Masterarbeit

Studienrichtung Informatik
Fakultät für Informatik
Sommersemester 2010

Informationsvisualisierung von verteilten Anlagen

Jürgen Konrad

Betreuer: Ass.Prof. Dipl.-Ing. Dr.techn. Univ.-Doz. Denis Helic
Institut: Institut für Wissensmanagment

Kooperationspartner: NTE Naturenergie. Technology and Engineering GmbH
Betreuer: Dipl.-Ing. Andreas Hafellner

Zusammenfassung

Das Visualisieren und Überwachen von Anlagedaten ist eine bekannte Disziplin in der Informatik. Will man diese Aufgabe auf verteilte Anlagen jeglicher Art und Ausprägung ausdehnen und zusätzlich Endbenutzern die Möglichkeit geben, das Aussehen der Visualisierung und die damit verbundene Datenanbindung selbst zu bestimmen, sind spezielle Tools nötigt.

Diese Arbeit widmet sich dem Entwurf und der Entwicklung einer Applikation mit der diese Aufgabe generisch gelöst werden kann - einer Designumgebung für die Visualisierung und Fernsteuerung von verteilten Anlagen. Neben den Anforderungen und Problemstellungen dieser Aufgabe werden verschiedenen GUI Frameworks und mögliche Architekturkonzepte zur Entwicklung einer solchen Designumgebung theoretisch diskutiert. Der praktische Teil dieser Arbeit beschreibt die Implementierung einer Designumgebung und präsentiert die darin verwendete Konzepte und Lösungsansätze.

Abstract

Visualization and monitoring of facility data is a well recognized discipline of applied computer science. Requirements for software solutions are dependent on the characteristics of the facility, individual preferences of the end-user and furthermore, can change in time. To address these issues in a more generic way which offers the possibility to design, modify and extend the visualization at runtime appropriate tools are needed.

This thesis addresses requirements and solutions in the development of such a generic visualization design tool. Required features and resulting problems are identified and described. Different possible GUI frameworks and architectures for the development of such a design tool are presented and discussed theoretically, referring to the requirements and problems. In the practical part of this thesis the implementation of a design tool is described and the used approaches are presented.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Inhaltsverzeichnis

1 Motivation **6**

2 Einleitung **6**
2.1 Informationsvisualisierung . . . 6
2.2 Verteilte Anlagen . . . 7
2.3 Informationsvisualisierung physikalischer Daten . . . 7
2.4 Informationsvisualisierung abstrakter Daten . . . 8
2.5 Fernsteuerung von verteilten Anlagen . . . 9
2.6 Herausforderung . . . 9

3 Anforderungen an eine Designumgebung **9**
3.1 Elemente des Designers . . . 10
3.1.1 Toolbar . . . 10
3.1.2 Toolbox . . . 10
3.1.3 Workspace . . . 11
3.1.4 Property Grid . . . 11
3.2 Ausprägungen der Designeritems . . . 11
3.2.1 Dekorative Designeritems . . . 12
3.2.2 Designeritems zur Visualisierung . . . 12
3.2.3 Designeritems zur Fernsteuerung . . . 12
3.3 Anforderungen an die Designumgebung . . . 13
3.3.1 Anwendungsfälle der Designumgebung . . . 13
3.3.2 Technische Anforderungen . . . 18

4 Problemstellungen **19**
4.1 GUI Framework . . . 19
4.1.1 Komponentenbibliothek . . . 20
4.1.2 Erweiterbarkeit . . . 21
4.1.3 Grafische Möglichkeiten . . . 21
4.1.4 Stylemöglichkeiten . . . 21
4.1.5 Layoutmöglichkeiten . . . 22
4.1.6 Wiederverwendbarkeit . . . 22
4.1.7 Trennung Design / Logik . . . 22
4.1.8 Datenanbindung . . . 23
4.2 Softwarearchitektur . . . 23
4.2.1 Entwurfsmuster . . . 23
4.2.2 Generische Architektur . . . 24
4.2.3 Erweiterbarkeit . . . 24
4.2.4 Zusammengesetzte Anwendungen . . . 25

5 Technologien und Konzepte **25**
5.1 XML basierte Userinterfaces . . . 25
5.1.1 XUL . . . 26
5.1.2 MXML und FLEX . . . 30
5.1.3 XAML und WPF . . . 36
5.1.4 Vergleich . . . 44

5.2 UI Architekturkonzepte . . . 46
5.2.1 MVC . . . 47
5.2.2 MVP . . . 51
5.2.3 MVVM . . . 54
5.2.4 Fazit . . . 58

6 Implementierung in der NTE CLM Plattform 58
6.1 NTE CLM Überblick . . . 58
6.1.1 Technologieentscheidungen . . . 60
6.2 Designeritems . . . 61
6.2.1 Einfache Designeritems . . . 62
6.2.2 Zusätzliche Logik . . . 64
6.3 Workspace . . . 66
6.3.1 DesignerItemContentControl . . . 66
6.3.2 DesignerControl . . . 68
6.3.3 MVVM Architektur . . . 71
6.3.4 Zusätzliche Logik . . . 73
6.4 Toolbar . . . 75
6.5 Toolbox . . . 75
6.6 Property Grid . . . 77
6.7 Datenanbindung . . . 78
6.7.1 Datenbasis . . . 78
6.7.2 WCF Services . . . 79
6.7.3 Designeritem . . . 79
6.7.4 Propertygrid . . . 80
6.7.5 Datenupdate . . . 83
6.7.6 Fernsteuerung . . . 83
6.8 Gesamtintegration . . . 85
6.8.1 Prism . . . 86
6.8.2 Architektur . . . 86

7 Zusammenfassung 90

1 Motivation

Ziel dieser Arbeit ist es Umgebungen (in weiterer Folge Designumgebungen genannt) zu betrachten, die es einem Benutzer erlauben Informationsvisualisierungen zu erstellen.

Diese Designumgebungen sind in einigen Domänen zwingend notwendig wenn es nicht ausreicht vorgegebene Visualisierungen in deren Aussehen oder Position anzupassen.

Der erste Teil dieser Arbeit widmet sich den Konzepten die notwendig sind, um flexible Designumgebungen zur Informationsvisualisierung zu schaffen. Diese Konzepte werden nicht nur allgemein erörtert sondern auch bereits anhand von Anforderungen eines Praxisbeispiels ausgerichtet.

Im zweiten Teil wird eine konkrete Implementierung einer Designumgebung für die Visualisierung von verteilten Anlagen präsentiert und diskutiert.

2 Einleitung

Zunächst werden die Begriffe Informationsvisualisierung und verteilte Anlagen bzw. die sich dahinter befindlichen Ideen abgesteckt.

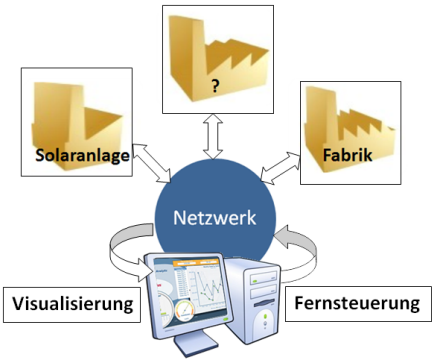


Abbildung 1: Visualisierung und Fernsteuerung

2.1 Informationsvisualisierung

Unter Informationsvisualisierung versteht man die Repräsentation abstrakter Daten, die nicht notwendigerweise einen physikalischen Bezug aufweisen. [1]

Über diese Definition bzw. anhand der zu Grunde liegenden Daten kann man folgende Kategorien ableiten:

- Informationsvisualisierung physikalischer Daten
- Informationsvisualisierung abstrakter Daten

Im speziellen wird in dieser Arbeit die Informationsvisualisierung von verteilten Anlagen betrachtet.

2.2 Verteilte Anlagen

Verteilte (technische) Anlagen

- sind eine Zusammenstellung von Maschinen/Apparaten/Geräten die einem bestimmten Zweck dienen
- besitzen selbst die Logik für die Ausführung ihrer notwendigen Prozesse
- sind durch ein Netzwerk über eine definierte Schnittstelle verbunden
- und können über diese definierte Schnittstelle überwacht und ferngesteuert werden

Dies reicht von Solaranlagen, Ampelsteuerungen bis hin zu Industriefließbändern, sofern diese Einrichtungen über eine Schnittstelle angesprochen werden können.

Verteilte Anlagen liefern über eine Netzwerkschnittstelle Daten aus einem Set von Datentypen, die je nach Wesen der Anlage in einem physikalischen Kontext stehen oder abstrakt sind.

2.3 Informationsvisualisierung physikalischer Daten

Durch die Visualisierung physikalischer Werte wie z.B. Temperatur, Geschwindigkeit oder Kraft von/auf Objekten wird versucht unsere Umwelt bzw. Teile davon nachzubilden: Bsp. ein Thermometer als Visualisierung des Zahlenwertes einer Temperatur.

Ein Beispiel für die Visualisierung von physikalischen Werten aus Prozessen ist das Open Source Paket Proview [2].

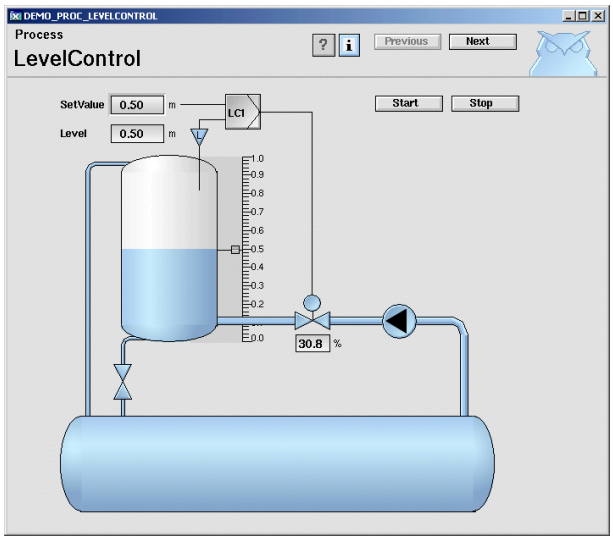


Abbildung 2: Proview Visualisierung

Die Visualisierung in Abbildung 2 kann vollständig vom Benutzer mit Hilfe einer Designumgebung erstellt werden. Proview ist ein Vertreter der im Allgemeinen unter dem Begriff SCADA (Supervisory Control and Data Acquisition) bekannten Systemen zur Überwachung und Steuerung von technischen Prozessen.

2.4 Informationsvisualisierung abstrakter Daten

Neben physikalischen Daten deren Visualisierung sich naturgemäß an der Abbildung der Umwelt orientiert, gibt es auch Daten deren Visualisierung kein entsprechendes Pendant in der Umwelt haben.

Für diese abstrakten Daten muss die entsprechende Visualisierungsform erst "erfunden" werden, in dem man aus einer Menge von Visualisierungsmöglichkeiten eine bestimmte auswählt und diese entsprechend konfiguriert.

Designumgebungen mit denen man diese Art von Informationsvisualisierungen erstellen kann sind sogenannte MVE's (Modular Visualization Environments) [3]. Ein Vertreter dieser Kategorie ist das Open Source Tool OpenDX [4].

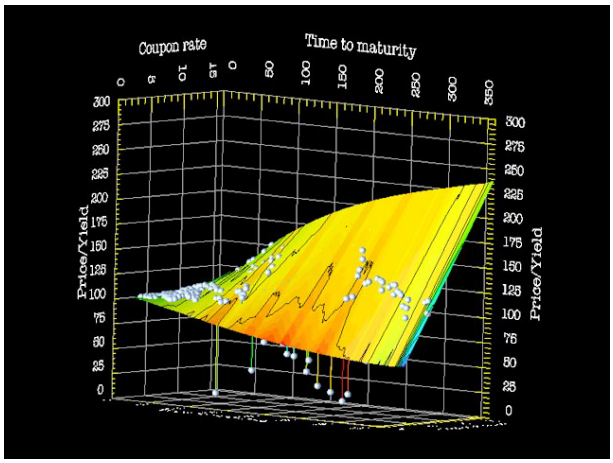


Abbildung 3: OpenDX Visualisierung

Mit Hilfe eines speziellen Tools (OpenDX Visualization Data Explorer) können zur Laufzeit abstrakte Daten an visuelle Repräsentationen angebunden und angezeigt werden.

Beide vorgestellten Softwarepakete haben eines gemeinsam - es handelt sich hierbei um Designumgebungen die dieselben Basisdatentypen als Eingang haben - aber auf unterschiedliche Weise visualisieren.

2.5 Fernsteuerung von verteilten Anlagen

Oft ist bei verteilten Anlagen nicht nur die Informationsvisualisierung sondern auch die Fernsteuerung erforderlich. GUI Designer, bekannt aus Softwareentwicklungsumgebungen wie z.B. Visual Studio, sind das Vorbild für die flexible Erstellung von Oberflächen mit denen Benutzer interagieren können.

Die Anbindung an Daten erfolgt hier aber über Programmcode und ist für normale Benutzer nicht einfach anwendbar - abgesehen vom Kompilieraufwand der eine Erstellung von Benutzeroberflächen zur Laufzeit außen vor lässt.

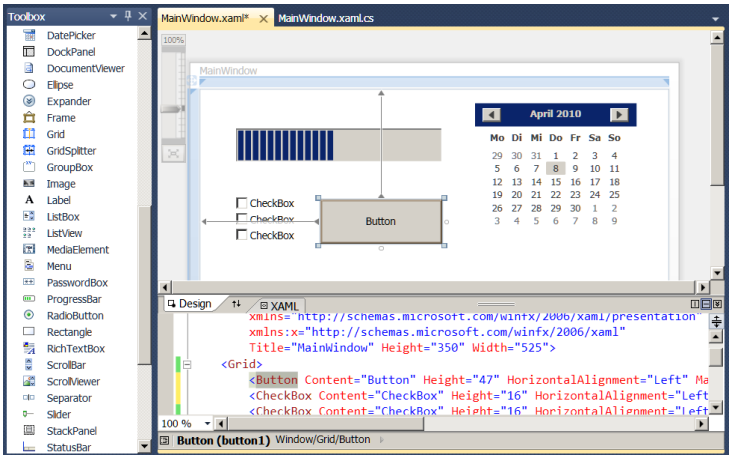


Abbildung 4: Visual Studio

GUI Designer aus Softwareentwicklungsumgebungen bieten auch nicht unbedingt die richtigen Elemente die für die Fernsteuerung von technischen Anlagen notwendig sind. Gegenstücke physikalischer Schaltelemente (z.B. Drehknöpfe, Schalter, Schieberegler) findet man hier meist nicht und wenn dann meist nur in einer visuell nicht ansprechenden Form.

2.6 Herausforderung

Alle drei aufgezeigten Disziplinen: die Informationsvisualisierung physikalischer Daten, abstrakter Daten und die Fernsteuerung verteilter Anlagen gilt es durch eine einzelne Designumgebung abzudecken. Die Anforderungen dafür werden im folgenden Kapitel aufgezeigt.

3 Anforderungen an eine Designumgebung

In diesem Kapitel werden allgemein die Anforderungen an eine Designumgebung zur Informationsvisualisierung und Fernsteuerung von verteilten Anlagen (in weiterer Folge auch Designer genannt) beschrieben.

Die hier beschriebenen Anforderungen wurden zusammen mit der Firma NTE Systems erhoben und spiegeln die Anforderungen an ein Realsystem wieder, das in weiterer Folge implementiert werden soll.

3.1 Elemente des Designers

Die hier vorgestellte Struktur orientiert sich an gängigen Userinterface Designumgebungen wie z.B. Visual Studio. Abbildung 5 zeigt schematisch den Grundaufbau des Designers.

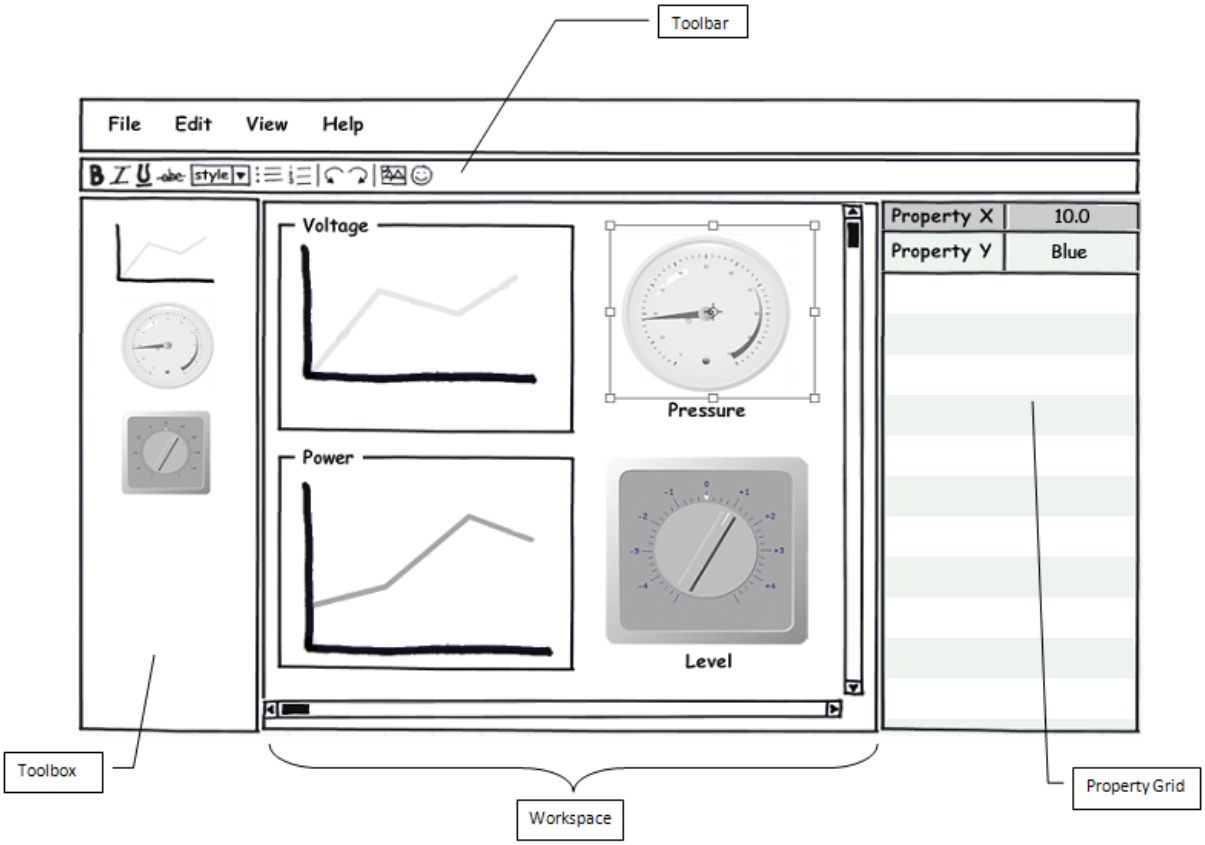


Abbildung 5: Grundstruktur des Designers

3.1.1 Toolbar

Die Schaltflächen in der Toolbar dienen als Shortcuts für Aktionen die auf Objekte im Workspace (in weiterer Folge Designeritems genannt) ausgeführt werden können. Dies sind Aktionen wie z.B. Copy, Paste, Cut, Delete, Zoom In, Zoom Out oder Aktionen die dazu dienen den Z-Index, oder die Position von Objekten zu verändern.

3.1.2 Toolbox

In der Toolbox befinden sich Objekte die auf den Workspace platziert werden können. Um schneller das benötigte Designeritem zu finden sind diese in Kategorien (Bsp.: Anzeigen, Knöpfe, Solartechnik ...) eingeteilt.

3.1.3 Workspace

Auf der Arbeitsfläche lassen sich Designeritems platzieren, selektieren und auch einige Eigenschaften (Größe) können hier direkt manipuliert werden. Die Designeritems können auf einem Raster ausgerichtet und zur besseren Usability auch gruppiert werden. Der Workspace vergrößert sich bei Bedarf (Scrollmöglichkeit) und ist zoomfähig.

3.1.4 Property Grid

Mit Hilfe des Property Grids lassen sich Eigenschaften die ein Designeritem besitzt manipulieren. Der Property Grid zeigt in der linken Spalte die Namen der Properties an und in der rechten Spalte den Wert des Properties.

Hauptzweck der Designumgebung ist es Designeritems zu selektieren, zu platzieren und deren Eigenschaften zu manipulieren. Auf Designeritems wird nun genauer eingegangen.

3.2 Ausprägungen der Designeritems

Designeritems sind die Basisbausteine einer Visualisierung - sie stehen im Mittelpunkt der Anwendung. Designeritems

- besitzen ein **Erscheinungsbild**
- haben Eigenschaften (**Properties**)
- besitzen eine **Logik**

Das Erscheinungsbild des Designeritems wird über die Logik manipuliert. Da es sich bei den Designeritems nicht um starre Anzeigen handelt greift die Logik auf sogenannte Properties zu. Mit Hilfe dieser Properties wird aber nicht nur das Erscheinungsbild parametrisiert, es werden auch Daten von verteilten Anlagen an die Logik angebunden.

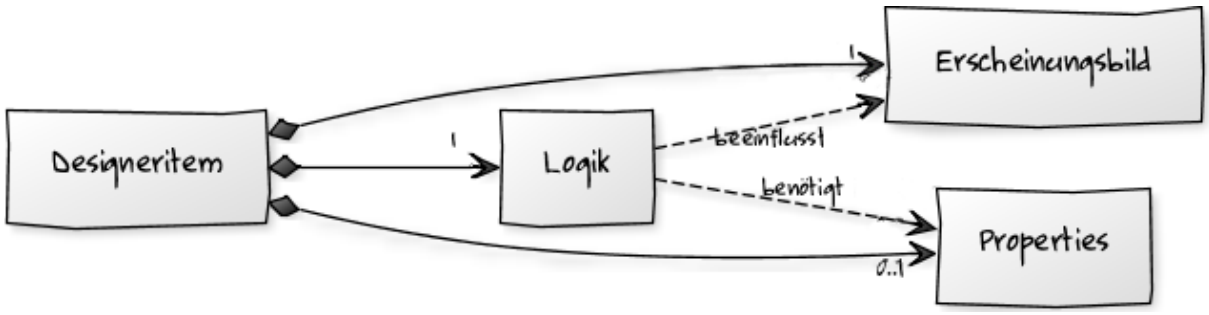


Abbildung 6: Logische Struktur eines Designeritems

Beispiele für Properties sind ein Farbwert, ein Zahlenwert oder z.B. ein textueller Wert. Properties können sowohl fixe Werte als auch Werte, die von verteilten Anlagen übermittelt werden, annehmen. Werte die von verteilten Anlagen übermittelt werden können in manchen Fällen nicht nur gelesen sondern geschrieben werden. Auf Basis der Properties eines Designeritems und deren Ausprägungen können Designeritems in drei Klassen eingeteilt werden wobei auch Mischungen erlaubt sind:

3.2.1 Dekorative Designeritems

Dekorative Designeritems dienen lediglich der Übersichtlichkeit. Mit Hilfe von Rahmen, Beschriftungen, Bildern und ähnlichen grafischen Elementen kann der Workspace optisch aufpoliert bzw. übersichtlicher gestaltet werden. Dekorative Designeritems besitzen nur Properties die einen fixen Wert annehmen können. In den meisten Fällen werden solche dekorativen Designeritems ohne Logik auskommen, da die Properties direkt im Erscheinungsbild wiedergegeben werden.

3.2.2 Designeritems zur Visualisierung

Diese Designeritems dienen zur direkten Anzeige von Werten die von verteilten Anlagen übermittelt werden (z.B. ein Thermometer). Sie können zur Parametrisierung des Erscheinungsbildes (z.B. Thermometer min, Thermometer max) auch fixe Properties haben, sind aber dadurch charakterisiert, dass sie mindestens ein Property haben, das an den Wert einer verteilten Anlage angebunden ist.

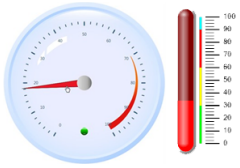


Abbildung 7: Designeritems zur Visualisierung

3.2.3 Designeritems zur Fernsteuerung

Designeritems zur Fernsteuerung haben die spezielle Eigenschaft, dass die Logik nicht nur Properties liest, sondern auch Properties setzt. Somit können Werte erzeugt werden, die wiederum an die verteilten Anlagen weitergegeben werden können. Beispiele für Designeritems zur Fernsteuerung sind Drehknöpfe, Schalter oder auch Slider. Natürlich werden Designeritems zur Fernsteuerung auch implizit zur Visualisierung verwendet, in dem der aktuelle Wert direkt angezeigt wird.

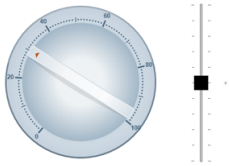


Abbildung 8: Designeritems zur Fernsteuerung

3.3 Anforderungen an die Designumgebung

Nachdem die einzelnen Elemente des Designers bekannt sind und die Basisbausteine (= Designeritems) eingeführt wurden, werden hier detaillierter Anforderungen an die Designumgebung aufgelistet.

3.3.1 Anwendungsfälle der Designumgebung

Die Designumgebung muss folgende minimale Usecases erfüllen, um dem Benutzer die Möglichkeit zu geben, Visualisierungen zu erstellen.

Platzieren eines Designeritems

Ablauf

Ein Designeritem wird aus der Toolbox ausgewählt und mittels "Drag and Drop" auf den Workspace gezogen.

Ergebnis

Das Designeritem wird automatisch selektiert und dessen Eigenschaften im Propertygrid angezeigt.

Hinweise

Ist ein Raster aktiviert, so wird das Designeritem dementsprechend am Raster ausgerichtet.

Auswahl eines Designeritems

Abbildung 9: Selektiertes Designeritem

Ablauf

Durch Linksklick auf ein Designeritem wird dieses selektiert.

Ergebnis

Die Selektion wird durch eine Umrandung / Hervorhebung des Designeritems (siehe Abbildung 9) angezeigt und die Eigenschaften des Designeritems werden im Propertygrid angezeigt.

Auswahl mehrerer Designeritems

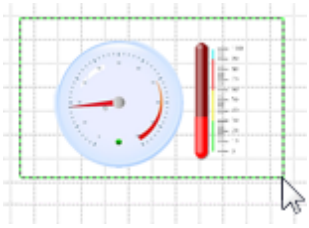


Abbildung 10: Rubberband Selektion

Ablauf

Durch Aufziehen eines Rechteckes (siehe Abbildung 10) mit gehaltener linker Maustaste können mehrere Designeritems selektiert werden. Alternativ können einer Selektion durch Linksklick und halten der Ctrl Taste zusätzlich Designeritems hinzugefügt werden.

