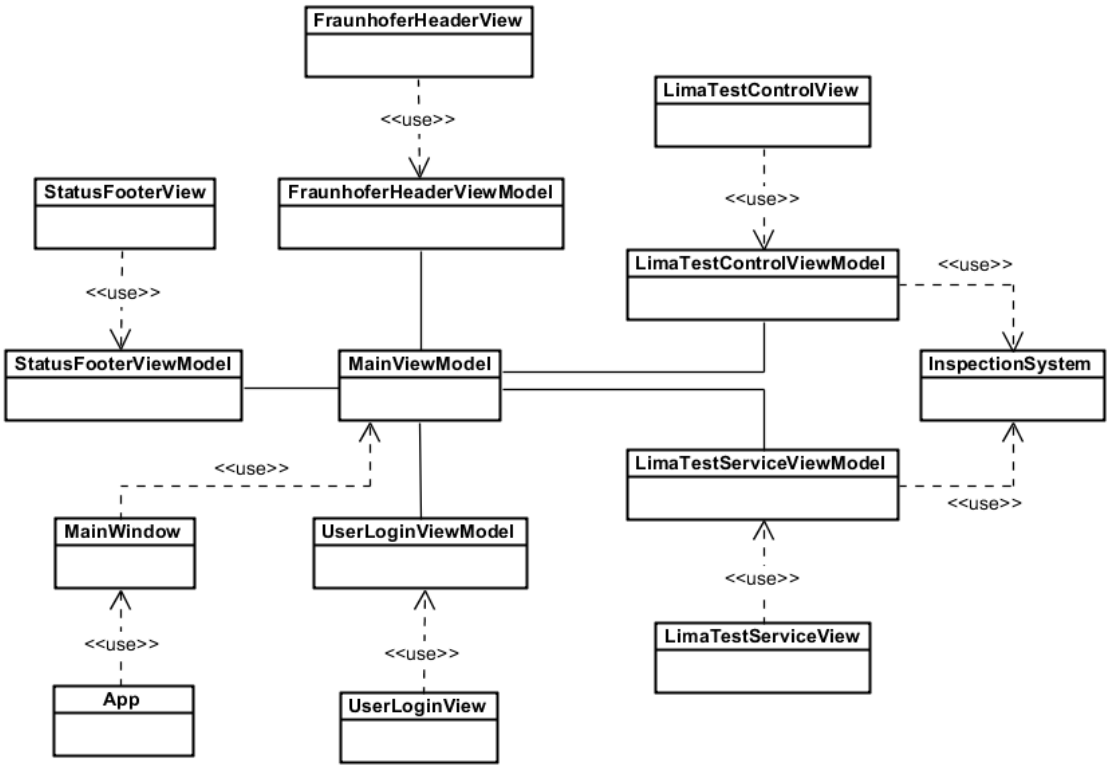

Abbildung 3.1: Erster Entwurf

Die Klasse App instanziiert ein Objekt von MainWindow. Dieses verwendet ein MainViewModel als DataContext und bindet jeweils ein Objekt von FraunhoferHeaderViewModel, StatusFooterViewModel und ein beliebiges Objekt, welches die Oberklasse ViewModelBase implementiert. Dies können alle implementierten ViewModels sein, es zielt aber auf das UserLogin-, LimaTestControl- und das LimaTestServiceViewModel ab.

4 Implementierung

Die Durchführung des Projektes war sehr umfangreich. Daher wird im folgendem auf einige der wichtigsten Punkte eingegangen.

4.1 Software-Design

In der Abbildung A.1 des Anhangs A wird das Klassendiagramm der gesamten Software gezeigt. Dabei wurde auf die Views und Hilfsklassen aufgrund der Übersicht verzichtet. Zu jedem ViewModel gibt es eine entsprechende View. Eine Ausnahme stellt das LimaTestMainViewModel als abstrakte Basisklasse der Prüfansichten, LimaTestControl- und LimaTestServiceViewModel, dar. Im folgendem werden die in der Kommunikation involvierten Klassen beschrieben, ebenso jene, die der Visualisierung der Messdaten dienen, die von der Hardware empfangen werden. Danach folgt eine Beschreibung der Views und ViewModels.

4.1.1 Hardwaresteuerung

In Abbildung 4.1 wird die Klassenhierarchie der Hardwaresteuerung dargestellt.

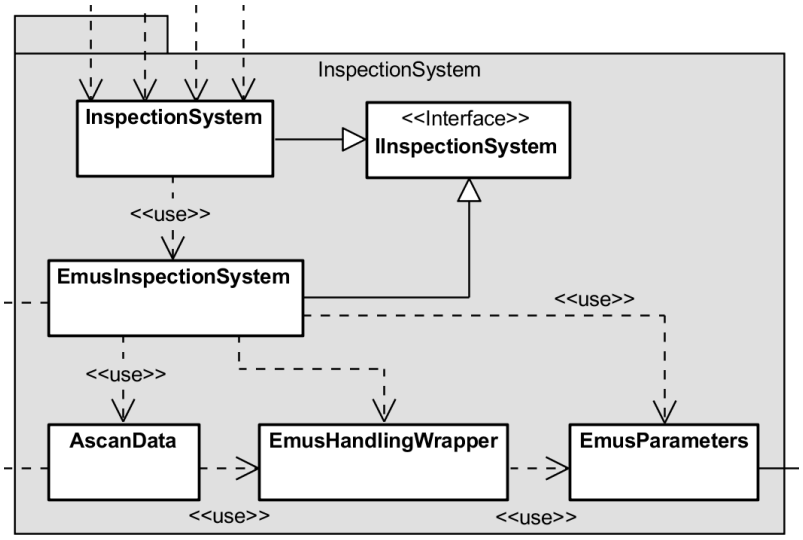

Abbildung 4.1: Klassenhierarchie der Hardwaresteuerung

4.1.1.1 IInspectionSystem

Das IInspectionSystem ist ein Interface, welches Vorgaben für die verschiedenen Funktionalitäten macht, die zur Steuerung der Hardware benötigt werden. Es stellt die Eigenschaft IsConnected bereit, die anzeigt, ob eine Verbindung zur Hardware besteht. Außerdem enthält es ein Ereignis, mit welchem nach einer Messung abschließende Methoden ausgeführt werden können. Dies betrifft zum Beispiel das Abspeichern der Messung in

eine Datei.
Das IInspectionSystem erbt von fünf Interfaces:

- IAdministrateConnection (Verbindungsaufbau und -prüfung)
- ISupportDataAcquirement (Start und Stopp einer Messung)
- ISupportCalibration (Start und Stopp einer Kalibrierung)
- IHandleEmusParameters (Setzen der Parameter für elektromagnetischen Ultraschall)
- ISupportOnlineParametrization (Setzen von Parametern während einer Kalibierung)

4.1.1.2 InspectionSystem

Die Klasse InspectionSystem ist eine Implementierung des Interfaces IInspectionSystem und abstrahiert den Zugriff zur Hardware. Sie und die Klasse EmusInspectionSystemImpl bilden zusammen das Brücken-Muster, wobei EmusInspectionSystemImpl den Implementor darstellt. Die genaue Vererbungshierarchie entnehmen Sie im Anhang A der Abbildung A.2.

Durch Dependency Injection wird der Implementor durch das MainViewModel gesetzt. In Zukunft soll die zu verwendende Implementierung anhand eines Eintrags in der Konfigurationsdatei ermittelt werden. Daher bot sich das Brücken-Muster an, da es einen Austausch ermöglicht. Innerhalb der Klasse wird der Zugriff auf den Implementor durch eine Get-Methode geprüft und löst gegebenenfalls eine Ausnahme aus, falls die Dependency Injection nicht durchgeführt wurde.

Aufgrund der Tatsache, dass über diese Klasse der Zugriff auf die Hardware gesteuert wird, wurde sie als Singleton implementiert. Dies sorgt für den Umstand, dass diese Klasse in insgesamt vier weiteren Klassen verwendet wird, wie es im UML-Diagramm Abbildung A.1 zu sehen ist.

4.1.1.3 EmusInspectionSystemImpl

Die Klasse EmusInspectionSystemImpl ist der Implementor von InspectionSystem und sorgt für die Kommunikation mit dem EmusHandlingWrapper. Die Klasse verknüpft die Hardwaresteuerung mit der Visualisierung, indem es das Ereignis AscanDataEvent mit der für den Messmodus entsprechende Methode abonniert (Listing 4.1). Beim Ausführen der Methoden StartDataAcquirement und StartCalib werden jeweils die passenden Schnittstellen zur Visualisierung der Messdaten übergeben. Beim Auslösen des Ereignisses werden diese Ziele verwendet. Die Variable receiverCount dient dem Zählen der empfangen Messdaten und vergleicht diesen mit der angeforderten Zahl von Ascans.

```
public bool StartDataAcquirement(IReceiveAScan destination, bool useMagnetization)
{
  this.receiver = destination;
  receiveCounter = 0;
  destination.Clear();
  handling.AScanDataEvent += handling_AScanDataEvent;
  if (this.handling.startDataAcquirement(useMagnet))
  {
    this.receiver.Start();
```

```
    return true;
  }
  return false;
}

public bool StartCalib(IReceiveMultipleAScans destination, bool useMagnetization)
{
  this.multipleAscanDataReceiver = destination;
  handling.AScanDataEvent += handling_MultipleAScansDataEvent;
  if (this.handling.startCalibration(useMagnet))
  {
    return true;
  }
  return false;
}
```

Listing 4.1: StartDataAcquirement und StartCalib

4.1.1.4 EmusHandlingWrapper

Die Klasse EmusHandlingWrapper dient dem Zugriff auf die Hardware-Klassenbibliothek und ist in C++/CLI programmiert. Die Klasse besitzt zum einem eine statische Referenz auf die native Hardware-Klasse EmusHandling als auch auf die zuletzt erstellte Instanz von sich selbst. Die native Hardware-Klasse erwartet für eine Messung einen Funktionszeiger. Dieser soll auf eine Funktion referenzieren, welche die empfangenen Messdaten in die nächst höhere Schicht kopiert. Dazu besitzt EmusHandlingWrapper je eine statische Methode für jeden Messungsmodus, also für die übliche Datenaufnahme und für die Kalibrierung. Das Umwandeln der Methoden in Funktionszeiger wird in Abschnitt 4.2 erklärt.
Die Klasse bietet außerdem eine Methode zum Setzen der Parameter von EmusHandling. Dazu wird der Methode ein Objekt der Struktur EmusParameters übergeben.

4.1.1.5 EmusParameters

EmusParameters ist eine in C++/CLI programmierte Struktur und enthält die von der Hardware benötigten Parameter als Eigenschaften.

4.1.1.6 AscanData

AscanData ist eine C++/CLI-Klasse, die zum Transport von Messdaten dient. Um die Messdaten auswerten zu können, werden drei Werte benötigt:

- ein Integer-Array, welches die eigentliche Messung enthält
- den Maximalwert der bei einer Messung vorkommen kann
- die Länge des Integer-Array

4.1.2 Visualisierung der Messdaten

Aufgrund der Tatsache, dass die Messdaten asynchron von der Hardware kommen, musste ein entsprechendes Threading-Modell (Abschnitt 4.1.2.5) entwickelt werden. Jedoch folgt zunächst eine Beschreibung der für die Visualisierung zuständigen Klassen (Abbildung 4.2).

