

Fachrichtung Informatik

Schuljahr 2016/2017

DIPLOMARBEIT

Gesamtprojekt

Package-Installer Creator

Ausgeführt von:

Florian Redinger, 5ahif-15

Florian Peter, 5ahif-14

Andreas Wageneder, 5ahif-20

Betreuer/Betreuerin:

Robert Grüneis

Grieskirchen, am 30.03.17

Abgabevermerk:

Datum: 30.03.17

Betreuer/Betreuerin:

Erklärung gemäß Prüfungsordnung

„Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.“

Grieskirchen, 04.04.2017

Verfasser/Verfasserinnen:

Florian Redinger

Florian Peter

Andreas Wageneder

Danksagung


An dieser Stelle möchten wir uns allen bedanken, die uns bei der Durchführung des Projektes und bei der Anfertigung dieser schriftlichen Arbeit unterstützten.

Ein Dank gebührt unserem Auftraggeber, Herrn Dipl.-Ing. Alexander Mann, ohne dessen Hilfe dieses Projekt nicht möglich gewesen wäre. Er unterstützte uns in fachlicher Hinsicht und informierte uns stets über notwendige Informationen zu den verschiedenen Komponenten der TGW.

Ein Dank gilt auch unserem Diplomarbeitsbetreuer, Herrn Dipl.-Ing. Robert Grüneis, welcher uns stets über den Ablauf der Diplomarbeit und über abzugebende Dokumente informierte. Er war uns auch immer eine große Unterstützung, wenn wir fachliche Probleme hatten.

Ferner möchten wir uns bei Frau Dipl.-Päd. Elisabeth Redinger und Frau Erna Wageneder, BEd bedanken, die sich uns als gewissenhafte Lektoren zur Verfügung gestellt haben.

Wir wollen uns auch bei Herrn Mag. Dr. Dietrich Birngruber, einem weiteren Angestellten der TGW, bedanken. Als Vertretung von Dipl.-Ing. Mann stand er uns häufig bei fachlichen Problemen zu Hilfe.

	HTBLA Grieskirchen
	Fachrichtung: Informatik


DIPLOMARBEIT DOKUMENTATION

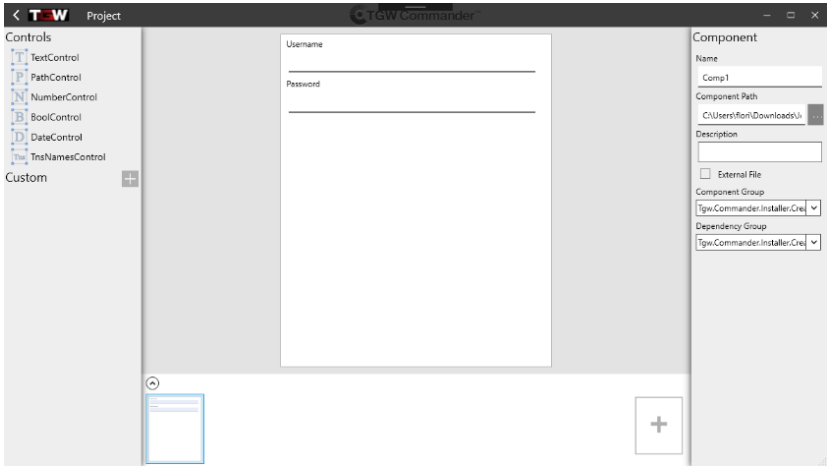
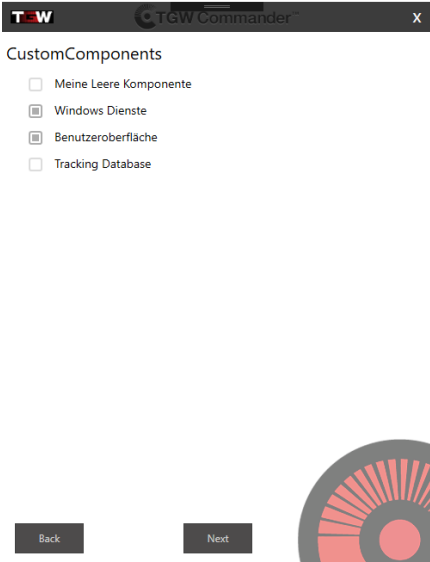
Namen der Verfasser/innen	Florian Redinger Florian Peter Andreas Wageneder
Jahrgang Schuljahr	5. Jahrgang 2016/2017
Thema der Diplomarbeit	Package-Installer Creator
Kooperationspartner	TGW Logistics Group

Aufgabenstellung	Erstellen einer Applikation (Package-Installer Creator) zur grafischen Erstellung einer XML-Datei, welches mit einer weiteren Applikation (Package-Installer Runtime) ausgelesen wird und die Installation verschiedener Komponenten ermöglicht.
-------------------------	--


Realisierung	<p>Sowohl der Package-Installer Creator als auch die Runtime sollen mit dem .NET Framework realisiert werden.</p> <p>Der Creator soll mithilfe verschiedener UserControls das Erstellen von Komponenten ermöglichen und diese zu einer XML-Datei serialisieren, welche in einer ebenfalls erstellten Ordnerstruktur liegt. Weiters soll es möglich sein, die Komponenten für verschiedene Sprachen zu erstellen.</p> <p>Die Runtime soll diese erstellte XML-Datei auslesen und daraus die Komponenten erstellen, grafisch anzeigen und ebenfalls eine Konsoleninstallation ermöglichen. Die zu installierenden Binaries (MSI, Database, etc.) werden im Creator ausgewählt und die Pfade dazu in der XML-Datei gespeichert. Diese Binaries werden von der Runtime installiert.</p>
---------------------	---

Ergebnisse	Sämtliche Anforderungen des Partners wurden erfüllt.
-------------------	--

	HTBLA Grieskirchen	
	Fachrichtung:	Informatik

Typische Grafik, Foto etc. (mit Erläuterung)	 <p>Hauptfenster des Package-Installer Creators zur Erstellung von Komponenten.</p>	
	 <p>Beispielhaftes Fenster der Package-Installer Runtime.</p>	

Teilnahme an Wettbewerben, Auszeichnungen	-	
Möglichkeiten der Einsichtnahme in die Arbeit	Die Diplomarbeit kann öffentlich eingesehen werden	
Approbation (Datum/Unterschrift)	Prüfer/Prüferin	Direktor/Direktorin

	HTBLA Grieskirchen
	Department: Informatik

DIPLOMA THESIS


Documentation

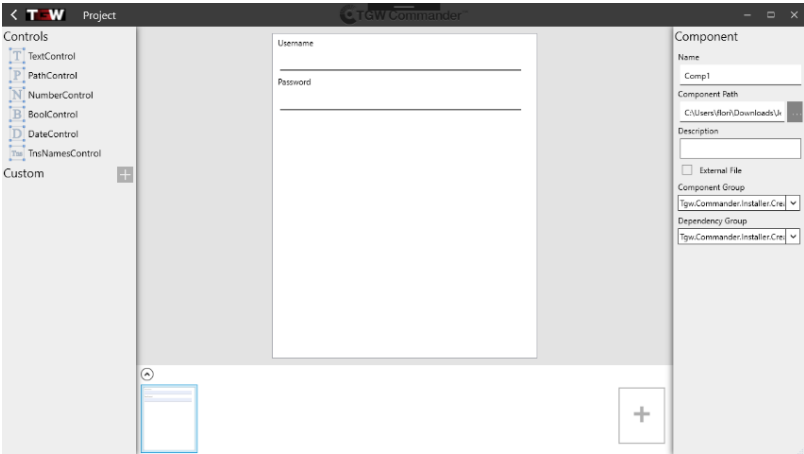
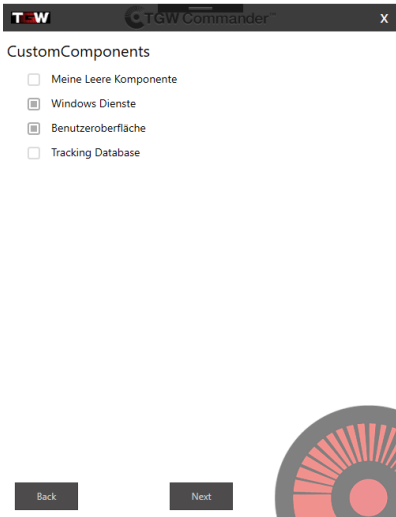
Author(s)	Florian Redinger Florian Peter Andreas Wageneder
Form Academic year	5 th Form 2016/2017
Topic	Package-Installer Creator
Co-operation partners	TGW Logistics Group

Assignment of tasks	An application (Package Installer Creator) capable of graphically creating an XML file which will be read by another application (Package Installer Runtime) and allows for the installation of various components.
----------------------------	---

Realisation	<p>Both the Package Installer Creator and the Runtime should be realized with the .NET framework.</p> <p>The Creator should be able to create components consisting of different UserControl controls and serializing them into an XML file, which is placed in the folder structure as well. Further-more, it should be possible to create the components for different languages.</p> <p>The Runtime should read this created XML file, generate XML from it and display them on the screen. It should support console installation. The binaries to install (MSI, database, etc.) are selected in the Creator, which saves the paths in the XML file. The binaries are installed by the runtime.</p>
--------------------	---

Results	Any request from the partner has been fulfilled.
----------------	--

	HTBLA Grieskirchen	
	Department:	Informatik

Illustrative graph, photo (incl. explanation)	 <p>Main window of the Package-Installer Creator to create components.</p>	
	 <p>Exemplary window of the Package-Installer Runtime.</p>	

Participation in competitions Awards	-
---	---

Accessibility of diploma thesis	The diploma thesis can be viewed publicly
--	---

Approval (date/signature)	Examiner	Head of College/Department
--------------------------------------	----------	----------------------------

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ist-Zustand	1
1.2	Aufgabenstellung	3
1.3	Fachliches und wirtschaftliches Umfeld	3
1.3.1	Fachlich	3
1.3.2	Wirtschaftlichkeit	5
2	Individuelle Zielsetzung und Aufgabenstellung.....	6
2.1	Individuelle Zielsetzung	6
2.1.1	Florian Redinger	6
2.1.2	Florian Peter	6
2.1.3	Andreas Wageneder	6
2.2	Terminplan (Initialer Plan)	7
2.2.1	Projektmanagement	7
2.2.2	Package-Installer Runtime (Features)	8
2.2.3	Package-Installer Creator (Features).....	9
2.2.4	Diplomarbeit	10
2.3	Backlogs	11
2.3.1	Package-Installer Runtime (ohne Tasks)	11
2.3.2	Package-Installer Creator (ohne Tasks)	12
3	Grundlagen und Methoden	14
3.1	Entwurfsmuster	14
3.1.1	Model View ViewModel (MVVM)	14
3.1.2	Singleton	17
3.1.3	Factory-Method	20
3.2	XML Schema Definition (XSD)	22
3.2.1	Das Schema bei unserer Diplomarbeit	25
3.3	Package-Installer Creator	30
3.3.1	Shell und Menüs.....	30
3.3.2	Window Behavior	37
3.3.3	Arten von Controls	38
3.3.4	Neues Installer Projekt	41
3.3.5	Existierendes Projekt bearbeiten	43
3.3.6	Existierendes Projekt löschen	44
3.3.7	Komponente erstellen	44
3.3.8	Benutzerdefiniertes Control.....	46
3.3.9	Controls hinzufügen & bearbeiten	48
3.3.10	„Builden“ des Installers	50

3.3.11	Komponenten-Typen	53
3.3.12	Lokalisierung	54
3.3.13	Lokalisierung verwalten	56
3.4	Package-Installer Runtime	58
3.4.1	Startup	58
3.4.2	Komponenten-Gruppen	60
3.4.3	Benutzerdefinierte Auswahl	62
3.4.4	Komponenten	63
3.4.5	Installation	65
3.4.6	Page-Switcher	66
3.4.7	Lokalisierung	67
3.4.8	Grundlegende Funktionen des Programms	70
4	Ergebnisse	71
4.1	Vorteile für die TGW	71
4.1.1	Aktuelle Umstände.....	71
4.1.2	Vorteile und Verbesserungen	71
4.2	Umfang	73
4.2.1	Arbeitsstunden.....	73
4.2.2	Source-Code.....	73
4.3	Lessons learned.....	74
4.3.1	Projekt.....	74
4.3.2	Erlertes Wissen	75
5	Anhang	82

1 Einleitung

1.1 Ist-Zustand

Die TGW Logistics Group mit Sitz in Wels ist ein österreichisches Unternehmen im Bereich Lagerlogistik. An diversen Standorten in der ganzen Welt werden komplexe Lagersysteme hergestellt, die von Großunternehmen stark nachgefragt werden. Ein besonders wichtiger Aspekt ist die Automatisierung, die in der heutigen Zeit eine immer wichtigere Rolle spielt.

Die Geschäftsbereiche der TGW sind oft umfassend und komplex, wobei in vielen von diesen Bereichen stets Lösungen zur Vereinfachung gesucht werden. Ein Aspekt dabei sind die unzähligen Installer-Programme, die für verschiedenste interne und auch externe (Kunden, etc.) Abläufe wichtige Daten bereitstellen. Bereits die Verwaltung aller Installations-Pakete trägt einiges zu erbrachten Arbeitsstunden bei, doch der eigentliche Codierungs-Aufwand, das heißt, den Installer für jeden Kunden- oder Abteilungswunsch hart codiert anzupassen, kostet den Programmierern den größten Personenstundenaufwand. Dabei gehen oftmals Hunderte von Stunden verloren, die in alternativen Projekten besseren Nutzen finden würden. Viele der Stunden werden verursacht, nicht bloß durch reine Re-Programmierung, sondern aufgrund langer, oft schwieriger Fehlerbehebungen, die in den schlimmsten Fällen schon Tage gedauert haben.

Erstmaligen Fortschritt gab es in diesem Bereich mit dem 2016 entwickelten statischen UI-Installer, dessen Funktionalitätsumfang auf dem damals aktuellen Konsolen-Programm basierte. Dieses sehr schwer zu bedienende Programm bedurfte langer, umfangreicher Einschulungen, welche nicht nur Dritten enorm zu Lasten fielen, sondern auch der TGW selbst, da diese für jede erneute Einschulung mindestens einen Mitarbeiter entsenden musste. Dabei stiegen weiterhin die Kosten, ohne tatsächlichen merkbaren wirtschaftlichen Nutzen. Hinzu kam die ständige Unzufriedenheit einiger Kunden, die sich über die fehlerhafte Nutzung des Installers durch Angestellte oder Lieferungsverzug der TGW ärgerten. Die UI-fähige Software konnte zwar noch als Konsolen-Applikation ausgeführt werden, um bereits eingeschulten Mitarbeitern den gewohnten Bedienungsweg zu ermöglichen, doch wurde nun dank selbsterklärender und einfach überschaubarer grafischer Masken jedem die Möglichkeit geboten, ohne Einschulung und ohne große Probleme, gewünschte Pakete ohne Verzug installieren zu können. Dies wurde bereits als großer Fortschritt in die richtige Richtung betrachtet, doch fehlte noch immer ein essentieller Aspekt.

Um endgültig die fundamentalsten Probleme der Software zu bereinigen, wurde im Zuge des Diplomarbeits-Projektes gemeinsam mit der TGW Logistics Group endlich an einer Problemlösung gearbeitet, die dem dringenden Bedarf nach mehr Dynamik gerecht wird. Nun können Mitarbeiter jeder Erfahrungsstufe und Dritte in wenigen Minuten funktionstüchtige, vollkommen individuelle und einzigartige Installer erstellen, ohne dabei auch nur ein Auge auf den Source-Code oder sonstige Klassen zu werfen. Der Creator wird dabei anhand einer grafischen Oberfläche bedient und beinhaltet alle benötigten Funktionen, um dem Benutzer die gewünschte Dynamik bereitzustellen. Neben den Installationspaketen, deren Bestandteile eigens definiert werden können, gibt es zusätzlich die Möglichkeit, aus Standard-Bedienelementen (Controls) individuelle benutzerdefinierte Controls zu erstellen. Der gesamte Funktionsumfang gebräuchlicher Installer wird mit dem Creator damit abgedeckt. Somit fungiert die Software als IDE.

Dabei basiert jeder neu erstellte Installer auf einer XML-Definitionsdatei, die alle benötigten Informationen, das heißt Name, Installations-Pakete, Komponenten und deren Abhängigkeiten speichert und, dank eines durchdachten Schemas, leserlich darstellt bzw. problemlos von der inneren Programmlogik des Installers, auch Runtime genannt, verarbeitet und darstellt. Dank der grafischen Benutzeroberfläche des Creators muss die Definitionsdatei nicht selbst geschrieben werden, was neben Rechtschreibfehlern auch Syntax- oder Validierungsfehler vermeidet bzw. ausschließt.

Dank der Realisierung dieses Projektes entsteht für die TGW jeden Monat ein respektable Gewinn, der nun in weitere Bereiche des Unternehmens investiert werden kann, um dieses noch weiter auszubauen. Auch die zuständigen Mitarbeiter sparen sich durch den Package-Installer Creator sehr viel Zeit, welche sie in andere Projekte investieren können.

1.2 Aufgabenstellung

Die Aufgabenstellung der TGW besteht darin, zwei Applikationen zu entwickeln.

Zum einen gibt es eine Runtime, die auf dem statischen UI-Installer basiert, und zum anderen die grafische IDE, den Creator, um die dynamische Installer-Erstellung überhaupt zu ermöglichen. Dabei wurde explizit der Wunsch ausgesprochen, den Creator in puncto Design nahe an den bekannten Microsoft Office Produkten zu halten, da diese bereits stark im Unternehmen verbreitet sind. Neben dem vorgegebenen Design-Rahmen muss die IDE imstande sein, eine vollständige und lückenlose Ordnerstruktur für das Installer-Projekt anlegen zu können, um dort später alle essentiellen Dateien zu speichern. Weiters soll die XML-Definitionsdatei ganz nach den gewählten Einstellungen und Eingaben des Benutzers dynamisch nach Schema mit einem XML-Serializer generiert werden. Diese Datei muss folgende Daten enthalten:

- Grundlegende Informationen des Installers
- Komponenten + Controls
- Komponentengruppen (automatische Auswahl mehrerer Komponenten)
- Abhängigkeitsgruppen (darin sind Abhängigkeiten zwischen Komponenten definiert)

... und deren Informationen:

- Name
- Pfad der Binärdatei
- komponentenspezifische obligatorische Daten
- Optionale Daten (kundenspezifisch)

Diese sind wichtig für die Verarbeitung der Komponenten im Installations-Prozess der Runtime.

Die zweite Applikation, die Runtime, soll es ermöglichen, die Daten der generierten XML-Definitionsdatei auszulesen und zu verarbeiten, um somit dynamisch einen lauffähigen Installer zu erstellen.

Des Weiteren soll der Creator auch über eine Bearbeitungs-Funktion verfügen, wonach XML-Dateien, die ohne Validierungsfehler gelesen und geöffnet werden, auch grafisch mit der IDE bearbeitet werden können.

1.3 Fachliches und wirtschaftliches Umfeld

1.3.1 Fachlich

Um gemeinsam mit der TGW an dieser Software zu arbeiten, mussten wir uns an deren Arbeitsweisen und Umstände anpassen. Deshalb war es notwendig, uns in vielen Bereichen Wissen anzueignen, um das richtige Verständnis für die Realisierung dieses Projektes zu haben.

1.3.1.1 Agiles Projektmanagement – Scrum

Scrum ist ein Vorgehensmodell der Projektplanung und des Projektmanagements, welches vor allem in der agilen Software-Entwicklung Nutzen findet. Herausstechende Merkmale dieses Modells sind die transparenten und anpassungsfähigen Arbeitsschritte. Dabei wird regelmäßig in kurzen Sitzungen der aktuelle Stand besprochen sowie Probleme diskutiert. Auch werden in periodischen Abständen Funktionalitäten der Software besprochen, geplant und entwickelt. Somit liegt ein geringerer Planungsaufwand am Beginn des Projekts vor bzw. muss eine Software nicht schon vollkommen durchdacht worden sein.

Da die TGW in erster Linie nur von agilen Methoden Gebrauch nimmt, wurde dies auch in das Projektmanagement der Diplomarbeit übernommen. Dabei wurden bei jedem Meeting die nächsten Schritte sowie Probleme besprochen. Dennoch wurden ein Zeitplan und Pflichtenheft erstellt, da zum einen nur Teilansätze von Scrum übernommen und zum anderen diese auch von schulischer Seite gefordert wurden.

1.3.1.2 WPF-Framework

Die TGW nutzt bei der Software-Entwicklung für Windows-Systeme vorrangig das WPF-Framework. Dieses bietet umfangreiche Bibliotheken, die dem Designer viele Design-Möglichkeiten offenlegen. Da an der Schule nur Windows Forms, anstatt WPF, gelehrt wurde, mussten wir im Zuge der Diplomarbeit dieses Framework erlernen.

1.3.1.3 Installations-Framework

Bereits beim statischen Konsolen-Installer benutzt die TGW ein eigens für diesen Gebrauch entwickeltes Framework zur Installation ihrer Komponenten. Dieses konnte für die Diplomarbeit übernommen werden, jedoch wurde es für neue Komponenten-Typen erweiterbar und ressourcen-effizienter programmiert. Um diese Änderungen zu ermöglichen, mussten wir uns zuerst mit dem Source-Code des Frameworks befassen und dieses studieren.

1.3.1.4 TGW Komponenten

Noch während die TGW den Konsolen-Installer nutzte, gab es bereits vordefinierte Installations-Komponenten. Diese mussten in das Diplomarbeits-Projekt übernommen werden, wobei sich jedes Teammitglied mit deren Eigenschaften und Verwendung im Code auseinandersetzen musste, um diese in der inneren Programmlogik dann auch richtig verwenden und verarbeiten zu können.

1.3.1.5 Microsoft

In der Software-Entwicklung setzt die TGW fast vollständig nur auf Microsoft-Produkte. Das gilt sowohl für Frameworks als auch für Programmierumgebungen. Gearbeitet wurde deshalb mit dem Visual Studio, unter Verwendung von C# und WPF. Geplant wird im Visual Studio Online und den Source-Code verwaltet man mit dem Team Foundation Server.

1.3.2 Wirtschaftlichkeit

Die Entwicklung dieser Software bringt viele wirtschaftliche Vorteile für die TGW. Programmier- und Wartungskosten werden gesenkt, die Zufriedenheit der Kunden sowie interner Abteilungen wird gewonnen.

1.3.2.1 TGW Kunden

Die Entwicklung des Creators und der dynamischen Runtime₂ ermöglicht der TGW ein bei weitem umfangreicheres Angebot für Kunden. Diese können nun ganz kundenspezifisch in kürzester Zeit, ohne großen Aufwand oder Einschulung, erstellt und dem Kunden übergeben werden. Dadurch entstand eine große Verantwortung für uns, eine hohe Qualität zu erzielen.

1.3.2.2 Bereitstellung von Daten

Nicht nur das äußere Erscheinungsbild muss einer hohen Qualität entsprechen, auch die innere Programm-Logik soll alle Daten richtig verarbeiten. Diese Daten werden auf unzähligen Systemen verteilt, eine fehlerhafte Verarbeitung der Komponenten darf daher auf keinen Fall auftreten.

2 Individuelle Zielsetzung und Aufgabenstellung

2.1 Individuelle Zielsetzung

Die Teammitglieder haben folgende Rolle(n) übernommen:

Tabelle 1: Rolleneinteilung der einzelnen Teammitglieder

Mitglied	Rolle(n)
Florian Redinger	Projektmanager, Designer, Programmierer
Florian Peter	DB-Architekt, Programmierer
Andreas Wageneder	Programmierer

2.1.1 Florian Redinger

Florian Redingers Aufgaben umfassten das Projektmanagement, die Erstellung des Designs der beiden Applikationen sowie deren technische Umsetzung. Weiters entwickelte Florian einen maßgeblichen Teil der Programmlogik, sowohl vom Package-Installer Creator als auch von der Package-Installer Runtime.

2.1.2 Florian Peter

Florian Peter war für die Umsetzung der Lokalisierung sowie für den Aufbau der Ordner-Struktur der dynamischen Package-Installer Runtime und für die Möglichkeit, den Installer per Konsole zu bedienen, zuständig. Des Weiteren entwickelte er den Installations-Algorithmus, der gemeinsam mit dem TGW Installations-Framework die Pakete installiert.

2.1.3 Andreas Wageneder

Andreas Wagenders Aufgaben bestanden darin, das XML-Schema für den Package-Installer Creator und für die Package-Installer Runtime zu erstellen. Ferner kümmerte er sich um einen maßgeblichen Teil der Programmlogik beider Applikationen.