Ergebnis

Die Selektion wird durch eine Umrandung / Hervorhebung aller selektierten Designeritems angezeigt und die gemeinsamen Eigenschaften der Designeritems werden im Propertygrid angezeigt.

Verschieben eines oder mehrere Designeritems

Ablauf

Beim überfahren eines oder mehrerer selektierter Designeritems wechselt der Mauszeiger auf  . Wie von anderen Applikation gewohnt, können durch Linksklick und bewegen der Maus die selektierten Elemente verschoben werden. Alternativ können selektierte Designeritems auch mit den Cursorpfeilen verschoben werden.

Ergebnis

Die Designeritems befinden sich an der neuen Position.

Hinweise

Ist ein Raster aktiviert, so werden die verschobenen Designeritem dementsprechend am Raster ausgerichtet.

Einblenden eines Aktionsmenüs

Ablauf

Durch Rechtsklick auf ein selektiertes Designeritem, oder den Workspace öffnet sich ein Aktionsmenü in dem Aktionen wie z.B. Kopieren, Einfügen, Entfernen angewendet werden können.

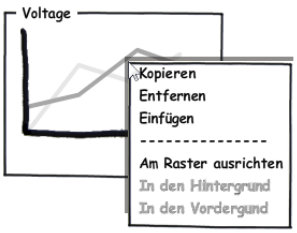


Abbildung 11: Aktionsmenü

Ergebnis

Das Aktionsmenü wird eingeblendet. Aktionen die nicht ausgeführt werden können sind ausgegraut.

Hinweise

Einige der Aktionen sind auch zusätzlich über die Toolbar erreichbar.

Kopieren/Einfügen/Entfernen von Designeritems

Ablauf

Diese Aktionen werden mittels Aktionsmenü, Toolbar oder Shortcut (Ctrl-C, Ctrl-V, Entf) initiiert.

Ergebnis

Die Aktion wurde ausgeführt und der Workspace aktualisiert.

Hinweise

Einige Aktionen werden nicht nur auf das aktuelle Designeritem angewandt, sondern auf alle selektieren.

Größenveränderung von Designeritems

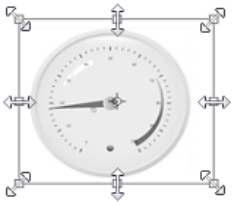


Abbildung 12: Größenveränderung

Ablauf

Ein selektiertes Designeritem kann in der Größe verändert werden. Beim Überfahren eines Designeritems mit der Maus, erscheint, wie von ähnlichen Applikation gewohnt, ein Mauszeiger der die Größenänderung anzeigt (Abbildung 12). Durch Linksklick und bewegen der Maus wird die Größe des Designeritems verändert.

Ergebnis

Alle selektierten Designeritems werden in der Größe verändert. Ist ein Raster aktiviert und ist eine Neuausrichtung notwendig, so wird diese auch durchgeführt.

Hinweise

Designeritems können die Eigenschaft besitzen, dass sie nicht in ihrer Größe verändert werden dürfen. Sie unterscheiden sich visuell von den anderen dadurch, dass die Vergrößerungspunkte fehlen.

Gruppieren mehrerer Designeritems

Ablauf

Mehrere Designeritems können optisch und logisch zu einem einzelnen Designeritem verschmolzen werden in dem auf selektierte Designeritems über das Aktionsmenü die Aktion "Gruppieren" aufgerufen wird.

Ergebnis

Das so erzeugte neue Designeritem verhält sich wie ein einzelnes. Eine Gruppierung kann durch die Aktion "Gruppierung auflösen" wieder rückgängig gemacht werden.

Hinweise

Werden Aktionen wie z.B. eine Größenveränderung auf ein gruppiertes Designeritem durchgeführt, so werden diese Aktionen automatisch auf alle Designeritems in der Gruppe durchgeführt.

Änderung der Z-Ebene von Designeritems

Ablauf

Verdeckt ein Designeritem ein anderes, und soll das darunterliegende in der Z-Ebene näher zum Betrachter gebracht werden, so können Manipulationen über das Aktionsmenü durchgeführt werden. "In den Vordergrund" bringt das selektierte Designerelement an die oberste Position, "Vorwärts" um eine Ebene höher und "In den Hintergrund" sowie "Rückwärts" verhalten sich analog in die andere Richtung.

Ergebnis

Veränderungen der Z-Ebene von Designeritems werden sofort angezeigt - d.h. der Benutzer bekommt unmittelbar Feedback über seine Aktion.

Hinweise

Designeritems werden in der Reihenfolge in der sie auf den Workspace gelangen dargestellt - d.h. ein Designeritem das später als ein anderes mittels Drag und Drop auf den Workspace platziert wurde, verdeckt jenes das früher platziert wurde.

Manipulation eines Designeritem Properties

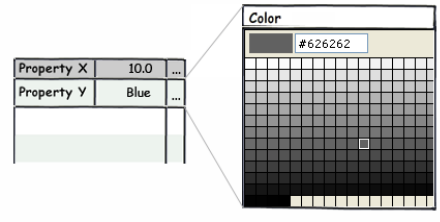


Abbildung 13: Manipulation eines Properties

Ablauf

Designeritems verfügen über Eigenschaften die sie in ihrer Darstellung beeinflussen. Wird ein Designeritem selektiert, so werden seine Eigenschaften im Propertygrid angezeigt. Jedes Property kann durch einfachen Klick editiert werden - es öffnet sich je nach Datentyp der entsprechende Editor.

Ergebnis

Nach der Eingabe des neuen Eigenschaftwertes wird dieser sofort übernommen und das Designeritem ändert entsprechend seiner Logik und des Wertes der Eigenschaft seine Gestalt.

Hinweis

Nicht jeder Datentyp verfügt über einen eigenen Editor im Propertygrid - Basisdatentypen wie z.B. Zahlen werden über einfache Textfelder eingegeben.

Anbindung eines Designeritem Properties an eine Datenquelle

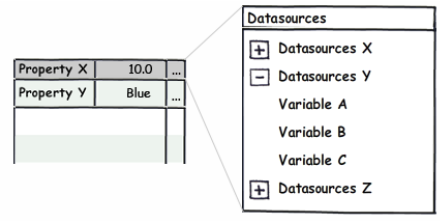


Abbildung 14: Anbindung an eine Datenquelle

Ablauf

Ist eine Eigenschaft eines Designeritems intern als Schnittstelle zu einer Datenquelle markiert, z.B. eine Temperaturanzeige hat eine Temperatur Eigenschaft, so öffnet sich beim Editieren des Properties nicht der dem Datentyp entsprechende Editor, sondern der Benutzer kann aus einer Reihe von Datenquellen auswählen die in weiterer Folge den Wert des Properties bestimmen.

Ergebnis

Bei der Eigenschaft wird die Verknüpfung zur Datenquelle gespeichert - ändert sich der Wert aus der Datenquelle, so wird diese Werteänderung an das Designeritem weitergegeben.

Hinweis

Umgekehrt ist es auch möglich, dass ein Designeritem auf eine Datenquelle schreibt - ein Drehregler z.B. schreibt einen Wert der mit einer Datenquelle verknüpft ist - in diesem Fall wird der Datenquelle der neue Wert mitgeteilt.

Laden/Speichern der Visualisierung

Ablauf

Im Topmenü können bestehende Visualisierungen in die Designumgebung geladen oder abgespeichert werden.

Ergebnis

Je nach Aktion wir die geladene Visualisierung dargestellt bzw. abgespeichert.

3.3.2 Technische Anforderungen

Zusätzlich zu den zuvor aufgelisteten Anwendungsfällen gibt es folgende technische Anforderungen an die Designumgebung bzw. deren Komponenten.

Modularität und Erweiterbarkeit

Die Grundidee der Designumgebung ist es verteilte Anlagen verschiedenster Ausprägungen zu visualisieren und fernzusteuern. Aus diesem Grund darf die Lösung keine Domänen ausgrenzen und muss modular aufgebaut sein. Die einfache Erweiterbarkeit der Designumgebung durch neue Designeritems oder Kombinierung bestehender Designeritems sind die Basis um diese Grundidee zu verwirklichen.

Toolbox Pluginfunktion

Die Toolbox bzw. die in der Toolbox enthaltenen Designeritems sollen nicht starr implementiert sein, sondern über einen Pluginmechanismus hinzugefügt werden können. D.h. Designeritems können jederzeit, z.B. durch kopieren eines Kompilates oder ähnlichem in einen definierten Ordner, automatisch eingebunden werden und scheinen danach in der Toolbox der Designumgebung auf.

Einfaches erstellen von Designeritems

Es soll möglich sein, z.B. durch implementieren eines Interfaces oder ableiten von einer vorgegebenen Klasse, geneigten Benutzern die Möglichkeit zu geben, selbst Designeritems zu schreiben. Das System soll somit für selbst geschriebene Designeritems offen sein - auf

Modularität und Erweiterbarkeit wird somit großer Wert gelegt. Zusätzlich soll es auch möglich sein für Nicht-Programmierer, durch Kombination bestehender Designeritems nach dem Baukastenprinzip, neue Designeritems zu erschaffen (Bsp. Bild + Gauge = Boiler).

Umgestalten von Designeritems

Wie bereits vorgestellt, können Designeritems von der Logik her identisch sein, sich aber durch die visuelle Darstellung stark unterscheiden (z.B.: ein Thermometer und eine Anzeige die den aktuellen Füllstand wiedergibt). Um Logik von bestehenden Designeritems weiterverwenden zu können und die Möglichkeit zu haben, die Darstellung separat anpassen zu können, muss die visuelle Repräsentation von der Logik entkoppelt sein. Dies bietet außerdem die Möglichkeit visuelle Anpassungen auch von Nicht-Softwareentwicklern (z.B. Designern) durchführen zu lassen.

Datenquellen

Datenquellen sind die Schnittstelle zur Anlage und besitzen jeweils einen eindeutigen Namen. Die Designumgebung wird mit Web-Services versorgt, die sie benutzen kann, um vorhandene Datenquellen abzurufen und Daten von diesen Datenquellen zu bekommen.

Datentypen der Datenquellen

Die Datenquellen die zur Verfügung gestellt werden, und die in weiterer folge mit Eigenschaften von Designeritems verknüpft werden können, beschränken sich auf folgende Datentypen: Integer, String, Real, DateTime, Blob, Bool und Arrays davon. Die Designumgebung hat mit den zur Verfügung gestellten Services Zugriff auf die dahinterliegenden Daten und gibt diese an die Designeritems weiter, welche die Visualisierung selbst durchführen. Zusätzlich existiert auch der umgekehrte Weg: Eigenschaften die ein Designeritem verändert sollen auch als Änderungen in eine Datenquelle zurück fließen (Fernsteuerung).

4 Problemstellungen

In diesem Kapitel werden Problemstellungen, die sich aus dem Anforderungen aus dem vorigen Kapitel ergeben, aufgeführt und kurz diskutiert.

4.1 GUI Framework

Die Wahl des richtigen GUI Frameworks und dessen optimale Verwendung ist wohl der wichtigste Punkt bei der Umsetzung der Designumgebung. In der modernen GUI Entwicklung ergeben sich folgende Problemstellungen:

Ein möglichst flexibler GUI Framework hat folgende Aufgabenstellungen bestmöglich zu erfüllen:

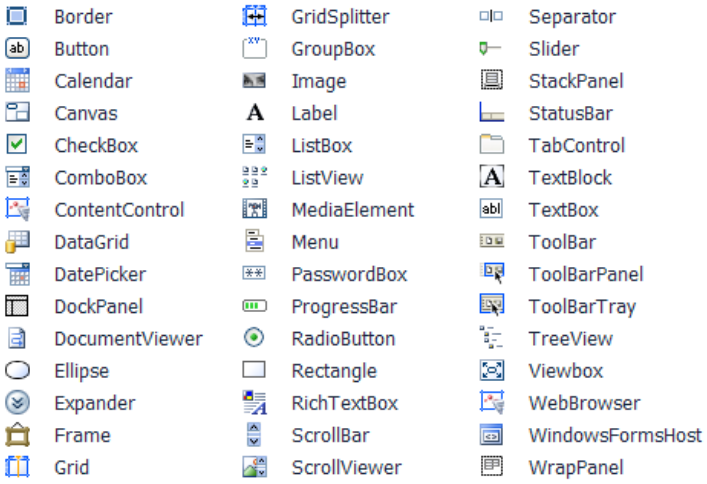


Abbildung 15: WPF Komponentenbibliothek

4.1.1 Komponentenbibliothek

Die Basis eines GUI Frameworks sind die verfügbaren Komponenten. Umso mehr Basiskomponenten verfügbar sind, desto besser die Verwendbarkeit des Frameworks.

Die meisten GUI Frameworks bieten 30 bis 40 Basiskomponenten welche durch Attribute und Optionen angepasst werden können. Diese Attribute wie z.B. Größe, Farbe, Textgröße können sowohl bei der Erzeugung als auch zur Laufzeit verändert werden [5].

```
Button btn1 = new Button();
btn1.Background = Brushes.LightBlue;
```

Listing 1: Programmatisches Setzen eines Attributes eines WPF [6] Buttons

Listing 1 zeigt das programmatische Setzen eines Attributes eines Buttons. Dies ist von der Funktionalität ausreichend, für die Wartbarkeit jedoch sehr schlecht, da hier Aussehen innerhalb des Codes definiert wird - nur ein Programmierer hat die Möglichkeit den Wert zu verändern und eine Neukompilierung ist ebenfalls notwendig. Besser ist die deklarative Definition von Komponenten wie sie Listing 2 zeigt.

```
<Button Name="btn1" Background="LightBlue" />
```

Listing 2: Deklaratives Setzen eines Attributes eines WPF Buttons

Im Zweiteren Fall ist das Aussehen vom Code getrennt, so dass auch Nicht-Programmierer (mit entsprechenden Tools oder Editoren) die Möglichkeit haben, das Aussehen anzupassen.

4.1.2 Erweiterbarkeit

Fast noch wichtiger als die Anzahl der verfügbaren Basiskomponenten, ist die Frage der Erweiterbarkeit. Erlaubt das Framework selbstgeschriebene Komponenten? Kann man bestehende Komponenten erweitern? Umso einfacher diese beiden Aufgaben gelöst werden können, desto höher wiederum die Akzeptanz bei den Entwicklern.

4.1.3 Grafische Möglichkeiten

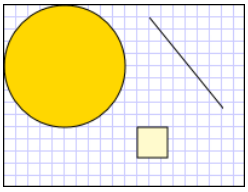


Abbildung 16: WPF Drawing

Ein weiteres entscheidendes Kriterium eines GUI Frameworks sind seine grafischen Zeichenmöglichkeiten. Sollen Charts wie Balkendiagramme oder ähnliche Dinge visualisiert werden, so benötigt man zumindest einfache Zeichenfunktionen.

Noch besser ist die Unterstützung von Standard Shapes wie z.B. Linien, Kurven und andere geometrische Formen. Abgerundet werden die Zeichenmöglichkeiten durch mögliche Darstellung von 3 D Inhalten und durch mögliche Transformationen.

4.1.4 Stylemöglichkeiten

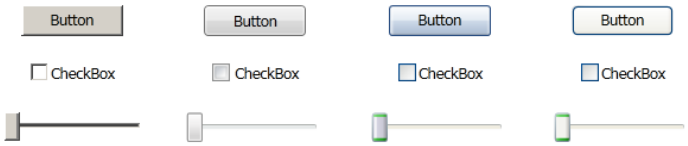


Abbildung 17: WPF Themes

Neben der Anpassbarkeit / Erweiterbarkeit von GUI Komponenten ist das anpassbare Aussehen von Komponenten ein wichtiges Thema. Hier sind nicht die Attribute bzw. Methoden gemeint, die einer Komponenten ihr Aussehen geben, sondern das zum Betriebssystem dazugehörige "Look and Feel" bzw. Themes die auf eine GUI angewandt werden können. GUIs die auf verschiedenen Betriebssystemen laufen, sollen auch genau so aussehen, wie die anderen Anwendungen die auf dem System laufen. Zusätzlich sind selbsterstellte Themes erwünscht.

Als Nebeneffekt von style baren Komponenten können diese ein völlig neues Aussehen bekommen, so dass z.B. aus einem herkömmlichen Button oder einer Checkbox ein Schalter oder Taster wird, der als Visualisierung zur Fernsteuerung verwendet werden kann.

4.1.5 Layoutmöglichkeiten

Komponenten müssen innerhalb einer GUI platziert werden können - die exakte Positionierung mittels Koordinaten erweist sich als nicht immer sinnvoll. Anwendungen sind z.B. in ihrer Größe veränderbar und auch die Größe einzelner Komponenten kann auch zur Laufzeit variieren. Aus diesem Grund sind flexible Layoutstrategien gefragt, die sich entsprechend ihrer beinhaltenden Elemente und deren Größe optimal verhalten.

```
Canvas cnv1 = new Canvas();
Button btn1 = new Button();
cnv.SetLeft(btn1, 50);
cnv.SetTop(btn1, 30);
```

Listing 3: Absolute Positionierung eines WPF Buttons auf einen Canvas

Aktuelle GUI Frameworks bieten verschiedene Layoutmöglichkeiten oder auch im optimalen Fall Möglichkeiten auf den Layoutprozess Einfluss zu nehmen.

4.1.6 Wiederverwendbarkeit

Ein weiteres wichtiges Kriterium für die Auswahl des GUI Frameworks bzw. der GUI Technologie ist die Art und Weise wie Wiederverwendbarkeit unterstützt wird. Können GUIs auf anderen Betriebssystemen wiederverwendet werden? Können GUIs in anderen Programmiersprachen angesprochen / verwendet werden?

4.1.7 Trennung Design / Logik

Die Trennung von Design und Logik soll durch den GUI Framework so gut wie möglich unterstützt werden. Folgende Beispiele sollen diese Problematik verdeutlichen:

```
Button btn1 = new Button();
btn1.Click  += new RoutedEventHandler(OnButton1Click);

void OnButton1Click(object sender, RoutedEventArgs e)
{
    //do something when the button was clicked
}
```

Listing 4: Programmatische Click Event Anbindung an einen Button

Listing 4 zeigt, dass durch programmatisches Verknüpfen des Button Click Events mit einer Funktion die GUI mit der Logik sehr eng gekoppelt ist. Listing 5 zeigt eine verbesserte Version:

```
<Button Name="btn1" Click="OnButton1Click" />
```

Listing 5: Deklarative Click Event Anbindung eines WPF Button

Wünschenswert wäre es, den Methodennamen ebenfalls wegzukapseln, so dass wirklich eine lose Kopplung zwischen der Anzeige und der Logik herrscht. Einige GUI Frameworks bieten hierfür ebenfalls Lösungen welche im nächsten Kapitel diskutiert werden.

4.1.8 Datenanbindung

Neben der Anbindung der Interaktionslogik an die visuelle Darstellung ist die Anbindung der Daten aus dem Datenmodell eine weitere wichtige Aufgabe die ein GUI Framework bestmöglich zu unterstützen hat.

Datenanbindung ist ein Prozess bei dem eine Verbindung zwischen dem Userinterface und der Business Logik hergestellt wird. Wenn diese Datenanbindung korrekt durchgeführt wird, so werden die Daten automatisch von den daran gebundenen UI Elementen angezeigt und auch vom UI Element durchgeführte Änderungen werden an das Datenmodell automatisch reflektiert. [7]

Bsp.: Eine Textbox soll an ein "Name" Attribut eines Objektes gebunden werden. Wird dieses Attribut im Objekt von der Business Logik verändert, so soll die Textbox automatisch den aktualisierten Namen anzeigen. Wird in der Textbox der Name verändert, so soll aber auch in die andere Richtung der Wert des Attributes des Objekts in der Business Logik aktualisiert werden.

Diese Datenanbindung ist keineswegs trivial und soll ebenfalls wie die Logikanbindung möglichst lose gekoppelt sein, so dass man im besten Fall die einzelnen Schichten austauschen kann, ohne Veränderungen in der andern Schicht vornehmen zu müssen.

4.2 Softwarearchitektur

Die der Designumgebung und den Designeritems zu Grunde liegenden Anforderungen werfen folgende Problemstellungen auf:

4.2.1 Entwurfsmuster

Die Wahl der richtigen Entwurfsmuster entscheidend über die Stabilität, Wartbarkeit und Erweiterbarkeit der Lösung. Zwar sind Model View Controller Muster in der Entwicklung mehrschichtiger Applikationen aktueller Stand der Technik, jedoch gibt es in diesem Bereich verschiedene Variationen des ursprünglichen Musters (z.B. MVP [8] oder MVVM [9]) die je nach Domäne und Anforderungen Stärken und Schwächen vorweisen.

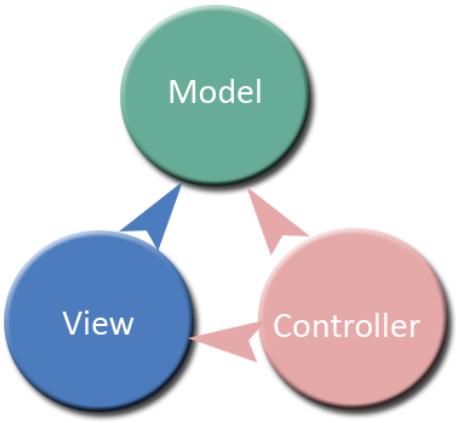


Abbildung 18: Model View Controller

Ein MVC Pattern muss nicht unbedingt für eine Webapplikation und eine Desktopapplikation gleichermaßen gut funktionieren - es kommt hier sehr auf die verwendete GUI Technologie und das dahinter verwendete Datenmodell an, welche Lösung zum Ziel führt.

4.2.2 Generische Architektur

Die Datentypen der Datenquellen, mit denen Eigenschaften der Designeritems verknüpft werden können, sind zwar festgelegt, auf ein allgemeines Design der Schnittstellen der Designeritems und der Designumgebung muss aber zusätzlich gelegt werden, da dies nur Initialanforderungen sind und Änderungen dieser nicht die Grenzen der Architektur aufzeigen sollen.

Zusätzlich muss beachtet werden, dass man Designeritems bzw. deren Eigenschaften nicht vollständig beschreiben kann. Sie sind Platzhalter für jede mögliche erdenkbare Visualisierungsausprägung und dieser Raum für mögliche Visualisierungen soll in keinster Weise durch Unzulänglichkeiten in der Softwarearchitektur eingeschränkt werden.

4.2.3 Erweiterbarkeit

Das Hinzufügen neuer Designeritems zur Applikation ist der wichtigste Punkt für den Erfolg der Designumgebung. Jedes neue zusätzliche Designeritem ist ein Mehrwert und vergrößert automatisch den Einsatzbereich der Applikation. Aus diesem Grund muss die Anpassung bestehender Designeritems, deren Wiederverwendung oder die Erzeugung neuer Designeritems ohne große Hürden bewältigt werden können.

Da die Gestaltung von Designeritems besser in den Händen von Grafikern bzw. Designern liegen sollte, muss eine Schnittstelle zur Verfügung gestellt werden, mit deren Hilfe grafische Experten einfach in den Erstellungsprozess von Designeritems einbezogen werden können.

4.2.4 Zusammengesetzte Anwendungen

Wie bereits erwähnt, soll das Zusammenspiel des GUI Frameworks mit der Logik bestmöglich durch entsprechende Entwurfsmuster abgedeckt werden. Es besteht durch den Einsatz dieser Muster trotzdem noch immer das Risiko, dass eine Anwendung zu monolithisch wird.

In diesem Kontext ist ein weiterer offener Punkt das Zusammenspiel der einzelnen Komponenten untereinander d.h. wie kommunizieren die einzelnen Module miteinander (z.B. Property Grid mit Workspace), aber auch wie eine Designerumgebung als Gesamtpaket in eine Applikation eingebunden werden kann.

In der Entwicklung zusammengesetzter Anwendungen ist es das Ziel Module, in der vorher beschriebenen Ausprägung, austauschbar zu machen und Abhängigkeiten bestmöglich zu minimieren.

Im Falle von unvermeidbaren Abhängigkeiten existieren Strategien wie z.B. das *Inversion of Control* und *Dependency Injection* Pattern welche sich für diese Aufgabe eignen. [10]

5 Technologien und Konzepte

5.1 XML basierte Userinterfaces

GUI Entwickler sind in der Regel keine ausgebildeten Designer und so verwundert es nicht, dass manche Anwendungen in Ihrem Erscheinungsbild verbesserungswürdig sind. Umgekehrt ist ein Designer kaum in der Lage mittels Softwareentwicklungstools seine Ideen umzusetzen - zu sehr sind die Daten und der Code mit der Anzeige verwoben. Altgedienten GUI Frameworks wie z.B. Windows Forms oder Java Swing wurden von Beginn an für Entwickler konzipiert und bieten keine klare Trennung von Daten/Logik und Design.

Ein Userinterface kann einfach als eine Version des Model View Controller (kurz MVC) Patterns betrachtet werden, in welchem die Daten in der Model Schicht und die Anzeige in der View Schicht beschrieben werden [11]. In Kapitel 5.2 werden verschiedene Ausprägungen von MVC Patterns betrachtet - für weitere Betrachtungen in diesem Kapitel reicht es aus, dass die beiden Ebenen Daten/Logik und Design voneinander getrennt sind bzw. sein müssen.

Die Designeritems aus dem vorigen Kapitel sind ein gutes Beispiel für Komponenten bei denen die Logik und das Design, aus der Überlegung der Wiederverwendbarkeit und der Anpassungsmöglichkeit durch professionelle Grafiker, getrennt werden sollte.

Aktuelle GUI Frameworks bieten, durch Verwendung von XML zur Beschreibung des Designs und mit Hilfe spezieller darauf aufbauender Tools, die Möglichkeit GUIs bzw. GUI Komponenten von Grafikern designen zu lassen, mit denen Programmierer im Softwareentwicklungsprozess ohne große Umwege weiterarbeiten können. Zusätzlich wird die Forderung, das Aussehen von der Logik / von den Daten zu trennen, ebenfalls erfüllt.

Auf drei aktuelle Vertreter der Gattung XML basierter Userinterface Frameworks, deren Unterschiede und Vorteile wird hier nun genauer eingegangen.