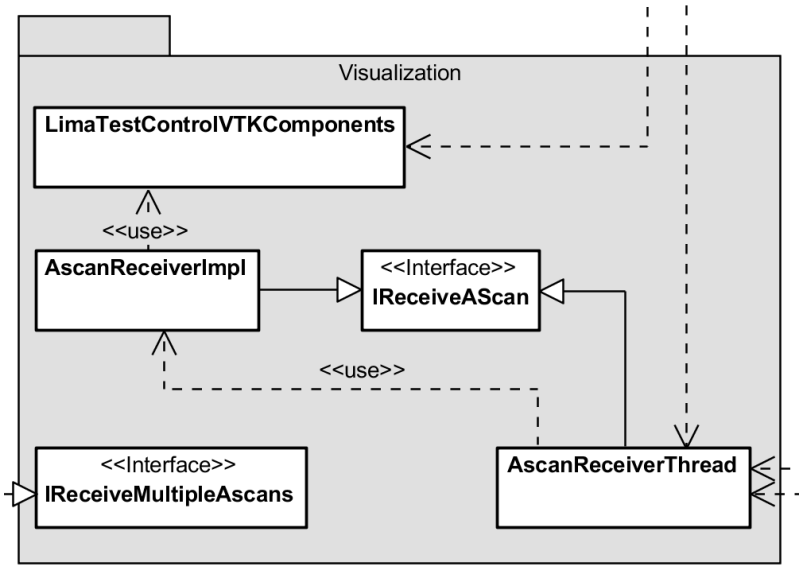

Abbildung 4.2: Klassenhierarchie der Hardwaresteuerung

4.1.2.1 IReceiveAscan und IReceiveMultipleAscans

Die Interfaces IReceiveAscan und IReceiveMultipleAscans definieren die Schnittstelle zur weiteren Verarbeitung der empfangenen Messdaten, nachdem sie von EmusInspectionSystemImpl aus Abschnitt 4.1.1.3 empfangen wurden.

IReceiveMultipleAscans legt eine einzelne Methode fest, die zum Empfang dreier Objekte vom Typ AscanData (Abschnitt 4.1.1.6) dient. Diese Schnittstelle wird lediglich für den Kalibrierungsmodus verwendet werden. Es werden keine weitere Methoden vorgegeben, da in diesem Modus die Messdaten nach der Visualisierung verworfen werden. Die Klasse LimaTestServiceViewModel implementiert dieses Interface.

Da bei der üblichen Datenaufnahme nur ein Objekt der Klasse AscanData übertragen wird, definiert IReceiveAscan eine entsprechende Methode zur Weiterverarbeitung. Jedoch wird in diesem Modus davon ausgegangen, das die Messdaten gespeichert werden müssen und nicht direkt verworfen werden. Daher bietet diese Schnittstelle Methoden zum Berechnen der benötigten Menge an Messdaten und zum Leeren des Zwischenspeichers. Zudem sollen Start- und Stopp-Methoden eine stark vereinfachte Thread-Schnittstelle darstellen. Damit nach dem Stoppen einer Messung und deren vollständiger Visualisierung eine beliebige Methode aufgerufen werden kann, um weitergehende Schritte einzuleiten, wird über dieses Interface eine geeignete Aktion übergeben.

4.1.2.2 AscanReceiverThread

Die Klasse AscanReceiverThread implementiert IReceiveAscan und dient als Puffer zwischen dem Empfang und der Visualisierung der Messdaten. Die Klasse ist als Brücken-Muster ausgelegt, deren eigentliche Implementierung in der Klasse AscanReceiverImpl steckt.
Von außen, über die Klasse EmusInspectionSystemImpl, wird es mit Messdaten gefüllt, die in eine Thread-sichere Warteschlange (ConcurrentQueue) eingereiht werden. Die Warteschlange arbeitet nach dem First-In-First-Out-Prinzip. Durch einen durch die Klasse erstellten Thread wird die Warteschlange abgearbeitet. Von dort gelangen sie in die Im-

plementierung, wo sie weiterverarbeitet werden.

Führt man die Start-Methode in Listing 4.2 aus, wird ein neuer Thread gestart, der die run-Methode (Listing 4.3) ausführt.

```
public void Start()
{
  if (thread == null || !thread.IsAlive)
  {
    thread = new Thread(new ThreadStart(delegate()
    {
      this.run();
    }));
    thread.Start();
  }
}
```

Listing 4.2: Methode Start

Die run-Methode (Listing 4.3) läuft, solange der Thread von außen nicht beendet wurde oder in der Warteschlange noch Objekte abgelegt sind. Dies sorgt dafür, dass, selbst wenn die Messdaten schneller ankommen als sie zur Visualisierung verarbeitet werden können, die Software nicht blockiert wird und somit nicht durch eine falsche Reaktion des Anwenders abstürzt.

```
private void run()
{
  IsRunning = true;
  while (IsRunning || !queue.IsEmpty)
  {
    AScanData scan;
    queue.TryDequeue(out scan);
    if (scan != null)
      implementor.AddScanData(scan);
  }
}
```

Listing 4.3: Methode run

Die Methode zum Stoppen des Thread (Listing 4.3) sorgt dafür, dass nach dem Abarbeiten der Warteschlange der Thread beendet werden kann.

```
public void Stop()
{
  this.IsRunning = false;
}
```

Listing 4.4: Methode Stop

Damit die Eigenschaft IsRunning Thread-Sicherheit bietet, besitzt sie Get- und Set-Methoden, die mittels Lock-Befehl abgesichert sind (Listing 4.5).

```
private readonly object locker = new object();
private book isRunning;
```

```
private bool IsRunning
{
  get
  {
    bool value;
    lock (locker)
    {
      value = isRunning;
    }
    return value;
  }
  set
  {
    lock (locker)
    {
      isRunning = value;
    }
  }
}
```

Listing 4.5: Threadsichere Eigenschaft IsRunning

4.1.2.3 AscanReceiverImpl

Die Klasse AscanReceiverImpl ist der Implementor von AscanReceiverThread und implementiert ebenfalls das IReceiveAscan-Interface. Es speichert die Menge an Messdaten in einem Objekt der Klasse AscanMatrix. Es besitzt zudem eine Referenz zur Klasse LimaTestControlVTKComponents über die Schnittstelle IVTKDataProvider2DHolder. Darüber füllt AscanReceiverImpl den DataProvider des VTK-2D-Viewer mit den Daten aus der AscanMatrix.

Die Methode AddScanData (Listing 4.6) wird durch AscanReceiverThread (Listing 4.3) aufgerufen, wenn ein AscanData-Objekt aus der Warteschlange genommen wurde. Beim ersten Aufruf der Methode wird zunächst eine neue Instanz der AscanMatrix generiert. Dies ist abhängig von der Ausrichtung, die der VTK-2D-Viewer haben soll. Als Dimensionen der Matrix werden die Anzahl an erwarteten Ascans und die Länge der erhaltenen Ascans gewählt. Die Länge ändert sich während einer Messung nicht. Danach wird das Flag isFirstScan zurückgesetzt und es folgt das Speichern der Messdaten in die Matrix und das Setzen der Matrix in den DataProvider.
Ein direktes Setzen eines einzelnen Datensatzes ist in der IZFPVisualizationLib nicht möglich. Es kann nur ein vollständiges Array von Integern entgegennehmen.
Am Ende der Methode AddScanData wird geprüft, ob die Matrix vollständig befüllt ist. Wenn dies zutrifft, wird die abschließende Aktion, welche zuvor definiert wurde, ausgeführt.

```
public void AddScanData(AScanData ascan)
{
  if (ascan.Size != 0)
  {
    if (isFirstAScan)
    {
      this.matrix = AScanMatrixFactory.GetSpezificFactory(control.IsVertical).
          CreateMatrix(numberOfAscans, ascan.Size);
      this.control.SetResolution(numberOfAscans, ascan.Size);
      isFirstAScan = false;
```

```
    }

    this.matrix.Add(ascan);
    this.control.SetData(this.matrix.GetData(), ascan.MaxValue);

    if (this.matrix.IsFull())
    {
      isFirstAScan = true;
      this.finishedAction.Invoke();
    }
  }
}
```

Listing 4.6: Methode AddScanData

4.1.2.4 LimaTestControlVTKComponents

Die Klasse LimaTestControlVTKComponents implementiert den Zugriff auf die VTK-Objekte. Sie ist als Singleton implementiert, da die Klasse an mehreren Stellen benötigt wird. Um die Zuständigkeiten entsprechend aufzuteilen, setzt diese Klasse verschiedene Interfaces um.

IVTKViewer2DProvider stellt die verschiedenen VTK-Objekte, wie Viewer und ColorTableEditor, bereit.

IVTKDataProvider2DHolder ermöglicht den Zugriff auf den DataProvider und die Ausrichtung der VTK-Viewer.

ISaveData stellt eine Methode zur Speicherung von weiteren zu den Messdaten gehörigen Informationen zur Verfügung.

ILoadData stellt eine Methode bereit, welche eine gespeicherte Messung lädt.

IVTKCrossCursorControl ermöglicht Einstellungen der Cursor innerhalb des VTK-2D-Viewer.

4.1.2.5 Threading-Modell

In Abbildung 4.3 wird der Ablauf einer Messung dargestellt. Es werden die oberen Schichten der Darstellung und die Abstraktion InspectionSystem aus Abschnitt 4.1.1.2 ignoriert, um das Sequenzdiagramm von Abbildung 4.3 zu vereinfachen. Zudem wird auch die Kommunikation mit dem EmusHandlingWrapper durch EmusInspectionSystemImpl nicht im Diagramm dargestellt, jedoch dessen Ereignis AscanDataEvent.

1. Im ersten Schritt wird das AscanDataEvent abonniert. Das Ziel, AscanReceiverThread, wird gesetzt. Über den EmusHandlingWrapper wird der Hardware-Klasse der Start einer Messung signalisiert. Dieser Aufruf geschieht asynchron, damit die Benutzeroberfläche nicht blockiert wird.

2. War der erste Schritt erfolgreich und die Hardware-Klasse signalisiert den Start der Messung, kann AscanReceiverThread gestartet werden.

3. Nun wird eine Thread-sichere Warteschlange erstellt und der Thread zum Entnehmen der AscanData-Objekte wird gestartet.

4. Nach wenigen Sekunden wird das AscanDataEvent ausgelöst. Ein AscanData-Objekt wird mitgeschickt.
5. Dem AscanReceiverThread wird das neue Objekt übergeben.
6. Dieses Objekt wird in die Warteschlange eingereiht.
7. Der nebenläufige Thread blockiert, solange kein Objekt in der Warteschlange liegt. Sobald sich dies ändert, wird das Objekt am Ende der Schlange entnommen.
8. Die Warteschlange liefert das AscanData-Objekt.
9. Nun wird das Objekt dem Implementor AscanReceiverImpl übergeben.
10. Dort wird das AscanData-Objekt in einer Matrix gespeichert.
11. Nun wird der Inhalt der kompletten Matrix an LimaTestControlVTKComponents übergeben.
12. Der DataProvider des VTK-2D-Viewers wird mit diesen Daten gefüllt. Dies löst eine Aktualisierung des Viewers aus und somit werden auch dem Anwender die empfangenen Daten angezeigt.