2.2 Terminplan (Initialer Plan)

2.2.1 Projektmanagement

Tabelle 2: Planung der einzelnen Projektmanagement-Phasen

Nr.	Arbeitspaket	Von	Bis
1.1	Modellplanung	11. Juli 2016	12. Juli 2016
1.1.1	Prototyping	11. Juli 2016	12. Juli 2016
1.1.2	Anforderungsanalyse	12. Juli 2016	12. Juli 2016
1.2	Vorstudie	18. Juli 2016	25. Juli 2016
1.2.1	XML-Schema	18. Juli 2016	25. Juli 2016
1.2.2	WPF	15. Juli 2016	22. Juli 2016
1.2.3	Regex	18. Juli 2016	20. Juli 2016
1.2.4	Balsamiq Mockups	15. Juli 2016	18. Juli 2016
1.2.5	XML Verarbeitung	19. Juli 2016	22. Juli 2016
1.2.6	Pflichtenheft Aufbau	19. Juli 2016	22. Juli 2016
1.2.7	OS Benutzerdaten auslesen	19. Juli 2016	19. Juli 2016
1.2.8	Team Foundation Server	18. Juli 2016	20. Juli 2016
1.3	Designplanung	15. Juli 2016	19. Juli 2016
1.3.1	Package-Installer Runtime	15. Juli 2016	19. Juli 2016
1.3.2	Package-Installer Creator	15. Juli 2016	19. Juli 2016
1.4	Strukturplanung	18. Juli 2016	22. Juli 2016
1.4.1	Backlog-Erstellung	18. Juli 2016	20. Juli 2016
1.4.2	Pflichtenheft	19. Juli 2016	22. Juli 2016
1.5	Zeitplanung	22. Juli 2016	22. Juli 2016
1.5.1	Zeitplan erstellen	22. Juli 2016	22. Juli 2016

2.2.2 Package-Installer Runtime (Features)

Tabelle 3: Zeitplan zur Erstellung der einzelnen Features der Package-Installer Runtime

Nr.	Arbeitspaket	Von	Bis
2.1	Startup	20. Juli 2016	22. Juli 2016
2.2	Komponenten-Gruppen	22. Juli 2016	26. Juli 2016
2.3	Benutzerdefinierte Auswahl	25. Juli 2016	26. Juli 2016
2.4	Komponenten	26. Juli 2016	26. Juli 2016
2.5	Summary	27. Juli 2016	28. Juli 2016
2.6	Installation	25. Juli 2016	27. Juli 2016
2.7	Page-Switcher	22. Juli 2016	22. Juli 2016
2.8	Lokalisierung	28. Juli 2016	28. Juli 2016
2.9	Grundlegende Funktionen	20. Juli 2016	22. Juli 2016

2.2.3 Package-Installer Creator (Features)

Tabelle 4: Zeitplan zur Erstellung der einzelnen Features des Package-Installer Creators

Nr.	Arbeitspaket	Von	Bis
3.1	Neues Installer-Projekt erstellen	29. Juli 2016	29. Juli 2016
3.2	Grundlegende Komponenten-Funktionen	29. Juli 2016	01. August 2016
3.3	Erstellung einer Komponente	01. August 2016	02. August 2016
3.4	Komponente bearbeiten	02. August 2016	05. August 2016
3.5	Komponenten-Vorlagen	29. Juli 2016	02. August 2016
3.6	Grundlegende Funktionen v. Controls	02. August 2016	04. August 2016
3.7	Erstellung eines Controls	03. August 2016	05. August 2016
3.8	Arten von Controls	04. August 2016	10. August 2016
3.9	Control-Vorlagen	05. August 2016	08. August 2016
3.10	Installer speichern	08. August 2016	12. August 2016
3.11	Erstellung d. Installers	10. August 2016	12. August 2016
3.12	Existierendes Projekt bearbeiten	05. August 2016	08. August 2016
3.13	Existierendes Projekt löschen	08. August 2016	12. August 2016
3.14	Grundlegende Software-Funktionen	29. August 2016	03. August 2016
3.15	Package-Installer Model	12. August 2015	30. September 2016
3.16	Komponenten-Typen	01. November 2016	01. Dezember 2016
3.17	Lokalisierung	01. November 2016	01. Dezember 2016

2.2.4 Diplomarbeit

Tabelle 5: Zeitplan zur Erstellung der schriftlichen Arbeit

Nr.	Arbeitspaket	Von	Bis
4.1	Schreiben der Diplomarbeit	01. November 2016	31. Jänner 2017

Die initiale Planung der Diplomarbeit sah ein frühzeitiges Ende der Software-Entwicklung mit Mitte Dezember 2016 vor. Aufgrund mehrerer wochenlanger Krankheitsfälle verschiedenster Teammitglieder und der Unterschätzung des Entwicklungsumfangs der Software, mussten einige Meilensteine (+ Arbeitspakete) zeitlich nach hinten verschoben werden (15 000 – 20 000 Zeilen Code, mehr als 600h Arbeitszeit).

2.3 Backlogs

Um den Umfang der Arbeiten sowie die benötigten Funktionalitäten besser planen und einschätzen zu können, haben wir Backlogs, mit Epics und User Stories, angelegt, in denen nicht nur grundlegende Arbeitspakete der Diplomarbeit dargestellt werden, sondern auch deren fundamentale Tasks.

2.3.1 Package-Installer Runtime (ohne Tasks)

Order	Work Item Type	Title	State	Effo...	Bus...	Value Area	Tags
2	Epic	Package Installer Runtime	New			Business	
	Feature	Startup	Closed			Business	
	User Story	Preload screens in background	Closed			Architectural	
	Feature	Component group	Closed			Business	
	User Story	Component group installation	Closed			Business	
	User Story	Custom installation	Closed			Business	
	Feature	Custom installation	Closed			Business	
	User Story	Select components	Closed			Business	
	User Story	Dependency information	Closed			Business	
	Feature	Component	Closed			Business	
	User Story	Generate controls	Closed			Architectural	
	User Story	Validity check	Closed			Architectural	
	User Story	Error information	Closed			Business	
	Feature	Summary	Closed			Business	
	User Story	Show components & controls	Closed			Business	
	User Story	Property binding	Closed			Business	
	Feature	Installation	Active			Business	
	User Story	Progress bar	Closed			Business	
	User Story	Installation components list	Closed			Business	
	User Story	Failure information	Closed			Business	
	Feature	Page Switching	Closed			Business	
	User Story	Back	Closed			Business	
	User Story	Next	Closed			Business	
	Feature	Localization	Closed			Business	
	User Story	Application language based on OS	Closed			Business	
	User Story	Add Database for Localization	Closed			Business	
	Feature	General application functions	Closed			Architectural	
	User Story	Shell	Closed			Architectural	
	User Story	Exit application	Closed			Business	
	User Story	Read and process Xml	Closed			Architectural	
	Feature	Component types	Active			Architectural	
	User Story	Trade-Aware MSI	Active			Architectural	
	User Story	TGW Deployment	New			Architectural	
	User Story	MS-SQL Database	New			Architectural	
	User Story	MSI	Closed			Architectural	
	User Story	Oracle Database	Closed			Architectural	

2.3.2 Package-Installer Creator (ohne Tasks)

Order	Work Item Type	Title	State	Effo...	Bus...	Value Area	Tags
3	Epic	Package Installer Creator	New			Architectural	
	Feature	Create new installer project	Closed			Business	
	User Story	Folder structure creation	Closed			Architectural	
	User Story	Creation of necessary files	Closed			Architectural	
	User Story	Enter overview data	Closed			Business	
	Feature	Basic component function	Active			Business	
	User Story	Display components	Active			Business	
	User Story	Delete component	Closed			Business	
	Feature	Create component	Active			Business	
	User Story	Component information input	Active			Business	
	Feature	Edit component	New			Business	
	User Story	Component group creation	New			Business	
	User Story	Dependency group creation	Active			Business	
	User Story	Modify properties	Closed			Business	
	User Story	Adding control (to component)	Closed			Business	
	User Story	Delete control (from component)	Closed			Business	
	Feature	Component templates	New			Business	
	User Story	Loading templates	New			Business	
	User Story	Saving templates	New			Business	
	Feature	Basic control functions	Closed			Business	
	User Story	Display control	Closed			Business	
	User Story	Control preview	Closed			Business	
	User Story	Control creation	Closed			Business	
	Feature	Create control	Closed			Business	
	User Story	Input basic information	Closed			Business	
	Feature	Edit control	Active			Business	
	User Story	Modify properties	Active			Business	
	User Story	Add controls	Closed			Business	
	User Story	Delete controls	Closed			Business	
	Feature	Control types	Closed			Business	
	User Story	Text	Closed			Business	
	User Story	Path	Closed			Business	
	User Story	Number	Closed			Business	
	User Story	Bool	Closed			Business	
	User Story	Date	Closed			Business	
	User Story	TNS-Names	Closed			Business	
	User Story	Custom	Closed			Business	
	Feature	Control templates	New			Business	
	User Story	Load templates	New			Business	
	User Story	Save templates	New			Business	
	Feature	Save installer	New			Business	
	User Story	Save installer	New			Business	

Order	Work Item Type	Title	State	Effo...	Bus...	Value Area	Tags
	Feature	▼ Build installer	● Closed			Architectural	
	User Story	■ Load Localization file	● New			Business	
	User Story	■ Copy Binaries to folder	● Closed			Business	
	User Story	■ Serialize Xml-File	● Closed			Business	
	User Story	■ Copy exe file for Runtime	● Closed			Business	
	Feature	▼ Edit existing project	● Closed			Business	
	User Story	> ■ Open window	● Closed			Business	
	Feature	▼ Delete existing project	● Closed			Business	
	User Story	> ■ Open window	● Closed			Business	
	Feature	▼ General application functions	● New			Business	
	User Story	> ■ Application menu	● Closed			Business	
	User Story	> ■ Exit application	● Closed			Business	
	User Story	> ■ Modified Check	● Closed			Business	
	User Story	> ■ Dialogs	● Closed			Business	
	User Story	> ■ Shell	● Closed			Architectural	
	User Story	> ■ Write to XML	● Closed			Architectural	
	User Story	> ■ Modify XML	● Closed			Architectural	
	Feature	▼ Package installer model	● New			Architectural	
	User Story	■ Project	● Closed			Architectural	
	User Story	■ Component	● Closed			Architectural	
	User Story	■ Component group	● Closed			Architectural	
	User Story	■ Component dependency group	● Closed			Architectural	
	Feature	▼ Localization	● New			Business	
	User Story	> ■ Enter languages	● New			Business	

Abb. 2: Backlogs des Package-Installer Creators inkl. Epics, Features und User Stories.

3 Grundlagen und Methoden

3.1 Entwurfsmuster

Um die Programmierung der einzelnen Bestandteile unserer Diplomarbeit benutzerfreundlicher, leichter und erweiterbarer zu machen, wurde eine Reihe von Entwurfsmustern verwendet. Dabei stachen drei besonders hervor, da sie einen sehr großen Einfluss auf unsere Diplomarbeit haben und sich durch das ganze Projekt ziehen.

3.1.1 Model View ViewModel (MVVM)

Das am häufigsten verwendete Entwurfsmuster in den beiden Projekten ist ohne Zweifel das Entwurfsmuster „Model View ViewModel“, welches bei jedem Fenster und bei einigen UserControl verwendet wird.

3.1.1.1 Grundlagen Model View ViewModel

„**Model View ViewModel (MVVM)** ist ein Entwurfsmuster und eine Variante des Model-View-Controller-Musters (MVC). Es dient zur Trennung von Darstellung und Logik der Benutzerschnittstelle (UI). Es zielt auf moderne UI-Plattformen wie Windows Presentation Foundation (WPF), JavaFX, Silverlight und HTML5 ab, da ein Datenbindungsmechanismus erforderlich ist. Gegenüber dem MVC-Muster kann die Testbarkeit verbessert und der Implementierungsaufwand reduziert werden, da keine separaten Controller-Instanzen erforderlich sind. MVVM erlaubt eine Rollentrennung von UI-Designern und Entwicklern, wodurch Anwendungsschichten von verschiedenen Arbeitsgruppen entwickelt werden können. Designer können einen Fokus auf User Experience setzen und Entwickler die UI- und Geschäftslogik schreiben.“ (https://de.wikipedia.org/wiki/Model_View_ViewModel) [12. März 2017]

3.1.1.2 Aufbau von MVVM

Der Aufbau von Model View ViewModel kann man folgendermaßen verstehen:

- Man hat ein Model:
 - Das ist eine sehr einfache Klasse, welche nur aus sog. Properties besteht.
 - Es kann mehrere Models geben, was in der Praxis fast immer der Fall ist.
 - Ein Model kann weitere Models beinhalten.
- Man hat eine View:
 - Hier wird das Design erstellt und jegliche Arbeit, welche nicht für die Programmlogik ausschlaggebend ist.
 - Es ist vorteilhaft, wenn bei der View der DataContext auf das ViewModel gesetzt wird.
- Man hat ein ViewModel:
 - Hier werden die Programmlogik, die Referenzen zum Model und auch eigene Properties erstellt. Alle Properties werden an das UI gebunden.
 - ViewModels können ebenfalls weitere ViewModels beinhalten oder auch von mehreren Views verwendet werden.
- Im Idealfall wissen weder das Model noch das ViewModel irgendetwas von der View, es soll also keine Referenzen auf die View geben.
- Events werden mit sog. Commands realisiert.
- Um die Properties zu aktualisieren, ist das Interface „INotifyPropertyChanged“ notwendig, damit die View von den Änderungen erfährt.

Durch dieses Vorgehen weiß das ViewModel nichts von der View und die View weiß nichts über die genaue Programmlogik. So ist es möglich, dass Designer und Programmierer getrennt arbeiten können.

Zum besseren Verständnis dieses Sachverhalts dient folgendes UML-Diagramm.

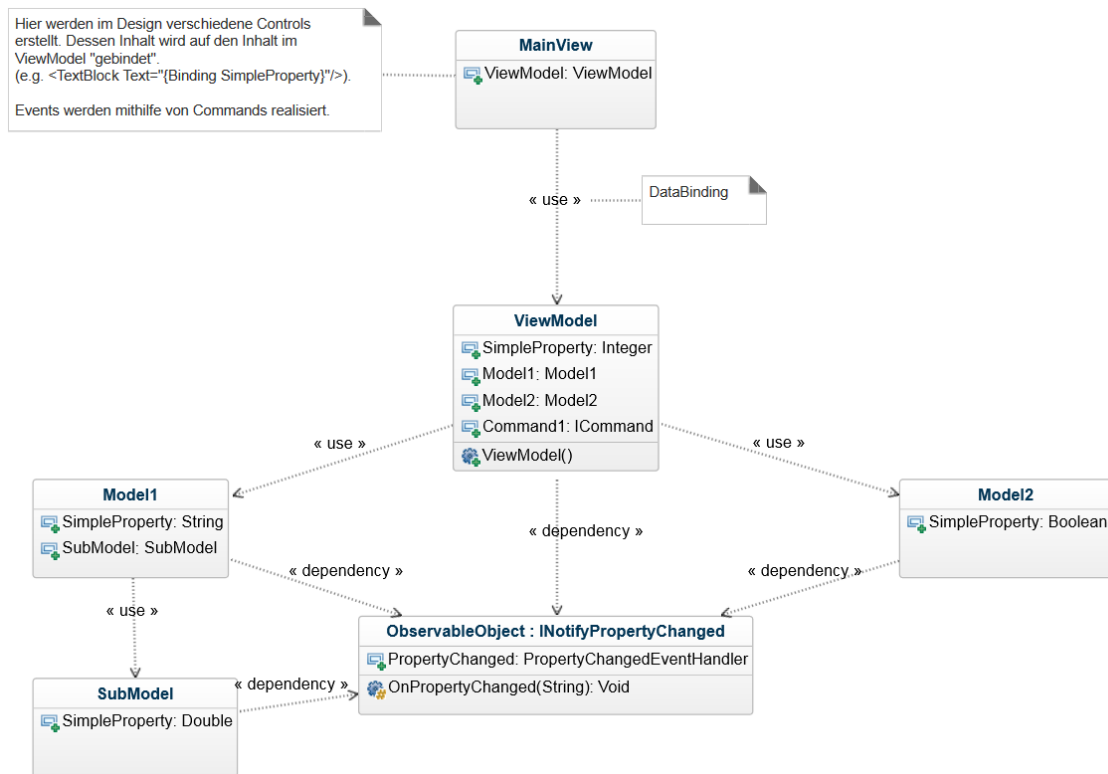


Abb. 3: Beispielschema eines nach MVVM aufgebauten Fensters.

Durch dieses Vorgehen ergeben sich viele Vor- und Nachteile.

Vorteile:

- Die Geschäftslogik ist unabhängig von der Darstellung.
- Die Testbarkeit wird erleichtert, da das ViewModel die UI-Logiken beinhaltet, welche unabhängig von der View sind, weshalb auch keine Coded UI Tests, sondern nur einfache Unit-Tests notwendig sind.
- Views können von reinen UI-Designern gestaltet werden, während Entwickler unabhängig davon die Models und ViewModels erstellen.
- Mehrere Views können dasselbe ViewModel nutzen, ohne etwas an der Programmierung zu ändern.

Nachteile:

- Ein Nachteil ist der erhöhte Rechenaufwand durch den enthaltenen Observer.
- Durch Databinding müssen keine Controller mehr implementiert werden. Um allerdings MVVM-Muster zu ermöglichen, ist Databinding zwingend erforderlich.

Das Designpattern wird einerseits verwendet, weil es der Standard bei WPF-Programmen ist, und die Erstellung der Projekte durch die große Anzahl von Events, Controls und Fenster andererseits nicht oder nur sehr schwer und kompliziert möglich wäre.

3.1.2 Singleton

Ein sehr wichtiges Designpattern in dieser Diplomarbeit ist ohne Zweifel das „Singleton“. Dieses Designpattern ermöglicht es uns, verschiedene beständige Daten in einer Config-Datei zu speichern.

3.1.2.1 Grundlagen Singleton

„Das **Singleton** (selten auch **Einzelstück** genannt) ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster und gehört zur Kategorie der Erzeugungsmuster (engl. *creational patterns*). Es stellt sicher, dass von einer Klasse genau ein Objekt existiert. Dieses Singleton ist darüber hinaus üblicherweise global verfügbar. Das Muster ist eines der von der sogenannten Viererbande (GoF) publizierten Muster.“ ([https://de.wikipedia.org/wiki/Singleton_\(Entwurfsmuster\)](https://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster))) [12. März 2017]

3.1.2.2 Aufbau vom Singleton

Man kann es sich also folgendermaßen vorstellen:

- Es existiert eine Klasse, welche statisch bzw. sealed ist.
- Es existiert ein privater Konstruktor, damit es nicht mehrere Instanzen dieser Klasse geben kann.
- Es existiert ein privates Feld namens „instance“.
- Es gibt ein public Property „Instance“ oder eine public Methode „GetInstance()“, welche es ermöglicht, eine Instanz von der Klasse zu erhalten.
- Wird die Methode „GetInstance“ aufgerufen, wird überprüft, ob das Feld „instance“ null ist.
- Ist das Feld null, wird eine neue Instanz zurückgegeben. Sollte das Feld nicht null sein, wird die bestehende Instanz zurückgegeben.

Dieser Vorgang ermöglicht es, dass es vom Singleton genau eine Instanz gibt und es nicht möglich ist, weitere zu erstellen.

Dieses Vorgehen wurde ebenfalls in einem UML-Diagramm dargestellt.

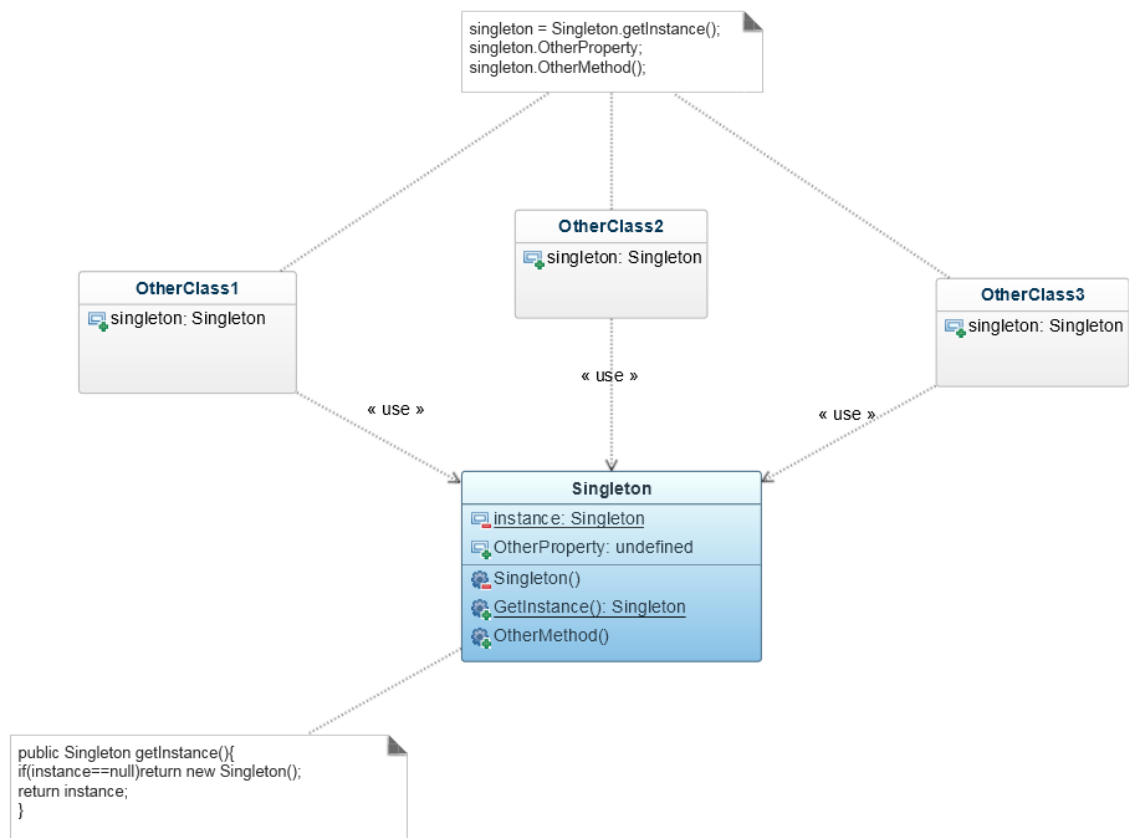


Abb. 4: Beispiel eines Singletons mit Hinweis auf Erstellung und Verwendung.

Das Entwurfsmuster „Singleton“ bietet viele Vorteile, jedoch kann es sich sehr schnell zu einer Gefahr entwickeln.

Vorteile:

- Das Singleton ist ein sehr einfaches Entwurfsmuster und ist somit sehr schnell und einfach zu schreiben.
- Die Erstellung des Singletons wird vom Singleton genau geregelt und somit auch der Zugriff.
- Durch ein Singleton bekommt man außerdem einen sauberen Namespace, da somit sehr viele globale Variablen den Namespace nicht überladen. Sie werden nur im Singleton bereitgestellt.
- Es gibt Lazy-Loading, da ein Singleton nur dann erzeugt wird, wenn es auch wirklich gebraucht wird.

Nachteile:

- Durch exzessive Verwendung kann der Programmierer Gefahr laufen, prozedural anstatt objektorientiert zu programmieren, da es sehr viel leichter ist.
- Der Bereich, in dem ein Singleton auch wirklich technisch „einzeln“ ist, muss nicht mit dem Bereich zusammenfallen, in dem es „einzeln“ sein soll.
- Das Testen eines Singletons kann kompliziert sein.
- Die Konfiguration des Singletons ist nur über andere Singletons möglich.
- Abhängigkeiten zur Singleton-Klasse werden verschleiert. Nur anhand der Implementierung, nicht aus dem Interface einer Klasse, erkennt man, ob eine Singleton-Klasse verwendet wird.
- Durch ein Singleton wird die Kopplung erhöht, was zur Folge hat, dass die Wiederverwendbarkeit und die Übersichtlichkeit eingeschränkt werden. Durch die große Anzahl von Nachteilen wird das „Singleton“ oft als schlechtes Entwurfsmuster (Anti-Pattern) bewertet. Sie können allerdings auch sehr sinnvoll sein, besonders wenn sie sich auf andere „einmalige Strukturen“ beziehen. Deshalb ist ein korrektes Design von Singleton und Disziplin notwendig, obwohl es sehr schwierig ist.

Der Grund für die Verwendung von Singletons bei diesem Projekt ist, dass es sehr viele Daten gibt, welche im ganzen Projekt gleich sind und diese auch nur einmal geben darf. Ein Beispiel dafür ist im Package-Installer Creator der XmlSerializer, aus welchem die XML-Datei für den Installer erstellt wird. Diese XML-Datei gibt es nur einmal und für einen Installer darf es nicht mehrere XML-Dateien geben. Es ist also ein idealer Einsatzbereich für ein Singleton. Ein weiteres Beispiel sind die DefaultControls. Diese Controls ändern sich ebenfalls im gesamten Programmablauf nicht und müssen von der gesamten Projektmappe verwendet werden können.

3.1.3 Factory-Method

Ein weiteres wichtiges Designpattern ist die „Factory Method“. Dieses ermöglichte uns die leichte Erweiterbarkeit der Controls, da diese dadurch ganz einfach mit Reflection erstellt werden können.

3.1.3.1 Grundlagen Factory-Method

„Der Begriff **Fabrikmethode** (englisch *factory method*) bezeichnet ein Entwurfsmuster aus dem Bereich der Softwareentwicklung. Das Muster beschreibt, wie ein Objekt durch Aufruf einer Methode anstatt durch direkten Aufruf eines Konstruktors erzeugt wird. Es gehört somit zur Kategorie der Erzeugungsmuster (engl. *creational patterns*).

Der Begriff *Fabrikmethode* kann jedoch insofern missverständlich sein, als er je nach Sprecher sowohl zwei leicht unterschiedliche Entwurfsmuster als auch die Methode selbst bezeichnet.

Das Muster ist eines der sogenannten GoF-Entwurfsmuster (*Gang of Four*, siehe Viererbande).“ (<https://de.wikipedia.org/wiki/Fabrikmethode>) [12. März 2017]

3.1.3.2 Aufbau der Factory-Method

Der Aufbau einer Factory Method ist folgender:

- Es gibt einen Erzeuger, welcher die Methoden zur Erstellung von Produkten beinhaltet.
- Es gibt einen konkreten Erzeuger, welcher vom Creator ableitet und dessen Methoden implementiert.
- Es gibt einen Basistypen für zu erzeugende Objekte (im Kontext Produkt genannt).
- Es gibt mehrere Unterprodukte, welche tatsächlich vom konkreten Erzeuger erzeugt werden.
- Die Basistypen gibt es, da damit die leichte Erweiterbarkeit gewährleistet wird.
- Im konkreten Erzeuger wird der Typ des zu erzeugenden Produkts ausgelesen und mit Reflection erzeugt.

Dieses Vorgehen ermöglicht es einem Entwickler, leicht neue Produkte hinzuzufügen, ohne am restlichen Programm etwas zu ändern.

Ein Klassendiagramm zur Factory-Method kann man sich folgendermaßen vorstellen:

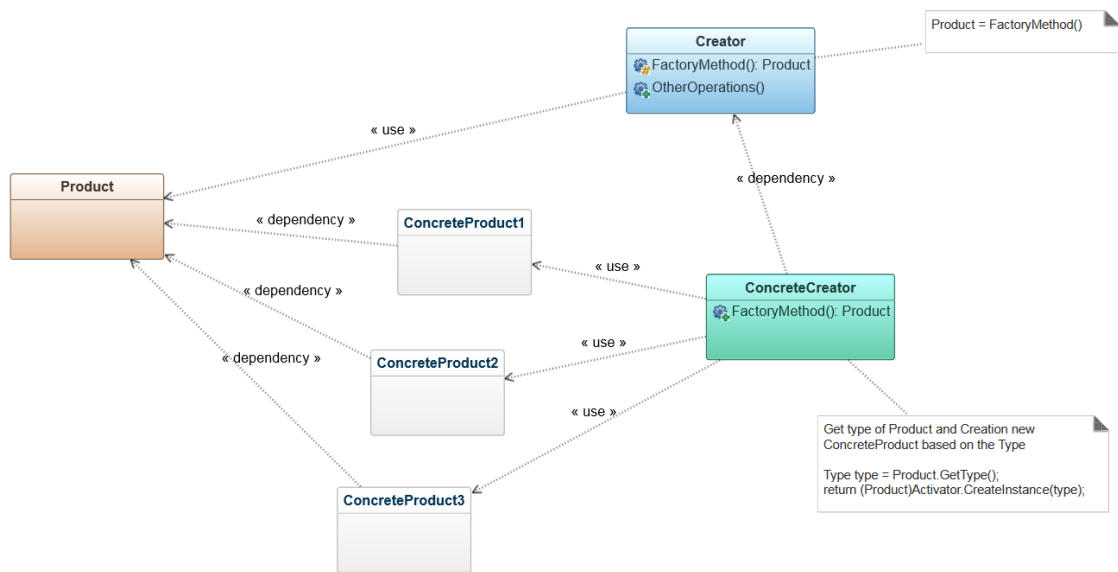


Abb. 5: Beispiel einer Factory-Method mit 3 konkreten Unterprodukten.