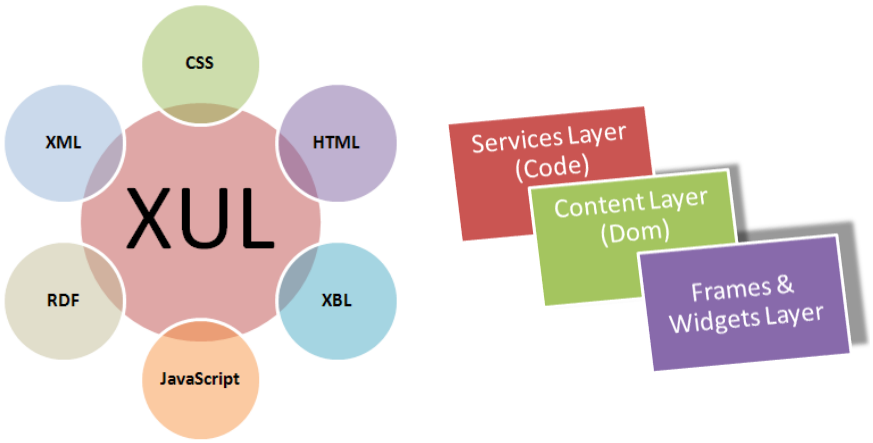
5.1.1 XUL

XUL [12] steht für XML User Interface Language und wurde ihm Rahmen des Mozilla Browser Projektes in erster Linie für die Darstellung von Plattform übergreifenden Benutzeroberflächen entwickelt.

Der Mozilla Browser behandelt XUL in ähnlicher Weise wie HTML - Style Sheets (CSS) und weitere aus HTML bekannte Konzepte wie z.B. der DOM Tree werden hier ebenfalls verwendet. [13]

XUL Anwendungen können aber nicht nur als Firefox Extension benützt werden, sondern auch standalone mittels der XULRunner Applikation [14] betrieben werden.

Abbildung 19(a) zeigt die von XUL verwendeten Technologien [13] - Abbildung 19(b) zeigt die Schichtarchitektur von XUL Anwendungen.



(a) XUL Technologien (b) XUL Architektur

Als dahinterliegende Programmiersprache zur Beschreibung der Logik dient JavaScript. Es gibt aber auch speziell angepasste XUL Frameworks die andere Sprachen unterstützen.

Ein einfaches Hello World Beispiel in XUL zeigt Listing 6

```
<?xml version="1.0" encoding="iso-8859-1"?>
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.
   is.only.xul" id="hello" width="300" height="300">
    <description value="Hello the World!" />
</window>
```

Listing 6: XUL Hello World

Komponentenbibliothek

Die standardmäßig ausgelieferte Komponentenbibliothek (Controls) beinhält alle gängigen UI Komponenten: Button, Menü, Checkbox, Colorpicker, Datepicker, Description, Groupbox, Image, Label, Listbox, Menulist, Progressmeter (Progressbar), Radio, Scale (Slider), Textbox, Timepicker, Tree und Toolbarbutton.

Erweiterbarkeit

XUL bietet die Möglichkeit mit sogenannten XUL Overlays das Aussehen von neuen selbst erstellten Komponenten zu definieren. Dazu wird das Aussehen einer neuen Komponente in eine separate Datei abgelegt und eingebunden.

```
<?xul-overlay id="mycomponent" href="mycomponent.xul"?>
```

Listing 7: XUL Overlay Definition

Über den DOM Tree des Dokuments kann mittels JavaScript auf Elemente zugegriffen und diese auch manipuliert werden.

```
<bindings xmlns="http://www.mozilla.org/xbl"
 xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.
    only.xul">
<binding id="myButton" extends="xul:button">
  <content>
    <xul:image src="button.png"/>
  </content>
</binding>
</bindings>
```

Listing 8: XUL Erweiterung eines Buttons

Eine Erweiterung von bestehenden XUL Komponenten ist ebenfalls möglich wie Listing 8 zeigt - ein Button wird hier in seinem Erscheinungsbild verändert.

Grafische Möglichkeiten

XUL selbst bietet vom Framework aus keine grafischen Basisoperationen an. Es kann jedoch das Canvas Element aus HTML 5 verwendet werden, in dem Grafiken gezeichnet werden können.

Eine weitere Methode um Grafiken selbst zu zeichnen bzw. zur Laufzeit zu manipulieren ist die Kombination von SVG und JavaScript. Folgende Vorgehensweise wird dazu angewandt [15]:

- Quelle ist ein XML das Daten enthält
- über eine XSLT Transformation wird ein SVG erzeugt
- das SVG wird eingebunden
- mittels JavaScript kann über den DOM Tree auf Elemente des SVGs zugegriffen werden

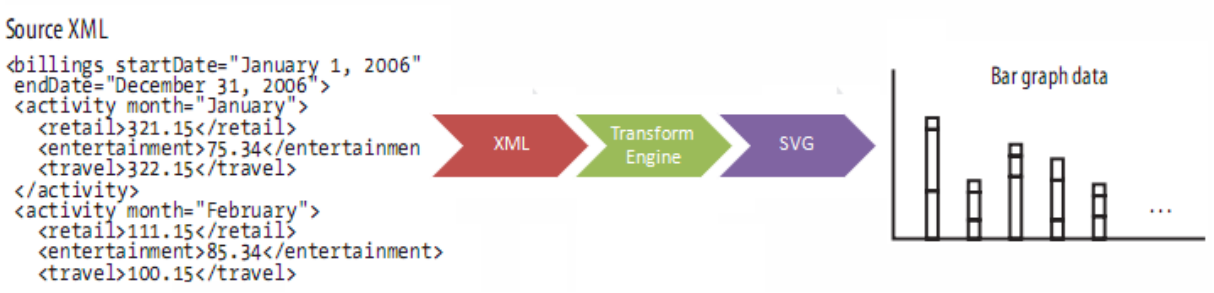


Abbildung 19: XUL SVG Erzeugung [15]

Stylemöglichkeiten

Ein Basisfeature von XUL ist die Unterstützung von Themes - durch Verwendung von verschiedenen CSS Files kann der Applikation ein unterschiedliches "Look and Feel" gegeben werden. Auf diesem Weg können auch bestehende Basiskomponenten in ihrem Erscheinungsbild verändert werden.

Layoutmöglichkeiten

Die XUL Layout Hierarchie basiert auf einem einzelnen Container Element, der Box. Alle weiteren XUL Layout basieren auf verschachtelten Box Konstrukten welche wiederum Boxes enthalten können. [13]

Diese Box Container können zusätzlich mit den Attributen *Flex* (für die automatische Ausdehnung), *Pack* (für die Platzierung der darunterliegenden Child Elemente) und *Align* (für die Ausrichtung der darunter liegenden Child Elemente) manipuliert werden.

Zusätzlich gibt es weitere Container Elemente die für das Layout verwendet werden können wie z.B. Stack, Deck, Grid und Variationen davon. Diese wiederum können mittels CSS in ihrem Erscheinungsbild angepasst werden.

Wiederverwendbarkeit

Eine grundlegende Forderung bei der Entwicklung von XUL war es einen GUI Framework zu entwickeln, der auf unterschiedlichen Plattformen dargestellt werden kann. So ist es kein Problem eine XUL Applikation auf verschiedenen Betriebssystemen laufen zu lassen.

Ebenfalls können XUL GUIs von verschiedenen Programmiersprachen angesprochen werden - neben der Standardsprache JavaScript bieten eigens angepasste XUL Toolkits Unterstützung für Java, C, C++, Python und Perl.

XUL bietet speziell für wiederverwendbare Module die APIs XPCOM (Cross Platform Component Object Model) und XPConnect an, mit deren Hilfe Module erzeugt werden können, die auch unter anderen Sprachen geschrieben wurden bzw. von anderen Sprachen benutzt werden können.

Trennung Logik / Design

Um Logik vom Design besser zu trennen bietet der XUL Framework die Verwendung von Commands an, d.h. es werden keine expliziten Funktionen bei Eventhandlern in XUL Dateien angegeben. Code 9 zeigt ein simples Beispiel hierfür.

```
<command id="cmd_openhelp" oncommand="alert('Help');"/>
<button label="Help" command="cmd_openhelp"/>
```

Listing 9: XUL Commands

Im vorigen Beispiel wurde der Funktionsaufruf aber nur aus dem GUI Element herausgezogen, der Funktionsaufruf findet sich an anderer Stelle in der XUL Datei wieder.

Ein noch besserer Weg um Code vollständig aus der GUI Beschreibung und somit in die Logik zu bekommen ist die Verwendung von Controllern. Listing 10 zeigt dessen Verwendung [16].

```
<window id="controller-example" title="Controller Example" onload
   ="init();" xmlns="http://www.mozilla.org/keymaster/gatekeeper/
   there.is.only.xul">
<script>
function init()
{
  var list = document.getElementById("theList");
  var listController = {
    supportsCommand : function(cmd){ return (cmd == "cmd_delete")
       ; },
    isCommandEnabled : function(cmd){
      if (cmd == "cmd_delete") return (list.selectedItem != null)
         ;
      return false;
    },
    doCommand : function(cmd){
      list.removeItemAt(list.selectedIndex);
    },
    onEvent : function(evt){ }
  };
  list.controllers.appendController(listController);
}
</script>
```

```
<listbox id="theList">
  <listitem label="Ocean"/>
  <listitem label="Desert"/>
</listbox>
</window>
```

Listing 10: XUL Controller

Der dazugehörige Code wird optimaler weise in einer eigenen JavaScript Datei ausgelagert - so ist die Logik vom Design strikt getrennt.

Datenanbindung

Um Daten an GUI Elemente anbinden zu können bietet XUL einen Template genannten Mechanismus an. Dazu wird bei einem GUI Element das Attribut Datasource gesetzt, gefolgt von einem Querytype Attribut.

Listing 11 zeigt die Erzeugung von Elementen einer Listbox mit dahinterliegendem XML als Datenquelle.

```
<listbox datasources="people.xml" ref="*" querytype="xml">
  <template>
    <query expr="person"/>
    <action>
      <listitem uri="?" label="?name"/>
    </action>
  </template>
</listbox>
```

Listing 11: XUL Datenanbindung mit Template [16]

Eine Stärke von XUL Templates ist die Möglichkeit Daten aus XML, RDF Dokumenten oder von SQL Queries zu lesen und darin gezielt Daten abzufragen und dementsprechend Content zu generieren.

Dieser Ansatz erlaubt es aber nur Daten aus einem Modell in die GUI zu bringen - um Änderungen von Daten aus der GUI in das Modell zu reflektieren muss selbst Logik geschrieben werden, die diese Aufgabe erledigt.

5.1.2 MXML und FLEX

MXML ist eine XML basierte Sprache die ursprünglich von Macromedia eingeführt wurde, um deklarativ User Interfaces zu beschreiben, welche von Flash Playern dargestellt werden kann.

Die Abkürzung MXML wurde ursprünglich als Kunstwort ohne Bedeutung eingeführt - im Nachhinein wurde ihr aber von der Community als Bedeutung *Macromedia XML* oder *Magic XML* gegeben.

Mittels MXML selbst ist es nur möglich User Interfaces zu beschreiben, um Logik und Interaktion zu verwenden wird zumindest ActionScript benötigt - beide Begriffe zusammen werden unter dem Begriff *FLEX* (aktuell in der Version 4) geführt. [17] MXML und ActionScript werden zusammen in eine Flash Datei kompiliert, welche dann von einem Flash Player dargestellt bzw. benutzt werden können.

Neben ActionScript als Sprache zur Beschreibung der Logik und Interaktion kann auch Java verwendet werden, um aufwändigere Applikationen entwickeln zu können - Abbildung 20 zeigt einen schematischen Ablauf beteiligter Komponenten.

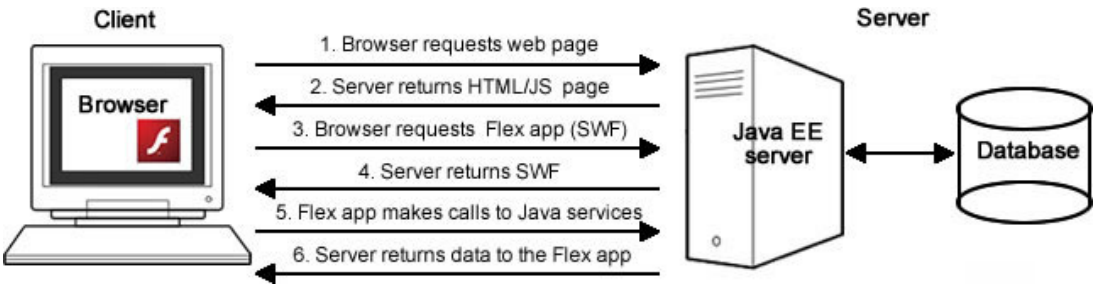


Abbildung 20: FLEX / Java Interaktion [18]

Zusätzlich neben der Anzeige von FLEX Applikationen im Webbrowser können mit Adobe AIR diese auch am Desktop unter Windows, Mac OS X und Linux verwendet werden. Adobe AIR ist sozusagen eine Shell für Flex, Flash und AJAX Programme. [19]

Ein einfaches Hello World Beispiel in MXML zeigt Listing 12

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Label="Hello World" fontSize="64" />
</s:Application>
```

Listing 12: MXML Hello World

Komponentenbibliothek

FLEX bietet eine sehr umfangreiche Komponentenbibliothek, auf die Entwickler zugreifen können - Abbildung 21 gibt den Überblick über die vorhandenen Basiskomponenten. [20]

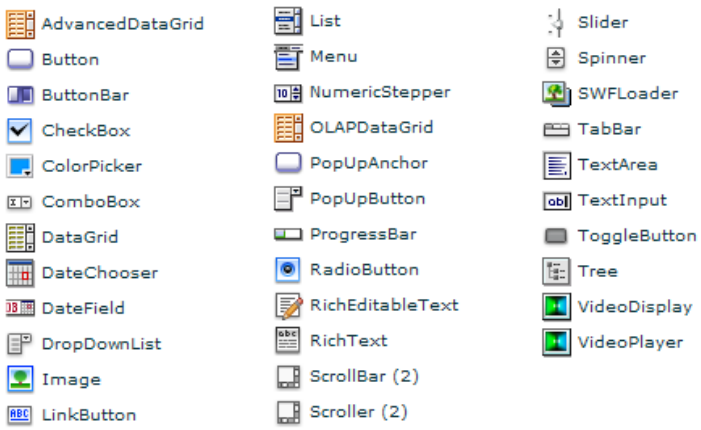


Abbildung 21: FLEX 4 Komponentenbibliothek

Erweiterbarkeit

Die Erstellung von eigenen Komponenten, als auch die Erstellung von um eigene Funktionalität erweiterten Basiskomponenten, ist in MXML bzw. FLEX 4 von Grund auf vorgesehen.

Will man zum Beispiel einen Button um Funktionalität erweitern, so wird von der entsprechenden Control Klasse einfach abgeleitet - diese kann nach Import des dazugehörigen Namespace in MXML wieder verwendet werden, was Listing 13 zeigt:

```
/* MyButton.as */
package myControls {
    import mx.controls.Button;
    public class MyButton extends Button {
        public function MyButton() { ... }
        ...
    }
}

<!-- MyApplication.mxml -->
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:s="library://ns.adobe.com/flex/spark" xmlns:
   cmp="myControls.*">
    <cmp:MyButton label="MyButton"/>
</s:Application>
```

Listing 13: Erweiterung eines Buttons

Ähnlich verhält es sich mit der Erstellung von komplett neuen Komponenten - hier wird von einer anderen Basisklasse abgeleitet *SkinnableComponent* - zusätzlich muss aber das Erscheinungsbild der neu definierten Komponente hinterlegt werden. Dies wird mit sogenannten *SkinParts* erledigt, welche das Erscheinungsbild der selbst erstellten Komponente definieren. [21]

Listing 14 zeigt eine selbst erstellte Komponente die einen Button beinhält.

```
/* MyComponent.as */
package myControls {
    public class MyComponent extends SkinnableComponent {

        SkinPart(required="true")]
        public var knob:Button;

        public function MyComponent() { ... }
        ...
    }
}

<!-- MyApplication.mxml -->
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:cmp="myControls.*">
    <cmp:MyComponent>
           <s:Button id="knob" label="MyComponent" />
    </cmp:MyComponent>
</s:Application>
```

Listing 14: Erscheinungsbild einer eigenen Komponente

Grafische Möglichkeiten

Flex 4 bietet grafische Primitive wie Image, Ellipse (siehe Listing 15), Graphic, Line, Path, Rect, RichEditableText und RichText, mit denen gezeichnet werden kann. [22]

```
<s:Graphic horizontalCenter="0" verticalCenter="0">
    <s:Ellipse height="100" width="250">
        <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="2"/>
        </s:stroke>
    </s:Ellipse>
</s:Graphic>
```

Listing 15: Anzeigen eines grafischen Primitivs in MXML

Zusätzlich können in MXML sogenannte Flash XML Grafiken (kurz FXG [23]) eingebunden werden - die äquivalent zu SVG Grafiken plattformübergreifend beschreiben sollen - jedoch für die Darstellung am Flash Player optimiert sind.

Stylemöglichkeiten

So wie in XUL können GUI Elemente auch in MXML Files mittels CSS gestylt werden - in der MXML Datei selbst (inline) als auch in separaten Style Dateien 16.

```
<fx:Style
  @namespace s "library://ns.adobe.com/flex/spark";
  s|Panel {
     color: #FF0000;
  }
</fx:Style>

<fx:Style source="styles.css" />
```

Listing 16: CSS Styles in MXML

Neben der Auslagerung von Styles ist es in MXML bzw. FLEX auch möglich Skins zu erstellen, die zur Laufzeit das Erscheinungsbild der Applikation ändern können. Dazu werden zu den Komponenten *SkinClass* Parameter angegeben und pro Skin eigene MXML Files hinterlegt die das Erscheinungsbild des Skins beschreiben.

Layoutmöglichkeiten

Neben der Möglichkeit Komponenten absolut in einem Fenster zu Positionieren bietet FLEX eine Reihe von Layoutmöglichkeiten. Mit der Hilfe von Constraints können Elemente gezwungen werden z.B. gewisse Abstände einzuhalten und es existiert auch die Möglichkeit Komponenten automatisch zu layouten (z.B. horizontales Layout reiht alle Elemente nebeneinander).

Wünscht man jedoch mehr Einfluss auf das Layout der platzierten Elemente, so können zu Verfügung gestellte Container verwendet werden, die darin eingebettete Elemente entsprechend ihrem Wesen / ihrer Logik layouten.

FLEX 4 bietet als Layout Container Canvas, ControlBar, Group, Panel, DataGroup, DividedBox, Form und einen Grid Container an. [17]

Wiederverwendbarkeit

Wie bereits erwähnt werden durch MXML definierte User Interfaces vom Flash Player gerendert, dieser ist auch auf den unterschiedlichsten Plattformen verfügbar. Zusätzlich können MXML Inhalte auch mittels Adobe Air ohne Browser bzw. Flash Player dargestellt werden - beides zeigt dass MXML bzw. FLEX Plattformübergreifend benützt werden kann.

Die Programmiersprache im Hintergrund von MXML ist ActionScript - es ist aber auch möglich z.B. von Java Servlets MXML Dateien erzeugen zu lassen, die zur Laufzeit kompiliert und dann vom Flash Player angezeigt werden.

Trennung Logik / Design

FLEX bietet so wie XUL die Möglichkeit Logik (in diesem Fall ActionScript) in eigene Dateien auszulagern. Es kann sowohl Logik-Code direkt in MXML aufgerufen (Listing 17) als auch Event Listener in den Logik Dateien hinzugefügt werden (Listing 18), um auf Benutzerinteraktion wie z.B. Mouse Clicks zu reagieren.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:fx="http://ns.adobe.com/mxml/2009">

<fx:Script source="myLogic.as" />

<s:Button id="button1" label="button 1" click="doSomething()" />

</s:Application>
```

Listing 17: Eventanbindung in MXML

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:fx="http://ns.adobe.com/mxml/2009">

<fx:Script>
<![CDATA[
privat function init():void {
    button1.addEventListener(MouseEvent.CLICK, doSomething);
}
]]>
</fx:Script>

<s:Button id="button1" label="button 1" />

</s:Application>
```

Listing 18: Eventlistener in FLEX

Datenanbindung

Um Daten in die Userinterfaces zu bringen, gibt es neben der Standardmöglichkeit sie programmatisch aus dem Code heraus in die entsprechenden Komponenten zu füllen, in FLEX einen Databinding Mechanismus.

Dabei hat man die Möglichkeit auf Werte anderer MXML Elemente zuzugreifen z.B. *<s:Label text="{textInput1.text}"/>*. Wie dieses einfache Beispiel zeigt werden Bindungen innerhalb geschwungener Klammern definiert.

Eine Eigenheit von Flex ist die Bindung in zwei Richtungen, so kann man z.B. den Inhalt zweier Textboxen verbinden 19.

```
<s:TextInput id="textInput1" text="@{textInput2.text}" />
<s:TextInput id="textInput2" />
```

Listing 19: Two Way Binding in MXML [22]

Um von MXML auf Variablen in der Logik (= Code Behind Dateien) zugreifen zu können bzw. GUI Elemente darauf binden zu lassen, muss eine Variable mit einem Attribut versehen werden (*Bindable*) um als Quelle eines Bindings akzeptiert zu werden 20.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:s="library://ns.adobe.com/flex/spark" xmlns:
   fx="http://ns.adobe.com/mxml/2009">

<fx:Script>
<![CDATA[
  [Bindable]
  private var _str:String = "Hello!";
]]>
</fx:Script>

<s:Button id="button1" label="@{_str}" />

</s:Application>
```

Listing 20: Anbindung von Variablen im Code Behind

5.1.3 XAML und WPF

XAML steht für Extensible Application Markup Language und ist eine von Microsoft entwickelte auf XML basierende deklarative Beschreibungssprache für die Definition von Objektbäumen. [24]

Die Begriffe Windows Presentation Foundation (kurz WPF) [6] und XAML werden oft in einem Atemzug genannt, da man mittels XAML auch Userinterfaces beschreiben kann, jedoch wird XAML aber auch für andere Zwecke verwendet.

Aus den durch XAML definierten Objektbäumen werden allgemein .NET Objekte erzeugt, die nicht nur auf GUI Objekte beschränkt sind - somit kann diese Beschreibungssprache

Basis für verschiedenste Anwendungen sein, im konkreten Fall werden hier aber nur die GUI Fähigkeiten besprochen. [25]

Durch die Einbettung von XAML bzw. WPF in den .NET Framework steht eine umfangreiche Bibliothek von Klassen, Services und Schnittstellen zur Verfügung, die vom GUI Framework aus genutzt werden kann.

Ergänzend soll hier erwähnt werden, dass XAML zur Beschreibung von Userinterfaces sowohl für Desktopanwendungen als auch zur Erstellung von Webanwendungen mit Hilfe des Microsoft Frameworks Silverlight [26] genutzt werden kann.

Listing 21 zeigt ein einfaches Hello-World Beispiel in XAML.

```
<Window x:Class="WpfApplication1.Window1" xmlns="http://schemas.
   microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://
   schemas.microsoft.com/winfx/2006/xaml" Title="Window Hello"
   Height="300" Width="300">
    <Grid>
       <TextBlock Text="Hello World!" />
    </Grid>
</Window>
```

Listing 21: XAML Hello World

Komponentenbibliothek

Die Windows Presentation Foundation bietet eine umfangreiche Anzahl an Basiskomponenten (Abbildung 19). Speziell hier hervorzuheben ist die Basiskomponente *WindowsFormsHost* mit Hilfe deren es möglich ist zusätzlich zu den WPF Komponenten auch Windows Forms Komponenten zu benützen.

Durch den Einsatz dieser Komponente wird die Anzahl der vorhandenen Basiskomponenten um einiges erhöht. Sie gewährt den Zugang zu sehr speziellen, in anderen Frameworks standardmäßig nicht enthaltenen Komponenten wie z.B. den Propertygrid.

Erweiterbarkeit

WPF bietet zahlreiche Möglichkeiten Komponenten zu erweitern oder eigene Komponenten zu erstellen - je nach dem man von welcher Basiskomponente oder von welchem Basis Framework Element man ableitet, bekommt man nötige Unterstützung und / oder Freiheiten. Dies reicht soweit, so dass man falls gewünscht auch die volle Kontrolle über die Render Engine erhalten kann.

Durch diese zahlreichen Möglichkeiten besteht zumindest die Gefahr, dass man aufgrund einer initial getroffenen Entscheidung einen bestimmten Weg zu gehen, im Nachhinein

auf bestimmte Einschränkungen trifft oder selbst zu viel Arbeit erledigen muss, die ein anderer Weg nicht mit sich gebracht hätte. [27]

Listing 22 zeigt die Erweiterung eines Buttons - Listing 23 die Erstellung eines eigenen Controls. Die beiden Controls können nach ihrer Definition einfach wieder in XAML eingebunden werden ($< CustomControl/ >$ bzw. $< MyControl/ >$).

```
/* CustomControl.cs */
namespace WpfApplication
{
    public class CustomControl : Button
    {
        static CustomControl()
        {
            ...
        }
        ...
    }
}

<!-- Generic.xaml -->
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx
   /2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/
   winfx/2006/xaml" xmlns:local="clr-namespace:WpfApplication">
    <Style TargetType="{x:Type local:CustomControl}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type local:
                   CustomControl}">
                    <!-- Definition des Erscheinungsbildes -->
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</ResourceDictionary>
```

Listing 22: Erweiterung eines Buttons

```
/* MyControl.cs */
namespace WpfApplication
{
    public partial class MyControl : UserControl
    {
        public MyControl()
        {
            InitializeComponent();
        }
        ...
```

```
    }
}

<!-- MyControl.xaml -->
<UserControl x:Class="WpfApplication.MyControl" xmlns="http://
   schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="
   http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
         <TextBlock Text="MyControl" />
    </Grid>
</UserControl>
```

Listing 23: Erstellung eines eigenen Controls

Grafische Möglichkeiten

WPF ermöglicht es auf Basis von *Drawings*, *Visuals* und *Shapes* sowohl 2D als auch 3D-Inhalte zu erzeugen. Folgender Zusammenhang besteht hinter den drei Elementen: Drawings sind einfache Beschreibungen von Pfaden und Formen die Füllungen enthalten können und mit verschiedenen Arten von Pinseln (Brushes) gezeichnet werden. Visuals ist ein spezieller Weg um Drawings zu zeichnen, diese sind leichter zu bedienen, da sie auch mit Gruppen von Objekten umgehen können. Die einfachste Form zu zeichnen ist die Verwendung von Shapes welche vordefinierte Visuals darstellen. [28]

Möchte man z.B. eine Gruppe von drei Kreisen zeichnen, so gibt es folgende Möglichkeiten

- man zeichnet drei Kreise innerhalb der Geometrie eines ImageDrawings mit Hilfe von XAML
- man erzeugt ein Visual und zeichnet in der OnRender Methode die drei Kreise - das Visual kann dann in XAML als eigenständiges Objekt deklariert werden
- oder man verwendet Circle Shapes und deklariert drei davon direkt in XAML

Zusätzlich bietet WPF auch die Möglichkeit 3D Objekte darzustellen und die Darstellung von 2D Objekten durch Transformationen, Effekten und Animationen anzureichern.

