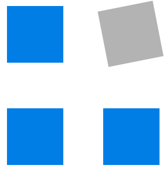


HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

the i-engineers

AttributeProfile Designer

Semesterarbeit

Herbstsemester 2009

Autoren:	**Christopher Guntli**
	Urs Wegmann
Betreuer:	**Hansjörg Huser**
Gegenleser:	**Stefan Keller**
Projektpartner:	**the i-engineers**

Erklärung der Selbstständigkeit

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.

Rapperswil, den 18. Dezember 2009 Christopher Guntli

Rapperswil, den 18. Dezember 2009 Urs Wegmann

Inhaltsverzeichnis

Abstract ix

Management Summary xi
1 Ausgangslage . . . xi
2 Vorgehen, Technologien . . . xi
3 Ergebnisse . . . xii
4 Ausblick . . . xii

1 Technischer Bericht 1
1.1 Einleitung . . . 1
1.1.1 Analyse der Masken-Definitionen . . . 1
1.1.2 Momentaner Workflow . . . 2
1.1.3 Ziel der Semesterarbeit . . . 3
1.2 Ergebnisse . . . 3
1.2.1 Workflow . . . 3
1.2.2 Dateiformat . . . 5
1.2.3 Applikation . . . 6
1.3 Schlussfolgerungen . . . 6
1.3.1 Dateiformat . . . 6
1.3.2 Applikation . . . 7

2 Projektplan 11
2.1 Projektübersicht . . . 11
2.1.1 Ausgangslage . . . 11
2.1.2 Ziel . . . 11
2.1.3 Rahmenbedingungen und Aufteilung . . . 12
2.2 Meilensteine . . . 12
2.3 Meetings . . . 12
2.4 Risiko Management . . . 13
2.4.1 Fehler im Domainmodel . . . 13
2.4.2 Probleme mit WPF . . . 13
2.4.3 Kein Modelling Framework . . . 14
2.4.4 Weitere Risiken . . . 14
2.5 Qualitätsmassnamen . . . 15
2.5.1 Versionsverwaltung . . . 15

2.5.2 Dokumentation . . . 15
2.5.3 Sitzungsprotokolle . . . 15
2.5.4 Zeitplan . . . 16
2.5.5 Style Guides . . . 16
2.5.6 Build Umgebung . . . 16
2.5.7 Dokument Review . . . 16
2.5.8 Unit Tests . . . 16
2.5.9 System Tests . . . 16

3 Analyse 17
3.1 Domain Model . . . 17
3.1.1 AttributeProfile . . . 18
3.1.2 ShowType . . . 18
3.1.3 Attribute . . . 18
3.1.4 Property . . . 20
3.2 Paper Prototype . . . 21
3.2.1 Items Panel . . . 22
3.2.2 ObjectTree Panel . . . 22
3.2.3 Property Panel . . . 22
3.2.4 ShowTypes Panel . . . 23
3.2.5 ShowType View . . . 23

4 Design 25
4.1 Features . . . 25
4.1.1 Core Features . . . 25
4.1.2 Zusatz Features . . . 25
4.2 Frameworks und Libraries . . . 26
4.2.1 AvalonDock . . . 26
4.2.2 Enterprise Library 4.1 . . . 26
4.3 Dateiformat . . . 26
4.3.1 Evaluation . . . 26
4.3.2 Schema . . . 27
4.3.3 Beispieldokument . . . 29
4.3.4 Beschreibung . . . 32
4.4 Architektur . . . 34
4.4.1 Ziele . . . 34
4.4.2 Evaluation . . . 34
4.5 Deployment . . . 36

5 Implementation 37
5.1 ProfileDesigner . . . 37
5.1.1 ProfileDesigner Namespace . . . 37

5.1.2 UnitiyContainer 37
5.2 ProfileDesigner.View 38
5.2.1 AttributeGrid 38
5.2.2 AttributeTemplatesPanel 41
5.2.3 AttributeView 41
5.2.4 EmptyAttributeGridCell 41
5.2.5 PropertyPanel 42
5.2.6 SelectionRectangle 42
5.2.7 Shell 42
5.2.8 ShowTypePanel 43
5.2.9 ShowTypeTemplatesPanel 43
5.2.10 ShowTypeView 43
5.2.11 Commands 45
5.2.12 Converter 45
5.3 ProfileDesigner.ViewModel 47
5.3.1 AttributeProfileViewModel 47
5.3.2 AttributeTemplateViewModel 47
5.3.3 AttributeViewModel 47
5.3.4 ItemViewModel 48
5.3.5 SelectionViewModel 49
5.3.6 ShellViewModel 49
5.3.7 ShowTypeTemplateViewModel 49
5.3.8 ShowTypeViewModel 49
5.4 ProfileDesigner.Bll 50
5.4.1 Implementationskonzept 50
5.4.2 Interfaces 52
5.4.3 Persitence Interfaces 52
5.4.4 Business Objekte 53
5.4.5 Templates Handling 54
5.5 ProfileDesigner.Common 56
5.5.1 CollectionUtils 56
5.5.2 ExceptionHelper 56
5.5.3 IModule 57
5.5.4 Bootstrapper 57

6 Projektabschluss 59
6.1 Risiken 59
6.1.1 Fehler im Domainmodel 59
6.1.2 Probleme mit WPF 59
6.1.3 Kein Modelling Framework 59
6.1.4 Fazit 60

6.2 Tests 60
6.2.1 Unit Tests 60
6.2.2 System Tests 60
6.3 Zeit Management 61
6.3.1 Zeitaufwand pro Teammitglied 61
6.3.2 Zeitaufwand pro Disziplin 61
6.4 Probleme 62
6.4.1 AvalonDock 62
6.4.2 MVVM 62
6.4.3 Mouse Events 63

7 Benutzerdokumentation 65
7.1 Installation 65
7.2 Ordnerstruktur 65
7.2.1 Sicherheitsrichtlinien in Windows Vista und 7 65
7.3 AttributeProfile 66
7.3.1 Erstellen 66
7.3.2 Bearbeiten 66
7.4 ShowType 67
7.4.1 Erstellen 67
7.4.2 Bearbeiten 67
7.5 Attribute 68
7.5.1 Erstellen 68
7.5.2 Bearbeiten 69
7.6 Templates 69
7.7 Copy & Paste 69
7.8 Fensteranordnung 70
7.8.1 Persistenz der Anordnung 70

8 Persönliche Berichte 71
8.1 Urs Wegmann 71
8.2 Christopher Guntli 72

Literaturverzeichnis 73

Abbildungsverzeichnis 75

Tabellenverzeichnis 77

Listings 79

Glossar 81

Abstract

Die Firma the i-engineers mit Sitz in Aarau ist ein Hersteller eines CMS mit Kunden im Bereich Gesundheitswesen, Industrie und der öffentlichen Verwaltung. Das von the i-engineers entwickelte Informationsverwaltungssystem *i-engine* verfügt sowohl über einen Client, der als .Net standalone Applikation realisiert ist, als auch über einen Client in Adobe Flex. Die Definitionen für die Dialoge werden in der Datenbank hinterlegt, woraus der jeweilige Client dynamisch zur Laufzeit die entsprechenden Dialoge erstellt. Im Moment werden solche Dialog-Definitionen mittels SQL[1]-Scripten in die Datenbank eingefügt und geändert.

Zur kompakten Beschreibung dieser Dialog-Definitionen wurde von uns ein XML[2] Dateiformat entwickelt, welches durch ein Schema definiert ist. So können bestehende Dokumente gegen dieses Schema geprüft werden um deren Gültigkeit zu testen.

Der in dieser Arbeit entwickelte *AttributeProfile Designer* bietet die Möglichkeit, diese Dialog-Definitionen komfortabel zu erstellen und zu bearbeiten. Die Dialog-Definitionen werden im zuvor definierten XML Dateiformat gespeichert. Zur Verbesserung der Bedienbarkeit sind Funktionen wie das Verschieben und Ändern der Grösse von Objekten per Drag & Drop möglich. Zusätzlich können Objekte als Templates exportiert und später wieder verwendet werden. Dies erspart dem Benutzer viele repetitive Tätigkeiten. Durch die Templates kann die Applikation flexibel an sich ändernde Bedingungen wie etwa Änderungen der *i-engine* angepasst werden. Weiter wurde Wert auf eine einfache Erweiterbarkeit der Applikation gelegt, damit später weitere Funktionen einfach hinzugefügt werden können.

1 Structured Query Language
2 Extensible Markup Language

Management Summary

1 Ausgangslage

Die Firma the i-engineers entwickelt das CMS[1] *i-engine* für das Informationsmanagement im Bereich Gesundheitswesen, Industrie und öffentliche Verwaltung. Für die Informationserfassung werden Masken verwendet, welche momentan noch von Hand per SQL in die Datenbank eingefügt und bearbeitet werden. Dies ist für die betroffenen Personen unkomfortabel und macht überdies den Ablauf fehleranfällig. Das Ziel dieser Semesterarbeit ist ein Programm zu erstellen, mit welchem diese Datenerfassungsmasken generiert und bearbeitet werden können. Ausserdem sollte ein geeignetes Format gefunden werden, um diese Masken zu speichern.

2 Vorgehen, Technologien

Der erste Teil der Arbeit bestand darin, ein geeignetes Format zu finden um die Maskendefinitionen abzuspeichern. Dazu gab es zwei Möglichkeiten: Einerseits die Verwendung eines bestehenden Formats wie XForms oder andererseits die Erstellung eines eigenen Formats. Nach der Analyse von bestehenden Formaten und dem Aufbau der Masken entschieden wir uns dafür, ein eigenes Format auf Basis von XML zu entwickeln, weil bestehende Formate entweder nicht alles benötigte unterstützen oder aber zu viel abdecken.

In einem zweiten Teil der Arbeit wurde ein Editor entwickelt, mit welchem die Masken erstellt und bearbeitet werden können. Der AttributeProfile Designer verwendet für das Speichern und Laden von Profilen sowie Templates das im ersten Teil der Arbeit erstellte XML Format.

Im Vordergrund bei der Entwicklung der Software stand vor allem eine intuitive Bedienbarkeit. Diese wurde erreicht, indem wir erheblichen Zeitaufwand für die Gestaltung des GUI[2] betrieben und Technologien wie Drag & Drop verwendeten. Aufgrund von Anforderungen des Kunden wurde die Applikation mit WPF[3] und .Net 3.5 realisiert.

1 Content Management System
2 Graphical User Interface
3 Windows Presentation Fondation

3 Ergebnisse

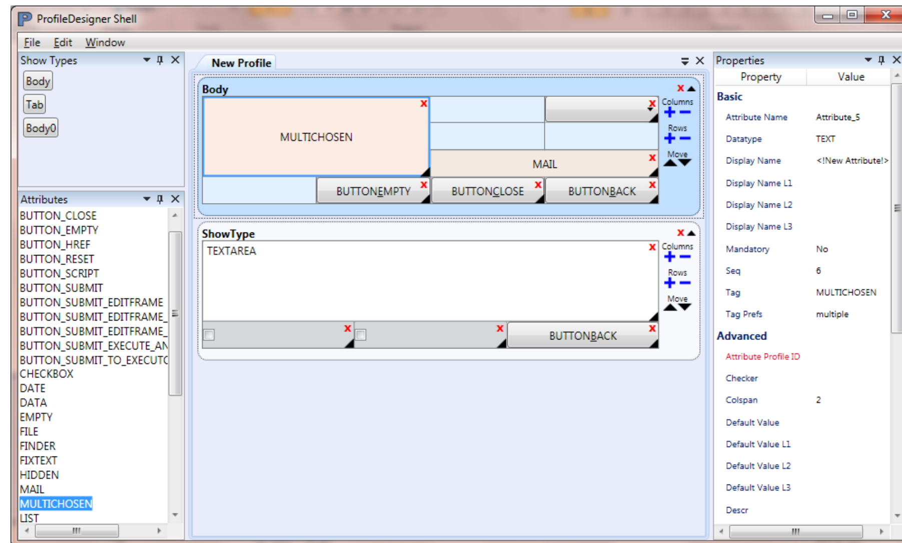


Abbildung 1: Die fertige Applikation mit einem neu erstellten AttributeProfile

Entstanden ist eine Applikation (Abbildung 1), welche dem Betreuer des Endkunden hilft, neue Maskendefinitionen zu erstellen oder bestehende zu bearbeiten. Diese werden in einem eigens definierten XML Dateiformat abgespeichert oder daraus geladen.

Das Programm ist in vielen Bereichen konfigurierbar:

- Für die Erzeugung von Masken und den darin enthaltenen Objekten können Templates verwendet werden.
- Die Anordnung der einzelnen Kontrollelemente kann frei definiert werden und wird jeweils beim Schliessen des Programms gespeichert und beim erneuten Starten wieder geladen.

4 Ausblick

Da das Programm so gestaltet wurde, dass es mit wenig Aufwand um neue Funktionen erweitert werden kann, ist es für TIE[1] jederzeit möglich, neue Features hinzuzufügen oder bestehende zu ersetzen. Diese kann beispielsweise nötig sein, wenn das CMS i-engine grundlegend geändert wird.

1 The i-engineers

Kapitel 1

Technischer Bericht

1.1 Einleitung

Die Firma TIE mit Sitz in Aarau ist ein Hersteller eines CMS mit Kunden im Bereich Gesundheitswesen, Industrie und der öffentlichen Verwaltung. Das von TIE entwickelte Informationsverwaltungssystem *i-engine* verfügt sowohl über einen Client der als .Net Standalone Applikation realisiert ist als auch über einen Client in Adobe Flex. Die Definitionen für die Eingabemasken zur Datenerfassung werden in der Datenbank hinterlegt, woraus der jeweilige Client dynamisch zur Laufzeit die entsprechenden Eingabemasken (Formulare), erstellt und dem Benutzer präsentiert.

1.1.1 Analyse der Masken-Definitionen

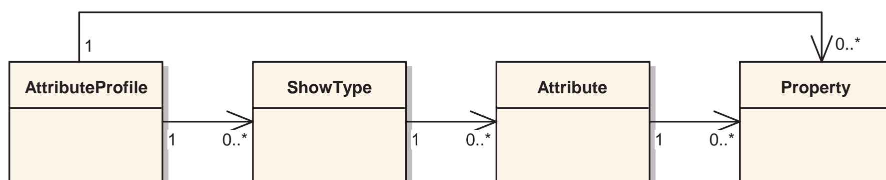


Abbildung 1.1: Objekte zum Beschreiben der Masken-Definitionen

AttributeProfile

Das AttributeProfile enthält mehrere zusammengehörige ShowTypes. Es hat ausserdem Properties, welche dessen Eigenschaften beschreiben. Hierzu gehört beispielsweise die *AttributeProfileId*.

ShowType

Die Eingabemasken werden aus mehreren ShowTypes zusammengesetzt wobei jeder ShowType einen Bereich der Maske beschreibt. Welche ShowTypes zu welcher Maske gehören wird über die *i-engine* definiert. Die ShowTypes werden jeweils durch einen eindeutigen Namen identifiziert.

Attribute

Die ShowTypes enthalten ihrerseits wieder Attribute, welche die Kontrollelemente wie etwa Buttons oder Textboxen repräsentieren. Die Eigenschaften der Attribute werden auch durch Properties beschrieben. Wichtig ist hier insbesondere das Property *Tag*, welches beschreibt welche Art (Textbox, Button,...) eines Attribute gemaint ist. Ausserdem gibt es Properties, welche die Platzierung und Grösse des Attributes im Grid, ein dynamisch skalierender Raster, des ShowType definieren.

Property

Properties beschreiben die Eigenschaften von AttributeProfilen und Attributen. Es gibt grundsätzlich drei verschiedene Properties:

StringProperties enthalten einen String mit definierter Länge

IntegerProperties enthalten eine Ganzzahl

FloatProperties enthalten eine Fliesskommazahl mit bestimmter Genauigkeit

Ausserdem muss definiert sein, ob ein Property einen Wert enthalten muss oder nicht.

1.1.2 Momentaner Workflow

Im bisherigen Workflow werden AttributeProfile von TIE-Mitarbeitern aufgrund von Kundenwünschen erstellt, indem sie direkt Manipulationen auf der Datenbank durchführen (Abbildung 1.2). Für Anpassungen muss der Kunde die so erstellten Masken ansehen und entscheiden, ob diese seinen Vorstellungen entsprechen.

Ablauf beim Erstellen einer Maske

1. Der Kunde beschreibt in Form von Text oder einer Skizze, wie die neue Maske aussehen soll.
2. Die vom Kunden verfasste Beschreibung wird von einem TIE-Mitarbeiter ausgewertet, welcher danach ein Script mit SQL Befehlen erstellt und damit die Maske in die Datenbank einträgt. Dabei muss er darauf achten, die Elemente der Maske korrekt miteinander zu verknüpfen.

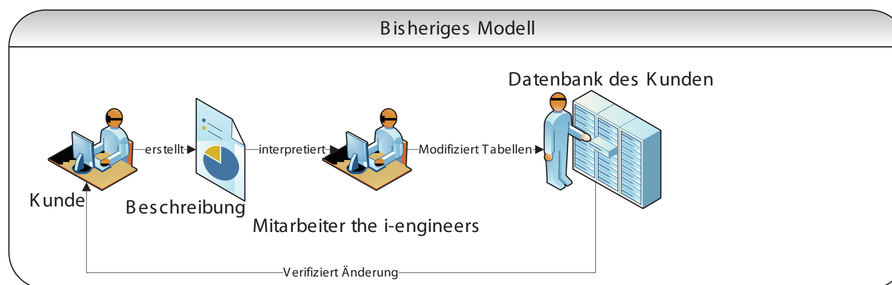


Abbildung 1.2: Worflow für die Erstellung einer neuen Maske (bisher)

3. Der Kunde überprüft ob die Umsetzung seinen Wünschen entspricht.
4. Falls noch Änderungen nötig sind führt der TIE Mitarbeiter diese durch. Danach kann sich der Kunde die Maske wieder ansehen, usw...

Problem

Da der TIE Mitarbeiter die neuen Masken und Änderungen an bestehen direkt in die Datenbank einträgt, muss er dazu die entsprechenden Rechte besitzen. Falls ihm dabei ein Fehler unterläuft kann es also passieren, dass er die Datenbank beschädigt. Ausserdem ist die Erstellung der dazu benötigten SQL Scripte aufwändig und damit zeitaufwändig.

1.1.3 Ziel der Semesterarbeit

Das Ziel dieser Semesterarbeit besteht darin, diesen Workflow effizienter und benutzerfreundlicher zu gestalten. Um dies zu realisieren soll eine Applikation (*AttributeProfile Designer*) erstellt werden, mit welcher die Dialog-Definitionen erstellt und bestehende bearbeitet werden können. In dieser Arbeit ist jedoch nicht vorgesehen, direkt Änderung an der Datenbank vorzunehmen. Stattdessen soll ein Format evaluiert werden, mit welchem diese Dialog-Definitionen abgespeichert werden können. Jedoch ist vorgesehen, die Anbindung an die Datenbank später noch zu realisieren und daher muss die Applikation entsprechend erweiterbar sein.

1.2 Ergebnisse

1.2.1 Workflow

Damit der Mitarbeiter nicht mehr von Hand und weniger oft auf die Datenbank zugreifen muss, kann er die Masken-Definitionen mit Hilfe des *AttributeProfile Designers* schnell und bequem erstellen und bearbeiten. Ausserdem kann er diese dem Kunden präsentieren, bevor sie in der Datenbank abgelegt werden (siehe Abbildung 1.3).

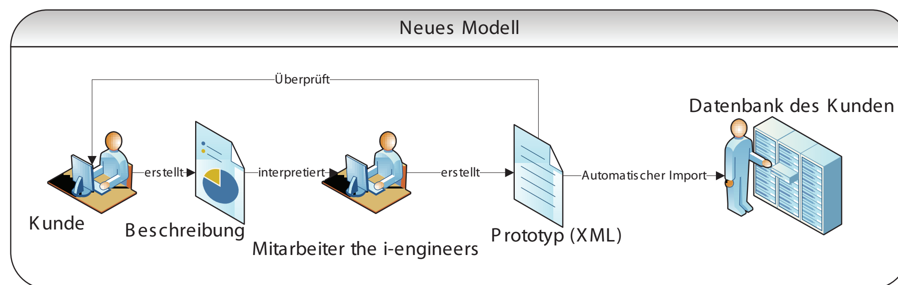


Abbildung 1.3: Angestrebter Workflow

Ablauf beim Erstellen einer Maske

1. Der Kunde beschreibt in Form von Text oder einer Skizze die gewünschten Masken.
2. Ein Mitarbeiter von TIE analysiert die vom Kunden erstellte Beschreibung und erstellt mit Hilfe des *AttributeProfile Designers* einen ersten Entwurf des AttributeProfiles.
3. Der Kunde überprüft die mit dem Designer erstellte Maske und gibt entsprechendes Feedback.
4. Der TIE Mitarbeiter führt die entsprechenden Änderungen durch bis ein Konsens mit dem Kunden gefunden werden kann.
5. Sobald das AttributeProfile fertig ist wird es automatisiert in die Datenbank eingetragen.