Die Factory-Method ist eines der beliebtesten Designpatterns und wird auch sehr häufig verwendet. Gründe dafür sind, dass diese sehr einfach zu erstellen ist, und dass es die Entwicklung neuer Produkte extrem beschleunigt. Dass die Factory-Method ein sehr gutes Designpattern ist, merkt man auch an den Vor- und Nachteilen.

Vorteile:

- Durch die Factory-Method wird der Aufrufer von der Implementierung neuer konkreter Produkt-Klassen entkoppelt, was bedeutet, dass für die Instanzierung kein „new“-Operator notwendig ist.
 - Dies ist besonders wertvoll, wenn sich eine Anwendung im Laufe der Lebenszeit weiterentwickelt, da hier neue Produkte erstellt werden können, ohne dass man die Applikation ändern muss.
- Es kann problemlos mehrere verschiedene Factory-Methods geben, welche alle einen aussagekräftigen Namen haben.

Nachteile:

- Da dieses Designpattern auf Unterklassenbildung hinausläuft, muss eine eigene Klasse vorhanden sein, die die Klassen-Methoden zur Erzeugung aufnehmen kann.

Der Grund für die Verwendung der Factory-Method ist, dass es mehrere UserControls gibt, welche alle, sowohl in der Runtime als auch im Creator, verwendet werden können. Diese sollen leicht und ohne Probleme erweiterbar sein.

3.2 XML Schema Definition (XSD)

Unsere Datenstruktur sollte im Format XML abgespeichert werden, wozu ein XML-Schema erarbeitet wurde. Ein „Schema“ definiert die genauen Bezeichnungen der einzelnen Elemente im XML-Dokument. Damit kann man also schnell feststellen, ob die gewünschte XML-Datei nicht nur syntaktisch richtig ist, sondern auch dem passenden Format entspricht.

Das Schema wurde manuell im Format „XML Schema Definition“ erstellt. Eine Alternative wäre die „Document Type Definition“ (DTD) gewesen. Diese hat aber gegenüber der XSD verschiedene Nachteile:

- DTD basiert noch auf der älteren SGML-Syntax, während XSD selbst ein XML-Dokument ist. XML ist übersichtlicher und simpler aufgebaut als SGML.
- DTD kann keine Datentypen definieren.
- DTD kann nicht Anzahl und Reihenfolge der Elemente festlegen

Vorteile von DTD:

- Man kann die DTD im eigentlichen XML-Dokument mitgeben, das ist bei XSD nicht möglich

Zum besseren Verständnis ist nachfolgend ein einfaches DTD-Schema abgedruckt. Es zeigt einen Tourenplan eines Transportunternehmens.

```
<!ELEMENT tourenplan (fahrzeug+,fahrer+,tour*)>
<!ELEMENT fahrzeug (#PCDATA)>
<!ATTLIST fahrzeug kennzeichen ID #REQUIRED>
<!ELEMENT fahrer (#PCDATA)>
<!ATTLIST fahrer personalnummer ID #REQUIRED>
<!ELEMENT tour (datum,uhrzeit,strecke)>
<!ATTLIST tour kennzeichen IDREF #REQUIRED>
<!ATTLIST tour fahrer IDREFS #IMPLIED>
<!ELEMENT datum (#PCDATA)>
<!ELEMENT uhrzeit (#PCDATA)>
<!ELEMENT strecke (von,bis)>
<!ELEMENT von (#PCDATA)>
<!ELEMENT bis (#PCDATA)>
```

Abb. 6: Beispiel eines DTD-Schemas anhand eines Tourenplans.

Im Vergleich dazu ist ein Auszug aus einer typischen XSD-Deklaration zu sehen. Diese beschreibt die Organisation einer Schule.

```

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/Schema"
        xmlns:tns="http://www.example.org/Schema"
        elementFormDefault="qualified">
  <element name="schule">
    <complexType>
      <sequence>
        <element name="klasse" minOccurs="1" maxOccurs="50">
          <complexType></complexType>
        </element>
        <element name="lehrer" type="string" maxOccurs="200"></element>
      </sequence>
    </complexType>
  </element>
  <attribute name="name" type="string"></attribute>
  <complexType name="person"></complexType>
</schema>

```

Abb. 7: Beispiel eines XSD-Schemas anhand der Organisation einer Schule.

Wie man an den beiden Grafiken sieht, ist XSD leichter zugänglich, da die Verschachtelungen in etwa dem tatsächlichen Endprodukt entsprechen. Die sequenzielle Angabe der DTD ist da weniger verständlich. Außerdem kann man bei der XSD Attribute wie „maxOccurs“ (Maximalanzahl) erkennen, die bei DTD völlig fehlen.

Unter <http://xsd2xml.com/> kann man sich aus einer XSD ein beispielhafte XML-Datei generieren lassen. Dies kann zu Testzwecken oftmals sehr hilfreich sein.

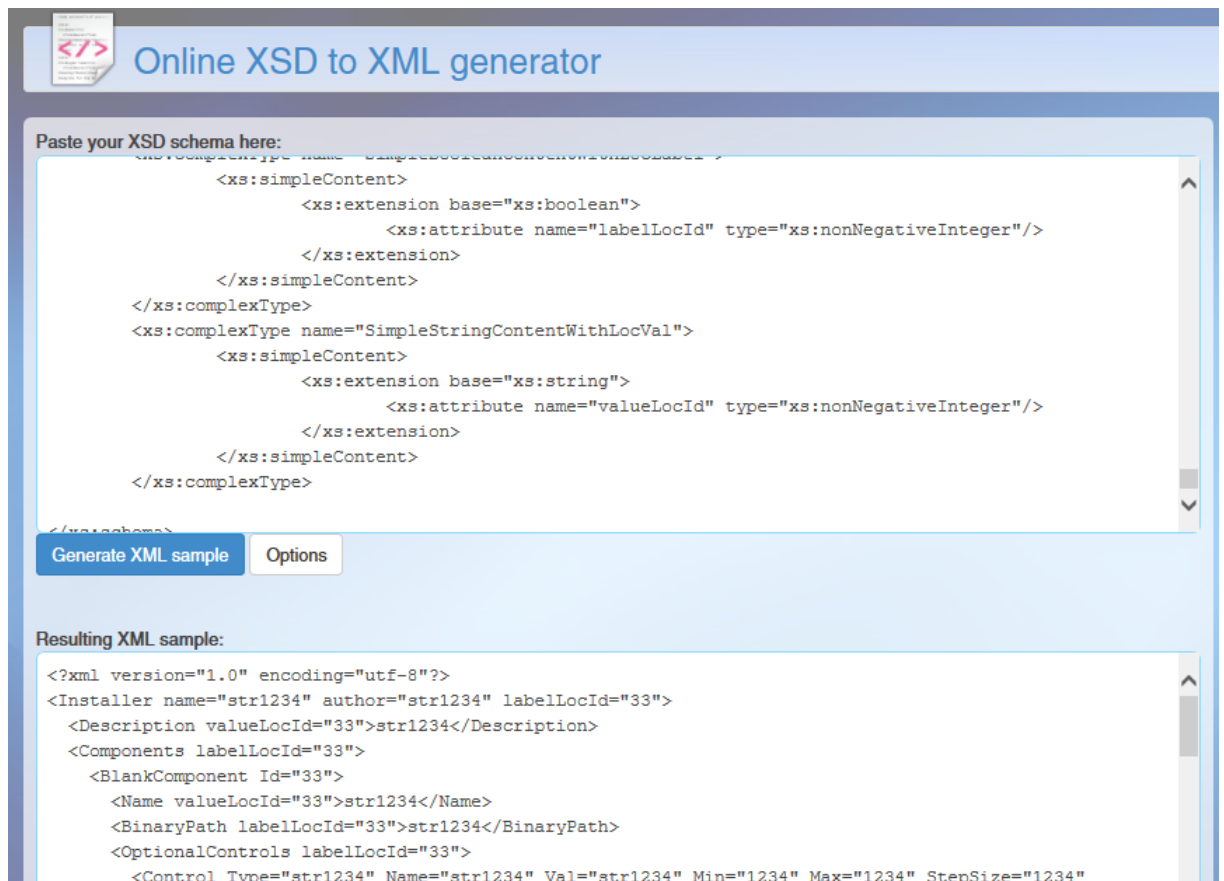


Abb. 8: Beispiel eines XSD-zu-XML-Generators.

Das hierdurch erstellte XML-Dokument ist natürlich semantisch nicht im Geringsten sinnvoll, zur kurzen technischen Tests o. ä. reicht dies aber allemal.

3.2.1 Das Schema bei unserer Diplomarbeit

Das Hauptelement trägt die Bezeichnung `Installer`. Es enthält die Attribute „Name“ (Bezeichnung des Installers) und „Author“ (erstellende Person). Darunter befindet sich zunächst das Element „Description“. Dieses enthält frei wählbare Erläuterungen, die im Creator angegeben werden können. Diese Funktion kann sehr nützlich sein, da damit der erstellte Installer gut dokumentiert werden kann. Wenn z.B. ein anderer Entwickler das Projekt übernimmt, weiß er sofort über alle Besonderheiten Bescheid, wenn der Vorgänger diese Anmerkungen ausreichend angegeben hat.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="XMLSchema.xsd">
  <xs:element name="Installer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Description" type="SimpleStringContentWithLocVal">

```

Abb. 9: Teilbereich des XSD-Schemas, welcher den Installer und die Description definiert.

Aus diesem Schema-Ausschnitt resultiert folgender XML-Code:

```
<?xml version="1.0" encoding="utf-8"?>
<Installer name="Installer" author="Florian Redinger" labelLocId="0">
  <Description valueLocId="1">Die Beschreibung des Installers</Description>
  <Components labelLocId="2">
    <BlankComponent Id="0">...</BlankComponent>
    <MsiComponent Id="1">...</MsiComponent>
    <TradeAwareMsiComponent Id="2">...</TradeAwareMsiComponent>
    <OracleDbComponent Id="3">...</OracleDbComponent>
  </Components>
  <ComponentGroups labelLocId="32">...</ComponentGroups>
  <DependencyGroups labelLocId="37">...</DependencyGroups>
</Installer>
```

Abb. 10: Darstellung der aus Abb. 9 erstellten XML-Elemente.

Danach folgt die Liste der Komponenten („Components“). Eine einzelne Komponente kann etwa ein „BlankComponent“ sein. Ein solches Element besteht dann aus den Elementen „Name“ (Bezeichnung des Elements) und „BinaryPath“ (Dateipfad zur Komponente). Eine solche Komponente ist außerdem durch eine ID (Attribut „Id“) eindeutig gekennzeichnet. Dass die ID eindeutig sein muss, ist allerdings in der XSD nicht angegeben.



```
<xs:element name="Components">
  <xs:complexType>
    <xs:sequence>
      <xs:element type="BlankComponent" name="BlankComponent" minOccurs="0"
        maxOccurs="unbounded"/>
      <xs:element type="MsiComponent" name="MsiComponent" minOccurs="0"
        maxOccurs="unbounded"/>
      <xs:element type="TradeAwareMsiComponent" name="TradeAwareMsiComponent"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element type="OracleDbComponent" name="OracleDbComponent" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="labelLocId" type="xs:nonNegativeInteger"/>
  </xs:complexType>
</xs:element>
```

Abb. 11: Darstellung der Komponenten-Liste in einem XSD-Schema.



```
<xs:complexType name="BlankComponent">
  <xs:sequence>
    <xs:group ref="BaseComponentGroup"/>
    <xs:element name="OptionalControls">
      <xs:complexType>
        <xs:group ref="ControlGroup"/>
        <xs:attribute name="labelLocId" type="xs:nonNegativeInteger"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="Id" type="xs:nonNegativeInteger"/>
</xs:complexType>
```

Abb. 12: Beispiel der XSD einer spezifischen Komponente.

Unter diesem Bereich des Schemas wäre etwa folgender XML-Code zulässig:

```
<BlankComponent Id="0">
  <Name valueLocId="3">MyBlankComponent</Name>
  <BinaryPath labelLocId="4">C:\Users\YPM\Desktop</BinaryPath>
  <OptionalControls labelLocId="5">...</OptionalControls>
</BlankComponent>
```

Abb. 13: Erstelltes XML-Element aus dem in Abb. 12 gegebenen Schema

Den beiden Elementen „Name“ und „BinaryPath“ folgend, findet sich ein weiteres Container-Element, nämlich „OptionalControls“. Dieses enthält ein oder mehrere Elemente vom Typ „Control“. Ein „Control“ stellt in der Runtime genau ein Eingabefeld dar. Ein „Control“ enthält folgende Attribute: „Name“ (Bezeichnung), „Val“ (Standardwert) und „ControlType“ (Art des Controls – dies kann sein: Text, Number, Password, etc.). Bei spezifischen Typen können weitere Attribute hinzukommen. Bei Number (Zahlenwerte) etwa „Min“, „Max“ und „StepSize“ (Schrittgröße des Number-Pickers), bei Text (StringControl) ein Regex-Pattern.

Eine Komponente mit einem optionalen Control könnte folgenden XML-Code enthalten:

```
<OptionalControls labelLocId="5">
  <Control Type="TextControl" Name="BlankControl" Val="1"
    ControlType="Normal" labelLocId="6" />
</OptionalControls>
```

Abb. 14: Erstelltes XML-Element „OptionalControls“ und dessen Kind-Elemente, aus dem in Abb. 12 gegebenen Schema

Ein Nicht-Blank-Component ist fast identisch aufgebaut. Es enthält aber zwischen dem „BinaryPath“ und den „OptionalControls“ noch die „MandatoryControls“. Dieses Element umfasst spezifische Eigenschaften des jeweiligen Komponententyps.



```
<xs:complexType name="MsiComponent">
  <xs:sequence>
    <xs:group ref="BaseComponentGroup"/>
    <xs:element name="MandatoryControls">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="MsiComponentGroup"/>
        </xs:sequence>
        <xs:attribute name="labelLocId" type="xs:nonNegativeInteger"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="OptionalControls">
      <xs:complexType>
        <xs:group ref="ControlGroup"/>
        <xs:attribute name="labelLocId" type="xs:nonNegativeInteger"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="Id" type="xs:nonNegativeInteger"/>
</xs:complexType>
```

Abb. 15: Beispiel eines XSD-Schemas für einen bestimmten Komponenten-Typen mit obligatorischen Elementen.

Folgende Arten von Komponenten sind derzeit möglich (eine genaue Beschreibung dieser findet sich in Kapitel 3.3.11 Komponenten-Typen):

Tabelle 6: Kurze Beschreibung der einzelnen Komponenten-Typen

Komponente	Bedeutung	enthaltene MandatoryControls
MsiComponent	.msi-Installer	InstallPath
TradeAwareMsiComponent	.msi-Installer mit „Trade“-Unterstützung	InstallPath, Branding, Server
OracleComponent	Oracle Datenbank	TnsNames, Username, Password, Schema, Drop, Update

Eine MSI-Komponente könnte in der XML-Datei so beschrieben sein:

```
<MsiComponent Id="1">
  <Name valueLocId="7">WindowsServices</Name>
  <BinaryPath labelLocId="8">C:\Users\YPM\Desktop</BinaryPath>
  <MandatoryControls labelLocId="9">
    <InstallPath labelLocId="10">C:\Users\YPM\Desktop</InstallPath>
  </MandatoryControls>
  <OptionalControls labelLocId="11">
    <Control Type="NumberControl" Name="Some Number" Val="0"
      ControlType="Normal" labelLocId="12" Max="100" />
  </OptionalControls>
</MsiComponent>
```

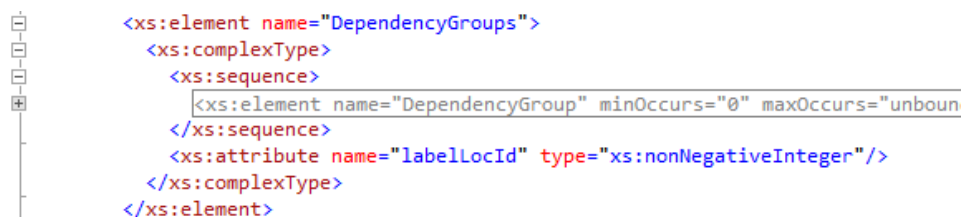
Abb. 16: Mögliche Erstellung eines XML-Elements nach dem in Abb. 15 gegebenen Schema.

Die Container-Elemente „ComponentGroups“ und DependencyGroups“ folgen den Components nach. Diese stellen die fixen Komponenten-Gruppen bzw. die Abhängigkeiten zwischen den Komponenten dar.



```
<xs:element name="ComponentGroups">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ComponentGroup" minOccurs="0" maxOccurs="unbound" />
    </xs:sequence>
    <xs:attribute name="labelLocId" type="xs:nonNegativeInteger"/>
  </xs:complexType>
</xs:element>
```

Abb. 17: XSD-Schema einer Komponenten-Gruppe



```
<xs:element name="DependencyGroups">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DependencyGroup" minOccurs="0" maxOccurs="unbound" />
    </xs:sequence>
    <xs:attribute name="labelLocId" type="xs:nonNegativeInteger"/>
  </xs:complexType>
</xs:element>
```

Abb. 18: XSD-Schema einer Abhängigkeits-Gruppe

Solche ComponentGroups könnten zum Beispiel so aussehen:

```
<ComponentGroups labelLocId="32">
  <ComponentGroup name="Client" compGrpId="0" labelLocId="33">
    <Description valueLocId="34">ClientInstallation</Description>
    <Comps>
      <Comp compId="1"/>
      <Comp compId="2"/>
    </Comps>
  </ComponentGroup>
  <ComponentGroup name="Server" compGrpId="1" labelLocId="35">
    <Description valueLocId="36">ServerInstallation</Description>
    <Comps>
      <Comp compId="3"/>
    </Comps>
  </ComponentGroup>
</ComponentGroups>
```

Abb. 19: Erstelltes XML-Element nach dem in Abb. 17 gegebenen Schema.

In diesem Beispiel besteht eine Client-Installation aus den Komponenten Nr. 1 und 2, eine Server-Installation nur aus der dritten Komponente.

Eine DependencyGroup könnte dann so aufgebaut sein:

```
<DependencyGroups labelLocId="37">
  <DependencyGroup name="Client" depGrpId="0" labelLocId="38">
    <Comps>
      <Comp compId="1"/>
      <Comp compId="2"/>
    </Comps>
  </DependencyGroup>
</DependencyGroups>
```

Abb. 20: Erstelltes XML-Element nach dem in Abb. 18 gegebenen Schema.

Wenn man Komponente Nr. 1 anwählt, muss man also auch Nr. 2 nehmen und umgekehrt.

Als bewusste Erfüllung der Anforderung „Erweiterbarkeit“ ist das ganze Programm so gestaltet, dass sich neue Komponententypen ohne viel Aufwand in das Programm integrieren lassen.

3.3 Package-Installer Creator

Mit dem Package-Installer Creator wird die Erstellung mehrerer Komponenten ermöglicht. Für jede dieser Komponenten gibt es ein „Binary“ (MSI, Zip, ...), welches die zu installierenden Dateien enthält. Diese Komponenten werden in eine XML-Datei serialisiert, welche von der Package-Installer Runtime verwendet wird. Weiters sollen die einzelnen Komponenten für mehrere Sprachen zur Verfügung gestellt werden können.

Zur besseren Übersicht lassen sich sämtliche Bilder zu den UI-Oberflächen des Creators im Anhang A-C finden.

3.3.1 Shell und Menüs

Die grafische Oberfläche ist vollständig mit WPF aufgebaut. Um das Aussehen ansprechender zu gestalten, wurde auf jegliche Form von Windows-Standard-Designelementen verzichtet. Stattdessen wurde das ganze Fenster selbst designt. Welchen Aufwand dies mit sich brachte, ist im nachfolgenden Kapitel 3.3.2 in diesem Dokument zu lesen.

Das „Drop-Down“-Menü ist vollständig dynamisch aufgebaut. Das bedeutet, um einen neuen Eintrag in das Menü hinzuzufügen, ist es nicht notwendig, den XAML-Code zu verändern. Im tatsächlichen XAML-Code (Datei: Shell.xaml) ist innerhalb des „Menu“ nur ein einziges „MenuItem“ zu finden. Bei diesem ist die Eigenschaft ItemsSource durch ein Binding auf die Property MenuItems im ShellViewModel verknüpft.

```
<Menu Grid.Column="2" VerticalAlignment="Stretch" Visibility="{Binding Path=MTopVis}"
      VerticalContentAlignment="Center" Background="Transparent" Name="TopMenu" FontSize="14">
  <MenuItem ItemsSource="{Binding Path=MenuItems}" VerticalAlignment="Stretch"
            HorizontalAlignment="Stretch" Height="35">
    <MenuItem.Header>
      <TextBlock Text="Project" FontSize="16" VerticalAlignment="Center"
                HorizontalAlignment="Stretch" Foreground="{StaticResource WhiteBackground}"/>
    </MenuItem.Header>
  </MenuItem>
</Menu>
```

Abb. 21: Menüdefinition im Shell-Fenster

```

private ObservableCollection<object> menuItems;
2 Verweise
public ObservableCollection<object> MenuItems
{
    get { return menuItems; }
    set
    {
        menuItems = value;
        OnPropertyChanged(nameof(MenuItems));
    }
}

```

Abb. 22: Die in XAML gebundene Liste für das Menü.

Bei der Erstellung des ShellViewModels (Konstruktor) wird unter anderem die Methode LoadCommands aufgerufen. Hier werden zunächst die Commands den jeweiligen Properties zugewiesen. Es handelt sich dabei um Exit, BackClicked, BuildInstaller, SaveInstaller, LoadPage. Die Klasse „RelayCommand“ wird in dem gleichnamigen Unterkapitel erklärt.

```

1 Verweis
private void LoadCommands()
{
    ExitCommand = new RelayCommand(x => ExitApplication());
    BackClickedCommand = new RelayCommand(x=>BackClicked());
    BuildInstallerCommand = new RelayCommand(x => BuildInstaller());
    SaveInstallerCommand = new RelayCommand(x => SaveInstaller());
    LoadPageCommand = new RelayCommand(x=>LoadPage((List<PageInformation>)x));
}

```

Abb. 23: Initialisierung der einzelnen Kommandos.

Das Menü wird aufgebaut, wenn das CreationViewModel erstellt wird (Konstruktor). Dort wird neben anderen Methoden „DefineMenu“ aufgerufen. Dort wird eine Liste von MenuItems angelegt und befüllt. ExitApplication() wird etwa dem ExitCommand zugewiesen. Erläuterungen zu dieser Methode sind im nächsten Kapitel zu finden.


```

public void DefineMenu()
{
    ICommand loadCompsCommand = new RelayCommand(x => LoadComponents());
    ICommand loadContsCommand = new RelayCommand(x => LoadControls());
    //ICommand loadLocCommand = new RelayCommand(x => LoadLocalization());

    List<FrameworkElement> menuItems = new List<FrameworkElement>();
    menuItems.Add(new MenuItem() { Header = "Build...", Command = Shell.BuildInstallerCommand, InputGestureText = "F5" });
    menuItems.Add(new MenuItem() { Header = "Save", Command = Shell.SaveInstallerCommand, InputGestureText = "CTRL+S" });
    menuItems.Add(new Separator());
    menuItems.Add(new MenuItem() { Header = "Load Components", Command = loadCompsCommand });
    menuItems.Add(new MenuItem() { Header = "Load Controls", Command = loadContsCommand });
    menuItems.Add(new Separator());
    menuItems.Add(new MenuItem()
    {
        Name = "miLocalization",
        Header = "Localization",
        Command = Shell.LoadPageCommand,
        CommandParameter = new List<PageInformation> { new PageInformation() { Page = new Localization(Shell) } }
    });
    menuItems.Add(new Separator());
    menuItems.Add(new MenuItem() { Header = "Exit", Command = Shell.ExitCommand, InputGestureText = "CTRL+Q" });

    menuItems.BuildMenu(Shell.MenuItems);
}

```

Abb. 24: Erstellung des Menüs

Ein MenuItem (System.Windows.Controls) hat einen Header (Titel), ein Command (in unserem Falle das soeben zugewiesene) und optional einen InputGestureText (Hinweis zur Tastenkombination).

Die tatsächliche Erstellung erfolgt mit dem Aufruf von BuildMenu.

```

1 Verweis
public static void BuildMenu(this List<FrameworkElement> menuItems, ObservableCollection<object> items)
{
    items.Clear();
    foreach (FrameworkElement frameworkElement in menuItems)
    {
        items.Add(frameworkElement as object);
    }
}

```

Abb. 25: Notwendige Umwandlung der Liste vom Typ „FrameworkElement“ in eine Liste vom Typ „object“.

BuildMenu ist eine Erweiterungsmethode von MenuItem. Sie kopiert die Items in die angegebene Collection hinein.

3.3.1.1 Menüpunkt „Exit“

Die Methode `ExitApplication()` überprüft zunächst auf ungespeicherte Änderungen. Wenn diese nicht vorhanden sind, wird mit `Application.Current.Shutdown()` das Programm beendet. Ungespeicherte Änderungen werden im Singleton abgefragt. Die dortige Property „`UnsavedChanges`“ ist zunächst `false` und wird bei der ersten Änderung auf `true` gesetzt. Ansonsten wird der Exit-`ProgramDialog` aufgerufen.

```
1 Verweis
private void ExitApplication()
{
    if (!config.UnsavedChanges)
        System.Windows.Application.Current.Shutdown();
    else
    {
        ExitProgramDialog dia = new ExitProgramDialog();
        dia.Owner = Shell;
        var result = dia.ShowDialog();
        if(result != null && result.Value)
        {
            SaveInstaller();
        }
    }
}
```

Abb. 26: Methode, welche aufgerufen wird, wenn das Programm geschlossen werden soll.

Dieser Dialog fragt, ob die Änderungen gespeichert werden sollen. Bei „Cancel“ geschieht nichts, bei „No“ wird das Programm beendet und bei „Yes“ wird der Installer gespeichert. Um das Speichern zu ermöglichen, wird die Methode `SaveInstaller()` aufgerufen.