In WPF werden alle Elemente als Vektoren gerendert, eine Anbindung von SVG für die Darstellung von Vektorinhalten entfällt dadurch somit - es ist zusätzlich aber auch möglich SVG Grafiken in XAML Code zu konvertieren.

Stylemöglichkeiten

Listing 23 zeigte bereits implizit die Verwendung von Styles bei der Erstellung eines eigenen Controls. In einem eigenen File (in WPF Resource Dictionary genannt) wurde der Style für das selbst definierte Control festgelegt.

Zusätzlich ist es aber auch möglich das Aussehen bestehender Komponenten zu verändern (siehe Listing 24).

```
<Window.Resources>
    <Style TargetType="{x:Type Button}">
        <Setter Property="Background" Value="Green" />
    </Style>
</Window.Resources>
```

Listing 24: Überschreiben von Styles

Abgerundet werden die grafischen Möglichkeiten von WPF durch die Verwendung von Themes - d.h. das Aussehen der visuellen Elemente passt sich dem Betriebssystem an, auf dem die WPF Anwendung läuft.

Layoutmöglichkeiten

WPF 4 bietet aktuell folgende Layoutcontainer an: Canvas, StackPanel, WrapPanel, DockPanel, Grid, TabPanel, ToolbarOverflowPanel, ToolbarTray, UniformGrid, Scrollviewer und Viewbox.

Aufgrund der offenen Architektur ist es aber auch möglich sich selbst einen Container zu schreiben, in dem man dann selbst für die Positionierung der Kinderelemente verantwortlich ist.

Wiederverwendbarkeit

Wie bereits eingangs erwähnt baut die Windows Presentation Foundation auf .Net Technologien auf - sie kann somit auch von allen .Net Sprachen verwendet werden (VB, C#, C++).

Die Beschränkung auf Windows basierten Betriebssytemen ist im Desktop Bereich aufrecht, es existiert keine alternative Implementierung auf anderen Betriebssystemen. Anders verhält es sich bei Silverlight, das ebenfalls WPF Elemente in sich trägt - hier sind Wiedergabeplugins für andere Betriebssystemplattformen vorhanden.

Trennung Logik / Design

Ebenfalls wie die zuvor vorgestellten GUI Frameworks bietet die WPF auch Mechanismen um die Logik vom Design der Anwendung zu trennen. Ein Control (Window, Button, UserControl, ...) muss immer als Paar einer XAML Datei und einer Source Datei (Code Behind File) gesehen werden.

In Listing 23 sieht man über die partielle Ableitung, dass beide zusammen zu einer Einheit kompiliert werden - die Frage die sich nun stellt ist, wie eine Interaktion beider Teile stattfinden kann.

Betrachten wir folgendes Beispiel: auf ein Klick Event eines Buttons soll reagiert werden - WPF sieht dafür folgende Möglichkeiten vor:

- Aufruf einer Methode von XAML aus (Listing 25)
- Hinzufügen eines EventHandlers im Code Behind File (Listing 26)
- Überschreiben des Standard Click Event Handlers im Code Behind File (Listing 27)
- Definition eines Commands das von XAML aus referenziert wird (Listing 28)

```
<!-- MyWindow.cs -->
private void Button_Click(object sender, RoutedEventArgs e)
{
    //Do something
}

<!-- MyWindow.xaml -->
<Button Height="10" Width="10" Click="Button_Click" Content="
   Click Me"/>
```

Listing 25: Button Interaktion Beispiel 1

```
<!-- MyWindow.cs -->
public MyWindow()
{
    InitializeComponent();
    button1.Click += new RoutedEventHandler(Button_Click);
}

void Button_Click(object sender, RoutedEventArgs e)
{
    //Do something
}

<!-- MyWindow.xaml -->
<Button Name="button1" Height="10" Width="10" Content="Click Me"
   />
```

Listing 26: Button Interaktion Beispiel 2

```
<!-- MyButton.cs -->
public class MyButton : Button
{
    protected override void OnClick(...)
    {
        //Do something
    }
}

<!-- MyWindow.xaml -->
```

```
<MyButton Height="10" Width="10" Content="Click Me"/>
```

Listing 27: Button Interaktion Beispiel 3

```
<!-- MyWindow.cs -->
public MyWindow()
{
    InitializeComponent();
    CommandBinding binding = new CommandBinding(
       ApplicationCommands.Open);
    binding.Executed += new ExecutedRoutedEventHandler(
       Button_Click);
    this.CommandBindings.Add(binding);
}

private void Button_Click(object sender, ExecutedRoutedEventArgs
   e)
{
    //Do something
}

<!-- MyWindow.xaml -->
<Button Command="Open" Height="10" Width="10" Content="Click Me"
   />
```

Listing 28: Button Interaktion Beispiel 4

Die zuvor gezeigten Beispiele lassen die Vielfältigkeit der Möglichkeiten von WPF erahnen - es existieren noch weitere Möglichkeiten auf Events zu reagieren, da Events nicht nur an ein Element ausgelöst werden, sondern diese über den visuellen Baum weiterpropagiert werden.

Datenanbindung

Ebenfalls wie bereits bei FLEX / MXML kurz vorgestellt verfügt WPF / XAML ebenfalls über einen Bindingmechanismus, der es erlaubt Eigenschaften eines Elements an andere Eigenschaften zu binden. Zusätzlich erlaubt es nicht nur an benannte Elemente zu binden, sondern der Zugriff auf Vorfahren oder Nachkommen im visuellen Baum ist ebenfalls möglich (siehe Listing 29).

```
//MyWindow.cs
public static DependencyProperty MyVarProperty =
   DependencyProperty.Register("MyVar", typeof(String), typeof(
   MainWindow), new FrameworkPropertyMetadata("Hello"));

public String MyVar { get; set; }
```

```
<!-- MyWindow.xaml -->
<Window x:Class="WpfApplication3.MainWindow" xmlns="http://
   schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="
   http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Name="myWindow" Height="350" Width="
           525">
  <StackPanel Height="200" Width="200">
     <!--1-->
     <Button x:Name="button1" Width="{Binding RelativeSource={
        RelativeSource FindAncestor, AncestorType=StackPanel},
        Path=Width}" Height="50" />

     <!--2-->
     <Button x:Name="button2" Width="{Binding ElementName=myImage
        , Path=ActualWidth}" Height="50">
                    <Image x:Name="myImage" Width="45" Source="/
                       MyApplication;component/Images/Image.jpg"
                       />
     </Button>

     <!--3-->
     <TextBox Text="{Binding ElementName=myWindow, Path=MyVar,
        Mode=TwoWay}"/>
   </StackPanel>
</Window>
```

Listing 29: Binding Mechanismus in XAML

(1) Die Eigenschaft *Width* des Button *button1* wird an die *Width* Eigenschaften des Stackpanels gebunden (durch finden eines Vorfahren eines Typs).

(2) Die Eigenschaft Width des Buttons *button2* wird an die *Width* Eigenschaft des benannten Elements *myImage* gebunden.

(3) Der Inhalt der Textbox wird beidseitig an die Variable *MyVar* in der Klasse *MyWindow* gebunden - Änderungen sowohl im GUI als auch in der Klasse werden in beide Richtungen reflektiert.

Neben den zuvor erwähnten Binding Mechanismen um Daten aus der Logik in der GUI anzuzeigen, gibt es noch eine weitere Möglichkeit eine Datenklasse direkt in eine entsprechende Visuelle Repräsentation umzuwandeln. Wie eingangs erwähnt beschränken sich die in XAML enthaltenen Elemente nicht nur auf visuelle Elemente, sondern jegliche Objekte können in einem XAML File deklariert werden.

Wird z.B. eine Datenklasse in XAML deklariert, so wird in der GUI nur ein String (von der toString Methode) angezeigt. Mit Hilfe von Data Templates kann das Erscheinungsbild einer Datenklasse definiert werden.

Ein Data Template besteht aus XAML Anweisungen, dass definiert wie die daran gebundenen Daten auszusehen haben. [29]

5.1.4 Vergleich

Betrachtet man alle drei hier vorgestellten XML basierten GUI Frameworks, mit dem Ziel die in Kapitel 3 vorgestellte Designumgebung umzusetzen bzw. Informationsvisualisierung zu betreiben, so kommt man in den einzelnen Kategorien zu folgenden Schlüssen:

Komponentenbibliothek

Es existiert in keinem der drei betrachteten GUI Frameworks ein Container der von Haus aus als Designerworkspace verwendet werden kann - dieser muss selbst entwickelt werden, auch eine Komponente eines Drittanbieters sucht man vergeblich.

Alle drei GUI Frameworks sind aber mit ausreichend Basiskomponenten ausgestattet um normale Userinterfaces zu erzeugen - um Informationsvisualisierung zu betreiben kommt man nicht an dem Weg vorbei, eigene Komponenten zu entwickeln, da die benötigten Komponenten dem Wesen von normalen UI Komponenten nicht sehr nahe kommen.

Ein großer Pluspunkt der für die Windows Presentation Foundation spricht ist, sie bietet als einziges einen Property Grid an. Diese Komponente ist zwar nur für Windows Forms vorhanden - kann aber in WPF Applikationen eingebunden werden.

Erweiterbarkeit

Sowohl XUL, MXML / FLEX und XAML / WPF bieten die Möglichkeit an, bestehende Komponenten zu erweitern und eigene Komponenten zu erstellen. Somit wären alle drei Frameworks prinzipiell geeignet um Designeritems zu erstellen.

Die Art wie dies umgesetzt wird unterscheidet sich jedoch, vor allem bei XUL, stark. Welche Lösung hier sich in der Praxis als die Beste erweist könnte nur durch einen "proof of concept" bewertet werden, da man Limitierungen meist erst erkennt, wenn man sich von der Komplexität her etwas von Tutorial Beispielen entfernt.

Grafische Möglichkeiten

Etwas ernüchternd sind die grafischen Möglichkeiten bei XUL - um Informationsvisualisierung zu betreiben kommt man nicht vorbei, Designeritems mit Grafiken auszustatten, welche sich zur Laufzeit verändern. Der Weg aus XML über XSLT ständig SVGs zu erzeugen scheint zur Laufzeit sehr aufwändig - der Canvas in HTML5 bietet jedoch ausreichend Basisfunktionalität um Grafiken zur Laufzeit zu erzeugen.

Besser verhält es sich bei MXML / FLEX welches von Haus aus grafische Primitive unterstützt, die nicht in einem speziellen Canvas eingebettet werden müssen. XAML / WPF bietet wohl mit Abstand die meisten grafischen Möglichkeiten um auf unterschiedlichste Art und Weise Grafiken zu erzeugen, zu Transformieren, mit Effekten und Animationen zu versehen und auch 3D Inhalte darzustellen.

Stylemöglichkeiten

Alle drei vorgestellten GUI Frameworks bieten die Möglichkeit Elemente in ihrem Aussehen zu beeinflussen und sind in diesem Punkt als Äquivalent zu betrachten.

Layoutmöglichkeiten

Ebenfalls wie mit den Stylemöglichkeiten verhält es sich mit den Layoutmöglichkeiten - diese sind in allen drei Frameworks vorhanden - auch über diese Kategorie lässt sich keine wirkliche Differenzierung schaffen.

Wiederverwendbarkeit

In der Verwendbarkeit unterscheiden sich die drei GUI Frameworks am meisten und verfolgen unterschiedliche Philosophien: XUL wurde von Anfang an Plattformübergreifend entworfen. MXML / FLEX lässt sich auf allen Plattformen ausführen auf denen ein Flash Player oder eine Air Umgebung vorhanden ist - XAML / WPF hingegen ist auf Windows Plattformen beschränkt - einziger Ausweg auf andere Plattformen wäre die Nutzung von Silverlight was aber den Funktionsumfang des Frameworks einschränkt. In diesem Punkt ist XUL und vor allem MXML / FLEX besser für die Verwendung auf mehreren Betriebssystemplattformen gerüstet.

Bei der Einbindung mehrere Programmiersprachen halten sich die Möglichkeiten der drei Frameworks die Waage.

Trennung Design / Logik

Am ausgereiftesten scheinen die Möglichkeiten von XAML / WPF um das Design von der Logik zu trennen - es bietet zahlreiche Möglichkeiten um das Design mit Logik zu verknüpfen. Dahinter liegen XUL und MXML / FLEX mit der Anbindung über EventListener als Alternative zu Codeaufrufen aus dem XML.

Datenanbindung

Ähnlich wie in der Kategorie Design / Logik verhalten sich die hier die einzelnen Frameworks - XAML / WPF bietet wiederum eine Fülle von Möglichkeiten zur Verknüpfung der Daten mit GUI Elementen. MXML / FLEX bietet etwas weniger Möglichkeiten aber ausreichende Lösungsansätze, XUL hingegen beschreibt mit dem Template Weg eher einen einseitigen Weg in eine Richtung - die Reflektion der Daten von der GUI in die Logik bzw. in das Datenmodell ist hier nicht ausgereift.

Fazit

Kann man bei der Betriebssystemabhängigkeit Abstriche machen, so scheint XAML / WPF am geeignetsten für die Entwicklung einer Designumgebung. Dafür sprechen der vorhandene Propertygrid, die herausragenden grafischen Möglichkeiten und die beste Unterstützung bei der Trennung von Logik und Design sowie bei der Datenanbindung.

Muss man von Betriebssystemen unabhängig sein so bietet sich MXML / FLEX an, da man nur einige Abstriche gegenüber XAML / WPF in den zuvor genannten Punkten machen muss.

XUL hingegen scheint, und dieses Fazit ist auch nicht verwunderlich, denn dafür wurde es entwickelt, am besten für die Darstellung und Interaktion mit normalen GUIs geeignet zu sein - und würde nur bedingt für die Entwicklung einer Designumgebung zur Informationsvisualisierung passen.

5.2 UI Architekturkonzepte

Das Entwickeln der Benutzeroberfläche einer professionellen Softwareanwendung ist nicht ganz einfach. Es kann eine schwer durchschaubare Mischung aus Daten und Interaktionen (funktionale Anforderungen) als auch visuellem Entwurf (nicht funktionale Anforderungen) sein. [9]

Eine Software-Architektur identifiziert Komponenten und regelt deren Zusammenspiel so, dass sowohl die funktionalen als auch die nicht funktionalen Anforderungen erfüllt sind. [10]

In diesem Kapitel wird auf drei verschiedene Architekturkonzepte für GUIs genauer eingegangen und deren Vor und Nachteile herausgearbeitet.

Vergleichendes Beispiel

Anhand folgendem Beispiel sollen die Abläufe der hier vorgestellten UI Architekturkonzepte verdeutlich werden:

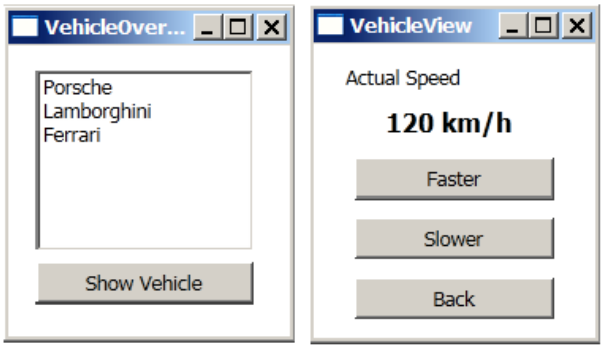


Abbildung 22: GUI Architektur vergleichendes Beispiel

Eine Applikation basiert auf folgendem Datenmodell: Ein einzelnes Objekt vom Typ Fahrzeug beinhaltet eine Eigenschaft: Geschwindigkeit - diese Eigenschaft kann im Objekt gelesen / geschrieben werden. Die Applikation soll über eine GUI die Möglichkeit bieten, die aktuelle Geschwindigkeit des Fahrzeuges anzuzeigen als auch diese über zwei

Buttons ("schneller", "langsamer") zu verändern. Zusätzlich gibt es eine Übersicht aller verfügbaren Fahrzeuge, in der verschiedene Fahrzeuge ausgewählt werden können.

Betrachtete Lösungen

Aus diesem Beispiel ergeben sich folgende Probleme die von den einzelnen UI Architekturkonzepten unterschiedlich gelöst werden:

- **Logik**: Wer hält die Logik für die Änderung von Datenobjekten?
- **Synchronisation** Wer ist für die Synchronisation von Datenänderungen zwischen dem Datenobjekt und der GUI verantwortlich?
- **Datenhaltung** Wer hält die Daten, die in der GUI angezeigt werden?
- **Kopplung** Wie eng sind die Daten mit der GUI gekoppelt?
- **Austauschbarkeit** Welche Komponenten innerhalb der Architektur sind wie stark aneinander gekoppelt bzw. welche sind leicht austauschbar und wie?

5.2.1 MVC

Das für GUI Architekturen am häufigsten verwendete Konzept ist das Model-View-Controller Pattern, es ist aber auch das am häufigsten missinterpretierte Pattern. Oft wird nur aufgrund der Tatsache, dass eine dreistufige Schichtenarchitektur verwendet wird, diese als Architektur MVC betitelt - meist handelt es sich jedoch um eine Variation oder Abwandlung des Patterns. [30] Dazu kommt noch der Umstand, dass die unterschiedlichsten Definitionen in Literatur, Dokumentationen und Internet von MVC kursieren.

Trygve Reenskaug hat das MVC Pattern für Benutzeroberflächen 1979 beschrieben und definiert die einzelnen Schichten folgendermaßen [31]:

- **Model** Modelle beschreiben Wissen. Ein Modell kann sowohl ein einzelnes Objekt als auch aus einer Struktur / Sammlung von Objekten bestehen.
- **View** Eine View ist eine grafische Repräsentation eines Modells. Es stellt einzelne Attribute eines Modells dar, und unterdrückt andere. Eine View ist mit seinem Modell verbunden und bekommt Daten die für die Präsentation notwendig sind von dem Modell durch Anfragen nach Ihnen.
- **Controller** Ein Controller ist die Schnittstelle zwischen Benutzer und dem System. Es bietet dem Benutzer die Möglichkeit auszuwählen, welche Views dargestellt werden sollen. Zusätzlich entscheidet der Controller über Benutzerinput (der aus der View kommt) wie welche Daten im Modell manipuliert werden sollen.

Abbildung 23 zeigt den Zusammenhang zwischen den drei Parteien, der hier nochmals textuell verdeutlicht werden soll:

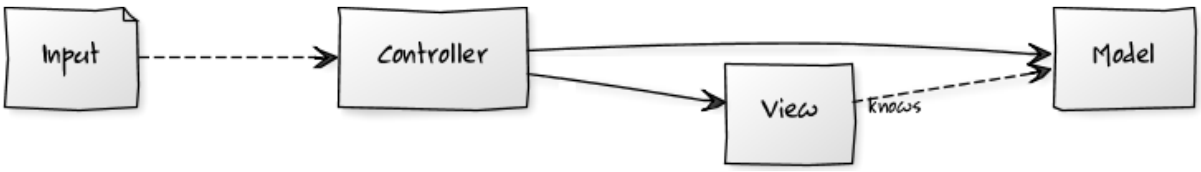


Abbildung 23: MVC Architektur

- Die View hängt vom dahinter liegenden Model ab - sie hat also Kenntnis darüber wie das Modell aufgebaut ist. Sie muss Daten vom Modell darstellen können - aus diesem Grund muss Sie wissen was sie geliefert bekommt.
- Umgekehrt hat bzw. benötigt das Modell keinerlei Wissen über die View - es kann selbstständig entwickelt werden, wird jedoch das Modell geändert - so muss auch die View angepasst werden.
- Muss nur die View geändert werden, so betreffen diese Änderungen das Modell nicht.
- Der Controller bekommt Input: dieser kann z.B. vom User kommen wenn er eine spezielle View angezeigt bekommen möchte oder der Controller entscheidet aufgrund von anderen Input welche View angezeigt werden soll.
- Zusätzlich zur View wählt der Controller auch das entsprechende Modell - er stellt die Verbindung beider Schichten her, in dem er die View an das Modell bindet.
- Im Controller befindet sich die Manipulationslogik des Modells - der Controller kann hierfür natürlich auch Methoden verwenden die von den Objekten im Modell zur Verfügung gestellt werden.
- Die UI-Logik um auf Userinput zu reagieren (z.B. auf einen Button Klick) steckt in der View, die View behandelt aber den Input nicht, sondern leitet Anfragen an den Controller weiter.

Vergleichendes Beispiel

Nach dem MVC Pattern ergeben sich, für das in der Einleitung dieses Kapitels vorgestellten Beispiels, folgende Abläufe zwischen Model View und Controller:

Initialisierung (Abbildung 25)

(1) Die VehicleOverviewView wird ausgeliefert - in ihr können verschiedene Fahrzeuge ausgewählt werden. Der Controller bekommt als Input die ID des gewünschten Fahrzeugs.

Fahrzeugauswahl (Abbildung 25)

(2) Nach dem der User ein Fahrzeug ausgewählt hat bekommt der Controller eine Nachricht das Fahrzeug mit einer bestimmten ID anzuzeigen.

(3) Der Controller erzeugt ein VehicleModel und initialisiert es.

(4) Der Controller bindet die View und das Modell aneinander in dem er das VehicleModel der View übergibt - danach wird die View ausgeliefert.

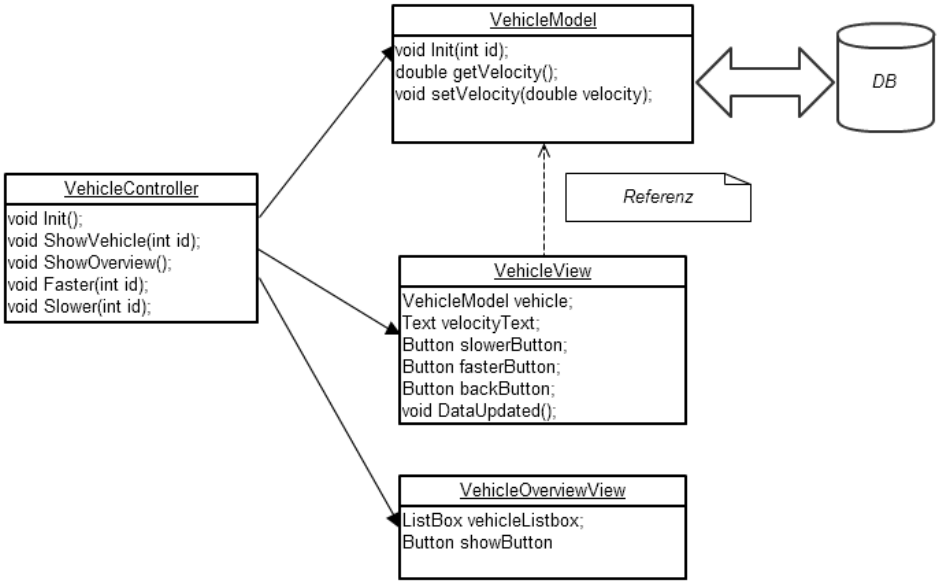


Abbildung 24: MVC Beispielapplikation

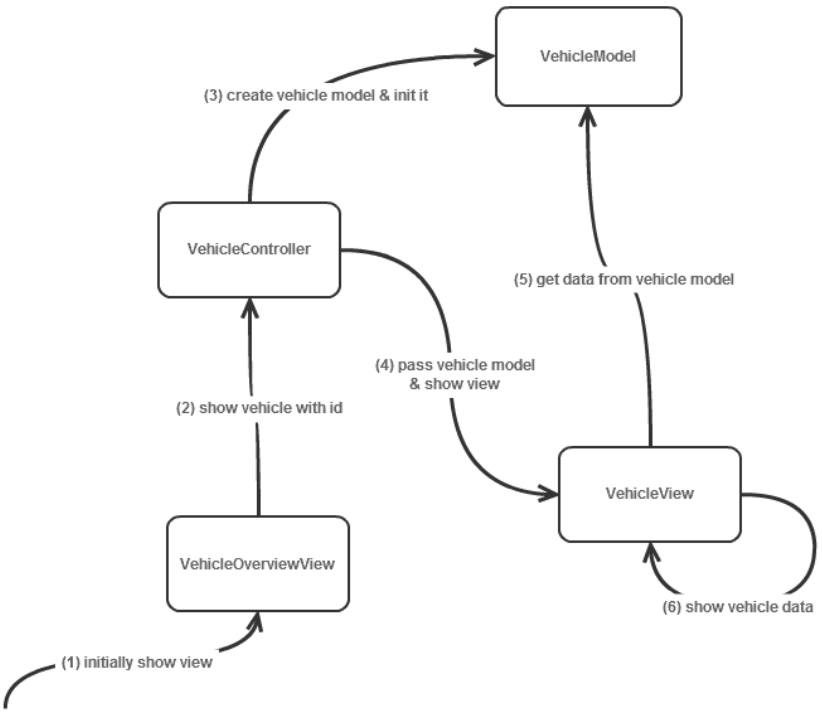


Abbildung 25: Initialisierung und Fahrzeugauswahl

(5 & 6) Die View holt sich vom VehicleModel (hier wird z.B. der Wert aus der Datenbank geholt) den aktuellen Geschwindigkeitswert und zeigt ihn an.

Geschwindigkeitsänderung (Abbildung 26)

(1) Der Controller bekommt je nach Klick auf den entsprechenden Button eine Nachricht die Geschwindigkeit des Fahrzeuges zu verändern.

(2) Im Controller befindet sich die Logik (Faster bzw. Slower) um das Modell zu verändern.

(3) Nachdem der Controller die Geschwindigkeit verändert hat wird der View mitgeteilt, dass sich der Status des Modells verändert hat.

(4) Der neue Geschwindigkeitswert wird in der View angezeigt bzw. aktuelle Daten vom Modell geholt.

