



WiX Toolset Erweiterungskurs

Seminarunterlagen WiX Toolset Erweiterungskurs

ActiveX, JScript, Visual C++, Visual C#, DotNet, Visual Basic, Microsoft, Windows, Windows NT, Windows XP, Windows 7, Windows 8, Windows 10, das Windows Logo und das Windows NT Logo sind Marken oder eingetragene Marken der Microsoft Corporation.

Die in diesem Dokument beschriebene Software wird von SourceForge gemäß den Bedingungen einer OpenSource-Software zur Verfügung gestellt und darf nur zu den Vereinbarungen des Lizenzabkommens zur Verfügung gestellt und nur zu den darin enthaltenen Bedingungen eingesetzt werden.

Diese Dokumentation ist Eigentum von Martin Aigner und darf in keiner Weise und mit keinen Mitteln weder teilweise noch vollständig vervielfältigt oder weitergegeben werden, weder elektronisch noch mechanisch, ohne ausdrückliche vorherige schriftliche Genehmigung von Martin Aigner. Dies betrifft folgende Möglichkeiten der Vervielfältigung oder Weitergabe, ist jedoch nicht darauf begrenzt: Fotokopien, Aufzeichnungen, Datenträger und Informationswiedergewinnungssysteme.

© 2022 SD-Technologies GmbH
Alle Rechte vorbehalten

SD-Technologies GmbH
Dennewartstrasse 25/27
D-52068 Aachen

Tel.: +49 (0)241 / 518 377 10
Fax.: +49 (0)241 / 518 377 29

Mail: info@sd-technologies.de
Web: www.sd-technologies.de

Stand 01.2022
Version 2.3.0

Inhalt

1 EINLEITUNG	12
1.1 Warum Windows Installer.....	13
1.1.1 Möglichkeiten des Windows Installers.....	13
1.1.2 Probleme mit herkömmlichen Installationen.....	14
2 VORBEREITUNG DER UMGEBUNG	15
2.1 Installation des WiX-Toolsets	15
2.2 Installation des Windows Installer SDKs	16
2.3 Installation von Orca.....	16
3 DAS ERSTE SETUP ENTSTEHT.....	17
3.1 Globally Unique Identifier (GUID).....	17
3.1.1 GUID-Generator (bis Visual Studio 2012)	18
3.1.2 GUID-Generator (Visual Studio 2013).....	18
3.2 Neue GUIDs erstellen	19
3.3 Aufbau der Installationsstruktur.....	20
3.4 Dateien hinzufügen	21
3.4.1 Medium erstellen	21
3.4.2 Wurzelverzeichnis (Root-Directory) erstellen	22
3.4.3 Komponente erstellen.....	22
3.4.4 Datei einbinden.....	23
3.4.5 Datei-Überschreibungs-Regeln	23
3.4.5.1 Die höchste Versionsnummer gewinnt	23
3.4.5.2 Unversionierte Dateien	24
3.5 Shortcut erstellen	24
3.6 Komponente einem Feature zuweisen.....	25
3.7 Das gesamte Skript im Überblick	25
3.8 Setup kompilieren.....	26
3.9 Setup ausführen	27
3.10 Installationsarten des Windows Installers	28
3.11 Setup mit Orca öffnen	28
4 PFAD ZU DEN QUELDATEIEN BESTIMMEN	30
4.1 Compiler, Linker und Binder.....	30
4.2 Binden der Dateien durch das Directory-Element.....	30
4.3 Binden der Dateien durch das File-Element	31
4.4 Binden der Dateien durch benannte Bindepfade	32
5 BEDINGUNGEN DEFINIEREN.....	33
5.1 Properties	33
5.2 Installationsbedingungen.....	34
5.3 System-Search	37
5.3.1 Werte aus der Registry lesen	37
5.3.2 Werte aus einer INI-Datei lesen	37
5.3.3 Dateien suchen.....	38
5.4 Bedingung bei Feature	38
5.5 Bedingung bei Komponenten	39
6 MODULARITÄT DURCH FRAGMENTS	40
6.1 Fragment erstellen	40
6.2 Fragment referenzieren.....	41

6.3	Fragment als Funktionsbibliothek	42
7	WIX-TOOLSET-VARIABLEN	43
7.1	WiX-Variablen	43
7.2	Projektreferenzvariablen	43
7.3	Präprozessorvariablen	44
7.4	Umgebungsvariablen	44
7.5	Automatisch generierte Versionsnummer	45
7.6	Bindervariablen	45
7.7	Iterationen	46
7.8	Escape-Sequenz	47
8	INCLUDE-DATEIEN	48
9	NEBEN DEN DATEIEN	50
9.1	Registry	50
9.2	Dateiendungen registrieren	51
9.3	INI-Dateien	51
9.4	HTML-Link erstellen	52
9.5	ODBC-Eintrag erstellen	52
9.5.1	<i>ODBC-DSN erstellen</i>	53
9.5.2	<i>ODBC-Treiber installieren</i>	55
9.6	Dateien löschen	56
10	DATEIEN IN DEN GLOBAL ASSEMBLY CACHE INSTALLIEREN	57
11	BENUTZERINTERFACE	58
11.1	WiX-Extension	58
11.2	Benutzerinterface einbinden	58
11.3	Jetzt wird's bunt	60
11.3.1	<i>Bitmaps austauschen</i>	60
11.3.2	<i>Lizenzvereinbarung</i>	60
11.4	Einstellungen beim Feature	60
12	LOKALISIERUNG	62
12.1	WixUI-Extension-Library	62
12.2	String-Verweise und WiX-Localisation-Files	64
13	EIN NEUER DIALOG ENTSTEHT	66
14	BILLBOARDS	72
15	SEQUENZEN	73
15.1.1	<i>Installationssequenz</i>	73
15.1.2	<i>Advertisement-Installation</i>	73
15.1.3	<i>Administrative Installation</i>	74
15.1.4	<i>Silent-Installation</i>	74
15.1.5	<i>InstallUISequence</i>	75
15.1.6	<i>InstallExecuteSequence</i>	76
15.2	Status- und Aktion-Flag einer Komponente	77
15.3	Die Sequenzen im WiX-Skript	78
16	EIGENE CUSTOM-ACTION ERSTELLEN	79
16.1	Property in Custom-Action setzen	79
16.2	Direcotry in Custom-Action setzen	79
16.3	EXE in Custom-Action aufrufen	80
16.4	C#-Custom-Action	83
16.4.1	<i>Ein neue C#-Klassenbibliothek erstellen</i>	83
16.4.1.1	Funktion anpassen	84
16.4.1.2	Build-Vorgang	85
16.4.2	<i>Custom-Action einbinden</i>	85

16.5 VB.NET-Custom-Action.....	86
16.5.1 Ein neue VB.NET-Klassenbibliothek erstellen	86
16.5.1.1 Funktionen anpassen	86
16.5.1.2 Build-Vorgang	87
16.5.2 Custom-Action einbinden	87
16.6 C++-Custom-Action.....	87
16.6.1 Eine neue C++-DLL erstellen	87
16.6.2 Funktion anpassen	88
16.6.3 Custom-Action einbinden	89
16.6.4 Die Wca-Library.....	90
16.7 VBScript-Custom-Action.....	90
16.7.1 VBScript direct in Custom-Action erstellen.....	91
16.7.2 Custom-Action über VBS-Datei erstellen	91
16.7.3 Ausgabe in die Logdatei des Windows Installers.....	91
16.8 JavaScript-Custom-Action.....	92
16.8.1 JavaScript in Custom-Action	92
16.8.2 JavaScript in JS-Datei	92
16.8.3 Ausgabe in die Logdatei des Windows Installers.....	93
16.9 PowerShell als Custom-Action aufrufen.....	93
16.9.1 PowerShell-Skript ausführen.....	94
17 CUSTOM-ACTION DEBUGGEN	97
17.1 C#-Action debuggen.....	97
17.1.1 Debugger über Umgebungsvariable aufrufen	97
17.1.2 Debugger über MessageBox an Prozess anhängen.....	98
17.1.3 Debugger über System.Diagnostics.Debugger aufrufen.....	99
17.2 VB.NET-Action debuggen	99
17.3 C++-Action debuggen	100
17.3.1 Debugger über Umgebungsvariable aufrufen	100
17.3.2 Debugger über MessageBox an Prozess anhängen.....	101
17.4 VBScript-Action debuggen	101
17.5 JavaScript-Action debuggen	102
17.6 Debug-Informationen mit DebugView	102
18 INSTALLEXECUTE-SEQUENZ NÄHER BETRACHTET	104
18.1 Installationsskripte.....	104
19 PARAMETERÜBERGABE AN INSCRIPT-CUSTOM-ACTIONS	106
19.1 Parameterübergabe in C#	106
19.1.1 Immediate-Action.....	106
19.1.2 InScript-Action	107
19.1.3 WiX-Skript.....	107
19.2 Parameterübergabe in VB.NET	108
19.2.1 Immediate-Action.....	108
19.2.2 InScript-Action	108
19.2.3 WiX-Skript.....	108
19.3 Parameterübergabe in C+.....	109
19.3.1 Immediate-Action.....	109
19.3.2 InScript-Action	110
19.3.3 WiX-Skript.....	110
19.4 Parameterübergabe in VBScript	111
19.4.1 Immediate-Action.....	111
19.4.2 InScript-Action	111
19.4.3 WiX-Skript.....	111
19.5 Parameterübergabe in JavaScript.....	112
19.5.1 Immediate-Action.....	112
19.5.2 InScript-Action	112

19.5.3 WiX-Skript.....	112
20 ANWENDUNG IM HINTERGRUND AUSFÜHREN.....	113
20.1 Immediate Execution.....	113
20.2 Deferred Execution.....	114
20.3 64-Bit-Anwendung silent starten	114
21 CUSTOM TABLE: EIGENE MSI-TABELLE ERSTELLEN	115
21.1 Tabelle definieren.....	115
21.2 Modularisierung	116
21.3 Werte in die Tabelle schreibe.....	117
22 WINDOWS INSTALLER SQL-SYNTAX.....	118
23 MSI-TABELLE ÜBER CUSTOM-ACTION AUSLESEN.....	120
23.1 MSI-Tabelle mit C# auslesen	120
23.2 MSI-Tabelle mit VB.NET auslesen.....	121
23.3 MSI-Tabelle mit C++ auslesen	121
23.4 MSI-Tabelle mit VBScript auslesen.....	122
23.5 MSI-Tabelle mit JavaScript auslesen	122
24 IN MSI-TABELLE MIT CUSTOM-ACTION SCHREIBEN	123
24.1 Vorbereitung	123
24.2 In MSI-Tabelle mit C# schreiben.....	124
24.3 In MSI-Tabelle mit VB.NET schreiben	124
24.4 In MSI-Tabelle mit C++ schreiben.....	125
24.5 In MSI-Tabelle mit VBScript schreiben	125
24.6 In MSI-Tabelle mit JavaScript schreiben.....	126
25 WIXNETFXEXTENSION	127
26 HIER KOMMT DIE SONNE – DIE ANWENDUNG HEAT	128
26.1 Ordnerstruktur mit vielen Dateien abscannen.....	128
26.1.1 <i>Tutorial einbinden</i>	128
26.2 Abscannen der Ordnerstruktur im Build-Prozess.....	130
26.3 COM-Server und ActiveX-Controls registrieren	131
26.3.1 <i>Was sind COM-Server bzw. ActiveX-Controls?</i>	131
26.3.2 <i>ActiveX-Control hinzufügen</i>	131
26.3.3 <i>COM-Server per Selbstregistrierung registrieren</i>	133
26.3.4 <i>.NET Assembly als COM Interop registrieren</i>	133
26.4 IIS Webseite abscannen	134
26.5 Visual Studio Projekt	134
27 WEITERE SYSTEMEINSTELLUNGEN.....	135
27.1 Einträge in der Systemsteuerung	135
27.2 Umgebungsvariablen definieren	135
27.3 Schriftarten installieren.....	136
27.4 Dienst installieren	137
27.5 Dienst starten/stoppen	138
28 BERECHTIGUNGEN SETZEN	139
28.1 Setzen der Berechtigung über das Permission-Element	139
28.1.1 <i>Verzeichnisberechtigung setzen</i>	139
28.1.2 <i>Dateiberechtigung setzen</i>	141
28.1.3 <i>Registry-Berechtigung setzen</i>	142
28.2 Setzen der Berechtigung über das PermissionEx-Element	143
28.2.1 <i>Die Security Descriptor Definition Language (SDDL)</i>	143
28.2.2 <i>Verzeichnisberechtigung setzen</i>	143
28.2.3 <i>Dateiberechtigung setzen</i>	144
28.2.4 <i>Registry-Berechtigung setzen</i>	144

28.2.5 Berechtigung für Dienst setzen	144
28.3 Berechtigung über PermissionEx von der WixUtilExtension.....	144
28.3.1 Verzeichnisberechtigung setzen.....	144
28.3.2 Dateiberechtigung setzen.....	145
28.3.3 Registry-Berechtigung setzen	145
28.3.4 Berechtigung für Dienst setzen	145
29 BOOTSTRAPPING MIT BURN	146
29.1 Das Burn-Projekt	146
29.2 Burn-Variablen	148
29.2.1 Built-in Variables.....	148
29.2.2 Eigene Variablen definieren	151
29.3 Installationsbedingung.....	151
29.3.1 Installationsbedingung über das Bundle definieren.....	151
29.3.2 Installationsbedingung über die BAL-Erweiterung erstellen.....	152
29.3.3 Registry auslesen für die Installationsbedingung.....	152
29.4 Bundles updaten	153
29.5 Der PackageCache von Burn.....	154
29.6 Setupkäste einbinden.....	154
29.6.1 MSI-Setups einbinden	154
29.6.2 Eine EXE aus dem Bundle starten	155
29.6.3 Patches installieren	156
29.6.4 Betriebssystem-Updates installieren	157
29.7 Packages vom Internet herunterladen	157
29.8 RollbackBoundary	158
29.9 Die Setupstruktur organisieren.....	159
29.10 Das PackageGroup-Element	160
30 BURN-USERINTERFACE MIT C# ERSTELLEN.....	161
30.1 Benutzerinterface-Extension für Burn erstellen	161
30.2 Das Benutzerinterface im Bundle referenzieren	163
30.3 Grundlegende Strukturen erstellen	163
30.4 Die Model-Klasse	165
30.4.1 Die Model-Klasse erstellen.....	165
30.4.2 Der Konstruktor.....	165
30.4.3 Verbindungsmethoden	165
30.4.4 Code der BurnUiModel-Klasse	167
30.5 Die ViewModel-Klasse	168
30.5.1 Hilfreiche Libraries einbinden	168
30.5.2 Die ViewModel-Klasse erstellen	168
30.5.3 Deklaration der Variablen	168
30.5.4 Der Konstruktor.....	170
30.5.5 Definition der Event-Handler	171
30.5.6 Code der BurnUiViewModel-Klasse	172
30.6 Die View-Klasse für die Erstinstallation.....	175
30.6.1 Der Aufbau der XAML-Datei.....	175
30.6.2 Die Code-Behind-Datei.....	176
30.7 Die View-Klasse für die Maintenance-Installation	176
30.7.1 Der Aufbau der XAML-Datei.....	176
30.7.2 Die Code-Behind-Datei.....	177
30.8 Userinterface von Burn erweitern.....	177
30.8.1 Eingaben an das MSI weitergeben	177
30.9 Setupvoraussetzungen installieren	178
30.10 Burn-Prerequisit-Dialog anpassen	179
31 SIGNIEREN VON WINDOWS INSTALLER SETUPS.....	180
31.1 Signieren des Setups	180