Vorteile

- Schnelleres Erstellen und Bearbeiten des Aussehens einer Maske durch grafische Unterstützung.
- Mehrfach verwendete Subelemente in einer Maske dank Templates. Dies Erspart dem Sachbearbeiter repetitive Tätigkeiten.
- WYSIWYG[1], der Kunde sieht sofort wie die Maske aussehen wird und kann so schnell Feedback geben.
- Masken werden erst in die Datenbank eingetragen, wenn sie fertig sind. So sind insgesamt weniger Änderungen an der Datenbank nötig.

1 What You See Is What You Get

- Der Sachbearbeiter muss nicht direkt auf die Datenbank zugreifen. Daher ist der Ablauf weniger Fehleranfällig.

1.2.2 Dateiformat

Die Evaluierung eines geeigneten Dateiformates war ein wesentlicher Teil der Arbeit. Grundsätzlich gab es dabei zwei Möglichkeiten:

Bestehendes Format verwenden

Die Verwendung eines bestehenden Dateiformates für die Abspeicherung von GUIs bietet den Vorteil, dass möglicherweise auf bestehendes Knowhow zurückgegriffen werden kann. Allerdings sind GUIs sehr vielfältig und es ist praktisch nicht möglich, alle denkbaren Variationen mit einem einzigen Dateiformat abzuspeichern. Dies bedeutet, dass ein erheblicher Overhead nötig wäre, um ein AttributeProfile in einem bestehenden Dateiformat abzuspeichern.

Zur Diskussion stand das Format XForms welches für die Beschreibung von Datenerfassungsmasken auf verschiedenen Geräten vorgesehen ist. Dies ist jedoch ein sehr umfangreicher Standart woraus wir uns dann die benötigten Features hätten picken müssen. Ausserdem sind nicht alle benötigten Features zur Beschreibung der AttributeProfile abgedeckt. Ein weiteres Problem besteht darin, dass keine geeignete Bibliothek für die .Net Plattform exisistiert was eine eigene Implementation nötig gemacht hätte.

Eigenes Format auf Basis von XML entwickeln

Die Alternative war, ein eigenes Format auf Basis von XML entwickeln. Der grösste Vorteil besteht darin, dass ein eigenes Format perfekt auf die konkreten Bedürfnisse zugeschnitten werden kann. Ausserdem besteht bereits eine Umfangreiche Unterstützung für XML auf der .Net Plattform, was einiges an Implementationsarbeit erspart. Ausserdem ist XML weit verbreitet und etabliert, wodurch von *AttributeProfile Designer* erstellte Profile beispielsweise auch mit einem anderen XML Editor bearbeitet werden können.

Entscheidung

Wir entschieden uns dazu, ein eigenes Format auf Basis von XML zu entwickeln. Dies insbesondere deshalb, weil wir keinen klaren Vorteil in der Verwendung eines bereits bestehenden Formates erkennen konnten. Wie oben beschrieben hätte dies sogar einen Mehraufwand in der Implementation bedeutet, da zuerst die Unterstützung für XForms hätte realisiert werden müssen. XML hingegen wird bereits sehr gut unterstützt und ausserdem sind sich viele Menschen bereits gewohnt, XML Dateien zu lesen.

1.2.3 Applikation

Bedienbarkeit

Im Vordergrund bei der Erstellung der Applikation stand eine gute Bedienbarkeit da die Sachbearbeiter später viel Zeit damit verbringen werden. Um dies zu erreichen, wurde sehr früh ein benutzbarer Prototyp erstellt wodurch die Betreuer und die Mitarbeiter von TIE sich ein Bild davon machen konnten, wie die Applikation zu bedienen sein wird. So bekamen wir auch wertvolle Feedbacks welche wiederum in die weitere Entwicklung der Applikation einflossen.

Wichtig für eine gute Bedienbarkeit ist insbesondere, dass der Benutzer den *AttributeProfile Designer* so bedienen kann, wie er es sich von ähnlichen Applikationen her gewohnt ist. In diesem Fall wurden vorallem bereits bestehende GUI- und Webdesign Applikationen als Vorbilder herangezogen. Nach einer genauen Analyse wurde klar, dass vor allem Drag & Drop beim Erstellen eines GUI eine zentrale Rolle darstellt. Ausserdem sollen Tastenkombinationen unterstützt werden, welchen geübten Benutzern den Zugriff per Maus auf das Menü erspart.

Konfigurierbarkeit

Wichtig für ein effizientes Arbeiten mit der Applikation ist ausserdem, dass diese flexibel an wechselnde Umbegungsbedingungen angepasst werden kann. Gemeint sind hier insbesondere Änderungen an der *i-engine*. Denkbar wäre auch, dass beispielsweise für verschiedene Kunden verschiedene Einstellungen nötig sind. Folgende Eigenschaften sind konfigurierbar:

- Grösse und Anordnung der Panels der Bedienoberfläche
- Voreingestellte Anzahl der Zeilen und Spalten im AttributeGrid
- Properties, welche in allen AttributeProfiles bzw. Attributes vorkommen müssen
- Attributes und ShowTypes, welche als Templates zur Verfügung stehen
- Defaultwerte von Properties

1.3 Schlussfolgerungen

1.3.1 Dateiformat

Die Erstellung eines eigenen Formates auf Basis von XML hat sich bewährt. Wir konnten Aufgrund des in der Analyse erstellten Domainmodels das XML Schema weitgehend

automatisch generieren und auch die Klassen zum Lesen und Schreiben der Attribute-Profile konnten dank entsprechenden Werkzeugen von Microsoft automatisch generiert werden. So konnten wir mehr Zeit in die Entwicklung eines benutzerfreundlichen GUIs und wertvoller Features investieren anstatt uns mit der Implementation des XForms Dateiformates zu beschäftigen.

Ein weiterer Vorteil bei der Verwendung von XML besteht darin, dass der Benutzer das Format auch ohne Werkzeuge lesen und verstehen kann. Dadurch ist es einfacher möglich, die Konfigurationsdateien und Templates nach eigenen Vorstellungen zu verändern.

1.3.2 Applikation

Das GUI der Applikation (Abbildung 1.4) ist Modular aufgebaut und so perfekt an die Bedürfnisse des Benutzers anpassbar. Um die Oberfläche anzupassen muss der Kunde lediglich die Panels an die gewünschte Position verschieben und die Grösse entsprechend anpassen.

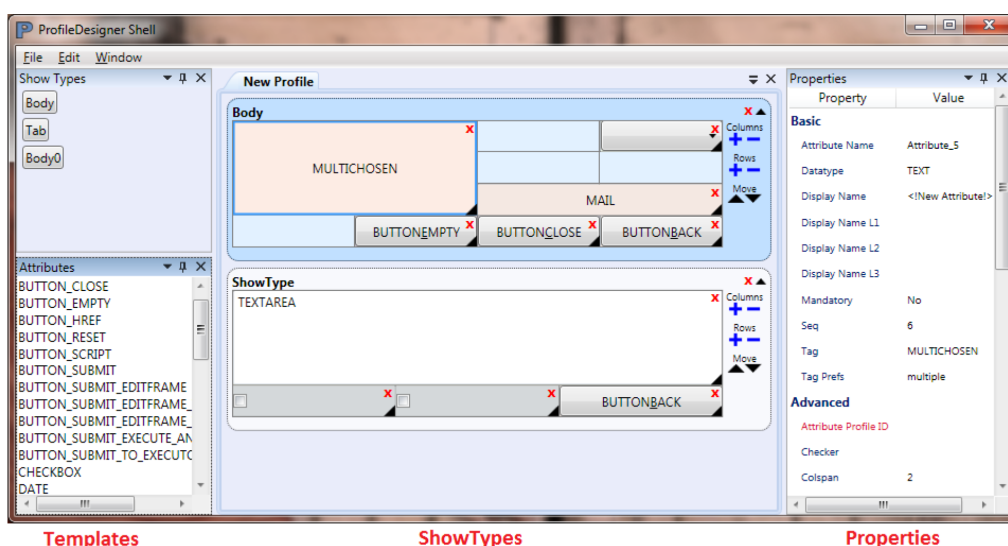


Abbildung 1.4: Screenshot der Software

Bedienbarkeit

Durch die einfache grafische Oberfläche der Software kann der Benutzer in wenigen Schritten ein AttributeProfile erstellen und mit den gewünschten Objekten füllen (WYSIWYG). Auch die Bearbeitung der Properties ist intuitiv und einfach. Die Möglichkeit, Objekte als Templates abzuspeichern und so wieder zu verwenden ermöglicht dem Kunden das einfache Erstellen von mehreren ähnlichen Objekten. Eine weitere Massname,

dem Benutzer das Leben zu vereinfachen besteht in der Verfügbarkeit von Copy & Paste Operationen. All dies führt zu einer guten Bedienbarkeit der Applikation, was erste Usability Tests auch bestätigen.

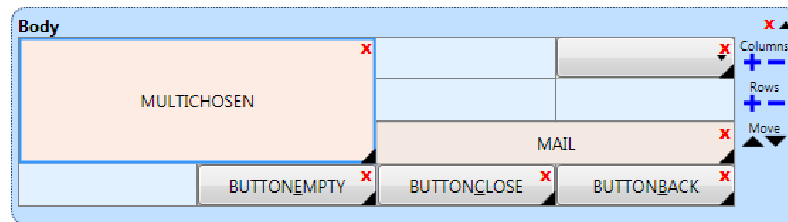


Abbildung 1.5: Detailansicht eines ShowTypes

In Abbildung 1.5 ist die Detailansicht eines ShowTypes zu sehen. Wichtig ist hier insbesondere das AttributeGrid, in welchem die im ShowType enthaltenen Attribute in einem Gitter an der entsprechenden Position dargestellt werden. Die Erstellung und Bearbeitung eines Attributes ist vergleichbar mit der Bearbeitung von Zellen in Excel. Attribute können auf die gewünschte Grösse aufgezogen, in der Grösse verändert oder in der Position verschoben werden. Ausserdem gib es für ShowTypes Kontrollelemente um diese zu verschieben oder zu löschen.

Konfigurierbarkeit

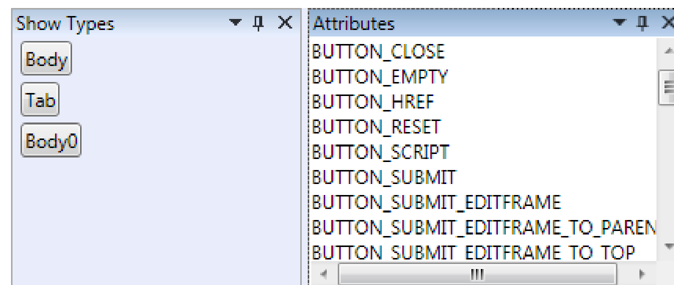


Abbildung 1.6: Detailansicht der geladenen Templats

Um den Benutzer immer wiederkehrende Tätigkeiten zu ersparen, haben wir den AttributeProfile Designer mit der Möglichkeit ausgestattet, Elemente aus Templates zu erstellen und bestehende Elemente wiederum als Templates abzuspeichern. Falls erforderlich können Templates auch direkt von Hand in einer XML Datei erstellt oder bearbeitet werden. In Abbildung 1.6 sind die beiden Panels zu sehen, in welche bestehende Templates beim Programmstart geladen werden. Links diejenigen für ShowTypes und rechts diejenigen für Attribute. Diese können dann ausgewählt und im ShowTypePanel erstellt werden.

Templates und die fertigen AttributeProfile werden im gleichen XML Format abgespeichert. Auch per Copy & Paste in die Zwischenablage kopierte Objekte werden im gleichen Format hinterlegt. Dies hat den Vorteil, dass der Benutzer, sofern er dies will, ein AttributeProfile per XML öffnen und dann beispielsweise daraus einem ShowType in der Zwischenablage ein Attribute hinzufügen kann. Es sind auch diverse andere Szenarien denkbar. Dies steigert die Flexibilität der Applikation.

Da es ziemlich redundant wäre, für jedes Objekt in den Templates alle Werte vorzugeben, wurden sogenannte *Defaults* eingeführt. Diejenigen Properties, welche in jedem AttributeProfile bzw. Attribute vorhanden sein müssen, werden in der *default.xml* Datei hinterlegt. Die Templates können dann diese Properties überschreiben oder weitere hinzufügen (Listings 1.1 und 1.2).

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Attributes>
  <Attribute>
    <IntegerProperty name="H_Pos" allowNull="false" readOnly="false"
      extended="true" />
    <IntegerProperty name="V_Pos" allowNull="false" readOnly="false"
      extended="true" />
    <StringProperty name="Tag" allowNull="false" readOnly="false" length="100"
      extended="false" />
  <Attribute>
<Attributes>
```

Listing 1.1: Mögliche default.xml Datei für die Attribute

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Attributes>
  <Attribute>
    <StringProperty name="Tag" allowNull="false" readOnly="true" length="100"
      extended="false" value="PASSWORD" />
  </Attribute>
<Attributes>
```

Listing 1.2: Attribute Template für das Password Attribute welches *default.xml* ergänzt

Anbindung an die Datenbank

Der *AttributeProfile Designer* wurde so erstellt, dass einfach neue Module für die Anbindung von Datenquellen hinzugefügt werden können. So wäre es beispielsweise möglich, eine Erweiterung hinzuzufügen, welche die Masken direkt in den SQL Server einspeist bzw. Masken aus diesem lädt und dem Benutzer zum Bearbeiten präsentiert. Eine solche Anbindung ist gemäss TIE auch in nächster Zeit geplant. Allerdings würde dies wieder vermehrten Datenbankzugriff des Personals bedeuten, was eventuell wieder weniger sicher als die entkoppelte Lösung wäre. So müsste analysiert werden, wie die Datenbank gegen ungewollte oder böswillige Manipulationen geschützt werden kann.

Features

Die Features wurden am Anfang des Projekts in zwei Gruppen aufgeteilt. Eine Gruppe waren die *Core Features*, welche die Kernfunktionalität der Applikation beschreiben und zwingend implementiert werden mussten. Daneben wurden noch *Zusatzfeatures*, welche das Arbeiten mit der Software erleichtern sollen aber für die Verwendung nicht zwingend erforderlich sind, definiert. Die zusätzlichen Features sollten wenn möglich implementiert werden, konnten aber bei Zeitdruck oder Komplikationen weggelassen werden. Wir konnten alle geplanten Core- und Zusatzfeatures implementieren:

Core Features

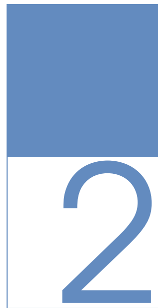
- Erstellen und Löschen von AttributeProfilen, ShowTypes und Attributes
- Bearbeiten von Properties
- Speichern und Laden von / in XML Dokumente

Zusatzfeatures

- Copy & Paste von ShowTypes und Attributes
- Erstellen und Verschieben von Elementen per Drag & Drop
- Benutzerdefinierte Anzeige von ShowTypes (einzelne ShowTypes ein- und ausblenden)
- Erstellen von ShowTypes und Attributes aus Templates welche in XML Dateien hinterlegt sind
- XML Elemente im Editor als Templates abspeichern

Verteilung der Software an den Kunden

Es ist denkbar, dass der *AttributeProfile Designer* an den Endkunden verteilt wird damit dieser selbständig Masken nach seinen Vorstellungen erstellen kann. Diese so erstellten Masken können dann an TIE übermittelt werden, wo sie geprüft und in die Datenbank eingetragen werden. Eine weitere Variante besteht möglicherweise darin, dass der Kunde seine Datenbank mit Hilfe des um einen Datenbankzugriff erweiterten *AttributeProfile Designers* selbst bearbeitet.



Kapitel 2

Projektplan

2.1 Projektübersicht

2.1.1 Ausgangslage

Die Firma TIE mit Sitz in Aarau ist ein Hersteller eines ECMS[1] mit Kunden im Bereich Spitäler, Industrie und der öffentlichen Verwaltung. Das ECMS (*i-engine*) verfügt sowohl über einen Client der als .Net Standalone Applikation realisiert ist als auch über einen Client in Adobe Flex. Die Definitionen für die Dialoge werden in der Datenbank hinterlegt und Dialoge werden dynamisch zur Laufzeit vom jeweiligen Client erzeugt. Im Moment werden solche Dialog-Definitionen per SQL-Scripts in die Datenbank eingefügt und geändert.

2.1.2 Ziel

Ziel ist die Realisierung eines Editors mit Microsoft .Net 3.5 Sp1, der es erlaubt Dialoge per Mausklick zu gestalten. Die Ausgabe des Editors soll ein Zwischenformat z.B XML sein, welches dann weiterverarbeitet und in die Datenbank importiert werden kann.

In einem weiteren Schritt muss das gewählte Zwischenformat auch aus den bestehenden Dialogen erzeugbar sein. Dieser Editor soll später auch in der Lage sein, Workflows grafisch zu gestalten und diese in einem Standardformat auszugeben oder Strukturbäume von Rollen zu bearbeiten.

1 Enterprise Content Management System

2.1.3 Rahmenbedingungen und Aufteilung

Die Semesterarbeit kann grob in zwei Teile aufgeteilt werden, welche mehr oder weniger sequentiell bearbeitet werden sollen.

1. Evaluation eines geeigneten Formats, welches einem gängigen Standard entspricht (XForms, etc. - unabhängig von .Net) und in welchem alle Anforderungen an einen Dialog der i-engine (AttributeProfile) dargestellt werden können (Textfelder, Buttons, Drop-Down Menüs, Validierung, ShowTypes etc). Dazu kann der Abschnitt zur AttributeProfile-Erstellung aus dem Systemhandbuch überflogen werden.

2. Erstellung eines Editors, mit welchem das in Punkt 1 gewählte Format gelesen und geschrieben werden kann. Beim Bau des Editors soll darauf geachtet werden, dass schon gängige Modelling-Frameworks für .Net benutzt werden. Im Hinblick darauf, dass später in diesem Editor auch Workflows modelliert oder Strukturbäume bearbeitet werden können, soll der Editor erweiterbar gestaltet sein. Optional soll auch darauf geachtet werden, dass der Editor auch im Browser lauffähig wäre (Silverlight + WPF) und Scripting-Fähigkeiten (um z.B. Abhängigkeiten zwischen den Feldern in einer Maske definieren zu können) aufweisen würde.

2.2 Meilensteine

Die Milestones sind so ausgelegt, dass noch jeder Phase konkrete Resultate präsentiert werden können. Da die verschiedenen Phasen aufeinander aufbauen und als zusätzliche Kontrolle wird jede Phase mit einem Milestone beendet, wie in Tabelle 2.1 zu sehen ist.

Tabelle 2.1: Meilensteine

MS	*Beschreibung*	*Arbeitsresultate*	*Iteration*	*Datum*
MS1	Projektplan	Templates, Projektplan und Zeitplan fertig	Inception	SW02
MS2	Analyse	Domain Modell und Externes Design fertig	Elaboration 1	SW04
MS3	Prototyp	Prototyp über alle Layer	Elaboration 2	SW07
MS4	Alpha Release	Core Features implementiert	Construction 1	SW09
MS5	Beta Release	Feature Freeze	Construction 2	SW12
MS6	Abgabe	Die Software ist getestet und funktioniert	Transition	SW14

2.3 Meetings

Einmal pro Woche setzen wir uns mit dem Betreuer zusammen und besprechen die Fortschritte der Arbeit sowie allfällige Fragen und das weitere Vorgehen. Von diesen

Meetings erstellt jeweils ein Teammitglied ein Sitzungsprotokoll und stellt dieses dem Betreuer bis spätestens 48 Stunden nach der Sitzung zu.

Das Meeting ist wöchentlich auf den **Dienstag um 15:00** festgelegt. Eventuelle Abweichungen werden vorher mit dem Betreuer abgemacht.

Ausserdem wird bei Bedarf ein Meeting in Aarau mit TIE durchgeführt. Dies ersetzt dann das normal Meeting am Dienstag.