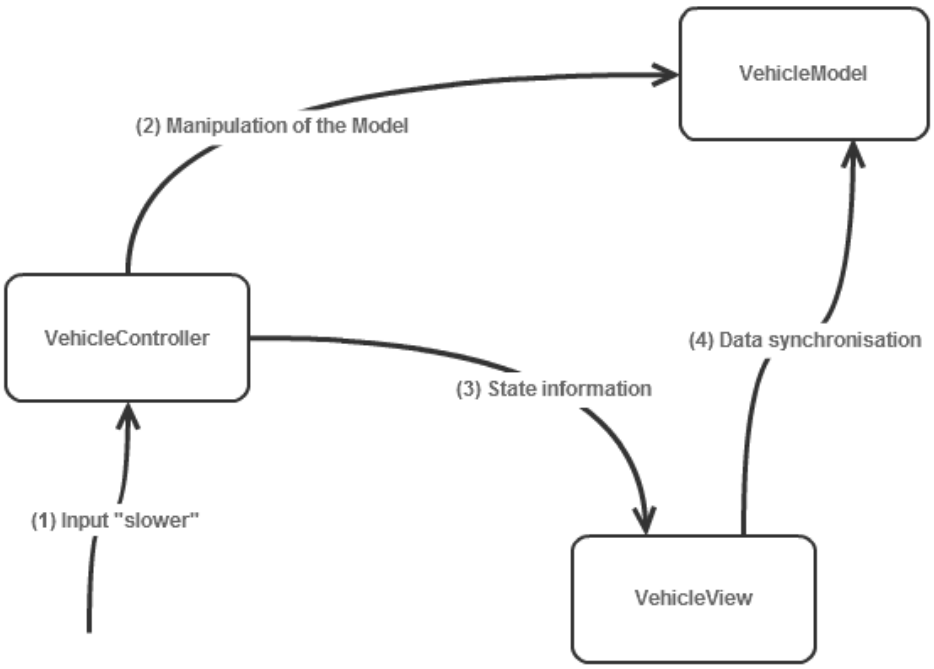


Abbildung 26: Geschwindigkeitsänderung

Lösungen im Detail

Bezugnehmend auf die Fragen in der Einleitung des Kapitels ergeben sich folgende Verantwortlichkeiten:

- *Logik* Der Controller enthält die Logik die das Modell verändert.
- *Synchronisation* Der Controller ist verantwortlich für die Synchronisation, in dem er der View mitteilt, dass sich die Daten verändert haben.

- *Datenhaltung* Die View hält permanent die Daten d.h. das Modell.
- *Kopplung* Da die View die Daten hält sind die Daten sehr eng daran gekoppelt - eine Änderung des Modells bedarf einer Änderung in der View.
- *Austauschbarkeit* In dieser Architektur können sowohl Controller als auch Views ohne Auswirkungen auf das Datenmodell ausgetauscht werden.

5.2.2 MVP

Das Model View Presenter Pattern erschien 1990 erstmals bei IBM und stellt eine Weiterentwicklung der MVC Architektur dar. MVP separiert nicht mehr View und Controller (hier nun Presenter) als unabhängige Bausteine und auch die Synchronisation der Daten passiert über den Presenter. [32] Abbildung 27 gibt Überblick über die Abhängigkeiten.

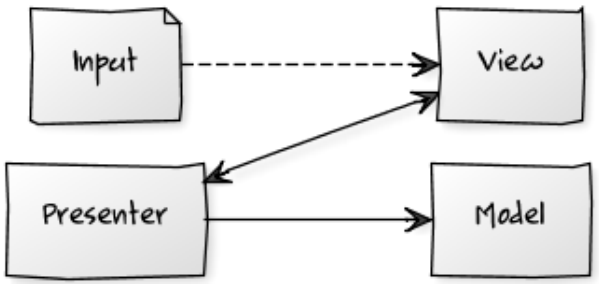


Abbildung 27: MVP Architektur

- Die View hängt wiederum vom dahinterliegenden Model ab - um diese Abhängigkeit etwas zu entkoppeln kann ein Interface eingeführt werden.
- Es existiert nicht mehr ein einzelner Controller, View und Presenter werden immer als Tupel gesehen die jeweils Referenzen in beide Richtungen haben und ihre Schnittstellen kennen.
- Der Input (z.B.: "Button clicked") geht nicht mehr an den Controller, sondern an die View - diese ruft danach entsprechende Methoden im Controller auf.
- Datenänderungen werden nicht von der View aktualisiert, sondern der Presenter bringt aktualisierte Daten in die View ein.
- Aus letzterem Grund und aus Gründen der besseren Testbarkeit wird die Funktionalität der View oft auch über ein Interface weg abstrahiert.
- Die UI-Logik um auf Userinput zu reagieren z.B. auf einen Button Klick steckt im Gegensatz zu MVC im Presenter.

Vergleichendes Beispiel

Aus dem in der Einführung dieses Kapitels vorgestellten Beispiel ergibt sich folgende MVP Architektur (Abbildung 28):

Daraus ergeben sich folgende Abläufe zwischen dem Model, den Views und den Presentern:

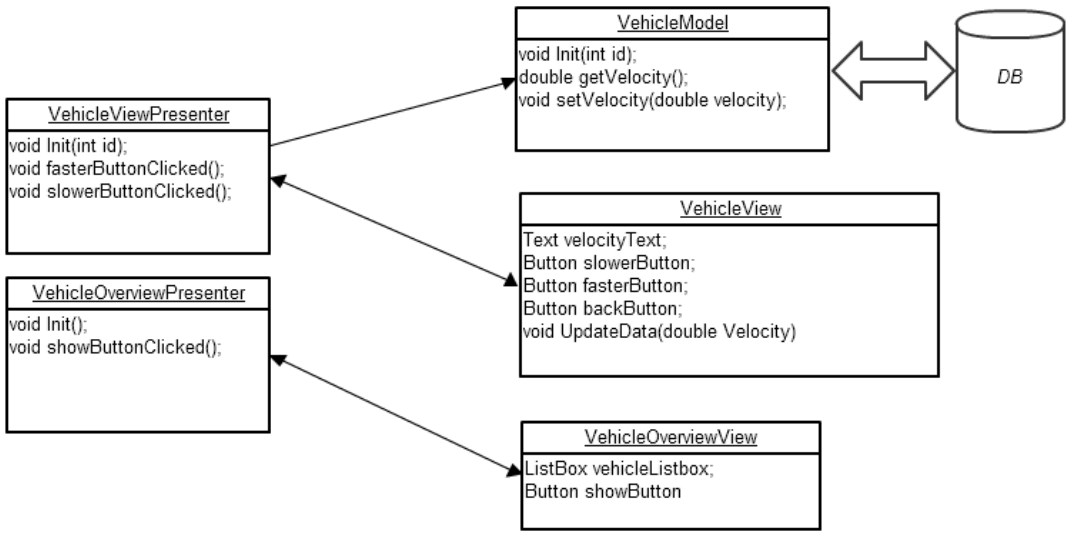


Abbildung 28: MVP Beispielapplikation

Initialisierung (Abbildung 29)

(1) Die VehicleOverviewView wird ausgeliefert - in ihr können verschiedene Fahrzeuge ausgewählt werden.

Fahrzeugauswahl (Abbildung 29)

(2) Die View bekommt als Input vom User das gewünschte Fahrzeug, übergibt diese Information an ihren dazugehörigen Presenter (VehicleOverviewPresenter).

(3) Der VehicleOverviewPresenter übergibt die Kontrolle dem VehicleViewPresenter - dieser ist nun zuständig für die Anzeige des Fahrzeugs.

(4) Der VehicleViewPresenter erzeugt initial das Modell und holt sich die Geschwindigkeitsdaten ab.

(5) Der VehicleViewPresenter übergibt die Geschwindigkeitsdaten der frisch erzeugten VehicleView und liefert diese aus.

(6) Die VehicleView zeigt die Geschwindigkeitsdaten an.

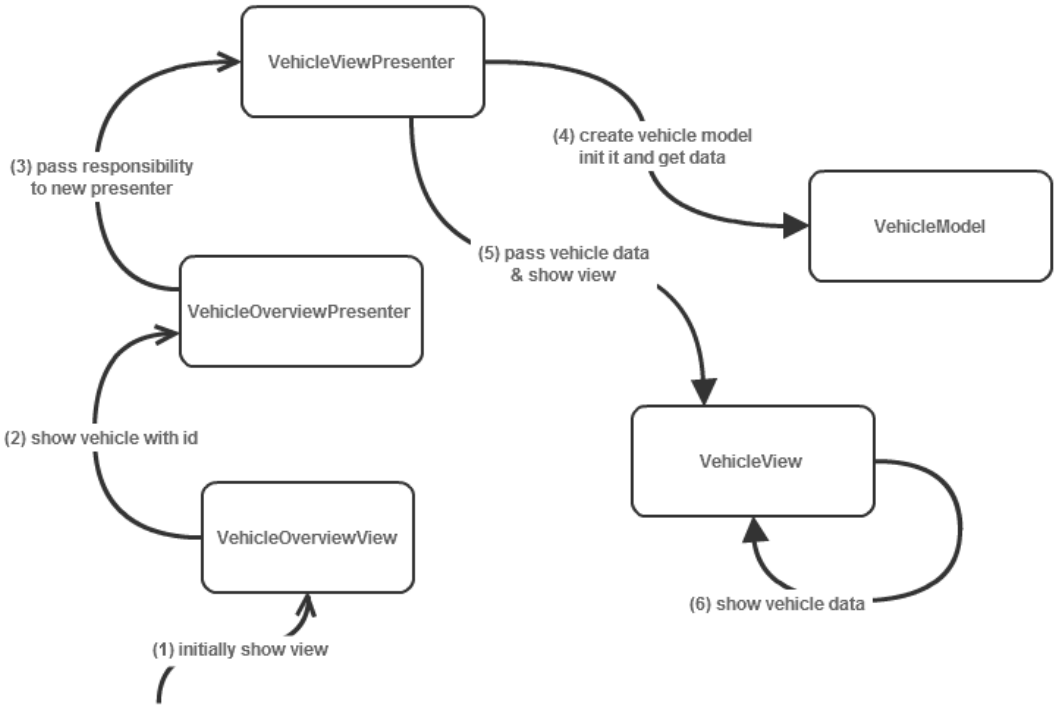


Abbildung 29: Initialisierung und Fahrzeugauswahl

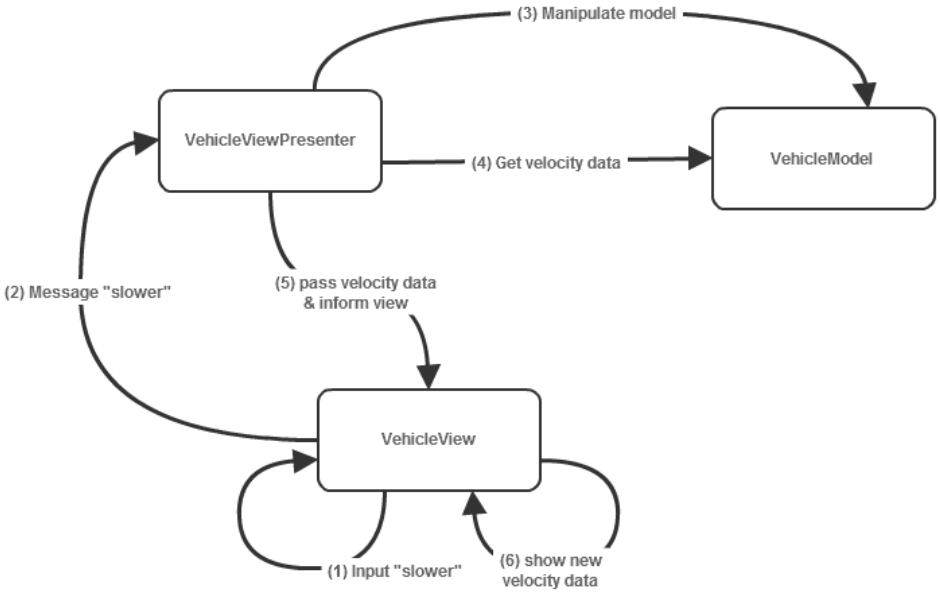


Abbildung 30: Geschwindigkeitsänderung

Geschwindigkeitsänderung (Abbildung 30)

(1) Der User gibt den Input "Langsamer" bzw. "Schneller" durch Klick auf einen Button.

(2) Die View ruft die ihr bekannte Methode im VehicleViewPresenter auf.

(3 & 4) Der Presenter manipuliert das Datenmodell, und holt danach für die View benötigte Daten (in unserem Fall die Geschwindigkeit ab).

(5) Der Presenter übergibt die neuen Daten an die View und informiert diese somit über eine Statusänderung.

(6) Die View zeigt die neuen Daten an.

Lösungen im Detail

Bei der MVP GUI Architektur ergeben sich folgende Verantwortlichkeiten:

- *Logik* Der Presenter enthält die Logik die das Modell verändert - jedoch gibt es im Gegensatz zu MVC pro View einen eigenen Presenter.
- *Synchronisation* Der Presenter ist verantwortlich für die Synchronisation, er teilt der View mit was sich geändert hat und somit auch wann sich was geändert hat.
- *Datenhaltung* Die View hält die Daten aus dem Modell - aber nicht mehr direkt, sondern nur mehr die von ihr benötigten Daten.
- *Kopplung* Die View ist nicht mehr direkt an das Modell gekoppelt. Jedoch sind View und Presenter hier stärker gekoppelt, da sie einander für die Interaktion kennen müssen.
- *Austauschbarkeit* Das Modell ist leichter austauschbarer wie bei MVC - die Logik wird anstatt in einem einzelnen Controller in verschiedenen Presentern gehalten.

5.2.3 MVVM

Das Model View Viewmodel Entwurfsmuster wurde 2005 von Softwarearchitekten bei Microsoft vorgestellt. Es ist eine Weiterentwicklung des Model View Presenter Patterns und verwendet spezielle Features wie Databinding oder Commands die in aktuellen GUI Frameworks unterstützt werden. [9]

- Im Gegensatz zu MVP existiert nur ein einziges ViewModel für verschiedene Views - es bereitet die Daten des Modells für die View auf, ist sozusagen eine Ansicht auf das Modell (vgl. ViewModel).
- Das ViewModel wird an die View einseitig gebunden, es hat keinen Zugriff auf Attribute oder Methoden der View - es kann lediglich die anzuzeigende View auswählen und aktivieren.

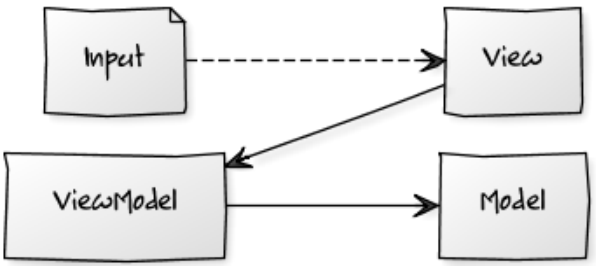


Abbildung 31: MVVM Architektur

- Der Input des Users wird so wie bei MVP von der View empfangen - da das View-Model an die View gebunden wurde kann es so Anfragen über einen Command Mechanismus weiterleiten.
- Das ViewModel selbst hat eine Referenz zum Modell und kann dieses manipulieren.
- Elemente in der View können an Objekte im ViewModel gebunden werden - die Synchronisation erfolgt über einen Databinding Mechanismus.
- Datenänderungen seitens des Modells werden an Objekte in der View reflektiert - in dem das ViewModel bekanntgibt das es eine Eigenschaft von sich verändert hat.
- Zusätzlich werden auch Änderungen in der GUI in das ViewModel bzw. in weiterer Folge in das Model reflektiert.

Vergleichendes Beispiel

Für das vergleichende Beispiel ergibt sich folgende MVPVM Architektur (Abbildung 32):

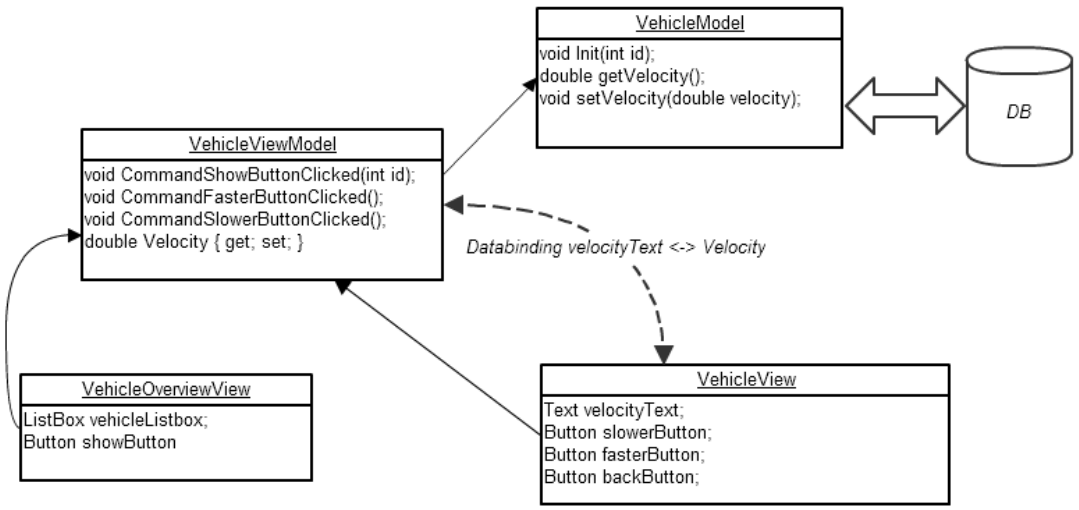


Abbildung 32: MVVM Beispielapplikation

Es ergeben sich für unsere Beispielapplikation folgende Abläufe:

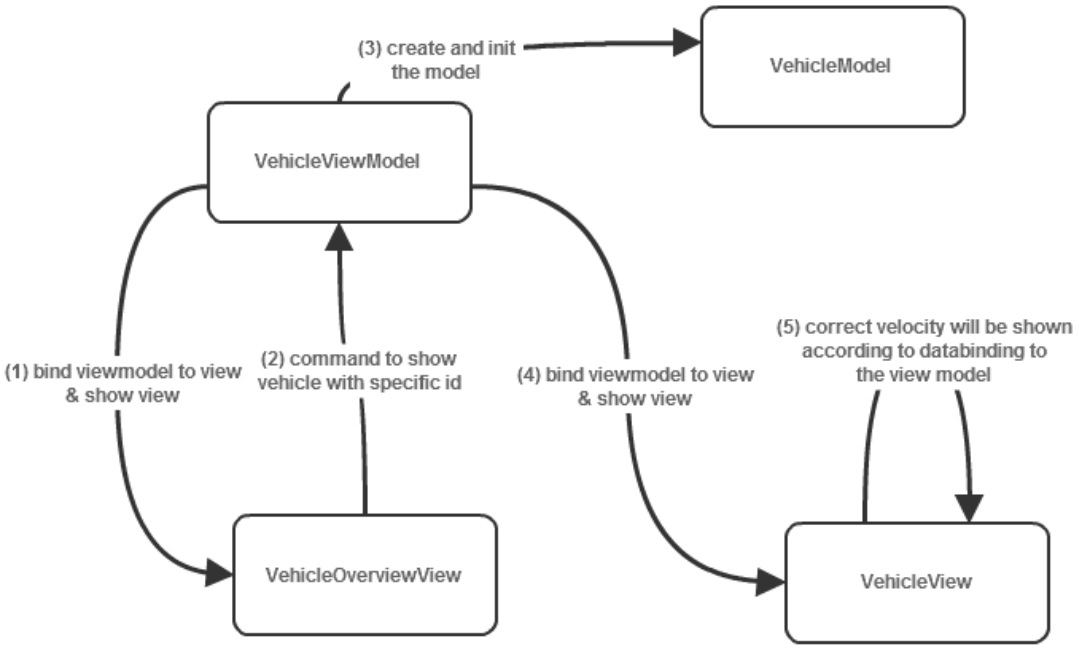


Abbildung 33: Initialisierung und Fahrzeugauswahl

Initialisierung (Abbildung 33)

(1) Das VehicleModel bindet sich selbst an die VehicleOverView und die VehicleOverview wird ausgeliefert. In ihr kann ein Fahrzeug ausgewählt werden.

Fahrzeugauswahl (Abbildung 33)

(2) Nach Klick auf den ShowButton wird ein Command aktiviert, das als Parameter die ID mit sich trägt. Da die View das ViewModel kennt wird das Command dementsprechend richtig aufgelöst und im VehicleViewModel ausgeführt.

(3) Das VehicleViewModel initialisiert ein neues VehicleModel und hält die Referenz darauf.

(4) Das VehicleViewModel bindet sich selbst an die VehicleView und liefert die View aus.

(5) Der velocityText in der View ist an das Velocity Property des ViewModels gebunden - über dieses Databinding wird die get-Methode des Velocity Properties aufgerufen, welche intern aus dem VehicleModel den richtigen Wert retourniert - somit wird automatisch in der VehicleView der richtige Wert angezeigt.

Geschwindigkeitsänderung (Abbildung 34)

(1) Die View bekommt über einen Button-Click den Userinput die Geschwindigkeit zu ändern.

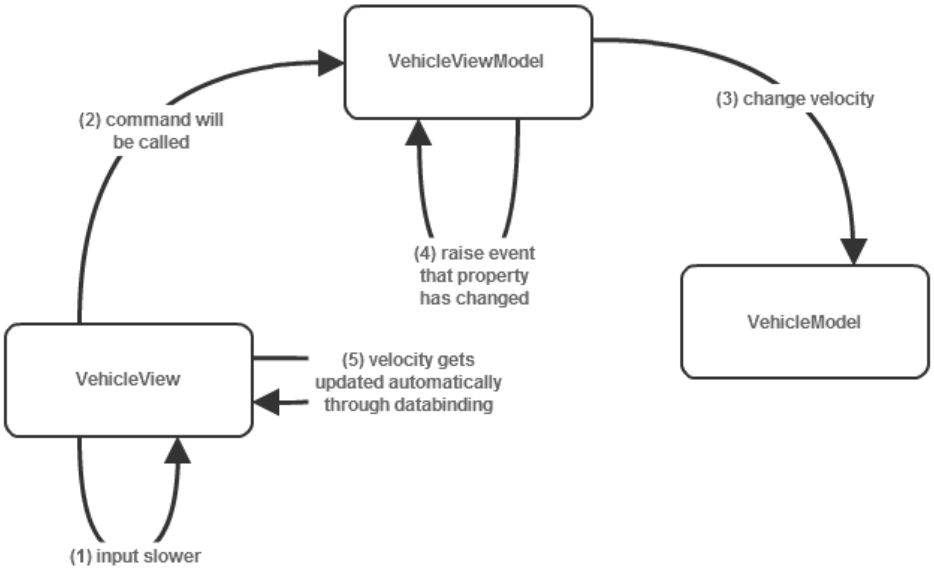


Abbildung 34: Geschwindigkeitsänderung

(2) Das dementsprechende Command wird aktiviert und vom Framework an das View-Model weitergeleitet.

(3) Das ViewModel manipuliert Daten im Modell.

(4) Das ViewModel gibt über einen Event Mechanismus, der für das Databinding notwendig ist, bekannt, dass sich das Velocity Property verändert hat.

(5) Über diesen Änderungsmechanismus des Databindings wird die Velocityänderung direkt in die View reflektiert und der neue Geschwindigkeitswert wird angezeigt.

Lösungen im Detail

Bei der MVVM GUI Architektur ergeben sich folgende Verantwortlichkeiten:

- *Logik* Die Logik zur Manipulation des Datenmodells liegt im ViewModel.
- *Synchronisation* Die Synchronisation erfolgt hier speziell über einen Databinding Mechanismus - je nach dem welche Seite (View oder ViewModel) Daten verändert, muss diese die andere Partei über eine Änderung informieren.
- *Datenhaltung* Das ViewModel hält sich die relevanten Daten - die View bindet direkt an Attribute im ViewModel.
- *Kopplung* Die View ist völlig entkoppelt vom Modell, die Kopplung zwischen View und ViewModel ist aufgrund von Command und Databinding Mechanismus sehr lose.
- *Austauschbarkeit* Aufgrund der zuvor erwähnten losen Kopplung lassen sich Views, das ViewModel oder das Model leicht austauschen, besonders wenn für das Model noch zusätzlich Interfaces verwendet werden.

5.2.4 Fazit

Ein direkter Vergleich der drei GUI Architekturen erscheint dahingehend nicht fair, da zwischen den jeweiligen Veröffentlichungen (1979, 1990, 2005) viele Jahre und doch einige Technologiesprünge stecken.

Vielmehr muss der Übergang von MVC über MVP zu MVVM als Evolution gesehen werden - hat man ein modernes GUI Framework zur Verfügung das Databinding und Commands erlaubt, so wird man zwischen den drei Architekturen kaum etwas anderes als MVVM wählen.

Hat man diese Technologien nicht zur Hand so muss man sich entscheiden solche Mechanismen nachzubauen oder zu MVC oder MVP zu greifen - je nach Anwendungszweck können diese Patterns auch ausreichen.

MVVM ist in der losen Kopplung und der fast automatischen Synchronisation von Datenänderungen den beiden anderen Architekturen überlegen - ein Grund auch warum dieses Pattern z.B. bei WPF Entwicklern so beliebt ist [9].

MVVM, und MVP haben auch einen großen Vorteil in der Testbarkeit gegenüber von MVC - durch die losere Kopplung können Teile durch Mockups ausgetauscht werden um z.B. bei MVVM die ViewModel Logik automatisiert testen zu können.

Es existieren aber auch sehr viele Abwandlungen der hier vorgestellten Patterns, und meist wird keines dieser Patterns zu 100 % bei der Implementierung eingehalten - viel zu oft entscheiden sich Entwickler zu kleinen Veränderungen die ihre Bedürfnisse besser befriedigen.

Und so sind die drei Vertreter nicht mehr als das was sie sind - Muster an die man sich halten "sollte" - es bleibt auch zu vermuten, dass hinsichtlich GUI Architekturen diese Evolution weiter geht und in Zukunft neue Formen sich durchsetzen werden.

6 Implementierung in der NTE CLM Plattform

Aufgabe des praktischen Teils dieser Arbeit war es, die in Kapitel 3 beschriebene Designumgebung zu implementieren und in die NTE CLM Plattform zu integrieren.

6.1 NTE CLM Überblick

Die Control Loop and Monitoring Plattform (kurz CLM) der Firma NTE Systems ermöglicht es Daten, die von Speicherprogrammierbaren Steuerungen (SPS) gelesen werden, auf einfachste Art und Weise zu visualisieren.

Dabei kann es sich, angefangen von Produktions- bis hin zu Gebäudedaten, um beliebige Inhalte handeln. Die Plattform besteht aus drei Teilen:

CLM System Designer

Diese Desktop Applikation erlaubt es Benutzern Anlagen zu verwalten und intuitiv eine ansprechende Visualisierung, Überwachung und Steuerung der Daten mittels Drag and Drop zu erstellen.

In diese Desktop Applikation wird die hier beschriebene und implementierte Designumgebung eingebettet, um Anlagenschemen zu erstellen und damit Visualisierung und Fernsteuerung von verteilten Anlagen zu ermöglichen.

Im CLM System Designer gibt es zusätzlich noch die Möglichkeit Diagramme für Historiendaten zu konfigurieren, Bilanzkennzahlen zu erstellen und regelbasierte Aktionen zu definieren.