31.2 Das Tool Insignia.....	180
31.3 Zertifikat zu Testzwecken erstellen	180
31.4 Konvertierung der Zertifikatdateien	183
31.5 Zertifikat installieren	183
32 TREIBERINSTALLATION MIT DIFXAPP	184
32.1 32-Bit- und 64-Bit-Treiber im selben Projekt installieren.....	185
33 XML-DATEIEN ÄNDERN	186
33.1 Attribut-Werte in XML-Datei ändern	186
33.2 Neue Elemente in XML-Datei eintragen.....	188
34 WIXUTILEXTENSION NÄHER BETRACHTET	189
34.1 Benutzer erstellen	189
34.2 Ordnerfreigabe einrichten.....	190
34.3 Verzeichnisse rekursiv löschen	191
35 LOGGING.....	193
35.1 Logdatei erstellen	193
35.2 Logdatei-Einträge richtig interpretieren	194
35.2.1 Client- und Server-Prozess	194
35.2.2 Feature- und Komponenten-Status	194
35.2.3 Aktionen.....	195
35.2.4 Properties	195
35.2.5 Gruppenrichtlinien.....	195
35.3 Analyse der Logdatei mit WiLogUtil	196
36 TRANSFORMATIONEN	197
36.1 Transformation mit Orca erstellen.....	197
36.2 Sprachtransformation erstellen	198
36.2.1 Setup in zwei Sprachen erstellen	198
36.2.2 Transformation über torch.exe erstellen.....	198
36.2.3 Transformation einbetten.....	199
36.2.4 Automatische Wahl der MST anhand der Betriebssystemsprache	199
36.2.5 Das Erstellen der Transformation in den Build-Prozess einbauen.....	200
37 MEHRERE INSTANZEN EINES SETUPS INSTALLIEREN	201
38 UPDATES	202
38.1 Update-Typen.....	202
38.1.1 ProductCode, PackageCode und UpgradeCode	202
38.2 Small Update	203
38.3 Minor Updates	203
38.4 Major Update	206
39 PATCHWORK.....	208
39.1 Arbeitsweise des Patches	208
39.2 Vorbereitung des Patches	210
39.3 Patch installieren	211
39.4 Patch über Patch-Creation-Properties-Datei (PCP-Datei) erstellen	211
39.4.1 Allgemeine Vorgehensweise	211
39.4.2 PCP-Datei erstellen	211
39.4.3 Properties der PCP-Datei	214
39.4.4 Patch erstellen	214
39.5 Patch über Pyro.exe und MSI erstellen.....	214
39.6 Patch über Pyro.exe und wixpdb-Datei erstellen	215
39.7 Fallbeispiele PCP-Patch.....	216
39.7.1 Erster Patch	216
39.7.2 Patch baut auf vorhergehendem Patch auf.....	216

39.7.3 Patch ersetzt vorhergehenden Patch (kumulativer Patch)	217
39.7.4 Patch setzt auf vorhergehendem Patch und Major-Update-MSI auf.....	218
39.7.5 Erstellung eines Multi-SKU-Patches.....	219
39.8 User-Account-Control-Patching	220
40 WINDOWS-FIREWALL KONFIGURIEREN	222
41 MERGE-MODULE.....	225
41.1 Merge-Module einbinden.....	225
41.2 Merge-Module erstellen.....	227
41.2.1 Abhängigkeiten.....	227
41.2.2 Festes Zielverzeichnis definieren	228
41.2.3 Dynamische Zielverzeichnisse definieren	229
41.2.4 Das Merge-Modul im Überblick	229
42 DAS WIX-PROJEKT AUF EINEM BUILD-SERVER ERSTELLEN	230
42.1 Buildprozess ohne installiertem WiX-Toolset.....	230
42.2 Buildprozess über MsBuild.exe starten.....	230
42.3 Variable dem Buildprozess übergeben	231
43 MSI IN WXS KONVERTIEREN.....	233
44 ANHANG	234
A. Standard-Verzeichnisvariablen des Windows Installers	234
B. Projekt-Referenz-Variablen	236
C. Windows Installer Versionen	237
D. WiXUI-Dialoge	238
E. Tools des WiX-Toolsets	244
F. Index.....	254

1 Einleitung

WiX ist ein Toolset, das Windows-Installationspakete aus XML-Quellcode erstellt. Das Toolset bietet eine Befehlszeilenumgebung, die Entwickler entweder in ihre herkömmliche Makefile-Buildprozesse oder in die neuere MSBuild-Technologie integrieren können. Innerhalb einer Entwicklungsumgebung - wie Microsoft Visual Studio oder SharpDevelop - kann das WiX-Toolset verwendet werden, MSI-Setup-Pakete zu erstellen.

WiX ist ein Open-Source-Projekt, das von Microsoft initiiert wurde und derzeit hauptsächlich von Rob Mensching vorangetrieben wird.

Das Toolset wurde mit .NET 3.5 in C# programmiert welches natürlich installiert sein muss, um das Toolset ausführen zu können. Dies gilt aber nur für das Toolset an sich. Für die Installationspakete, die mit dem Toolset erstellt werden, bedarf es keinerlei zusätzlicher Systeme oder Software, um diese Pakete auf dem Zielrechner installieren zu können. Es können möglicherweise einige zusätzliche Spezialanwendungen (Merge Modules, Patches) erforderlich sein, das aber nur auf dem Entwicklungsrechner. Der Client kommt allein mit den fertigen und eigenständigen Installer-Paketen aus.

Es besteht eine Community von WiX-Entwicklern und -Benutzern in einer thematisierten Mailingliste. Die Mailingliste wird von SourceForge betreut. Dies ist die Stelle, bei der man Rat und Informationen zu jedem Aspekt von WiX bekommt. Falls man schon einen Account auf SourceForge hat, ist es auch möglich, der Website beizutreten.

Die Entwicklung einer Anwendung abzuschließen bedeutet noch lange nicht, die Entwicklung zu Ende gebracht zu haben. In den letzten Jahren haben sich Benutzer daran gewöhnt, eine vollwertige und komplett Setup-Lösung zu erhalten — und da das Setup das Allererste ist, das die Endnutzer sehen, kann die Wichtigkeit eines Setups nicht genug betont werden.

Herkömmliche Setup-Tools benutzen eine programmatische und skriptbasierte Methode, um die unterschiedlichen Schritte (wie z. B. Daten kopieren, Registry-Einträge erstellen, Gerätetreiber und Services zum Starten) der zu installierenden Anwendung auf dem Zielrechner zu beschreiben. Der Windows Installer folgt einer anderen Philosophie: Bei ihm werden alle Schritte mithilfe einer Datenbank beschrieben.

Anders als viele andere Programme (wie z. B. InstallShield) beschreibt das Toolset das Setup über eine Art Programmiersprache. Perfekt in Visual Studio integriert, benutzt es eine Textdatei (die auf dem immer populärer werdenden XML-Format basiert), um alle Elemente des Installationsprozesses zu beschreiben. Das Toolset hat einen Compiler und einen Linker, die das Setup-Programm genauso erstellen, wie ein herkömmlicher Compiler die Anwendung aus den Quelldateien erstellt. Daher kann WiX ganz einfach zum Teil eines automatisierten Build-Prozesses gemacht werden.

Ein weiterer Vorteil von WiX ist, dass das Setup bereits von Anfang an in den Entwicklungsprozess mit eingeschlossen werden kann. Normalerweise werden Setup-Programme erst geschrieben, wenn die Hauptanwendung bereits fertiggestellt ist – oft sogar von unterschiedlichen Entwicklern. Diese Methode erfordert ein Integrationsdokument, in dem alle Informationen über alle Ressourcen eingetragen werden müssen. Bei WiX kann jeder Entwickler den Teil, den er selbst programmiert mit allen notwendigen Informationen in die XML-Datei eintragen und dokumentiert so bereits zu einem sehr frühen Zeitpunkt alle notwendigen Ressourcen. Wird WiX in den automatisierten Build-Prozess mit einbezogen, steht zu jedem Zeitpunkt eines Entwicklungsprozesses ein fertiges Setup bereit.

Da der Lernprozess relativ komplex und zeitintensiv ist und sich der Entwickler mit den Einzelheiten der zugrunde liegenden Windows Installer Technologie beschäftigen muss, hilft dieses Buch dem Entwickler.

1.1 Warum Windows Installer

Seit der Einführung von Windows 2000 sind Setups ein ganz selbstverständlicher Bestandteil des Betriebssystems. Dabei werden die Installationen vom Windows Installer-Service ausgeführt, der Funktionen wie z. B. selbstreparierende Anwendungen und On-Demand-Installationen unterstützt.

1.1.1 Möglichkeiten des Windows Installers

- Selbstreparierende Anwendungen sind durch die MSI API möglich. Wenn Dateien der Anwendung versehentlich gelöscht wurden, kann die betroffene Anwendung (oder auch Active Desktop bzw. Active Directory) über die API des Windows Installers diese Komponenten nachinstallieren und so die Anwendung reparieren.
- Mithilfe von Advertising braucht man Programmmodulen erst dann zu installieren, wenn sie tatsächlich benötigt werden. So kann z. B. das Hilfesystem nachgeladen werden, wenn der Benutzer die Hilfefunktion aufruft.
- Über die Veröffentlichung einer Anwendung kann der Administrator einem Benutzer die Installation einer bestimmten Software anbieten. Diese Software kann der Anwender dann über die Softwaresteuerung installieren. Zusätzlich werden veröffentlichte Applikationen installiert, wenn der Anwender über den Explorer eine Datei anklickt, deren Dateiverknüpfung mit der veröffentlichten Anwendung verknüpft ist (so wird z. B. Microsoft Word nachinstalliert, wenn der Anwender auf eine .docx-Datei klickt).
- Über Zuweisung kann der Administrator bestimmen, dass Anwendungen automatisch bei bestimmten Benutzern im Advertise-Modus installiert werden. Dieser Vorgang läuft wie folgt ab: Der Benutzer meldet sich bei seiner übergeordneten Domäne an. Dort werden die Grupperichtlinien des entsprechenden Benutzerprofils ausgelesen. In diesem Profil ist dann das zu installierende MSI-Paket vermerkt, woraufhin der Windows Installer dieses Paket installiert.
Da das Paket im Advertise-Modus installiert wird, merkt der Anwender von diesem Vorgang nicht viel. Es werden nur die Startmenü-Einträge und die Dateiverknüpfungen installiert. Die eigentliche Anwendung wird erst installiert, wenn sie benötigt wird.
- Mit Merge-Modulen steht eine standardisierte Technologie zur Verfügung, über die Laufzeit-Bibliotheken verteilt werden können. Der Installationsentwickler kann diese Module als „Blackbox“ betrachten und sie auf einfache Art und Weise in das Setup einbinden.
- Windows Sicherheitsprobleme werden minimiert, da die Installationen von einem System-service bearbeitet werden. Deshalb ist es nicht weiter zwingend erforderlich, dass der Anwender Admin-Rechte hat, um eine Anwendung korrekt zu installieren.
- Über die Installationsaffinität kann der Administrator bestimmen, wo sich die Anwendungsdateien zur Laufzeit befinden. Anstatt alle Anwendungsdateien auf die Zielmaschine zu installieren, kann man einige oder alle Anwendungsdateien von einem Netzwerklaufwerk oder von einer CD-ROM starten.
- Rollback-Änderungen an der Zielmaschine werden bei der neuen Technologie als Transaktionen durchgeführt. Somit kann das System nach einem abgebrochenen Setup in seinen ursprünglichen Zustand zurückgesetzt werden.

1.1.2 Probleme mit herkömmlichen Installationen

Die Anwender sahen sich bei herkömmlichen Installationsprogrammen zahllosen Problemen ausgesetzt, die wiederum die Total Cost of Ownership (TCO) der Systeme erhöht haben. Zu diesen Problemen gehören:

- **Gescheiterte Installationen:** Eine gescheiterte Installation kann Bestandteile (Dateien, Registry-Einträge usw.) hinterlassen, die das System in einen instabilen oder gar unbrauchbaren Zustand versetzen. In herkömmlichen Setup-Programmen gab es keine standardisierten Rollback-Features für gescheiterte Installationen.
- **Zerstörte Anwendungen:** Anwender konnten wichtige Dateien (auch Schlüsseldateien genannt, z. B. DLLs) löschen und damit die Applikation funktionsuntüchtig machen.
- **Probleme beim Deinstallieren:** Bei herkömmlichen Setups konnte es passieren, dass nicht alle Bestandteile einer Anwendung beseitigt wurden. Andererseits konnte es zu Problemen kommen, wenn Dateien und Registry-Einträge von mehreren Anwendungen gemeinsam genutzt wurden. Dabei wurden nämlich solche Dateien bisweilen versehentlich entfernt, obwohl sie von anderen Anwendungen noch benötigt wurden.
- **Administration:** Es war oft schwierig, Anwendungen auf eine große Anzahl von Kundenmaschinen zu verteilen. Administratoren mussten selbst bei jeder Maschine vor Ort sein. Es gab keine Standardmethode, Anwendungen über das LAN zu verteilen.

All diese Probleme werden mit der Verwendung des Windows Installers weitgehend beseitigt.

2 Vorbereitung der Umgebung

2.1 Installation des WiX-Toolsets

Das WiX-Toolset kann von Sourceforge unter folgender URL heruntergeladen werden:

<http://wix.sourceforge.net/>

Zum Zeitpunkt, als diese Unterlagen erstellt wurden, war die Version 3.10 release. Also haben wir die Datei WiX310.exe heruntergeladen. Die Version 3.10 enthält das Programm **Votive**, das in Visual Studio Syntaxhervorhebung (syntax highlighting) und IntelliSense für WXS-Dateien bereitstellt.

Wir gehen davon aus, dass das Visual Studio bereits auf dem Arbeitsplatz installiert ist und können daher das Setup durch einen Doppelklick ausführen.



Wenn das Setup erfolgreich ausgeführt worden ist, findet man das Framework unter folgendem Verzeichnis:

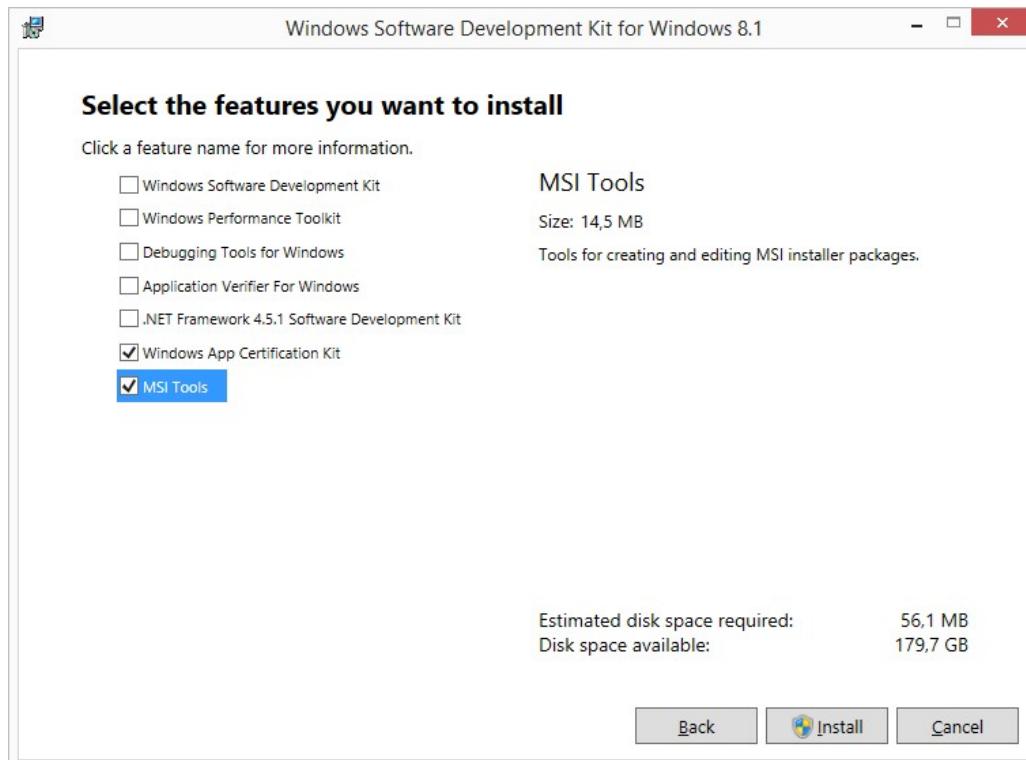
`C:\Program Files\WiX Toolset v3.10`

Im Unterverzeichnis *BIN* stehen dann alle zum Kompilieren von WiX-Dateien notwendigen Programme bereit. Im Verzeichnis *DOC* befinden sich die Onlinehilfen des Windows Installers und des Toolsets selbst und unter *SDK* sind dann noch weitere Programme zu finden, um z. B. selbst extrahierende EXE-Dateien aus dem Setup zu erstellen.

2.2 Installation des Windows Installer SDKs

Da das WiX-Toolset MSI-Setups erstellt, ist es sinnvoll, auch das **Windows Installer-SDK** von Microsoft zu installieren. Das Windows Installer SDK ist Bestandteil des Windows-Software-Development-Kits (SDK), das von der Microsoft-Webseite <http://msdn.microsoft.com> heruntergeladen werden kann. Hier erhält man nützliche Tools und Beispielskripte, um Windows Installer Setups zu bearbeiten.