Die Schritte 4 bis 12 wiederholen sich nun, bis die Messung automatisch oder manuell beendet wird. Die Betrachtung bleibt jedoch die selbe, da die auslösende Methode in beiden Fällen ausgeführt wird. Eine automatische Beendigung der Messung erfolgt, sobald die festgelegte Anzahl an Ascans erreicht wurde. Eine manuelle Beendigung erfolgt durch den Anwender.

13. Die Messung wird beendet. Sofort wird das Ereignis AscanDataEvent gekündigt, damit keine weiteren Ascans ankommen können. Daraufhin wird über den EmusHandlingWrapper das Ende der Messung signalisiert.
14. Dem AscanReceiverThread wird ebenfalls ein Halt signalisiert. Der Thread zur Entnahme der Ascans aus der Warteschlange läuft nun nur noch solange, wie Objekte in der Warteschlange sind.

4.1.3 Steuerung und Darstellung

Im folgendem wird auf die verschiedenen ViewModels und ihre Views eingegangen. Zunächst wird jedoch die Basisklasse App erläutert.

4.1.3.1 App

Die Klasse App sorgt für den Start der Anwendung. Sie legt das MainWindow an und somit die erste Darstellung, die dem Anwender präsentiert wird. An dieser Stelle werden auch die VTK-Elemente instanziiert. Dies verhindert ein Blockieren der Anwendung beim Anmelden in die Prüfansicht. Während dem Erstellen der VTK-Elemente wird dem Anwender ein Wartebildschirm in Form eines Splashscreens angezeigt.

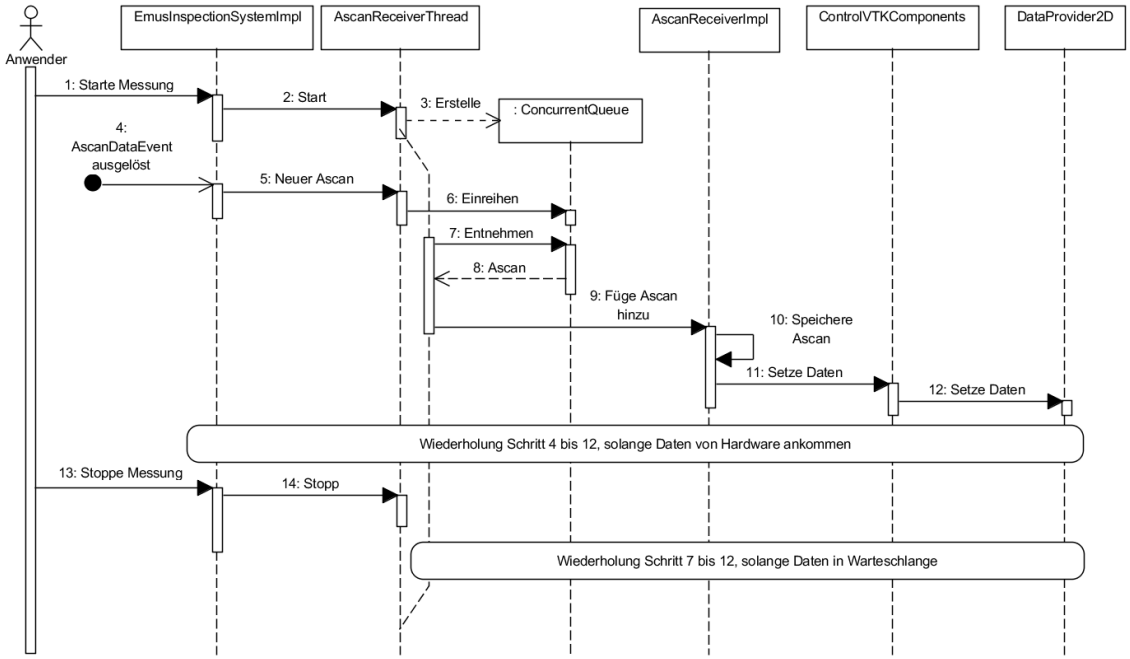

Abbildung 4.3: Threading-Modell

```
protected override void OnStartup(StartupEventArgs e)
{
  try
  {
    Splashscreen splashscreen = new Splashscreen();
    splashscreen.Show();

    base.OnStartup(e);
    MainWindow main = new MainWindow();

    buildVTKElements();

    main.Show();
    splashscreen.Close();
  }
  catch (Exception ex)
  {
    if (ShowExceptionRecursive(ex))
    {
      App.Current.Shutdown();
    }
  }
}
```

Listing 4.7: Der Start-Methode der Anwendung

In Listing 4.7 wird die überschriebene Start-Methode gezeigt. Zuerst wird ein Splashscreen erstellt und angezeigt, gefolgt vom Ausführen der geerbten Start-Methode. Die Hauptansicht MainWindow wird angelegt und die VTK-Elemente werden an eine temporäre Instanz eines WindowFormHost übergeben. Dies löst im Hintergrund den kompletten Aufbau der VTK-Elemente aus. Sind die VTK-Elemente angelegt, wird die Hauptansicht dargestellt und der Splashscreen entfernt.
Kommt es während des Programmstarts zu Ausnahmen, werden diese rekursiv dem

Anwender dargestellt und das Programm wird heruntergefahren.

4.1.3.2 MainWindow und MainViewModel

Die Klasse MainWindow ist das Hauptfenster der Anwendung. Sie stellt sich aus einer XAML- und einer Code-Behind-Datei zusammen und definiert das grundlegende Aussehen der Anwendung, indem am oberen Rand eine FraunhoferHeaderView und am unterem Rand eine StatusFooterView angezeigt werden. Im Zentrum der Anwendung kann ein beliebiges ViewModel gebunden werden, welche die Klasse ViewModelBase implementiert.

Durch das Anlegen von DataTemplates werden die Views und ViewModels, die auf dem MainWindow angezeigt werden können, miteinander verbunden. Wird eine Instanz eines ViewModels an ein ContentControl gebunden, wird die entsprechende View an dieser Stelle angezeigt.

Beim Anlegen des MainViewModel wird je eine Instanz des FraunhoferHeaderViewModel, des StatusFooterViewModel und des UserLoginViewModel erstellt. Da das FraunhoferHeaderViewModel für das Umschalten der Sprache zuständig ist, sorgt das MainViewModel für das Abonnieren des Ereignisses mit den entsprechenden Methoden aus dem StatusFooterViewModel und dem UserLoginViewModel. Die Sprachumschaltung wird in Abschnitt 4.3 behandelt.

4.1.3.3 FraunhoferHeaderView und -ViewModel

Die FraunhoferHeaderView zeigt folgende Informationen:

- Logo des Fraunhofer IZFP
- Name der LimaTest-Software
- Name des angemeldeten Benutzers
- Das aktuelle Datum und die Uhrzeit im spezifischen Format der aktuell ausgewählten Kulturinformation

Außerdem kann sich der Anwender über einen Button am oberem rechten Rand aus der Prüfansicht abmelden. Befindet sich der Anwender im Anmeldebereich, kann über diesen Button die Anwendung beendet werden. Zum Umschalten des Buttons wird ein Style verwendet. Der Style im Listing 4.8 verwendet als Basis das für die Anwendung definierte Aussehen eines Buttons. Somit unterscheidet er sich äußerlich nicht von den anderen Buttons. In Abhängigkeit des bool'schen Wertes Logged, wird der angezeigte Text und die Bindung an das Kommando gewechselt. Logged, ShutdownCmd und LogoutCmd sind Eigenschaften des FraunhoferHeaderViewModels. Die Texte werden mittels dynamischer Bindung konfiguriert. Genaueres dazu gibt es in Abschnitt 4.3.

```
<Button>
 <Button.Style>
  <Style TargetType="{x:Type Button}" BasedOn="{StaticResource ResourceKey={x:Type
      Button}}">
   <Setter Property="Content" Value="{DynamicResource FraunhoferHeader-BtnLogout}"/>
   <Setter Property="Command" Value="{Binding LogoutCmd}"/>
   <Style.Triggers>
    <DataTrigger Binding="{Binding Logged}" Value="False">
```

```
    <Setter Property="Content" Value="{DynamicResource FraunhoferHeader-BtnExit}"/>
    <Setter Property="Command" Value="{Binding ShutdownCmd}"/>
   </DataTrigger>
  </Style.Triggers>
 </Style>
</Button.Style>
</Button>
```

Listing 4.8: Style des Abmelden/Beenden-Buttons

Unterhalb des Buttons befindet sich eine Combobox, welche zur Auswahl einer der verfügbaren Sprachen dient.

Das FraunhoferHeaderViewModel implementiert die Sprachauswahl, die Anzeige der Uhrzeit und des Datums. Wie die Sprachauswahl funktioniert, wird in Abschnitt 4.3 behandelt.

4.1.3.4 StatusFooterView und -ViewModel

In der StatusFooterView wird der aktuelle Status der Anwendung angezeigt. Der Status ist an eine Enumeration gebunden, welche die möglichen Status abbildet. Durch einen Konverter kann aus der Enumeration ein String generiert werden, der von der aktuellen Sprachdatei stammt. Im Listing 4.9 wird dessen Anwendung demonstriert.

```
<Label Content="{Binding UIStatus, Converter={StaticResource StatusToResourceConverter}}
    "/>
```

Listing 4.9: Label mit dem aktuellem Status

Die Konvertierungsmethode aus Listing 4.10 verwendet ausschließlich den übergebenen Wert (value). Zunächst wird geprüft, von welchem Aufzählungstyp der Wert ist. Danach erfolgt die Suche in der aktuellen Sprachdatei anhand der Ausgabe der toString-Methode.

```
public object Convert(object value, Type targetType, object parameter, System.
    Globalization.CultureInfo culture)
{
  String status = String.Empty;
  if (value is UIStates)
  {
    status = ((UIStates)value).ToString();

  }
  else if (value is SoftwareState)
  {
    status = ((SoftwareState)value).ToString();
  }

  return Application.Current.FindResource(status);
}
```

Listing 4.10: Konvertierungsmethode des StatusToResourceValueConverter

Sie bietet zudem einen ToggleButton (Listing 4.11), der den aktuellen Verbindungsstatus zur Hardware zeigt. Ist die Hardware nicht verbunden, kann über diesen Button ein

neuer Verbindungsversuch gestartet werden. Während diesem Versuch kann die Schaltfläche nicht erneut betätigt werden. Da ein ToggleButton normalerweise nur zwei Zustände kennt, gedrückt und nicht gedrückt, wird über die Eigenschaft IsEnabled ein weiterer Zustand hinzugefügt. Diese Eigenschaft ist an einen bool'schen Wert des StatusFooterViewModel gebunden. Dieser wird, solange ein Hintergrundprozess versucht, eine Verbindung aufzubauen, negiert.

```
<ToggleButton ... IsEnabled="{Binding CanExecuteConnector}">
 <ToggleButton.Style>
  <Style TargetType="{x:Type ToggleButton}" BasedOn="{StaticResource
      ToggleButtonWithColor}">
   <Setter Property="Content" Value="{DynamicResource StatusFooterView-Disconnected}"/>
   <Setter Property="Background" Value="Red"/>
   <Setter Property="Command" Value="{Binding ConnectCmd}"/>
   <Setter Property="IsChecked" Value="{Binding IsConnected}"/>
   <Style.Triggers>
    <Trigger Property="IsChecked" Value="True">
     <Setter Property="Content" Value="{DynamicResource StatusFooterView-Connected}"/>
     <Setter Property="Background" Value="Green"/>
    </Trigger>
    <Trigger Property="IsEnabled" Value="False">
     <Setter Property="Content" Value="{DynamicResource StatusFooterView-Reconnecting}"/
         >
     <Setter Property="Background" Value="Gray"/>
    </Trigger>
   </Style.Triggers>
  </Style>
 </ToggleButton.Style>
</ToggleButton>
```

Listing 4.11: Style des Verbindungsbutton

Das StatusFooterViewModel stellt mittels des IReceiveHardwareStatusChanges-Interface eine Methode bereits, welches die Prüfansichten, LimaTestControlViewModel und LimaTestServiceViewModel, dazu verwenden, den Status der letzten Tätigkeit zu setzen. Unter die Statusmeldungen fallen:

- Erfolgreiches und fehlgeschlagenes Starten einer Messung
- Erfolgreiches und fehlgeschlagenes Stoppen einer Messung
- Das Speichern und Laden einer vorangegangen Messung
- Das erfolgreiche Parametrieren der Hardware bzw. Änderungen im Parametersatz
- u.ä.