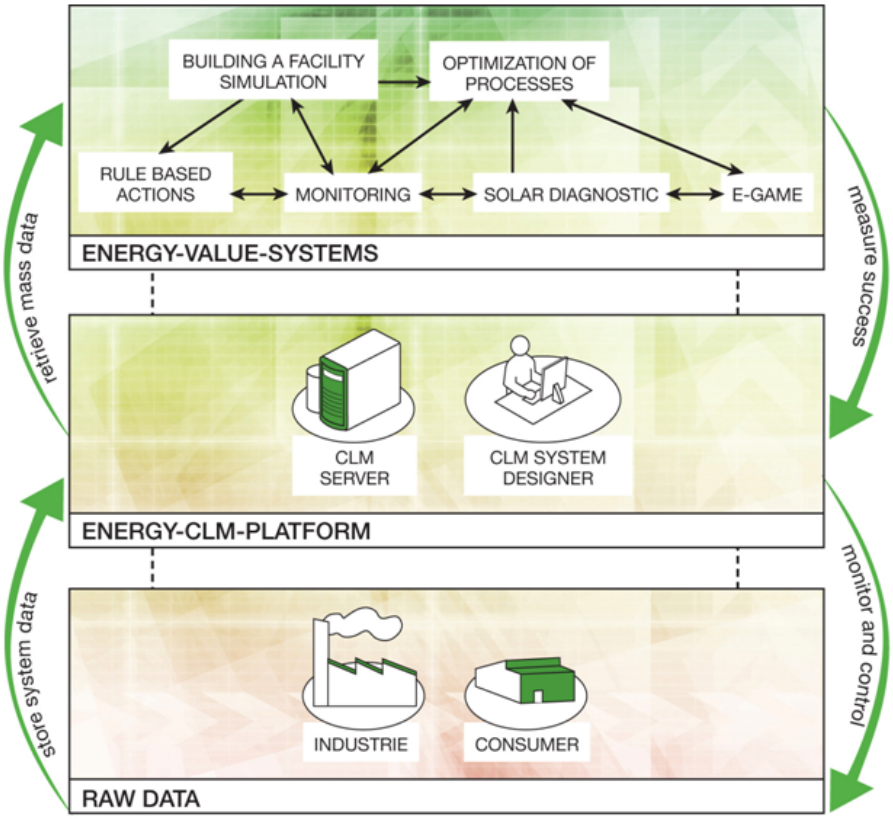


Abbildung 35: Die NTE CLM Plattform

CLM Server

Der CLM Server ist das Herz der Plattform - auf ihm werden alle Konfigurationen und Anlagendaten gespeichert. Dieser zentrale Kommunikationspunkt kümmert sich aber nicht nur um die Persistierung der Daten sondern ist auch durch Redundanz und Security für die Sicherung der Daten verantwortlich.

Über die OPC UA Schnittstelle ist der CLM Server auch mit Speicherprogrammierbaren Steuerungen verbunden, um zyklisch Daten zu lesen und im Fernsteuerungsfall Daten zu schreiben und somit direkt auf die dahinter liegenden verteilten Anlagen einzuwirken.

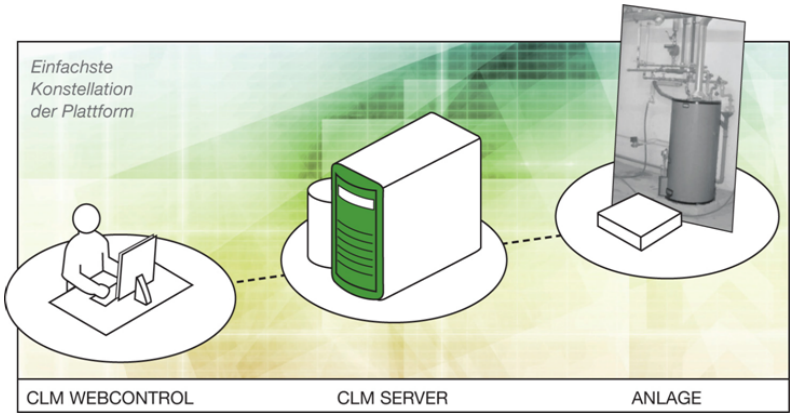


Abbildung 36: NTE WebControl

CLM WebControl

Das CLM WebControl ist eine Browseranwendung die es ermöglicht, überall auf der Welt alle Daten der verwalteten Anlagen zu sehen und auch zu bearbeiten. Schwerpunkt dieser Anwendung ist die Visualisierung von Historiendaten als auch Echtzeitdaten aber auch die Fernsteuerung ist in diesem Client möglich.

Die in den CLM System Designer integrierte Designumgebung hat somit auch Auswirkungen auf diese Webanwendung, da hier die über den System Designer erstellten Anlagenschemen auch angezeigt werden können.

6.1.1 Technologieentscheidungen

Die gesamte Plattform wurde mit dem .NET Framework realisiert - die Implementierung aller drei Teile wurde durch die Verwendung aktuellster .NET Frameworks und Tools bestmöglich unterstützt.

Hier ein kurzer Auszug aus den verwendeten Frameworks und Tools in den einzelnen Bereichen:

- **CLM System Designer**: WPF, WCF, Prism
- **CLM Server**: WCF, Entity Framework, IIS7 & AppFabric
- **CLM WebControl**: Silverlight, WCF, Prism

Aus den zuvor genannten Technologieentscheidungen wurde auch die Designumgebung mit der WPF (XAML) realisiert. Die Implementierung der Designumgebung wird ausgehend von der Basis (Implementierung der Designeritems) bis hin zur Spitze (Integration in den CLM System Designer) erläutert, jedoch werden hier zur besseren Lesbarkeit stark vereinfachte Codebeispiele präsentiert, die nicht den vollen Funktionsumfang der Implementierung, jedoch aber die dahinterliegenden Konzepte vorstellen.

6.2 Designeritems

Wie in den Anforderungen 3.2 beschrieben, besitzt ein Designeritem, das später zur Visualisierung oder Fernsteuerung verwendet wird

- ein **Erscheinungsbild**
- mehrere Eigenschaften (**Properties**) (= Daten)
- **Logik**

Durch die Verwendung von XAML und den beiden Mechanismen *Databinding* und *Datatemplates* (siehe 5.1.3) wurde versucht diese drei Teile bestmöglich zu entkoppeln. Für die Abarbeitung von Userinteraktionen (z.B. Mouse Events) stehen mehrere Möglichkeiten (siehe Kapitel 5.1.3) zur Verfügung.

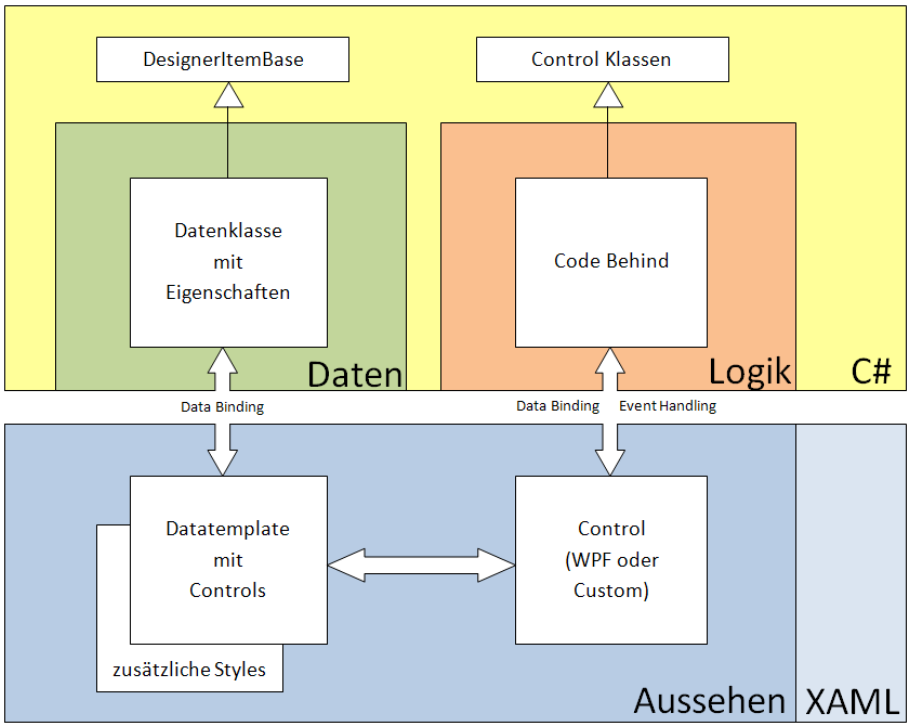


Abbildung 37: Designeritem Basisarchitektur

Es folgt an dieser Stelle eine Referenzimplementierung eines einfachen Designeritems das eine Zahl anzeigen soll, welches in einem Zusatzbeispiel um Logik angereichert wird.

6.2.1 Einfache Designeritems

Die Implementierung dieses Designeritems zur Anzeige einer Zahl besteht aus folgenden Dateien:

- NumberDesignerItem.cs (**Properties**)
- NumberDesignerItem.xaml (**Erscheinungsbild**)

```
using System.ComponentModel;
using System.Xml.Serialization;

namespace Nte.SolarChecker.Client.Wpf.Controls.DesignerItems
{
    [DesignerItem("NumberDesignerItem", version = 1.0,
       description = "Zahl", category = "Allgemein, icon = "
       numberDesignerItemIcon")]
    public class NumberDesignerItem : DesignerItemBase
    {
        public NumberDesignerItem()
        {
            this.Number = 0.0;
            this.Resizable = true;
            this.InitialHeight = 30;
            this.InitialWidth = 40;
        }

        [DisplayName("Zahl")]
        [BindableAttribute(true)]
        public double Number { get; set; }
    }
}
```

Listing 30: NumberDesignerItem.cs

Die Klasse NumberDesignerItem stellt eine einfache Datenklasse dar, die eine Eigenschaft besitzt: *Number*. Die Basisklasse DesignerItemBase stellt Eigenschaften zur Verfügung, die alle DesignerItems gemeinsam haben: initiale Größe, minimale Größe, Vergrößerung möglich sowie eine eindeutige ID.

Ziel dieser Implementierung war es die Klassen der Designeritems so einfach wie möglich zu halten, um später mit Hilfe eines Property Grids ohne Umwege die Eigenschaften von Objekten dieser Klassen 1 zu 1 editierbar zu machen. Die Annotationen *DisplayName* zeigen bereits diesen Zusammenhang - die Eigenschaften werden später unter dem in der Annotation angegebenen Namen angezeigt.

Zusätzlich bietet dieser gewählte Weg, Designeritems als schlichte Datenklassen zu sehen, die Möglichkeit mittels .NET Serializer einfach in XML und wieder zurück umzuwandeln.

Eine weitere Annotation findet sich am Anfang der Klasse - diese wird verwendet, um Designeritems dynamisch aus DLLs zu laden - somit hat die Toolbox die Möglichkeit diese zu finden und auch anzubieten.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx
   /2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/
   winfx/2006/xaml" xmlns:local="clr-namespace:Nte.SolarChecker.
   Client.Wpf.Controls.DesignerItems">

    <!-- Style for NumberDesignerItem -->
    <DataTemplate DataType="{x:Type local:NumberDesignerItem}">
        <Viewbox StretchDirection="Both"
                 Stretch="Fill">
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Number}" />
            </StackPanel>
        </Viewbox>
    </DataTemplate>

</ResourceDictionary>
```

Listing 31: NumberDesignerItem.xaml

Listing 31 zeigt das dazugehörige DataTemplate - wird ein Objekt vom Typ NumberDesignerItem z.B. in ein ContentControl platziert, so wird der darin definierte Style zur Anzeige des Designeritems verwendet.

Um die Zahl zusätzlich zu formatieren (z.B. auf zwei Kommastellen) wäre es beispielsweise möglich mit Hilfe eines IValueConverters, den man bei Bindings angeben kann, die Formatierung vorzunehmen. Somit kann unabhängig von der Datenklasse das Aussehen von bestimmten Properties weiter verändert werden. WPF bietet in diesem Fall noch weitere Möglichkeiten wie z.B. über Trigger auf Werte von Eigenschaften Styles im Template umzusetzen.

Ergänzend muss hier angemerkt werden, wird der Wert von Number geändert - so muss, da es sich um ein herkömmliches Property handelt, und nicht um ein DependencyProperty, über eine Schnittstelle mitgeteilt werden, dass sich dieser Wert geändert hat.

Dies kann explizit erfolgen, in dem man die *OnPropertyChanged* Methode der Basisklasse, welche das *INotifyPropertyChanged* Interface implementiert, nach dem setzen des Wertes aufruft, oder in dem man beim Setter der Eigenschaft die OnPropertyChanged Methode automatisch aufrufen lässt.

```
private double value;

[Bindable(true)]
```

```
[DisplayName("Wert")]
public double Value { get { return this.value; } set { this.value
    = value; OnPropertyChanged("Value"); } }
```

Listing 32: Implizite OnPropertyChanged Notification

6.2.2 Zusätzliche Logik

Das vorige Beispiel deckt die Anforderungen hinsichtlich Eigenschaften und Erscheinungsbild ab. Dies ist in vielen Fällen ausreichend, jedoch hat das Designeritem noch keinerlei Logik. Da das Designeritem eine reine Datenklasse ist, bietet es keinerlei Möglichkeit auf Interaktion, um z.B. auf Mouseevents (wie es WPF Controls bieten), zu reagieren.

Nehmen wir folgendes Beispiel: Ein Designeritem zeigt eine Zahl an, und beinhaltet einen Button um einen Wert zurück zu setzen. Die Logik für dieses Zurücksetzen wird in einem eigenen Usercontrol beschrieben:

- ResetButton.cs (**Logik**)
- ResetButton.xaml (**Erscheinungsbild**)

```
<UserControl x:Class="MyDesignerItems.ResetButton"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
                presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/
                xaml">
    <Button Content="Reset" Click="ButtonClick" />
</UserControl>
```

Listing 33: ResetButton.xaml

Dieses ResetButton UserControl besteht aus einer XAML Datei (ResetButton.xaml) und einer Code-Behind Datei (ResetButton.cs). Alternativ könnte man auch gleich von Button ableiten und in der OnMouseDown Event Routine das Reset durchführen.

Zusätzlich sind neben der direkten Angabe der Funktion *ButtonClick* auch alle anderen Varianten, welche im Kapitel 5.1.3 beschrieben wurden, um auf Events zu reagieren, möglich.

```
using System.Windows;
using System.Windows.Controls;

namespace MyDesignerItems
{
    public partial class ResetButton : UserControl
    {
```

```
        public ResetButton()
        {
            InitializeComponent();
        }

        public readonly static DependencyProperty ValueProperty =
            DependencyProperty.Register(
        "Value", typeof(double), typeof(ResetButton));

        public double Value
        {
            get { return (double)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }

        private void ButtonClick(object sender, RoutedEventArgs e
           )
        {
            //Logic
            this.Value = 50;
        }
    }
}
```

Listing 34: ResetButton.cs

Die Reset Methode repräsentiert in diesem einfachen Beispiel den Platzhalter für komplexere Logik - an dieser Stelle ist anzumerken, dass hier DependencyProperties manipuliert werden. Sie bilden die Grundlage, um Databinding anwenden zu können. Listing 35 zeigt das weitere Vorgehen:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx
   /2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/
   winfx/2006/xaml" xmlns:local="clr-namespace:Nte.SolarChecker.
   Client.Wpf.Controls.DesignerItems">

    <!-- Style for NumberDesignerItem -->
    <DataTemplate DataType="{x:Type local:NumberDesignerItem}">
        <Viewbox>
            <StackPanel>
                <TextBlock Text="{Binding Number}" />
                <local:ResetButton Value="{Binding Number, Mode=
                   TwoWay}" />
            </StackPanel>
        </Viewbox>
    </DataTemplate>

</ResourceDictionary>
```

Listing 35: NumberDesignerItem.xaml

Das Property Number der zuvor eingeführten NumberDesignerItem Klasse wird an das Value Property des ResetButton Control angebunden. Manipuliert die Logik des Reset Buttons ihr eigenes Property Value, so wird über das Binding die Wertänderung auf das Number Property des NumberDesignerItems reflektiert. Da auch der TextBlock an dasselbe Property gebunden ist, wird eine Änderung sofort angezeigt.

Umgekehrt funktioniert auch der Weg von der Datenklasse aus: wird hier ein Wert gesetzt (z.B. von außen durch eine Datenquelle), so werden die Änderungen an die beiden Properties Text bzw. Value weiter propagiert.

Dies ist nur ein sehr einfaches Beispiel, jedoch können die hier vorgezeigten Konzepte auf alle erdenklichen Visualisierungen angewandt werden - Eigenschaften der Datenklassen sind beidseitig an Eigenschaften in den darstellenden Controls verbunden.

Abbildung 38 zeigt eine im Rahmen der CLM Plattform entwickeltes Gauge Designeritem, dass über verschiedene Properties (Skala Minimum, Skala Maximum, Skala Schriftgröße, Beschriftung Schriftgröße, Wert, Skalierungsfaktor, Hintergrundfarbe) zur Laufzeit verändert werden kann.

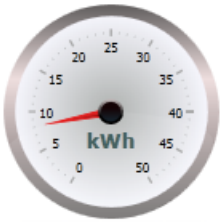


Abbildung 38: Gauge Designeritem

6.3 Workspace

Der Workspace ist im Grunde ein Canvas, auf dem Designeritems platziert werden können. Da es sich bei den Designeritems wie zuvor angemerkt um reine Datenklassen handelt, und nicht um Controls - wurden die Designeritems in eine ContentControl Klasse eingepackt *DesignerItemContentControl*, welche für die Interaktion auf der Designeroberfläche (vergrößern, verschieben, ...) verwendet wird.

6.3.1 DesignerItemContentControl

Diese ContentControl Klasse verfügt über ein Attribut Content, welches Objekte vom Typ Object annimmt. In der Style Definition der DesignerItemContentControl Klasse kann mit Hilfe des *ContentPresenter* Elements dieses Objekt ausgegeben werden - das entsprechende DataTemplate und somit das Erscheinungsbild des Designeritems wird an dieser Stelle präsentiert.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx
   /2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/
   winfx/2006/xaml" xmlns:controls="clr-namespace:Nte.SolarChecker
   .Client.Wpf.Controls.Designer.Controls">

    <!-- Style for DesignerItemContentControl -->
    <Style TargetType="{x:Type controls:
       DesignerItemContentControl}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type controls:
                   DesignerItemContentControl}">
                        <ContentPresenter Name="PART_DesignerItem
                           " /> <!-- CONTENT -->
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>

</ResourceDictionary>
```

Listing 36: DesignerItemContentControl.xaml

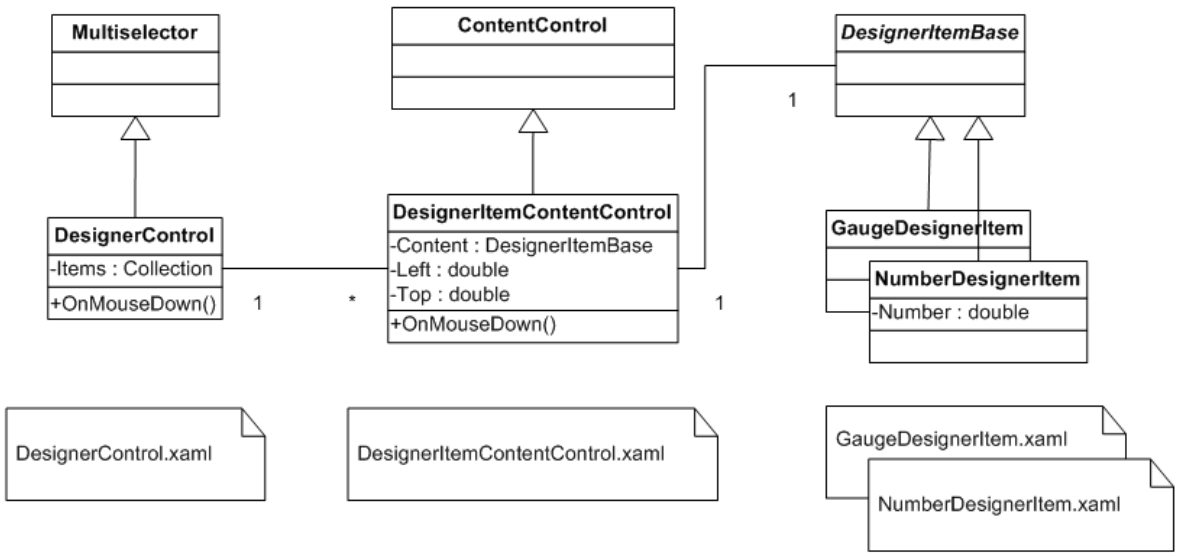


Abbildung 39: Designeritem, ContentControl und DesignerControl Zusammenhänge

Über dieses Wrapping wird aus einer einfachen Datenklasse ein Control - die DesignerItemContentControlKlasse besitzt zusätzliche Attribute, die für die Platzierung auf dem Canvas benötigt werden (Top, Left) als auch Eventhandler wenn auf das Designeritem geklickt wurde, um das entsprechende Designeritem auf dem Workspace zu selektieren.

6.3.2 DesignerControl

Es existiert weder ein frei verfügbares noch kommerzielles Control für WPF das einem Workspace ähnelt - aus diesem Grund wurde dieses Control (*DesignerControl*) selbst entwickelt.

Abbildung 39 zeigt den Zusammenhang der drei Klassen DesignerControl, DesignerItemContentControl und DesignerItemBase.

Die DesignerItemContentControl Objekte werde über eine Style Definition auf einen Canvas platziert - über Databinding an die Top und Left Eigenschaften der DesignerItemContentControl Klasse erfolgt die Anbindung an die Attached Properties *Canvas.Left* und *Canvas.Top*.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx
   /2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/
   winfx/2006/xaml" xmlns:controls="clr-namespace:Nte.SolarChecker
   .Client.Wpf.Controls.Designer.Controls">

    <!-- Style for DesignerControl -->
    <Style TargetType="{x:Type controls:DesignerControl}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type controls:
                   DesignerControl}">
                    <Canvas IsItemsHost="True"
                            Name="designerCanvas"
                            AllowDrop="True">
                    </Canvas>
                </ControlTemplate>
            </Setter.Value>
        </Setter>

        <!-- bind item.Left & item.Top to the canvas -->
        <Setter Property="ItemContainerStyle">
            <Setter.Value>
                <Style TargetType="{x:Type controls:
                   DesignerItemContentControl}">
                    <Setter Property="Canvas.Left"
                            Value="{Binding Path=Left,
                               RelativeSource={RelativeSource Self
                               }}" />
                    <Setter Property="Canvas.Top"
                            Value="{Binding Path=Top,
                               RelativeSource={RelativeSource Self
                               }}" />
                </Style>
            </Setter.Value>
        </Setter>
```

```
    </Style>
</ResourceDictionary>
```

Listing 37: DesignerControl.xaml

Abbildung 40 zeigt die Kette der Abhängigkeiten zwischen den einzelnen XAML Dateien: Im DesignerControl.xaml wird ein Canvas definiert, der als Host für die Items Objekte dient. Als Style für diese Items wird ein DesignerItemContentControl herangezogen, welcher wiederum das Data Template des entsprechenden Designeritems präsentiert.

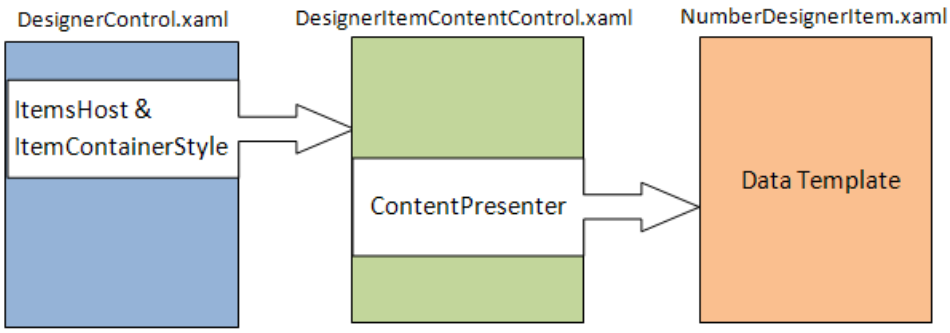


Abbildung 40: XAML Zusammenhänge

Somit ist der Weg wie eine einfache Datenklasse auf dem Canvas platziert wird beschrieben - das DesignerControl verfügt jedoch über mehr Funktionalität: Selektieren, Verschieben und Vergrößern von Designeritems ist die Hauptaufgabe dieses Controls.

Einfache und Rubberband Selektion

Das DesignerControl abgeleitet von MultiSelector bietet wie der Name vermuten lässt einen Selektionsmechanismus - jedoch nur auf logischer Ebene. Über *SelectedItems.Add* lassen sich Objekte die in der Items-Collection liegen in eine eigene Collection befördern, mit der in weiterer Folge weiter gearbeitet werden kann.

Für die einfache Selektion reicht es in der MouseDown Funktion des ContentControl Wrappers des Designeritems die entsprechende Methode aufrufen - hier lassen sich auch Spezialfälle wie gedrückte Control Taste etc. abdecken.

Etwas aufwändiger ist die geforderte Rubberband Selektion, bei der man durch aufziehen eines Rechtecks mehrere Designeritems gleichzeitig auswählen kann. Hierfür wurde das Adorner Konzept von WPF verwendet, dass es ermöglicht eine "unsichtbare" Zeichenebene einzufügen, die das DesignerControl dekoriert, und auf der das Rubberband gezeichnet werden kann.

Im Konstruktor wird ein Adorner erzeugt, der sich über das DesignerControl legt, beim MouseDown Event wird das Rechteck initialisiert, beim MouseMove Event aktualisiert,

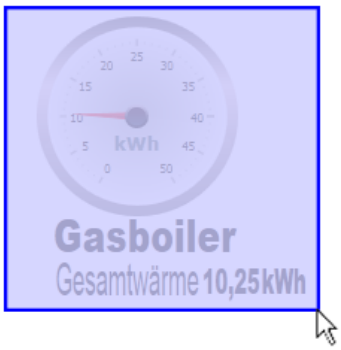


Abbildung 41: Rubberband Selektion

und beim MouseDown Event wird über eine geometrische Abfrage herausgefunden, welche Designeritems sich innerhalb des aufgezogenen Rechtecks befinden und diese anschließend selektiert.

Selektierte Designeritems werden mit einer Umrandung angezeigt - 42 zeigt ein selektiertes Designeritem in der Designumgebung.