2.4 Risiko Management

2.4.1 Fehler im Domainmodel

Falls beim Erstellen des Domainmodels etwas übersehen wurde und deshalb nicht alle benötigten Details in der daraus definierten Dateistruktur dargestellt werden können, ist das Projekt ernsthaft gefährdet, da dies je nach Fortschritt des Entwicklungsstandes aufwendige Änderungen nach sich ziehen kann.

Wahrscheinlichkeit: ca. 5%

Zeitverlust: ca. 75h

Gewichtet: 4h

Massnamen zur Verhinderung

Um Problemen mit dem Domain Modell vorzubeugen, investieren wir viel Zeit in das Studieren von Unterlagen und Besprechen von Ideen. Ausserdem besprechen wir das Domain Modell mit den verantwortlichen Personen (Dozent, Vertreter von TIE).

Massnamen bei Eintreten

Wenn trotzdem Probleme auftreten, suchen wir den Fehler und machen einen Plan, wie wir das Problem mit möglichst geringem Aufwand beheben können. Je nach Fortschritt des Projekts ist der Aufwand unterschiedlich gross.

2.4.2 Probleme mit WPF

Da wir nur grundlegende Erfahrung mit WPF haben, können wir im Vorhinein nicht genau abschätzen, wie viel Zeit uns einzelne Features wie Drag & Drop kosten.

Wahrscheinlichkeit: ca. 10%

Zeitverlust: ca. 20h

Gewichtet: 2h

Massnamen zur Verhinderung

Wir beginnen frühzeitig mit dem Technologiestudium im Bereich WPF. Dazu lesen wir Bücher und informieren uns im Internet.

Ein Buch, das uns schon in früheren Projekten weiterhalf, mit welchem wir uns in die Materie vertiefen ist das „Pro WPF in C# 2008“ [Mac08].

Massnamen bei Eintreten

Wir versuchen das Problem mit vernünftigem Zeitaufwand zu beheben oder ändern die Funktionalität des Programms entsprechend ab, sollte dies nicht möglich sein.

2.4.3 Kein Modelling Framework

Wenn wir kein passendes Modelling Framework für .Net finden, müssen wir diese Elemente selber implementieren und verlieren dadurch Zeit.

Wahrscheinlichkeit: ca. 50%

Zeitverlust: ca. 20h

Gewichtet: 10h

Massnamen zur Schadensminimierung

Wir beginnen frühzeitig mit der Evaluation eines geeigneten Frameworks.

Massnamen bei Eintreten

Falls kein passendes Modelling Framework gefunden werden kann, implementieren wir die benötigten Funktionen mit möglichst minimalistisch implementiert, sodass nur die Features, die für die Umsetzung benötigt werden unterstützt.

2.4.4 Weitere Risiken

Oben wurden die drei Risiken genannt, welche wir als am gefährlichsten für den erfolgreichen Abschluss des Projekts betrachten. Es existieren jedoch noch zahlreiche weitere Risiken wie etwa Probleme mit der Infrastruktur oder teaminterne Angelegenheiten, welche einen zusätzlichen Zeitaufwand bedeuten, das Projekt jedoch nicht existentiell gefährden.

2.5 Qualitätsmassnamen

2.5.1 Versionsverwaltung

Um jederzeit auf die aktuelle oder alte Versionen der Dokumentation sowie des Quelltextes zugreifen zu können, verwenden wir das Versionsverwaltungssystem SVN[1], welches auf einem Server der HSR[2] gehostet wird. Das SVN-Repository wird die in der Tabelle 2.2 aufgelisteten Struktur für die Ordner verwenden.

Tabelle 2.2: Ordnerstruktur des SVN Repositories

Ordner	*Beschreibung*
/	Build Scripte
/Graphics	Grafiken im proprietären Format der Bildbearbeitung
/Latex	In LaTeX erstellte Dokumente
/Models	UML Diagramme
/Protocols	Sitzungsprotokolle
/Source	Quelltexte des Programms
/Specifications	Dokumente, welche von Betreuer zur Verfügung gestellt werden
/ThirdParty	Verwendete Komponenten von Drittanbietern
/Timemanagement	Zeitplan und individuelle Zeiterfassung

2.5.2 Dokumentation

Für ein Gelingen des Projekts ist die Qualität der Dokumentation sehr wichtig. Um dies zu gewährleisten, basieren alle Dokumente auf dem selben LaTeX Template. Dadurch können wir entweder ein Dokument mit allen Unterdokumenten oder die Unterdokumente wie beispielsweise den Projektplan separat generieren. Dies ist aufgrund der Anforderungen der HSR nötig.

2.5.3 Sitzungsprotokolle

Um klar festzuhalten, was an den Sitzungen jeweils besprochen wurde, erstellt jeweils eines der Teammitglieder ein Sitzungsprotokoll und stellt dieses auf den SVN Server. Ausserdem wird dem Betreuer innerhalb von 48 Stunden eine Kopie zugestellt.

1 Subversion
2 Hochschule für Technik Rapperswil

2.5.4 Zeitplan

Die Zeitplanung wird in einem separaten Dokument geführt. Darin ist ersichtlich, wer wie lange an was gearbeitet hat und wer in Zukunft wie lange an was arbeiten soll. Die Planung wird laufend den aktuellen Gegebenheiten angepasst und immer weiter verfeinert. Ausserdem führt jedes Teammitglied eine separate Zeiterfassung, woraus die Zeiten wöchentlich in den Zeitplan übernommen werden.

2.5.5 Style Guides

Einheitlicher Code erhöht die Lesbarkeit und Verständlichkeit des Codes. Die genauen Coding Guidelines sind mit StyleCop definiert, zudem gibt es noch die ausformulierten Ergänzungen, siehe Anhang.

2.5.6 Build Umgebung

Wir erstellen Build Scripts mit *MSBuild*, mit welchen jederzeit eine funktionsfähige Version der Applikation erstellt werden kann. Auch die Dokumentation kann automatisch per Build Script generiert werden.

2.5.7 Dokument Review

Um eine hohe Qualität der Dokumentation zu gewährleisten werden alle erstellten Dokumente von allen Teammitgliedern gelesen und allfällige Fehler korrigiert. Zur Verbesserung der Rechtschreibung wird zusätzlich eine automatische Rechtschreibkorrektur für LaTeXeingesetzt.

2.5.8 Unit Tests

Um die Funktionalität des Codes zu gewährleisten, werden Unit Tests erstellt, wobei eine möglichst grosse Abdeckung des Codes angestrebt wird. Die Abdeckung im Business Layer sollte mindestens 80% sein. Fehler im Code werden zuerst durch einen Unit Test reproduziert und erst dann behoben, um ein weiteres Auftreten desselben Fehlers auszuschliessen. Klassen, welche durch Unit Tests schlecht abzudecken (z.B. GUI) sind werden durch System Tests geprüft.

2.5.9 System Tests

Das Verhalten der gesamten Applikation wird durch System Tests geprüft. Dadurch wird sichergestellt, dass die implementierten Features auch wie erwünscht benutzbar sind.

Kapitel 3

3 Analyse

3.1 Domain Model

Abbildung 3.1: Domain Model

Das Domain Model (Abbildung 3.1) wurde aufgrund der durchgeführten Analyse erstellt und dann mit Hilfe von den Mitarbeitern von TIE korrigiert und verfeinert. Die einzelnen Klassen des Domain Models werden nachfolgend beschrieben.

3.1.1 AttributeProfile

Das AttributeProfile definiert eine Einheit und enthält die einzelnen ShowTypes. Die Zusammensetzung der Masken wird nicht beachtet, da diese in der *i-engine* festgelegt ist. In Tabelle 3.1 sind die Properties aufgelistet, welche nötig sind, um ein AttributeProfile zu beschreiben.

Tabelle 3.1: Properties des AttributeProfiles mit ihrem SQL Datentyp

NAME_L1 NOT NULL VARCHAR2(100)	**NAME_L2** NOT NULL VARCHAR2(100)
NAME_L3 NOT NULL VARCHAR2(100)	**ATTRIBUTE_PROFILE_ID** NOT NULL NUMBER
NAME NOT NULL VARCHAR2(100)	**TYPE** VARCHAR2(100)
DESCR VARCHAR2(100)	**PREFS_TOP** VARCHAR2(2000)
PREFS_BOTTOM VARCHAR2(2000)	

3.1.2 ShowType

Die ShowTypes gruppieren Attribute in einer Maske. Masken wiederum werden aus einzelnen ShowTypes zusammengebaut. Die ShowTypes werden durch einen eindeutigen Namen identifiziert. Die Namen der ShowTypes können beliebig sein, in Tabelle 3.2 werden die in der Praxis am häufigsten vorkommenden aufgelistet. Jedoch haben gewisse Namen von ShowTypes eine spezielle Bedeutung. So beschreibt etwa *TAB* einen ShowType welcher Tabs enthält.

Tabelle 3.2: Die am meisten verwendeten ShowTypes

BODY	BODY1	XML
HEADER	BODY0	BODY2
CONFIRMATION	BUTTON	HIDDEN
HDR-FILE	TAB	BODY3

3.1.3 Attribute

Attribute sind die Elemente in einem ShowType. Sie erfüllen eine bestimmte Funktion wie Darstellung eins Textfeldes oder eines Buttons. Ihre Eigenschaften werden durch Properties beschrieben. Das Property *TAG* definiert den Typ des Attributes. Alle Attribute können in allen ShowTypes vorkommen. In Tabelle 3.3 sind alle Tags aufgeführt. Es kann jedoch sein, dass nach Fertigstellung der Applikation noch weitere hinzukommen.

Tabelle 3.3: Vordefinierte Tags

BUTTON_BACK	DATE	POPUP
BUTTON_CLOSE	EMPTY	RADIO
BUTTON_EMPTY	FILE	SELECT
BUTTON_HREF	FINDER	SELECTMASTER
BUTTON_RESET	FIX	SELECTSLAVE
BUTTON_SCRIPT	FIXTEXT	SINGLECHOSEN
BUTTON_SUBMIT	HIDDEN	TAB
BUTTON_SUBMIT_EDITFRAME	INDEX_EDIT	TAB_SUBMIT
BUTTON_SUBMIT_EDITFRAME_TO_PARENT	MAIL	TEXT
BUTTON_SUBMIT_EDITFRAME_TO_TOP	MULTICHOSEN	TEXT_ML
BUTTON_SUBMIT_EXECUTOR_AND_CLOSE	NONE	TEXTAREA
BUTTON_SUBMIT_TO_EXECUTOR	PASSWORD	TITLE
CHECKBOX		

Es werden jedoch nicht alle Tags gleich häufig genutzt, deshalb werden in Tabelle 3.4 die häufigsten aufgelistet.

Tabelle 3.4: Meistverwendete Tags

TEXT	SELECT	DATE
TEXT_ML	NONE	TAB_SUBMIT
FIXTEXT	EMPTY	TITLE

Die Eigenschaften der Attribute werden durch Properties beschrieben. Dies betrifft sowohl Aussehen, Platzierung innerhalb des ShowTypes sowie den Inhalt des Attributes. In Tabelle 3.5 werden alle Properties aufgelistet. Auch hier ist es jedoch möglich, dass nach Fertigstellung der Applikation noch weitere Properties hinzukommen. Fett gedruckt ist jeweils der Name des Properties, danach in Normalschrift der SQL Datentyp, wie das Property in der Datenbank hinterlegt wird.

Tabelle 3.5: Properties mit ihrem SQL Datentyp

DISPLAY_NAME_L1 VARCHAR2(500 CHAR)	**DISPLAY_NAME_L2** VARCHAR2(500 CHAR)
DISPLAY_NAME_L3 VARCHAR2(500 CHAR)	**DEFAULT_VALUE_L1** VARCHAR2(500)
DEFAULT_VALUE_L2 VARCHAR2(500)	**DEFAULT_VALUE_L3** VARCHAR2(500)
ATTRIBUTE_PROFILE_ID NOT NULL NUMBER	**SEQ** NOT NULL NUMBER
SHOW_TYPE VARCHAR2(100)	**ATTRIBUTE_NAME** NOT NULL VARCHAR2(500)
DISPLAY_NAME NOT NULL VARCHAR2(500 CHAR)	**H_POS** NUMBER
V_POS NUMBER	**MANDATORY** VARCHAR2(100)
POPUP_TYPE VARCHAR2(100)	**POPUP_OBJ_ID** NUMBER
POPUP_MANDATORY VARCHAR2(100)	**CHECKER** VARCHAR2(100)
TAG VARCHAR2(100)	**TAG_PREFS** VARCHAR2(500)
COLSPAN NOT NULL NUMBER	**FORMAT** VARCHAR2(100)
DEFAULT_VALUE VARCHAR2(500)	**SERVER_CHECKER** VARCHAR2(1000)
ROWSPAN NOT NULL NUMBER	**PAGE_NR** NOT NULL NUMBER
DISPLAY_NAME_WIDTH NOT NULL NUMBER(7,2)	**VALUE_STMT_ID** NOT NULL NUMBER
REFERENCE VARCHAR2(500)	**POPUP_P1** VARCHAR2(100)
POPUP_DIALOGPREFS VARCHAR2(1000)	**POPUP_PROFILE_ID** NUMBER
DESCR VARCHAR2(100)	**DATATYPE** NOT NULL VARCHAR2(100)

3.1.4 Property

Wie in Tabelle 3.5 zu erkennen, gibt es grundsätzlich drei verschiedene Typen von Properties. Zusätzlich gibt es einen weiteren Typ, der nicht direkt anhand der Tabelle erkennbar ist, jedoch trotzdem Sinn macht, das *EnumProperty*.

Für jeden Typen muss ausserdem definiert werden, ob ein Property *notNull* ist. Mit diesem Wert kann man erzwingen, das ein Property einen gültigen Werte besitzen muss, bevor das AttributeProfile gespeichert werden darf.

String Property Speichert einen String mit definierter Länge.

Integer Property Speichert einen Ganzzahlwert.

Float Property Speichert einen Dezimalzahlwert mit bestimmter Genauigkeit.

Enum Property lässt nur eine definierte Liste von Strings als Wert zu.

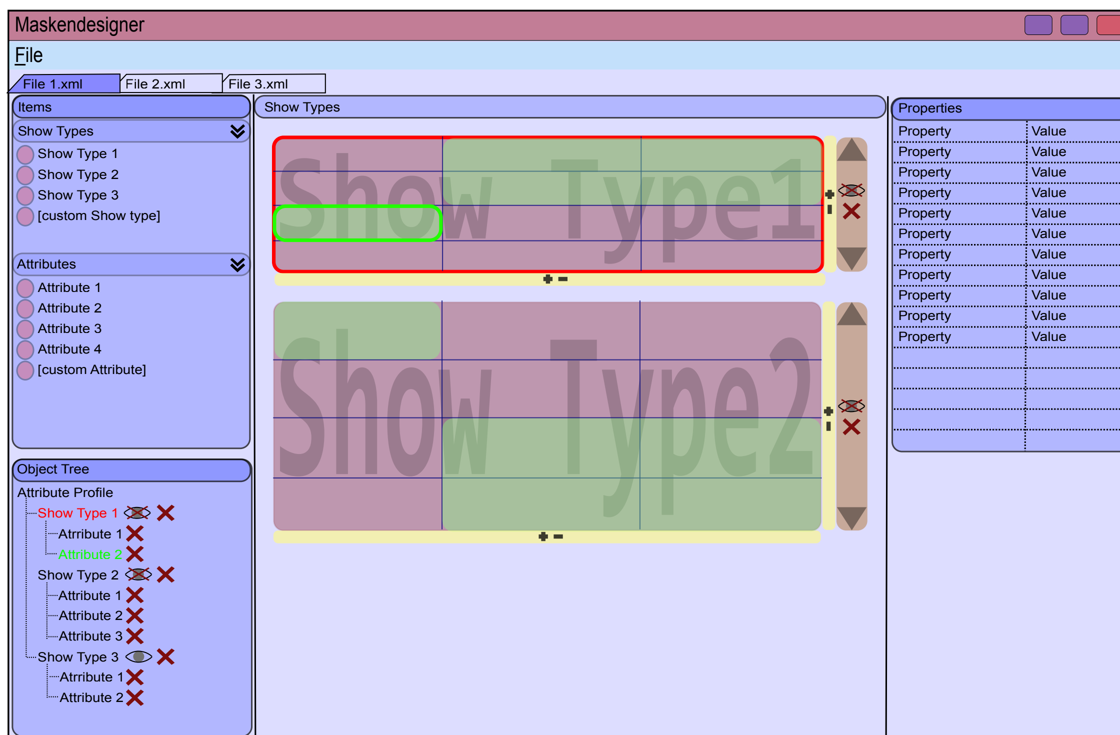


Abbildung 3.2: Paper Prototype

3.2 Paper Prototype

Menüleiste

Eine Menüleiste stellt die Funktionen für das Speichern, Laden und Erstellen von AttributeProfilen zur Verfügung. Für diese Funktionen werden dann die Standarddialoge von WPF verwendet.

Tabs

Wie üblich in Editoren für Benutzerinterfaces werden die geöffneten Dokumente in Tabs dargestellt. Dies ermöglicht ein angenehmeres Arbeiten mit mehreren offenen Dokumenten. Innerhalb eines Tabs werden alle Panels, welche für die Bearbeitung des AttributeProfiles benötigt werden, dargestellt.

Panels

Im GUI werden sogenannte Panels verwendet, welche jeweils eine bestimmte Funktionalität zur Verfügung stellen. Diese können vom User nach seinen persönlichen Vorlieben

angeordnet werden. Für kleinere Bildschirme gibt es noch die Möglichkeit des Autohides. Damit werden nicht verwendete Panels am Rand versteckt und kommen beim Darüberfahren mit der Maus zum Vorschein.

3.2.1 Items Panel

Hier stehen vordefinierte Attribute und ShowTypes zur Auswahl, welche zur Erstellung des AttributeProfiles verwendet werden können. Das sind diejenigen, die erfahrungsgemäss am häufigsten gebraucht werden. Zusätzlich können eigene ShowTypes und Attribute definiert und dann verwendet werden.

Erzeugen eines ShowTypes

Wenn der Benutzer einmal auf einen vordefinierten ShowType klickt erscheint ein Button mit dem der ShowType erstellt werden kann. Bei einem Doppelklick wird er direkt erstellt. Frisch erstellte ShowTypes werden im *ShowTypes* Fenster an oberster Stelle angezeigt. Um einen eigenen ShowType zu erstellen klickt Benutzer auf *Custom ShowType* und ändert nach der Erstellung im Property Grid den Namen und andere Properties nach seinen Vorlieben ab.

Erzeugen eines Attributs

Um ein Attribute zu erzeugen klickt der Benutzer auf das gewünschte Attribute und markiert anschliessend im zuvor erstellten ShowType diejenigen Felder, in welche er das Attribute erzeugen will. Anschliessend kann er die Properties des gerade erstellten Attributes im *Property Panel* bearbeiten.

3.2.2 ObjectTree Panel

Im ObjectTree werden alle im AttributeProfile enthaltenen ShowTypes und Attribute mit Hilfe eines TreeControls hierarchisch angezeigt. Dies ermöglicht es dem Benutzer, auch versteckte Objekte anzuzeigen und erhöht ausserdem die Übersicht. Im Object Tree können ShowTypes ein- und ausgeblendet oder gelöscht werden. Auch Attribute können direkt gelöscht werden. Das im *ShowTypes Panel* markierte Element wird grafisch hervorgehoben und ist immer synchron zur Auswahl in dem Object Tree.

3.2.3 Property Panel

Im Property Panel werden die Properties des momentan ausgewählten Objektes angezeigt. Diese können dort auch vom Benutzer nach seinen Vorlieben geändert werden. Jedoch wird immer geprüft, ob die Eingabe gültig ist, damit keine ungültigen Einstellungen möglich sind.

3.2.4 ShowTypes Panel

Das ShowTypes Panel besitzt eine List von ShowType Views, welche untereinander angeordnet sind. Das zu bearbeitende Element wird immer farblich hervorgehoben. Mittels Drag & Drop können die einzelnen ShowTypes in der Liste angeordnet werden.

3.2.5 ShowType View

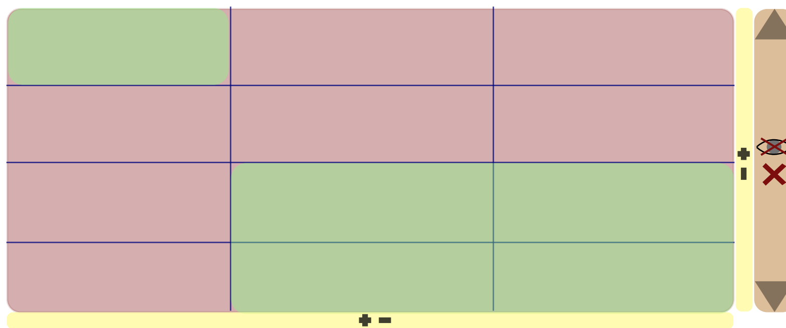


Abbildung 3.3: Paper Prototype: Show Type

Die ShowType View (Abbildung 3.3) repräsentiert immer nur einen ShowType. Die ShowType View besteht aus drei Teilen.