Nachdem das Windows-Software-Development-Kit heruntergeladen und gestartet ist, wählen wir das Feature „MSI-Tools“ an. Wenn wir unsere Dateien später auch noch signieren wollen, dann ist das Feature „Windows App Certification Kit“ auch noch sinnvoll:



Nachdem das Windows-Software-Development-Kit installiert ist, findet man das Windows Installer SDK unter folgendem Verzeichnis:

C:\Program Files (x86)\Windows Kits\8.1\bin\x86\

2.3 Installation von Orca

Das Windows Installer SDK enthält ein wichtiges Tool namens **Orca**, mit dem MSI-Dateien geöffnet und bearbeitet werden können und mit dem wir als Setup-Entwickler täglich arbeiten werden. Orca wird im SDK-Verzeichnis als MSI-Setup geliefert und muss separat installiert werden.

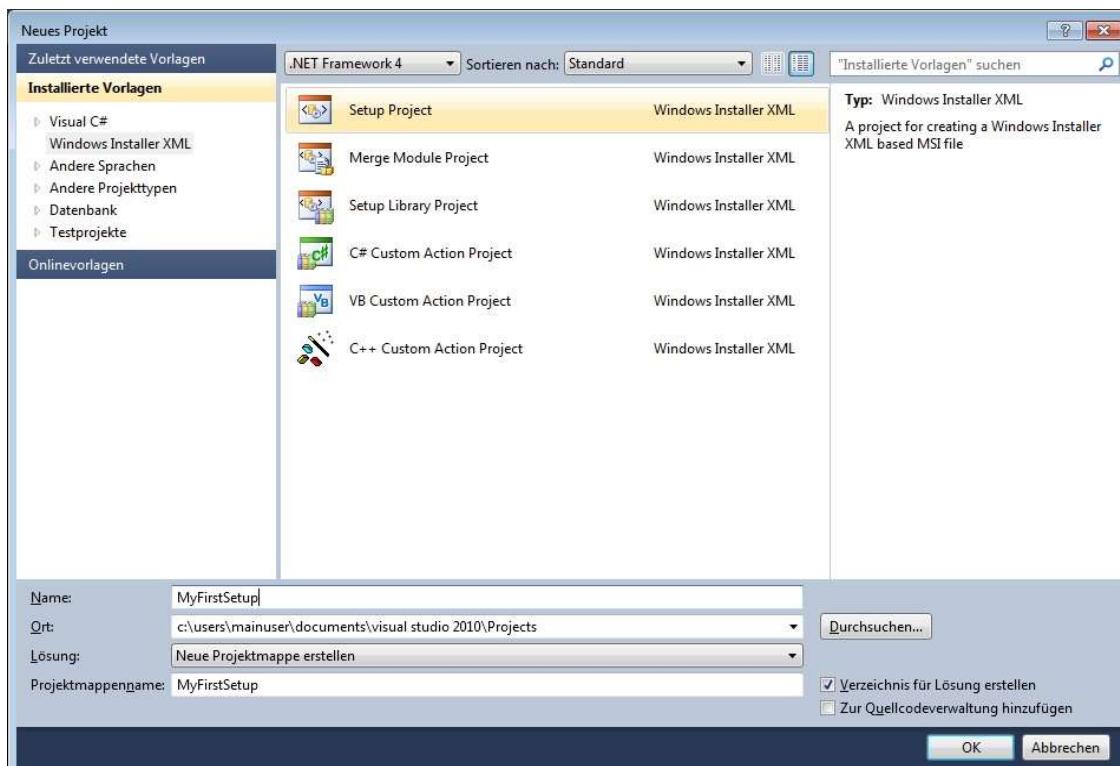
Nach der Installation von Orca stehen uns alle nötigen Tools bereit und wir könnten nun mit der Erstellung unseres ersten Setups beginnen.

3 Das erste Setup entsteht

Nachdem wir nun unsere Umgebung vorbereitet haben, können wir zur Tat schreiten und unser erstes Setup erstellen. In unserem ersten WiX-Beispiel werden wir eine sehr einfache Anwendung installieren. Die Anwendung besteht aus nur einer ausführbaren Datei. Die Datei wird in einen anwendungsspezifischen Ordner kopiert. Zusätzlich möchten wir, dass eine Verknüpfung im Startmenü und eine auf dem Desktop erstellt wird.

So einfach dieses Installationspaket auch sein mag, so wird das Windows Installer Setup über alle Funktionen eines richtigen MSI-Setups verfügen, inklusive der automatischen Aufnahme des Programms in die **Systemsteuerung ▶ Programme hinzufügen oder entfernen**. Um sicherzustellen, dass der Windows Installer unser Programm lokalisieren kann, sollten wir einige Identifikationsmaßnahmen treffen.

In Visual Studio erstellen wir ein neues Projekt und wählen dort die Vorlage „Windows Installer XML“ aus. Wir erstellen dort ein „Setup Project“ mit dem Namen MyFirstSetup:



3.1 Globally Unique Identifier (GUID)

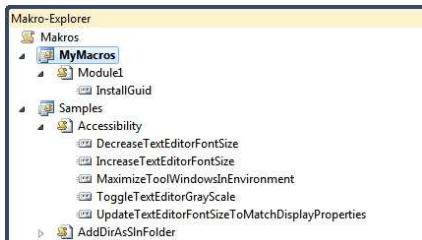
Obwohl alle Anwendungen eine leicht zu merkende Bezeichnung haben, braucht der Windows Installer eine eindeutigere Identifikation in Form eines **GUIDs** (Global Unique Identifier). GUIDs kann man über eine Windows-API-Funktion generieren lassen. Jeder GUID ist garantiert einzigartig – egal, wer und wo sonst noch GUIDs auf dieser Welt erstellt.

Es gibt viele Programme, mit denen man GUIDs erstellen kann. Das Betriebssystem stellt die notwendigen Win32-Funktionen (*CoCreateGuid* und *StringFromCLSID*) bereit, über die Anwendungen GUIDs erstellen können. Alternativ können viele Editorprogramme und integrierte Entwicklungsumgebungen auf Anfrage eine neu generierte GUID in den Quellcode einfügen.

► **Hinweis:** Der Windows Installer arbeitet nur mit großgeschriebenen GUIDs. Sind in einem GUID Kleinbuchstaben enthalten, wandelt der Compiler diese in Großbuchstaben um.

3.1.1 GUID-Generator (bis Visual Studio 2012)

Da wir bei WiX relativ viele GUIDs benötigen, ist es sinnvoll, sich ein Makro zum Erstellen von GUIDs zu definieren. Hierzu öffnet man den Makro-Explorer (Extras ► Makros ► Makro-Explorer) und öffnet dort unter MyMacros ein Modul (hier Module1):



Im sich dann öffnenden Editor gibt man dann folgende Zeilen ein:

```
Public Module Module1
    Sub InstallGuid()
        Dim objTextSelection As TextSelection
        objTextSelection = CType(DTE.ActiveDocument.Selection(), EnvDTE.TextSelection)
        objTextSelection.Text = System.Guid.NewGuid.ToString("D").ToUpperInvariant
    End Sub
End Module
```

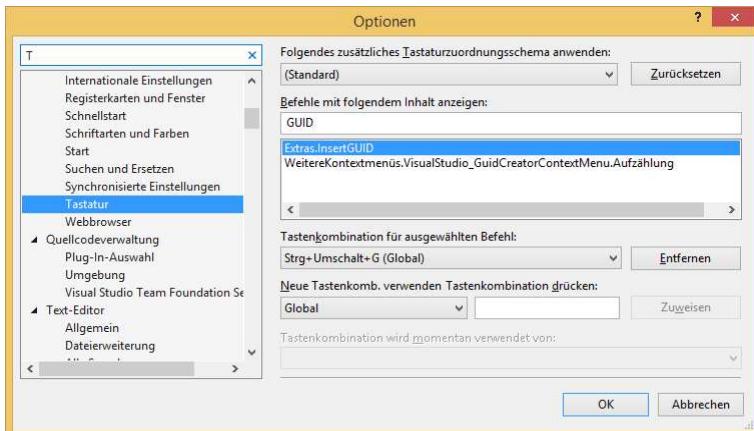
Nachdem man die Zeilen gespeichert hat, kann der GUID-Generator über den eingeblendeten Makro-Explorer (Extras ► Makros ► Makro-Explorer) via Doppelklick verwendet werden.

► **Hinweis:** Die GUID wird an der aktuellen Cursorposition im WiX-Skript eingefügt. Diese Vorgehensweise gilt für Visual Studio bis zur Version 2012.

3.1.2 GUID-Generator (Visual Studio 2013)

Um eine neue GUID in Visual Studio 2013 zu erstellen, verwenden wir die Visual-Studio-Extension „VisualStudio 2013-GuidCreator.vsix“. Diese finden wir auf der Schulungs-DVD im Verzeichnis „VisualStudio 2013-GuidCreator“. Mit einem Doppelklick wird die Installation gestartet. Wenn die Installation von VisualStudio 2013-GuidCreator abgeschlossen ist, muss Visual Studio neu gestartet werden. Im Visual Studio kann man nun über Extras ► Insert Guid eine Guid erstellen.

Am einfachsten weisen wir nun diesem Menüpunkt eine Tastenkombination zu. Das machen wir im Menü Extras ► Optionen ► Umgebung ► Tastatur. Dort suchen wir nach GUID und können dann den Befehl „Extras.InsertGUID“ einer Tastenkombination (hier: Strg+Umschalten+G) zuweisen:



► **Hinweis:** Die GUID wird an der aktuellen Cursorposition im WiX-Skript eingefügt!

3.2 Neue GUIDs erstellen

Für den Anfang braucht man drei IDs, einen für das Produkt (den sogenannten *Product-Code*) einen für das Installationspaket (den sogenannten *Package-Code*) und einen für das Update-Verhalten (den *Upgrade-Code*). Man sollte unbedingt darauf achten, diese GUIDs für jedes neue Setup-Projekt zu generieren:

```
<?xml version="1.0" encoding="UTF-8"?>
<WiX xmlns="http://schemas.microsoft.com/WiX/2006/wi">
  <Product Id="YOURGUID" Name="MyFirstSetup" Language="1033"
    Version="1.0.0.0" Manufacturer="SD" UpgradeCode="YOURGUID">
    <Package Id="*" InstallerVersion="200" Description="My very first setup"
      Comments="Comment of MyFirstSetup" SummaryCodepage="1252" Compressed="yes"/>
    ...
  </Product>
</WiX>
```

► **Hinweis:** Alle GUIDs in diesen Unterlagen sind ungültig und werden als YOURGUID eingetragen. Dies bedeutet, dass die Beispiele nicht ohne Weiteres kompiliert werden können, ohne vorher eigene GUIDs zu generieren (man würde beim Kompilieren die Meldung “fatal error CNDL0027: The ‘Id’ attribute has an invalid value according to its data type” erhalten).

Produktnamen und Beschreibungen kann man natürlich selbst wählen. Für die Erkennung des Attributes **Version**, sollte man das Standardformat *major.minor.build* benutzen. Der Windows Installer wird ein eventuelles viertes Feld ignorieren.

Bekanntlich ist XML ziemlich liberal in Bezug auf Formate. Einzüge und leere Zeilen können auf Wunsch eingesetzt werden. Man sollte alle Attributwerte in Anführungszeichen einfügen, hat aber – je nach Wunsch – die Wahl zwischen einfachen und doppelten. Dadurch ist es sehr einfach, einen Wert, der mit einem Anführungszeichen versehen ist, zu definieren.

Das MSI-File kann sowohl UTF-8 als auch das ANSI-Format benutzen. Wenn man nichts anderes außer den üblichen ASCII-Schriftzeichen oder den Akzentbuchstaben benötigt, ist die Windows-1252-Einstellung (dargestellt in unserem Beispiel) durchaus ausreichend. Wenn man aber eine größere Zahl von unterschiedlichen Zeichensätzen im Interface braucht, sollte man auf UTF-8 umschalten und die entsprechenden Language- und Codepage-Numbers (siehe Anhang) einsetzen.

Benötigt man im MSI-Interface zum Beispiel Japanisch, dann sieht der XML-Code so aus:

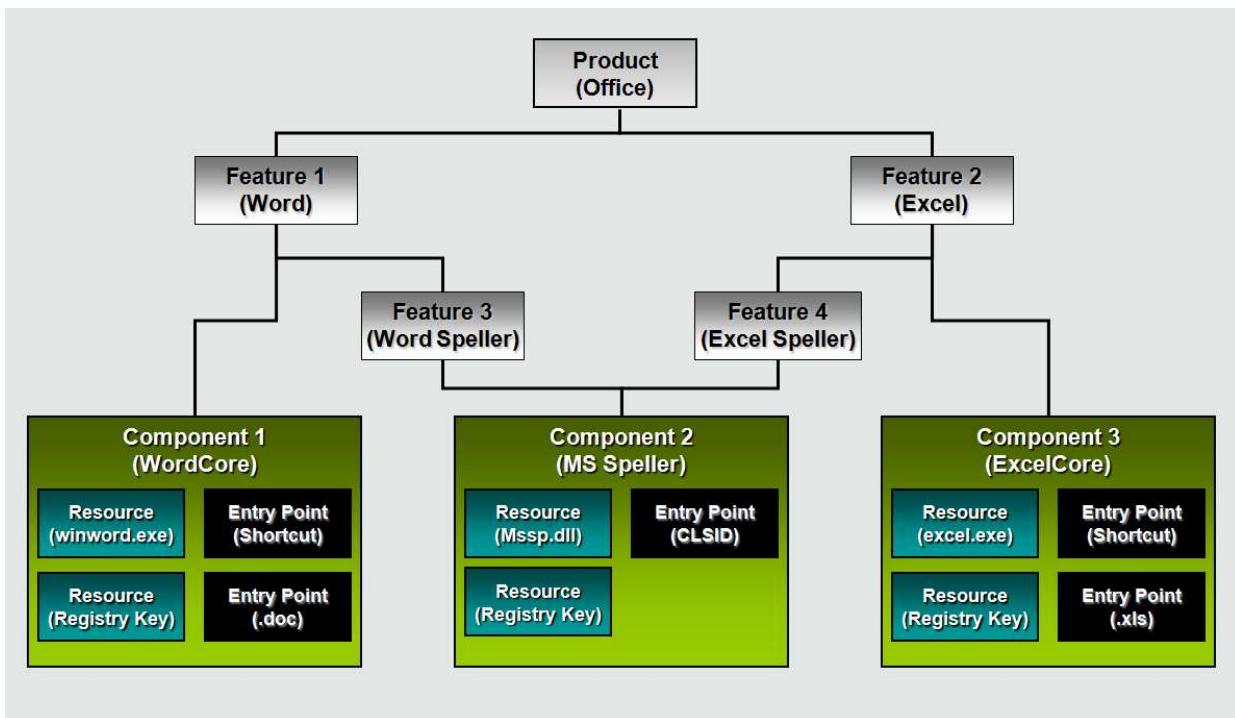
```
<Product Id="YOURGUID" Name="MyFirstSetup" Language="1041" ...="">
  <Package Id="*" InstallerVersion="200" Languages="1041" SummaryCodepage="932"/>
```

Die Angaben im Element „Package“ werden in den Header des MSI-Setups geschrieben und können über den Explorer (Rechtsklick und Eigenschaften) ausgelesen werden. Diese Werte sagen Windows vor dem eigentlichen Start des Windows Installers, welche Windows Installer Version benötigt wird (siehe Attribut *InstallerVersion*, der Wert 200 bedeutet Windows Installer Version 2.00), ob es sich um ein 32-Bit- oder 64-Bit-Setup handelt und welche Codepage Verwendung findet.

► **Hinweis:** Der Stern in der Package-ID bedeutet, dass WiX automatisch bei jedem Build-Prozess eine neue ID generiert.

3.3 Aufbau der Installationsstruktur

Ein für den Windows Installer entwickeltes Installationsprogramm wird in drei Ebenen unterteilt: **Produkt**, **Feature** und **Komponente**.



Das Produkt stellt die höchste Organisationsebene in einem Installationsprojekt dar. Ein Produkt ist normalerweise eine Anwendung und enthält alle erforderlichen Dateien und Registry-Einträge; einige Produkte können mehr als eine Anwendung umfassen.