4.1.3.5 UserLoginView und -ViewModel

Die Funktion der UserLoginView ist es, dem Anwender einen Anmeldebildschirm und Benachrichtigungen über den Zustand der Soft- und Hardware anzuzeigen. Entsprechende Zustände sind als Enumeration angelegt und werden wie in Abschnitt 4.1.3.4 verwendet. Für die Auswahl eines Benutzerprofils wird eine Combobox angezeigt. Das Feld zur Passworteingabe wird ausgeblendet, wenn kein Passwort vom ausgewählten Benutzerprofil erwartet wird.

Die auswählbaren Benutzerprofile lädt das ViewModel von der Klasse UserAdministration, welche für die Verwaltung von Benutzerprofilen verantwortlich ist. Zudem ist die UserAdministration-Klasse für das Prüfen des eingegebenen Passwortes verantwortlich.

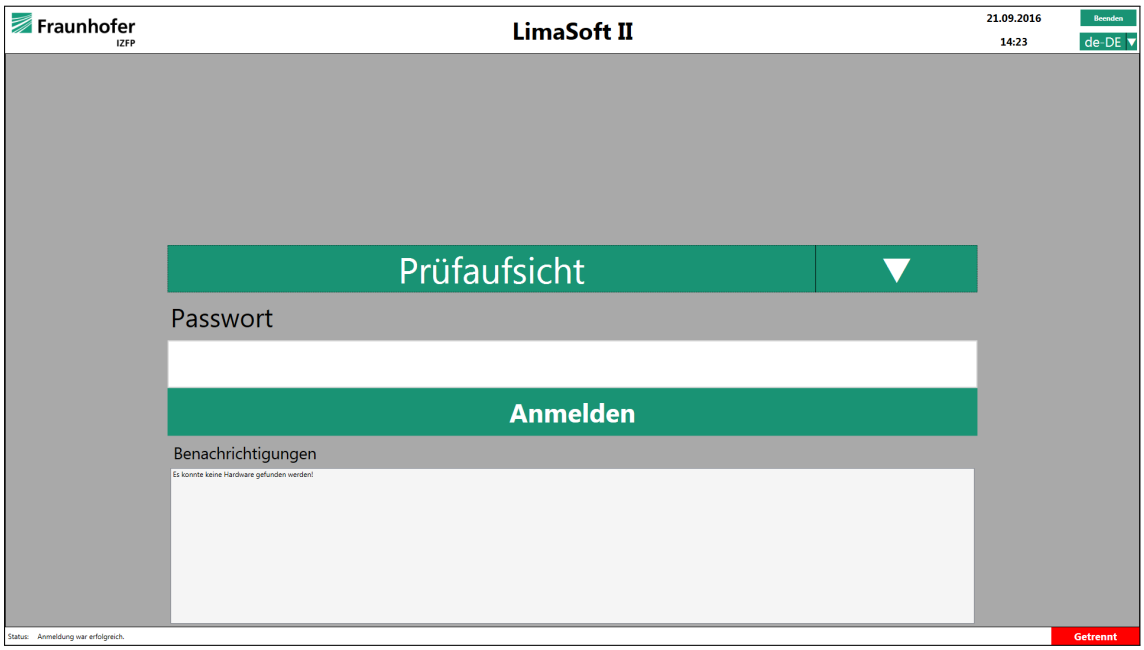

Abbildung 4.4: UserLoginView

4.1.3.6 LimaTestMainViewModel

Das LimaTestMainViewModel ist die Basisklasse von LimaTestControlViewModel und -ServiceViewModel. Es definiert einige Eigenschaften, die von beiden Unterklassen benötigt werden. Dazu zählen:

- Eine Hash-Tabelle für BackgroundWorker, die dem asynchronen Aufruf von Methoden der Klassen der Hardwaresteuerung dienen.
- Ein Ereignis, damit die Kommandos zum Starten und Stoppen von Messungen aktiviert und deaktiviert werden können, abhängig von einer entsprechenden bool'schen Eigenschaft.
- Ein Objekt der Klasse ProfilesViewModel (siehe Abschnitt 4.1.3.10)
- Eine Referenz auf ein Objekt, das das Interface IReceiveHardwareStatusChanges implementiert.

4.1.3.7 LimaTestControlView und -ViewModel

Die LimaTestControlView ist eine der beiden Prüfansichten. Sie dient als Bedienoberfläche der normalen Benutzer und des Administrators. Über diese View können Messungen durchgeführt werden und die empfangenen Messdaten werden an dieser Stelle angezeigt. Wie Messungen vonstatten gehen, wird in Abschnitt 4.1.2.5 erklärt. Die VTK-Viewer des LimaTestControlVTKComponents (Abschnitt 4.1.2.4) erhalten die View über das Interface IVTKViewer2DHolder. Der Zugriff erfolgt innerhalb der Code-Behind-Datei. Beim Laden der View werden zunächst die VTK-Viewer als Kindelemente an den WindowsFormHosts

übergeben. Danach erfolgt das Setzen einer Methode zum Speichern eines Bildschirmfotos. Diese wird nach einer Messung ausgeführt.

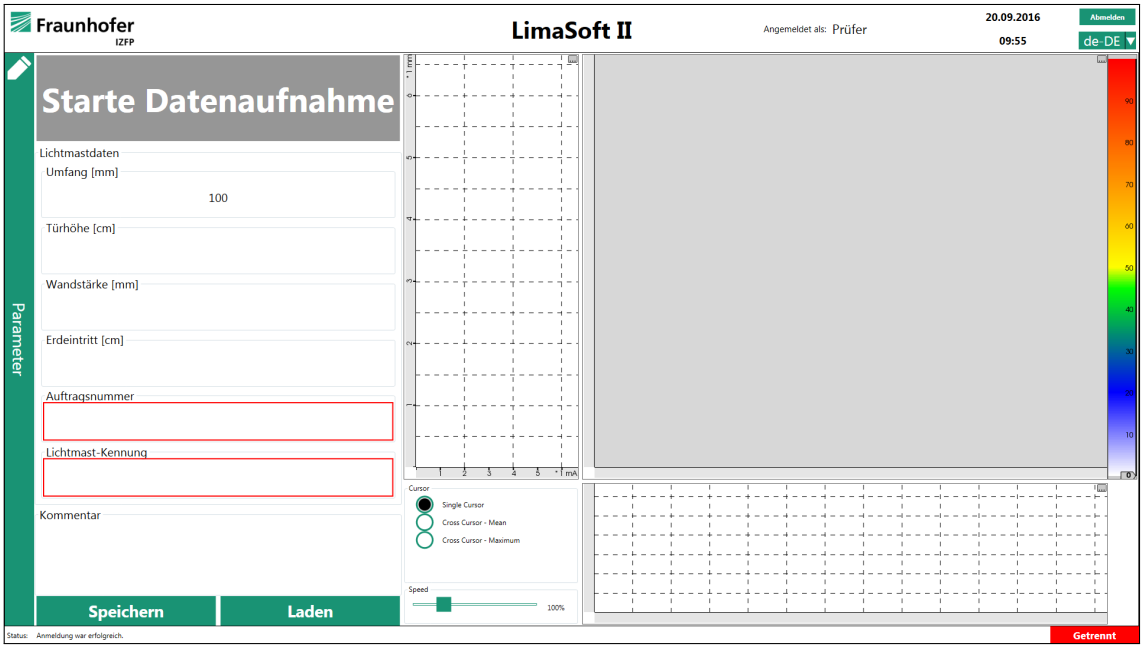

Abbildung 4.5: Prüfansicht

Loggt man sich als Administrator ein, werden Funktionen freigeschaltet, die der normale Benutzer nicht verwenden darf. Dazu zählt zum Einen die Verwaltung von Benutzerprofilen, dazu weiteres in Abschnitt 4.1.3.9, sowie das Setzen der Rechte zum Verändern der einzelnen Parameter. Dies wird umgesetzt, indem man das Style-Property eines ContentControl dazu verwendet, abhängig vom angemeldeten Benutzer, verschiedene DataTemplates zu benutzen (Listing 4.12).

```
<ContentControl Content="{Binding}" ...>
 <ContentControl.Style>
  <Style TargetType="ContentControl">
   <Setter Property="ContentTemplate" Value="{StaticResource MeasuringTab}"/>
   <Style.Triggers>
    <DataTrigger Binding="{Binding ProfilesVM.UserRole, Converter={StaticResource
        UserRoleEnumToBooleanConverter}}" Value="True">
     <Setter Property="ContentTemplate" Value="{StaticResource AdministationTabControl}"
         />
    </DataTrigger>
   </Style.Triggers>
  </Style>
 </ContentControl.Style>
</ContentControl>
```

Listing 4.12: Bindung unterschiedlicher Darstellung mittels DataTrigger

Das LimaTestControlViewModel stellt einige Eigenschaften bereit, die durch die View gesetzt werden können und zur Protokollierung der Messungen dienen. Der Anwender muss mindestens die Auftrags- und die Mastnummer eintragen, damit der Start der Messung freigegeben wird. Es werden BackgroundWorker angelegt, welche Start und Stopp der Messung verarbeiten. BackgroundWorker haben Methoden, welche bei ihrem Start

und Ende ausgeführt werden. Die Kommandos starten die BackgroundWorker asynchron, damit die Benutzeroberfläche nicht blockiert. Um ein mehrfaches Ausführen der BackgroundWorker zu verhindern, wird zuvor geprüft, ob diese nicht bereits laufen.

```
this.worker = new BackgroundWorker();
this.worker.DoWork += doSomething;
this.worker.RunWorkerCompleted += doFinalActivity;

Command startWorkerCmd = new Command(() =>
{
  if (!worker.IsBusy)
    worker.RunWorkerAsync();
}, new Predicate<object>(b => this.CanExecute));
this.CanExecute += new EventHandler(startWorkerCmd.RaiseCanExecuteChanged);

this.StartWorkerCmd = (ICommand)startWorkerCmd;
```

Listing 4.13: Beispiel: BackgroundWorker und Command

Im Listing 4.13 wird beispielhaft gezeigt, wie in den Klassen LimaTestControlViewModel und LimaTestServiceViewModel die Kommandos mit den Funktionalitäten des BackgroundWorker ausgestattet werden. Nach der Instanziierung des *worker* werden die entsprechenden Methoden registriert zum Verarbeiten der Ereignisse DoWork und RunWorkerCompleted. Es wird eine neue Instanz von Command erstellt. Command implementiert das ICommand-Interface, welches per Command-Binding an einen Button der View gebunden werden kann. In der Aktion, welche durch das Kommando ausgeführt werden soll, wird zunächst geprüft, ob der BackgroundWorker nicht bereits beschäftigt ist. Ein weiterer Start des Workers würde zu einer Ausnahme führen. Ist dies nicht der Fall, wird der BackgroundWorker asynchron gestartet. Durch das Setzen eines Predicate-Delegaten kann dafür gesorgt werden, dass das Kommando nur ausgeführt wird, wenn dessen Bedingung erfüllt ist. Dadurch wird es auch notwendig, die Prüfung der Bedingung auszulösen. Daher wird die Methode RaiseCanExecuteChanged beim Ereignis CanExecute von der Basisklasse LimaTestMainViewModel registriert.