Grid

Der grösste und wichtigste Teil der View ist das Grid. Hier werden die einzelnen Attribute dargestellt, welche mit Drag & Drop verschoben und skaliert werden können. Das im Grid selektierte Attribut wird auch farblich hervorgehoben.

+ und - Buttons

Auf der rechten Seite und am unteren Rand des Grids sind jeweils zwei Buttons um das Grid zu vergrössern oder zu verkleinern.

Toolbar

Am rechten Rand des ShowTypes befindet sich eine weitere Toolbar. Mit den darin enthaltenen Kontrollelementen kann ein ShowType in der Reihenfolge innerhalb des Panels nach oben oder unten verschoben werden. Ausserdem kann er gelöscht oder unsichtbar gemacht werden.

Kapitel 4

Design

4.1 Features

4.1.1 Core Features

Die Core Features sind zwingend nötig für einen erfolgreichen Abschluss des Projekts. Sie stellen die grundlegende Funktionalität der Applikation zur Verfügung, welche nötig sind, um AttributeProfile zu erstellen oder zu bearbeiten. Die Core Features werden bereits in der Alpha Version implementiert:

- Erstellen und Löschen von AttributeProfiles, ShowTypes und Attributes
- Bearbeiten von Properties
- Speichern und Laden von / in XML Dokumente

4.1.2 Zusatz Features

Die Zusatzfeatures erleichtern das Bedienen der Anwendung für den Benutzer. Sie sind zum Gelingen des Projektes nicht zwingend notwendig, sollten aber soweit möglich implementiert werden. Die Zusatzfeatures werden in der Beta Version implementiert:

- Copy & Paste von ShowTypes und Attributes
- Erstellen und Verschieben von Elementen per Drag & Drop
- Benutzerdefinierte Anzeige von ShowTypes (einzelne ShowTypes ein- und ausblenden)
- Erstellen von ShowTypes und Attributes aus Templates welche in XML Dateien hinterlegt sind
- XML Elemente im Editor als Templates abspeichern

4.2 Frameworks und Libraries

4.2.1 AvalonDock

AvalonDock ist eine Window/Docking Library, welche es ermöglicht ein auf Komponenten basiertes GUI zu erstellen und dieses frei anzupassen. AvalonDock bietet Panels und Tabbed Documents, welche mittels Drag & Drop vom User verschoben und angepasst werden können. Ausserdem orientiert es sich im Look & Feel an Visual Studio 2008 und ist vollständig in WPF implementiert. Es ist daher wahrscheinlich, dass der Benutzer bereits mit ähnlichen Oberflächen gearbeitet hat, was der Bedienbarkeit förderlich ist. AvalonDock ist Open Source und kann von *http://avalondock.codeplex.com* heruntergeladen werden. Es steht unter einer vereinfachten BSD-Lizenz.

4.2.2 Enterprise Library 4.1

Die von Microsoft *Patterns and Practices* veröffentlichte Enterprise Library ist nun in der Version 4.1 verfügbar und wird auch auf *CodePlex* veröffentlicht. Die Enterprise Library bietet Komponenten für Datenbanken, Logging, Exceptionhandling, Dependency Injection und noch mehr. Wir verwenden aus der Enterprise Library folgende Komponenten:

- Unity - Dependecy Injection
- Logging
- Exception Handling

Auch die Enterprise Library ist Open Source und darf für jegliche Projekte, auch kommerzielle, verwendet werden.

4.3 Dateiformat

4.3.1 Evaluation

Die Evaluierung eines geeigneten Dateiformates war ein wesentlicher Teil der Arbeit. Grundsätzlich gab es dabei zwei Möglichkeiten:

Bestehendes Format verwenden

Die Verwendung eines bestehenden Dateiformates für die Abspeicherung von GUIs bietet den Vorteil, dass möglicherweise auf bestehendes Knowhow zurückgegriffen werden kann. Allerdings sind GUIs sehr vielfältig und es ist praktisch nicht möglich, alle denkbaren Variationen mit einem einzigen Dateiformat abzuspeichern. Dies bedeutet, dass ein erheblicher Overhead nötig wäre, um ein AttributeProfile in einem bestehenden Dateiformat abzuspeichern.

Zur Diskussion stand das Format XForms welches für die Beschreibung von Datenerfassungsmasken auf verschiedenen Geräten vorgesehen ist. Dies ist jedoch ein sehr umfangreicher Standart woraus wir uns dann die benötigten Features hätten picken müssen. Ausserdem sind nicht alle benötigten Features zur Beschreibung der AttributeProfile abgedeckt. Ein weiteres Problem besteht darin, dass keine geeignete Bibliothek für die .Net Plattform exisistiert was eine eigene Implementation nötig gemacht hätte.

Eigenes Format auf Basis von XML entwickeln

Die Alternative war, ein eigenes Format auf Basis von XML entwickeln. Der grösste Vorteil besteht darin, dass ein eigenes Format perfekt auf die konkreten Bedürfnisse zugeschnitten werden kann. Ausserdem besteht bereits eine Umfangreiche Unterstützung für XML auf der .Net Plattform, was einiges an Implementationsarbeit erspart. Ausserdem ist XML weit verbreitet und etabliert, wodurch von *AttributeProfile Designer* erstellte Profile beispielsweise auch mit einem anderen XML Editor bearbeitet werden können.

Entscheidung

Wir entschieden uns dazu, ein eigenes Format auf Basis von XML zu entwickeln. Dies insbesondere deshalb, weil wir keinen klaren Vorteil in der Verwendung eines bereits bestehenden Formates erkennen konnten. Wie oben beschrieben hätte dies sogar einen Mehraufwand in der Implementation bedeutet, da zuerst die Unterstützung für XForms hätte realisiert werden müssen. XML hingegen wird bereits sehr gut unterstützt und ausserdem sind sich viele Menschen bereits gewohnt, XML Dateien zu lesen.

4.3.2 Schema

Das Schema (Listing 4.1) beschreibt den XML-Syntax der verwendet wird um die AttributeProfile zu speichern. Diese können dann beim Laden gegen das Schema geprüft werden um sicherzustellen, dass die Informationen im XML Dokument korrekte Werte enthalten.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Attribute" type="Attribute" />
  <xs:complexType name="Attribute">
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="StringProperty" type="StringProperty" />
        <xs:element name="EnumProperty" type ="EnumProperty" />
        <xs:element name="IntegerProperty" type="IntegerProperty" />
        <xs:element name="FloatProperty" type="FloatProperty" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Properties" type="Properties" />
```

```
<xs:complexType name="Properties">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="StringProperty" type ="StringProperty" />
      <xs:element name="EnumProperty" type ="EnumProperty" />
      <xs:element name="IntegerProperty" type ="IntegerProperty" />
      <xs:element name="FloatProperty" type ="FloatProperty" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:element name="ShowTypes" type="ShowTypes" />
<xs:complexType name="ShowTypes" >
  <xs:sequence>
    <xs:element name="ShowType" type="ShowType" minOccurs="0"
        maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="Attributes" type="Attributes" />
<xs:complexType name="Attributes" >
  <xs:sequence>
    <xs:element name="Attribute" type="Attribute" minOccurs="0"
        maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="AttributeProfile" type="AttributeProfile"/>
<xs:complexType name="AttributeProfile">
  <xs:sequence>
    <xs:element name="Properties" type="Properties" minOccurs="0"
        maxOccurs="1"/>
    <xs:element name="ShowTypes" type="ShowTypes" minOccurs="0"
        maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="Property" type="Property"/>
<xs:complexType name="Property">
  <xs:attribute name="name" type ="xs:string" use="required" />
  <xs:attribute name="allowNull" type="xs:boolean" use="required" />
  <xs:attribute name="value" type="xs:string" default="" />
  <xs:attribute name="extended" type="xs:boolean" default="false" />
  <xs:attribute name="readOnly" type="xs:boolean" default="false"
      use="optional" />
</xs:complexType>
<xs:element name="ShowType" type="ShowType"/>
<xs:complexType name="ShowType">
  <xs:sequence>
    <xs:element name="Attribute" type="Attribute" minOccurs="0"
        maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required" />
</xs:complexType>
<xs:element name="StringProperty" type="StringProperty"/>
```

```
  <xs:complexType name="StringProperty">
    <xs:complexContent>
      <xs:extension base="Property">
        <xs:attribute name="length" type="xs:int" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="EnumProperty" type="EnumProperty"/>
  <xs:complexType name="EnumProperty">
    <xs:complexContent>
      <xs:extension base="StringProperty">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
          <xs:element name="EnumValue" type="xs:string" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="IntegerProperty" type="IntegerProperty"/>
  <xs:complexType name="IntegerProperty">
    <xs:complexContent>
      <xs:extension base="Property">
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="FloatProperty" type="FloatProperty"/>
  <xs:complexType name="FloatProperty">
    <xs:complexContent>
      <xs:extension base="Property">
        <xs:attribute name="precision" type="xs:int" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Listing 4.1: XML Schema

4.3.3 Beispieldokument

Im Listing 4.2 ist ein Beispieldokument zu sehen, welches ein fiktives AttributeProfile repräsentiert.

```
<?xml version="1.0" encoding="utf-8"?>
<AttributeProfile xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Properties>
    <StringProperty name="Name_L1" allowNull="false" value="L1_New Profile"
        length="100" />
    <IntegerProperty name="Attribute_Profile_ID" allowNull="false" value="1"
        />
    <StringProperty name="Name" allowNull="false" value="New Profile"
        length="100" />
```

```
</Properties>
<ShowTypes>
  <ShowType name="Body">
    <Attribute>
      <StringProperty name="Tag" allowNull="false" value="BUTTON_BACK"
          readOnly="true" length="100" />
      <StringProperty name="Attribute_Name" allowNull="false"
          value="BUTTON_BACK" length="500" />
      <EnumProperty name="Mandatory" allowNull="true" value="No"
          length="100">
        <EnumValue>Yes</EnumValue>
        <EnumValue>No</EnumValue>
      </EnumProperty>
      <IntegerProperty name="Seq" allowNull="false" value="1" />
      <StringProperty name="Display_Name" allowNull="false" value="&lt;!New
          Attribute!&gt;" length="500" />
      <FloatProperty name="Disply_Name_Width" allowNull="false" value="32"
          extended="true" precision="2" />
      <IntegerProperty name="Attribute_Profile_ID" allowNull="false"
          value="1" extended="true" />
      <IntegerProperty name="Value_Stmt_ID" allowNull="false" value="0"
          extended="true" />
    </Attribute>
    <Attribute>
      <StringProperty name="Attribute_Name" allowNull="false"
          value="CHECKBOX" length="500" />
      <StringProperty name="Tag" allowNull="false" value="CHECKBOX"
          readOnly="true" length="100" />
      <IntegerProperty name="Seq" allowNull="false" value="2" />
      <EnumProperty name="Mandatory" allowNull="true" value="No"
          length="100">
        <EnumValue>Yes</EnumValue>
        <EnumValue>No</EnumValue>
        <EnumValue>Fix</EnumValue>
      </EnumProperty>
      <FloatProperty name="Disply_Name_Width" allowNull="false" value="32"
          extended="true" precision="2" />
      <IntegerProperty name="Attribute_Profile_ID" allowNull="false"
          value="1" extended="true" />
      <IntegerProperty name="Value_Stmt_ID" allowNull="false" value="0"
          extended="true" />
    </Attribute>
      <StringProperty name="Tag" allowNull="false" value="FILE"
          readOnly="true" length="100" />
      <IntegerProperty name="Seq" allowNull="false" value="5" />
      <IntegerProperty name="Value_Stmt_ID" allowNull="false" value="0"
          extended="true" />
    </Attribute>
  </ShowType>
  <ShowType name="Body0">
    <Attribute>
```

```
        <StringProperty name="Tag" allowNull="false" value="MAIL"
            readOnly="true" length="100" />
        <FloatProperty name="Disply_Name_Width" allowNull="false" value="32"
            extended="true" precision="2" />
        <IntegerProperty name="Attribute_Profile_ID" allowNull="false"
            value="1" extended="true" />
        <IntegerProperty name="Value_Stmt_ID" allowNull="false" value="0"
            extended="true" />
      </Attribute>
        <StringProperty name="Tag" allowNull="false" value="TEXT_ML"
            readOnly="true" length="100" />
        <IntegerProperty name="Seq" allowNull="false" value="3" />
        <StringProperty name="Attribute_Name" allowNull="false"
            value="Attribute_2" length="500" />
        <StringProperty name="Display_Name" allowNull="false" value="&lt;!New
            Attribute!&gt;" length="500" />
        <IntegerProperty name="H_Pos" allowNull="false" value="1"
            extended="true" />
        <IntegerProperty name="V_Pos" allowNull="false" value="1"
            extended="true" />
        <EnumProperty name="Mandatory" allowNull="true" value="No"
            length="100">
          <EnumValue>Yes</EnumValue>
          <EnumValue>No</EnumValue>
          <EnumValue>Fix</EnumValue>
        </EnumProperty>
        <StringProperty name="Datatype" allowNull="false" value="TEXT"
            length="100" />
        <IntegerProperty name="Colspan" allowNull="false" value="1"
            extended="true" />
        <IntegerProperty name="Rowspan" allowNull="false" value="1"
            extended="true" />
        <IntegerProperty name="Page_Nr" allowNull="false" value="0"
            extended="true" />
        <FloatProperty name="Disply_Name_Width" allowNull="false" value="32"
            extended="true" precision="2" />
        <IntegerProperty name="Attribute_Profile_ID" allowNull="false"
            value="1" extended="true" />
        <IntegerProperty name="Value_Stmt_ID" allowNull="false" value="0"
            extended="true" />
      </Attribute>
    </ShowType>
  </ShowTypes>
</AttributeProfile>
```

Listing 4.2: XML Beispiel

4.3.4 Beschreibung

Im Folgenden werden die einzelnen Elemente des XML-Formates und dessen Attribute beschrieben, welche verwendet werden um ein AttributeProfile zu beschreiben.

AttributeProfile

Ein AttributeProfile ist der Roottag und definiert ein AttributeProfile, welches aus Properties und ShowTypes besteht.

Properties

Das Properties-Element ist ein Container für die Properties des AttributeProfiles.

Property

Ein Property ist lediglich der Basistyp für alle Properties, welche in einem AttributeProfile oder Attribute definiert werden können. Für die eigentlichen Properties müssen die konkreten Elemente verwendet werden:

- IntegerProperty
- FloatProperty
- StringProperty
- EnumProperty

Die Indirektion über Property-Elemente wurde gewählt, um flexibel zu bleiben und damit die Applikation auch mit Änderungen in der Datenbank klar kommt. Folgende Attribute sind in einem Property definiert:

name definiert ein String, welcher verwendet wird um das Property zu referenzieren

allowNull definiert mit einem boolschen Wert ob das Property auch weggelassen werden darf bzw. keinen Wert haben darf

value definiert den eigentlichen Wert des Properties, welcher aber nicht zwingend ein gültiger Wert sein muss d.h. in einem IntegerProperty kann der Wert auch *Hello* sein, die Applikation ist dafür verantwortlich, dass der Wert als ungültig markiert wird.

extended wird verwendet, um zu entscheiden ob dieses Property ein primäres oder ein erweitertes ist

readOnly kann ein Attribut für den Editor blockieren, so dass dieses vom Benutzer nicht verändert werden kann

IntegerProperty

Der Typ IntegerProperty definiert, dass das Property einen ganzzahligen Wert haben muss.

StringProperty

Der Typ StringProperty definiert, dass das Property als Wert eine beliebige Zeichenkette haben muss, welche aber die Maximallänge nicht überschreiten darf.

length gibt die maximale Länge des Strings an.

FloatProperty

Der Typ FloatProperty definiert, dass das Property eine Gleitkommazahl sein muss, welche eine bestimmte Genauigkeit erwartet.

precision gibt an, wie viele Nachkommastellen zugelassen sind.

EnumProperty

Mit einem EnumProperty kann ein StringProperty noch genauer spezifiziert werden, so dass nur noch bestimmte Strings (keine Wildcards) zulässig sind.

EnumValue Ein EnumProperty kann eine Liste von EnumValue Elementen besitzen, welche die gültigen Werte des Properties definieren. Diese werden in der Software mittels DropDownBox dargestellt.

ShowTypes

Das ShowTypes Element ist ein Container für die ShowTypes, die in einem AttributeProfile definiert werden können.

ShowType

Ein ShowType ist ein ShowType (Region) welche ein Grid (Tabelle) gefüllt mit Attributes besitzt. Ein ShowType besitz eine Liste von Attributes. Jeder ShowType besitzt einen eindeutigen Namen innerhalb des AttributeProfiles.

Attribute

Ein Attribute ist ein Feld, welches in einem ShowType angezeigt wird. Ein Attribute verweist auf ein Feld innerhalb der Datenbank und wird mit Properties parametrisiert, welche die Position im ShowType sowie dessen Darstellung bestimmen.

4.4 Architektur

4.4.1 Ziele

Gekapseltes UI

Das GUI soll vom Domain Layer vollständig getrennt sein und dessen Funktionalität dem User zur Verfügung stellen.

Erweiterbarkeit

Das System soll so konzipiert sein, dass es zu einem späteren Zeitpunkt erweitert werden kann um zum Beispiel eine Anbindung an eine Datenbank für direktes Publishen oder Laden von AttributeProfiles zu ermöglichen.

KIS - Keep it Simple

Da der Zeitrahmen begrenzt ist und eine eigenständige Applikation und kein Framework erwartet wird, wird der Fokus auf eine möglichst einfache Architektur gelegt, ohne sich jedoch zu stark einzuschränken.

Desktop Applikation

Die Applikation wird in erster Linie als *Stand-Alone* Dektop Applikation für Windows entwickelt. Optionen wie die Portierbarkeit auf *Silverlight* in einem ersten Schritt nicht berücksichtigt, jedoch sollte die Möglichkeit offen gehalten werden, dies später zu realisieren.

4.4.2 Evaluation

Für die Realisierung der Architektur gibt es grundsätzlich zwei Varianten, welche sich gut eignen würden:

Layer

- Vorteile:
  - Einfache Aufteilung
  - Klar definierte Funktionalitäten
- Nachteile:
  - Nachträgliches Hinzufügen von Features kann Änderungen an den Schnittstellen nötig machen

Komponenten

- Vorteile:
  - Einfaches Hinzufügen von weiteren Features oder neuen Schnittstellen
  - Flexibel und konfigurierbar
  - Komponenten sind überschaubarer
- Nachteile:
  - Abhängigkeiten müssen gelöst werden
  - Braucht mehr Infrastruktur Code

Entscheid

Wir haben uns für eine Kombination aus komponenten- und layerbasierter Architektur entschieden. Dies bringt uns den Vorteil, dass wir das GUI gut vom Rest der Applikation trennen können und trotzdem einzelne Teile als einfach austauschbare Komponenten realisieren können.

Aufbau

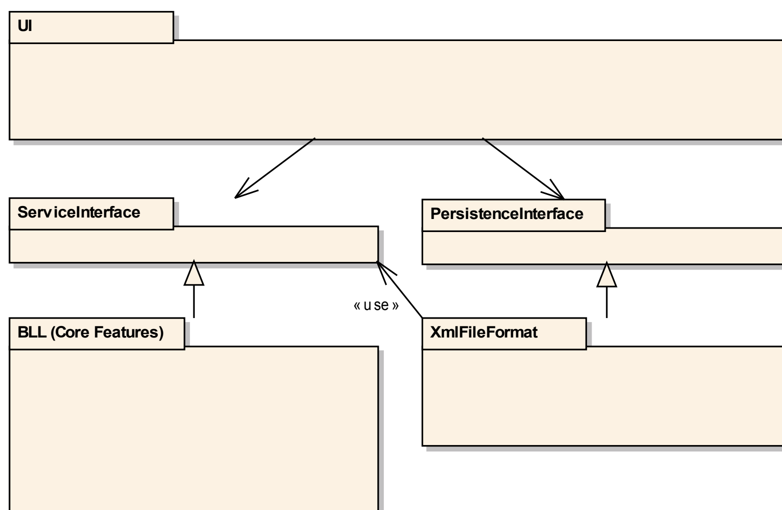


Abbildung 4.1: Package Model

In Abbildung 4.1 wird der Aufbau der Architektur des *AttributeProfile Designer* aufgezeigt. Das UI[1] benutzt Interfaces, um auf die Funktionalität für Businesslogik und

1 User Interface

Datenpersistenz zuzugreifen. So kann beispielsweise das Modul zur Datenpersistenz ohne grösseren Aufwand ausgetauscht werden.