```
1 Verweis
private void BtnYes_OnClick(object sender, RoutedEventArgs e)
{
    DialogResult = true;
}

1 Verweis
private void BtnNo_OnClick(object sender, RoutedEventArgs e)
{
    System.Windows.Application.Current.Shutdown();
}

1 Verweis
private void BtnCancel_OnClick(object sender, RoutedEventArgs e)
{
    DialogResult = false;
}
```

Abb. 28: Button-Events des Dialogs



Abb. 27: Dialog für ungespeicherte Änderungen

3.3.1.2 RelayCommand

```
27 Verweise
public class RelayCommand : ICommand
{
    readonly Action<object> execute;
    readonly Predicate<object> canExecute;

25 Verweise
    public RelayCommand(Action<object> execute) : this(execute, null)
    {
    }

    1 Verweis
    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        if (execute == null) throw new ArgumentNullException(nameof(execute));

        this.execute = execute;
        this.canExecute = canExecute;
    }

    public event EventHandler CanExecuteChanged {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }

    1 Verweis
    public bool CanExecute(object parameter)
    {
        return canExecute?.Invoke(parameter) ?? true;
    }

    10 Verweise
    public void Execute(object parameter)
    {
        execute(parameter);
    }
}
```

Abb. 29: Definition eines RelayCommands

RelayCommand ist eine Eigenimplementierung von ICommand. Diese tut nichts, außer eine angegebene Methode mit eventuellem Parameter aufzurufen. Das ist eine häufige Vorgehensweise und wird als Common-Practice angesehen.

ICommand ist ein Interface zur Modellierung von Befehlen. Für das MVVM-Muster (siehe Kapitel 3.1.1 „Model View ViewModel (MVVM)“) ist ein solches Element unverzichtbar.

ICommand besteht im Prinzip aus drei zu implementierenden Methoden. „Execute“, „CanExecute“ und „CanExecuteChanged“.

„Execute“ wird ausgeführt, wenn der Command aufgerufen wird. Dies wird so gelöst, dass der gewünschte Parameter einfach an die angegebene Methode weitergegeben wird.

Mit „CanExecute“ kann man eine Fallunterscheidung angeben, ob der Command überhaupt ausgeführt werden soll oder nicht. Falls gewünscht, leiten wir das Ganze ebenfalls weiter. Davon wird in diesem Projekt aber kaum Gebrauch gemacht.

3.3.1.3 AsyncRelayCommand

```
public class AsyncRelayCommand : ICommand
{
    private readonly Func<object, Task> _execute;
    private readonly Func<object, bool> _canExecute;
    private bool _isExecuting;

    2 Verweise | Florian Redinger, vor 57 Tagen | 1 Autor, 2 Änderungen
    public AsyncRelayCommand(Func<object, Task> execute) : this(execute, (x) => true)
    {
    }

    1 Verweis | Florian Redinger, vor 50 Tagen | 1 Autor, 1 Änderung
    public AsyncRelayCommand(Func<object, Task> execute, Func<object, bool> canExecute)
    {
        _execute = execute;
        _canExecute = canExecute;
    }

    1 Verweis | Florian Redinger, vor 50 Tagen | 1 Autor, 3 Änderungen
    public bool CanExecute(object parameter)
    {
        if(parameter==null) return !(_isExecuting && _canExecute(true));
        return !(_isExecuting && _canExecute(parameter));
    }

    public event EventHandler CanExecuteChanged;
}
```

Abb. 30: Definition eines AsyncRelayCommands

```
13 Verweise | Florian Redinger, vor 71 Tagen | 1 Autor, 1 Änderung
public async void Execute(object parameter)
{
    _isExecuting = true;
    OnCanExecuteChanged();
    try
    {
        await _execute(parameter);
    }
    catch (ArgumentOutOfRangeException exc)
    {
        Console.WriteLine(exc.Message);
    }
    finally
    {
        _isExecuting = false;
        OnCanExecuteChanged();
    }
}

2 Verweise | Florian Redinger, vor 71 Tagen | 1 Autor, 1 Änderung
protected virtual void OnCanExecuteChanged()
{
    if (CanExecuteChanged != null) CanExecuteChanged(this, new EventArgs());
}
}
```

Abb. 31: Execute Methode des AsyncRelayCommands und das Event zur Überwachung von CanExecute.

AsyncRelayCommand ist eine weitere Implementierung von ICommand. Sie ist dem RelayCommand sehr ähnlich aufgebaut. Der Unterschied besteht in der Verwendung eines AsyncTasks. Damit ist es leicht möglich, asynchrone Programmierung im Projekt anzuwenden. Das Funktionsprinzip ist simpel: Statt in der Execute-Methode die mitgelieferte Action direkt auszuführen,

wird das Schlüsselwort „await“ davorgesetzt. Der Methodenaufruf bleibt aber ansonsten identisch. Der Befehl wird damit im Hintergrund ausgeführt. Diese Technik der asynchronen Programmierung wird sehr gerne angewendet, um Performance-Flaschenhälse zu vermeiden und insgesamt die ganze Anwendung flüssiger erscheinen zu lassen. Auf eine Implementierung von „CanExecute“ wurde aus dem oben genannten Grund verzichtet.

Anstelle dieser Eigenentwicklung könnte man auch ein fertiges MVVM-Framework zurückgreifen. Entwickler der TGW empfehlen zum Beispiel die Microsoft Prism Library.

3.3.2 Window Behavior

Um es zu ermöglichen, eine selbst designte Toolbar zu kreieren, ist es notwendig, das Window-Behaviour von Windows 7 selbst zu erstellen.

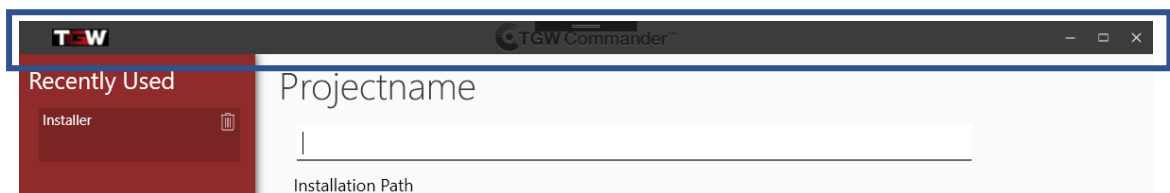


Abb. 32: Selbst designte Toolbar

Um diese Funktion bereitzustellen, sind mehrere Schritte notwendig. Einerseits muss das Vergrößern/Verkleinern vom Fenster mit dem WPF-Attribute `ResizeMode` zugelassen werden, andererseits ist es notwendig, das sogenannte „WindowChrome“ zu verwenden, um grundlegende, nicht fertig verwendbare Funktionen von Windows 7 zu ermöglichen (Vergrößern, Verkleinern, auf halber/viertel Seite sichtbar, etc.).

Um das Fenster auf den ursprünglichen Zustand zurückzusetzen, sind weitere Schritte notwendig.

Zuerst müssen alle Betriebssystem-Nachrichten überwacht werden. Dies geschieht, indem man sich über einen `WindowInteropHelper`, welchem man das aktuelle Objekt übergibt, den Pointer zum Fenster holt. Über eine `HwndSource`, welche die Methode „`AddHook`“ zur Verfügung stellt, ist es möglich, alle Fensternachrichten zu überwachen.

Wenn eine Fensternachricht auftritt, wird der Callback „`WindowProc`“ aufgerufen, bei welchem überprüft wird, um welche Art von Message es sich handelt. In unserem Fall ist es die Message „`0x0024`“, bei welcher die Methode „`WmGetMinMaxInfo`“ aufgerufen wird.

Bei dieser Methode gibt es einige Kalkulationen zum Herausfinden von Monitor-Informationen. Es ist für das Fenster nämlich wichtig, auf welchem Monitor sich das Fenster befindet. Um dies herauszufinden, wird die Methode „MonitorFromPoint“ aus der „user32.dll“ verwendet. Damit ist es nämlich auch möglich herauszufinden, ob der Monitor, auf dem man sich gerade befindet, auch der primäre Monitor ist. Dies ist notwendig zu wissen, da die Monitorinformationen, abhängig davon, ob diese beide gleich sind, anders ausgelesen werden müssen.

Abschließend werden noch mit „Marshal.StructureToPtr“ die Monitor-Informationen auf den Pointer des Fensters geschrieben. Damit wird der Speicherbereich des Fensters beschrieben und dieses weiß genau, wie es sich skalieren muss.

3.3.3 Arten von Controls

Um die leichte Erweiterbarkeit in dem Bereich der Controls zu ermöglichen, wird eine Factory-Method eingebaut. Es gibt einerseits eine Basisklasse „ControlBase“, welche sich um das Binding an bestimmte, für alle gleiche Attribute kümmert. Sie verwaltet auch Dependency Properties (z. B. ControlTitle), welche ebenfalls für alle Controls gleich sind. Dies ermöglicht es den Weiterentwickelnden, sehr einfach neue Controls hinzuzufügen, solange sie sich an die korrekte Namensgebung und die richtige Struktur der verschiedenen WPF-Controls halten.

Weiters wird ein Interface „IDefaultControl“ eingebaut. Dieses ermöglicht es, bestimmte Methoden, welche von allen „DefaultControls“ benötigt werden, die allerdings controlspezifisch sind, einzubinden (z. B. GetValue).

Da es Situationen gibt, in denen das Label und der Inhalt des Controls nur in einer Zeile angezeigt werden, und andere, in denen diese beiden Bestandteile in zwei Zeilen angezeigt werden, muss ein sogenannter „ShapeChanger“ eingebaut werden. Dieser ändert das Aussehen der UserControls basierend auf dem Property „OneRow“, welches vom Benutzer zur Designzeit angegeben werden kann.

Die Default-Controls, welche derzeit verwendet werden, sind:

- TextControl
 - Das TextControl ist ein sehr einfaches Control zur Eingabe eines Textes.
 - Es wird auch ermöglicht, das Control auf ein PasswordControl umzustellen, indem der Benutzer das Property „IsPasswordBox“ auf „true“ setzt.



Abb. 33: TextControl

- PathControl
 - Mithilfe des PathControls ist es möglich, Dateien auszuwählen.
 - Die Arten des PathControls sind SaveFile, SaveFolder, OpenFile und OpenFolder.



Abb. 34: PathControl

- NumberControl
 - Diese Control stellt einen sog. NumberPicker zur Verfügung, also ein Control zur Eingabe von Zahlen.
 - Dem Benutzer wird dabei ermöglicht, ein Maximum, Minimum und eine Step-Size einzustellen.

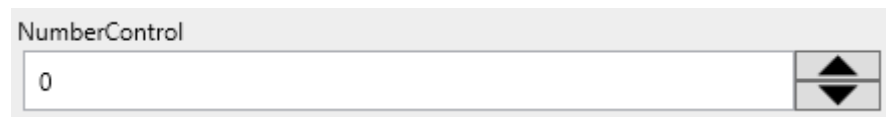


Abb. 35: NumberControl

- BoolControl
 - Mit dem BoolControl wird eine einfache „CheckBox“ mit Titel dargestellt.

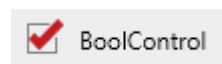


Abb. 36: BoolControl

- DateControl
 - Mit dem DateControl wird es dem Benutzer ermöglicht, ein bestimmtes Datum zu wählen.

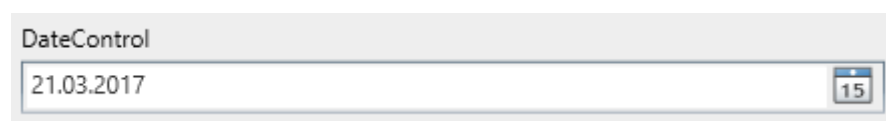


Abb. 37: DateControl

- TnsNamesControl

- Beim TnsNamesControl wird die TnsNames.ora Datei ausgelesen und jeder TnsName wird in der Combobox angezeigt. Der Pfad zur TnsNames.ora Datei wird in den Systemvariablen als „TNS_ADMIN“ angelegt.



Abb. 38: TnsNamesControl

- CustomControl
 - Während des gesamten Creations-Prozesses wird es dem Benutzer ermöglicht, neue, selbst gebaute Controls zu erstellen.
 - Dadurch kann man zum Beispiel ein Control „Login“, bestehend aus zwei Controls, zur Wiederverwendung erstellen.
 - Diese CustomControls können ebenso von einer externen Quelle geladen werden.
 - Es ist möglich, CustomControls für spätere Verwendung zu speichern.

Die Struktur der DefaultControls kann mit folgendem UML-Diagramm dargestellt werden:

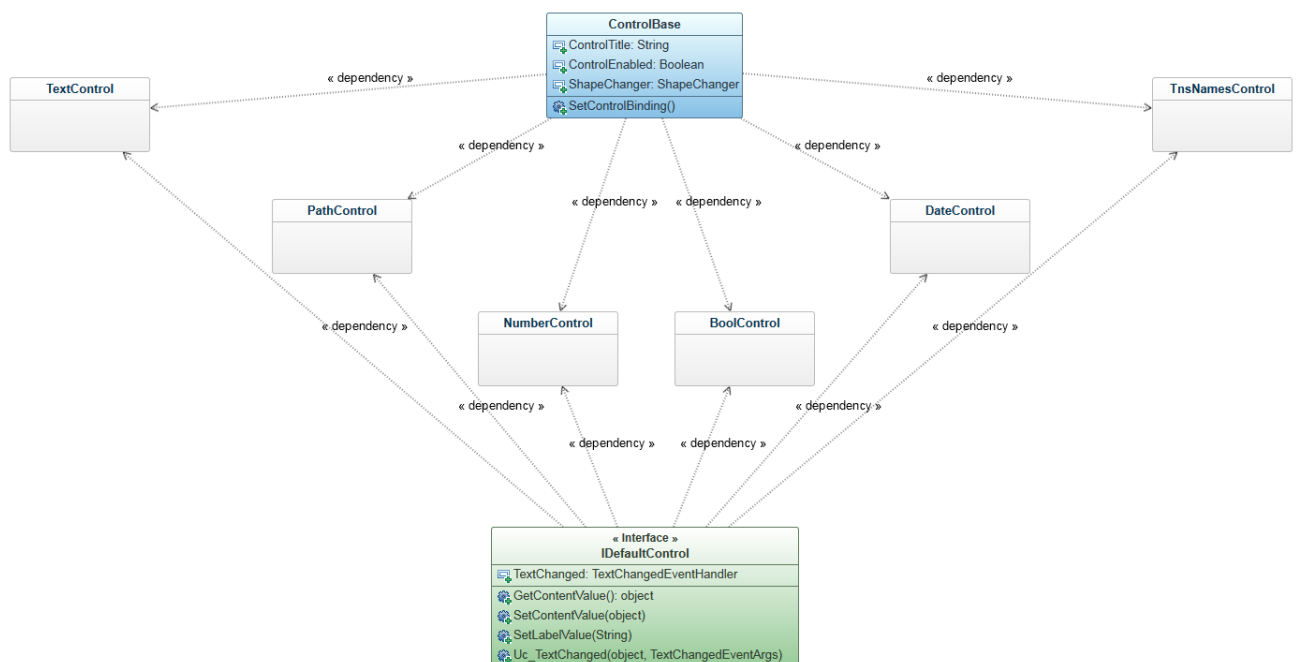


Abb. 39: Schema des Aufbaus der verschiedenen Controls

3.3.4 Neues Installer Projekt

Wird der Package-Installer Creator gestartet, erinnert er an Microsoft PowerPoint. Dies ist beabsichtigt, da sehr viele Kunden von der TGW größtenteils Microsoft-Produkte gewöhnt sind. Das bringt mit sich, dass der Package-Installer Creator für die Kunden leichter zu bedienen ist und nur eine geringe Einschulung erfordert.

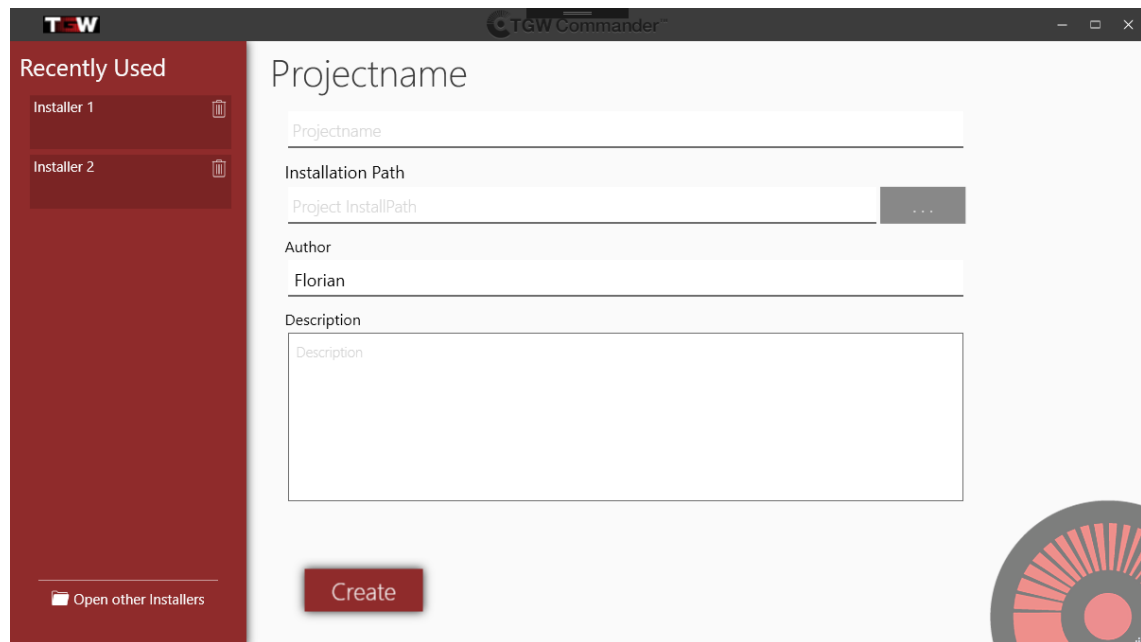


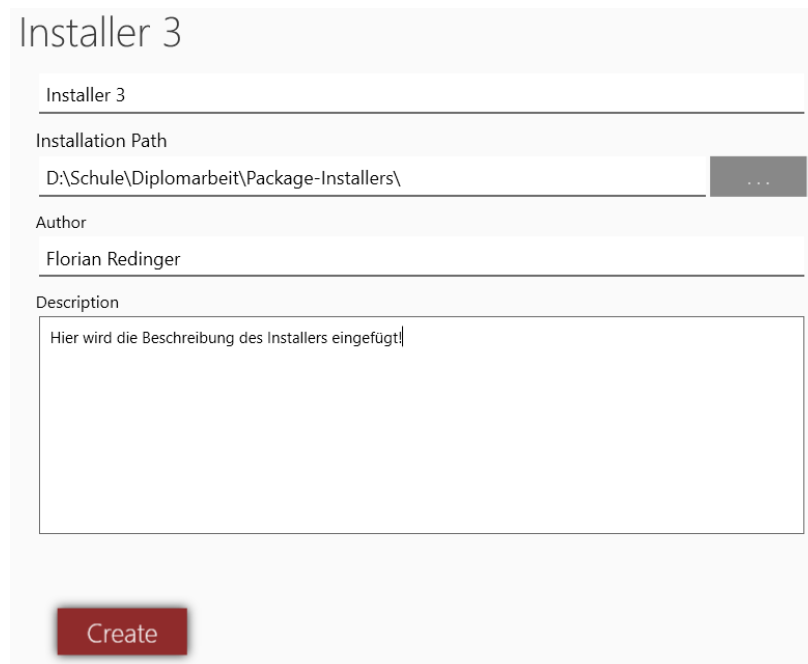
Abb. 40: Startup-Fenster des Package-Installer Creators

3.3.4.1 Projektinformationen eingeben

Auf diesem Bildschirm ist es für den Benutzer möglich, die wichtigsten Informationen für den Installer einzugeben. Dazu gehören der Name, der Pfad, auf dem der Installer erstellt werden soll, und der Autor, welcher beim Start des Projekts aus dem Betriebssystem geladen wird. Wenn das Programm keinen Autor findet, bzw. dieser nicht korrekt ist, kann der Benutzer diesen bearbeiten. Abschließend ist es noch möglich, eine Beschreibung anzugeben, um für spätere Nutzer die Verwendung zu vereinfachen. Nach dem Lesen der Beschreibung, wissen Nutzer sofort, für welche Komponenten der Installer verwendet wird.

Hier wird auch schon, bevor auf „Create“ geklickt wird, das Creation-Fenster asynchron erstellt. Beim Creation-Fenster gibt es nämlich eine große Anzahl an Controls und Daten. Werden diese erst geladen, wenn auf „Create“ geklickt wird, dann würde es zur sogenannten „Frozen-UI“ kommen. Das Problem daran ist, dass User, falls sich in der GUI nichts mehr bewegt, oft im ganzen Fenster umherklicken, was oft zu neuen, noch größeren Problemen führt.

Nachdem der Benutzer die Informationen zum Installer eingegeben hat, könnte das Formular folgendermaßen aussehen.



Installer 3

Installer 3

Installation Path

D:\Schule\Diplomarbeit\Package-Installers\

Author

Florian Redinger

Description

Hier wird die Beschreibung des Installers eingefügt

Create

Abb. 41: Beispiel-Eingabe eines Installers

Anhand dieser Grafik erkennt man, dass beim Package-Installer Creator auf ein leicht verständliches und ansprechendes Design geachtet wurde.

Wenn der Benutzer hier auf Create klickt, wird sofort die Ordner-Struktur asynchron mit den notwendigen Ordnern erstellt. Ebenfalls werden die restlichen notwendigen Daten für das Creation-Fenster geladen, welches daraufhin angezeigt wird. Dabei ist zu erwähnen, dass es durch die asynchrone Erstellung der Ordner-Struktur möglich ist, während der Erstellung schon am Installer selbst zu arbeiten.

Folgende Struktur wird erstellt:

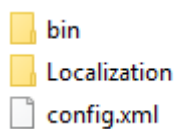


Abb. 42: Die Ordner-Struktur, welche nach dem Klick auf „Create“ erstellt wird

Im Code sieht diese Erstellung folgendermaßen aus:

```
private void GetFolderStructure()
{
    string installerPath;
    string targetPath;

    if (Directory.Exists(creationModel.InstallPath) && creationModel.InstallPath.Length > 0) installerPath = creationModel.InstallPath;
    else
    {
        installerPath = Directory.GetParent(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)).FullName;
        if (Environment.OSVersion.Version.Major >= 6) installerPath = Directory.GetParent(installerPath).ToString();
    }

    targetPath = Path.Combine(installerPath, creationModel.Name);

    if (!Directory.Exists(targetPath)) Directory.CreateDirectory(targetPath);

    creationModel.InstallPath = targetPath;
    config.InstallPath = targetPath;

    Directory.CreateDirectory(Path.Combine(targetPath, "bin"));
    Directory.CreateDirectory(Path.Combine(targetPath, "Localization"));