Anzumerken ist, dass bei der Klasse LimaTestControlViewModel weitere Refaktorisierungen notwendig sind. Die Klasse wuchs iterativ und zu Beginn wurden noch weitere Klassen, wie SupervisorViewModel und ProfilesViewModel, erstellt, um die Zuständigkeiten besser aufzuteilen. Als nächstes werden die einzutragenden Lichtmastdaten, wie Umfang, Wanddicke und Erdeintritt, in ein eigenes ViewModel überführt, welches durch das LimaTestControlViewModel instanziiert wird. Danach soll die Umsetzung einer Validierung der Lichtmastdaten erfolgen.

4.1.3.8 LimaTestServiceView und -ViewModel

Die LimaTestServiceView zeigt drei VTK-Viewer zur Betrachtung der empfangenen Ascans. Das LimaTestViewModel implementiert das IReceiveMultipleAscans selbst und schreibt die Daten selbst in DataProvider der 1D-Viewer. Beim Start einer Kalibrierung wird eine Referenz an die StartCalib-Methode des Singleton InspectionSysten übergeben.

4.1.3.9 SupervisorView und -ViewModel

Über die SupervisorView können neue Benutzer angelegt und vorhandene Benutzer entfernt werden. Es können jedoch keine weiteren Administrator- und Service-Benutzerprofile

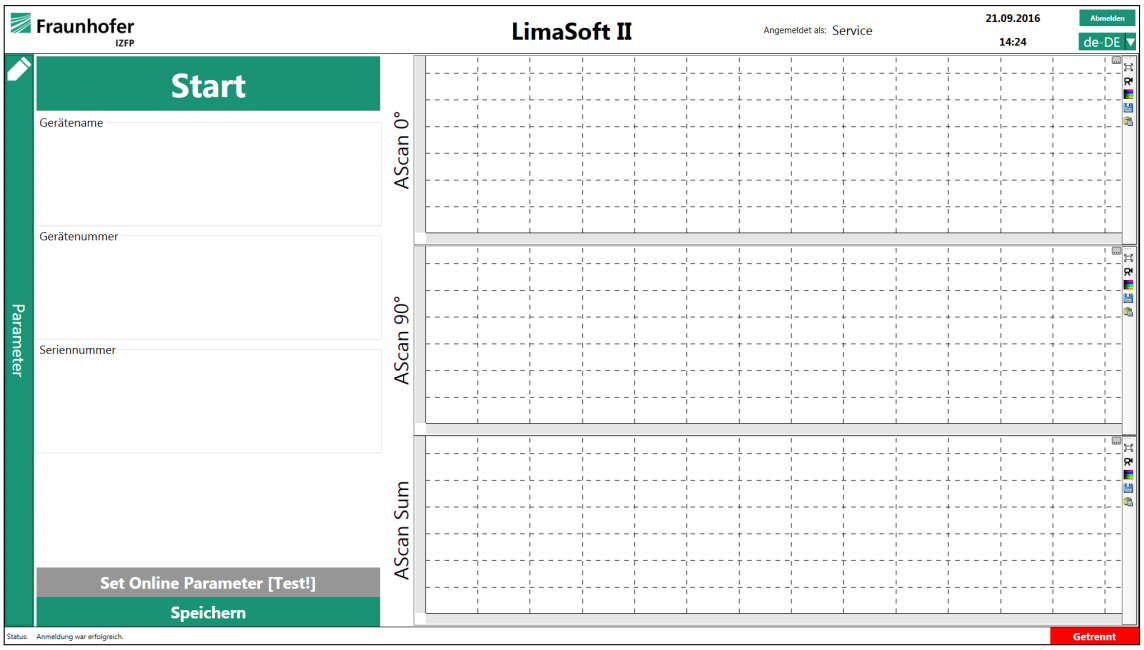

Abbildung 4.6: Service

erstellt werden. Diese View ist nur für den Administrator zugänglich. Das ViewModel enthält eine Referenz auf ein Objekt der Klasse UserAdministration, worüber die eigentliche Verwaltung der Benutzerprofile stattfindet.

4.1.3.10 ProfilesView und -ViewModel

Das ProfilesViewModel dient der Verwaltung von Parameterprofilen. Es können neue Profile angelegt und existierende kopiert oder gelöscht werden. Sie enthält zudem eine Liste von ParameterViewModels, welche bei einer Änderung des aktuell ausgewählten Profiles neu gesetzt wird.

4.1.3.11 ParameterViews und -ViewModels

Für das Durchführen von Messungen müssen verschiedene Parameter festgelegt werden, die auch von unterschiedlichen Datentyp sind. Daher wurde eine flache Hierarchie entwickelt, die leicht zu erweitern ist (Abbildung 4.7).

Durch eine abstrakte Basisklasse lässt sich eine Liste mit Objekten des ParameterBaseViewModels anlegen und bearbeiten. Mittels der generischen Klasse ParameterViewModel können primitive Datentypen als Typ des Wertes Value verwendet werden. Die ParameterView besitzt keinerlei Abhängigkeiten zum verwendeten Datentyp, sie zeigt jede Instanz des ParameterViewModels gleich an. Muss für einen bestimmten Datentyp eine andere View verwendetet werden, so leitet man die generische Klasse mit dem entsprechenden Datentyp ab.

Hier wurde für das Anzeigen und Bearbeiten von bool'schen Werten das BoolParameterViewModel und die BoolParameterView erstellt. Die BoolParameterView zeigt, statt einer Texteingabe und der Einheit, eine Checkbox an. In der Abbildung 4.8 wird eine Liste von ParameterViewModels angezeigt, darunter auch ein BoolParameterViewModel.

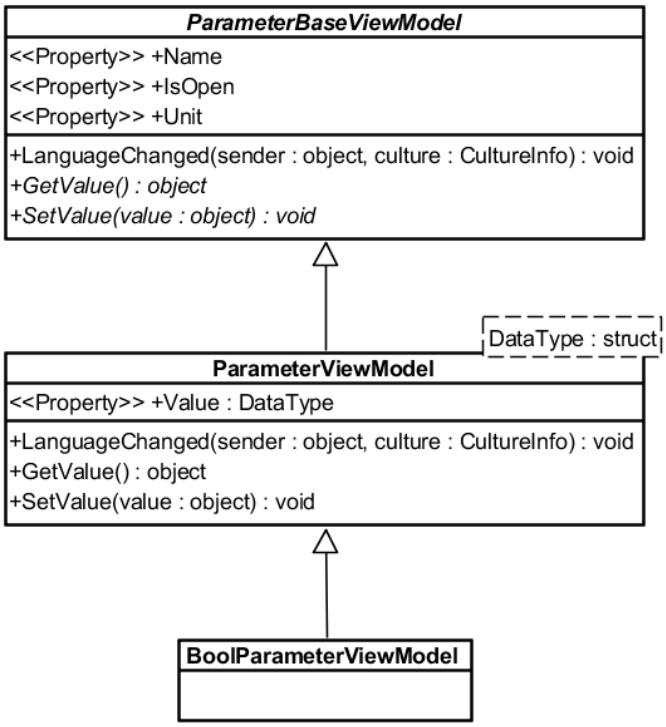

Abbildung 4.7: ParameterViewModel

Der Name, der im ParameterViewModel gesetzt wird, hat zwei Aufgaben. Zum einen wird der Name zum Binden an eine Sprachdatei verwendet. In Listing 4.25 im Abschnitt 4.3 wird eine solche Bindung demonstriert. Der Name wird außerdem dazu verwendet, über Reflections auf die Eigenschaften der Klasse EmusParameters zuzugreifen. Somit kann der Wert innerhalb eines ParameterViewModels geändert werden, ohne dessen Datentyp zu kennen. Umgekehrt kann ein EmusParameters-Objekt angepasst werden, ohne es komplett neu aus der Sammlung von ParameterViewModel zu generieren.

In Listing 4.14 wird beispielhaft die Verwendung von Reflexion gezeigt. Über den typeof-Operator kann auf Metadaten von EmusParameters zugegriffen werden. Mit der Methode SetValue der Klasse FieldInfo kann der Wert aus dem ParameterViewModel

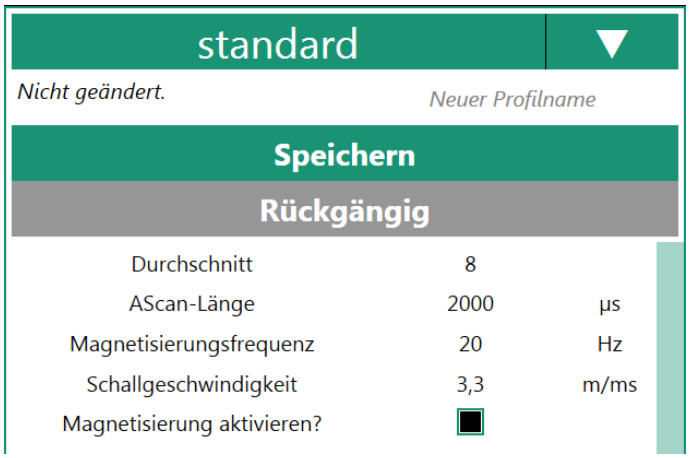

Abbildung 4.8: ProfilesView

gesetzt werden. Dazu benötigt man lediglich eine beliebige Instanz. Dies wird zum Beispiel dann verwendet, wenn sich der Wert eines ParameterViewModel verändert und das PropertyChanged-Ereignis ausgelöst wird.

```
EmusParameters parameters;

FieldInfo field = typeof(EmusParameters).GetField(parameterVM.Name);
field.SetValue(parameters, parameterVM.GetValue());
```

Listing 4.14: Beispiel: Verwendung von Reflections zum Setzen von einzelnen Werten

4.1.4 Datenhaltung

Für die Verwendung der Software müssen verschiedene Daten gespeichert werden.