UI

Das UI wird mit dem MVVM[1]-Pattern und WPF erstellt. Mit Hilfe von AvalonDock wird ein komponentenartiges GUI erstellt, wodurch das GUI einfach und schnell erweitert werden kann.

Service Interface

Das Service Interface stellt dem GUI eine bestimmte Funktionalität zur Verfügung, welche jedoch auch von anderen Komponenten verwendet werden kann. Jedes Service Interface braucht, um verwendet werden zu können, eine registrierte Implementierung. Dies ermöglicht ein einfaches Interagieren zwischen den Komponenten und verhindert eine zu hohe Kopplung.

BLL (Core Features)

Die BLL[2] Komponente implementiert die Core Features der Business Logik und stellt diese anderen Komponenten und dem UI zur Verfügung.

PersistenceInterface

Das PersistenceInterface ist eine Schnittstelle für den Import/Export des AttributeProfiles.

XmlFileFormat

Das XmlFileFormat implementiert das PersistenceInterface um die AttributeProfile in XML zu speichern bzw. aus diesen zu laden. Hier sind in zukunft auch andere Implementierungen wie beispielsweise ein Modul für den direkten Datenbankzugriff denkbar.

4.5 Deployment

Da es sich in diesem System um eine Desktop Applikation handelt, werden für das Deployment alle Files in einen Ordner auf dem Zielrechner kopiert und dort ausgeführt. Das Kopieren erfolgt über ein standard .Msi File, durch welches die Applikation im System registriert und auch wieder entfernt werden kann.

1 Model View Viewmodel
2 Business Logic Layer

Kapitel 5

Implementation

5.1 ProfileDesigner

Das Projekt ProfileDesigner ist die Application selbst und somit die ausführbare Datei. Dieses Projekt beinhaltet die UI-Logik und ist in die Namespaces ProfileDesigner.View, ProfileDesigner.ViewModel und ProfileDesigner aufgeteilt.

5.1.1 ProfileDesigner Namespace

Hier ist nur der „Bootstrap“-Code, der Code der die Applikation initialisiert und die Templatefiles lädt, vorhanden. Er instanziert die View und Viewmodles und Zeigt nach dem Laden das Hauptfenster an und übergibt diesem die Kontrolle.

5.1.2 UnitiyContainer

Das GUI erzeugt beim Starten der Applikation einen UnityContainer und registriert ein paar Klassen, welche dann über die Applikation verteilt als Service gebraucht werden können. Zusätzlich werden noch Klassen registriert, welche den globalen Applikationszustand repräsentieren, diese müssen als *ContainerControlledLiftime* registriert werden, da immer mit der gleichen Instanz gearbeitet werden muss.

BasicProfileDesigner ist der Service, der gebraucht wird um die Business Objekte zu erstellen (Templates).

XmlFormatter ist ein Service um XmlAttributeWriters und XmlAttributeReaders zu erzeugen sowie um XML Dokumente zu Laden und Speichern.

XmlFolderTemplateProvider liest die Konfiguration für die Ordner der Templates aus, welche dann beim Starten der Applikation geladen werden.

PersistencyManager ist ein Service, welcher die Integration in die Windows Umgebung kapselt, darunter fallen Dienste wie Copy & Paste sowie das Öffnen des FileOpen und FileSave Dialogs mit welchen Objekte gespeichert bzw. geladen werden können.

SelectionViewModel verwaltet und speichert die aktuelle Markierung von Objekten in der Applikation welche für Copy & Paste sowie für *Save as Template* und *Insert ShowType* verwendet wird.

5.2 ProfileDesigner.View

Die mit dem Paper Prototype erarbeiteten Ideen konnten grösstenteils realisiert werden. Gewisse Aspekte des GUI sind gegenüber dem Paperprototype optimiert worden. Dadurch ist die Verwendung besser an die Bedürfnisse der späteren Benutzer angepasst. Eine dieser Optimierungen war das weglassen des Object Trees, da Gespräche mit TIE gezeigt haben, dass dieser nicht von grossem Nutzen für den späteren Benutzer ist. Viele Detailoptimierungen sind auch in die ShowTypeView geflossen, da diese ein sehr wichtiger Teil der Applikation ist. ShowTypes können wegen dem nicht mehr vorhandenen Object Tree nicht komplett ausgeblendet sonder einfach verkleinert werden.

Bei der Implementierung haben wir darauf geachtet, dass in den codebehind Files der XAML[1] Dateien möglichst kein Code steht und alle Logik des UI in den View Models platziert wird. Dies war jedoch nicht immer möglich, da wir keine Möglichkeit gefunden haben, Commands direkt von einem anderen Element als den Buttons auszulösen. So sind im codebehind die Eventhandlers für die übrigen Events platziert, welche den entsprechenden Command im Viewmodel aufrufen.

Die Abbildung 5.1 zeigt einen Screenshot der Applikation mit den verschiedenen Panels, welche für das erstellen und bearbeiten von AttributeProfilen verwendet werde.

5.2.1 AttributeGrid

Das AttributeGrid ist das Herzstück der ShowTypeView. Hier werden alle Attribute angezeigt und können auch bearbeitet werden. Da dies einiges an Logik erfordert haben wir uns dazu entschlossen, eine separate Klasse dafür zu machen, welche vom WPF Grid Control ableitet und dieses mit den benötigten Funktionen erweitert.

1 Extensible Application Markup Language

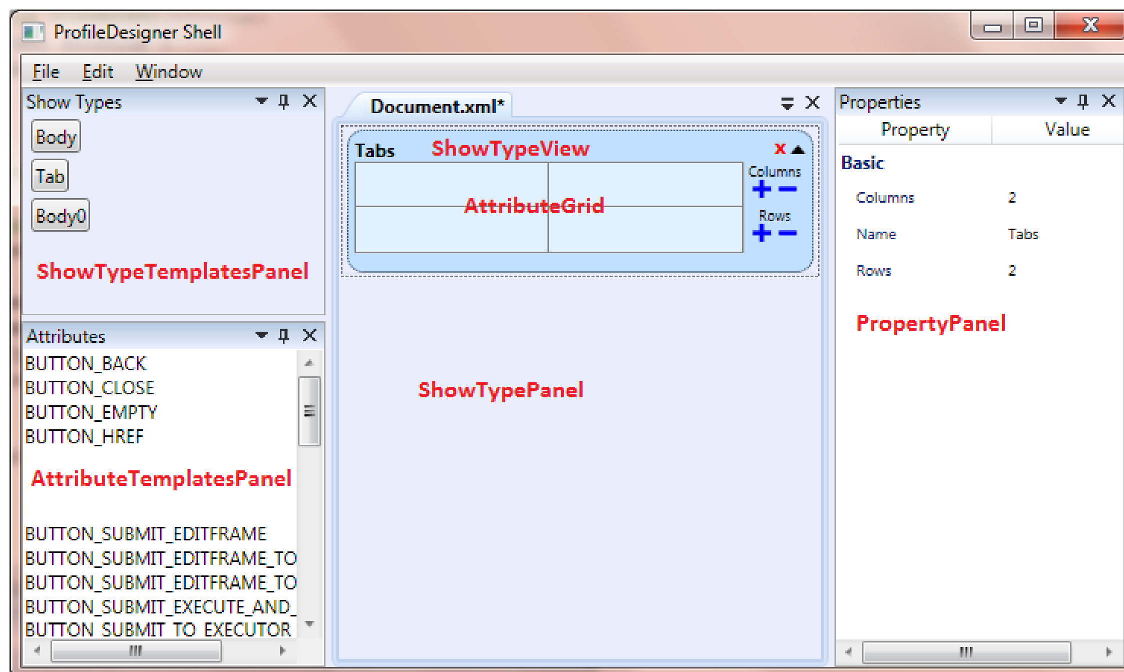


Abbildung 5.1: Übersicht über die Applikation

Das AttributeGrid ist zwar in der View, ist aber sehr eng mit dem *ShowTypeViewModel* verbunden und enthält sämtlichen Code welcher für die Darstellung des AttributeGrids notwendig ist. Diese starke Kopplung war notwendig, da die Darstellung der Attribute und des Grids stark von den Properties des ShowTypes sowie die der Attributes abhängig ist.

Zum AttributeGrid gehören indirekt auch das *SelectionRectangle* sowie das *EmptyAttributeGridCell*. Dies sind Hilfsklassen, welche das Erstellen und Verschieben von Attributen ermöglichen. Ausserdem wird die *AttributeView* im AttributeGrid gebraucht, welches die Darstellung der Attribute ermöglicht.

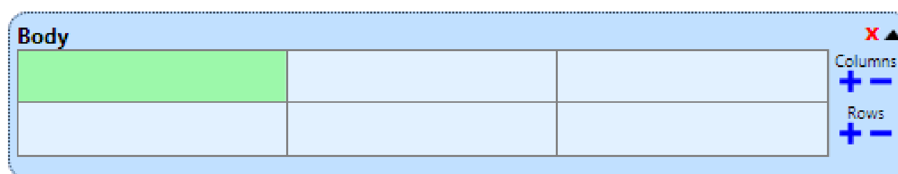


Abbildung 5.2: Mouseover auf einer leeren Zelle

In der Abbildung 5.2 wird ein leeres AttributeGrid gezeigt. Jede der Zellen in dem leeren Grid wurde mit einer *AttributeGridEmptyCell* gefüllt, welches die Hintergrundfarbe

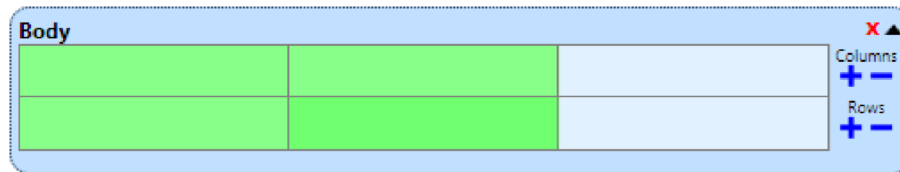


Abbildung 5.3: Erstellen eines neuen Attributs

auf Grün ändert sobald sich die Maus darüber befindet. Das AttributeGrid registriert einen Eventhandler auf den erzeugten AttributeGridEmptyCells für den Mousedown Event. Wird dieser ausgelöst stellt das AttributeGrid ein *SelectionRectangle* bereit, welches in der Zelle, die den Event ausgelöst hat platziert wird. Das SelectionRectangle, welches in Abbildung 5.3 gezeigt wird, überlagert die AttributeGridEmptyCell. Wird nun die Maus bewegt, so wird das *Colspan* und *Rowspan* des SelectionRectangle angepasst.

Bei einem Mouseup Event auf dem Grid wird ein neues Attribute anhand des selektierten Attribute Templates erzeugt und mit der Grösse und Position des SelectionRectangles eingefügt. Die *AttributeGridEmptyCells* bleiben erhalten, liegen aber unter dem erstellten Attribute, was durch eine entsprechende Z-Index Einstellung realisiert wird.

Drag & Drop

Um das Drag & Drop zu realisieren wurde eine ähnliche Lösung wie beim Erstellen von Attributen implementiert. Dazu wird der Mousedown Event auf dem Attribute registriert welcher dann den Drag & Drop Status setzt. Wird nun die Maus bei gedrückt gehaltener Maustaste über eine leere Zelle gefahren wird die Position des Attributes auf die Position der leeren Zelle gesetzt. Dabei wird laufend überprüft, ob ein entsprechendes Verschieben überhaupt möglich ist.

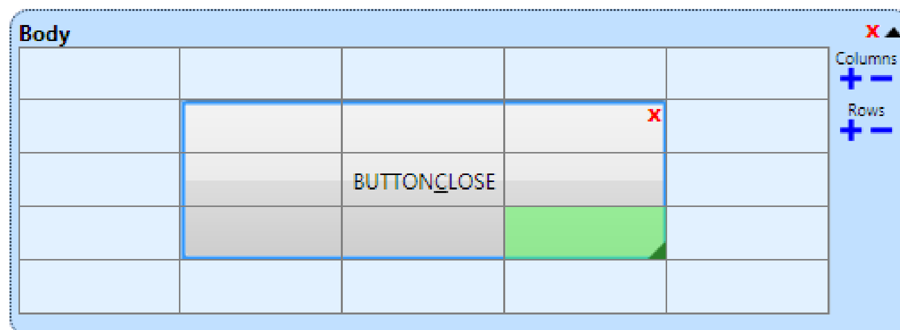


Abbildung 5.4: Skalieren eines bestehenden Attributes mit der Maus

Skalierung der Attribute

Um erstellte Attribute mit der Maus zu skalieren bedarf es eines kleinen Tricks welcher das Event Handling vereinfacht. Es wurde in der AttributeView ein Polygon über das Eigentliche Attribut gelegt, welches bei einem Mouse Down einen Event auslöst. Im AttributeGrid wird in diesem Event Handler der Attribute-Skale-State gesetzt welcher das aktuelle Element durch Anpassung des Z-Index hinter die *AttributeEmptyCells* setzt damit das Gitter sichtbar wird (siehe Abbildung 5.4). Diese ermöglicht auch das Abfangen des Events, wenn die Maus über eine Zelle fährt welche sich unter dem zu bearbeitenden Attribute befindet.

5.2.2 AttributeTemplatesPanel

DataContext: ShellViewModel

Im AttributeTemplatesPanel kann der Benutzer auswählen, welches Attribut er erzeugen will. Dazu werden alle aus den XML Files gelesenen Attribute untereinander dargestellt. Wenn der Benutzer auf eins klickt wird das jeweils aktive grafisch hervorgehoben.

5.2.3 AttributeView

DataContext: AttributeViewModel

Die AttributeView ist zuständig für die korrekte Darstellung der Attribute. Falls vorhanden werden die von TIE zur Verfügung gestellten Controls verwendet. Dies wird durch den StringToTieUserControlConverter realisiert, welcher einen String entgegen nimmt und das jeweils entsprechende Control zurückgibt. Wenn kein passenden Control vorhanden ist generiert der Converter ein Label mit einer Farbe, welche anhand des Tags generiert wird. So haben jeweils alle Attribute mit dem selben Tag die gleiche Farbe.

5.2.4 EmptyAttributeGridCell

Die EmptyAttributeGridCell's werden in jeder leeren Zelle des AttributeGrids als Platzhalter hinterlegt. Damit kann die das AttributeGrid erkennen, auf welche leere Zelle der Benutzer geklickt hat. EmptyAttributeGridCell's hätten auch direkt im Code des AttributeGrids erzeugt werden können, wir haben uns jedoch dazu entschlossen hierfür ein separates UserControl zu machen um das Aussehen des Grids per XAML definieren zu können. Beispielsweise die Farbe beim Mouseover oder die Gitternetzlinien innerhalb des Grids.

5.2.5 PropertyPanel

DataContext: AttributeProfileViewModel

Im PropertyPanel werden die Properties aller Elemente, welche im Programm vorkommen können angezeigt. Je nach Parametriesierung des PropertyTypes können die Properties unterschiedlich bearbeitet werden.

Es werden immer die Properties des gerade selektierten Elements angezeigt. Das selektierte Element wird zudem grafisch hervorgehoben. Bei den Attributen werden die Properties ausserdem in wichtige und weniger wichtige aufgeteilt (Basic und Advanced).

Im PropertyGrid ist eine intuitive Bedienung für das zügige Arbeiten sehr wichtig. So mussten wir im Codebehind einige Hilfsfunktionen ausprogrammieren, welche aber nur die View betreffen.

Wird ein Property verändert, so wird automatisch mit Hilfe des PropertyTypes der neue Wert überprüft. Kommt diese Überprüfung zum Schluss, dass das Property einen fehlerhaften Wert aufweist, so wird es Farblich hervorgehoben.

5.2.6 SelectionRectangle

Das SelectionRectangle ist eine Hilfsklasse des AttributeGrids. Seine Aufgabe besteht darin, denjenigen Bereich hervorzuheben welcher der Benutzer im AttributeGrid per Drag & Drop markiert, um ein neues Attribute platzieren zu können. Auch hier haben wir uns dazu entschlossen, ein eigenes UserControl zu machen damit das Aussehen per XAML definiert werden kann.

5.2.7 Shell

DataContext: ShellViewModel

Die Shell stellt das Grundgerüst der Applikation zur Verfügung, welches mit Hilfe von AvalonDock realisiert ist. Dine einzelnen Panels, welche für die Bearbeitung zur verfügung stellen können so einfach nach den Vorlieben des Users angeordnet werden.

Die Shell ist das Hauptfenster, welches beim Start der Applikation erstellt wird. Sie initialisiert und verwaltet alle weiteren Fenster und Dateien, welche geladen werden. Sie ist verantwortlich, dass allen Views der richtige DataContext zugewiesen wird. Wenn die Shell geschlossen wird, so beendet sich auch die Applikation.

5.2.8 ShowTypePanel

DataContext: AttributeProfileViewModel

Das ShowTypePanel stellt das AttributeProfile dar und enthält dessen ShowTypes. Die ShowTypes werden untereinander dargestellt. Ausserdem wird durch farbliche Hervorhebung angezeigt, wenn das Attribut Profil selber markiert ist. Ausserdem kann die Reihenfolge der ShowTypes frei definiert werden.

5.2.9 ShowTypeTemplatesPanel

DataContext: ShellViewModel

Im ShowTypeTemplatesPanel werden alle Templates von ShowTypes angezeigt. Der Benutzer kann also hier ShowTypes aus Vorlagen erstellen, indem er auf den gewünschten ShowType klickt, welcher sogleich im ShowTypePanel angezeigt wird. Dort kann er dann mit Hilfe des PropertyPanel den Namen sowie andere Einstellungen verändern.

5.2.10 ShowTypeView

DataContext: ShowTypeViewModel

Die ShowTypeView stellt jeweils einen ShowType dar. Dabei gibt es zwei Möglichkeiten, wie ein ShowType dargestellt werden kann. Einerseits die normale Darstellung mithilfe des Attribute Grids (Abbildung 5.6) und andererseits eine spezielle *Tab* Darstellung, in der alle Attribute untereinander dargestellt werden.

Die *Tab* Darstellung (Abbildung 5.5) wird automatisch verwendet, wenn der Name des ShowTypes *Tab* ist. Die Auswahl der Darstellung ist im XAML realisiert, im Hintergrund werden immer beide Views gehalten, die jeweils andere wird ausgeblendet.

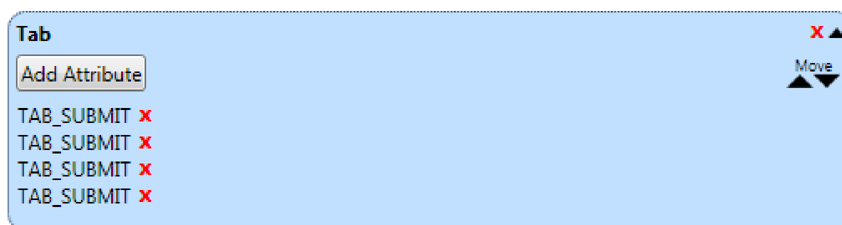


Abbildung 5.5: Ein Beispiel einer ShowTypeView mit Tab Ansicht

Grid-Darstellung

Die Standarddarstellung ist ein Grid (siehe Abbildung 5.6), welche es erlaubt verschiedene Attribute unter- und nebeneinander auszurichten. Die Grid-Darstellung überprüft auch ob sich die Attribute überschneiden oder verdeckend.Beim Speichern wird ein Fehler falls dies so ist, da solche ShowTypes von der i-engine nicht gerendert werden können.

Tab-Darstellung

Eine sogenannte *Tab*-Darstellung ordnet die Attribute in einer Liste an, wie in der Abbildung 5.5 zu sehen ist. Die Attribute werden nach Sequenznummer der Attribute sortiert, wobei die kleinste nummer zuerst kommt.