    config.AddLogEntry("folder structure successfully created");
}
```

Abb. 43: Methode zur Erstellung der Ordner-Struktur

Wurde kein gültiger Installationspfad gesetzt, so wählt das Programm automatisch das Benutzerverzeichnis aus. Danach werden der „bin“ und „Localization“ Ordner erstellt.

Das bei Button-Click geladene Creation-Fenster kann man Anhang B entnehmen.

3.3.5 Existierendes Projekt bearbeiten

Oftmals ergibt sich der Wunsch, bereits vorhandene Installer zu verändern bzw. zu erweitern, ohne diese jedoch wieder vollkommen neu anlegen zu müssen. Da eine derartige Flexibilität mit dieser Software nun endlich gewährleistet wird, kann der Benutzer entweder ein Installer-Projekt (durch Auswahl der XML-Datei) oder kürzlich bearbeitete Installer öffnen. Das ausgewählte Projekt und dessen Informationen sowie Komponenten und Controls werden dann mit einem „XML Deserializer“ ausgelesen und aufgebaut und im „Creation“-Fenster dargestellt. Sollte ein Projekt fehlerhafte Informationen enthalten, bzw. sollten fehlende Dateien vorliegen, so wird eine entsprechende Fehlermeldung an den Benutzer ausgegeben und das Projekt vom PC entfernt.

3.3.6 Existierendes Projekt löschen

Will der Benutzer eines seiner Projekte löschen, so muss er dies nicht zwingend durch manuelles Löschen im Explorer machen, sondern kann ganz einfach auch die dazu bereitgestellte Funktion des Creators nutzen. Dabei steht dem Benutzer für jedes Projekt ein „Löschen“-Icon (siehe Abb. ...) zur Verfügung, das bei Betätigung den gesamten Projekt-Ordner inklusive aller Dateien vom PC entfernt. Dieses Löschen wird in einem rekursiven Prozess abgehandelt, welcher jede Datei jedes Ordners löscht, bevor die Verzeichnisse selbst entfernt werden können.

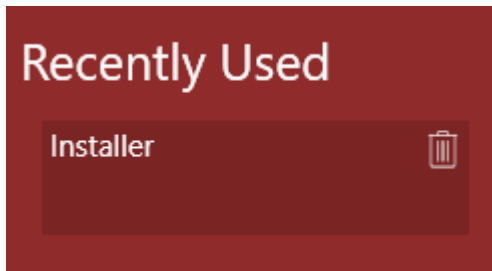


Abb. 44: „Löschen“-Icon

3.3.7 Komponente erstellen

Auf dem in Anhang B gezeigten Fenster werden alle Komponenten erstellt, verwaltet und bearbeitet.

Bei diesem Bestandteil wird es dem Benutzer ermöglicht, für einen neuen Installer eine neue Komponente hinzuzufügen. Wenn der Benutzer auf das Kreuz in Abb. 46 klickt, erscheint ein Dialog (Abb. 47), welcher es möglich macht, einen Namen für die Komponente, den Typ und, falls nötig, ein Template zu wählen. Templates sind im Vorhinein abgespeicherte Komponenten-Prototypen, die man als Ausgangsbasis für neue verwenden kann. Solche Vorlagen werden in externen Dateien gespeichert. Damit wird der Arbeitsaufwand bei der Erstellung vieler ähnlicher Komponenten verringert.

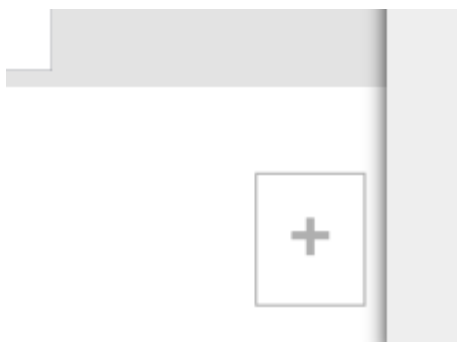


Abb. 46: Button zur Erstellung einer neuen Komponente

Abb. 45: Dialog zur Erstellung einer neuen Komponente

Wird im Dialog (Abb. 47) auf „ADD“ geklickt, werden eine neue Komponente und die dazugehörigen Properties generiert (Abb. 41). Aus dem XML-Schema werden automatisch die verpflichtenden Controls („MandatoryControls“) des ausgewählten Komponenten-Typs ausgelesen und eingefügt. Zum technischen Hintergrund dieses Vorgangs ist auf die Kapitel 3.2.1 „Das Schema bei unserer Diplomarbeit“ und 3.3.11 „Komponenten-Typen“ zu verweisen. Der Benutzer hat die Option, diese Properties zu bearbeiten. Weiters wird eine Kleinansicht der Komponente generiert, in welcher man zwischen den einzelnen Komponenten hin- und herschalten, um diese zu bearbeiten.

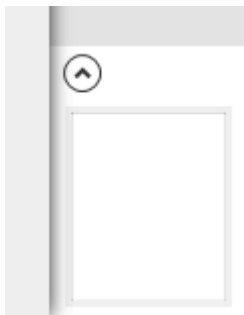


Abb. 47: Standard-Miniatursicht einer Komponente

Abb. 48: Bearbeitbare Properties der Komponente

Nun ist es für den Benutzer möglich, einen Pfad zur Binary der Komponente (Component Path) auszuwählen. Ferner kann der Benutzer/die Benutzerin weitere Controls hinzufügen. Diese Controls sind TextControl, PathControl, NumberControl, BoolControl, DateControl und TnsNamesControl. Weiters hat der Benutzer die Möglichkeit, eigene Controls zu erstellen, indem er/sie auf den +-Button klickt (Abb. 49).

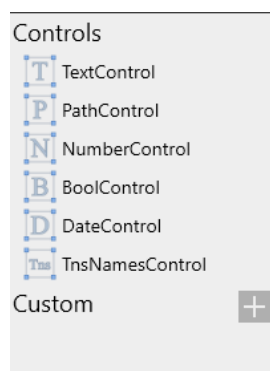


Abb. 49: Alle festgelegten DefaultControls

Der Benutzer kann bei diesen Controls eine Vorschau der Controls erhalten, indem er/sie mit der Maus für längere Zeit über das Listitem fährt.

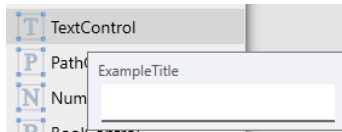


Abb. 50: Exemplarische Vorschau eines Controls

Dies ist natürlich auch bei benutzerdefinierten Controls möglich. Bei diesen werden auch die Defaultwerte angezeigt.

3.3.8 Benutzerdefiniertes Control

Es kann natürlich sein, dass es bei mehreren Komponenten eine Gruppe von Controls gibt, die überall gleich sind. Ein Beispiel dafür ist ein Login-Control, welches ein Benutzer-Feld und ein Passwort-Feld beinhaltet. Die Möglichkeit, ein solches selbst zu erstellen, wäre also vorteilhaft.

Um dies zu tun, muss man auf den +-Button unter den Controls klicken (siehe Anhang B). Nach dem Klick erscheint folgendes Fenster.

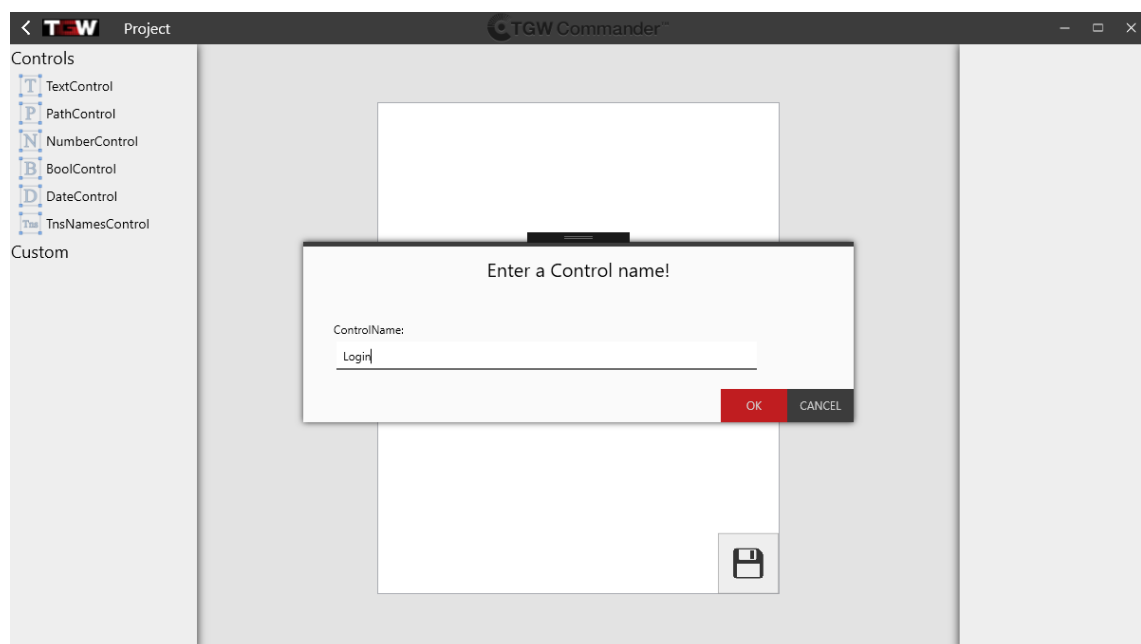


Abb. 51: Custom-Creation-Fenster mit dem Control-Namen „Login“

Bei diesem Fenster ist es möglich, wie in der Abbildung bereits geschehen, einen Namen für das benutzerdefinierte Control einzugeben. In diesem Fall wird das neue Control „Login“ heißen. Bei Klick auf den Button „OK“ wird die Bearbeitung ermöglicht.

Auf den ersten Blick merkt man die Ähnlichkeit zu dem Creation-Fenster. Es fehlt allerdings die Miniaturansicht, aber genauso wie beim Creation-Fenster ist es möglich, Controls hinzuzufügen und dessen Properties zu bearbeiten.

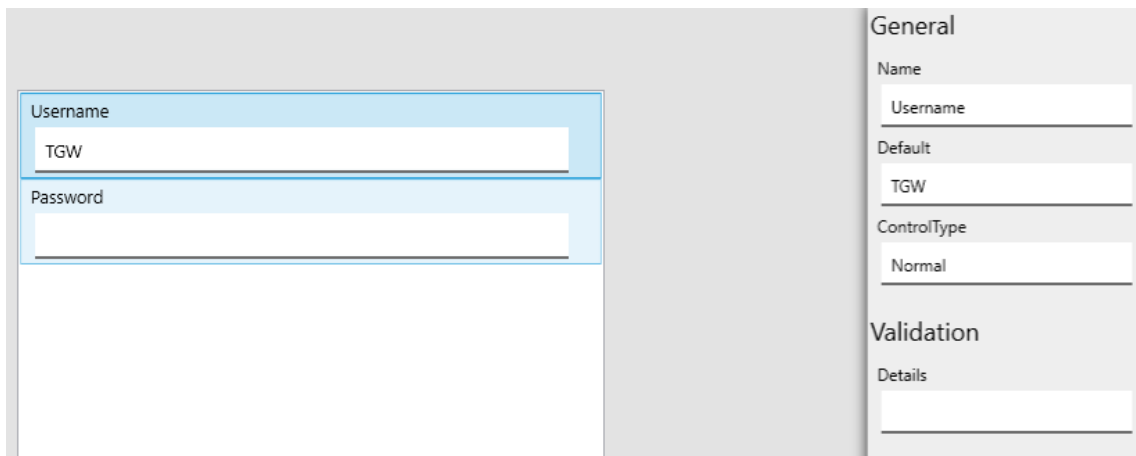


Abb. 52: Bearbeitung der einzelnen Controls eines Custom-Controls

Ist das Control fertig erstellt, kann der Benutzer auf den Speichern-Button (siehe Anhang C) klicken. Dieses neue Control (oder auch Control-Gruppe) wird sofort in einer Liste im Singleton gespeichert und im Creation-Fenster angezeigt.

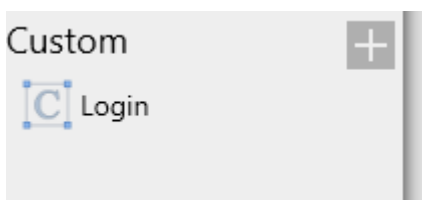


Abb. 53: Neues Control im Creation-Fenster

Genauso wie die DefaultControls hat auch das benutzerdefinierte Control eine Vorschau. Diese hat allerdings schon die definierten Default-Werte, welche im CustomControlCreation-Fenster angegeben werden.

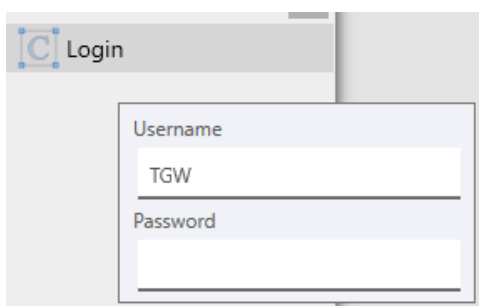


Abb. 54: Vorschau eines neu erstellten Controls

Wird auf dieses ListItem doppelt geklickt, werden beide Controls individuell hinzugefügt, um die weitere Bearbeitung der einzelnen Controls zu ermöglichen.

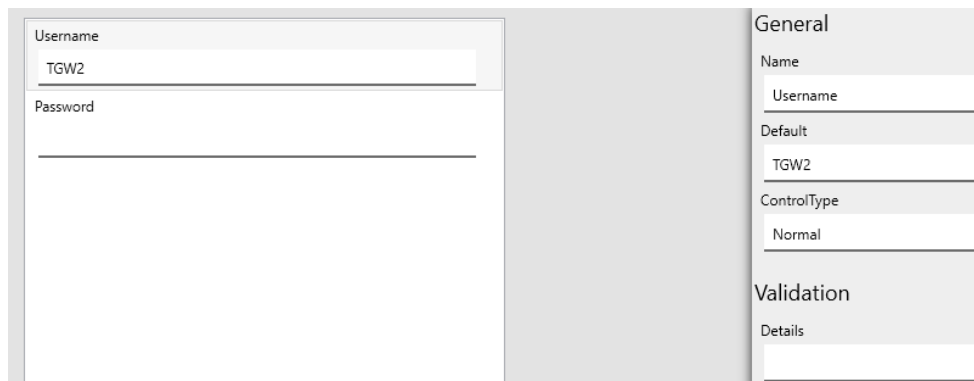


Abb. 55: Hinzufügen eines neu erstellten Controls mit den Default-Werten

3.3.9 Controls hinzufügen & bearbeiten

Durch einen Doppelklick auf ein ListItem wird das Control zur Komponente hinzugefügt und die Bearbeitung ermöglicht. Die Properties der Controls können ebenso wie die Property der Komponente in Echtzeit bearbeitet werden.

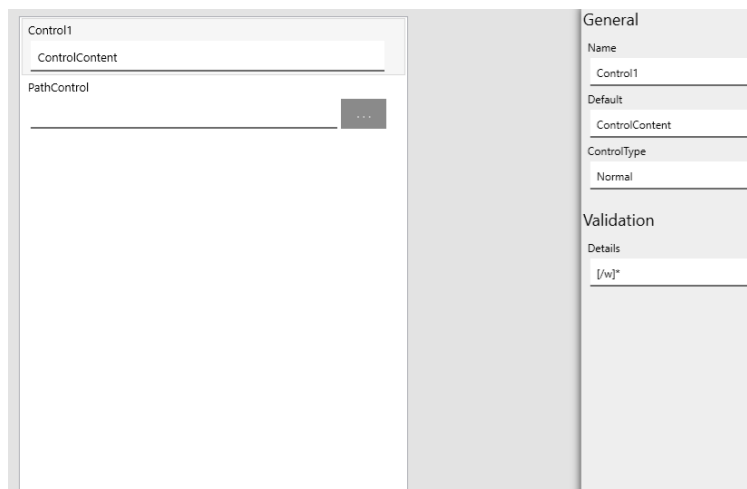


Abb. 56: Komponente mit bearbeitbarem Control

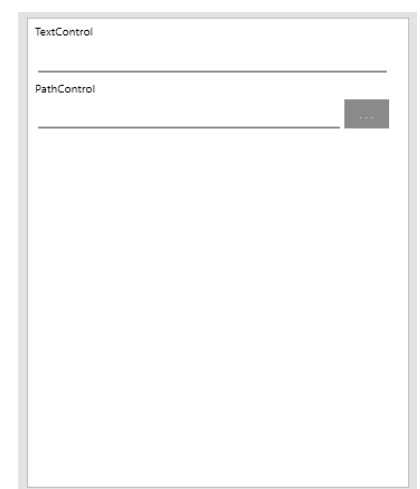


Abb. 57: Komponente mit Controls

Der aktuelle Inhalt der Komponenten kann in der Miniaturansicht gesehen werden. Dies hat den Vorteil, dass ein Ersteller eines Installers bei einer großen Anzahl von Komponenten schnell wieder erstellte Komponenten finden kann.

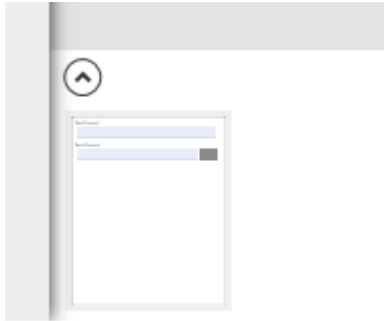


Abb. 58: "Geupdatete" Miniaturansicht der Komponente

Um die Bearbeitung in Echtzeit zu erreichen, kann nicht mit einfachem Binding gearbeitet werden, sondern mit einer Kombination von Reflection, Events und Attributen. Natürlich haben nicht alle Controls dieselben Properties, was zur Folge hat, dass es bei den einzelnen Properties notwendig ist festzulegen, bei welchen Controls sie angezeigt werden und bei welchen nicht. Dafür verwendet man ein Attribut namens „ExcludeForAttribute“, welches bei Erstellung der Property-Controls mit Reflection abgeprüft wird. Mit diesem Attribut kann man ein Array mit Control Namen festlegen. Somit werden die Properties, welche dieses Attribut haben, nicht für bestimmte Controls dargestellt.

Durch ein weiteres Attribut wird festgelegt, welche Art von Control (z. B. TextControl, PathControl, NumberControl) angezeigt wird (sog. `UiElementAttribute`). Hier wird auf eine sehr leichte Verwendung geachtet, weshalb acht Konstruktoren erstellt werden.

```

28 Verweise | Florian Redinger, vor 44 Tagen | 1 Autor, 1 Änderung
public class UiElementAttribute : System.Attribute
{
    14 Verweise | Florian Redinger, vor 44 Tagen | 1 Autor, 1 Änderung
    public enum PropertyArea
    {
        General, Validation
    }

    6 Verweise | Florian Redinger, vor 44 Tagen | 1 Autor, 1 Änderung
    public bool Hide { get; private set; }
    2 Verweise | Florian Redinger, vor 44 Tagen | 1 Autor, 1 Änderung
    public string PropertyName { get; private set; }
    6 Verweise | Florian Redinger, vor 44 Tagen | 1 Autor, 1 Änderung
    public string ControlType { get; private set; }
    6 Verweise | Florian Redinger, vor 44 Tagen | 1 Autor, 1 Änderung
    public PropertyArea Area { get; private set; }
    0 Verweise | Florian Redinger, vor 44 Tagen | 1 Autor, 1 Änderung
    public UiElementAttribute([CallerMemberName] string propertyName = "")
    {
        SetDefaults(propertyName);
    }
}

```

Abb. 59: Properties und Default-Konstruktor des `UiElementAttributes`

Dieser Vorgang kann für mehrere Komponenten wiederholt werden und abschließend kann der Benutzer über die Taste F5 oder über die Menüpunkte „Projekt“ → „Build“ den Installer „builden“. Das heißt, dass alle benötigten Binaries, die benötigten DLL-Dateien und die .exe-Datei zur Ausführung des Installers in der zuvor selektierten Ordner-Struktur erstellt werden.

Project	
Build...	F5
Save	CTRL+S
Load Components	
Load Controls	
Localization	
Exit	CTRL+Q

Abb. 60: Das Menü der Shell

3.3.10 „Builden“ des Installers

Ist die Arbeit am Installer beendet und der Benutzer mit all seinen Einstellungen zufrieden, so kann er nun das Projekt zu einem funktionstüchtigen Installer „builden“ lassen, der später zur zweckmäßigen Benutzung freisteht. Dieser Prozess kann beliebig oft wiederholt werden, da beim erneuten „Builden“ des Projektes alte Dateien gegen neue ersetzt werden.

Der gesamte Prozess des „Buildens“ wird dabei in einem Dialog-Fenster verarbeitet und der Fortschritt dem Benutzer anhand einer „Progress Bar“ dargestellt. Der aktuelle „Build“-Fortschritt wird mit einer Textbox dargestellt. Der Prozess kann jederzeit abgebrochen werden, da bereits erstellte Dateien beim nächsten „Build“-Vorgang einfach überschrieben werden. Der vollständige Prozess des „Buildens“ wird mit Hilfe eines sogenannten „BackgroundWorkers“ asynchron abgearbeitet. Dieser wird vor allem für die laufende Aktualisierung der „Progress Bar“ genutzt. Der „BackgroundWorker“ ändert dabei Properties, welche an die Property „Value“ der „Progress Bar“ gebunden ist und dadurch diese stetig aktualisiert:

```

private string _output;

5 Verweise | Florian Peter, vor 34 Tagen | 1 Autor, 1 Änderung
public string Output
{
    get { return _output; }
    set {
        _output = value;
        OnPropertyChanged();
    }
}

private int _progressPercentage;

2 Verweise | Florian Peter, vor 34 Tagen | 1 Autor, 1 Änderung
public int ProgressPercentage
{
    get { return _progressPercentage; }
    set {
        _progressPercentage = value;
        OnPropertyChanged();
    }
}

```

Abb. 61: Überwachte Properties des Build-Dialogs

So wird der BackgroundWorker im Code initialisiert und gestartet:

```

_worker = new BackgroundWorker()
{
    WorkerReportsProgress = true,
    WorkerSupportsCancellation = true
};
_worker.DoWork += worker_DoWork;
_worker.ProgressChanged += worker_ProgressChanged;
_worker.RunWorkerCompleted += worker_RunWorkerCompleted;

```

Abb. 62: Definition des BackgroundWorkers

Mit der StartProcess-Methode beginnt der BackgroundWorker-Prozess:

```

public void StartProcess()
{
    Output = "";
    if (!_worker.IsBusy)
    {
        _worker.RunWorkerAsync();
    }
    CancelEnabled = _worker.IsBusy;
    FinishedEnabled = !_worker.IsBusy;
}

```

Abb. 63: Methode, welche sich um den Start des BackgroundWorkers kümmert

3.3.10.1 Erzeugen und kopieren der benötigten Dateien

Damit der fertige Installer voll lauffähig ist, müssen einige Dateien, die von Projekt zu Projekt vollkommen einzigartig sind, erzeugt werden. Beispielsweise muss die XML-Datei auf Basis der erstellten Komponenten und derer Eigenschaften generiert werden. Dies geschieht mit dem eigens dafür entwickelten „XML-Serializer“. Weiters muss eine CSV-Datei für die lokalisierten Felder, im richtigen (genormten) Format, aufgebaut werden, damit diese später von der Runtime problemlos übernommen werden kann. Dieser Schritt wird im „Build“-Prozess jedoch übersprungen, sollte der Benutzer schon zuvor im Lokalisierungs-Fenster die Datei generiert haben lassen, und sollten seitdem keine weiteren Änderungen aufgetreten sind.

Neben neu erzeugten Dateien gibt es einige, die nicht vom Projekt selbst abhängig sind und problemlos in den Installer-Ordner kopiert werden können. So geschieht es mit angegebenen Binary-Dateien, welche für jede Komponente obligatorisch anzugeben sind. Diese werden, sofern sie nicht als „External“ (External = müssen nicht in den Projekt-Ordner integriert werden) gekennzeichnet sind, in das „bin“-Verzeichnis für die Komponenten-Binaries kopiert.

Zusätzlich greift jede Runtime auf dieselben Libraries und Dateien (.exe, .dll) zu. Deshalb werden diese einmalig kompilierten Dateien (werden beim „Build“-Vorgang des Runtime-Projekts erzeugt) in ein ZIP-Archiv dem Creator zur Verfügung gestellt. Dieser kopiert dessen Inhalt in den Projekt-Ordner des Installers und entpackt das Archiv. Somit muss der Benutzer nur noch die „PackageInstaller.exe“-Datei ausführen, welche wiederum den erstellten Package-Installer auf Basis der XML-Datei ausführt.

Dieser gesamte Prozess wird vom BackgroundWorker in der „DoWork“-Methode abgearbeitet. In diesem Fall wird dort die Methode „Build“ aufgerufen, welche alle Dateien kopiert und erzeugt.

Soll die Progress-Bar nun Fortschritt anzeigen, bzw. soll ein Informationstext im „Build“-Dialog angezeigt werden, so müssen nun die beiden bereits erwähnten Properties geändert werden. Dies geschieht durch Aufruf folgender Methode:

```
private void worker_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    ProgressPercentage = e.ProgressPercentage;
    Output = (string)e.UserState;
}
```

Abb. 64: Methode, welche sich um das Updaten der UI kümmert

So wird dem BackgroundWorker mitgeteilt, dass es einen Fortschritt gab:

```
_worker.ReportProgress(0, "Installation initialized");
```

Abb. 65: Aufruf der in Abb. 65 definierten Methode

3.3.11 Komponenten-Typen

Jede Komponente besitzt einen bestimmten Typ, von dem aus auch auf die dazugehörigen obligatorischen Controls und der Art der Binary geschlossen werden kann. Insgesamt werden fünf Arten genutzt, die im täglichen Geschäftsablauf der TGW und derer Kunden üblich, wenn nicht sogar nötig sind. Eine dynamische Verwaltung dieser ist nicht vorgesehen.

3.3.11.1 MSI

Komponenten vom Typ „MSI“ basieren auf MSI-Binaries, welche für die Installation obligatorisch sind. Dabei können Controls wie „Datei-Pfad“, Installations-Typ („Manuell“, „Nicht-Manuell“) und viele weitere (oftmals individuell abhängig von der verwendeten MSI-Datei) verwendet werden.

3.3.11.2 Trade-Aware MSI

Die Trade-Aware-MSI-Komponente ist eine Erweiterung der oben genannten MSI. Diese besitzt die Eigenschaft, einen Trade Server zum Hochladen der MSI anzugeben, sowie dessen „Branding“ zu bestimmen. Die Möglichkeit zur Nutzung eines derartigen Servers muss nicht in Anspruch genommen werden, sie ist also optional, die Komponente wird dennoch in jedem Fall als „Trade-Aware“ im Installations-Prozess der Runtime verarbeitet.

3.3.11.3 Oracle & MS-SQL Datenbank

Manche Komponenten der TGW Commander Software benötigen eine Datenbank. Diese Datenbanken werden in eigenen Installationspaketen bereitgestellt, da diese unabhängig von der zugehörigen .NET Komponente entweder in eine Oracle oder MS-SQL Datenbank installiert werden. Die Installationspakete sind komprimierte Archive, welche Scripts und Dateien enthalten, die vom jeweiligen Kommandozeilen-Interface (SQL Plus, sqlcmd) benötigt werden.

Im Kontext des Installers werden die Archive entpackt und das Kommandozeilen-Interface mit den entsprechenden Parametern aufgerufen.

3.3.11.4 TGW Deployment Package

Die Komponenten der TGW Commander Software sind einerseits durch XML-Konfigurationsdateien parametrierbar sowie andererseits durch .NET Assemblies kundenspezifisch anpass- und erweiterbar. Die XML-Konfigurationsdateien bzw. die .NET Assemblies müssen nach der Basisinstallation der Software-Komponente in mehrere definierte Verzeichnisse kopiert werden, damit die Komponenten diese laden können. Um diesen Prozess zu vereinfachen, werden die Dateien in der vordefinierten Verzeichnisstruktur in sogenannte TGW Deployment Packages (Dateiendung *.tgwdp) komprimiert zur Verfügung gestellt. Diese Archive werden dann durch das sogenannte TGW Deployment Center auf der Zielformaschine entpackt und in die Dateien in die jeweiligen Verzeichnisse kopiert. Das TGW Deployment Center kann nun entweder manuell (durch Doppelklick auf die *.tgwdp Datei) oder "silent" über ein Kommandozeilen-Interface aufgerufen werden.

Im Kontext des Installers wird das Kommandozeilen-Interface verwendet und das Deployment eines TGW Deployment Package kann so in den Installationsprozess integriert werden.

Die beiden letzten Komponenten, MS-SQL Datenbank und TGW Deployment Package, wurden dem Creator noch nicht als fertige Klassen zur Verfügung gestellt. Dies kann jedoch dank der simplen Erweiterbarkeit der Software ohne größeren Aufwand umgesetzt werden.

3.3.12 Lokalisierung

Der Auftraggeber ist ein internationales Unternehmen. Um diesem Umstand gerecht zu werden, müssen besondere Vorkehrungen getroffen werden, damit die erstellten Programme auch in aller Welt einsetzbar sind. Hier kommt das Schlagwort „Lokalisierung“ ins Spiel. Darunter versteht man, dass sich ein Programm automatisch an die Sprache des Benutzers anpasst. Es werden für jedes Wort Übersetzungen in mehreren Sprachen mitgegeben und die aktuelle Systemsprache wird automatisch ausgelesen. Da sich der Endanwender nicht lange mit den Programmeinstellungen herumschlagen muss, um zur gewünschten Anzeigesprache zu kommen, ist dies sehr benutzerfreundlich.

Der Entwickler des Installer-Packages kann selbst, mit Hilfe der eingebauten Lokalisierungs-Funktion, die Übersetzungen verwalten. Dabei wird er über den dazugehörigen Menüpunkt auf ein neues Fenster weitergeleitet, in dem alle zur Verfügung stehenden Felder zur Lokalisierung bereitstehen. Dabei kann in der obersten Leiste (siehe Abb. ...) zwischen den Komponenten gewechselt

werden. „Miscellaneous“ ist dabei aber keine Komponente, sondern eine Sammlung allgemeiner Felder, die nicht zugeordnet werden können bzw. sollen, wie z.B. der Name des Installers.

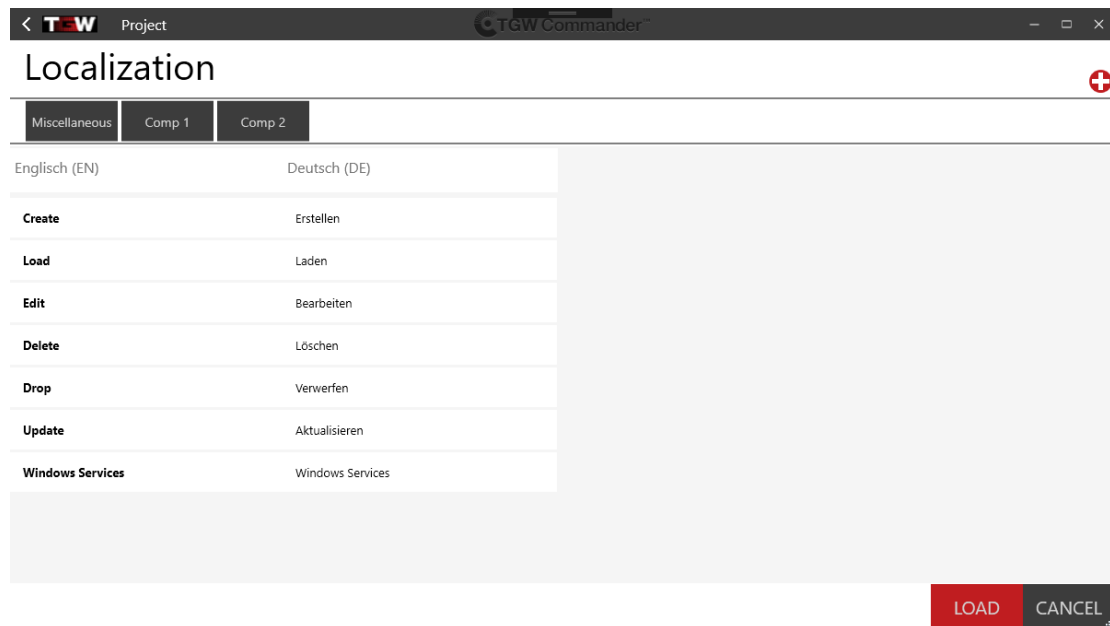


Abb. 66: Lokalisierungs-Fenster

3.3.12.1 Sprachen hinzufügen

Bei jedem neuen Projekt sind die Sprachen „Deutsch“ (de) und „Englisch“ (en) standardmäßig vorhanden, jedoch reicht dies oft nicht aus, bzw. wünscht man sich für gewisse Zwecke eine weitere Sprache. Dies kann durch das dazugehörige „Hinzufügen“-Icon (siehe Abb. ...) problemlos umgesetzt werden. Es erscheint dabei ein Dialog mit einer Combo-Box (siehe Abb. ...), welche mit allen auf dem PC vorhandenen Sprachen aufgefüllt wird, um den vollen Umfang zu gewährleisten. Hat man sich für eine Sprache entschieden, wird diese als weitere Spalte in den Tabellen für alle Komponenten hinzugefügt (siehe Abb. ...) und die betroffenen Werte werden in der Datenbank aktualisiert.

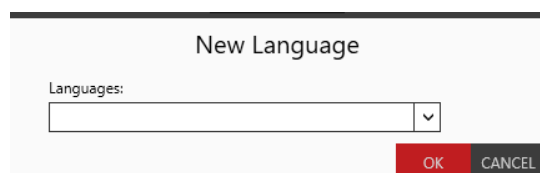


Abb. 67: Dialog zum Hinzufügen einer neuen Sprache

3.3.13 Lokalisierung verwalten

Wie bereits erwähnt, wird die Lokalisierung aller Felder in einem Lokalisations-Fenster verwaltet. Die dort eingetragenen Werte und Sprachen müssen jedoch irgendwo zwischengespeichert werden, bevor daraus eine fehlerfreie CSV-Datei aufgebaut werden kann. Dies wird durch eine eigens dafür erstellte Datenbank abgewickelt, welche alle Werte für alle Sprachen und alle Komponenten und Controls sowie Sonstigem speichert. Wird ein Installer geladen, muss jedoch zuerst geklärt werden, welche Felder zur Lokalisierung bereitstehen. Daher werden in einem asynchronen Verfahren alle lokalisierbaren Felder gefiltert und dessen Werte Schritt für Schritt in der Datenbank eingetragen, bevor daraus die für den Benutzer sichtbaren Tabellen aufgebaut werden.

Mit diesen Tabellen können dann die Lokalisierungs-Werte für jede Sprache eingetragen werden. Nach Beendigung erfolgt eine Sicherung alle Änderungen in der Datenbank. Diese werden beim nächsten Bedarf wieder geladen.

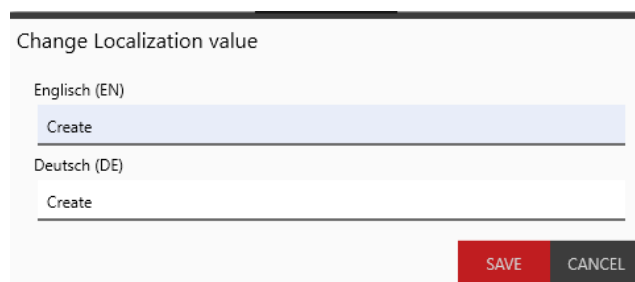


Abb. 68: Dialog zur Bearbeitung der Daten (erscheint durch Klick auf ein Lokalisierungsfeld)

Arbeitet der Benutzer nun an einem neuen Projekt, so wird jedes Mal, wenn eine neue Komponente oder Control hinzugefügt wird, ein neuer Eintrag in der Datenbank gespeichert.

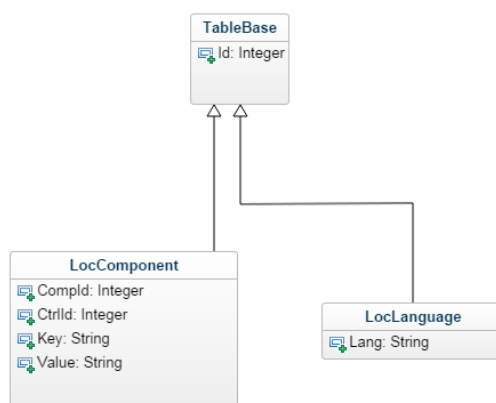


Abb. 69: Abbildung der Lokalisierungs-Datenbank des Creators

3.3.13.1 Lokalisierungs-Datei generieren

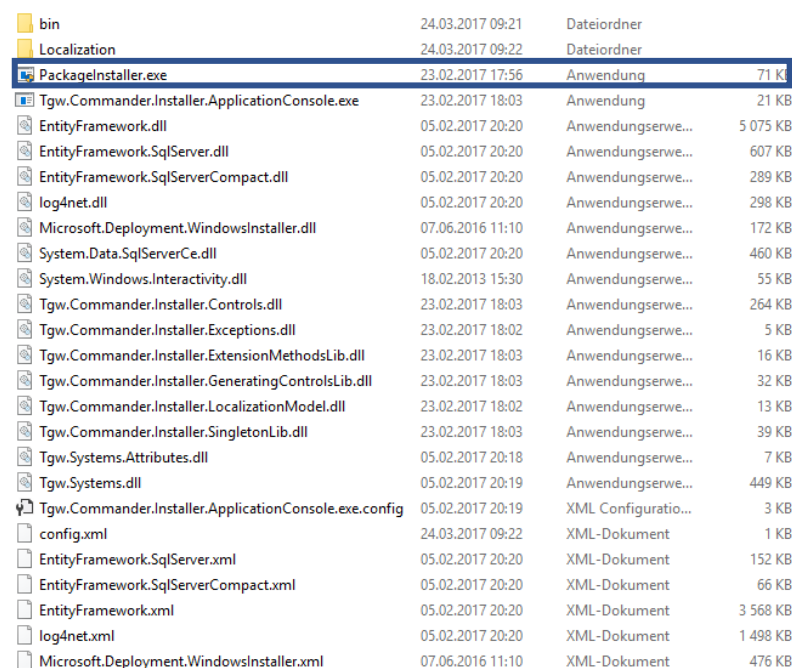
Jede Runtime soll für jede Sprache benutzbar sein. Deshalb muss die Software auf Lokalisierungs-Werte für all ihre Felder zurückgreifen können. Da jedoch nicht etwa eine exportierte Kopie der Creator Lokalisierungs-Datenbank verwendet wird, nutzt die Runtime eine nach festen Schema aufgebaute CSV-Datei. Diese wird beim Bauen des Installers oder durch Betätigung des „Build CSV“-Buttons im Lokalisierungs-Fenster in einem asynchronen Verfahren aufgebaut, basierend auf den aktuellen Werten der Datenbank.

3.4 Package-Installer Runtime

Zur besseren Übersicht lassen sich sämtliche Bilder zu den UI-Oberflächen des Runtime im Anhang D-I finden.

3.4.1 Startup

Wenn nun der Installer vom Creator „gebildet“ wird, ist es möglich, diesen per Klick auf die Exe-Datei zu öffnen:



bin	24.03.2017 09:21	Dateiordner	
Localization	24.03.2017 09:22	Dateiordner	
PackageInstaller.exe	23.02.2017 17:56	Anwendung	71 KB
Tgw.Commander.Installer.ApplicationConsole.exe	23.02.2017 18:03	Anwendung	21 KB
EntityFramework.dll	05.02.2017 20:20	Anwendungserwe...	5 075 KB
EntityFramework.SqlServer.dll	05.02.2017 20:20	Anwendungserwe...	607 KB
EntityFramework.SqlServerCompact.dll	05.02.2017 20:20	Anwendungserwe...	289 KB
log4net.dll	05.02.2017 20:20	Anwendungserwe...	298 KB
Microsoft.Deployment.WindowsInstaller.dll	07.06.2016 11:10	Anwendungserwe...	172 KB
System.Data.SqlServerCe.dll	05.02.2017 20:20	Anwendungserwe...	460 KB
System.Windows.Interactivity.dll	18.02.2013 15:30	Anwendungserwe...	55 KB
Tgw.Commander.Installer.Controls.dll	23.02.2017 18:03	Anwendungserwe...	264 KB
Tgw.Commander.Installer.Exceptions.dll	23.02.2017 18:02	Anwendungserwe...	5 KB
Tgw.Commander.Installer.ExtensionMethodsLib.dll	23.02.2017 18:03	Anwendungserwe...	16 KB
Tgw.Commander.Installer.GeneratingControlsLib.dll	23.02.2017 18:03	Anwendungserwe...	32 KB
Tgw.Commander.Installer.LocalizationModel.dll	23.02.2017 18:02	Anwendungserwe...	13 KB
Tgw.Commander.Installer.SingletonLib.dll	23.02.2017 18:03	Anwendungserwe...	39 KB
Tgw.Systems.Attributes.dll	05.02.2017 20:18	Anwendungserwe...	7 KB
Tgw.Systems.dll	05.02.2017 20:19	Anwendungserwe...	449 KB
Tgw.Commander.Installer.ApplicationConsole.exe.config	05.02.2017 20:19	XML Configuratio...	3 KB
config.xml	24.03.2017 09:22	XML-Dokument	1 KB
EntityFramework.SqlServer.xml	05.02.2017 20:20	XML-Dokument	152 KB
EntityFramework.SqlServerCompact.xml	05.02.2017 20:20	XML-Dokument	66 KB
EntityFramework.xml	05.02.2017 20:20	XML-Dokument	3 568 KB
log4net.xml	05.02.2017 20:20	XML-Dokument	1 498 KB
Microsoft.Deployment.WindowsInstaller.xml	07.06.2016 11:10	XML-Dokument	476 KB

Abb. 70: Ordner-Struktur nach dem Build-Prozess

Wird die Exe-Datei mit einem Doppelklick geöffnet, erscheint das Bild von der Startup-Page (siehe Anhang D). Es ist allerdings auch möglich, den Installer per Konsole zu starten. Um eine vollständige Konsoleninstallation zu ermöglichen, ist es notwendig, Parameter anzugeben. Es kann entweder, mit „-h“, „-help“ oder „-?“, eine Hilfe zur Installation angezeigt werden, falls der Inbetriebnehmende nicht darüber informiert wird, wie die Kommandos zur Installation von Komponenten lauten. Die möglichen Parameter sind:

Tabelle 7: Parameter der Konsolenbedienung

Kommando	Beschreibung
-P/-p	Prefix für jeden Oracle User
-I/-i	Datenbank Instanz
-U/-u-	Datenbank-Schemas werden nur aktualisiert
-L/-l	Pfad zu den Installer Binaries

Für die Installation von Komponenten oder Gruppen gibt es auch noch folgende Parameter:

- -Components [Component Name] [...] → installiere Komponenten
- -Group [Group Name] [...] → installiere Komponenten Gruppen

```

file:///C:/Tgw/TGW-Installer/Main/Products/PackageInstallerRuntime/Source/ApplicationConsole/bin/Debug/Tgw.Commander.Installer.ApplicationCo...
SETUP [-p <prefix> -i <instance> -u -d -l <path>]

Database Options:
GLOBAL  LOCAL
-P      -p          : Prefix for each oracle user
-I      -i          : Database instance
-U      -u          : Database schemas will be updated only
-D      -d          : Existing database schemas will be dropped
-L      -l          : Location of the install binaries

Installation Options:
-Components [Component Name] [...] : Install Components
-Groups [Group Name] [...]         : Install Component Groups

Others:
-?                                  : Show usage
  
```

Abb. 71: Runtime als Konsolenprogramm (wenn PackageInstaller.exe mit Parameter „-h“ aufgerufen wird)

Werden keine Parameter angegeben, so wird ebenfalls das Bild von der Startup-Page (siehe Anhang D) angezeigt.

Bei dieser „Startup-Page“ werden im Hintergrund alle nötigen Komponenten und Binaries aus der XML-Datei gelesen, außerdem werden alle für die spezifische XML-Datei benötigten Fenster geladen, sowohl die Fenster für die benötigten Komponenten als auch die immer benötigten Fenster (Component-Groups-Page, Custom-Components-Page, Summary-Page, Installation-Page). All das wird auf diesem Fenster vorgeladen, um dafür während der Ausführung weniger Zeit zu benötigen und den Installer sehr viel benutzerfreundlicher bedienen zu können.

Um dies zu ermöglichen und dabei die UI zu aktualisieren, wird ein Background-Worker eingesetzt.

3.4.2 Komponenten-Gruppen

Oft werden bestimmte Komponenten immer wieder gemeinsam, sozusagen in Paketen, installiert. Diese Pakete werden als Komponenten-Gruppen bezeichnet (diese wurden zuvor im Creator definiert). Diese Komponenten-Gruppen erleichtern dem Benutzer die Auswahl der benötigten Komponenten erheblich.

Hierfür werden am zweiten Fenster der Applikation alle in der XML-Datei definierten Gruppen mit deren Überbegriff sowie Beschreibung dargestellt. Wählt der Benutzer eine der Gruppen, werden im Hintergrund alle dazugehörigen Komponenten für die nächsten Fenster ausgewählt.

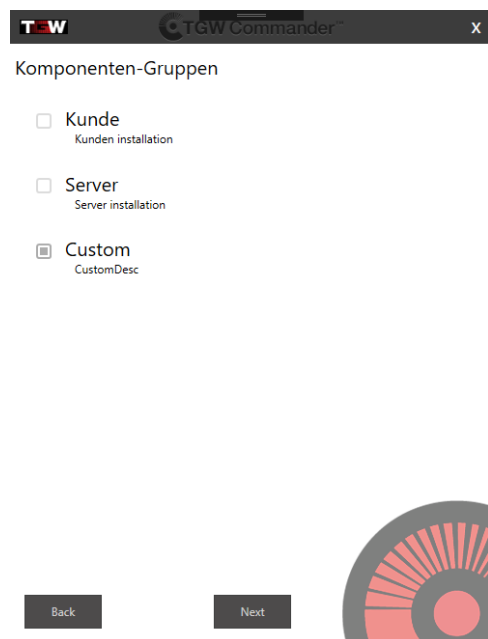


Abb. 72: Die Komponentengruppen der Runtime (exemplarisches Bild)

Um die ausgewählte Komponenten-Gruppe vollständig zu installieren, müssen alle betreffenden Komponenten konfiguriert werden. Nach Auswahl der Gruppe steht für jede ihrer Bestandteile ein eigenes Fenster mit angepassten Controls zur Verfügung, mit denen der Benutzer die Installations-Parameter nach Belieben bearbeiten kann. Alle Controls sind obligatorisch zu bearbeiten. Sind alle Komponenten bearbeitet, werden diese installationsfähig umgebaut und mit Hilfe eines Installations-Frameworks installiert.

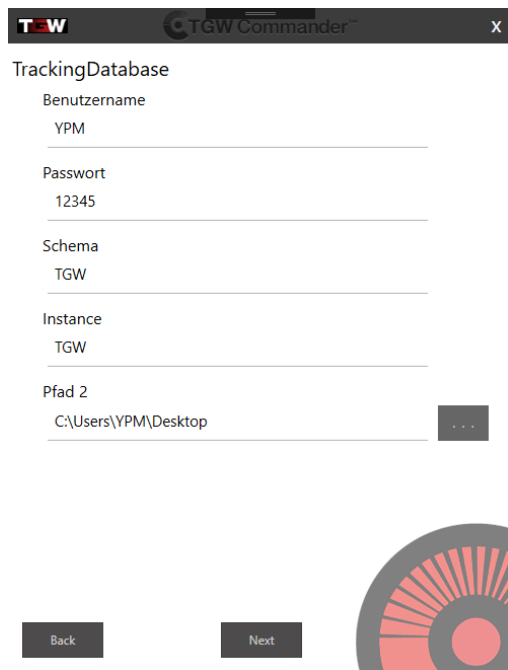


Abb. 73: Exemplarisches Fenster einer Komponente

Neben vorgefertigten Paketen kann der Benutzer auch individuelle Pakete auswählen, um ein „benutzerdefiniertes“ Paket zusammenzustellen. Dies wird durch die Gruppe „Benutzerdefiniert“ ermöglicht.

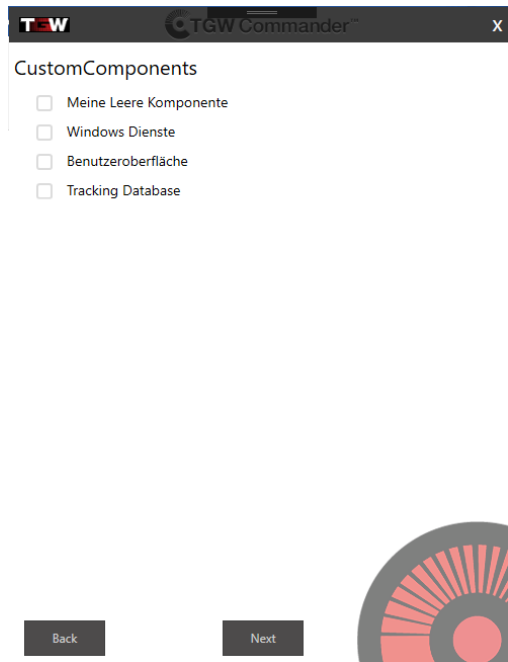


Abb. 74: Auflistung aller verfügbaren Komponenten für die benutzerdefinierte Installation

3.4.3 Benutzerdefinierte Auswahl

Hat sich der Benutzer für eine eigene Auswahl der Komponenten entschieden, so werden ihm auf dem nächsten Fenster alle vorhandenen Auswahlmöglichkeiten, sogenannte Installations-Komponenten, aufgelistet. Diese können dann ganz einfach per Selektierung der Check-Box ausgewählt werden. Die Auswahl wird dann im Hintergrund für die nächsten Fenster zusammengestellt (lediglich Informationen der Komponenten, die als Fenster erscheinen, sollen bereits im Startup geladen und zusammengestellt werden).

Sollte eine Komponente ausgewählt werden, die in der XML-Datei als Teil einer Abhängigkeits-Gruppe definiert ist, so werden vom Programm automatisch alle davon abhängigen Komponenten selektiert. Deselektiert der Benutzer eine Auswahl, so geschieht dies auch mit allen anderen (abhängigen). Diese automatische Abhängigkeits-Erkennung ist mitunter eine der neuen Erweiterungen, welche durch den Creator ermöglicht und bereitgestellt werden. Im aktuellen Verfahren müssen Benutzer von derartigen Verkettungen Bescheid wissen, was vor allem für Kunden ein schwerwiegendes Problem darstellt.

So werden die Abhängigkeiten im Code ausgelesen:

```
/// <summary>
/// Gets all dependent Components of a component in the DependencyGroup
/// </summary>
/// <param name="compId">The Id of the Component</param>
/// <returns>The Ids of all dependent Components</returns>
1 Verweis | Florian Redinger, vor 9 Tagen | 1 Autor, 3 Änderungen
internal List<int> GetComponentDependency(int compId)
{
    try
    {
        XmlNode depGrp = config.XmlData.DependencyGroups
            .FirstOrDefault(x =>
                x.FindAllInChildrenRecursive("Comp")
                    .FirstOrDefault(y =>
                        y.FindInAttributes("compId").Value == compId.ToString()) != null);

        List<int> compDepSiblings =
            depGrp?.FindInChildren("Comps")
                .FindAllInChildren("Comp")
                .GetAttributesOfChildren("compId")
                .Where(x => x != compId.ToString()).Select(int.Parse)
                .ToList();
        return compDepSiblings;
    }
    catch (InvalidOperationException)
    {
        return null;
    }
}
```

Abb. 75: Herausfinden der Abhängigkeiten einer Komponente

Hierbei werden die Abhängigkeiten einer bestimmten Komponente (Parameter: ID der Komponente) gelesen. Zunächst werden die dazugehörige Abhängigkeitsgruppe geladen, bevor die IDs angehöriger Komponenten in eine Liste gespeichert und zurückgegeben werden.

3.4.4 Komponenten

Bei diesem Fenster werden nun alle Controls, welche in der XML-Datei angegeben werden, dynamisch geladen. Dabei wird anfangs untersucht, welcher Komponententyp gerade geladen wird (BlankComponent, MsiComponent, OracleDbComponent, TradeAwareMsiComponent). Für diesen Typ werden zuerst die obligatorischen Controls geladen. Da dabei besonders auf leichte Erweiterbarkeit geachtet wird, werden die Eigenschaften mit Reflection geladen. Zur Generierung der obligatorischen Controls wird folgender Algorithmus verwendet:

- Zuerst werden mithilfe von Reflection die nötigen Properties für die spezifische Komponente geladen.
- Diese Properties werden dann einem Objekt vom Typen „ComponentInformation“ zugewiesen.
- Mithilfe dieses Objektes werden in der Methode „GetMandatoryControls“ die Properties durchgegangen.
- Für jedes Property wird mit der Methode „GetMatchingInstance“ ein UserControl geladen und einer Liste hinzugefügt, welche in der Components-Page angezeigt wird.

Nachdem die obligatorischen Controls geladen worden sind, erfolgt gleich die Ladung der optionalen Controls. Auch hier wird sehr großer Wert auf die leichte Erweiterbarkeit gesetzt. Es ist beispielsweise möglich, selbstständig eigene Attribute zu dem XML-Element „Control“ hinzuzufügen. Diese Attribute werden ebenfalls ausgelesen und über Reflection dem angegebenen Control zugewiesen, so lange es dieses Property gibt. So kann man zum Beispiel bei dem einen Control mit dem Typen „NumberControl“ das Attribut „Max“ hinzufügen. Da es ein Control mit diesem Property gibt, ist es sofort anwendbar.

Um dies zu ermöglichen, wird folgender Algorithmus angewandt:

- Zuerst werden Kind-Elemente des Elements „OptionalControls“ ausgelesen. Da es von diesem Element sowieso nur eines geben darf, wird nur das „OptionalControls“-Element ausgelesen.
- Um die Identifizierung des notwendigen UserControls für das OptionalControl zu ermöglichen und diesem auch gleich Wert und Name zu geben, werden zuerst die Attribute „Type“, „Name“ und „Val“ ausgelesen.
- Mit Reflection wird das UserControl mit dem Typen „Type“ geladen.
- Die restlichen Attribute werden ebenfalls ausgelesen und, falls es eine Property mit dem Namen des Attributes gibt, wird es mit Reflection dem Property zugewiesen.
- Letztendlich wird das UserControl wieder der Liste hinzugefügt, welche in der ComponentsPage angezeigt wird.

Durch diese Generierung entsteht beispielsweise aus dem in Abb. 78 dargestellten XML-Dateibereich folgendes Bild (Abb. 79):

```
<TradeAwareMsiComponent Id="2">
  <Name valueLocId="13">UserInterface</Name>
  <BinaryPath labelLocId="14">C:\Users\YPM\Desktop</BinaryPath>
  <MandatoryControls labelLocId="15">
    <InstallPath labelLocId="16">C:\Users\YPM\Desktop</InstallPath>
    <Branding labelLocId="17">TGW</Branding>
    <Server labelLocId="18">TGW</Server>
  </MandatoryControls>
  <OptionalControls labelLocId="19">
    <Control Type="TextControl" Name="Password" Val=""
      RegexPattern="^[a-z]*$" ControlType="Password" labelLocId="20" />
  </OptionalControls>
</TradeAwareMsiComponent>
```

Abb. 76: Komponente eines Installers

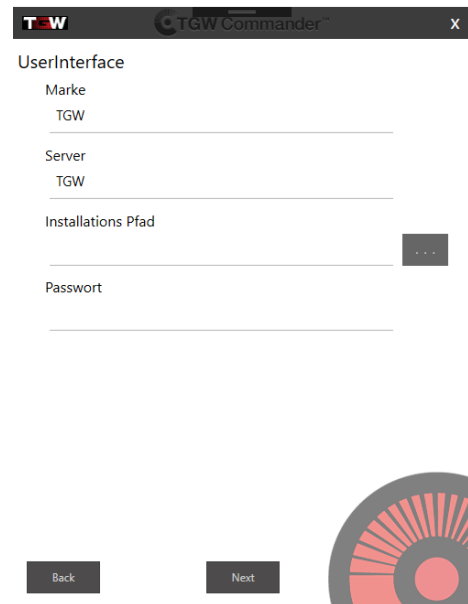


Abb. 77: Erstellte UI aus der, in Abb. 78, definierten Komponente

Die Properties, ob obligatorisch oder optional, werden natürlich alle validiert. Um diese Funktion bereitzustellen, wird es dem Benutzer ermöglicht, ein Regex-Pattern anzugeben, welches bei User-Eingabe überprüft wird. Falls der Inhalt nicht dem Pattern entspricht, wird eine Fehlermeldung ausgegeben.

Wenn man im Feld „Passwort“ einen Inhalt eingibt, welcher nicht dem RegexPattern „^[a-z]*\$“ entspricht, beispielsweise „12345“, so wird eine Fehlermeldung angezeigt. Mit dieser Funktion wird verhindert, dass der Benutzer zur nächsten Komponente wechselt.

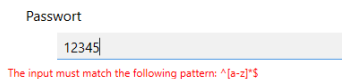


Abb. 78: Error-Message eines Controls

3.4.5 Installation

Die Installation ausgewählter Komponenten, unter Berücksichtigung obligatorischer sowie optionaler Parameter, ist die fundamentale Hauptfunktionalität der Runtime, ohne deren fehlerfreien Verlauf vorhergehende Prozesse keine Auswirkung zeigen würden. Diese Installation wird mit Hilfe eines von der TGW bereits verwendeten Frameworks unterstützt, welches es dem Programm ermöglicht, verschiedene Komponenten-Typen mit deren Parametern zu installieren. Um für alle Typen einen einheitlichen Prozessverlauf zu ermöglichen, müssen Komponenten zuerst in „Build“-fähige Komponenten umgebaut werden, welche dann das Framework übernimmt und entsprechend verarbeitet. Das Runtime Installations-Framework nimmt auch neuere Komponenten-Typen an, welche noch nicht Teil der statischen Software sind.

Um dem Benutzer die Orientierung über den aktuellen Installations-Prozess zu erleichtern, wurde eigens eine „Progress-Bar“ platziert, mit farblicher und prozentueller Kennung des Fortschrittes. Dieser wird nach jedem abgeschlossenen Installations-Schritt aktualisiert, bis der Prozess beendet ist.

Neben der Fortschrittsleiste gibt es auch noch eine Auflistung der zu installierenden Komponenten. Jede einzelne wird zusätzlich mit einem Statussymbol ausgestattet. Dieses ermöglicht es dem Benutzer, jederzeit kontrollieren zu können, welche Komponenten bereits installiert sind, welches aktuell bearbeitet wird und welche fehlgeschlagen sind.

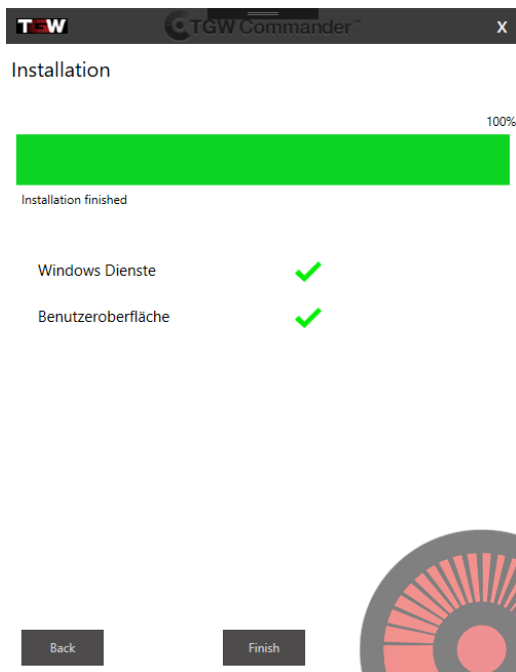


Abb. 79: Installations-Fenster der Runtime

3.4.6 Page-Switcher

Um zwischen den einzelnen Fenstern zu wechseln, wird ein sogenannter „Page-Switcher“ eingebaut. Dieser stellt ein Interface zur Verfügung, welches die Methode „Navigate“ enthält. Dieses Interface ermöglicht es, das Hauptfenster (sog. Shell) zu identifizieren, welches die Methode „Navigate“ definiert. Diese wird aufgerufen, wenn die Methode „Switch“ von der statischen Klasse „Switcher“ aufgerufen wird. Da diese Methode ebenfalls statisch ist (sie ist für alle Klassen in der Projektmappe gleich), kann der Aufruf überall stattfinden. Das hat zur Folge, dass es egal ist, von wo es aufgerufen wird. Es wird entweder auf das nächste, das vorherige oder auf ein genau definiertes Fenster gewechselt. Die Reihenfolge der Pages wird im Singleton „Config“ definiert, welches sich im Laufe des Programms entwickelt. Um immer zum richtigen Fenster zu navigieren, wird die Methode „MakeSwitch“ kreiert.