Aus Sicht des Endanwenders stellt das Feature den kleinsten installierbaren Teil eines Produktes dar. Als Entwickler von Installationsprogrammen überlassen wir normalerweise dem Anwender die Wahl, welche Features installiert werden sollen. In einer Büro-Anwendung wird das Textverarbeitungsprogramm als Feature repräsentiert (im obigen Beispiel das Feature Word). Die Rechtschreibprüfung (Word Speller) wird als Subfeature definiert und verweist auf die entsprechenden Komponenten.

Jedes Feature sollte unabhängig von anderen Feature sein, d. h. ein Feature sollte keine anderen Features derselben Ebene benötigen. Wir können jedoch Features erstellen, die Subfeatures enthalten, mit denen der Anwender steuern kann, welche Zusatzfunktionen installiert werden sollen.

Jedes Feature in einem Produkt besitzt mindestens eine Komponente. Eine Komponente kann man sich wie einen Container vorstellen, der Dateien, Registry-Einträge und Shortcuts enthalten kann. Die Komponenten stellen quasi das Gehirn des Windows Installers dar.

Das Wissen über Komponenten und dazugehörige Regeln sind die wichtigsten Konzepte der Windows Installer Technologie. Dass diese Regeln nicht beachtet wurden, ist Hauptursache für fehlgeschlagene Setups. Daher ist es sehr wichtig, dass man ein gutes Verständnis der Komponente entwickelt.

Komponenten sind für Anwender unsichtbar, stellen aber aus Sicht des Entwicklers den kleinsten installierbaren Teil eines Produktes dar. Jede Komponente enthält Dateien mit ähnlichen Eigenschaften. Alle Dateien in einer Komponente werden z. B. im selben Verzeichnis auf dem PC des Anwenders installiert und müssen für dasselbe Betriebssystem bzw. für dieselbe Sprache gelten; ein Wörterbuchfeature kann verschiedene sprachspezifische Wörterbuchkomponenten enthalten. Zusätzlich zu den enthaltenen Dateien, umfassen Komponenten häufig Registry-Einträge und andere Systemdaten.

Eine Komponente sollte nur Elemente enthalten, die zusammengehören, da sie immer zusammen installiert oder entfernt werden. Dateien, die nicht zusammengehören, sollte man immer in separaten Komponenten unterbringen. Es ist nicht selten, dass man in WiX-Projekten für jede Datei eine separate Komponente vorsieht.

Jede Komponente kann genau einen **KeyPath** (dies kann entweder eine Datei oder ein Registry-Eintrag sein) definieren. Dem KeyPath einer Komponente kommen besondere Aufgaben zu. Wird die Anwendung über einen sogenannten **Entry Point** aufgerufen (ein Entry Point kann entweder ein Shortcut, eine Dateiverknüpfung oder eine COM-Klasse sein), so überprüft der Windows Installer alle KeyPaths der Komponenten, die sich im selben Feature des Entry Points befinden. Stellt der Windows Installer fest, dass ein KeyPath nicht auf dem PC vorhanden ist, so wird zuerst eine Reparatur des Setups aufgerufen. Erst wenn alle Komponenten von der Reparatur nochmals auf den PC übertragen wurden, wird die Anwendung gestartet. Dieser Mechanismus wird **Selbstreparatur** genannt.

Dies ist aber nicht die einzige Aufgabe eines KeyPaths. Ist der KeyPath einer Komponente eine versionierte Datei, dann prüft der Windows Installer bei der Installation zuerst, ob nicht bereits eine gleichnamige Datei existiert. Ist dies der Fall, vergleicht der Installer die Versionsnummern. Hat die vorhandene Datei eine höhere Versionsnummer, dann entscheidet der Windows Installer, dass er die Datei nicht installieren darf. Da diese Datei der KeyPath der Komponente ist, wird die gesamte Komponente abgewählt. Dies hat zur Folge, dass alle anderen Teile (z. B. Dateien) in derselben Komponente ebenfalls nicht mitinstalliert werden. Der KeyPath definiert somit eine Art Chefrolle.

Dieses Beispiel zeigt bereits, dass man sich schon beim Erstellen des Setups sehr genaue Gedanken machen muss, welche Dateien zusammen in einer Komponente abgelegt werden und welche Datei der KeyPath der Komponente wird.

Microsoft hat deshalb in den Richtlinien für den Windows Installer, den **Windows Installer Best Practices**, definiert, dass für jede Anwendungsdatei (EXE, DLL, OCX) und jede Hilfdatei (HLP, CHM) eine eigene Komponente erstellt und als KeyPath gekennzeichnet werden muss. Zusätzlich sollte nie die gleiche Datei in mehr als einer Komponente vorhanden sein.

WiX prüft die Windows Installer Best Practices bei jedem Build-Vorgang mit der sogenannten **Validierung**. Die Validierung prüft das fertige MSI-Setup anhand von **ICEs** (Internal Consistency Evaluators) und gibt bei einem Verstoß eine entsprechende Warnung oder Fehlermeldung aus.

3.4 Dateien hinzufügen

3.4.1 Medium erstellen

Im nächsten Schritt müssen wir das Medium bestimmen, von dem wir installieren wollen. Das kann über das **MediaTemplate**-Element gemacht werden. Im Normalfall braucht man im Zeitalter von CDs und DVDs kaum Installationsdateien, die mehrere übergreifende Medien benötigen. Aber die Möglichkeit existiert. Falls so etwas gebraucht wird, kann man hier auf unterschiedliche Disks bzw. Kabinettdateien verweisen.

```
<MediaTemplate CompressionLevel="high" EmbedCab="yes" />
```

Das Attribut EmbedCab definiert, ob die **Kabinettdatei** in das MSI-Paket eingebettet oder separat abgelegt werden soll. Ob man die Kabinettdatei in das MSI einbettet oder nicht, hängt in der Regel vom Verteilmedium ab. Soll das Setup über eine CD oder DVD verteilt werden, so kann die Kabinettdatei ohne Probleme extern liegen. Will man das Setup per Download im Internet ablegen, so ist es sinnvoll, die Kabinettdatei in das MSI einzubetten.

► **Hinweis:** Ob die Dateien in eine Kabinettdatei gepackt werden oder nicht, wird über das Attribut Compressed im Package-Element entschieden.

Über das Attribut MaximumUncompressedMediaSize kann eine Maximalgröße der Kabinettdateien festgelegt werden. Über das Attribut CabinetTemplate kann man ein Namensschema angeben, nach dem die Kabinettdateien benannt werden. Alternativ zu MediaTemplate können auch ein oder mehrere **Media** Elemente verwendet werden.

3.4.2 Wurzelverzeichnis (Root-Directory) erstellen

Wie bereits in der Einleitung erwähnt, ist der Windows Installer vom früheren programmatischen Ansatz auf einen erklärenden und beschreibenden Ansatz umgestiegen: Wir beschreiben die hierarchische Struktur unserer Quellordnerstruktur unter Benutzung von hierarchisch geschachtelten XML-Strukturen und erwarten vom Windows Installer, dass diese Struktur auf dem Zielrechner wiederhergestellt wird.

Um die Installationsstruktur abilden zu können, müssen wir mit dem fest definierten Identifier **TARGETDIR** (Wurzelverzeichnis oder auch Root-Directory) beginnen. Dieses Wurzelverzeichnis hat auch einen festgelegten Namen, nämlich SourceDir. Das Wurzelverzeichnis stellt die Basisverbindung zwischen „Woher kommen die Dateien?“ und „Wo müssen die Dateien hin?“ her.

```
<Directory Id="TARGETDIR" Name="SourceDir">
```

Innerhalb dieses Wurzelverzeichnisses gehen wir weiter mit unserer Struktur. Zu beachten ist, dass jede Verzeichnisebene separat definiert wird:

```
<Directory Id="ProgramFilesFolder">
  <Directory Id="CompanyName" Name="MyCompany">
    <Directory Id="INSTALLDIR" Name="MyFirstSetup">
```

In Übereinstimmung mit den Microsoft-Richtlinien werden Programme in ein Unterverzeichnis (hier MyCompany\MyFirstSetup) von **ProgramFilesFolder** installiert. ProgramFilesFolder ist eine vom Windows Installer vordefiniertes Property und zeigt in das Programme-Verzeichnis (also z. B. c:\Program Files) des Zielrechners. Da Directory-Variablen bei der Initialisierung auf den Wert gleichnamiger Properties gesetzt werden, können wir hier Directory-Variablen und Properties gleichsetzen. Das gilt allerdings nur für den Zeitpunkt der Initialisierung!

► **Hinweis:** Verknüpfungen, Desktop-Icons, Benutzereinstellungen usw. haben alle von Windows festgelegte Verzeichnisse. Zu unserer Erleichterung werden die gebräuchlichsten Zielverzeichnisse bereits als Properties vordefiniert. Welche Variablen das sind, kann man im Anhang sehen.

In unserem jetzigen Beispiel werden wir drei dieser Variablen, **ProgramFilesFolder**, **ProgramMenuFolder** und **DesktopFolder** benutzen. Man sollte beachten, dass diese festgelegten Directory-Variablen an vollständige Pfade verweisen. Die ID der Directories sollten mit sinnvollen Namen versehen werden, da wir im gesamten Setup auf diese verweisen werden.

3.4.3 Komponente erstellen

Als nächsten Schritt erstellen wir über das **Component**-Element eine Komponente, welche die Datei und zwei Verknüpfungen aufnehmen wird:

```
<Component Id='IpwiSample.exe' Guid=' YOURGUID' Directory='INSTALLDIR'>
```

Jede Komponente braucht eine eigene ID sowie einen GUID (die WiX-Compiler und -Linker werden Sie warnen, wenn Sie eines dieser beiden zum zweiten Mal verwenden). Dies ist sehr wichtig, da diese GUIDs das einzige Mittel für den Windows Installer sind, den Überblick über die verschiedenen Komponenten zu behalten. Ein Verstoß gegen die Komponentenregel hat fatale Folgen: Ressourcen können während einer Deinstallation verwaist auf dem Rechner zurückbleiben; eine gemeinsame Ressource könnte irrtümlich entfernt werden, während eine andere Anwendung sie noch braucht; die Neuinstallation eines bestehenden Produkts könnte fehlschlagen.

3.4.4 Datei einbinden

Nun binden wir über das **File**-Element eine Datei ein. Das File-Element wird hierbei als Child-Element des Component-Elements - also innerhalb des Component-Elements - definiert:

```
<Component Id='IpwiSample.exe' Guid=' YOURGUID' Directory='INSTALLDIR'>
  <File Id='IpwiSample.exe' Source='.\SourceDir\IpwiSample.exe' KeyPath='yes' />
</Component>
```

Die zu installierende Datei wird über das Source-Attribut angegeben. Der Pfad kann entweder wie hier relativ oder über einen qualifizierten Pfadnamen angegeben werden. Wird der Pfad relativ angegeben, dann bezieht sich der Pfad auf das aktuelle Arbeitsverzeichnis – was beim Erstellen im Visual Studio der Ordner ist, in der die Projektdatei von WiX liegt.

Soll die Datei auf dem Zielsystem einen anderen Namen bekommen, dann kann der Zielname über das Attribut Name angegeben werden. Neben dem eigentlichen Namen kann man die Datei mit mehreren Attributen versehen. Wird z. B. **Vital** auf „no“ gesetzt, informieren wir den Windows Installer darüber, dass die Installation dieser Datei nicht von entscheidender Bedeutung ist. Normalerweise wird das Setup abgebrochen, wenn die Installation einer Datei fehlschlägt. Ist die Datei jedoch mit dem Attribut **Vital="yes"** versehen, kann der Benutzer das Problem ignorieren. Weitere Attribute sind **ReadOnly**, **Hidden**, **System**, die alle dafür sorgen, dass für die Datei das entsprechende File-Attribut gesetzt wird.

Wie bereits weiter oben erwähnt, braucht jede Komponente einen KeyPath. Wir haben entschieden, dass die zu installierende Datei den KeyPath darstellen soll. Deshalb haben wir das Attribut KeyPath auf „Yes“ gesetzt.

3.4.5 Datei-Überschreibungs-Regeln

Der Windows Installer gibt feste Regeln vor, wann eine vorhandene Datei überschrieben wird und wann nicht.

3.4.5.1 Die höchste Versionsnummer gewinnt

Sind beide Dateien, die Datei im Zielverzeichnis und die Datei im Setup, versioniert, so gewinnt grundsätzlich die höchste Versionsnummer. Versionierte Dateien gewinnen immer gegenüber unversionierten Dateien.

Haben beide Dateien dieselbe Versionsnummer, so prüft der Windows Installer auch noch die Produktsprache der Dateien. Hat die zu installierende Datei eine andere Sprache als die Datei, die bereits auf dem Zielverzeichnis vorhanden ist, gewinnt die Datei, die dieselbe Sprache hat wie das Setup. Ist eine Datei sprachunabhängig, so ist das ebenfalls so, als handele es sich um eine andere Sprache.

Unterstützen beide Dateien mit derselben Versionsnummer die Setup-Sprache, dann gewinnt die Datei, die mehr Sprachen unterstützt. Dasselbe gilt auch, wenn keine der Dateien die Setup-Sprache unterstützt.

3.4.5.2 Unversionierte Dateien

Hat weder die Datei im Setup noch die auf dem Zielverzeichnis eine Versionsnummer, so wird nur dann die vorhandene Datei überschrieben, wenn diese nach dem Erstellen auf dem PC nicht mehr verändert wurde (d. h. das Erstellungsdatum ist älter oder gleich dem Änderungsdatum). Dieses Verhalten schützt eine vom Benutzer gefüllte Datenbank davor, dass sie bei einem Reparaturlauf mit einer leeren Datenbank überschrieben wird.

-
- **Hinweis:** Die oben beschriebenen Regeln können über das Property **REINSTALLMODE** verändert werden.
-

3.5 Shortcut erstellen

Shortcuts enthalten neben dem Namen auch andere wichtige Elemente, wie z. B. das Arbeitsverzeichnis, Aufrufparameter sowie die Icon-Spezifikationen.

```
<Shortcut Id="ProgramMenuShortcut" Directory="ProgramMenuDir" Name="My First Sample"
          WorkingDirectory="INSTALLDIR" Icon="IpwiSample.exe" IconIndex="0" Advertise="yes" />
<Shortcut Id="DesktopShortcut" Directory="DesktopFolder" Name="My First Sample"
          WorkingDirectory='INSTALLDIR' Icon="IpwiSampe.exe" IconIndex="0" Advertise="yes" />
```

Obwohl unsere IpwiSample.exe das Icon selbst enthält, verweist das Icon-Attribut hier nicht direkt auf diese Datei, sondern vielmehr auf ein noch zu definierendes **Icon**-Element, das irgendwo in unserem WiX-Skript angegeben werden muss:

```
<Icon Id="IpwiSample.exe" SourceFile=".\\SourceDir\\Program V1.0\\IpwiSample.exe" />
```

Shortcuts können angekündigt oder nicht angekündigt sein. Dies wird über das Attribut Advertised entschieden. Ein nicht angekündigter Shortcut ist ein einfacher Link auf die Datei. Ein angekündigter Shortcut sorgt dafür, dass der Windows Installer die Anwendung überwacht und im Notfall repariert.

Wie man hier schon leicht erkennt, kann ein Setup zu definieren, das mitunter Hunderte von Dateien und Komponenten hat, sehr aufwendig werden. Es gibt zwar im Toolset ein kleines Tool namens Heat.exe (mehr dazu später), aber eine echte Lösung kann nur eine konzeptionelle Änderung der Art und Weise sein, wie Setups erstellt werden.

-
- **Hinweis:** Ein Setup sollte nicht länger als eigenständige Anwendung betrachtet werden, die, nachdem die Hauptanwendung schon fertig ist, schnell auch noch geschrieben wird. Da sich die WiX-Quelldateien und das Toolset nahtlos in die Entwicklungsumgebung integrieren, sollte das Setup parallel zur Entwicklung der Anwendung laufend aktuell gehalten werden. Sobald Sie mit einem neuen Modul anfangen oder einen neuen Registry-Eintrag zu Ihrem Programm hinzufügen, sollten die verknüpften WiX-Quelldateien gleich mit angepasst werden.
-

Auf diese Weise wird das Setup zeitgleich mit der Anwendung fertig und man muss später nicht alle für die Installation notwendigen Dateien und sonstigen Informationen zusammentragen. Da das WiX-Projekt modularisiert werden kann (dazu später mehr), funktioniert dieser Ansatz, egal, ob Sie als einzelner Entwickler oder in einem großen Team an einem Projekt arbeiten.