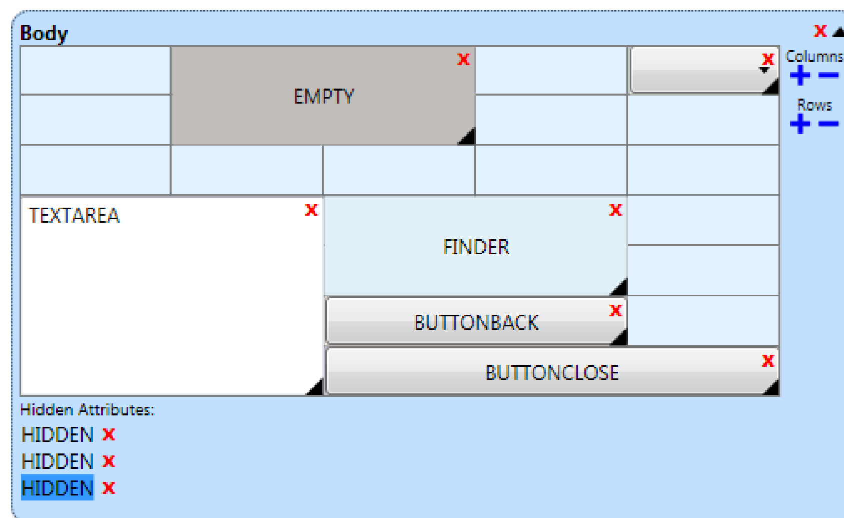


Abbildung 5.6: Ein Beispiel einer ShowTypeView mit dem AttributeGrid

Funktionen

Die ShowTypeView stellt auf der rechten Seite Kontrollelemente dar mit denen die ShowTypeView sowie das AttributeGrid konfiguriert werden können (siehe Abbildung 5.7).

Rows Mit den Symbolen + und - kann eine Zeile hinzugefügt oder entfernt werden. Es kann jedoch keine Zeile entfernt werden, in der sich ein Attribut befindet.

Columns Mit den Symbolen + und - kann eine neue Spalte hinzugefügt oder entfernt werden, auch hier kann keine Spalte entfernt werden, in der sich noch ein Attribut befindet.

Move Mit den beiden Pfeilen kann ein ShowType in der Liste nach oben oder unten verschoben werden.

Abbildung 5.7: ShowTypeView mit Kontrollelementen auf der rechten Seite

5.2.11 Commands

Da wir das MVVM Pattern verwenden, geschieht das Auslösen von Aktionen über Commands. Da die normalen ICommands für unsere Zwecke nicht ausreichten, verwenden wir die Hilfsklasse RelayCommand von Josh Smith [Smi09], um Commands zu implementieren, welche einen hinterlegten Delegate ausführen. In Listing 5.1 wird gezeigt, wie dies mit Hilfe von Lambdas sehr elegant gelöst werden kann.

Ausserdem verwenden wir die von WPF zur Verfügung gestellten ApplicationCommands, um Standardfunktionalität mit den dafür vorgesehenen Tastenkombinationen zu implementieren. Die ApplicationCommands können irgendwo im Programm ausgelöst werden und werden dann im ShellViewModel abgearbeitet.

```
if (removeColumnCommand == null)
    removeColumnCommand = new RelayCommand(
        obj => Columns--, param => Columns > 1);

return removeColumnCommand;
```

Listing 5.1: Verwendung des RelayCommands

5.2.12 Converter

Da im ViewModel keine Werte vorkommen sollten, welche spezifisch die View betreffen wie etwa Brushes oder Visibilities, haben wir Converter eingesetzt welche die Werte des ViewModels in die spezifischen Werte der View konvertieren und gegebenenfalls wieder zurück.

AttributeSeqSortConverter

Da die Attribute in der ShowTypeTabView nach ihrer Sequenz sortiert werden sollten wurde ein Converter erstellt, welcher eine Liste von Attributen entgegennimmt und diese nach ihrer Sequenznummer sortiert wieder zurückgibt.

PropertyListGroupConverter

Dieser Converter gruppiert die Properties anhand ihres *Extended* Flags, um so die Properties in *Normale* und *Erweiterte* zu unterteilen.

RelayConverter

Wie das RelayCommand bietet der RelayConverter die Möglichkeit, einen neuen Converter mithilfe von Delegates zu erstellen. Diese können gebraucht werden um Filter- und Sortfierfunktionalität in einer CollectionView hinzuzufügen ohne dafür neue Klassen zu definieren und mit Delegates die Konvertierung vorzunehmen.

```
ItemViewModel selected = Container.Resolve<SelectionViewModel>().SelectedItem;
ICollectionView view = CollectionViewSource.GetDefaultView(
    selected.PropertyList.Properties);
view.GroupDescriptions.Add(
    new PropertyGroupDescription("Type.Extended", new RelayConverter
    {
        Converter = RelayConverter.CreateTypedConverter<bool, string>(
            val => val ? "Advanced" : "Basic", "Advanced"),
        ConverterBack = RelayConverter.CreateTypedConverter<string, bool>(
            val => (val != "Basic"), true)
    }
));

view.SortDescriptions.Add(
    new SortDescription("Type.Name", ListSortDirection.Ascending));
return view;
```

Listing 5.2: Beispiel einer Gruppierung einer Collection

In Listing 5.2 wird anhand eines konkreten Beispiels gezeigt, wie eine PropertyList in eine gruppierte CollectionView konvertiert wird.

StringToTieUserControlConverter

Dieser Converter ist sehr wichtig, denn er wandelt einen String, welcher den Tag des Attribute beinhaltet, in das entsprechende UserControl von TIE um. Falls für den Tag kein entsprechendes User Control hinterlegt ist, generiert er ein Label, welches den Tag darstellt damit auch solche Attribute im AttributeGrid sichtbar sind.

VisibilityConverter

Der VisibilityConverter wandelt einen boolschen Wert in einen Visibility Wert um, damit im ViewModel ein boolsches Flag genügt, um zu bestimmen, ob etwas sichtbar ist oder nicht. Bei False wird Visibility auf *Collapsed* gesetzt um den Platz in der View freizugeben sofern das Objekt nicht angezeigt wird.

5.3 ProfileDesigner.ViewModel

Die ViewModels dienen dazu, die Informationen des Business Layer mit der View zu verbinden. Die Anbindung der View an das ViewModel ist per Databinding realisiert, was dem normalen Weg in WPF entspricht. Änderungen, welche von der View kommen, werden an den Business Layer weitergegeben und umgekehrt. Dies entspricht dem MVVM Pattern. In Abbildung 5.8 ist das Klassendiagramm der ViewModels dargestellt.

5.3.1 AttributeProfileViewModel

Das AttributeProfileViewModel hält alle Daten, welche zu einem AttributeProfile gehören und welche die View betreffen. Dazu gehören alle ShowTypes sowie die Properties welche das AttributeProfile betreffen. Ausserdem wird der Filename sowie die Information ob das Profil seit dem letzten Speichern geändert wurde gehalten. Falls dies der Fall ist wird dem Filenamen ein Sternchen hinzugefügt, um dies dem Benutzer grafisch darzustellen.

5.3.2 AttributeTemplateViewModel

Ein AttributeTemplateViewModel repräsentiert jeweils ein Template eines Attributes. Dieses Template kann dann im ShowType instanziiert werden. Dazu muss ein AttributeTemplateViewModel selektiert sein, welches dann im SelectionViewModel registriert wird. Sobald der Benutzer die Erstellung des Attributes auslöst, wird dieses im SelectionViewModel registrierten AttributeTemplateViewModel im entsprechenden ShowType instanziiert.

5.3.3 AttributeViewModel

Jedes AttributeViewModel entspricht einem Attribute bzw. einer PropertyCollection im Business Layer. Die für die Anzeige wichtigen Properties des Attributes werden überwacht und die jeweiligen Werte der View als Property zur Verfügung gestellt. Hier geht es vor allem um *Row*, *RowSpan*, *Column*, *ColumnSpan* und *Tag*. Sprich die Platzierung und Grösse des Attributes auf dem AttributeGrid.

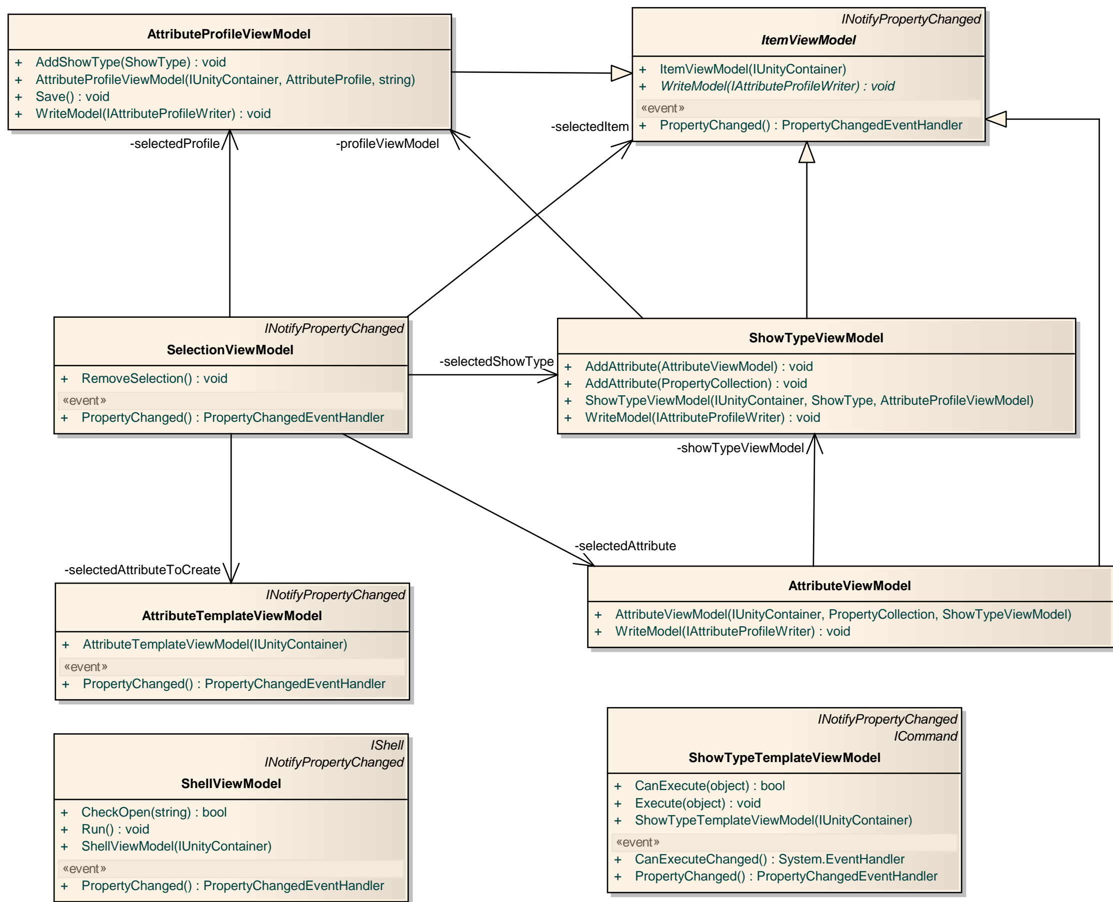


Abbildung 5.8: Klassendiagramm der ViewModels

5.3.4 ItemViewModel

Das ItemViewModel ist die Basisklasse von *AttributeProfileViewModel*, *ShowTypeViewModel* und *AttributeViewModel*. Es fasst diejenigen Eigenschaften zusammen, über welche alle Items verfügen müssen. Wichtig ist hier insbesondere, dass für jedes Item definiert sein muss, ob es gerade selektiert ist oder nicht. Dies wird insbesondere für Copy & Paste gebraucht. Ausserdem stellt es die Funktion bereit, ein Item als Template zu speichern. Dies kann jeweils über das Kontextmenü ausgelöst werden. Auch stellt es für alle Items das *SelectionViewModel* zur Verfügung, welches sicherstellt, dass zur selben Zeit immer nur ein ShowType und ein Attribute bzw. das AttributeProfile als selektiert markiert ist.

5.3.5 SelectionViewModel

Das SelectionViewModel hält die aktuelle Selektierung im Designer fest und definiert dadurch, welche Properties gerade im PropertyPanel angezeigt werden. Dieses ViewModel wird benötigt, um bei Copy & Paste den Inhalt am richtigen Ort einzufügen sowie um den selektierten Inhalt in die Zwischenablage zu kopieren.

5.3.6 ShellViewModel

Das ShellViewModel stellt die elementaren Funktionen der Applikation bereit. Dazu gehören das Speichern und Laden von AttributeProfiles sowie das Bereitstellen von Templates für ShowTypes und Attributes.

5.3.7 ShowTypeTemplateViewModel

Damit der Benutzer ShowTypes aus Vorlagen erstellen kann, gibt es das ShowTypeTemplateViewModel. Dieses repräsentiert alle Vorlagen, welche für ShowTypes zur Verfügung stehen. Diese wiederum werden aus XML Dokumenten gelesen und dem Benutzer zur Verfügung gestellt. Wenn der Benutzer ein ShowType aus einem Template instanziieren möchte, klickt er auf den entsprechenden ShowType und der ShowType wird im gerade aktiven AttributeProfile erstellt.

5.3.8 ShowTypeViewModel

Jedes ShowTypeViewModel entspricht einem ShowType im Business Layer. Es enthält zusätzlich Daten, welche nur zur Anzeige benötigt, jedoch nicht abgespeichert werden. Dies sind insbesondere die Anzahl der Im AttributeGrid dargestellten Zeilen und Spalten. Auch wird eine Observable Collection mit allen AttributeViewModels gehalten. Zusätzlich existiert eine weitere Collection mit allen Hidden Attributes, welche automatisch immer auf dem aktuellen Stand gehalten wird.

Da die ShowTypes im Business Layer keine PropertyList haben, welche im PropertyPanel angezeigt werden könnte, wird zur Laufzeit eine temporäre generiert, welche den Namen sowie die Anzahl Zeilen und Spalten des Grids enthält. Diese wird dann im PropertyPanel angezeigt wodurch der Benutzer auf gewohnte Weise die Properties des ShowTypes bearbeiten kann.

5.4 ProfileDesigner.Bll

5.4.1 Implementationskonzept

Der Business Layer lädt die verschiedenen Elemente (AttributeProfiles, ShowTypes, ...) mittels XML Serialisierung. Für die XML Serialisierung wurde ein XML Schema definiert, aus welchem dann mittels eines Programms des Visual Studios (xsd.exe) typsichere Klassen generiert. Diese Klassen entsprechen bei einer Serialisierung dem dazugehörigen XML Schema. Dies ermöglicht eine sehr einfache Implementation des XML Schemas innerhalb des Business Layer.

```
public void WriteProfile(Generated.AttributeProfile profile,
    XmlWriter xmlWriter){

    XmlSerializer ser =
        new XmlSerializer(typeof(Generated.AttributeProfile));
    ser.Serialize(xmlWriter, profile);
}

public Generated.AttributeProfile ReadProfile(XmlReader xmlReader){
    XmlSerializer ser =
        new XmlSerializer(typeof(Generated.AttributeProfile));
    return ser.Deserialize(this.xmlReader)
        as Generated.AttributeProfile;
}
```

Listing 5.3: Xml Serialisierung

In Listing 5.3 wird gezeigt, wie mit Hilfe der von xsd.exe generierten Klassen, welche sich im Namespace „Generated“ befinden von einem XML Stream die Elemente innerhalb des XMLs ausgelesen bzw. geschrieben werden.

Die Verwendung dieser generierten Klassen hat zudem den Vorteil für die Fehlertolleranz innerhalb der XML Dateien. Fehlerhafte Einträge werden automatisch ignoriert falls, das XML sonst in einem korrekten Format ist.

Da der Code für die Generierten Klassen für die weitere Verarbeitung ungeeignet ist, da er z.B mit Arrays statt Collections arbeitet, verwenden wir für den Businesslayer nochmals separate Klassen. Dies hat allerdings zur Folge, das alle geladenen Element für die Verwendung nochmals in neue Businessobjekte gemapped werden müssen. Dadurch kann aber die Sichtbarkeit und Read/Write zugriffe besser kontrolliert werden.

Klassendiagramm

Die Abbildung 5.9 zeigt das UML Diagramm für den Businesslayer.

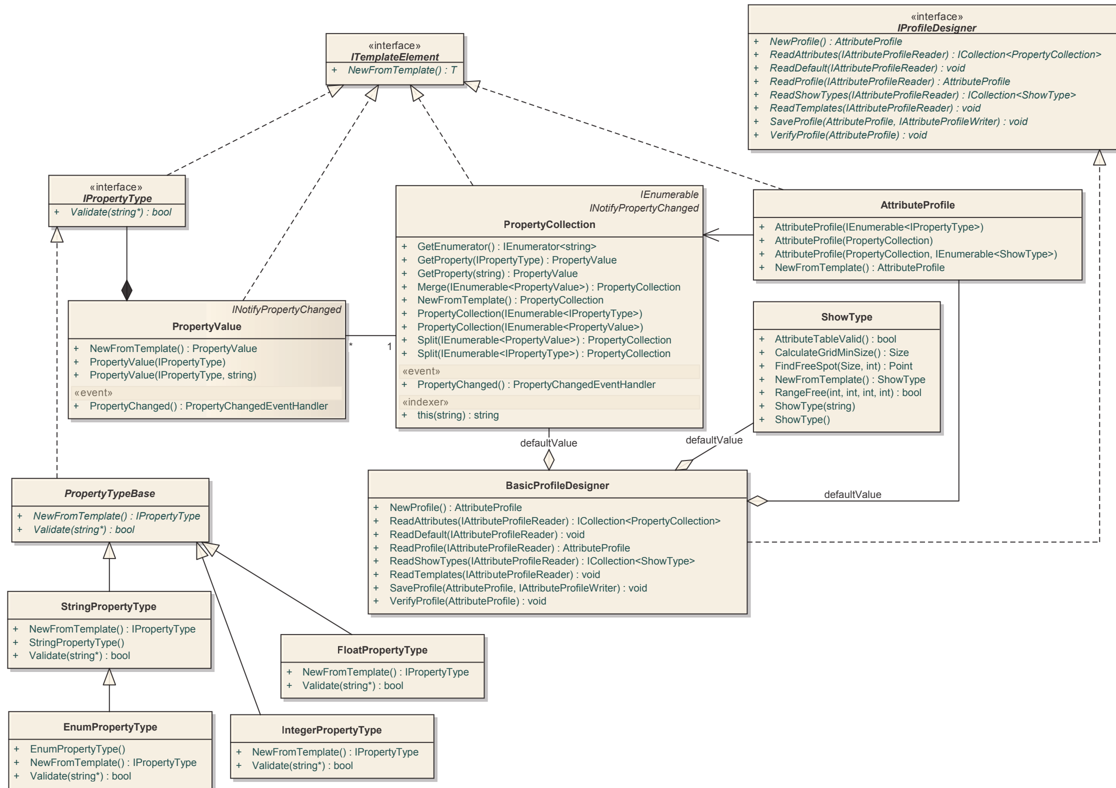


Abbildung 5.9: Klassendiagramm der Business Objekte

Templates

Um Templates unterstützen zu können, besitzen die meisten Klassen des Business Layer eine Methode (Prototype Pattern) um eine neue Instanz aus einer bestehenden zu erstellen. Dadurch ist es möglich, für Templates die selben Klassen und somit auch das selbe XML Schema sowie dessen Read/Write Funktionen zu verwenden, was den Code flexibler und einfacher macht.

PropertyCollection

Die Klasse PropertyCollection implementiert das PropertyList Pattern, um flexibel Properties für Elemente wie das AttributeProfile oder die Attribute zu definieren. Die Properties werden mit Hilfe eines Typs auf ihre Gültigkeit überprüft und konvertiert.

5.4.2 Interfaces

IProfileDesigner

IProfileDesigner ist das Service Interface mit dem der Business Layer verwendet werden kann. Damit können die Templates und Defaults für neue Instanzen der Business Objekte sowie das Instanziieren und Verifizieren neuer und bestehender Business Objekte (Siehe Abbildung 5.9) verwaltet werden.

ITemplateElement

ITemplateElement ist das Interface, welches die Methode definiert, um aus bestehenden Business Objekten (Templates) neue zu erzeugen. Die Methode sollte immer ein *Deep* Copy vollziehen (Siehe Abbildung 5.10), um sicherzustellen, dass das Template nicht durch Seiteneffekte ungewollt verändert wird.