```
2 Verweise | Florian Redinger, vor 232 Tagen | 1 Autor, 2 Änderungen
public interface IPageSwitcher
{
    2 Verweise | Florian Redinger, vor 232 Tagen | 1 Autor, 2 Änderungen
    void Navigate(Page nextPage, int idxPage);
    2 Verweise | Florian Redinger, vor 232 Tagen | 1 Autor, 2 Änderungen
    void Navigate(Page nextPage, object state, int idxPage);
}
```

Abb. 80: Interface, welches vom Page-Switcher-Fenster implementiert wird

```
2 Verweise | Florian Redinger, vor 232 Tagen | 1 Autor, 2 Änderungen
public static void Switch(Page newPage = null)
{
    MakeSwitch(newPage, 1);
}

0 Verweise | Florian Redinger, vor 232 Tagen | 1 Autor, 2 Änderungen
public static void Switch(object state, Page newPage = null)
{
    MakeSwitch(newPage, 1, state);
}

/// <summary>
/// Makes a Switch to a new Page
/// The count of switches is based on the summand
/// </summary>
/// <param name="newPage">The new Page which should be switched to</param>
/// <param name="summand">How many Pages should be switched (Only necessary when newPage is null</param>
/// <param name="state">The switching state</param>
4 Verweise | Florian Redinger, vor 44 Tagen | 2 Autoren, 10 Änderungen
private static void MakeSwitch(Page newPage, int summand, object state = null)
{
```

Abb. 81: Methoden, welche sich um Findung der neuen Fenster kümmern

3.4.7 Lokalisierung

Die im Creator definierte Lokalisierung wird natürlich in der Runtime abgerufen und in eine Datenbank eingelesen. Die Datenbank basiert auf folgendem Datenmodell:

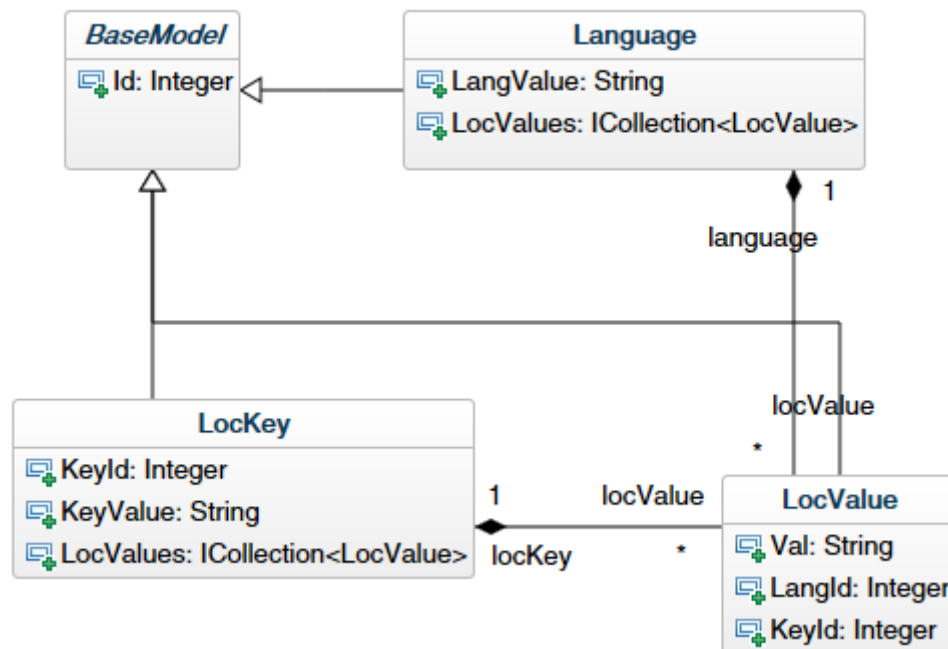


Abb. 82: Datenbank-Modell der Lokalisierung in der Runtime

Das **BaseModel** ist eine abstrakte Klasse, von der alle Datenbank-Entitäts ableiten. Damit erspart man sich, jeder einzelnen Klasse das Feld „Id“ geben zu müssen.

„**Language**“ enthält die Bezeichnung der jeweiligen Sprache und eine Liste der Wörter in dieser Sprache. „**LocKey**“ ist ein Schlüssel, der ein bestimmtes Wort darstellt. Hierbei sind „**LocValues**“ die einzelnen Übersetzungen für dieses Wort. „**LocValue**“ enthält mit „**Val**“ einen bestimmten Begriff in einer bestimmten Sprache.

Um die Datenbank mit Werten zu befüllen, wird die CSV-Datei, welche die Sprachinformationen enthält, ausgelesen. Dies geschieht in einem sogenannten „Datenbank-Initializer“:

```

public class LocalizationInitializer :
    DropCreateDatabaseAlways<LocalizationContext> //DropCreateDatabaseIfModelChanges<LocalizationContext>
{
    private List<List<string>>> csvMatrix;
    1 Verweis | Florian Redinger, vor 237 Tagen | 1 Autor, 1 Änderung
    public LocalizationInitializer()
    {
        csvMatrix = new List<List<string>>>();
        LoadCsv();
    }

    1 Verweis | Florian Redinger, vor 8 Tagen | 1 Autor, 1+1 Änderungen
    protected override void Seed(LocalizationContext context)
    {
        base.Seed(context);

        List<Language> languages = new List<Language>();
        List<LocKey> locKeys = new List<LocKey>();
        List<LocValue> locValues = new List<LocValue>();

        languages = GetLanguages();
        locKeys = GetLocKeys();
        if(languages.Count!=0&&locKeys.Count!=0)
            locValues = GetLocValues(languages, locKeys);

        context.Languages.AddRange(languages);
        context.LocKeys.AddRange(locKeys);
        context.LocValues.AddRange(locValues);
    }
}

```

Abb. 83: Initializer der Lokalisierungs-Datenbank

LoadCsv liest die Datei zeilen- und spaltenweise als zweidimensionale Listenstruktur ein:

```

private void LoadCsv()
{
    string locCsv = Properties.Resources.Loc;
    //string locCsv = new StreamReader("Loc.csv").ReadToEnd();

    string line;
    StringReader sr = new StringReader(locCsv);
    while ((line=sr.ReadLine())!=null)
    {
        csvMatrix.Add(line.Split(';').ToList());
    }
}

```

Abb. 84: Methode zum Lesen der benötigten Daten zur Lokalisierung

GetLanguages() generiert aus dieser Matrix die Liste der Sprachen.

```
private List<Language> GetLanguages()
{
    List<string> langs = new List<string>();
    csvMatrix[0].Where(x => !(x.Equals("ID") || x.Equals("Keys")))
        .ToList().ForEach(x=>langs.Add(x));

    return langs.Select(x=>new Language {
        LangValue = x
    }).ToList<Language>();
}
```

Abb. 85: Methode zum Lesen der Sprachen aus der CSV-Matrix

Wenn eine bestimmte Spalte nicht als „ID“ oder „Keys“ gekennzeichnet ist, dann kann man davon ausgehen, dass es sich um eine Sprachbezeichnung handelt. Aus den Strings mit dieser Bezeichnung werden mit einem Linq-Ausdruck Language-Objekte generiert.

GetLocKeys() lädt die einzelnen Keys.