Aber nun wieder zurück zu unserem WiX-Skript. Wir sind nämlich mit den Shortcuts noch nicht fertig. Wir müssen ja noch die **Directory**-Elemente **ProgramMenuFolder** und **DesktopShortcut** definieren.

Diese definieren wir innerhalb des bereits definierten TARGETDIR-Elements:

```
<Directory Id="TARGETDIR" Name="SourceDir">
  <Directory Id="ProgramMenuFolder" Name="Programs">
    <Directory Id="ProgramMenuDir" Name="MyCompany"/>
  </Directory>
  <Directory Id="DesktopFolder" Name="Desktop"/>
</Directory>
```

Alternativ könnte man auch weiter unten mit dem Element **DirectoryRef** auf TARGETDIR referenzieren und danach die Elemente ProgramMenuFolder und DesktopFolder eintragen:

```
<DirectoryRef Id="TARGETDIR">
  <Directory Id="ProgramMenuFolder" Name="Programs">
    <Directory Id="ProgramMenuDir" Name="MyCompany"/>
  </Directory>
  <Directory Id="DesktopFolder" Name="Desktop"/>
</DirectoryRef>
```

Da wir den Ordner „MyCompany“ im Startmenü bei der Deinstallation auch wieder löschen müssen, ist noch ein **RemoveFolder**-Element in diese Komponente mit aufzunehmen:

```
<RemoveFolder Id="ProgramMenuDir" Directory="ProgramMenuDir" On="uninstall" />
```

Über das **On**-Attribut legen wir fest, wann der Ordner entfernt wird (mögliche Werte sind *install*, *uninstall* und *both*). Beim Kompilieren werden wir später eine Warnung von der Validierung bekommen:

Fehler 1 ICE64: The directory ProgramMenuDir is in the user profile but is not listed in the RemoveFile table.

Diese Warnung können wir jedoch ignorieren, da wir nicht ins Userprofil, sondern nur in das ALLUSERS-Profil installieren.

3.6 Komponente einem Feature zuweisen

Als letzten Schritt müssen wir die Komponente noch einem Feature zuweisen:

```
<Feature Id="ProductFeature" Title="MyFirstSetup" Level="1">
  <ComponentRef Id="IpwiSample.exe" />
</Feature>
```

3.7 Das gesamte Skript im Überblick

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/Wix/2006/wi">
  <Product Id="YOURGUID" Name="MyFirstSetup" Language="1033" Version="1.0.0.0"
    Manufacturer="MyFirstSetup" UpgradeCode="YOURGUID">
    <Package Id="*" InstallerVersion="200" Compressed="yes"/>

    <MediaTemplate CompressionLevel="high" EmbedCab="yes" />
    <Property Id="DiskPrompt" Value="My First Project [1]" />

    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="ProgramMenuFolder" Name="Programs">
        <Directory Id="ProgramMenuDir" Name="MyCompany"/>
      </Directory>
      <Directory Id="DesktopFolder" Name="Desktop"/>
      <Directory Id="ProgramFilesFolder">
        <Directory Id="CompanyName" Name="MyCompany">
          <Directory Id="INSTALLDIR" Name="MyFirstSetup" />
        </Directory>
      </Directory>
    </Directory>
  </Product>
</Wix>
```

```
</Directory>

<Property Id="ALLUSERS" Value="1"/>

<Icon Id="IpwiSample.exe" SourceFile=".\\SourceDir\\Program V1.0\\IpwiSample.exe" />

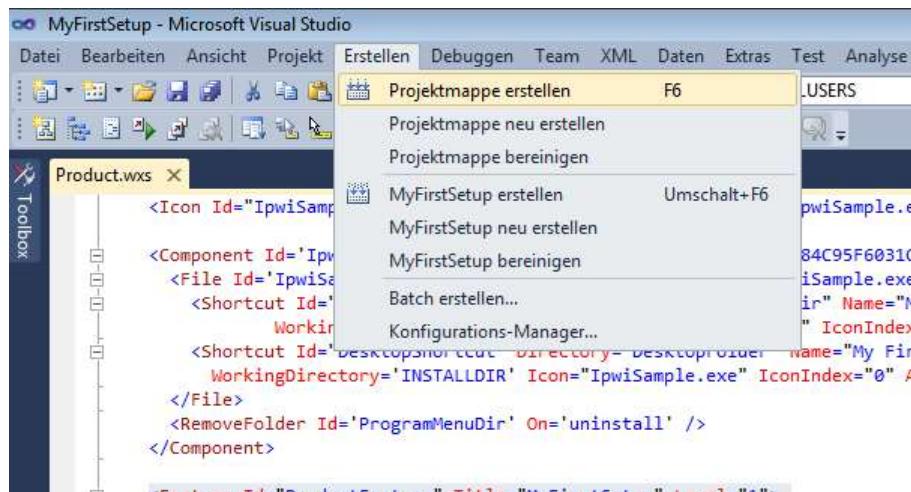
<Component Id="IpwiSample.exe" Guid="YOURGUID" Directory="INSTALLDIR">
  <File Id="IpwiSample.exe" Source=".\\SourceDir\\Program V1.0\\IpwiSample.exe"
    KeyPath="yes">
    <Shortcut Id="ProgramMenuShortcut" Directory="ProgramMenuDir"
      Name="My First Sample" WorkingDirectory="INSTALLDIR"
      Icon="IpwiSample.exe" IconIndex="0" Advertise="yes" />
    <Shortcut Id="DesktopShortcut" Directory="DesktopFolder"
      Name="My First Sample" WorkingDirectory="INSTALLDIR"
      Icon="IpwiSample.exe" IconIndex="0" Advertise="yes" />
  </File>
  <RemoveFolder Id="ProgramMenuDir" Directory="ProgramMenuDir" On="uninstall" />
</Component>

<Feature Id="ProductFeature" Title="MyFirstSetup" Level="1">
  <ComponentRef Id="IpwiSample.exe" />
</Feature>
</Product>
</Wix>
```

► **Hinweis:** Alle Beispiele finden Sie auf unserer Seminar-DVD.

3.8 Setup kompilieren

Nun sind wir so weit, dass wir unser erstes Setup erstellen können.



Das Setup können wir in Visual Studio über den Menüpunkt **Erstellen ► Projektmappe erstellen** kompilieren.

Das erstellte Setup finden wir dann unterhalb der WXS-Dateien im Unterverzeichnis *bin\Debug* bzw. *bin\Release*.

Wenn das Setup nicht über Visual Studio kompiliert werden soll, kann man auch folgende Kommandozeile benutzen:

```
Set Path=%Path%;c:\Program Files\WiX Toolset v3.9\bin  
candle.exe MyFirstSetup.wxs  
light.exe MyFirstSetup.wixobj
```

Das WiX-Tool **candle.exe** generiert zunächst aus den WXS-Dateien WIXOBJ-Dateien, die dann mittels **light.exe** zu einem MSI zusammengebunden werden.

3.9 Setup ausführen

Um den ersten Installer zu testen, klicken Sie einfach darauf. Es wird Sie nicht begrüßen oder irgendwelche Optionen anbieten, es wird nur während einiger Sekunden ein Fortschrittsdialog zeigen. Sobald dies aber ohne Fehler beendet ist, sollten Sie in der Lage sein, die Datei IpwiSample.exe unter folgendem Pfad zu finden:

```
C:\Program Files\MyCompany\MyFirstSetup
```

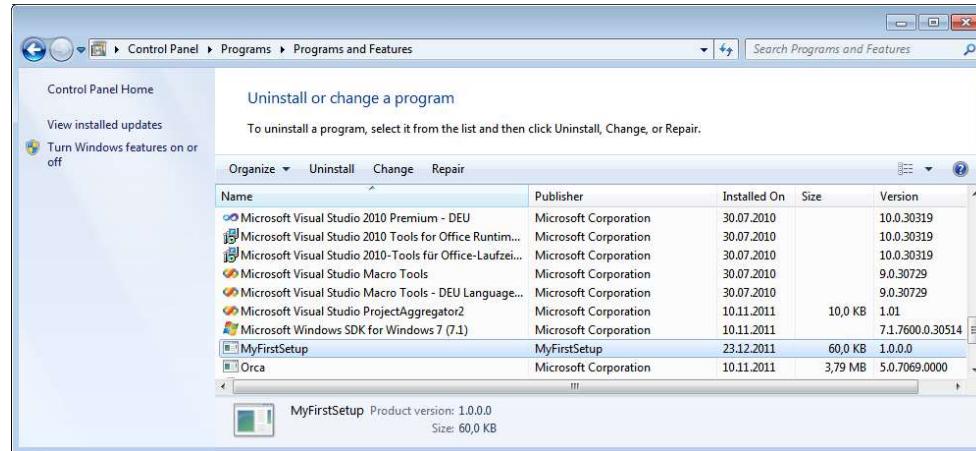
Im Startmenü sollten Sie dann die Anwendung vor sich haben und auf dem Desktop sollte ebenfalls ein Link auf unser Programm eingetragen sein.

Falls bei der Installation Probleme auftauchen – oder nur aus Spaß – können wir den Installer mit eingeschalteter Protokollierung (Logging) starten:

```
msiexec.exe /i MyFirstSetup.msi /l*v MyFirstSetup.log
```

Das Protokoll wird ziemlich lang sein, aber es wird den Fehler, der den Installationsabbruch verursacht, ganz genau festlegen können.

Um diese Anwendung wieder zu entfernen, gehen wir zu **Control Panel ▶ Uninstall a Program**, wählen dort die Anwendung MyFirstSetup an und klicken dann auf den Button Remove:



Die Datei samt Unterordner sollte dann verschwinden. Alternativ kann die Deinstallation natürlich auch über die Kommandozeile aufgerufen werden:

```
msiexec.exe /x MyFirstSetup.msi
```

3.10 Installationsarten des Windows Installers

Anbei eine Übersicht der Installationsarten des Windows Installers, die über die **msiexec.exe** aufgerufen werden:

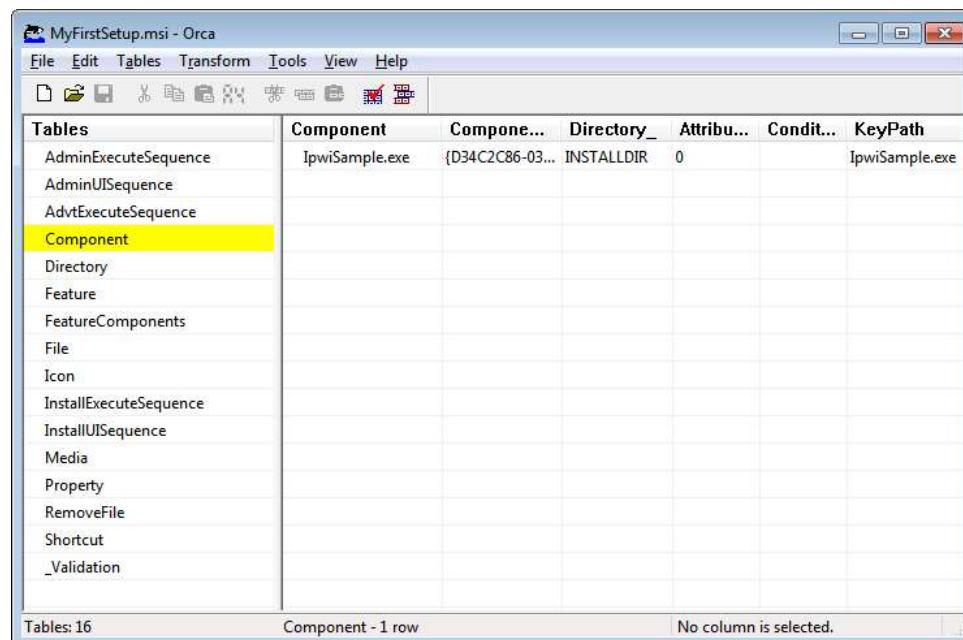
Installationsart	Kommandozeile
Installieren	msiexec.exe /i D:\Example.msi
Deinstallieren	msiexec.exe /x D:\Example.msi
Ausführen eines administrativen Setups	msiexec.exe /a D:\Example.msi
Reparaturmodus ausführen	msiexec.exe /f D:\Example.msi
Ankündigen (der Maschine)	msiexec.exe /jm D:\Example.msi
Ankündigen (dem aktuellen Benutzer)	msiexec.exe /ju D:\Example.msi

Ist das Projekt bereits installiert, akzeptiert msiexec.exe anstatt des Pfades zu einem MSI-Paket auch den **ProduktCode** (siehe Property-Tabelle).

3.11 Setup mit Orca öffnen

Um ein „Gefühl“ für den Windows Installer zu bekommen, werden wir jetzt das Setup einmal mit dem Tool **Orca.exe** ansehen. Über Orca können wir die Windows Installer Datenbank als solche sehen und können dort sogar Änderungen durchführen. Wir werden dort wieder alle Elemente finden, die wir in unserem WiX-Skript definiert haben.

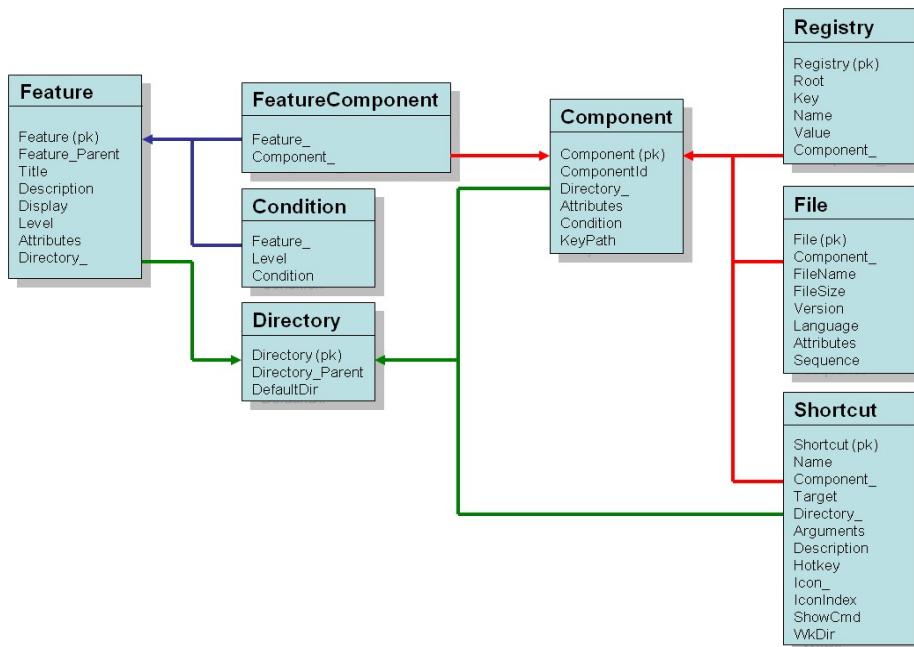
Um Orca zu öffnen, klicken wir mit der rechten Maustaste auf unsere MSI-Datei und wählen dort „Edit with Orca“:



In Orca sehen wir, dass unser Setup sechzehn Tabellen hat.

Die Anzahl der Tabellen kann je nach Verwendung der Funktionen variieren. Es können bis zu hundert und mehr Tabellen werden.

Im folgenden Schaubild kann man nochmals sehen, wie die einzelnen Tabellen zusammenhängen:



4 Pfad zu den Quelldateien bestimmen

Im WiX-Toolset gibt es drei unterschiedliche Methoden, um den Pfad zu den Quelldateien zu bestimmen:

- über die explizite Angabe der Quelldateien (absolut oder relativ)
- über den Dateinamen und das Directory-Element
- über sogenannte Bindepfade

4.1 Compiler, Linker und Binder

Das WiX-Toolset arbeitet wie ein typischer C-Compiler. Zuerst werden die Quelldateien mittels Candle.exe zu Objektdateien kompiliert, danach über den Linker (Light.exe) zum tatsächlichen Ausgangsformat zusammengebunden, wobei das Ausgangsformat bei WiX entweder ein MSI-Package, ein Merge-Module oder eine WiX-Library ist. Das WiX-Toolset hat aber noch eine dritte Phase: Es bindet bzw. packt die zu installierenden Dateien. Das Binden wird auch von der Light.exe übernommen. Der Compiler ignoriert die Quelldateien und fügt nur Referenzen auf diese in die Objektdatei ein. Erst beim Binden werden dann die Quelldateien in die Cabinet-Dateien bzw. den Ausgangsordner kopiert.

Der Binder kann über einen oder mehrere Bindepfade versorgt werden, sucht aber auch im aktuellen Arbeitsverzeichnis nach den Dateien. Bindepfade werden bei der Light.exe durch Angabe des Kommandozeilenparameters -b angegeben. Wird kein Name vor dem Bindepfad angegeben, so wird der Pfad als allgemeiner Bindepfad verwendet. Wir können aber auch den Bindepfaden einen Namen geben. Der Name muss mindestens zwei Zeichen lang sein und wird wie folgt definiert:

```
Light -b MyFolder1=C:\Folder1\ -b MyFolder2=C:\Folder2\...
```

Wie benannte Bindepfade im WiX-Skript referenziert werden, sehen wir weiter unten im entsprechenden Kapitel.

4.2 Binden der Dateien durch das Directory-Element

Verwendet man beim File-Element statt des Source-Attribut das Name-Attribut, dann konstruiert der Compiler einen impliziten Pfad, der auf dem Zielpfad – also der Directory-Struktur basiert.

Betrachten wir einmal folgenden Code:

```
<Directory Id="TARGETDIR" Name="SourceDir">
    <Directory Id="ProgramFilesFolder">
        <Directory Id="INSTALLFOLDER" Name="MyProduct">
            <Component Id="Test.txt" Guid="YOURGUID">
                <File Id="Test" Name="Test.txt" KeyPath="yes"/>
            </Component>
        </Directory>
    </Directory>
</Directory>
```

Gibt man keinen allgemeinen Bindepfad an, so wird die Test.txt-Datei im folgenden Verzeichnis gesucht:

```
<Arbeitsverzeichnis>\SourceDir\MyProduct\Test.txt
```

Der Verzeichnisname „SourceDir“ wird vom Root-Element der Directory-Struktur, also von TARGETDIR, entnommen. Das darunterliegende Unterverzeichnis „MyProduct“ wurde über das Directory-Element INSTALLFOLDER definiert usw.. Gibt man einen allgemeinen Bindepfad an, so ändert sich der Suchpfad wie folgt:

```
<Bindepfad>\ MyProduct\Test.txt
```

Wie wir an unserem Beispiel sehen, setzt der Binder das Quellenverzeichnis anhand des Name-Attributes des Directory-Elements zusammen. Ist das nicht gewünscht, kann der Quellpfad durch Setzen des FileSource-Attributes beim Directory-Element geändert werden:

```
<Directory Id="TARGETDIR" Name="SourceDir">
  <Directory Id="ProgramFilesFolder" FileSource="ProgramFilesFolder">
    <Directory Id="INSTALLFOLDER" Name="MyProduct" FileSource="MyFolder\x86">
      <Component Id="Test.txt" Guid="{57DD0248-88B2-4399-B211-B2BB13EA6CEC}">
        <File Id="Test" Name="Test.txt" KeyPath="yes"/>
      </Component>
    </Directory>
  </Directory>
</Directory>
```

Der resultierende Suchpfad (mit Bindepfad) sieht dann so aus:

`<Bindepfad>\ProgramFilesFolder\MyFolder\x86\Test.txt`

► **Hinweis:** Beim Binden durch das Directory-Element muss die Komponente innerhalb des Directory-Elements stehen. Es reicht also nicht aus, dass die Komponente außerhalb des Directory-Elementes definiert wird und nur auf das Directory verweist. Deshalb wird diese Art der Bindung relativ schnell unübersichtlich.

4.3 Binden der Dateien durch das File-Element

Das Binden der Dateien durch das Source-Attribut des File-Elements haben wir bereits im vorhergehenden Kapitel verwendet:

```
<Component Id='IpwiSample.exe' Guid='YOURGUID' Directory='INSTALLDIR'>
  <File Id='IpwiSample.exe' Source='.\Program\IpwiSample.exe' KeyPath='yes' />
</Component>
```

Diese Bindemethode bietet sich immer dann an, wenn die Directory-Struktur der Quelldateien nicht der Zielstruktur entspricht. Diese Methode ist auch etwas übersichtlicher, da die Komponenten nicht immer innerhalb eines Directory-Elements liegen müssen.

Wie wir im obigen Beispiel sehen, müssen wir im File-Element das Name-Attribut nicht mehr angeben. Der Binder geht davon aus, dass die Zielfile genau so heißen soll wie die Quelldatei. Das muss aber nicht sein. Gibt man trotzdem über das Name-Attribut einen Wert an, dann kann die Quelldatei anders heißen als die Zielfile:

```
<Component Id='IpwiSample.exe' Guid='YOURGUID' Directory='INSTALLDIR'>
  <File Id='IpwiSample.exe' Source='.\Program\IpwiSample.exe' Name='MySample.exe'
    KeyPath='yes' />
</Component>
```

► **Hinweis:** Das Binden durch das File-Element ist stärker als das Binden durch das Directory-Element. Wird eine Komponente innerhalb des Directory-Elements definiert und im zugehörigen File-Element das Attribut Source angegeben, dann wird die Datei im Pfad gesucht, der im File-Element angegeben wurde.

4.4 Binden der Dateien durch benannte Bindepfade

Wie schon erwähnt, werden die benannten Bindepfade über den Light-Parameter *-b* angegeben:

Light -b MyPath=C:\SourceDir\..

Im File-Element verweisen wir hierbei auf die **Bindpath**-Variable in folgender Form:

```
<Component Id='IpwiSample.exe' Guid='YOURGUID' Directory='INSTALLDIR'>
    <File Id='IpwiSample.exe' Source='!(bindpath.MyPath)\x86\IpwiSample.exe'
          KeyPath='yes' />
</Component>
```

Bei Light.exe ist es auch möglich, mehrere Quellpfade für denselben benannten Bindepfad anzugeben:

Light -b MyPath=C:\SourceDir\ -b MyOtherPath=C:\OtherPath\ -b MyPath=C:\OtherFiles\..

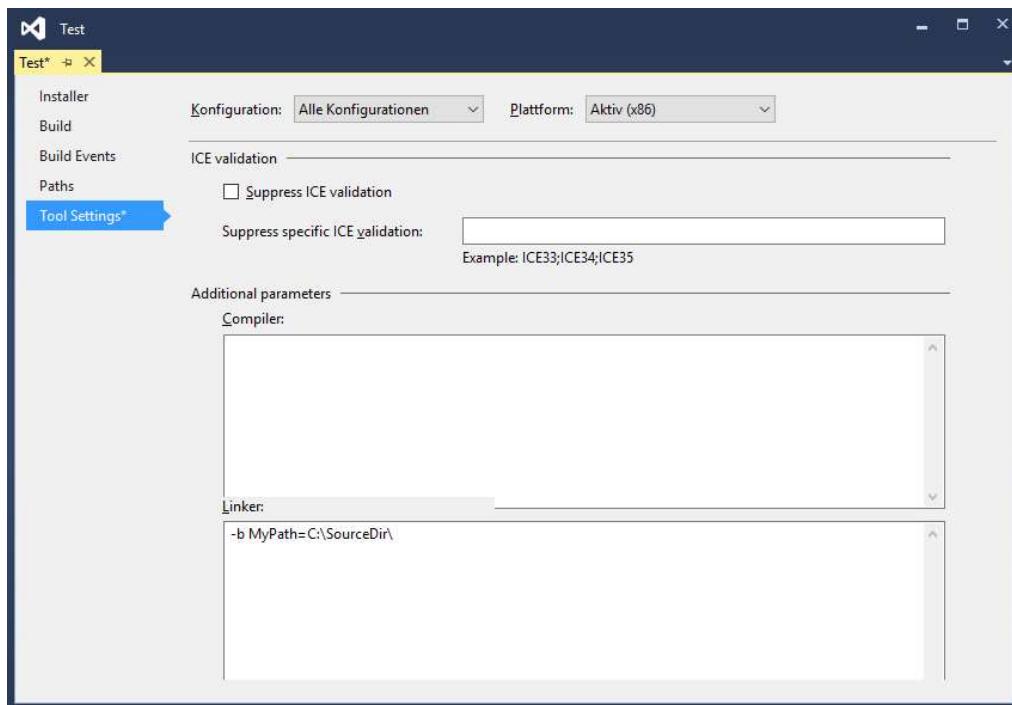
Beim obigen Beispiel würde Light.exe die IpwiSample.exe zuerst in diesem Verzeichnis suchen:

C:\SourceDir\x86\IpwiSample.exe

Wird die Datei dort nicht gefunden, sucht die Light.exe im Verzeichnis:

C:\OtherFiles\x86\IpwiSample.exe

Erstellen wir das Setup innerhalb von Visual Studio (was ja die Regel sein wird), dann können wir die Bindepfade über die Projekt-Einstellungen des WiX-Setups unter dem Reiter *Tool Settings* ► *Linker* setzen:



► **Hinweis:** Steht im Source-Attribut vom File-Element zwischen der Bindepfad-Variablen und dem eigentlichen Dateinamen kein Backslash, dann muss der Dateinamen explizit im Name-Attribut angegeben werden.

5 Bedingungen definieren

5.1 Properties

Der Windows Installer speichert variable Werte in sogenannten Properties. Ein Property ist eine Variable vom Typ String und wird entweder über die Property-Tabelle oder durch Zuweisung eines Wertes erstellt. Properties müssen also nicht explizit deklariert werden.

Über die Properties kann das Verhalten des Windows Installers verändert werden. So wird z. B. über das Property ALLUSERS definiert, ob das Setup für alle Benutzer oder nur für den derzeit angemeldeten Benutzer installiert werden soll. Ebenso setzt der Windows Installer beim Starten eine ganze Reihe an Properties über die das Setup herausfinden kann, auf welchem Betriebssystem das Setup ausgeführt wird, welche Bildschirmauflösung derzeit eingestellt wird usw. Properties, die das Verhalten des Windows Installers beeinflussen bzw. vom Windows Installer beim Start des Setups gesetzt werden, nennt man Standard-Properties. Name und Bedeutung der Standard-Properties kann über die MSI-Hilfe unter dem Stichwort **Property Reference** ermittelt werden.

Grundsätzlich werden zwei unterschiedliche Typen von Properties unterschieden:

- private Properties (private property)
- öffentliche Properties (public property)

Private Properties sind Variablen, die nur für den internen Gebrauch (z. B. Vorbelegung bestimmter Eingabefelder auf Dialogen) zuständig sind. Diese Properties erkennt man daran, dass mindestens ein Buchstabe kleingeschrieben ist. Diese Properties können von außen (über die Kommandozeile des Windows Installers) nicht gesetzt werden.

Öffentliche Properties können hingegen über die Kommandozeile gesetzt werden:

```
msiexec.exe /i „MySetup.msi“ ALLUSERS=1
```

Im oben dargestellten Beispiel wird das öffentliche Property ALLUSERS über die Kommandozeile gesetzt und bewirkt damit, dass das Setup für alle Benutzer installiert wird. Öffentliche Properties erkennt man daran, dass alle Buchstaben großgeschrieben sind.

Ein Property (hier ALLUSERS) kann bei WiX über folgende Zeile definiert werden:

```
<Product ...=" ">
...
<Property Id="ALLUSERS" Value="1"/>
...
</Product>
```

Tables	Property	Value
AdminExecuteSequence	Manufacturer	MyFirstSetup
AdminUISequence	ProductCode	{6EDD1F03-7981-4071-9098-7B9191829881}
AdvtExecuteSequence	ProductLanguage	1033
Directory	ProductName	MyFirstSetup
Feature	ProductVersion	1.0.0.0
File	UpgradeCode	{99822D01-10BE-49C1-88FC-8ACA3441AABA}
InstallExecuteSequence	ALLUSERS	1
InstallUISequence		
Media		
Property		
_Validation		

Öffnet man das kompilierte MSI mit Orca, kann man das hier definierte Property ALLUSERS sehen. Die anderen Zeilen in der Property-Tabelle wurden über das Product-Element hinzugefügt.

Soll das Property über eine Eingabe im Userinterface gesetzt und anschließend in die Registry o. Ä. geschrieben werden, so sollte bei der Definition des Properties das Attribut *Secure* auf den Wert *yes* gesetzt werden:

```
<Property Id="SERIALNUMBER" Secure="yes"/>
```

Secure="yes" bewirkt, dass das Property vom Userinterface in den Ausführen-Teil der Installation übernommen wird. Näheres finden Sie weiter hinten bei Kapitel **Sequenzen**.

Properties werden normalerweise am Ende einer Installation in das Logfile geschrieben. Dies kann u. U. problematisch sein, wenn das Property z. B. Passwörter zu einem Benutzerkonto enthält. In diesem Fall möchte man verständlicherweise nicht, dass das Property im Logfile ausgegeben wird. Über das Attribut *Hidden* kann genau dieses verhindert werden:

```
<Property Id="PASSWORD" Secure="yes" Hidden="yes"/>
```

Das setzen des Attributes *Hidden* bewirkt, dass das Property in das Windows Installer-Property **MsiHiddenProperties** geschrieben wird.

5.2 Installationsbedingungen

Bevor ein Setup auf dem Zielrechner ausgeführt wird, müssen wir zunächst prüfen, ob es überhaupt sinnvoll ist, das Programm auf diesem Rechner zu installieren. Es könnte ja sein, dass das Programm nur auf bestimmten Betriebssystemen lauffähig ist oder noch andere Bedingungen abgefragt werden müssen.

Im folgenden Beispiel prüfen einige globale Properties z. B., ob der Benutzer Administratorrechte besitzt, und beenden die Installation mit einer aussagekräftigen Fehlermeldung, falls unsere Bedingungen nicht erfüllt sind. Hierzu fügen wir über das Element **Condition** eine Bedingung hinzu:

```
<Condition Message="You need to be an administrator to install this product.">
  Privileged
</Condition>
```

► **Hinweis:** Beachten Sie, dass die Meldung nur dann erscheint, falls die Bedingung zwischen dem Starting- und Closing-Element FALSE ist. Anders gesagt: Sie sollten nicht die Fehlereigenschaft definieren, sondern den Sachverhalt beschreiben, der für die Installation benötigt wird.

Soll die ausgegebene Nachricht einen Wagenrücklauf (**CRLF**) enthalten, so können die Steuerzeichen **&xA;** **&xD;** verwendet werden.

Im Folgenden sind einige Properties aufgeführt, die der Windows Installer setzt und für die Definition von Bedingungen verfügbar sind:

Betriebssystem-Properties

Property	Funktion
VersionNT	Gibt die Versionsnummer des Betriebssystems zurück. Windows 7 hat die Versionsnummer 601, Windows 8 ist 602 und Windows 8.1 bzw. Windows 10 hat die Versionsnummer 603
ServicePackLevel	Enthält die Anzahl der installierten Service-Packs
WindowsBuild	Gibt die Build-Nummer des Betriebssystems zurück
SystemLanguageID	SystemLanguageID enthält die Standardsprache des Betriebssystems. Diese wird über einen Aufruf der API-Funktion GetSystemDefaultLangID ermittelt

Anwender-Properties

Property	Funktion
AdminUser	Der Windows Installer setzt dieses Property, wenn der Benutzer administrative Rechte besitzt. Als Bedingung für Adminrechte sollte jedoch das Property Privileged verwendet werden
UserLanguageID	UserLanguageID enthält die Standardsprache des angemeldeten Benutzers. Diese wird über einen Aufruf der API-Funktion GetUserDefaultLangID ermittelt
LogonUser	Gibt den Namen des angemeldeten Benutzers zurück
UserSID	Dieses Property enthält den User Security Identifier (SID) des angemeldeten Benutzers. Diese ist vor allem beim Setzen von Zugriffs-Berechtigungen interessant

Hardware- / System-Properties

Property	Funktion
PhysicalMemory	Gibt die Größe des physikalischen Speichers in Megabytes zurück
ScreenX, ScreenY	Gibt die aktuelle Auflösung in X- bzw. Y-Richtung wieder
ComputerName	Enthält den Namen des Computers

Windows Installer Properties

Property	Funktion
Privileged	Der Windows Installer setzt dieses Property, wenn der Benutzer entweder administrative Rechte besitzt oder das Setup mit erhöhten Rechten ausgeführt wird
Installed	Über dieses Property kann ermittelt werden, ob das Setup bereits installiert ist oder nicht. Ist das Property Installed nicht definiert, dann ist das Setup im Erstinstallations-Modus
Preselected	Dieses Property ist immer dann definiert, wenn ein oder mehrere Features über die Kommandozeile mittels der Properties ADDLOCAL, REMOVE, REINSTALL (und weitere) an- bzw. abgewählt wurden

Soll z.B. angeben werden, dass das Setup nicht unter Windows 95, 98 und ME (oder kurz Windows 9x) installiert werden kann, so definiert man dieses mit folgender Bedingung:

```
<Condition Message='This application does not run on Windows 95/98/ME.'>
    NOT Version9X
</Condition>
```

Eigentlich ist das Properties Version9X (wie auch VersionNT) keine boolesche Variable, sondern vom Typ Integer. Die oben beschriebene Bedingung ist immer dann wahr, wenn das Property nicht definiert ist – also keinen Wert enthält.

Es ist aber auch möglich, nur bestimmte Werte abzufragen:

```
<Condition Message='This application does not run on Windows NT 4.'>
    VersionNT>400
</Condition>
```

Es steht noch eine Vielzahl vergleichbarer Properties zur Auswahl. Zum Beispiel versetzt uns das Property `MsiINTProductType` in die Lage, zwischen Workstation, Domain-Controller und Server zu differenzieren.

► **Hinweis:** Ein Property mit dem Wert 0 ist nicht das gleiche wie ein undefiniertes Property. Die Bedingung `<PropertyName>` (die ohne weitere Operatoren) ist auch dann wahr, wenn das Property den Wert 0 besitzt.

Natürlich besteht auch die Möglichkeit, mehrere Bedingungen logisch miteinander zu verknüpfen.

Hierzu sind folgende Operatoren zuständig:

Operator	Funktion
NOT	Unäre Negation; Invertiert den Status des folgenden Terms
AND	Wahr, wenn beide Vergleichsterme den Wert „wahr“ haben
OR	Wahr, wenn einer der beiden Vergleichsterme den Wert „wahr“ hat
XOR	Wahr, wenn immer nur einer der beiden Vergleichsterme den Wert „wahr“ hat
EQV	Wahr, wenn entweder beide Vergleichsterme den Wert „wahr“ haben oder keiner
IMP	Wahr, wenn entweder der linke Vergleichsterm den Wert „falsch“ hat oder der rechte Vergleichsterm den Wert „wahr“ hat

Ein Ausdruck besteht hierbei immer aus Properties bzw. Konstanten, die mit folgenden Operatoren verglichen werden können:

Operator	Funktion
=	Überprüfung auf Gleichheit
<>	Überprüfung auf Ungleichheit
>	Linker Wert muss größer als der rechte Wert sein
>=	Linker Wert muss größer als der rechte Wert oder gleich dem rechten Wert sein
<	Linker Wert muss kleiner als der rechte Wert sein
<=	Linker Wert muss kleiner oder gleich dem rechten Wert sein
><	Bei Strings muss der linker String den rechten enthalten, bei numerischen Werten werden die beiden Operanden Bit für Bit mit UND verknüpft
<<	Linker String muss mit rechtem String beginnen
>>	Linker String muss mit rechtem String enden
~	Kann mit allen String-Operatoren kombiniert werden und besagt, dass der String-Vergleich nicht case sensitive durchgeführt wird.

5.3 System-Search

Benötigt das Programm ein anderes Programm (wie z. B. einen Acrobat Reader), so muss diese Überprüfung in zwei Schritten aufgeteilt werden. Im ersten Schritt ermitteln wir, ob ein Acrobat Reader installiert ist und schreiben das Ergebnis in ein Property. Im zweiten Schritt erstellen wir dann die Installationsbedingung und fragen dort das Property wieder ab.

5.3.1 Werte aus der Registry lesen

Viele Programme schreiben Werte wie z. B. die Versionsnummer oder das Installationsverzeichnis in die Registry. Möchten wir ermitteln, ob dieses Programm installiert ist, brauchen wir nur den entsprechenden Registry-Wert auslesen. Das wird in WiX über das **RegistrySearch**-Element gemacht:

```
<Property Id="ACROBAT_READER_PATH" Secure="yes">
  <RegistrySearch Id="AcrobatReaderSearch" Type="directory" Root="HKLM"
    Key="SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\AcroRd32.exe"
    Name="Path"/>
</Property>
```

Im gezeigten Beispiel lesen wir den Pfad zum Acrobat Reader aus der Registry. Benötigt unser Setup einen installierten Acrobat Reader, dann können wir nun eine entsprechende Installationsbedingung definieren:

```
<Condition Message='This application needs Acrobat Reader to run.'>
  ACROBAT_READER_PATH
</Condition>
```

Wir werden später noch sehen, dass die Reihenfolge der beiden Schritte im Skript nicht relevant ist. Es ist also gleich, ob wir zuerst das Condition-Element und danach das Property-Element im Skript definieren oder umgekehrt. Die Reihenfolge wird in der Sequenz definiert. Aber wie gesagt – dazu hören wir später noch mehr.

Wenn wir uns das RegistrySearch-Element noch einmal genauer ansehen, fällt auf, dass es ein Attribut namens *Type* umfasst. Mit diesem Attribut definieren wir, ob wir einen Pfad (directory), eine Datei (file) oder einen beliebigen Wert (raw) suchen. Setzen wir *Type* auf *directory*, bekommen wir nur dann einen Wert im Property zurück, wenn der ausgelesene Registry-Wert auch tatsächlich einen vorhandenen Pfad beschreibt. Wenn es uns nur um den Wert in der Registry ankommt und es nicht relevant ist, ob der Pfad existiert oder nicht, dann sollte *Type* auf *raw* gesetzt werden.

5.3.2 Werte aus einer INI-Datei lesen

Über das **IniFileSearch**-Element können INI-Dateien ausgelesen werden. Die INI-Dateien müssen allerdings im Windows-Verzeichnis liegen. Stellen wir uns vor, wir hätten eine Datei MySample.ini im Windows-Verzeichnis, die folgenden Aufbau hat:

```
[Sample]
InstallDir=C:\InstallHere
```

Über folgende Zeile können wir den Key *InstallDir* auslesen:

```
<Property Id='INI_ENTRY' Secure="yes">
  <IniFileSearch Id='MyIniSearch' Type='directory' Name='MySample.ini' Section='Sample'
    Key='InstallDir' />
</Property>
```

5.3.3 Dateien suchen

Über die Elemente **DirectorySearch** und **FileSearch** können Dateien gesucht werden. Suchten wir z. B. nach einem Acrobat Reader, der mindestens Versionsnummer 9.0 hat, dann könnte das etwa so aussehen:

```
<Property Id="ACRORD32" Secure="yes">
  <DirectorySearch Id="CheckFileDir" Path="[ProgramFilesFolder]" Depth="5">
    <FileSearch Id="AcroRd32File" Name="AcroRd32.exe" MinVersion="9.0"/>
  </DirectorySearch>
</Property>
```

Mit dem Attribut *Depth* kann man die Suchtiefe angeben, also die Anzahl der Unterverzeichnisse, in denen nach der Datei gesucht wird. Zero bzw. das Fehlen des *Depth*-Attributs bedeutet, dass nur in den angegebenen Ordner und nicht in den Unterverzeichnissen gesucht wird. Im Attribut *Path* geben wir den Startpfad der Suche an. Dort können wir über die eckigen Klammern auch Properties referenzieren. Geben wir keinen Pfad an, dann beginnt die Suche im Root-Verzeichnis aller fest eingebauten Laufwerke.

Als letztes Beispiel sei noch gezeigt, wie man den Pfad zu einer Datei zuerst aus der Registry liest und dann nach dieser Datei auf dem Filesystem sucht. Wenn die Datei gefunden wurde, soll in das Property der vollständige Pfad samt Dateinamen der gesuchten Datei stehen.

Als Beispiel wollen wir die Datei AcroRd32.exe suchen:

```
<Property Id="ACROBAT_READER_DC_FOUND" Secure="yes">
  <RegistrySearch Id="AcrobatReaderDc" Root="HKLM"
    Key="SOFTWARE\Adobe\Acrobat Reader\DC\InstallPath" Type="directory">
    <FileSearch Name="AcroRd32.exe"/>
  </RegistrySearch>
</Property>
```

5.4 Bedingung bei Feature

Wir haben bereits über Installationsbedingungen gesprochen. Wenn die angegebene Bedingung falsch ist, wird die gesamte Installation abgebrochen. Manchmal möchte man jedoch nur bestimmte Programmteile (in unserem folgenden Beispiel bestimmte Feature) vom Installationsprozess ausschließen.

Da unser bisheriges Setup aus nur einem Feature besteht, werden wir nun hier ein neues Feature samt der zu installierenden Komponente erstellen:

```
<Feature Id='AcrobatReaderPlugIn' Title='Acrobat Reader Plug In' Level='1'>
  <Condition Level='0'>NOT ACROBAT_READER_PATH</Condition>
  <Component Id='AcrobatPlugIn' Guid='YOURGUID' Directory='AcrobatPlugin'>
    <File Id='hdl.api' Source='.\SourceDir\AcrobatReaderPlugIn\hdl.api' KeyPath='yes'/>
  </Component>
</Feature>
```

Die Datei hdl.api wird in das Verzeichnis AcrobatPlugin installiert. Dieses haben wir weiter oben bereits als Unterverzeichnis von ACROBAT_READER_PATH definiert. Neu ist hier das Element **Condition**. Über dieses Element können wir eine Bedingung für das Feature definieren und den **Level** neu definieren.

Doch wie funktioniert das mit dem Level genau? Um das zu verstehen, muss man wissen, dass es ein Property namens **INSTALLLEVEL** gibt. Da dieses in unserem XML-Skript derzeit nicht definiert ist, wird dieses vom Windows Installer auf den Wert 1 gesetzt. Es gelten folgende Grundregeln:

Bedingung	Bedeutung
Level <= [INSTALLLEVEL]	Das Feature ist sichtbar und angewählt
Level > [INSTALLLEVEL]	Das Feature ist sichtbar und abgewählt, kann aber über das User-Interface wieder angewählt werden
Level = 0	Das Feature ist unsichtbar und abgewählt, kann somit über das User-Interface auch nicht wieder angewählt werden

5.5 Bedingung bei Komponenten

Auch Komponenten können Bedingungen haben. Das ist sinnvoll, wenn zum Beispiel je nach Setup-Sprache eine deutsche oder englische Hilfe installiert werden soll. Auch das Zielbetriebssystem kann Grund dafür sein, dass bei einer Komponente eine Bedingung definiert wird:

```
<Component Id='Help' Guid='YOURGUID' Directory='INSTALLDIR'>
  <Condition>ProductLanguage = 1031</Condition>
</Component>
```

In Bezug auf die Bedingung der Komponente ist das Component-Attribut *Transitive* ganz interessant. Wird das Attribut Transitive auf den Wert yes gesetzt, dann wird die Bedingung auch bei einer Reparatur des Setups neu ausgewertet. War die Bedingung bei einer früheren Installation nicht erfüllt und ist nun im Zuge der Reparatur erfüllt, dann wird die Komponente neu installiert. War die Bedingung bei einer früheren Installation erfüllt und ist nun erfüllt, dann wird die Komponente entfernt.

► **Hinweis:** Die Bedingungen bei den Komponenten werden wie auch bei den Features mit der Standard-Aktion **CostFinalize** ausgewertet.

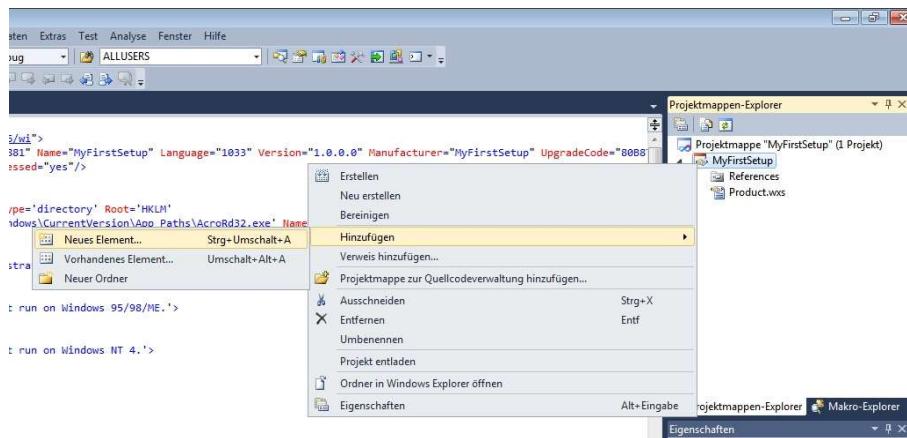
6 Modularität durch Fragments

Wie bereits erwähnt, ist es erst dann richtig sinnvoll, mit dem WiX-Toolset zu arbeiten, wenn das Setup bereits parallel zur Entwicklung der Software gepflegt wird. Das ist nicht zuletzt eine Grundforderung von agilen Softwareentwicklungsmethoden wie Scrum. Wenn mehrere Entwickler an einem Projekt arbeiten, ist es hilfreich, das Setup in einzelne Module aufzuteilen. Und genau das kann man mit **Fragments** machen. Selbstverständlich wird diese Methode auch angewandt, um den Code übersichtlicher zu gestalten und nach dem Top-down-Prinzip in kleine überschaubare „Häppchen“ zu gliedern.

Ein Fragment ist eine WXS-Datei, die mittels candle.exe kompiliert wird. Alle daraus resultierenden WIXOBJ-Dateien werden danach mit light.exe zu einem Setup zusammengebunden.

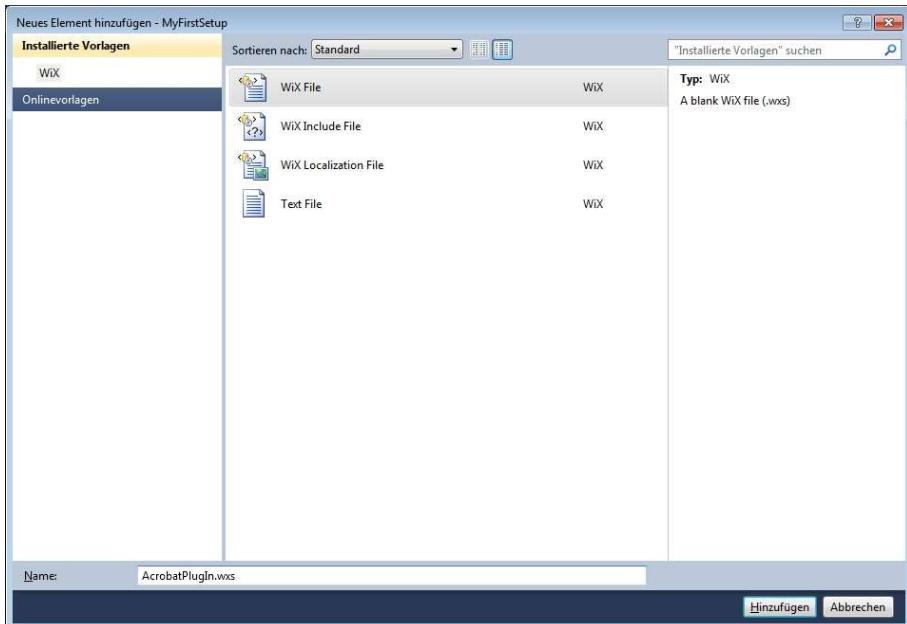
6.1 Fragment erstellen

Haben wir z. B. ein Plug-in für den Acrobat Reader, dann würden wir dieses wie folgt in eine eigene WXS-Datei auslagern.



Dazu gehen wir im Visual Studio zum Projektmappen-Explorer (wenn er nicht sichtbar ist, kann man ihn über den Menüpunkt **Ansicht▶Projektmappen-Explorer** sichtbar machen) und erstellen dort ein neues Element:

Im folgenden Fenster wählen wir dann eine neue WiX-Datei aus und nennen diese AcrobatPlugIn.wxs:



Die neue WXS-Datei bekommt schon einen entsprechenden Header und einen Fragment-Element. Diesem Element geben wir nun eine neue ID „Acrobat PlugIn“:

```
<?xml version="1.0" encoding="UTF-8"?>
<WiX xmlns="http://schemas.microsoft.com/WiX/2006/wi">
  <Fragment Id="Acrobat PlugIn">
    <!-- TODO: Put your code here. -->
  </Fragment>
</WiX>
```

Nun kopieren wir folgende Teile aus dem Haupt-Projekt in das Fragment:

```
<?xml version="1.0" encoding="UTF-8"?>
<WiX xmlns="http://schemas.microsoft.com/WiX/2006/wi">
  <Fragment Id="AcrobatPlugIn">
    <Property Id="ACROBAT_READER_PATH">
      <RegistrySearch Id="AcrobatReaderSearch" Type="directory" Root="HKLM"
        Key="SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\AcroRd32.exe"
        Name="Path"/>
    </Property>