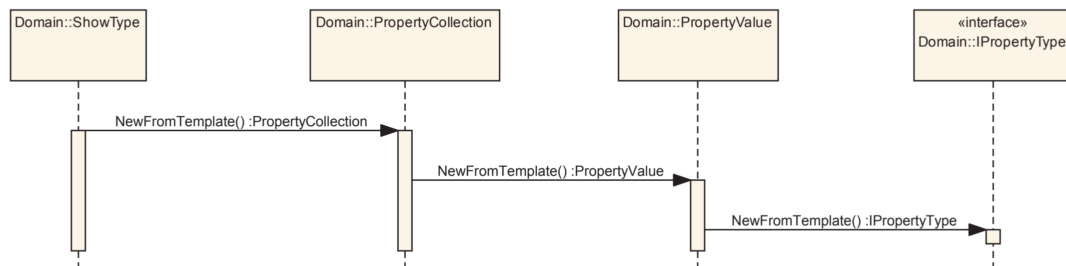


Abbildung 5.10: Deep Copy eines ShowTypes

IPropertyType

Dieses Interface ist eine Art TypeObject [FB96] und wird von den verschiedenen Propertytypen (String, Float, Enum...) implementiert. Diese verifizieren die Werte auf gültigkeit basierend auf der Parametrisierung.

5.4.3 Persitence Interfaces

IAttributeProfileReader und *IAttributeProfileWriter* müssen nur dann paarweise auftreten, wenn es gewünscht ist Lese- und Schreibfunktionalität zu unterstützen. Falls nur ein Export in eine Datenbank oder nur ein Import benötigt wird, so kann man auch nur eines der beiden Interfaces implementieren.

IAttributeProfileReader

Der IAttributeProfileReader wird verwendet um eine generelle Schnittstelle zu bieten mit welcher Business Objekte gelesen werden können. Es muss mit jeder Implementation

möglich sein, ganze AttributeProfiles oder auch nur einzelne Business Objekte zu lesen, da der Reader auch genutzt wird um Templates zu lesen. XmlAttributeProfileReader ist eine Referenzimplementierung, welche die vom XML Schema generierten Klassen verwendet um ein XML Schema konformes Dokument auszulesen. Ausserdem wird das Interface auch für Copy & Paste verwendet.

IAttributeProfileWriter

Der IAttributeProfileWriter wird verwendet um eine generelle Schnittstelle zu bieten, mit welcher Business Objekte geschrieben werden können. Auch hier muss es möglich sein, nur Teile eines AttributeProfiles zu schreiben. XmlAttributeProfileWriter ist eine Referenzimplementierung, welche die Business Objekte dem XML Schema entsprechend abspeichert. Dieser wird für Copy & Paste verwendet, da alles was von ihm gespeichert wurde vom XmlAttributeProfileReader wieder gelesen werden können muss.

IPersistenceFormatter

Bietet Methoden an um neue Reader und Writer zu instanziieren. Dies wird verwendet um passende Reader und Writer zu kombinieren. Der IPersistenceFormatter sollte sicher stellen, dass die von ihm erzeugten Writer mit den Readern kompatibel sind. Für die XML Reader und Writer wird der XmlFormatter verwendet (siehe Abbildung 5.11).

5.4.4 Business Objekte

PropertyCollection

Die PropertyCollection ist eine Liste mit Properties als MetaObjekt [FB96], welche für ein bestimmtes Element definiert sind.

AttributeProfile

Das Profile besitzt Properties, ShowTypes und eine Location, um das Profile wieder dahin zurück zu speichern von wo es geladen wurde.

ShowType

Repräsentiert die ShowTypes in einem AttributeProfile. Darin werden grundlegende Verifizierungen implementiert, um sicherzustellen, dass die Platzierung der einzelnen Attribute innerhalb des ShowTypes korrekt platziert sind und gültige Werte aufweisen. Die verschiedenen Attribute werden als PropertyCollection’s gespeichert.

5.4.5 Templates Handling

Laden eines default Templates

Die default Templates werden anderst geladen als die normalen Templates, da sie die normalen Templates ergänzen, damit auch Elemente, die in den Templates nicht explizit angegeben wurden, in der Applikation bearbeitet werden können.

Für jeden Typ (AttributeProfile, ShowType und Attribute) kann nur ein default Template geladen werden welches dann als Master für alle weiteren Templates dient. Es wird immer das zuletzt geladene default Template pro Typ verwendet.

Laden eines Templates

Falls ein default Template gesetz wurde wird dieses kopiert und mit den Elementen die vom IAttributeReader gelesen werden ergänzt. Sind Elemente doppelt vorhanden, also im default Template und im zu ladenden Template, werden die Elemente des neu geladenen Templates mit denjenigen des default Templates überschreiben. Wurde kein default Template geladen, wird ein neues, leeres Template erstellt und mit den Werten, welche vom Reader gelesen werden, gefüllt.

Speichern eines Objektes

Beim Speichern werden nur Werte gespeichert, welche nicht *null* sind. Dadurch werden die XML Dokumente kleiner und übersichtlicher.

Laden von gespeicherten Objekten

Da nur Werte welche nicht *null* sind gespeichert wurden, können auch nur diese wieder geladen werden. Die restlichen Werte werden durch die des dazugehörigen default Templates für den Typen ergänzt.

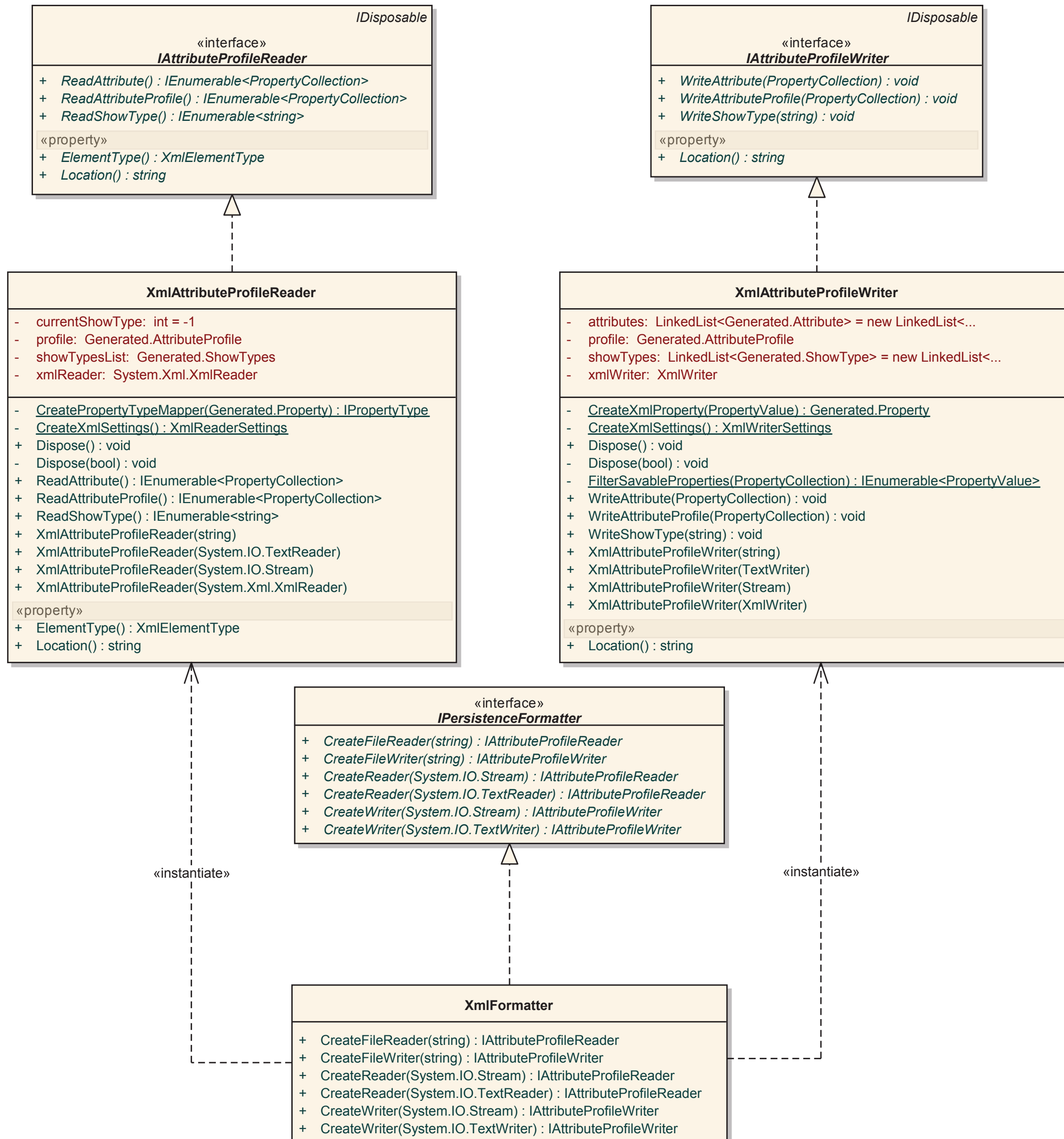


Abbildung 5.11: Klassendiagramm XML Interface

5.5 ProfileDesigner.Common

Hier sind nützliche Hilfsklassen, welche in allen Projekten verwendet werden, implementiert.

5.5.1 CollectionUtils

Hier sind Extension Methods für .Net Collections und Iteratoren (`IEnumerable<T>`) definiert, welche helfen, das Programmieren zu erleichtern.

Die Methode *SyncCollectionTo* hilft dabei, ObservableCollections mit den Domain Collections zu verbinden. Damit wird jedes Mal wenn ein Objekt in eine ObservableCollection eingefügt oder entfernt wird ein entsprechendes Element in der Domain Collection eingefügt oder entfernt.

```
    ShowTypeViewModels.SyncCollectionTo(
        Profile.ShowTypes, stvm => stvm.ShowType);
```

Listing 5.4: Verknüpfung zwischen ShowTypeViewModel-Collection und der ShowType-Collection

In Listing 5.4 wird eine ObservableCollection an die normale Collection *Profile.ShowTypes* gebunden. Das zu erzeugende oder zu löschende Objekt wird über das ShowType Property des ViewModels gefunden.

5.5.2 ExceptionHelper

Ist eine Hilfsklasse, die sicherstellt, dass das Exception Handling mit Hilfe der EnterpriseLibrary richtig verwendet und einen Log Eintrag in den Windows Event Log erstellt wird.

```
public AttributeProfile ReadProfile(IAttributeProfileReader reader) {
    return ExceptionHelper.SafeCall(
    "BLL",
    () => {
        AttributeProfile profile;
        ...
        return profile;
  });
}
```

Listing 5.5: Verknüpfung zwischen ShowTypeViewModel-Collection und der ShowType-Collection

Im Listing 5.5 wird gezeigt, wie das Laden eines AttributeProfiles mit Hilfe der *BLL* Policy des ExceptionHandlings der EnterpriseLibrary gemacht wird. Würde das Laden eines Profiles eine Exception werfen, würde diese automatisch geloggt werden anhand der definierten Policy.

5.5.3 IModule

Das IModule ist ein Interface um Objekte in dem ApplicationContainer (Unity) hinzuzufügen.

5.5.4 Bootstrapper

Ist eine kleine Hilfsklasse um beim Applikationsstart IModules zu registrieren und den Unity Container zu füllen.

Kapitel 6

Projektabschluss

6.1 Risiken

Zu Beginn des Projekts haben wir die Risiken analysiert und entsprechende Vorkehrungsmassnahmen getroffen. Nachfolgend wird analysiert, welche Risiken eingetreten und wie wir damit umgegangen sind.

6.1.1 Fehler im Domainmodel

Durch die Meetings in Aarau bei TIE konnten wir Fehler im Domainmodel rechtzeitig erkennen und so das Domainmodel so gut als möglich an die Bedürfnisse des Kunden anpassen und Fehler beseitigen.

6.1.2 Probleme mit WPF

Mit WPF sind keine grösseren Probleme aufgetreten; wir konnten alles benötigte innert nützlicher Zeit wie geplant realisieren. Das frühzeitige Technologiestudium hat sich also ausgezahlt.

6.1.3 Kein Modelling Framework

Wir konnten tatsächlich kein passendes Modelling Framework finden und mussten so die benötigte Funktionalität selber implementieren. Jedoch ist dies nicht so tragisch, da die Einarbeitung in das Framework auch Zeit gekostet hätte und wir uns mittlerweile genug gut mit WPF auskennen, um die benötigte Funktionalität ohne grössere Probleme selber zu implementieren. Ausserdem hätte so ein Framework auch ein gewissen Overhead mit sich gebracht. Das Hinzufügen weiterer Funktionen ist Aufgrund der Verwendung von AvalonDock trotzdem relativ einfach möglich.

6.1.4 Fazit

Es traten keine grösseren Risiken ein und wir waren immer recht gut im Zeitplan. Das Abschätzen der Risiken und die entsprechenden Vorkehrungsmassnahmen haben sich also gelohnt und sich in Form eines reibungslosen Projektablaufs ausgezahlt.

6.2 Tests

6.2.1 Unit Tests

Um die Qualität des Codes zu sichern und um aufgetretene Fehler am erneuten Auftreten zu hindern, haben wir Unit Tests erstellt. Hierbei lag der Fokus insbesondere auf dem Business Layer, welchen wir nahezu vollständig abdecken konnten (Tabelle 6.1). Auch das Viewmodel testeten wir, soweit dies sinnvoll war.

Tabelle 6.1: Codeabdeckung

Namespace	*%*
Gesamtes Projekt	48%
ProfileDesigner	1%
ProfileDesigner.Properties	28%
ProfileDesigner.View	0%
ProfileDesigner.ViewModel	49%
ProfileDesigner.Domain	91%
ProfileDesigner.Domain.Generated	100%
ProfileDesigner.Common	60%

6.2.2 System Tests

Um zu überprüfen, ob die implementierten Features auch so arbeiteten wie geplant, führten wir System Tests durch (siehe separates Dokument). Ausserdem halfen uns System Tests beim Testen des GUIs, was über Unit Tests nur schwer möglich ist. Durch die System Tests konnten wir noch einige Schwachstellen in der Applikation finden und diese beheben.

6.3 Zeit Management

6.3.1 Zeitaufwand pro Teammitglied

Vorgegeben für die Semesterarbeit ist pro Teammitglied ein Zeitaufwand von ungefähr 240 Stunden. Die ergibt pro Woche knapp 18 Stunden. Wir haben uns daher als Ziel einen Arbeitsaufwand von 18 Stunden pro Woche gesetzt. Wie in Abbildung 6.1 zu sehen, waren wir bis und mit Woche 10 immer etwa in diesem Bereich. In Woche 11 sollte gemäss Zeitplan die Applikation soweit fertig gestellt sein, dass alle Features enthalten sind und wir uns der Dokumentation sowie den Tests widmen können. Dies resultierte in einem höheren Zeitaufwand (Tabelle 6.2). Über die gesamte Dauer des Projekts haben wir das gesteckte Ziel von 240 Stunden knapp überschritten was aber im erwarteten Rahmen liegt.

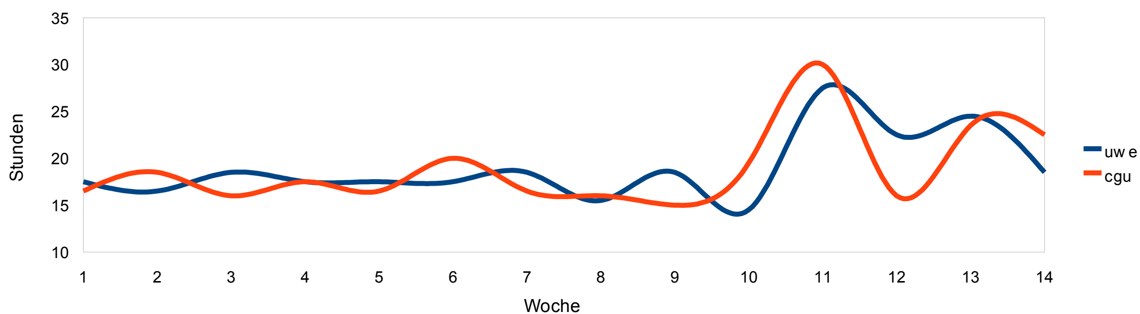


Abbildung 6.1: Verlauf des Zeitaufwands

Tabelle 6.2: Zeitaufwand pro Person

Wer	*Abkürzung*	*Zeitaufwand*
Urs Wegmann	**uwe**	265 h
Christopher Guntli	**cgu**	264 h

6.3.2 Zeitaufwand pro Disziplin

In Abbildung 6.2 ist der Zeitaufwand pro Disziplin aufgelistet. Es ist klar erkennbar, dass die meiste Zeit in die Implementation geflossen ist doch auch in die Dokumentation haben wir einiges an Zeit investiert. Die Zeitplanung haben wir jeweils eine Woche im Voraus gemacht und daher haben wir keine grösseren Abweichungen von Ist- und Sollzeit.

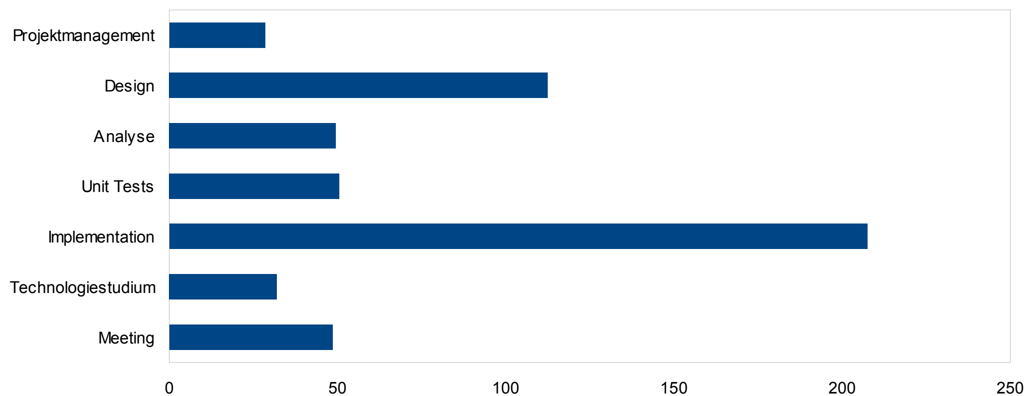


Abbildung 6.2: Zeitaufwand pro Disziplin

6.4 Probleme

Grundsätzlich wurde das Projekt ohne grösseren Komplikationen und Probleme beendet. Doch wie in jedem Projekt gibt es Punkte, bei denen nicht die gewünschte Lösung oder Kompromisse implementiert wurden.

6.4.1 AvalonDock

Bei AvalonDock mussten wir ein paar Unschönheiten was das Mapping des DataContext beim Herausziehen von Fenstern aus der Applikation betrifft feststellen. Da es aber ein Open Source Projekt auf CodePlex ist, konnten wir die fehlenden Funktionalitäten selber implementieren wodurch wir die Funktionalität des *AttributeProfile Designers* gewährleisten konnten. Geändert haben wir folgendes:

Tabs DataSource Die DataSource für die Tabs und Templates für die Darstellung der ViewModels als TabPanel musste selber implementiert werden. Da aber WPF hier schon eine gute Unterstützung bietet, war dies keine grosse Herausforderung.

DataContext in den Panels Wenn die Panels ihr Verhalten (Flyout, Floating oder Docking) änderten verloren sie ihren DataContext, dies wurde so geändert, dass der DataContext bei Änderungen am Panel gesichert und nach der Änderung neu gesetzt wird.

6.4.2 MVVM

Das MVVM-Pattern ist eine elegante Lösung für viele Aspekte einer Software, allerdings stösst man schnell auf Grenzen, wenn man die Events der UI-Controls verwenden will. So musste zum Beispiel der Code für Drag & Drop im Codebehind implementiert werden, was teilweise zu expliziten Casts des DataContext führte.

6.4.3 Mouse Events

Das Abfangen des richtigen MouseEvents mit dem Tunneling und Bubbling führte bei stark verschachtelten UI-Elementen wie etwa dem ShowTypePanel zu Problemen, da oft der Preview des Events verwendet werden musste um sicherzustellen, dass der Event auf dem richtigen Control abgefangen wurde. Insgesamt mussten wir ziemlich viele Mouse Events behandeln, was zum Teil etwas unübersichtlich war.

Kapitel 7

Benutzerdokumentation

7.1 Installation

Zu Installation des Programms muss lediglich die Installationsroutine gestartet und den Anweisungen gefolgt werden.

7.2 Ordnerstruktur

Bei der Installation wird folgende Ordnerstruktur erstellt:

Tabelle 7.1: Ordnerstruktur des Programmordners

Ordner	*Beschreibung*
/	Das Programm selber sowie die benötigten DLL[1]'s
/Templates	Vorgefertigte Elemente, welche ins Programm geladen werden können
/Templates/Attributes	Alle Attribut Templates welche sich als Attribute instantiieren lassen
/Templates/Profiles	Vorgefertigte Profiles welche dann erweitert werden können
/Templates/ShowTypes	ShowTypes welche zur instantiierung ausgewählt werden können

Die Ordnerstruktur kann jedoch über die Konfigurationsdatei ProfileDesigner.exe.config welche sich im Hauptverzeichnis befindet verändert werden. Dies kann hilfreich sein, wenn beispielsweise gemeinsame Templates auf einem Fileserver verwendet werden sollen.