- Die Daten des Rechtesystems, welches die Parameter festlegt, die der Prüfer ändern darf.
- Die gehashten Passwörter der Benutzerprofile Administrator und Service.
- Den Speicherort der abgeschlossen Messungen.
- Die Parametersätze, die von den Benutzern angelegt werden. Das Service-Benutzerprofil kann zusätzlich Parametersätze sperren, damit diese ausschließlich dem IZFP-Service zur Verfügung stehen.

Die Daten werden im XML-Format abgelegt, um zum einem eine gute Lesbarkeit zu erhalten und zudem die Software von außen leicht parametrierbar zu machen.

4.2 Interaktion zwischen Native und Managed Code

Die Klassenbibliothek der Emus-Prüfhardware benötigt eine Funktion, die aufgerufen wird, sobald Messdaten von der Hardware empfangen werden. Da die Klassenbibliothek jedoch in nativem C++ geschrieben wurde und die oberen Schichten in C# programmiert werden, muss eine Zwischenschicht in C++/CLI entwickelt werden.

4.2.1 Ablauf einer Messung aus Software-Sicht

Die Klasse EmusHandling der Hardware-Klassenbibliothek [Osw16] sorgt für den Zugriff auf die Klasse EmusFrontend, die wiederum den Zugriff auf die eigentlich Prüfhardware steuert. Wird eine Messung ausgelöst, wird im EmusFrontend ein Thread zum Empfangen der Messdaten gestartet. Bei jedem empfangenen Paket an Messdaten wird die übergebene Funktion ausgeführt. Diese muss dann dafür sorgen, dass die Messdaten an die darüber liegenden Schichten weitergegeben werden.

4.2.2 Planung

Für die native Klassenbibliothek muss eine Wrapper-Klasse in C++/CLI angelegt werden. Die Wrapper-Klasse definiert einen Delegaten und entsprechende Methoden, die diesen Delegaten implementieren. In einer Methode soll aus einem Delegaten ein Funktionspointer generiert werden, der anschließend an die Prüfhardware-Klasse übergeben wird.

```
bool EmusHandlingWrapper::setDataEventFunction(GetAScanData^ fp)
{
  GCHandle gch = GCHandle::Alloc(fp);
  System::IntPtr ip = Marshal::GetFunctionPointerForDelegate(fp);
  void(*cb)() = static_cast<void(*)()>ip.ToPointer());
  System::GC::Collect();

  return this->nativeHandling->getHardWare()->setDataEventFunction(cb);
}
```

Listing 4.15: Vom Delegaten zum Funktionspointer

4.2.3 Implementierung

Umgesetzt wurde die Wrapper-Klasse als EmusHandlingWrapper. Seine allgemeine Funktion wird im Abschnitt 4.1.1.4 erklärt. Der Ablauf, um aus einem Delegaten einen Funktionspointer zu generieren, lässt sich in fünf Schritte einteilen:

1. Zunächst wird verhindert, dass das Delegatenobjekt vom Garbage Collector erfasst wird, bevor es seine Aufgabe erledigt hat.
2. Durch den Aufruf einer Interop-Methode wird aus dem Delegaten, anders als der Methodenname vermuten lässt, ein Integerzeiger.
3. Es folgt eine Typumwandlung des Integerzeiger in einen Funktionszeiger.
4. Der Funktionszeiger wird der nativen Klassenbibliothek übergeben.
5. Danach erfolgt ein expliziter Aufruf des Garbage Collectors.

Genau diese fünf Schritte werden in der Methode aus dem Listing 4.15 durchgeführt.

4.2.4 Implementierungen des Delegaten

Der zu implementierende Delegat besitzt void als Rückgabetyp und verfügt über keine Übergabeparameter und besitzt somit die von der Hardware-Klassenbibliothek vorgegebenen Signatur. Zusätzlich müssen sie statisch deklariert sein, damit sie unabhängig eines spezifischen Objektes von EmusHandlingWrapper aufgerufen werden können. Dies hat natürlich den Nachteil, dass die Methoden nicht mehr auf private Felder ihrer Klasse zugreifen können. Daher muss die Klasse eine private statische Instanz von sich selbst bereit halten. Es wurde so implementiert, das von dieser Klasse nur eine einzige Instanz erstellt werden kann. Jeder weitere Versuch löst eine Ausnahme zur Laufzeit aus.

Innerhalb der statischen Methoden wird auf die native Klasse EmusHandling zugegriffen, um die empfangen Messdaten weiterverarbeiten zu können. Um auch hier eine einfache Weiterleitung der Messdaten zu erhalten, werden sie über ein Ereignis an die nächst höhere Schicht weitergegeben.

4.3 Dynamische Sprachumstellung

Eine der Zielsetzungen war, eine dynamische Sprachumstellung während der Programmlaufzeit zu ermöglichen. Diese sollte für den Anwender möglichst einfach zu bedienen sein und es sollen weitere Sprachen in der Software implementiert werden können, ohne diese neu zu kompilieren.

4.3.1 Problembeschreibung

Leider werden in WPF, anders als noch in Windows Forms, die Informationen der aktuell eingestellten Kultur nicht übernommen. So verwenden die WPF-Elemente standardmäßig die englisch-amerikanische Kultur, auch auf einem Windows-Betriebssystem mit deutscher Lokalisierung.
Über die Verwendung eines CultureConverter können zwar Zahl und Datum in beliebiger Lokalisierung anzeigen werden, jedoch lässt sich dieser Konverter nicht an eine Eigenschaft binden. Somit bleibt die eingetragene Lokalisierung nach dem Kompilieren die gleiche. Es gibt die Möglichkeit, zum Programmstart eine bestimmte Sprache zu definieren, die dann auch in der Applikation verfügbar ist. Jedoch lässt sich diese Funktion nur einmalig ausführen [Kal10]. Da eine dynamische Sprachauswahl gewünscht ist, muss eine andere Lösung gefunden werden.

4.3.2 Voraussetzung

Damit Texte ausgetauscht werden können, müssen diese in WPF an ein String-Objekt innerhalb eines sogenannten ResourceDictionary gebunden werden. Diese werden in XAML geschrieben. Die String-Objekte besitzen einen Schlüssel und den Textinhalt. Die WPF-Textelemente, wie Label oder TextBox, verwenden diese Schlüssel zum Laden der Textinhalte, indem sie statische oder dynamische Bindung verwenden.

In einem ResourceDictionary (Listing 4.16) befinden sich zwei String-Objekte.

```
<ResourceDictionary ...>
 <system:String x:Key="Bezeichner">Hier steht Beispieltext!</system:String>
</ResourceDictionary>
```

Listing 4.16: Beispiel: ResourceDictionary mit String-Objekten

In einem WPF-Control (Listing 4.17), zum Beispiel in einem UserControl, können beliebige Steuerelemente, die Text anzeigen können, nun auf diese Ressource zugreifen, indem sie deren Schlüssel verwenden.

```
...
<TextBox Text="{DynamicResource Bezeichner}" />
...
```

Listing 4.17: Beispiel: WPF-Textelemente mit dynamischer Bindung

Wird das WPF-Control in einem Programm verwendet und dieses ausgeführt, dann werden erst zur Laufzeit die Schlüssel ausgewertet und die entsprechenden Ressourcen eingesetzt.

Leider büßt man durch das Verwenden von dynamischem Binden Performanz ein, da die Ressource zur Laufzeit gesucht werden muss. Jedoch kann mit DynamicResource das Ladeverhalten verbessert werden, da die Ressource erst bei Bedarf abgerufen wird [Weg13, S. 147].

4.3.3 Aufbau

Es gibt drei Klassen, die die Sprachauswahl ermöglichen (Abbildung 4.9):

Language dient der Abstraktion. Objekte dieser Klasse enthalten den Bezeichner der Lokalisierung.

LanguageUtil dient dem Laden der Sprachen und der Umschaltung der Software in die gewünschte Sprache.

LimaTestHeaderViewModel stellt verschiedene Eigenschaften bereit. Zum Einen hält sie eine Liste der verfügbaren Sprachen und eine Referenz auf die derzeit selektierte Sprache. Außerdem besitzt es Eigenschaften für die Uhrzeit und das Datum. Da ein Sprachwechsel gegebenenfalls Auswirkungen auf andere Objekte haben kann, implementiert das LimaTestHeaderViewModel das dafür erstelle Ereignis LanguageChangedEventHandler.

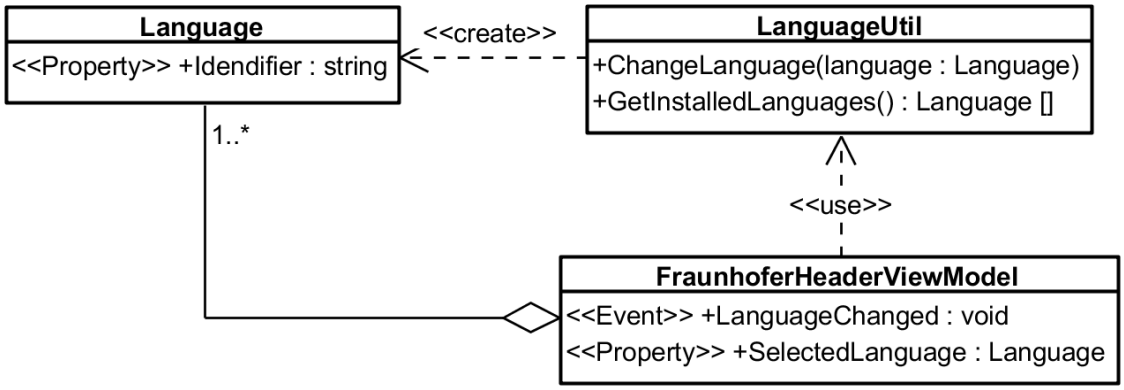

Abbildung 4.9: Kompontenten der Sprachauswahl

4.3.4 Aufbau von Sprachdateien

Da die Sprachdateien ResourceDictionaries sind und somit auch andere Objekte, wie WPF-Controls, enthalten können, wurde eine Namenskonvention eingeführt. Der Dateiname muss mit „CultureDictionary“ beginnen. Es folgt ein Punkt und die Bezeichnung der Sprache, zum Beispiel „de-DE“ oder „en-US“. Der Dateiname endet mit der üblichen Dateierweiterung „.xaml“.

Die Sprachdatei beginnt mit dem Kopf (Listing 4.18). Dieser enthält zunächst die üblichen Inkludierungen, damit XAML-Code verwendet werden kann. Danach wird noch aus dem Namensraum System das Assemby mscorlib inkludiert, damit String-Objekte angelegt werden können.