Platzierung

Das Verschieben von Designeritems erfolgt direkt in den Mouse Event Handler Funktionen des DesignerControls - in den OnMouseMove bzw. OnMouseDown Events werden die Properties Left bzw. Top der selektierten DesignerItemContentControls gesetzt. Über den Databinding Mechanismus, werden die aktualisierten Werte weiter an den Canvas übergeben, welche die Designeritems an die neuen Stellen platziert.

Zusätzlich wurde eine Grid Funktion integriert, die es erlaubt Designeritems nur an bestimmten Positionen zu platzieren - dieser Grid kann aktiviert/deaktiviert werden und in seiner Feinheit verändert werden.

Größenveränderung

Um Größenveränderungen auf Designeritems durchführen zu können werden spezielle Controls benötigt, die es erlauben durch deren Verschiebung auf Mouse Input zu reagieren und dementsprechend das Designeritem in der Größe zu verändern.

Diese Resize Controls, bekannt aus verschiedenen Designumgebungen (siehe Abbildung 42) - werden wiederum als Adorner über das Designeritem gelegt - und werden nur angezeigt, wenn das entsprechende Designeritem selektiert ist.

Die 8 gezeichneten Kreise die mit Hilfe eines Adorners über das Designeritem gelegt werden sind selbst wieder Controls und können somit wiederum Mouse Events behandeln. Je nach gewähltem / benütztem Resize Thumb werden die Top, Left bzw. Height, Width Eigenschaften des Designeritem Objektes neu gesetzt.

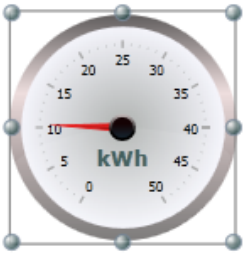


Abbildung 42: Selektiertes Designeritem mit Resize Adorner

Durch den Databinding Mechanismus wiederum werden die Änderungen dieser Eigenschaften wieder direkt in die GUI reflektiert und die Größenänderung sofort angezeigt.

Zusätzliche Features

In der DesignerControl Komponente wurden zusätzliche Features wie eine Zoomfunktionalität, zwei Betriebsmodi (Designmode Ein/Aus) welcher es erlaubt die Manipulation von Designeritems zu erlauben/sperren, Raster Ein/Aus, Änderung der Rastergröße sowie das Ausrichten aller Elemente am Raster implementiert.

6.3.3 MVVM Architektur

Aufbauend auf den zuvor eingeführten Klassen wurde eine MVVM Architektur eingeführt um die Daten (Items Collection), die Anzeige (DesignerControl) und weitere Logik zur Manipulation von Designeritems sauber zu trennen.

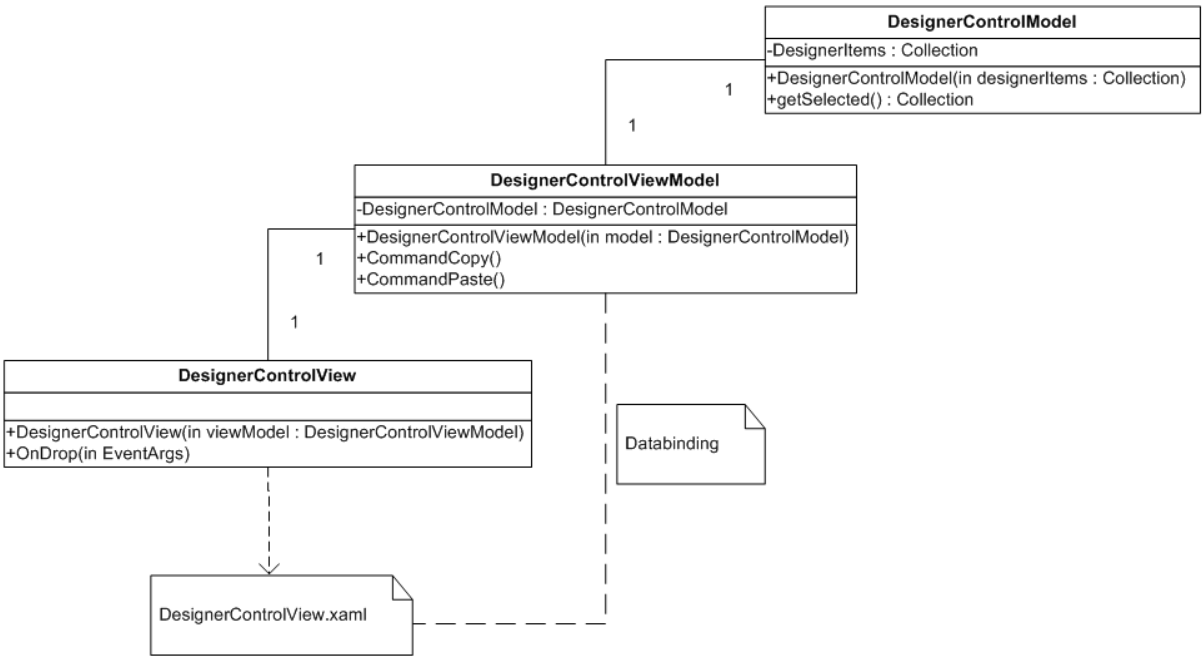


Abbildung 43: Designer MVVM Architektur

Die Initialisierung der MVVM Architektur erfolgt in folgenden Schritten:

Erzeugung des Datenmodells

Als erstes wird das DesignerControlModel erzeugt, dies kann eine leere Designeritem Collection sein - oder eine Deserialisierung eines bestehenden Modells. Listing 38 zeigt ein vereinfachtes XML, das mit Hilfe von .NET XML Writern aus einem bestehenden Modell erzeugt wurde und zur Persistierung einer Visualisierung dient.

```
<FacilitySchema>
  <DesignerItems>
    <DesignerItem>
      <Left>40</Left>
      <Top>20</Top>
      <Height>50</Height>
      <Width>75</Width>
      <Content Type="Nte.SolarChecker.Client.Wpf.Controls.
         DesignerItems.NumberDesignerItem">
       <NumberDesignerItem>
        <Resizable>true</Resizable>
        <Number>0.01</Number>
         ...
       </NumberDesignerItem>
      </Content>
    </DesignerItem>
    ...
  </DesignerItems>
</FacilitySchema>
```

Listing 38: Serialisiertes Modell

Erzeugung des Viewmodels

Im Viewmodel steckt die Logik, um das Modell zu manipulieren welche später mittels Commands angebunden werden. Das ViewModel wird einfach durch Übergabe des Modells konstruiert.

Erzeugung der View

Die View ist das letzte Glied der Kette - es wird konstruiert, in dem das ViewModel im Konstruktor mit übergeben wird. Wichtig ist hier, dass der DataContext auf das ViewModel gesetzt wird, damit später Databindings an das Viewmodel weitergeleitet werden.

```
public DesignerControlView(DesignerControlViewModel viewModel)
{
    InitializeComponent();
```

```
    this.viewModel = viewModel;
    this.DataContext = viewModel;
    ...
}
```

Listing 39: View Initialisierung

Databinding der View

Das ItemsSource Property des DesignerControls, das in dieser View platziert wird, wird über Databinding an die DesignerItems Collection des DesignerModels angebunden.

```
<UserControl x:Class="Nte.SolarChecker.Client.Wpf.Controls.
   Designer.DesignerControlView" xmlns="http://schemas.microsoft.
   com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.
   microsoft.com/winfx/2006/xaml" xmlns:controls="clr-namespace:
   Nte.SolarChecker.Client.Wpf.Controls.Designer.Controls">

    <ScrollViewer Background="White"
                  HorizontalScrollBarVisibility="Auto"
                  VerticalScrollBarVisibility="Auto">
        <controls:DesignerControl ItemsSource="{Binding
           DesignerControlModel.DesignerItems}" />
    </ScrollViewer>

</UserControl>
```

Listing 40: DesignerView.xaml

Somit steht die MVVM Architektur: werden nun Designeritems im Modell manipuliert, so werden Änderungen in das DesignerControl und somit in die View reflektiert. Werden umgekehrt Eigenschaften einzelner Designeritems über das DesignerControl in der View manipuliert, so finden sich die Änderungen auch im Modell wieder.

6.3.4 Zusätzliche Logik

Im DesignerControlViewModel wurde zusätzliche Logik für folgende Aktionen implementiert:

- **Kopieren, Einfügen, Entfernen**
- **Gruppieren / Gruppierung auflösen**
- **In den Vordergrund, Vorwärts, Rückwärts, In den Hintergrund**
- **Undo / Redo**

Diese Zusatzfunktionalität wird durch Commands seitens des ViewModels angeboten, dass direkt auf dem DesignerControlModel arbeitet. Diese Commands können z.B. durch ein Kontextmenü oder eine Toolbar einfach angesprochen werden und sorgen für eine saubere Trennung von Logik und Logikaufrufen.

```
public DesignerControlViewModel(DesignerControlModel model)
{
    ...
    this.Copy = new DelegateCommand<object>(Copy_Executed,
       Copy_Enabled);
    ...
}

private void Copy_Executed(object o)
{
    //Copy Logic
}

private bool Copy_Enabled(object o)
{
    return this.DesignerControlModel.getSelectedDesignerItems().
       Count() > 0 ? true : false;
}
```

Listing 41: DesignerControlViewModel Commands

Eine Besonderheit der hier verwendeten DelegateCommands ist es, dass eine Funktion angegeben werden kann, die überprüft, ob das Command überhaupt sinnvollerweise ausgeführt werden kann. Somit können Toolbar oder Kontextmenüeinträge dementsprechend inaktiv geschaltet werden.

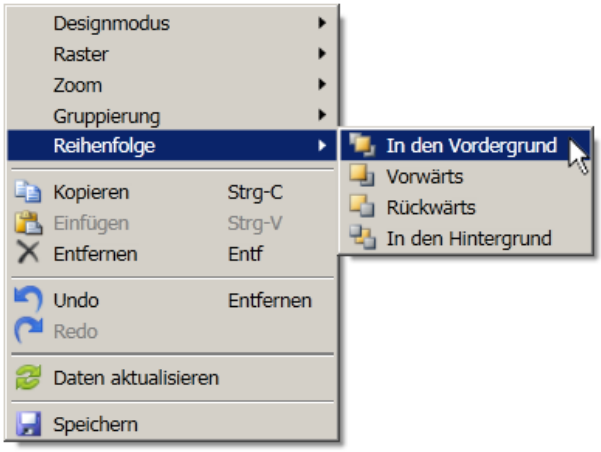


Abbildung 44: DesignerControlView Kontextmenü

6.4 Toolbar

Anstatt einer Toolbar wurde in das Control ein Kontextmenü integriert, welches an die zuvor erklärten Commands angebunden werden kann. Eine Toolbar selbst würde denselben Mechanismus nützen, und unterscheidet sich nur durch die View von dieser Implementierung.

Die Anbindung eines Kontextmenüeintrages an das entsprechende Command zeigt Listing 42.

```
<ContextMenu>
    ...
    <MenuItem Header="Kopieren"
              Command="{Binding Copy}"
              InputGestureText="Strg-C">
        <MenuItem.Icon>
            <Image Source="Images/Copy.png"
                   Width="16" />
        </MenuItem.Icon>
    </MenuItem>
    ...
</ContextMenu>
```

Listing 42: Command Anbindung

6.5 Toolbox

Die Toolbox präsentiert dem User alle verfügbaren Designeritems und bietet mittels Drag and Drop die Möglichkeit Designeritems auf den Designerworkspace zu ziehen.

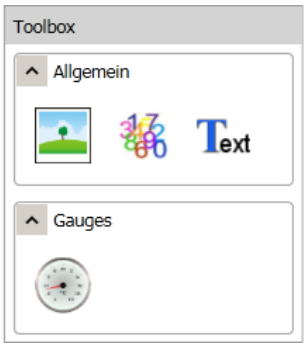


Abbildung 45: Designer Toolbox

Um größtmögliche Erweiterbarkeit zu gewährleisten besteht diese Toolbar nicht aus einer vordefinierten Anzahl von Designeritems, sondern sie ist mit einem Pluginmechanismus ausgestattet, um leicht neue Designeritems hinzufügen zu können.

Wie bereits in Listing 30 angedeutet, werden Designeritems mit einer speziellen Annotation versehen:

[DesignerItem(" NumberDesignerItem", version = 1.0, description = " Zahl", category = " Allgemein", icon = " numberDesignerItemIcon")]

Beim Start der Applikation werden sämtliche DLLs mit Hilfe des Reflection Mechanismus nach dieser speziellen Designeritem Annotation durchsucht. Gefundene Designeritems werden über das Category Attribut in die richtige Kategorie eingeordnet und mit einem Icon versehen.

Diese Toolboxeinträge selbst sind nur vereinfachte visuelle Repräsentationen der richtigen Designeritems - beim Drag and Drop Vorgang wird nur der Klassenname an das DesignerControl mit übergeben. Damit das DesignerControl aus dem Klassennamen das richtige Designeritem instanziieren kann wurde eine Factory Klasse eingeführt, die von der Toolbox befüllt und von der DesignerControlView verwendet werden kann.

Abbildung 46 zeigt schematisch die zuvor beschriebenen Abläufe.

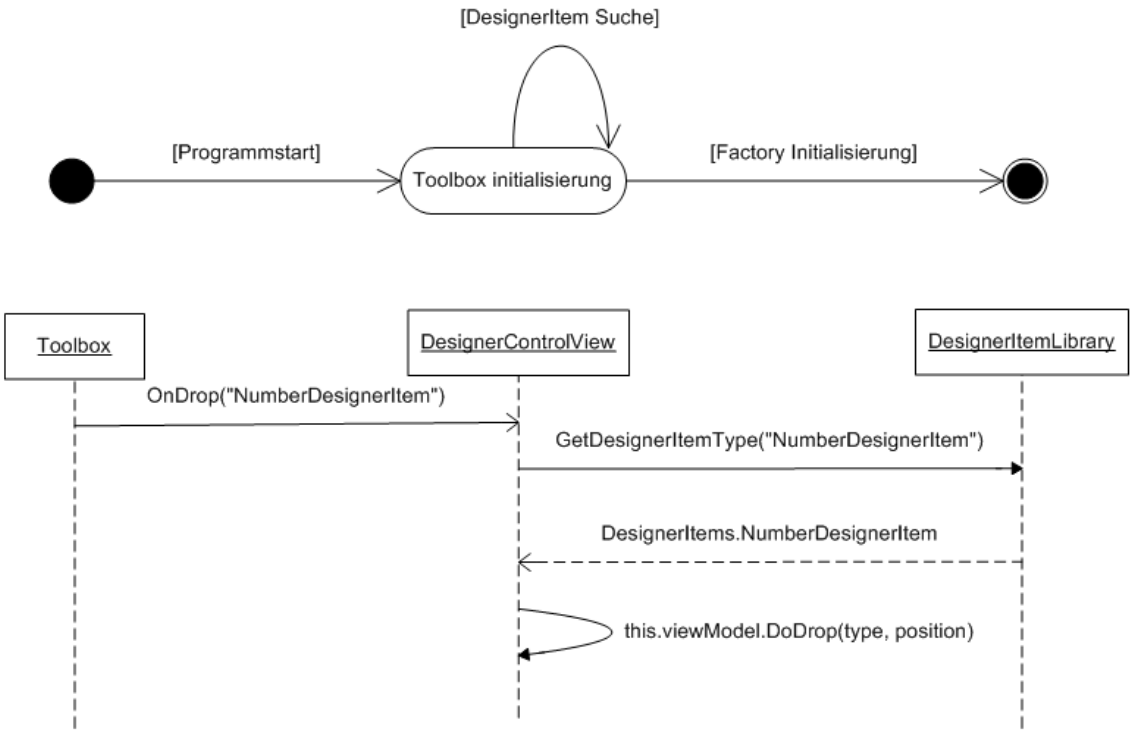


Abbildung 46: Toolbox Initialisierung und Drag and Drop Ablauf

Die Toolbox wurde wie die DesignerControl ebenfalls mit einer eigenen View und einem ViewModel ausgestattet - auf Implementierungsdetails wird an dieser Stelle jedoch verzichtet.

6.6 Property Grid

WPF selbst bietet kein eigenes Property Grid Modul, jedoch ist es möglich durch Verwendung des WPF WindowsFormsHost das Propertygrid Control von Windows Forms einzubinden.

```
<Window x:Class="Nte.Client.Wpf.ServiceManager.Modules.
   PropertyGrid.Views.PropertyGridView" xmlns="http://schemas.
   microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://
   schemas.microsoft.com/winfx/2006/xaml" xmlns:swf="clr-namespace
   :System.Windows.Forms;assembly=System.Windows.Forms">
    <Grid>
      <WindowsFormsHost>
        <swf:PropertyGrid x:Name="propertyGrid"/>
      </WindowsFormsHost>
    </Grid>
</Window>
```

Listing 43: Verwendung des Windows Forms Property Grid

Um die Eigenschaften eines Objekts im Propertygrid anzuzeigen bzw. editierbar zu machen reicht folgender programmatischer Aufruf:

this.View.propertyGrid.SelectedObjects = someObject;

Abbildung 48 zeigt ein im Propertygrid selektiertes Designeritem.

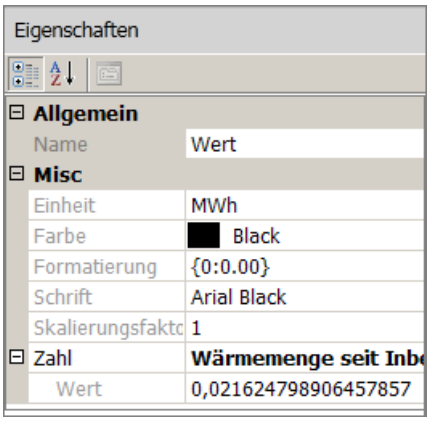


Abbildung 47: Designer PropertyGrid

Auf die Präsentation der einzelnen Eigenschaften eines Designeritems kann wie in Listing 44 gezeigt Einfluss genommen werden. Neben der Sichtbarkeit, dem angezeigten Namen können hier auch Editoren für die entsprechende Eigenschaft festgelegt werden.

```
[BrowsableAttribute(false)]
public bool Resizable { get; set; }

[DisplayName("Bild")]
[Editor(typeof(FileToByteArrayEditor), typeof(UITypeEditor))]
[TypeConverter(typeof(CustomTypeConverter))]
public byte[] ImageData { get; set; }

[DisplayName("Wert")]
[BindableAttribute(true)]
public double Score { get; set; }
```

Listing 44: Designeritem Propertygrid Annotationsbeispiele

Vorteil bei der Verwendung des Windows Forms Propertygrid ist es, dass man bereits auf eine Basis von Editoren zurück greifen kann - es ist jedoch auch möglich durch Ableitung von *UITypeEditor* selbst Editoren einzubinden wie z.B. der hier angedeutete *FileToByteArrayEditor*.

Das Propertygrid wurde so wie die beiden anderen Controls (DesignerControl, ToolboxControl) ebenfalls wieder in eine MVVM Architektur gepackt.

6.7 Datenanbindung

Die bis hier erläuterte Implementierung bezog sich auf die rein visuellen Möglichkeiten der Designumgebung. In weiterer Folge wird nun hier die Datenbasis eingeführt, die es mit den Designeritems zu visualisieren gilt.

6.7.1 Datenbasis

Wie im Einführungskapitel erklärt ist es Hauptaufgabe der Applikation verteilte Anlagen zu visualisieren und fernzusteuern. Die Datenbasis besteht aus Kunden die Anlagen besitzen, welche Steuereinheiten haben die wiederum Datenquellen enthalten.

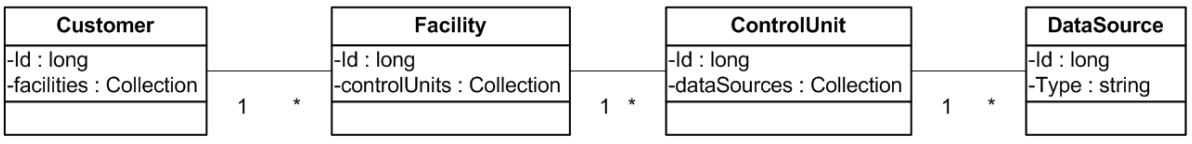


Abbildung 48: Datenbasis

Bei den ControlUnits handelt es sich um abstrakte Stellvertreter für Speicherprogrammierbaren Steuerungen - Variablen die auf diesen Steuerungen gelesen bzw. geschrieben werden, sind in der Datenbasis als DataSource abgelegt und können als Typen jeweils Integer, String, Real, DateTime, Blob, Bool und Arrays davon annehmen.

6.7.2 WCF Services

Der gesamte Zugriff auf die Datenbasis wurde mit Hilfe von Proxy Klassen, die über WCF Services am CLM Server angesprochen werden können, weggekapselt. Die für die Designumgebung relevanten WCF Services sind:

- **XMLStorageService**: zur Speicherung des Modells am CLM Server.
- **DataSourceService**: zum abholen aller verfügbarer Datenquellen einer Anlage.
- **LiveDataService**: um Livedaten von Datenquellen zu erhalten.
- **CommandService**: um Daten auf Datenquellen zu schreiben.

Die Designumgebung wird somit bereits mit einer Facility aufgerufen, und braucht nur mehr bei Bedarf die entsprechenden Proxy Methoden der Services aufrufen um an die gewünschten Daten zu gelangen.

Der CLM Server ist dafür verantwortlich, zyklisch Daten von den Speicherprogrammierbaren Steuerungen abzuholen, und diese über Webservices zur Verfügung zu stellen. Er nutzt das OPC-UA Protokoll um mit den ControlUnits zu kommunizieren.

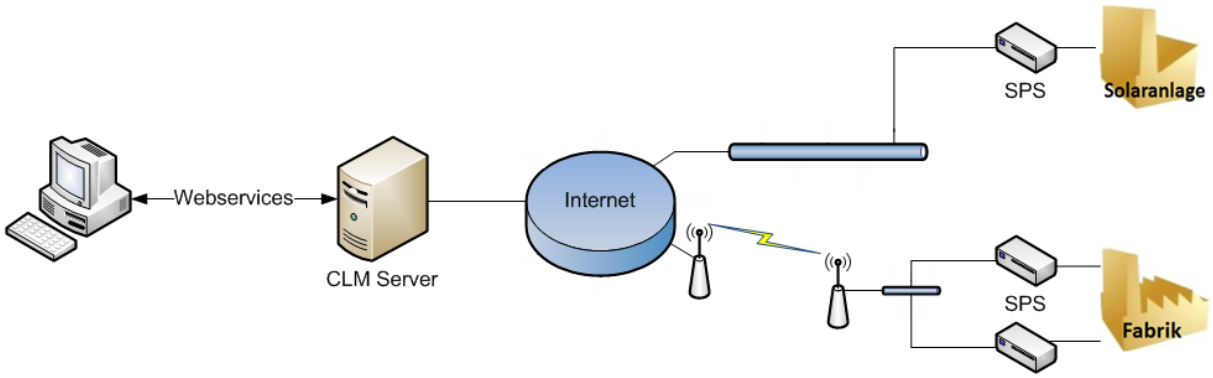


Abbildung 49: CLM Server Interaktion

6.7.3 Designeritem

Um Variablen von ControlUnits visualisieren und fernsteuern zu können, müssen Eigenschaften von Designeritems zu Datenquellen gebunden werden. Hierzu verfügt die DesignerItemBase Basisklasse über eine Collection, welche beschreibt welches Property mit welcher Datenquelle verknüpft wurde. Listing 45 zeigt die programmatische als auch bereits serialisierte Form dieser Datenanbindung.

```
public abstract class DesignerItemBase : BindableTypeDescriptor,
   INotifyPropertyChanged {
...
[BrowsableAttribute(false)]
public List<DesignerItemBinding> Bindings { get; set; } }
...
}

<FacilitySchema>
  <DesignerItems>
    <DesignerItem>
      ...
      <Content Type="Nte.SolarChecker.Client.Wpf.Controls.
         DesignerItems.NumberDesignerItem">
       <NumberDesignerItem>
         <Bindings>
          <DesignerItemBinding>
           <Property>Number</Property>
           <ID>38</ID>
          </DesignerItemBinding>
         ...
        </Binding>
         ...
       </NumberDesignerItem>
      </Content>
    </DesignerItem>
    ...
  </DesignerItems>
</FacilitySchema>
```

Listing 45: DesignerItem Eigenschaften Datenquellenanbindung

Über diesen im Modell verfügbaren Zusammenhang können in weiterer Folge Eigenschaften den Wert einer Datenquelle annehmen.

6.7.4 Propertygrid

Die Verknüpfung von Eigenschaften zu Datenquellen durch den Benutzer erfolgt über das Propertygrid. Das Propertygrid ermittelt anzeigbare Eigenschaften von Designeritems zur Laufzeit und gewinnt über den Reflection Mechanismus den PropertyDescriptor dieser Eigenschaft welcher den Typ, Namen und andere Attribute beschreibt.

Mit Hilfe dieses PropertyDescriptors kann das Propertygrid die entsprechende Eigenschaft präsentieren, editieren und validieren. Die Number Eigenschaft aus dem vorigen Beispiel würde ohne unser zu tun als einfache Zahl präsentiert werden.

Der Propertygrid wurde für die Datenanbindungszwecke dahingehend modifiziert, dass er Eigenschaften die mit dem *[BindableAttribute(true)]* annotiert wurden, dekoriert und beim entsprechenden Property einen Editor zur Auswahl zulässt (siehe Abbildung 50).

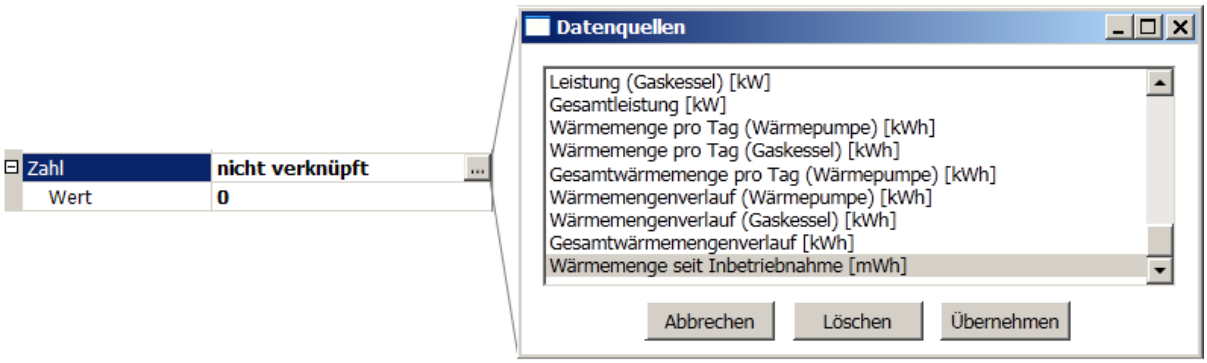


Abbildung 50: Propertygrid Datenanbindung

Diese Dekoration wurde folgendermaßen erreicht: die DesignerItemBase Klasse implementiert das Interface *ICustomTypeDescriptor* mit dessen Hilfe es möglich ist, Properties eines Objektes zur Laufzeit zu manipulieren. In unserem Fall wurde der originale PropertyDescriptor der Eigenschaft welches mit dem Bindable Attribut annotiert wurde entfernt und durch einen selbst geschriebenen PropertyDescriptor ersetzt, der sowohl Datenquelle als auch den Wert des ursprünglichen Properties enthält.