```
private List<LocKey> GetLocKeys()
{
    List<LocKey> locKeys = new List<LocKey>();
    int keyCol = csvMatrix[0].FindIndex(x => x == "Keys");
    csvMatrix.Skip(1).ToList().ForEach(x => {
        LocKey key = new LocKey()
        {
            KeyId = int.Parse(x[0]),
            KeyValue = x[keyCol]
        };
        locKeys.Add(key);
    });
    return locKeys;
}
```

Abb. 86: Methode zum Lesen der Keys aus einer CSV-Matrix

Es wird jene Spalte gesucht, die mit „Keys“ betitelt ist. Alle Einträge dieser Spalte werden dann in eine Liste eingefügt und zurückgegeben.

3.4.8 Grundlegende Funktionen des Programms

Gewisse Funktionen sind grundlegend und fordern fehlerfreien Ablauf, um konsistenten Nutzen aus dem Programm zu ziehen. Neben dem Installations-Framework gibt es weitere, die von der TGW gefordert werden:

3.4.8.1 Bedienbarkeit durch Konsolenbefehle

Noch bevor die grafische Version des statischen Installers entwickelt wurde, nutzte die TGW Konsolenbefehle, um die zu installierenden Pakete zu definieren. Da dies ein sehr ressourcenschonendes Verfahren ist und bereits Mitarbeiter an die Bedienung mit der Konsole gewöhnt sind, soll auch diese dynamische Version vollständig mit Kommandos ausführbar sein. Dies schränkt die Auswahl der zu bearbeitenden Parameter wie auch in aktueller Nutzung ein. Nicht für jeden Parameter wird ein Kommando zur Verfügung gestellt, was in der Konsolenbedienung auch zur Einschränkung der Nutzbarkeit führt. Diese ist jedoch gewollt, da hier nur grundlegende Operationen durchgeführt werden (ohne sich in den Details der verschiedenen Parameter zu verlieren). Sollte der Benutzer Befehle vergessen haben, so kann sich dieser mit „-Help“ behelfen.

Für eine detaillierte Beschreibung, siehe 3.4.1 Startup.

3.4.8.2 Programm verlassen

Jedes Programm sollte zu jeder beliebigen Zeit beendet werden können, ohne dass Probleme auftreten, mit denen sich der Benutzer befassen muss. Die Runtime erfüllt diese Funktion ebenfalls. Die Software sieht kein Verlassen des Programmes während einer laufenden Installation vor. Da dies jedoch nur den allerletzten Schritt des Installers darstellt, kann der Benutzer davor jederzeit aussteigen.

3.4.8.3 Lesen und verarbeiten von XML-Dateien

Im Ordnerverzeichnis der selbst erstellten Package-Installer Runtime wird nach dem „Build“-Vorgang (Build-Befehl des Creators) eine XML-Datei generiert, welche von der Software automatisch erkannt und verarbeitet wird. Mithilfe eines eigens dafür entwickelten „XML Serializers“ liest das Programm nicht nur allgemeine Informationen des Installers aus, sondern auch alle Komponenten und deren zugehöriger Controls, um damit die Installations-Parameter zu bearbeiten. Auch werden hier alle Komponenten-Gruppen und Abhängigkeits-Gruppen verarbeitet und für den späteren Gebrauch zur Verfügung gestellt.

4 Ergebnisse

4.1 Vorteile für die TGW

Die Entwicklung des Package-Installer Creator löst viele Probleme vorrangegangener Programme. Dabei erspart sich die TGW hohe Kosten, da nicht nur intern, sondern auch extern an Effektivität gewonnen wird.

4.1.1 Aktuelle Umstände

Derzeit müssen für ein Projekt insgesamt ca. 25 Komponenten installiert werden, wobei jedoch jeder Mitarbeiter wissen muss, welche vorhanden sind und wo man diese findet.

Jede Komponente muss einzeln installiert werden. Pro Komponente kann mit ca. fünf Minuten gerechnet werden. Daraus werden dann $25 \cdot 5\text{min} = 2\text{h}$, die der Installierende entweder bei dem Rechner verbringen oder zumindest immer wieder kontrollieren muss.

Die Komponenten sind oftmals voneinander abhängig. Teilweise müssen diese nicht nur gemeinsam installiert werden, sondern auch dieselbe Versionsnummer haben.

Der Installationsprozess erfolgt oft mehrfach auf verschiedenen Systemen (Entwicklungssystem, Testsystem) in unterschiedlichen Kombinationen, aufgrund regelmäßiger Releases und Service-Packs zumindest mehrmals pro Monat.

Die Anforderungen hinsichtlich dem Know-How der Personen, die eine solche Installation durchführen können, sind beträchtlich hoch. Die Chancen, bei diesem Prozess einen Fehler (Komponenten fehlen, inkompatible Versionen) zu machen, sind extrem hoch. Dies führt zu Fehlfunktionen, deren Identifikationen oft Stunden, wenn nicht sogar Tage beanspruchen.

4.1.2 Vorteile und Verbesserungen

Dank unserer Entwicklung ist es nun möglich, alle Komponenten in einem einzigen Installer zu vereinen, welche mit der richtigen Version in einem einzigen Paket vorhanden und installierbar sind. Die Flexibilität, nur einzelne Pakete zu installieren, bleibt stets erhalten (benutzerdefinierte Installation).

Abhängige Komponenten werden automatisch erkannt, wodurch die Installationen dann immer konsistent sind (Abhängigkeitsgruppen).

Die Möglichkeit, Komponenten für einen bestimmten Server/Client zu gruppieren, reduziert die Know-How Anforderungen massiv, da zum Beispiel niemand mehr wissen muss, welche Komponenten auf welchen Server übertragen werden müssen (Komponentengruppen).

Der Installationsprozess funktioniert einheitlich über alle Komponenten-Typen. Durch die Konsolen-Option kann die Installation für Testsysteme automatisiert (ohne menschlich verrichtete Arbeit) erfolgen.

Der Installierende wählt die zu installierenden Komponenten aus und danach erfolgt die Installation (vorausgesetzt die Parameter wurden richtig definiert).

Dank der gewählten Architektur des Installers kann dieser beliebig oft erweitert und zusätzlicher Code ausgeführt werden, der beispielsweise die Konfiguration der Komponenten übernimmt, wodurch neben der Installation auch die Konfiguration der Komponenten zumindest teilautomatisiert werden kann.

Das bedeutet in Summe statt 2 Stunden nur mehr 2 Minuten Aufwand. Über die gesamte TGW verteilt werden dabei pro Monat Ersparnisse in Höhe von €6.000 erzielt da für jeden Monat ca. 50 Installationen geschätzt werden, wobei ein Inbetriebnehmender / eine Inbetriebnehmende einen Stundenlohn von €60/h hat.

4.2 Umfang

4.2.1 Arbeitsstunden

4.2.1.1 Geplante Arbeitsstunden

Tabelle 8: Geplante und maximale Stunden der einzelnen Teammitglieder

Mitglied	Stunden in h (geplant)	Stunden in h (maximal)
Florian Redinger	180	200
Florian Peter	180	200
Andreas Wageneder	180	200

4.2.1.2 Aktuelle Arbeitsstunden

Tabelle 9: Die tatsächlich verbrachte Arbeitszeit der einzelnen Teammitglieder

Mitglied	Stunden in h (erbracht)
Florian Redinger	300
Florian Peter	190
Andreas Wageneder	210

4.2.2 Source-Code

4.2.2.1 Package-Installer Creator

Tabelle 10: Menger der LOC, der einzelnen Projekte des Package-Installer Creators

Projektmappe	Codezeilen
Application	8768
ExtensionMethods	487
ExtensionMethodsTests	81
PageSwitcher	107
SingletonLib	986
XmlSerialization	1620

4.2.2.2 Package-Installer Runtime

Tabelle 11: Menge der LOC, der einzelnen Projekte der Package-Installer Runtime.

Projektmappe	Codezeilen
ApplicationConsole	617
ApplicationUI	2889
ConsoleTests	90
Controls	500
Exceptions	28
ExtensionMethodsLib	514
GeneratingControlsLib	840
LocalizationModel	200
PageSwitcherFunction	124
SingletonLib	958

4.3 Lessons learned

Durch die Arbeit an einer Diplomarbeit, bei welcher man sich mit verschiedenen Technologien beschäftigen muss, gibt es einen dauerhaften Lernprozess. Dies war natürlich auch bei dieser Diplomarbeit der Fall. Neues wurde vor allem in den Bereichen des Projektmanagements gelernt, aber auch in technischen Bereichen.

4.3.1 Projekt

Im Bereich Projektmanagement haben wir viele neue Erfahrungen gewonnen. Ein Projekt dieser Größenordnung wurde noch nie von uns durchgeführt. Durch die bereits erwähnten Umstände hat sich das Projekt unerwartet verzögert. Wir mussten lernen, mit den Gegebenheiten zurechtzukommen und den Zeitplan neu anpassen. Auch Nebengebiete wie Berichtswesen waren deutlich genauer zu beachten, als dies bei bisherigen Projekten der Fall war.

4.3.2 Erlerntes Wissen

Aber nicht nur im Bereich des Projektmanagements haben wir viele Erfahrungen gesammelt, sondern auch in technischen Bereichen, wie beispielsweise C#, WPF, XSD und Designpatterns.

4.3.2.1 C#

Die Programmiersprache C# wurde uns zwar im Unterricht beigebracht, was sich allerdings nur auf die Basics bezog. Einerseits wurde unser Wissen in diesen Bereichen erweitert, andererseits durften wir mit völlig neuen Aspekten Bekanntschaft machen. Vor allem in den Aspekten Reflection und Rekursion durften wir sehr viel Neues dazulernen, da ein großer Teilbereich unserer Diplomarbeit aus der Dynamik der Projekte „Package-Installer Creator“ und „Package-Installer Runtime“ bestand. Die Rekursion wurde ebenfalls sehr häufig verwendet, da bei der Package-Installer Runtime auf einen XML-Serializer verzichtet wurde, denn diese soll auch selbst hinzugefügte Attribute verarbeiten können. Dies bedeutet, dass die zu verarbeitende XML-Datei, welche wie ein Baum aufgebaut ist, vollständig und dynamisch auf Kinder und Attribute von Kindeskindern, untersucht werden können muss.

Völlig neue Herausforderungen ergaben sich allerdings für uns, als wir das Window Behavior selbst programmieren mussten. Dabei lernten wir, wie man Betriebssystemnachrichten überwachen kann und die dazugehörigen Pointer so verarbeitet, dass man die gewünschten Daten erhält. Außerdem wurde oftmals die externe Bibliothek „user32.dll“ verwendet, welche ebenfalls bestimmte Betriebssystemmethoden zur Verfügung stellt.

4.3.2.2 WPF

Mit dem Grafik-Framework WPF (Windows Presentation Foundation) haben wir keinerlei Erfahrung gehabt. Deshalb mussten wir uns dieses völlig neu aneignen. Da die Verwendung und das Wissen darüber für die gesamte Diplomarbeit von sehr hoher Bedeutung waren, war die Menge an zu erlernendem Wissen natürlich sehr groß. Im Laufe unserer Informationssammlung trafen wir auch auf das Designpattern MVVM (Model View ViewModel), welches eine sehr große Unterstützung für uns brachte. Dadurch wurde nämlich das „Binding“ und die Trennung zwischen Designer und Programmierer sehr viel einfacher.

Weiters konnten wir uns mit den Problemen, welche WPF bereithält, vertraut machen. So wird zum Beispiel in diesem Framework sehr stark auf individuelle Designmöglichkeit gesetzt. Es ist allerdings bei groben Änderungen notwendig, das gesamte Template eines bestimmten Controls zu kopieren und dieses zu bearbeiten. Ein weiteres Problem ergibt sich dadurch, dass diese Templates sehr umfangreich sind und es somit auch eine Weile dauert, bis man die richtigen Stellen gefunden hat, an denen man arbeiten muss, um das gewünschte Ergebnis zu erzielen.

4.3.2.3 XSD

Die XML Schema Definition ist im Programmierunterricht der dritten Klasse behandelt worden. Hierbei haben wir uns allerdings nicht sehr ausführlich mit dem Thema beschäftigt. Nachdem wir unser Wissen in diesem Gebiet wieder aufgefrischt hatten, mussten wir uns auch in diesem Fachgebiet viel selbst aneignen, um unser komplexes Datenschema hinreichend darstellen zu können.

4.3.2.4 Patterns

Das MVVM-Muster war uns zu Beginn der Softwareentwicklung noch nicht bekannt. Erst im Laufe der Programmierung klärte uns ein Mitarbeiter der Partnerfirma darüber auf. Wir mussten daraufhin einen großen Teil des Codes umstrukturieren. Anfangs waren wir noch etwas unsicher, doch wir wurden mit der Materie schnell vertraut. Wir profitieren definitiv davon, da wir solche Muster auch in unserer zukünftigen Berufspraxis oft anwenden werden.

Andere Patterns wie Singleton und Factory wurden im Schulunterricht fast nur theoretisch behandelt und waren daher gewissermaßen ebenfalls neu für uns.

Quellen-/Literaturverzeichnis

- <https://msdn.microsoft.com/ro-ro/library/mt674882.aspx> [8. März 2017]
<https://msdn.microsoft.com/ro-ro/library/mt674882.aspx> [8. März 2017]
- https://de.wikipedia.org/wiki/Model_View_ViewModel [12. März 2017]
- <https://de.wikipedia.org/wiki/Fabrikmethode> [12. März 2017]
- [https://de.wikipedia.org/wiki/Singleton_\(Entwurfsmuster\)](https://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster)) [12. März 2017]
- https://sourcemaking.com/design_patterns/singleton [12. März 2017]
- <https://de.wikipedia.org/wiki/C-Sharp> [16. März 2017]
- https://de.wikipedia.org/wiki/Windows_Presentation_Foundation [16. März 2017]
- <https://de.wikipedia.org/wiki/Scrum> [16. März 2017]
- https://www.w3schools.com/xml/xml_schema.asp [20. März 2017]
- <http://stackoverflow.com/questions/1544200/what-is-difference-between-xml-schema-and-dtd> [20. März 2017]
- <http://xsd2xml.com> [20. März 2017]

Abbildungsverzeichnis

Abb. 1: Backlogs der Package-Installer Runtime inkl. Epics, Features und User Stories. .	11
Abb. 2: Backlogs des Package-Installer Creators inkl. Epics, Features und User Stories. .	13
Abb. 3: Beispielschema eines nach MVVM aufgebauten Fensters.	16
Abb. 4: Beispiel eines Singletons mit Hinweis auf Erstellung und Verwendung.	18
Abb. 5: Beispiel einer Factory-Method mit 3 konkreten Unterprodukten.	21
Abb. 6: Beispiel eines DTD-Schemas anhand eines Tourenplans.	22
Abb. 7: Beispiel eines XSD-Schemas anhand der Organisation einer Schule.	23
Abb. 8: Beispiel eines XSD-zu-XML-Generators.	24
Abb. 9: Teilbereich des XSD-Schemas, welcher den Installer und die Description definiert.	25
Abb. 10: Darstellung der aus Abb. 9 erstellten XML-Elemente.	25
Abb. 11: Darstellung der Komponenten-Liste in einem XSD-Schema.	26
Abb. 12: Beispiel der XSD einer spezifischen Komponente.	26
Abb. 13: Erstelltes XML-Element aus dem in Abb. 12 gegebenen Schema.	26
Abb. 14: Erstelltes XML-Element „OptionalControls“ und dessen Kind-Elemente, aus dem in Abb. 12 gegebenen Schema	27
Abb. 15: Beispiel eines XSD-Schemas für einen bestimmten Komponenten-Typen mit obligatorischen Elementen.....	27
Abb. 16: Mögliche Erstellung eines XML-Elements nach dem in Abb. 15 gegebenen Schema.	28
Abb. 17: XSD-Schema einer Komponenten-Gruppe	28
Abb. 18: XSD-Schema einer Abhängigkeits-Gruppe	28
Abb. 19: Erstelltes XML-Element nach dem in Abb. 17 gegebenen Schema.....	29
Abb. 20: Erstelltes XML-Element nach dem in Abb. 18 gegebenen Schema.....	29
Abb. 21: Menüdefinition im Shell-Fenster.....	30
Abb. 22: Die in XAML gebundene Liste für das Menü.	31
Abb. 23: Initialisierung der einzelnen Kommandos.	31
Abb. 24: Erstellung des Menüs	32
Abb. 25: Notwendige Umwandlung der Liste vom Typ „FrameworkElement“ in eine Liste vom Typ „object“	32
Abb. 26: Methode, welche aufgerufen wird, wenn das Programm geschlossen werden soll.....	33
Abb. 27: Dialog für ungespeicherte Änderungen	34
Abb. 28: Button-Events des Dialogs.....	34
Abb. 29: Definition eines RelayCommand.....	35
Abb. 30: Definition eines AsyncRelayCommands	36

Abb. 31: Execute Methode des AsyncRelayCommands und das Event zur Überwachung von CanExecute.	36
Abb. 32: Selbst designte Toolbar	37
Abb. 33: TextControl	39
Abb. 34: PathControl	39
Abb. 35: NumberControl.....	39
Abb. 36: BoolControl.....	39
Abb. 37: DateControl	39
Abb. 38: TnsNamesControl	40
Abb. 39: Schema des Aufbaus der verschiedenen Controls	40
Abb. 40: Startup-Fensterdes Package-Installer Creators.....	41
Abb. 41: Beispiel-Eingabe eines Installers	42
Abb. 42: Die Ordner-Struktur, welche nach dem Klick auf „Create“ erstellt wird	42
Abb. 43: Methode zur Erstellung der Ordner-Struktur.....	43
Abb. 44: „Löschen“-Icon	44
Abb. 45: Dialog zur Erstellung einer neuen Komponente	44
Abb. 46: Button zur Erstellung einer neuen Komponente	44
Abb. 47: Standard-Miniaturansicht einer Komponente	45
Abb. 48: Bearbeitbare Properties der Komponente.....	45
Abb. 49: Alle festgelegten DefaultControls	45
Abb. 50: Exemplarische Vorschau eines Controls.....	46
Abb. 51: Custom-Creation-Fenster mit dem Control-Namen „Login“	46
Abb. 52: Bearbeitung der einzelnen Controls eines Custom-Controls	47
Abb. 53: Neues Control im Creation-Fenster.....	47
Abb. 54: Vorschau eines neu erstellten Controls	47
Abb. 55: Hinzufügen eines neu erstellten Controls mit den Default-Werten	48
Abb. 56: Komponente mit bearbeitbarem Control	48
Abb. 57: Komponente mit Controls	48
Abb. 58: "Geupdatete" Miniaturansicht der Komponente.....	49
Abb. 59: Properties und Default-Konstruktor des UiElementAttributes.....	49
Abb. 60: Das Menü der Shell	50
Abb. 61: Überwachte Properties des Build-Dialogs.....	51
Abb. 62: Definition des BackgroundWorkers	51
Abb. 63: Methode, welche sich um den Start des BackgroundWorkers kümmert	51
Abb. 64: Methode, welche sich um das Updaten der UI kümmert	52
Abb. 65: Aufruf der in Abb. 65 definierten Methode	53

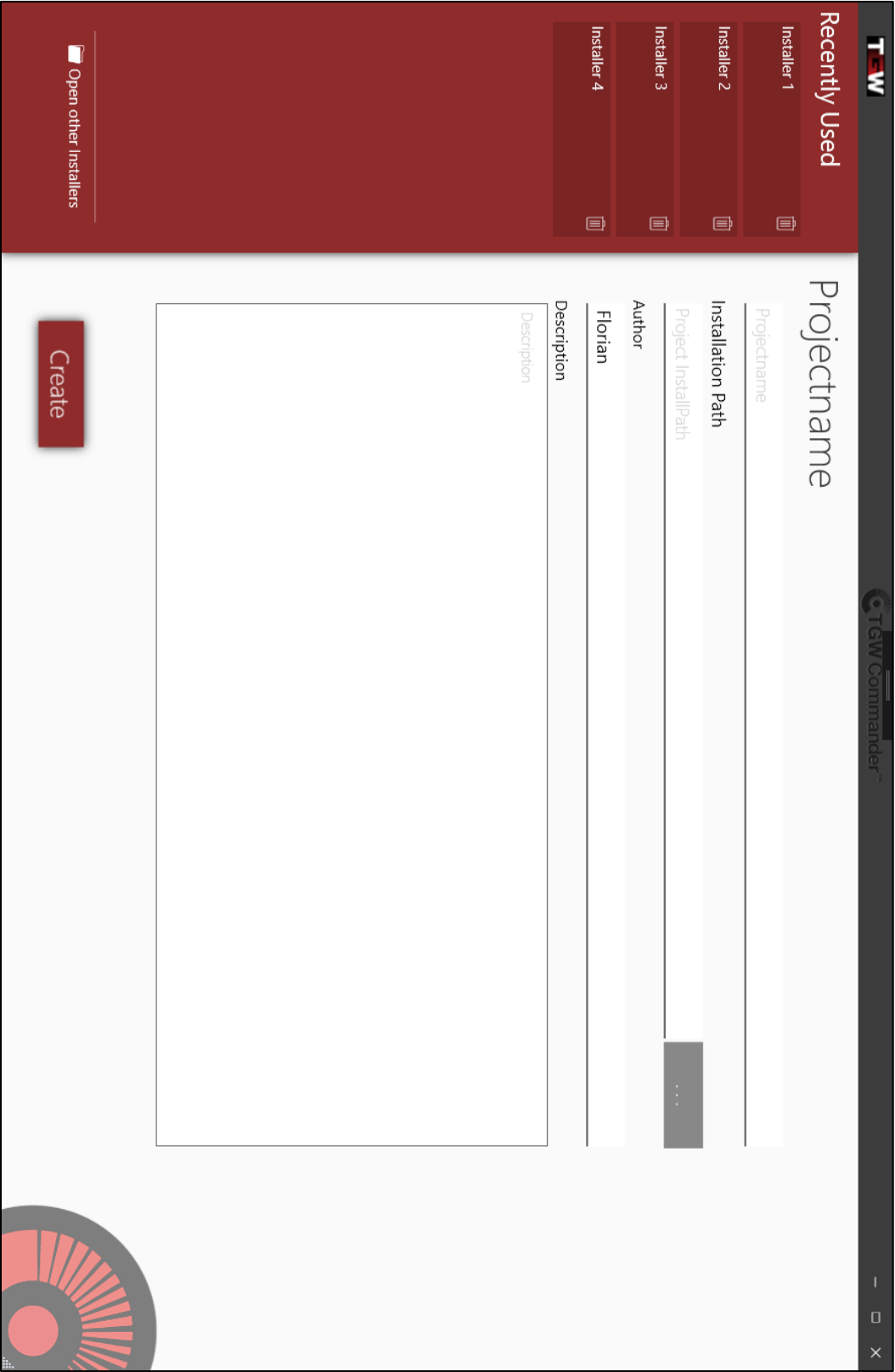
Abb. 66: Lokalisierungs-Fenster	55
Abb. 67: Dialog zum Hinzufügen einer neuen Sprache	55
Abb. 68: Dialog zur Bearbeitung der Daten (erscheint durch Klick auf ein Lokalisierungsfeld)	56
Abb. 69: Abbildung der Lokalisierungs-Datenbank des Creators	56
Abb. 70: Ordner-Struktur nach dem Build-Prozess	58
Abb. 71: Runtime als Konsolenprogramm (wenn PackageInstaller.exe mit Parameter „-h“ aufgerufen wird)	59
Abb. 72: Die Komponentengruppen der Runtime (exemplarisches Bild)	60
Abb. 73: Exemplarisches Fenster einer Komponente	61
Abb. 74: Auflistung aller verfügbaren Komponenten für die benutzerdefinierte Installation	61
Abb. 75: Herausfinden der Abhängigkeiten einer Komponente	62
Abb. 76: Komponente eines Installers	64
Abb. 77: Erstellte UI aus der, in Abb. 78, definierten Komponente	64
Abb. 78: Error-Message eines Controls	64
Abb. 79: Installations-Fenster der Runtime	65
Abb. 80: Interface, welches vom Page-Switcher-Fenster implementiert wird	66
Abb. 81: Methoden, welche sich um Findung der neuen Fenster kümmern	66
Abb. 82: Datenbank-Modell der Lokalisierung in der Runtime	67
Abb. 83: Initializer der Lokalisierungs-Datenbank	68
Abb. 84: Methode zum Lesen der benötigten Daten zur Lokalisierung	68
Abb. 85: Methode zum Lesen der Sprachen aus der CSV-Matrix	69
Abb. 86: Methode zum Lesen der Keys aus einer CSV-Matrix	69

Tabellenverzeichnis

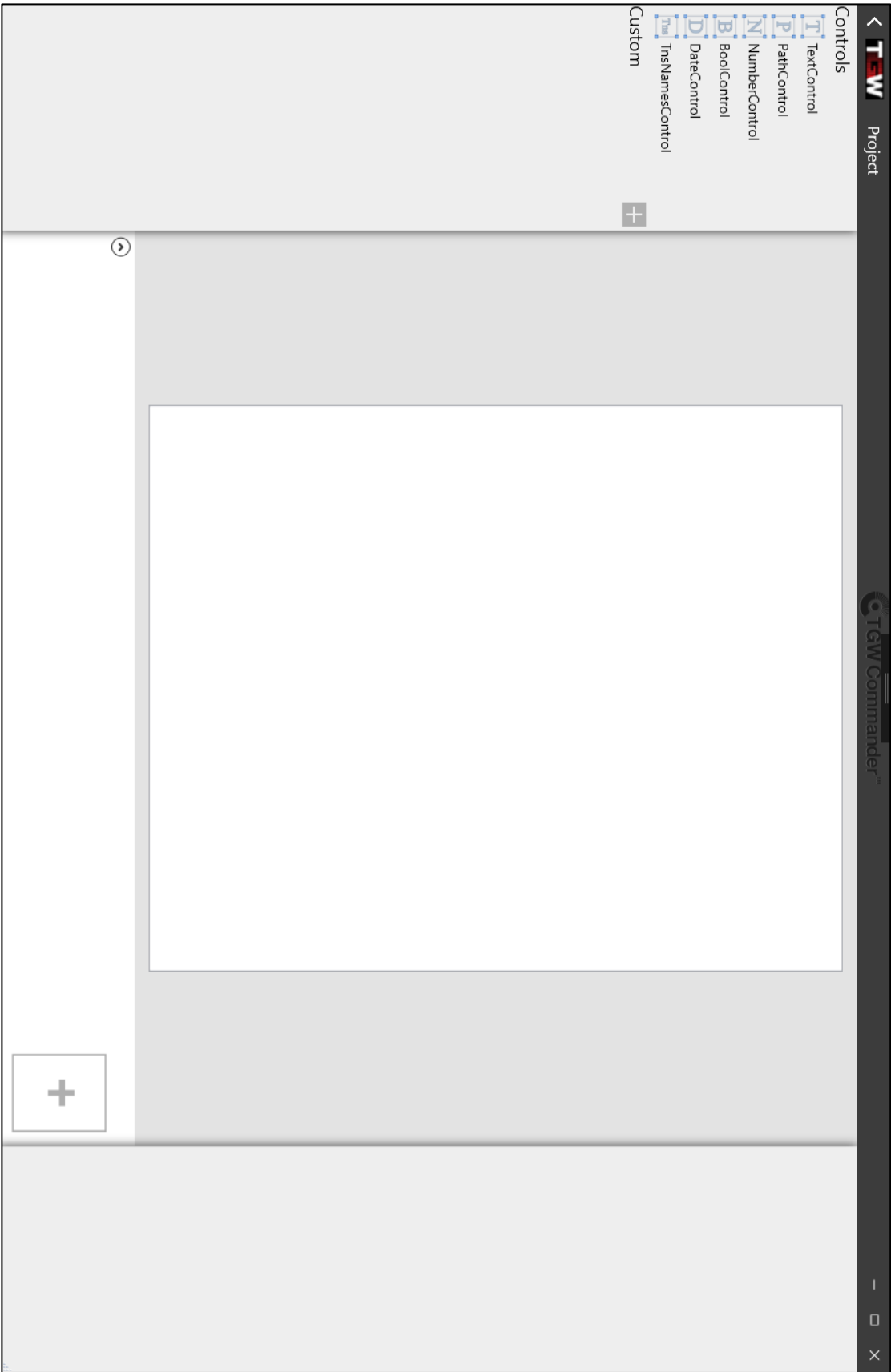
Tabelle 1: Rolleneinteilung der einzelnen Teammitglieder	6
Tabelle 2: Planung der einzelnen Projektmanagement-Phasen.....	7
Tabelle 3: Zeitplan zur Erstellung der einzelnen Features der Package-Installer Runtime	8
Tabelle 4: Zeitplan zur Erstellung der einzelnen Features des Package-Installer Creators	9
Tabelle 5: Zeitplan zur Erstellung der schriftlichen Arbeit.....	10
Tabelle 6: Kurze Beschreibung der einzelnen Komponenten-Typen.....	28
Tabelle 7: Parameter der Konsolenbedienung	59
Tabelle 8: Geplante und maximale Stunden der einzelnen Teammitglieder	73
Tabelle 9: Die tatsächlich verbrachte Arbeitszeit der einzelnen Teammitglieder	73
Tabelle 10: Menger der LOC, der einzelnen Projekte des Package-Installer Creators	73
Tabelle 11: Menge der LOC, der einzelnen Projekte der Package-Installer Runtime.....	74

Abkürzungsverzeichnis

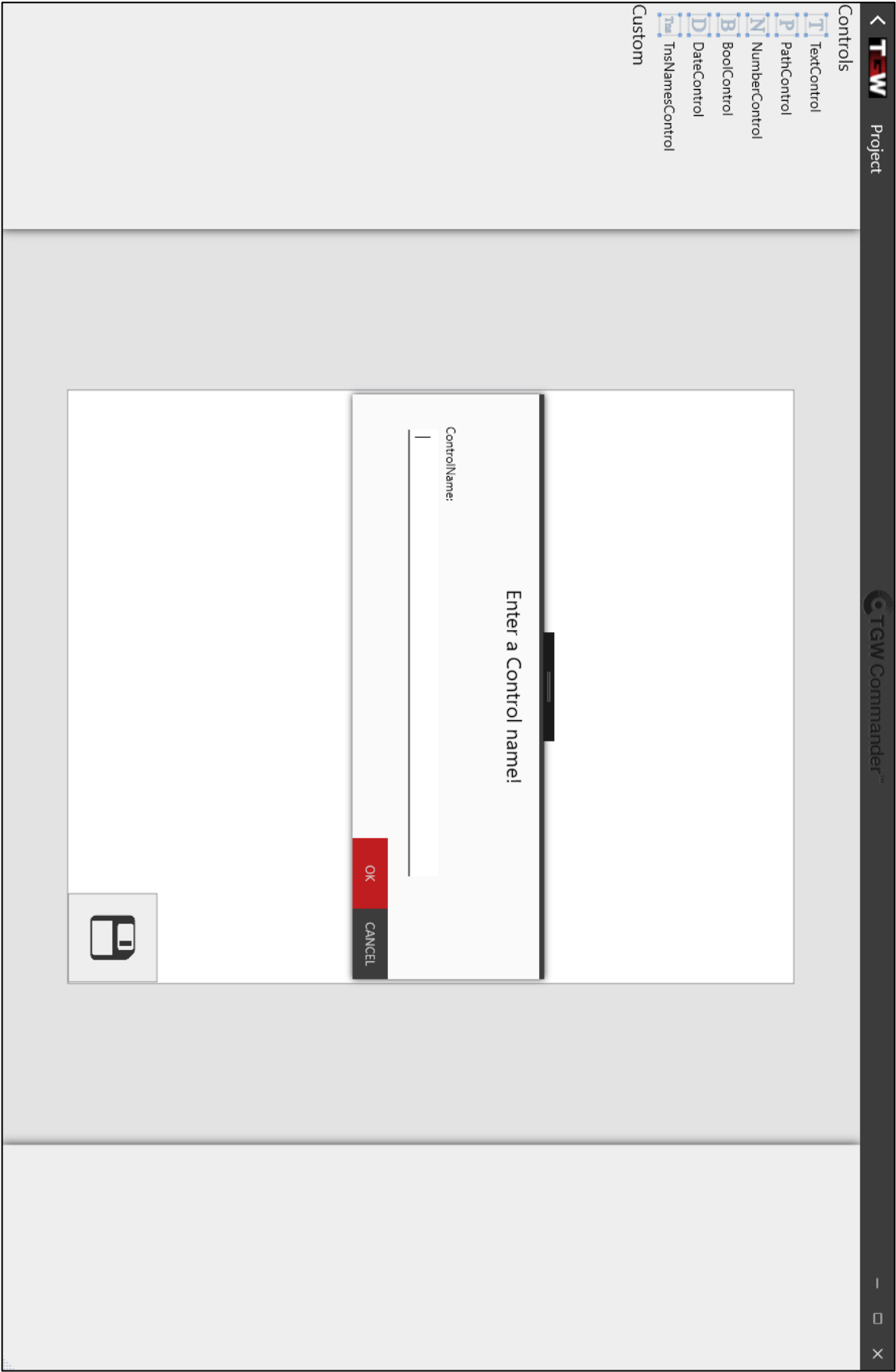
Abkürzung	Bedeutung
XML	Extensible Markup Language
LOC	Lines of Code
IDE	integrated development environment (Integrierte Entwicklungsumgebung)
CSV	Comma-separated values
UI	User Interface



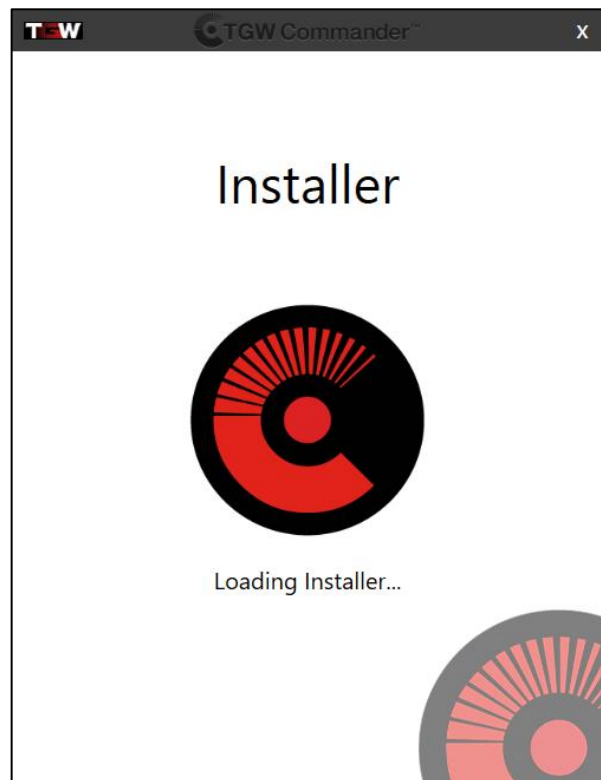
Anhang B



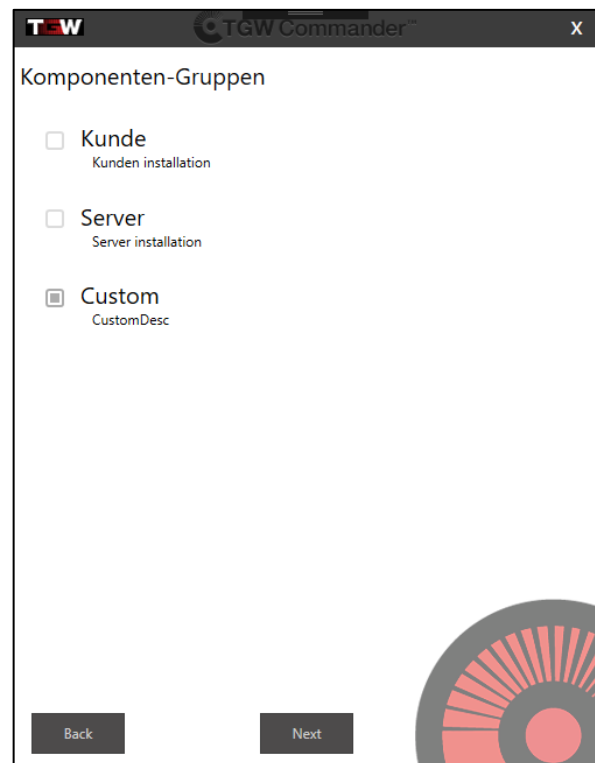
Anhang C



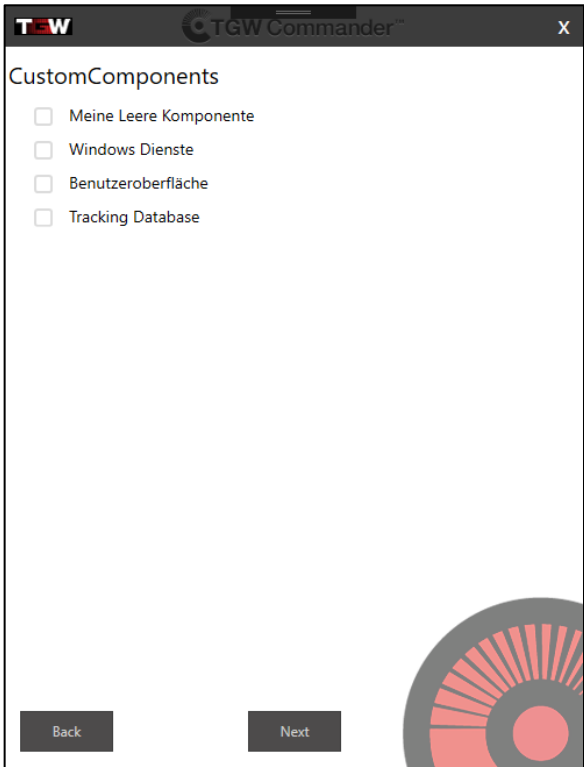
Anhang D



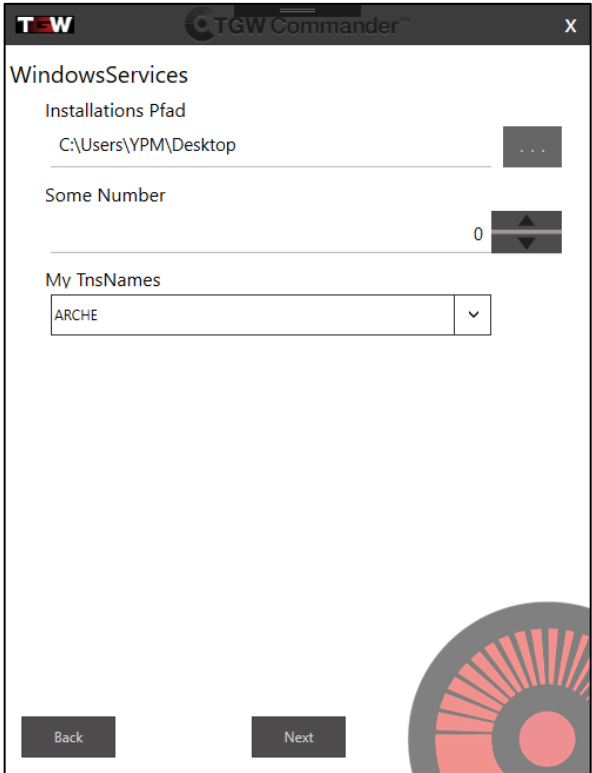
Anhang E



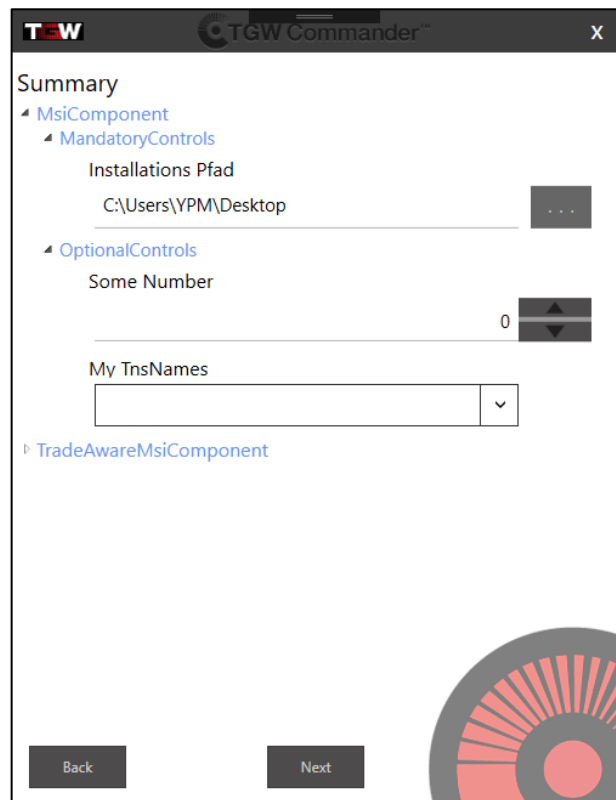
Anhang F



Anhang G



Anhang H

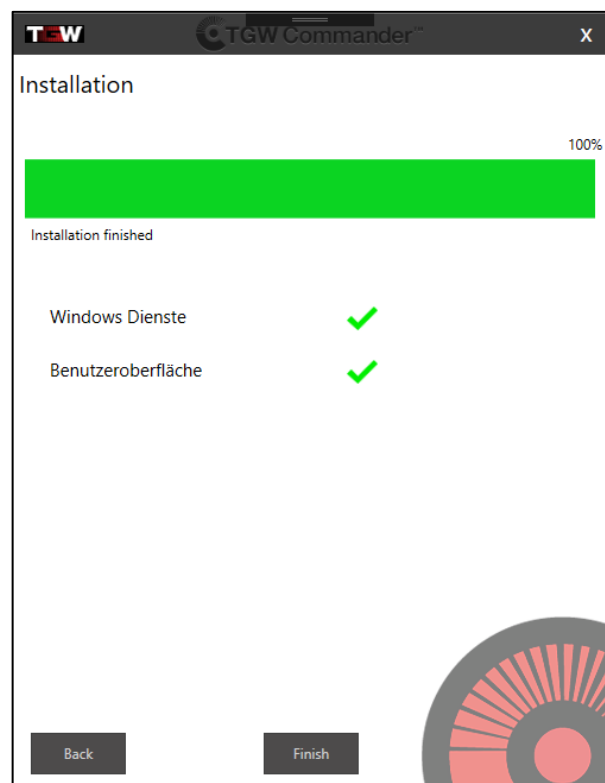


The screenshot shows the 'Summary' screen of the TGW Commander application. The window title is 'TGW Commander'. The main content area is titled 'Summary' and contains a tree view on the left with the following structure:

- Summary
 - MsiComponent
 - MandatoryControls
 - Installations Pfad
 - C:\Users\YPM\Desktop
 - OptionalControls
 - Some Number
 - 0
 - My TnsNames
 - [Empty text box with a dropdown arrow]
 - TradeAwareMsiComponent

At the bottom of the window, there are two buttons: 'Back' and 'Next'. A decorative graphic of a red and grey fan-like shape is visible in the bottom right corner.

Anhang I



The screenshot shows the 'Installation' screen of the TGW Commander application. The window title is 'TGW Commander'. The main content area is titled 'Installation' and features a progress bar at the top right indicating '100%'. Below the progress bar, the text 'Installation finished' is displayed. A list of components is shown with green checkmarks indicating successful installation:

Component	Status
Windows Dienste	✓
Benutzeroberfläche	✓

At the bottom of the window, there are two buttons: 'Back' and 'Finish'. A decorative graphic of a red and grey fan-like shape is visible in the bottom right corner.