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:system="clr-namespace:System;assembly=mscorlib">
```

Listing 4.18: Kopfzeilen einer Sprachdatei

Es folgen die String-Objekte, wie sie in Listing 4.19 beispielhaft gezeigt werden. Für die Bezeichnung wurde ebenfalls eine Namenskonvention eingeführt. Zum einem wird das WPF-Control, wenn möglich, spezifiziert, welches den Text verwenden wird. Auf

eine Trennung der Sprachdateien, damit jedes WPF-Control seine eigene Sprachdatei verwendet, wurde aufgrund der Menge der daraus resultierenden Sprachdateien zunächst verzichtet. In der Bezeichnung wird zudem der Kontext des Textes spezifiziert. Löst beispielsweise das Betätigen eines Button das Löschen eines Profiles aus und dies ist aus dem Kontext zu schließen, verwendet man als Text natürlich nur "Löschen".

```
<system:String x:Key="WpfControl-Context">Text</system:String>
```

Listing 4.19: String-Objekt einer Sprachdatei

Die Sprachdatei endet mit dem Abschluss des ResourceDictionary (Listing 4.20).

```
</ResourceDictionary>
```

Listing 4.20: Abschluss einer Sprachdatei

4.3.5 Funktionsweise von LanguageUtil

Die Klasse LanguageUtil verwendet eine Menge von ResourceDictionaries. Diese befinden sich im Programmordner und werden beim Programmstart geladen. Damit diese nicht bei jedem Sprachwechsel neu geladen werden müssen, werden sie in einer statischen Liste bereitgestellt. ResourceDictionaries können während der Laufzeit innerhalb der Anwendung entfernt und hinzugefügt werden. Durch das Hinzufügen einer Sprachdatei wechselt die dynamische Bindung der WPF-Elemente auf das neue ResourceDictionary. Nun muss nur dafür gesorgt werden, dass kein ResourceDictionary mehrfach eingebunden wird. Das mehrfache Hinzufügen hat zwar keine Auswirkungen auf den Sprachwechsel, jedoch wird die Liste an gebundenen ResourceDictionaries mit jedem Sprachwechsel größer und somit der Speicherverbrauch der Anwendung.

Der aktuelle Thread muss die Kulturinformationen der aktuellen Sprache erhalten, damit die Konvertierung von Fließkommazahlen durch den StringConverter funktioniert. Dazu benötigt dieser lediglich den Bezeichner der Sprache. Zusätzlich wird noch die aktuelle Sprache als Standard für neue Threads definiert.

4.3.6 Darstellung von Fließkommazahlen

Leider aktualisiert sich die Darstellung von Fließkommazahlen nicht nach einem Sprachwechsel. Stattdessen wird standardmäßig die englische Zahlendarstellung mit einem Punkt als Dezimaltrennzeichen verwendet. Um dieses Problem zu lösen, musste ein Konverter entwickelt werden, der StringConverter, der aus einer Zahl ein String-Objekt generiert (Listing 4.21) und umgekehrt, aus einem String-Objekt die ursprüngliche Zahl wiederherstellt (Listing 4.22). Außerdem müssen alle entsprechenden Eigenschaften beim Auslösen des Ereignisses auch das Ereignis PropertyChanged auslösen, damit diese auf der View aktualisiert werden.

4.3.6.1 Funktionweise der Konvertierungsmethoden

Aufgrund der Tatsache, dass der Konverter innerhalb eines WPF-Textelements verwendet wird, muss der übergebene Wert (value) in einen String (targetType) konvertiert wer-

den (Listing 4.21). Jedoch übergibt das aufrufende WPF-Element nur die amerikanisch-englischen Kulturinformationen (culture). Daher wird auf dessen Verwendung verzichtet und die derzeitig verwendeten Kulturinformationen genutzt. Diese werden durch das LanguageUtil richtig gesetzt.

```
public object Convert(object value, Type targetType, object parameter, System.
    Globalization.CultureInfo culture)
{
  return System.Convert.ChangeType(value, targetType, Thread.CurrentThread.
      CurrentCulture);
}
```

Listing 4.21: Konvertierung von einem beliebigen Wert in einen String

Bei Rückkonvertierung (Listing 4.22) wird mithilfe der Klasse TypeDescriptor ein Typenkonverter entsprechend dem übergebenen Typ (targetType) referenziert. Der übergebene Typ entspricht dem Typ des Wertes (value), wie er an das WPF-Textelement durch das ViewModel gebunden ist. Konnte ein entsprechender Typenkonverter gefunden werden und diesem ist es auch möglich, String zu konvertieren, erfolgt die Umwandlung des übergebenen Wertes vom String zum ursprünglichen Typ. Auch hier wird statt der übergebenen die aktuell verwendete Kulturinformation benutzt.

```
public object ConvertBack(object value, Type targetType, object parameter, System.
    Globalization.CultureInfo culture)
{
  var converter = TypeDescriptor.GetConverter(targetType);
  if (converter != null && converter.CanConvertFrom(typeof(string)))
  {
    try
    {
      var result = converter.ConvertFrom(null, Thread.CurrentThread.CurrentCulture,
          value);
      return result;
    }
    catch (Exception) { }
  }
  return DependencyProperty.UnsetValue;
}
```

Listing 4.22: Konvertierung eines Strings in einen passenden Wert

4.3.6.2 Verwendung des Konverters

Bei der Verwendung des StringConverter muss darauf geachtet, dass die Eigenschaft UpdateSourceTrigger explizit auf den Wert LostFocus eingestellt ist. Der Wert PropertyChanged würde nach jeder Veränderung des Textes der TextBox einen Aufruf des StringConverters zur Folge haben. Wenn zum Beispiel eine Fließkommazahl eingegeben wird, wäre es nicht möglich, ein Dezimaltrennzeichen einzugeben, da der Konverter diesen entfernen würde. Der Konverter würde eine nicht vollständige Fließkommazahl erkennen und nur den ganzzahligen Teil der Zahl darstellen.

In Listing 4.23 wird die korrekte Anwendung am Beispiel eines UserControl demonstriert, einschließlich der Inkludierung des entsprechenden Namensraumes und dem An-

legen einer Instanz des StringConverters.

```
<UserControl ...
 xmlns:converters="clr-namespace:LimaTestII_WPF.Utilities.Converter">
  <UserControl.Resources>
   <converters:StringConverter x:Key="StringConverter"/>
 </UserControl.Resources>
...
 <TextBox Text="{Binding Value, Converter={StaticResource StringConverter},
     UpdateSourceTrigger=LostFocus}"/>
...
</ResourceDictionary>
```

Listing 4.23: Beispiel: Verwendung des StringConverter

4.3.7 Darstellung von String-Eigenschaften

Ein ähnliches Problem, wie bei den Fließkommazahlen, besteht bei String-Eigenschaften. Soll ein String sich durch den Sprachwechsel ändern, ist dies nicht ohne weiteres möglich. Dazu wird ebenfalls ein Konverter benötigt.

Die Aufgabe des StringToResourcesConverter ist es, mittels einer String-Eigenschaft die passende Ressource aus der aktuell verwendeten Sprachdatei zu suchen und diese statt der ursprünglichen String-Eigenschaft anzugeben. Findet der Konverter die entsprechende Ressource nicht, gibt er die ursprüngliche Zeichenkette zurück. In Listing 4.24 wird die Konvertierungsmethode gezeigt. Die übergebenen Kulturinformationen (culture) werden ignoriert, ebenso werden keine weiteren Parameter überreicht. Dabei ist eine Rückkonvertiertung nicht nötig und damit auch nicht implementiert.

Auch in diesem Fall muss für die String-Eigenschaft nach einem Umschalten der Sprache das PropertyChanged-Ereignis ausgelöst werden.

```
public object Convert(object value, Type targetType, object parameter, System.
    Globalization.CultureInfo culture)
{
  String stringValue = value.ToString();

  if (!String.IsNullOrWhiteSpace(stringValue))
  {
    if (Application.Current.Resources.Contains(stringValue))
    {
      return Application.Current.FindResource(stringValue) as String;
    }
    else
    {
      return resourceStr;
    }
  }
  return String.Empty;
}
```

Listing 4.24: Konvertierung eines Strings in eine Ressource

Der Konverter wird auch hier im XAML-Code verwendet, wie in Listing 4.25 zu sehen ist.