Dieser selbstgeschriebene PropertyDescriptor ist für die Speicherung der Datenquellenverknüpfung in die Bindings Collection des entsprechenden Designeritems zuständig.

Listing 46 zeigt die Eckpunkte der Implementierung dieser Dekoration:

```
public abstract class BindableTypeDescriptor :
   ICustomTypeDescriptor {
...
PropertyDescriptorCollection ICustomTypeDescriptor.GetProperties(
    Attribute[] attributes)
    {
        PropertyDescriptorCollection props = new
           PropertyDescriptorCollection(null);
        foreach (PropertyDescriptor prop in TypeDescriptor.
           GetProperties(
           this, attributes, true)) {

            bool isBindable = false;

            //wurde das Property mit dem Bindable Attribut
               versehen? -> isBindable setzen
```

```
            ...

            if (isBindable)
            {
                //Dekoration des originalen Property Descriptors
                   durch einen eigenen
                ArrayList attrs = new ArrayList(prop.Attributes);
                //Definition des Datenquellen Editors
                attrs.Add(new EditorAttribute(typeof(
                   BindingEditor), typeof(UITypeEditor)));
                props.Add(new BindablePropertyDescriptor(prop,
                   this, (Attribute[])attrs.ToArray(typeof(
                   Attribute))));
            }
            else
            {
                //Behalte originalen Property Descriptor bei
                ...
            }
        }
        return props;
    }
...
}

public class BindablePropertyDescriptor : PropertyDescriptor {
...

    public override object GetValue(object component) {
        DesignerItemBase c = (DesignerItemBase)component;
        //hole Datenquelle aus Bindings Collection -> dataSource
        ...
        return dataSource;
    }
...

    public override void SetValue(object component, object value)
    {
        DesignerItemBase c = (DesignerItemBase)component;
        DesignerItemBinding dataSource = (DesignerItemBinding)
           value;
        //speichere dataSource in Bindings Collection
        ...
    }
}
```

Listing 46: DesignerItem Bindable Property Dekoration

Durch diese interne Dekoration müssen Entwickler von Designeritems sich mit dem gesamten Datenbindungsmechanismus nicht auseinander setzen - eine einfache Annotation

[BindableAttribute(true)] genügt, und schon kann die Eigenschaft der DesignerItem Klasse komfortabel an Datenquellen angebunden werden.

6.7.5 Datenupdate

Über den zuvor hergestellten Zusammenhang zwischen DesignerItem Properties und DataSources (siehe 51) hat das DesignerControlModel die Möglichkeit Livedaten von einem WCF Proxy Service von Datenquellen anzufordern, und die erhaltenen Werte als Wert für das angebundene Property zu setzen.

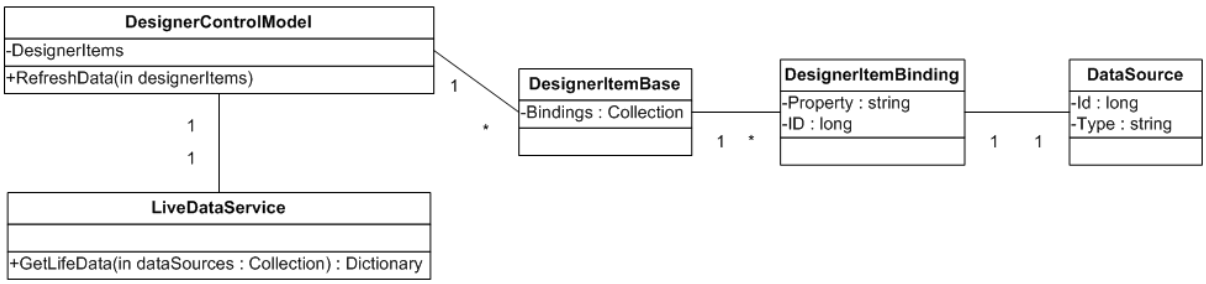


Abbildung 51: Designer Datenanbindung Zusammenhang

Dieses Update erfolgt entweder über ein im Kontextmenü verfügbares Command "Daten aktualisieren" oder kann zyklisch im Hintergrund im Sekundentakt erfolgen um Werte aktuell von der dahinterliegenden Anlage zu visualisieren.

Da die Designeritem Objekte ihre Eigenschaften direkt mittels DataBinding an das XAML DataTemplate angebunden haben, wird die Wertänderung sofort angezeigt.

Anzumerken ist, dass die Livedaten als Object zurückgeliefert werden, da man aber über den Zusammenhang mit der Datenquelle den Typ kennt, und auch den Typ des zu setzenden Properties, so kann an dieser Stelle eine geeignete Konvertierung erfolgen oder in Folge einer Inkompatibilität andere Maßnahmen durchgeführt werden.

6.7.6 Fernsteuerung

Der umgekehrte Weg des Datenupdates, das Schreiben von Werten auf eine Datenquelle, wird von Designeritems beschritten, die Fernsteueraufgaben übernehmen (Wählscheiben, Schalter, ...).

Zu diesem Zweck muss ein Property einer DesignerItem Datenklasse wiederum mit dem Bindable Attribut versehen werden. Zusätzlich wird aber bei diesem Attribut angegeben, dass es sich um eine Verbindung in beide Richtungen handelt - siehe Listing 49.

```
private double score;

[DisplayName("Wert")]
[BindableAttribute(true, BindingDirection.TwoWay)]
public double Score {
    get { return this.score; }
    set { this.score = value; OnPropertyChanged("Score"); }
}
```

Listing 47: Designeritem Annotation im Fernsteuerfall

Wie schon in Kapitel 6.2.1 erklärt muss hier explizit beim Setzen des Wertes ein Event erzeugt werden, um höherer Ebenen informieren zu können, dass sich etwas geändert hat.

Die weitere Strategie um auf Wertänderungen zu reagieren sieht nun folgendermaßen aus:

(1) Die DesignerItemsCollection wurde als ObservableCollection gehalten, d.h. es werden automatisch Events erzeugt wenn ein Element der Collection hinzugefügt oder entfernt wurde. Das DesignerControlViewModel registriert bei einem solchen Event einen Event-Handler der beim PropertyChanged Event des Designeritems aufgerufen wird.

```
DesignerItems.CollectionChanged +=
   NotifyCollectionChangedEventHandler(OnItemsChanged)

//Event Handler auf DesignerItems Collection Changed Events
void OnItemsChanged(object sender,
   NotifyCollectionChangedEventArgs e) {

    if (e.Action == NotifyCollectionChangedAction.Add) {
        foreach (DesignerItemBase item in e.NewItems) {
            item.PropertyChanged += new
               PropertyChangedEventHandler(OnPropertyChanged);
        }
    } else if (e.Action == NotifyCollectionChangedAction.Remove)
       {
        foreach (DesignerItemBase item in e.OldItems) {
            item.PropertyChanged -= new
               PropertyChangedEventHandler(OnPropertyChanged);
        }
    }
}
```

Listing 48: Registrierung des PropertyChanged Event Handlers

(2) Trifft der Event ein, so wird nachgesehen, ob es sich beim Property um ein Bindable Property handelt, dass auch TwoWay Binding erlaubt.

```
void OnPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
    PropertyDescriptor propDesc = TypeDescriptor.GetProperties(
       sender).Find(e.PropertyName, false);
    BindableAttribute attrib = (BindableAttribute)propDesc.
       Attributes[typeof(BindableAttribute)];
    if (attrib.Direction == BindingDirection.TwoWay) {
        //Aufruf des Datenservices
    }
}
```

Listing 49: Behandlung PropertyChanged Event Handlers

(3) Ist dies der Fall, wird der neue Wert auf die verknüpfte Datenquelle geschrieben.

Abbildung 52 zeigt ein im Rahmen des NTE CLM Projekts entwickeltes Designeritem zur Fernsteuerung.

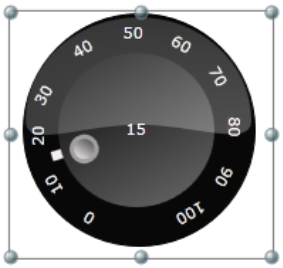


Abbildung 52: Designeritem zur Fernsteuerung

6.8 Gesamtintegration

Bis zu diesem Punkt wurden Module (DesignerControl, Toolbox und Propertygrid) als auch Services (XMLStorageService, DataSourceService, LiveDataService und CommandService) nur einzeln präsentiert. Im Rahmen einer lauffähigen Gesamtapplikation wird aber ein Mechanismus benötigt, der die Module und Services miteinander kommunizieren lässt.

Zusätzlich existieren hier noch nicht genannte Module die für die Designumgebung notwendig sind z.B. um eine Anlage auszuwählen. Als Basisframework für diese Gesamtintegration wurde das Framework Prism [33], dass vom Patterns & Practices Team von Microsoft stammt, verwendet.

6.8.1 Prism

Prism bietet Unterstützung bei der Entwicklung zusammengesetzter Anwendungen, in dem es die Möglichkeit bietet, Teile der Applikation zu modularisieren und gemeinsam Services zu nützen.

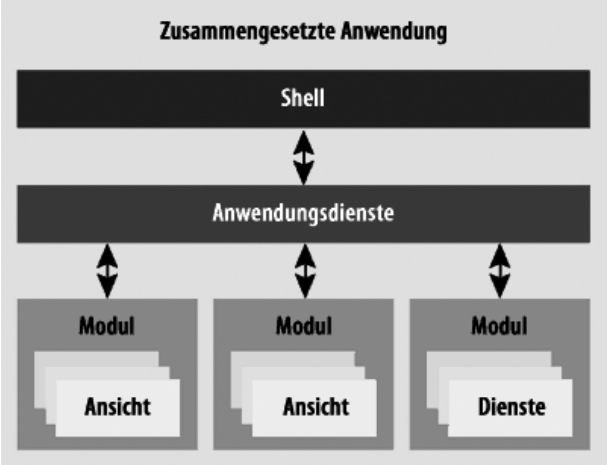


Abbildung 53: Prism Architektur

Um die Abhängigkeiten der einzelnen Module bestmöglich zu entkoppeln nutzt Prism den Unity Application Block [34] - ein Dependency Injection Container der dafür verwendet werden kann um z.B. einzelnen Views benötigte Services im Konstruktor mitzugeben, ohne diese selbst in der Applikation herumzureichen.

Zusätzlich bietet Prism einen EventAggregator-Dienst an, mit dem es möglich ist sich zu Events anzumelden bzw. diese zu publizieren, um eine einfache Kommunikation unter den Modulen zu gewährleisten.

6.8.2 Architektur

Abbildung 54 zeigt ein vereinfachtes Diagramm der Applikationsarchitektur - hier werden nur die Abhängigkeiten des Workspace Moduls gezeigt. Die anderen Module wie z.B. PropertyGridModule, ToolboxModule, ... verfügen je nach Wesen ebenfalls Abhängigkeiten zu den einzelnen Service.

Bereits aus diesem Diagramm lässt sich erahnen, dass es sehr aufwändig wäre, stets selbst die einzelnen Services an die einzelnen Views herum zu reichen - Prism bietet mit dem Unity Container einen komfortablen Mechanismus, der hier nun genauer besprochen wird.

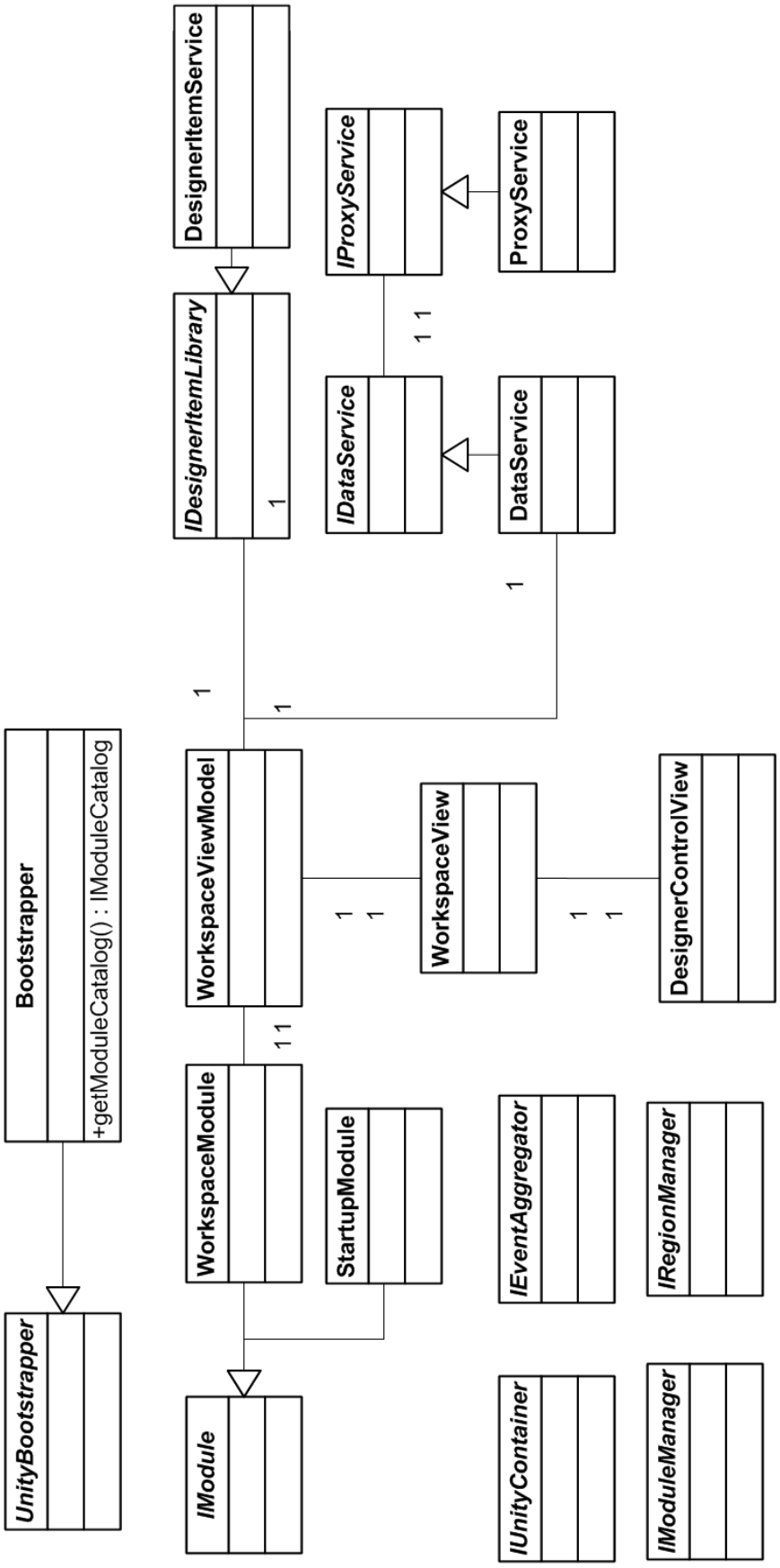


Abbildung 54: Applikationsarchitektur

Initialisierung

Einstiegspunkt der Applikation ist die Klasse Bootstrapper. Sie ist für die Initialisierung der später verwendeten Module zuständig ist. Sie überschreibt die Methode *getModuleCatalog* in der alle benötigten Module registriert werden und gibt auch explizit an, welches Modul als erstes geladen wird.

```
protected override IModuleCatalog GetModuleCatalog()
{
    ModuleCatalog catalog = new ModuleCatalog()
        .AddModule(typeof(StartupModule))
        ...
        .AddModule(typeof(WorkspaceModule), InitializationMode.
           OnDemand)
        .AddModule(typeof(ToolboxModule), InitializationMode.
           OnDemand)
        .AddModule(typeof(PropertyGridModule), InitializationMode
           .OnDemand);

    return catalog;
}
```

Listing 50: Prism Modul Initialisierung

Zusätzlich wird eine zweite Methode *CreateShell* überschrieben, welche die initiale View zurückgibt - diese wird von der Applikation als erstes angezeigt. In dieser View werden Regionen definiert, in der später Module platziert werden können, z.B:

<ContentControl prism:RegionManager.RegionName="WorkspaceRegion" />

Wie Listing 50 zeigt, ist das StartUpModule nicht mit einem zusätzlichen *OnDemand* Parameter versehen - es wird als erstes geladen, in ihm passiert die Initialisierung der Services und die benötigten Module werden geladen.

```
public StartupModule(IUnityContainer container, IRegionManager
   regionManager)
{
    this.container = container;
    this.regionManager = regionManager;
}

public void Initialize()
{
    this.container.RegisterType<IProxyService, ProxyService>(new
       ContainerControlledLifetimeManager());
    this.container.RegisterType<IDataService, DataService>(new
       ContainerControlledLifetimeManager());
```

```
    this.container.RegisterType<IDesignerItemLibrary,
       DesignerItemService>(new ContainerControlledLifetimeManager
       ());

    //show login screen
    StartupViewModel viewModel = this.container.Resolve<
       StartupViewModel>();
    IRegion mainRegion = this.regionManager.Regions["
       ApplicationRegion"];
    mainRegion.Add(viewModel.View);
}
```

Listing 51: StartupModule

Nach dem nun alle Services und Module bekannt sind, kann die Applikation entscheiden, wann welches Modul präsentiert werden soll. Ist z.B. das Login erfolgreich so kann mittels Aufruf von:

this.moduleManager.LoadModule(" WorkspaceModule");

ein Module gezielt nachgeladen werden. Das Modul selbst kann in seiner Initialisierungsfunktion wiederum selbst wieder ViewModels erzeugen und Views auf Regionen platzieren.

Hervorzuheben ist hier der Dependency Injection Mechanismus - benötigt z.B. das Workspace-ViewModel im Konstruktor ein DataService, so wird dieses automatisch vom Unity Container mit übergeben.

public DesignerWorkspaceItemViewModel(IDataService dataService, ...)

Kommunikation

Da nun alle Module an die entsprechenden Services angebunden sind fehlt nur mehr ein Mechanismus, mit dem die einzelnen Services untereinander kommunizieren können. Auch hier bietet Prism eine fertige Lösung an.

Prism stellt Events zur Verfügung, die man publishen und subscriben kann. Im konkreten Beispiel möchte das Propertygrid informiert werden, wenn im Workspace ein Designeritem selektiert wurde. Listing 52 zeigt die dafür gewählte Lösung.

```
//Eventklasse
public class SelectedWorkspaceItemChangedEvent :
   CompositePresentationEvent<DesignerItemBase> { ... }

//Event Subscription mit Callback Funktion
this.eventAggregator.GetEvent<SelectedWorkspaceItemChangedEvent
   >().Subscribe(this.SelectedWorkspaceItemChanged);
```

```
//Callback Funktion
private void SelectedWorkspaceItemChanged(DesignerItemBase item)

//Event Publishing
this.eventAggregator.GetEvent<SelectedWorkspaceItemChangedEvent
   >().Publish(designerItem);
```

Listing 52: Prism Events

7 Zusammenfassung

Die für die Implementierung ausgewählten Technologien und Konzepte haben sich für die Umsetzung der Designumgebung zur Informationsvisualisierung und Fernsteuerung von verteilten Anlagen hervorragend bewährt.

Die Windows Presentation Foundation stellt bereits durch ihre mit ausgelieferten Komponenten eine sehr gute Basis zur GUI Entwicklung dar. In Verbindung mit den grafischen Möglichkeiten die die WPF bietet, steht ein sehr mächtiger GUI Framework, zur Entwicklung von Designeritems zur Visualisierung und Fernsteuerung, zur Verfügung.

Die Stylemöglichkeiten von WPF durch XAML und spezielle darauf aufbauende Tools für Designer (Microsoft Expression Blend [35]) erlauben es Nicht-Entwicklern Komponenten in ihrem Erscheinungsbild zu verändern und zu gestalten. Auf diesem Weg können Experten auf diesem Gebiet direkt in den Softwareentwicklungsprozess eingebunden werden.

Über die verschiedenen Mechanismen die die WPF zur Verfügung stellt ist eine Trennung von Logik und Design einfach möglich. Dies wirkt sich nicht nur auf die Lesbarkeit und Wartbarkeit des Sourcecodes aus - die Wiederverwendbarkeit einzelner Komponenten als höheres Ziel wird durch diese Mechanismen bestens unterstützt.

In Kombination mit dem MVVM Architektur Pattern, dass auf WPF Features wie Databinding oder Commands zurück greift, wurde eine Architektur geschaffen, die Präsentation, Daten und Logik sauber trennt aber trotzdem so zusammen koppelt, dass Änderungen, seitens der Präsentationsebene oder des Datenmodells, in beide Richtungen reflektiert werden.

Abgerundet wird die Architektur dieser Zusammengesetzen Anwendung durch Verwendung des Prism Frameworks, welcher Module (Ansichten und Dienste) in deren Abhängigkeiten entkoppelt und die Applikation für zukünftige Erweiterungen / Modifikationen rüstet.

Die in dieser Arbeit erstellte Designumgebung stellt nun in der CLM Plattform die Basis für weitere Designumgebungen dar, welche auch andere Aufgaben (z.B. Modellbildung, Logik Editoren) bewältigen sollen.

Abbildung 55: Die Designumgebung im Einsatz

Literatur

[1] Nahum D. Gershon and Stephen G. Eick, editors. *Foreword. IEEE Symposium On Information Visualization 1995, InfoVis 1995, 30-31 October 1995, Atlanta, Georgia, USA*. IEEE Computer Society, 1995.

[2] Proview Team. Proview - open source process control. http://www.proview.se, 2010.

[3] Gordon Cameron. Modular visualization environments: past, present, and future. *SIGGRAPH Comput. Graph.*, 29(2):3–4, 1995.

[4] OpenDX Team. Proview - open visualization data explorer. http://www.opendx.org, 2010.

[5] Judith Bishop. Multi-platform user interface construction: a challenge for software engineering-in-the-small. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 751–760, New York, NY, USA, 2006. ACM.

[6] Windows Presentation Foundation. Wpf. http://windowsclient.net, 2010.

[7] Microsoft Developer Network. Msdn. http://msdn.microsoft.com, 2010.

[8] M. Potel. Mvp: Model-view-presenter: The taligent programming model for c++ and java. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf, 1999.

[9] Josh Smith. Wpf-anwendungen mit dem model-view-viewmodel-entwurfsmuster. *MSDN Magazin Februar 2009*, (2), 2009.

[10] Dietmar Winkler Stefan Biffl Erik Gostischa-Franta und Thomas Östreicher Alexander Schatten, Markus Demolsky. *Best Practice Software-Engineering*. Spektrum Akademischer Verlag, Heidelberg, 2010.

[11] Andreas Müller, Peter Forbrig, and Clemens H. Cap. Model-based user interface design using markup concepts. In *DSV-IS '01: Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification-Revised Papers*, pages 16–27, London, UK, 2001. Springer-Verlag.

[12] Mozilla Developer Center. Xul. https://developer.mozilla.org/en/XUL, 2010.

[13] *Essential XUL programming*. John Wiley and Sons Ltd., Chichester, UK, 2001.

[14] Mozilla Developer Center. Xulrunner. https://developer.mozilla.org/en/XULRunner, 2010.

[15] Kenneth Feldt. *Programming Firefox: Building Rich Internet Applications with Xul*. O'Reilly Media, Inc., 2007.

[16] Mozilla Developer Center. Xul tutorial. http://mdn.beonex.com/en/XUL_Tutorial/, 2010.

[17] John C. Bland II Tariq Ahmed, Dan Orlando and Joel Hooks. *Flex 4 in Action*. Manning Publications, Greenwich, UK, 2010.

[18] Jeanette Stallons. The architecture of flex and java applications. http://www.adobe.com/devnet/flex/articles/flex_java_architecture.html, 2010.

[19] Victor Rasputins Yakov Fain and Anatole Tartakovsky. *Enterprise Development with Flex*. O'Reilly Media, Inc., 2010.

[20] Adobe Systems. Tour de flex. http://www.adobe.com/devnet/flex/tourdeflex/web/, 2010.

[21] Adobe Systems. Flex sdk. http://opensource.adobe.com/wiki/display/flexsdk/Flex+SDK, 2010.

[22] Peter Armstrong. *Hello! Flex 4*. Manning Publications, Greenwich, UK, 2009.

[23] Adobe Systems. Adobe fxg. http://opensource.adobe.com/wiki/display/flexsdk/FXG+1.0+Specification, 2010.

[24] Microsoft Corporation. Xaml overview. http://msdn.microsoft.com/en-us/library/ms752059.aspx, 2010.

[25] Thomas Claudius Huber. *Windows Presentation Foundation - Das umfassende Handbuch*. Galileo Press, Bonn, 2008.

[26] Microsoft Corporation. Silverlight. http://www.microsoft.com/silverlight/, 2010.

[27] Pavan Podila und Kevin Hoffman. *WPF Control Development (Unleashed)*. Sams, Indianapolis, IN, USA, 2009.

[28] Adam Nathan. *Windows Presentation Foundation Unleashed (WPF) (Unleashed)*. Sams, Indianapolis, IN, USA, 2006.

[29] Matthew MacDonald. *Pro WPF in C# 2010: Windows Presentation Foundation with .NET 4*. APress, 2010.

[30] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[31] Trygve M. H. Reenskaug. Mvc xerox parc 1978-79. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, 1979.

[32] Daniel Liebhart, Guido Schmutz, Marcel Lattmann, Markus Heinisch, Michael Könings, Mischa Kölliker, Perry Palkul, and Peter Welkenbach. *Architecture Blueprints: Leitfaden zur Konstruktion von Softwaresystemen mit Java Spring, .Net, ADF, Forms und SOA*. Hanser, München, 2. edition, 2007.

[33] Glenn Block. Muster für das erstellen zusammengesetzter anwendungen mit wpf. *MSDN Magazin Februar 2009*, (9), 2008.

[34] Microsoft Corporation. The unity application block. http://msdn.microsoft.com/en-us/library/ff648512.aspx, 2010.

[35] Microsoft Corporation. Expression blend. http://www.microsoft.com/expression/products/blend_overview.aspx, 2010.