7.2.1 Sicherheitsrichtlinien in Windows Vista und 7

Da Windows Vista und 7 nur mit Administratorenrechten den Zugriff auf den Programmordner (Program Files) zulassen, können die Templates über das Kontextmenu nicht direkt im Templatesordner gespeichert werden.

Es empfiehlt sich die Templates auf einen Netzwerkshare oder zumindest in einen Anderen Ordner zu verschieben. Für eine Lokale installation auf Windows 7 empfiehlt sich der *ProgramData* Ordner, da alle User darauf zugreifen können.

7.3 AttributeProfile

7.3.1 Erstellen

Ein AttributeProfile kann auf verschiedene Arten erstellt werden. Es gibt das *File* Menü in welchem sich der Menüpunkt *New* gewählt werden kann, um ein Neues AttributeProfile zu erstellen. Wer nicht gerne in Menüs klickt, hat die Möglichkeit das gleiche über eine Tastenkombination zu erledigen mit **Ctrl + N**.

Für Alle anderen gängigen Methode, welche sich zwar im Menü befinden, sind aber auch die Windows Standardtastenkombinationen implementiert.

Es können mehrere AttributeProfile gleichzeitig geöffnet werden. Bei mehreren geöffneten AttributeProfiles werden die verschiedenen geöffneten Instanzen in Form von Tabs im ShowTypePanel dargestellt. Tabs können auch aus der Anwendung herausgezogen werden, worauf sie in einem eigenen Fenster erscheinen (Abbildung 7.1).

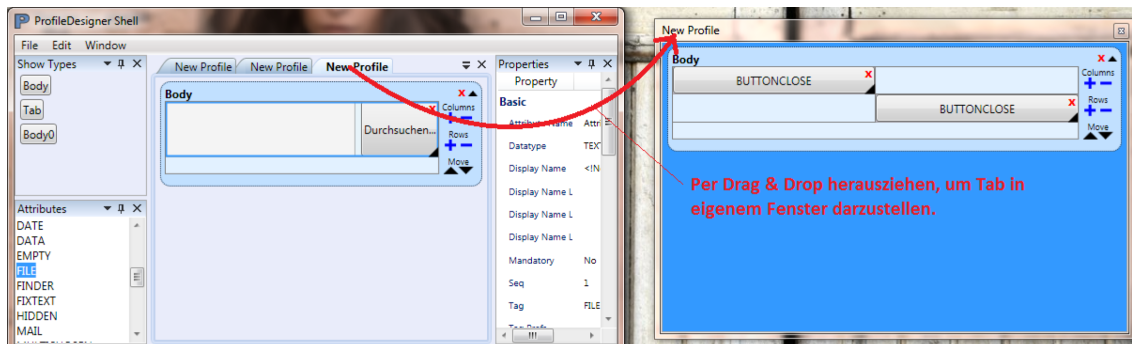


Abbildung 7.1: Tab per Drag & Drop aus der Anwendung ziehen um ihn in einem separaten Fenster darzustellen

7.3.2 Bearbeiten

Um die Properties des AttributeProfiles bearbeiten zu können, muss das AttributeProfile selektiert werden. Dazu muss man auf einen leeren Bereich im ShowTypePanel klicken (Abbildung 7.2), anschliessend werden die Properties des AttributeProfiles im PropertyPanel angezeigt.

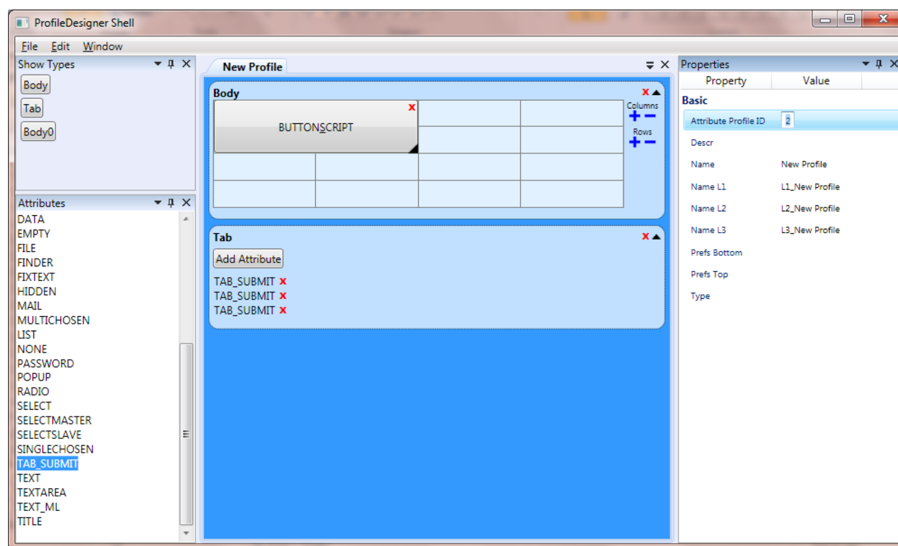


Abbildung 7.2: Bearbeiten der Properties eines AttributeProfiles

7.4 ShowType

Es gibt zwei verschiedene Arten, wie ein ShowType dargestellt werden kann (Abbildung 7.3): Einerseits die normale Anzeige, in der die Attribute im *AttributeGrid* an der entsprechenden Position angezeigt werden und andererseits die Anzeige für die *Tabs*, wo die Attribute in einer Liste untereinander dargestellt werden. Siehe dazu Abbildung 7.3. In der *Tab* Darstellung werden die Attribute anhand ihrer Sequenznummer (Property *Seq*) sortiert dargestellt.

7.4.1 Erstellen

Das Erstellen eines ShowTypes ist sehr einfach. Es muss einfach auf das gewünschte Show-Type Template geklickt werden, worauf der entsprechende ShowType dem momentan aktiven AttributeProfile hinzugefügt wird.

7.4.2 Bearbeiten

Um die Properties eines ShowTypes bearbeiten zu können muss dieser markiert sein. Dazu muss einfach auf den ShowType geklickt werden worauf dieser farbig hervorgehoben und die entsprechenden Properties im PropertyPanel angezeigt werden. Die Reihenfolge der ShowTypes im ShowTypePanel kann mit Hilfe der schwarzen Pfeile bei *Move* geändert werden. In der *AttributeGrid* Ansicht des ShowTypes sind ausserdem Kontrollelemente

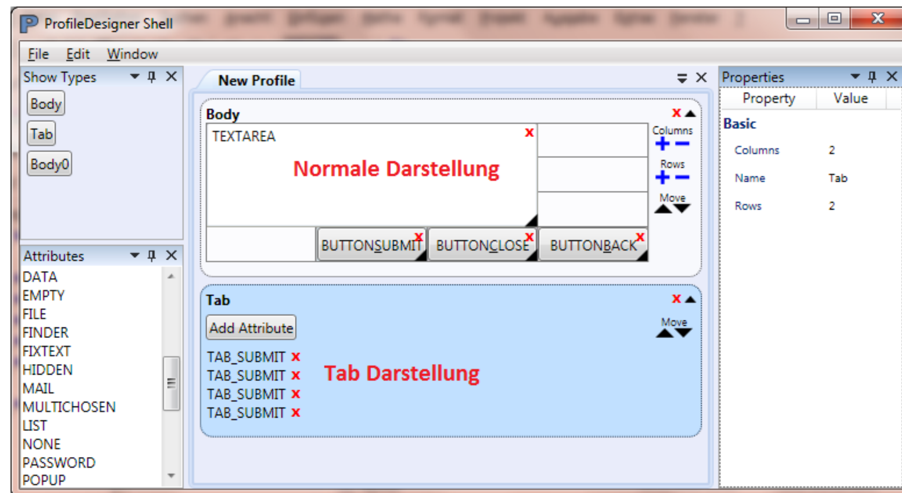


Abbildung 7.3: Die verschiedenen Ansichten eines ShowTypes

vorhanden, um dem *AttributeGrid* Spalten bzw. Zeilen hinzuzufügen oder zu entfernen (Abbildung 7.4).

Abbildung 7.4: Die Kontrollelemente der ShowTypes

7.5 Attribute

7.5.1 Erstellen

Um ein Attribute zu erstellen muss zuerst das Attribute Template gewählt werden, nach welchem das Attribute erstellt werden soll. In der *AttributeGrid* Darstellung muss das AttributeGrid auf genügend Spalten und Zeilen erweitert werden, damit das Attribute in den gewünschten Abmessungen platziert werden kann. Nun kann die gewünschte obere linke Zelle im AttributeGrid angeklickt und das Attribute durch Drag & Drop auf die gewünschte Grösse aufgezogen werden.

In der *Tab* Darstellung muss zu erzeugen des Attributes lediglich das gewünschte Template ausgewählt und anschliessend auf *Add Attribute* geklickt werden.

7.5.2 Bearbeiten

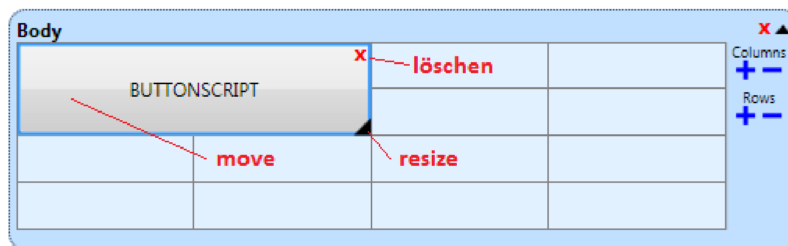


Abbildung 7.5: Bearbeiten eines Attributes

Diejenigen Properties welche die Platzierung im Grid sowie die Grösse des Attributes betreffen können intuitiv im GUI bearbeitet werden. Um das Attribut zu löschen muss das kleine rote Kreuz oben rechts gedrückt werden, mit dem schwarzen Pfeil unten links kann die Grösse geändert werden und zum Verschieben packt man das Attribut mit der Maus und verschiebt es an die gewünschte Position (Abbildung 7.5).

7.6 Templates

Die Templates für Attribute, ShowTypes und AttributeProfile können innerhalb des Programms erstellt werden. Um ein Template zu erstellen muss im Programm mit der rechten Maustaste auf das gewünschte Element geklickt werden und dann im Kontextmenü die Option *Save as Template* gewählt werden. Damit das Programm bei nächsten Starten die neuen Templates lädt, müssen diese im entsprechenden Ordner, wie im Kapitel Ordnerstruktur beschrieben, abgespeichert werden.

7.7 Copy & Paste

Um ein Element zu kopieren muss das Element durch Anklicken selektiert werden und dann entweder über das *Edit* Menü oder die Tastenkombination **Ctrl + C** der Kopiervorgang ausgelöst werden. Danach wird die XML Beschreibung des Elements in die Zwischenablage kopiert. Dies hat den Vorteil, dass das Element auch in einen normalen Texteditor eingefügt und dort bearbeitet werden kann. Den gleichen Effekt hat das Ausschneiden des Elementes (**Ctrl + X**) nur dass hierbei das originale Element gelöscht wird.

Um ein Element wieder einzufügen, muss dasjenige Element markiert werden, in das eingefügt werden soll. Um also ein Attribute einzufügen, muss der gewünschte ShowType markiert sein. Das Einfügen geschieht entweder über die Tastenkombination **Ctrl + V** oder das *Edit* Menü.

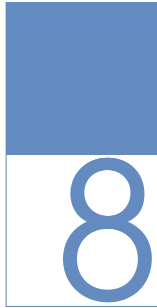
Elemente können innerhalb des gleichen oder zwischen verschiedenen AttributeProfilen hin und her kopiert werden.

7.8 Fensteranordnung

Die Anordnung der Fenster innerhalb des Programms kann frei per Drag & Drop verändert werden. Ausserdem können auch Fenster aus der Anzeige gelöscht werden. Wenn ein gelöschtes Fenster wieder angezeigt werden soll kann dies über das *Window* Menü erreicht werden. Ausserdem kann dort das Layout des Programms gespeichert und auch wieder geladen werden.

7.8.1 Persistenz der Anordnung

Beim Beenden der Applikation wird die aktuelle Fensteranordnung in einer Datei gespeichert, welche beim nächsten Start wieder ausgelesen wird. Somit kann man immer mit der gespeicherten Anordung weiter arbeiten, ohne jedesmal die Fenster neu anzuordnen zu müssen. Für den Fall, dass die Anordnung Korrupt oder föllig verstellt sein sollte, kann die default Anordnung wieder hergestellt werden. Die datei wird in den ApplicationData gespeichert. Bei Windows 7 ist dies der Ordner *%windir%/ProgramData*.



Kapitel 8

Persönliche Berichte

8.1 Urs Wegmann

Für mich war diese Arbeit ein positives Erlebnis. Anfangs war zwar lange nicht klar welche Arbeit wir bekommen würden und schlussendlich bekamen wir auch nicht unsere favorisierte Semesterarbeit. Doch es stellte sich dann schnell heraus, dass die Arbeit mit einem externen Industriepartner auch einige Vorteile mit sich brachte. So merkte man, dass die Mitarbeiter von TIE später wirklich mit dem von uns erstellten Programm arbeiten wollen und stellten daher auch alle benötigten Informationen immer schnell und Umfangreich zur Verfügung. Allgemein war die Zusammenarbeit mit TIE angenehm und interessant.

Auch die Betreuung von Herrn Huser und Herrn Schneble empfand ich als kompetent und motiviert. Das Feedback war meist positiv, was hoffentlich daran lag, dass wir uns auf dem richtigen Weg befanden. Allgemein verlief die gesamte Arbeit erstaunlich problemlos, was wir uns vom SE2 Projekt her eher weniger gewohnt waren.

Was mir persönlich im Gegensatz zum SE2 Projekt auch sehr gefallen hatte war die Tatsache, dass hier nicht die Dokumentation im Vordergrund stand sondern das Endprodukt. Jedoch konnten wir auch sehr aus den im SE2 Projekt gesammelten Erfahrungen profitieren und konnten gewisse Sachen vermeiden. So kümmerten wir uns etwa nicht schon zu Beginn zeitaufwendig um kleine Details welche später im Gesamtprojekt keine grosse Rolle mehr spielten. Insgesamt finde ich also sinnvoll, dass erst das SE2 Projekt, dann die Semesterarbeit und zuletzt die Bachelorarbeit durchgeführt wird.

Wie schon erwähnt war für mich die Semesterarbeit ein positives Erlebnis. Die Zusammenarbeit im Team wie auch diejenige mit den Betreuern und dem Kunden war angenehm und konstruktiv. Auch konnte ich durch diese Arbeit erstmals Erfahrungen

mit einem realen Kunden sammeln, was sehr wertvoll ist. Ich fühle mich nun gut vorbereitet auf die Bachelorarbeit und freue mich auf die Herausforderung.

8.2 Christopher Guntli

Da wir durch das SE2-Pojekt schon einiges an Erfahrung im Bereich WPF sowie dem Dokumentieren nach RUP in LaTeXsammeln konnten, waren wir bestens gerüstet für dieses Projekt und kamen ohne grösseren Komplikationen aus. Dies war nicht die erste Arbeit die wir in diesem Team erledigt haben, deswegen konnten wir auch ohne unnötigen Diskussionen uns auf das Ziel konzentrieren und speditiv arbeiten.

Die Erfahrungen aus dem SE2-Projekt halfen uns auch die Zeiten besser einschätzen zu können, somit wussten wir ziemlich gut, welche Teilschritte wie viel Aufwand bedeuteten und konnte so ohne böse Überraschungen oder Zeitdruck die geforderte Leistung erbringen.

Da wir hauptsächlich in der Schule entwickelten wussten wir immer über den Stand der Arbeit bescheid. Dies ermöglicht ein einfaches Entwickeln ohne komplizierte Tools.

Gegen Ende des Projekt wurde mein Teamkamerad leider krank und kam ein paar Tage nicht in die Schule. Dies war aber kein grösseres Problem, da ich über den Stand der Arbeit bescheid wusste. Somit konnte ich das Meeting in dieser Woche auch ohne ihn durchführen. Mittels *Instant Messenger* konnte ich den Verlauf des Meetings und die wichtigsten Punkte rasch mitteilen. Da mein Kamerad zwar krank aber nicht faul war, arbeitete er zu Hause weiter.

Der wichtigste Teil für das Gelingen der Arbeit war jedoch die Meetings in Aarau bei TIE. Vor dem ersten Meeting war mir nicht klar was genau von uns gefordert wurden. Doch im Verlauf der Meetings bekam ich ein klares Verständnis über das Problem und die erwartete Lösung. Durch weiteres Meeting konnten wir die Software noch genauer auf ihre Bedürfnisse anpassen.

Ich hoffe, dass die von uns gelieferte Arbeit nicht nur den Wünschen der Anwender entspricht, sondern auch weitergeführt wird und produktiv eingesetzt wird.

Literaturverzeichnis

[FB96] FRANK BUSCHMANN, Hans Rohnert Peter Sommerlad Mechael Stal, Regine Meunier: *Pattern - Oriented Software Architecture, A System of Patterns*, John Wiley & Sons Ltd (1996)

[Mac08] MACDONALD, Matthew: *Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5*, Apress, second edition Aufl. (2008)

[Smi09] SMITH, Josh: WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN* (2009), URL http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Abbildungsverzeichnis

1 Die fertige Applikation mit einem neu erstellten AttributeProfile xii

1.1 Objekte zum Beschreiben der Masken-Definitionen 1
1.2 Worflow für die Erstellung einer neuen Maske (bisher) 3
1.3 Angestrebter Workflow 4
1.4 Screenshot der Software 7
1.5 Detailansicht eines ShowTypes 8
1.6 Detailansicht der geladenen Templats 8

3.1 Domain Model 17
3.2 Paper Prototype 21
3.3 Paper Prototype: Show Type 23

4.1 Package Model 35

5.1 Übersicht über die Applikation 39
5.2 Mouseover auf einer leeren Zelle 39
5.3 Erstellen eines neuen Attributs 40
5.4 Skalieren eines bestehenden Attributes mit der Maus 40
5.5 Ein Beispiel einer ShowTypeView mit Tab Ansicht 43
5.6 Ein Beispiel einer ShowTypeView mit dem AttributeGrid 44
5.7 ShowTypeView mit Kontrollelementen auf der rechten Seite 45
5.8 Klassendiagramm der ViewModels 48
5.9 Klassendiagramm der Business Objekte 51
5.10 Deep Copy eines ShowTypes 52
5.11 Klassendiagramm XML Interface 55

6.1 Verlauf des Zeitaufwands 61
6.2 Zeitaufwand pro Disziplin 62

7.1 Tab per Drag & Drop aus der Anwendung ziehen um ihn in einem separaten Fenster darzustellen 66
7.2 Bearbeiten der Properties eines AttributeProfiles 67
7.3 Die verschiedenen Ansichten eines ShowTypes 68
7.4 Die Kontrollelemente der ShowTypes 68
7.5 Bearbeiten eines Attributes 69

Tabellenverzeichnis

2.1 Meilensteine 12
2.2 Ordnerstruktur des SVN Repositories 15

3.1 Properties des AttributeProfiles mit ihrem SQL Datentyp 18
3.2 Die am meisten verwendeten ShowTypes 18
3.3 Vordefinierte Tags 19
3.4 Meistverwendete Tags 19
3.5 Properties mit ihrem SQL Datentyp 20

6.1 Codeabdeckung 60
6.2 Zeitaufwand pro Person 61

7.1 Ordnerstruktur des Programmordners 65

Listings

1.1 Mögliche default.xml Datei für die Attribute 9
1.2 Attribute Template für das Password Attribute welches *default.xml* ergänzt 9

4.1 XML Schema 27
4.2 XML Beispiel 29

5.1 Verwendung des RelayCommands 45
5.2 Beispiel einer Gruppierung einer Collection 46
5.3 Xml Serialisierung 50
5.4 Verknüpfung zwischen ShowTypeViewModel-Collection und der ShowType-Collection 56
5.5 Verknüpfung zwischen ShowTypeViewModel-Collection und der ShowType-Collection 56

Glossar

ECMS Enterprise Content Management System

CMS Content Management System

SQL Structured Query Language

XML Extensible Markup Language

WPF Windows Presentation Fondation

SVN Subversion

HSR Hochschule für Technik Rapperswil

UI User Interface

GUI Graphical User Interface

MVVM Model View Viewmodel

XAML Extensible Application Markup Language

TIE The i-engineers

BLL Business Logic Layer

DLL Dynamic Link Library

WYSIWYG What You See Is What You Get