```
<Label Content={Binding Name, Converter={StaticResource StringToStaticResourceConverter
    }} .../>
```

Listing 4.25: Ausschnitt aus der ParameterView

5 Zusammenfassung

Alle gesteckten Ziele wurden erreicht. Die Messdaten werden während der Messung bereits visualisiert, danach erfolgt das automatische Speichern zur Protokollierung. Die Benutzerverwaltung und das Rechtesystem funktionieren, wie vom Kunden gewünscht. Die Passwörter zum Absichern der Administrator- und Service-Konten können mit einem externen Werkzeug in der Konfigurationsdatei gesetzt werden. Dank des XML-Formats lassen sich die Konfigurationsdateien und Prüfprofile leicht auf mehrere Rechner verteilen, wodurch die Installationszeit entsprechend gering ausfällt und der Austausch besonders schnell vonstatten geht. Im Laufe der Arbeit wurden die Anforderungen im Rahmen der Praxistauglichkeit mehrfach geändert beziehungsweise erweitert. Zum Beispiel war zu Beginn keine Parametrierung während der Kalibrierung geplant. Dabei wird der Prüfhardware bestimmte Parameter übergeben, ohne dass diese zuvor anhalten muss.

Eine Vorabversion der Software wurde bereits an den Kunden ausgeliefert. Er verwendet diese bereits bei offiziellen Aufträgen parallel zur vorherigen LimaTest-Software, um die Ergebnisse zu vergleichen und damit die Prüfer die neue Software kennenlernen. Zwischenzeitlich gibt es vom Kunden die ersten Rückmeldungen, welche bisher ausschließlich positiv waren und, motiviert von der moderneren Umsetzung, Vorschläge bezüglich weiterer vorstellbarer Verbesserungen und Erweiterungen an der Software zum Inhalt hatten. Zum Beispiel war in der Vorabversion die Anzeige der Profilverwaltung und die des aktuellen Parametersatzes in zwei separaten Expandern. Die Anwendung zeigte jedoch, dass immer beides zusammen verwendet wird. In der aktuellen LimaTest-Software wurde dieser Umstand bereinigt und der Kunde ist mit der Anpassung überaus zufrieden. Das Rechtesystem kommt beim Kunden auch sehr gut an, da die Prüfer nicht mehr versehentlich Parameter verändern können. Dadurch müssen die Parametersätze nicht mehr per Fernwartung zurückgesetzt werden, wodurch sich die Zahl der Serviceanfragen stark reduziert haben. Die Prüfer bekommen zusätzlich üblicherweise nur die freigeschalteten Parameter angezeigt und können über einen Button am unterem linken Rand eine Ansicht des kompletten Parametersatzes aktivieren. Durch ein visuelles Feedback kann der Anwender sehen, welchen Parameter er verändern darf und welche nicht. Wenn er mit dem Mauszeiger über einem freigeschalteten Parameter ist, dann wird dieser in einem leichten grün angezeigt, ein gesperrter Parameter dagegen in grau.

Durch die positiven Rückmeldungen des Kunden wird die neue Software auch innerhalb des Instituts bekannter. Wenn das Interesse weiter steigt, wäre eine Erweiterung der Zuständigkeiten der Software vorstellbar. Die Struktur der neue LimaTest-Software ermöglicht es, weitere EMUS-Prüfsysteme zu integrieren. Da sich die Parameter zwischen den Systemen nicht unterscheiden, sondern grundsätzlich nur in ihren Werten, wäre eine Integration sogar in relativ kurzer Zeit möglich. Durch die Aufteilung in verschiedene Prüfansichten wäre es sogar möglich, komplexere Arten der Darstellung von Messdaten zu wählen, zum Beispiel als dreidimensionales Objekt.

Die dynamische Sprachumstellung demonstriert außerdem eine große Stärke der neuen Software. In den bisherigen Entwicklungen war eine Sprachumstellung nur schwer möglich beziehungsweise wurde für jede zu implementierende Sprache eine neue Version

mit der entsprechenden Sprache erstellt. In der neuen Software muss lediglich eine neue Sprachdatei erstellt werden, welche zur Laufzeit automatisch erkannt und verwendbar gemacht wird.

Nach dem Abschluss dieser Arbeit steht der Auslieferung der Software nichts mehr im Wege. Die Notebooks, die die Prüfer später im Außeneinsatz verwenden werden, stehen bereit. Es folgen noch kleinere Anpassungen des Aussehens verschiedener WPF-Elemente, die durch die Bildschirmgröße der Notebooks nicht optimal dargestellt werden.

Literatur

[Gam+95] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. 1st ed. Reprint. Amsterdam: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[Izf] *How To Use Visualization - Internes Dokument zur Verwendung der IZFPVisualizationLib*.

[Kal10] Jochen Kalmbach. *Sprachumschaltung und WPF*. [Online; Stand 30. September 2016]. 2010. URL: http://blog.kalmbach-software.de/de/2010/10/29/sprachumschaltung-und-wpf/.

[Küh12] Andread Kühnel. *Visual C# 2012*. 6., aktualisierte und erweiterte Auflage 2012. http://openbook.rheinwerk-verlag.de/visual_csharp_2012/index.html. Rheinwerkalle 4, 53227 Bonn: Galileo Press, Bonn, 2012. ISBN: 978-3-8362-1997-6.

[Mau15] Manual Mauky. *MVVM mit JavaFX*. [Online; Stand 30. August 2016]. 2015. URL: https://www.informatik-aktuell.de/entwicklung/programmiersprachen/mvvm-mit-javafx.html.

[Mic] Microsoft. *C#-Programmierhandbuch*. [Online; Stand 28. September 2016]. URL: https://msdn.microsoft.com/de-de/library/67ef8sbd.aspx.

[Osw16] Jan Oswald. „Entwicklung einer erweiterbaren C++-Klassenbibliothek zur Ansteuerung eines Ultraschallprüfsystems“. Bachelorthesis. Fraunhofer IZFP, 2016.

[The12] Thomas Theis. *Einstieg in WPF : Grundlagen und Praxis*. 1. Auflage 2012. Rheinwerkalle 4, 53227 Bonn: Galileo Press, Bonn, 2012. ISBN: 978-3-8362-1776-7.

[Vtk] *VTK 5.6.1 Documentation*. [Online; Stand 30. August 2016]. 26.03.2010. URL: http://www.vtk.org/doc/release/5.6/html/index.html.

[Weg13] Jörg Wegener. *WPF 4.5 und XAML : Grafische Benutzeroberflächen für Windows inkl. Entwicklung von Windows Store Apps*. Carl Hanser Verlag München, 2013. ISBN: 978-3-446-43467-7.

Abbildungsverzeichnis

Listings

Anhang

A Klassendiagramme

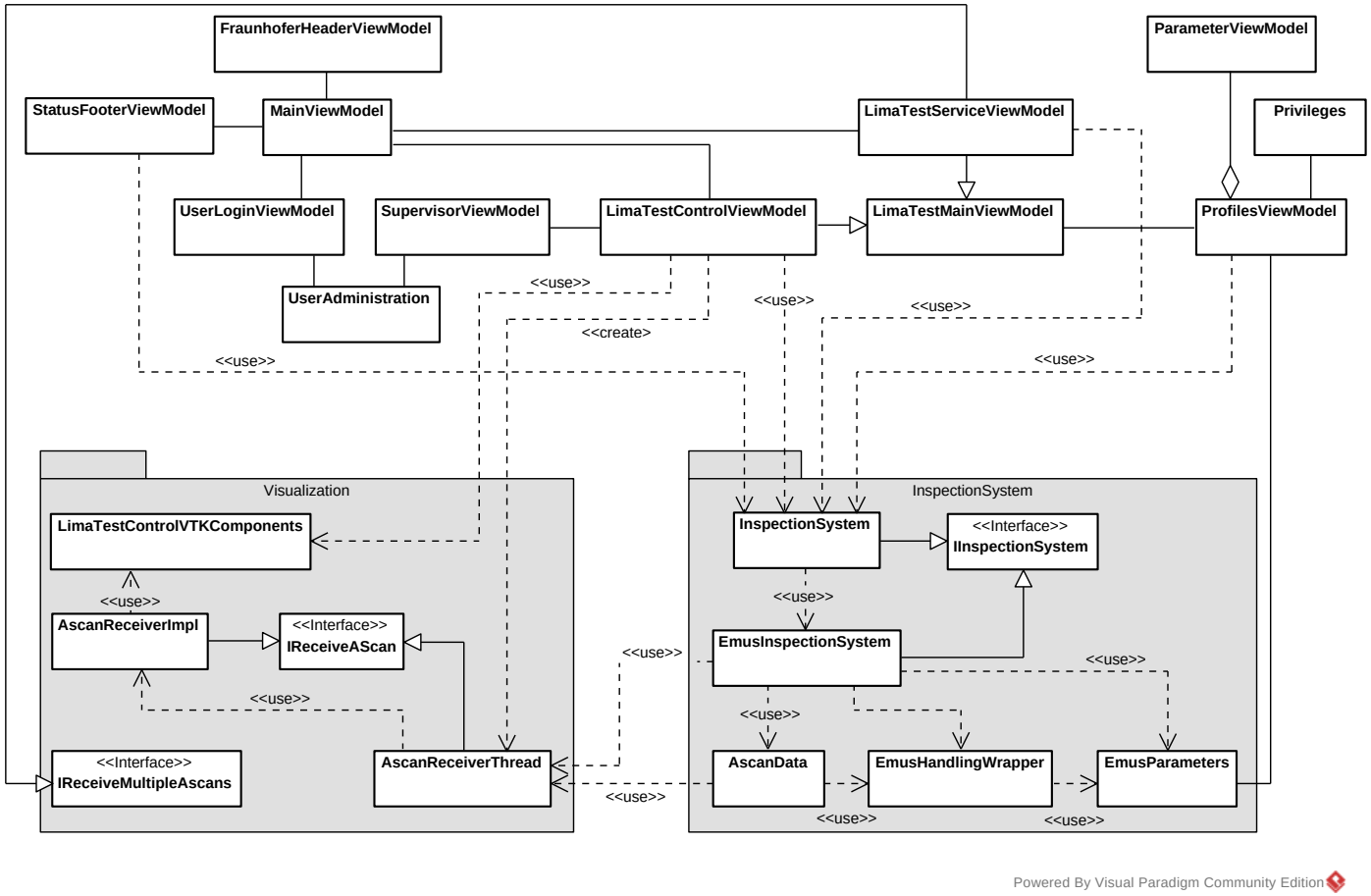

Abbildung A.1: Übersicht

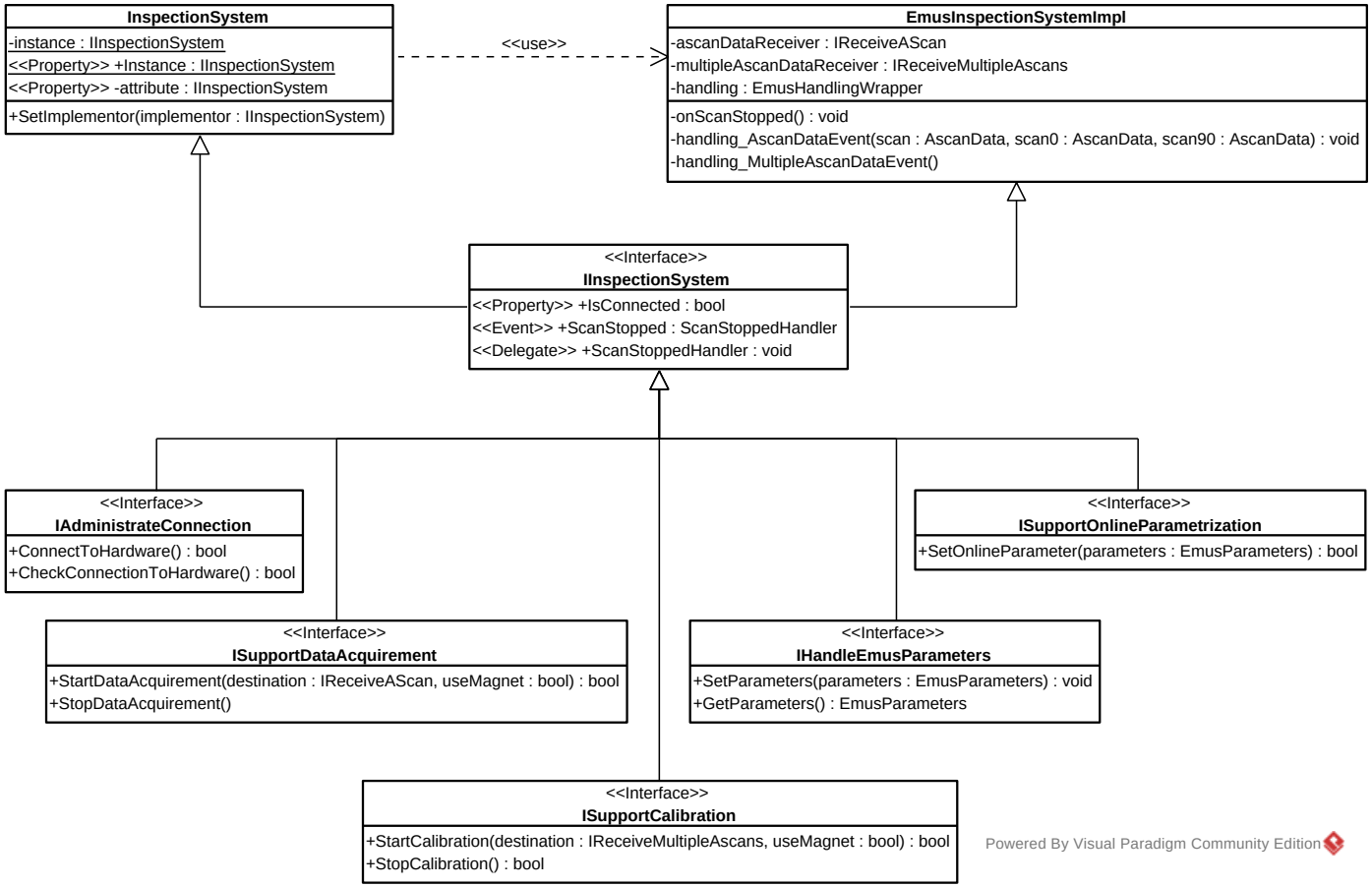

Abbildung A.2: Klassendiagramm InspectionSystem