    <DirectoryRef Id="TARGETDIR">
      <Directory Id="ACROBAT_READER_PATH" FileSource="AcrobatReader">
        <Directory Id="AcrobatPlugin" Name="plug_ins"/>
      </Directory>
    </DirectoryRef>

    <Component Id="AcrobatPlugIn" Guid="YOURGUID" Directory="AcrobatPlugin">
      <File Id="hdl.api" Source=".\\SourceDir\AcrobatReaderPlugIn\hdl.api"
        KeyPath="yes"/>
    </Component>
  </Fragment>
</WiX>
```

Da die Directory-Variable ACROBAT_READER_PATH ein Child von TARGETDIR ist und TARGETDIR bereits im Hauptsetup definiert ist, dürfen wir diese Variable im Fragment nicht nochmals definieren. Deshalb verweisen wir auf diese Variable mit dem Element **DirectoryRef**.

6.2 Fragment referenzieren

Im Hauptsetup bleibt nur noch ein kleiner Teil des Plug-ins übrig, nämlich das Feature und eine Referenz auf die Komponente:

```
<Feature Id="AcrobatReaderPlugIn" Title="Acrobat Reader Plug In" Level="1">
  <Condition Level="0">NOT ACROBAT_READER_PATH</Condition>
  <ComponentRef Id="AcrobatPlugIn" />
</Feature>
```

Beim Erstellen des Setups wird Visual Studio selbständig beide Teile mittels candle.exe und light.exe zusammenführen. Im Visual Studio werden die Fragmente wie folgt kompiliert:

```
candle.exe Product.wxs AcrobatPlugIn.wxs
light.exe -out MyFirstSetup.msi Product.wixobj AcrobatPlugIn.wixobj
```

Fragmente werden nur dann in das MSI-Setup übernommen, wenn im Hauptsetup ein Verweis (Referenz) auf ein WiX-Element enthalten ist. Das WiX-Element kann eine Komponente, ein Feature oder nur ein Property sein. Sobald ein Element aus einem Fragment referenziert ist, wird das gesamte Fragment in das Setup mit übernommen.

Wenn im Fragment kein Element vorhanden ist, das als Referenz eingebunden werden kann, gibt es noch den Ausweg, als Workaround dort einfach ein Dummy-Property zu definieren und im Hauptsetup zu referenzieren – das ist zwar nicht schön, aber pragmatisch!

6.3 Fragment als Funktionsbibliothek

Fragmente können beliebig komplex sein und können auch in mehreren Setup-Projekten verwendet werden. Wird z. B. ein Gerätetreiber in ein Fragment ausgelagert und in mehreren Setup-Projekten eingebunden, so sorgt der Windows Installer selbstständig dafür, dass der Gerätetreiber erst dann vom PC deinstalliert wird, wenn das letzte Setup deinstalliert wird, das den Gerätetreiber per Fragment eingebunden hatte.

Manchmal werden in einem Fragment viele Komponenten definiert. Um diese dann einem Feature zuzuweisen, müsste man für jede Komponente eine Referenz per ComponentRef angeben. Deshalb ist es manchmal sinnvoll, Komponenten zu Gruppen zusammenzufassen. Das macht man mit dem Element **ComponentGroup**:

```
<ComponentGroup Id="AcrobatPlugIn">
  <Component Id="AcrobatPlugIn" Guid='YOURGUID' Directory="AcrobatPlugin">
    <File Id="hdl.api" Source=".\\SourceDir\\AcrobatReaderPlugIn\\hdl.api" KeyPath="yes"/>
  </Component>
</ComponentGroup>
```

Dem Feature werden diese Komponentengruppen über das Element **ComponentGroupRef** wie folgt zugewiesen:

```
<Feature Id="AcrobatReaderPlugIn" Title="Acrobat Reader Plug In" Level="1">
  <Condition Level="0">NOT ACROBAT_READER_PATH</Condition>
  <ComponentGroupRef Id="AcrobatPlugIn"/>
</Feature>
```

7 WiX-Toolset-Variablen

Im WiX-Toolset gibt es mehrere Varianten von Variablen. Es gibt WiX-Variablen, Projekt-Referenz-Variablen und Präprozessorvariablen. Die Präprozessorvariablen werden mit Candle.exe aufgelöst, die WiX-Variablen erst mit light.exe.

7.1 WiX-Variablen

Anders als bei den Präprozessorvariablen werden WiX-Variablen erst mit light.exe aufgelöst. WiX-Variablen werden entweder über das Element **WixVariable**:

```
<WixVariable Id="MyProductName" Value="MyFirstSetup"/>
```

oder beim Aufruf von light.exe über den Parameter -d definiert:

```
light.exe -ext WiXUIExtension MySample.wixobj -d MyProductName=MyFirstSetup
```

Um sicherzugehen, dass die WiX-Variable einen initialen Wert hat, kann sie im Skript mit dem Attribut **Overridable** definiert werden. Eine so definierte Variable kann später noch einmal an einer anderen Stelle überschrieben werden. Das Attribut Overridable wird bei der Überladung nicht mehr angegeben:

```
<WixVariable Id="MyProductName" Value="MyFirstSetup" Overridable="yes"/>
```

Verwendet wird eine WiX-Variable durch folgenden Ausdruck:

```
<Product ... Name="!(wix.MyProductName)" ...>
```

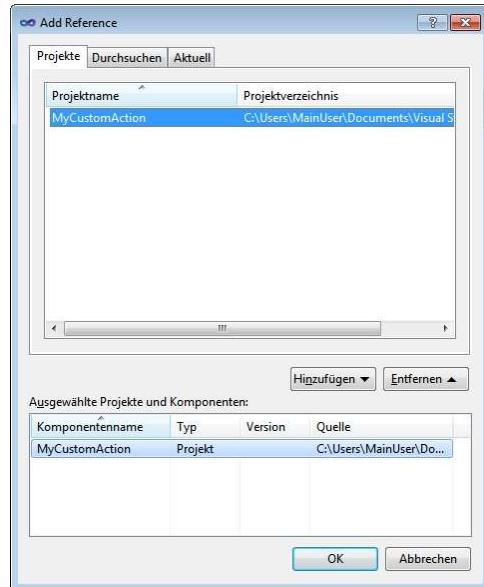
Es gibt auch noch eine Kurzform, über die man eine WiX-Variable zur gleichen Zeit definieren und den Default-Wert festlegen kann:

```
<Product ... Version="!(wix.Version=1.0.0)" ...>
```

Diese Zeile erstellt die WiX-Variable Version und initialisiert den Default-Wert mit der Versionsnummer 1.0.0. Somit kann die Versionsnummer jederzeit über die Kommandozeile von Candle überschrieben werden.

7.2 Projektreferenzvariablen

Neben den WiX-Variablen definiert uns das Visual Studio auch **Projektreferenzvariablen**, die auf andere Projekte (wie z. B. C# DLLs, die als Custom-Actions eingebunden werden) in derselben Solution zeigen.



Um die Projektreferenzvariablen benutzen zu können, muss man zuerst eine Referenz zum betreffenden Projekt hinzufügen.

Nachdem wir eine Referenz zum entsprechenden Projekt hinzugefügt haben, könnten wir das hier dargestellte Projekt unter dem Projektnamen „MyCustomAction“ referenzieren.

Möchte man z. B. das Kompilierungsergebnis von MyCustomAction referenzieren, dann sieht das folgendermaßen aus:

```
<Binary Id="MyCustomAction" SourceFile="$(var.MyCustomAction.TargetPath)" />
```

Im Anhang finden wir eine umfassende Auflistung der Projektreferenzvariablen.

► **Hinweis:** Lässt man den Projektnamen weg (wie zum Beispiel `[$var.Configuration]`), so bezieht sich die Variable auf das aktuelle WiX-Projekt.

7.3 Präprozessorvariablen

Die **Präprozessorvariablen** in WiX werden über das Element **define**: definiert:

```
<?define ProductName="MyProduct"?>
```

Im WiX-Skript kann dann diese Variable wie folgt eingefügt werden:

```
<Product Id="YOURGUID" Name="$(var.ProductName)" ...>
```

Die Variablen müssen aber nicht unbedingt im WiX-Skript vorbelegt werden. Sie können auch bei candle.exe per Kommandozeile (Parameter -d) angegeben werden:

```
candle.exe Product.wxs -dLanguage=1031
```

Um sicherzugehen, dass die Variable auf jeden Fall einen Wert hat, kann man auch mit den Elementen **ifndef** arbeiten:

```
<?ifndef ProductName ?>
  <?define ProductName="MyDefaultName"?>
<?endif?>
```

Selbstverständlich gibt es auch das Element **ifdef**. Alternativ zu Präprozessorvariablen kann man auch WiX-Variablen verwenden.

Über ein **error**-Element kann der Compiler-Lauf mit einer Fehlermeldung abgebrochen werden. Dieses Element wird z. B. dazu verwendet, anzuzeigen, dass eine bestimmte Variable gesetzt werden muss:

```
<?ifndef PropertyRegKey?>
  <?error Variable PropertyRegKey is required but not set.?>
<?endif?>
```

Einen Anwendungsfall sehen wir im Kapitel der Include-Dateien.

7.4 Umgebungsvariablen

Im WiX-Toolset können auch Umgebungsvariablen verwendet werden. Umgebungsvariablen sind ein einfaches und effizientes Mittel, wenn der Build-Prozess z. B. über eine Batch-Datei aufgerufen wird. Auf Umgebungsvariablen verweist man in WiX-Skript wie folgt:

```
<Product Id="*" Version="$(env.Productversion)" ...>
```

Möchte man sicher gehen, dass ein sinnvoller Wert verwendet wird, obwohl die Umgebungsvariable nicht definiert ist, kann man folgende Zeilen verwenden:

```
<?ifdef env.Version?>
  <?define Version = "$(env.Version)"?>
<?else?>
  <?define Version = "1.0.0"?>
<?endif?>
<Product Id="*" Name="Test" Version="$(var.Version)" ...>
```

7.5 Automatisch generierte Versionsnummer

Das WiX-Toolset unterstützt die Funktion **AutoVersion**, mit der man Versionnummern automatisch generieren lassen kann. Diese Funktion gibt man wie folgt an:

```
<Product Id="*" Version="$(fun.AutoVersion(2.0))" ...>
```

Die AutoVersion hat dasselbe Schema wie das AssemblyVersion-Attribut in .Net wobei die angegebenen Versionsnummern die Major- und Minor-Versionsnummern sind. Die nächste Stelle (Build-Nummer) wird automatisch generiert und erhält die Anzahl der Tage, die seit dem 01.01.2000 vergangen sind. Die letzte Stelle (Revisionsnummer) enthält die Anzahl der vergangenen Sekunden seit Mitternacht. Das oben angegebene Beispiel erzeugt somit Versionsnummern, die etwa so aussehen: 2.0.5592.10301.

► **Hinweis:** Die ermittelten Werte bei AutoVersion werden mithilfe der UTC (koordinierte Weltzeit) ermittelt.

7.6 Bindervariablen

Über die sogenannten **Bindervariablen** ist es u. A. möglich, die Versionsnummer des Setups an der Versionsnummer einer bestimmten Datei festzubinden. Somit muss nur die Versionsnummer der eingebetteten Ressource (EXE oder DLL) gepflegt werden und das Setup bekommt dann automatisch dieselbe Versionsnummer zugeteilt:

```
<Product Id="*" Name="BinderDemo" Version="!(bindFileVersion.Sample.exe)" ...>
  ...
  <Component Id="IpwiSample.exe" Guid="YOURGUID" Directory="INSTALLFOLDER">
    <File Id="IpwiSample.exe" Source=".\\Sample.exe"/>
  </Component>
</Product>
```

Wie wir sehen, sprechen wir Bindervariablen mit den Ausrufenzeichen, gefolgt vom Schlüsselwort bind, an. Das WiX-Toolset hat aber noch eine ganze Reihe anderer Bindevariablen, die wir hier nicht verschweigen wollen:

Variablenname	Beispiel
bind.fileLanguage.FileID	1033
bind.fileVersion.FileID	1.2.0.0
bind.assemblyCulture.FileID	Neutral
bind.assemblyFileVersion.FileID	1.2.0.0
bind.assemblyFullName.FileID	MyAssembly, version=1.0.0.0, culture=neutral, publicKeyToken=0123456789ABCDEF, processorArchitecture=MSIL Der publicKeyToken wird in Großbuchstaben umgewandelt und ausgegeben
bind.assemblyFullNamePreservedCase.FileID	MyAssembly, version=1.0.0.0, culture=neutral, publicKeyToken=0123456789abcdef, processorArchitecture=MSIL Die Schreibweise von publicKeyToken bleibt erhalten
bind.assemblyName.FileID	MyAssembly

Variablenname	Beispiel
bind.assemblyPublicKeyToken.FileID	publicKeyToken=0123456789ABCDEF Der publicKeyToken wird in Großbuchstaben umgewandelt und ausgegeben
bind.assemblyPublicKeyTokenPreservedCase.FileID	publicKeyToken=0123456789abcdef Die Schreibweise von publicKeyToken bleibt erhalten
bind.assemblyProcessorArchitecture.FileID	MSIL
bind.assemblyType.FileID	Win32
bind.assemblyVersion.FileID	1.0.0.0

► **Hinweis:** Bindervariablen werden erst beim Linken (Light.exe) aufgelöst. So können sie auch problemlos in WiX-Libraries verwendet werden.

Zusätzlich zu den Bindervariablen, die Informationen aus einer Datei referenzieren, gibt es auch noch die Property-Bindervariablen. Property-Bindervariablen geben den Wert des angegebenen Properties zurück. Möchte man z. B. ein Feature erstellen, das immer genau so heißt, wie das Property *ProductName*, dann sieht das wie folgt aus:

```
<Product Id="*" Name="MyBinderSample" ...>
  ...
  <Feature Id="MainFeature" Display="=(bind.property.ProductName)">
  ...
</Feature>
</Product>
```

Enthält das Property eine Versionsnummer, dann kann über Major, Minor, Build und Revision die erste, zweite, dritte und vierte Stelle der Versionsnummer verwendet werden. Ist die Versionsnummer des Setups z. B. 2.5.6.1 und man gibt !(bind.property.ProductVersion.Major) an, dann wird der Wert 2 an entsprechender Stelle vom Linker eingesetzt.

► **Hinweis:** Bei der Definition von Properties können keine Property-Bindevariablen verwendet werden.

7.7 Iterationen

Im WiX-Toolset gibt es eine Präprozessordirektive namens **foreach**, über die man über eine Stringliste iterieren kann. Möchte man z. B. für jede Sprache, die eine Anwendung unterstützt, ein eigenes Verzeichnis definieren, dann kann man das mit folgenden Zeilen machen:

```
<!-- Define language list -->
<?define Languages=en;de;fr;it?>

<!-- Create director variables for each language -->
<?foreach Language in $(var.Languages)?>
  <DirectoryRef Id="INSTALLFOLDER">
    <Directory Id="$(var.Language)" Name="$(var.Language)"/>
  </DirectoryRef>
<?endforeach?>
```

Man sieht: Wir erstellen uns zunächst eine durch Semikolon getrennte Liste mit allen Sprachen und iterieren dann mit der Foreach-Anweisung durch die Liste und erstellen uns Directory-Variablen mit den entsprechenden Namen. Nun erstellen wir noch eine Komponente, die die Verzeichnisse anlegt:

```
<Component Id="CreateLanguageFolder" Guid="YOURGUID" Directory="INSTALLFOLDER" >
    <?foreach Language in $(var.Languages)?>
        <CreateFolder Directory="$(var.Language)"/>
    </foreach?>
</Component>
```

Die Foreach-Anweisung muss innerhalb der Komponente eingetragen werden, da der Präprozessor die Zeilen innerhalb der Foreach-Anweisung mehrmals einträgt. Wenn die Foreach-Anweisung außerhalb der Komponente stehen würde, hätten wir Komponenten mit identischen IDs und identischen GUIDs. Der Compiler würde das so nicht akzeptieren.

7.8 Escape-Sequenz

Wie wir in den vorhergehenden Kapiteln gesehen haben, markieren das Dollar- und Ausrufezeichen Präprozessor bzw. WiX-Variablen. Doch was, wenn wir tatsächlich ein Dollar bzw. Ausrufezeichen im Text haben wollen? Wie in anderen Programmiersprachen auch, gibt es hierfür **Escape-Sequenzen** – wir verdoppeln das Zeichen einfach. Der Ausdruck \$\$Test ergibt dann den Text \$(Test), !!Test ergibt !(Test).

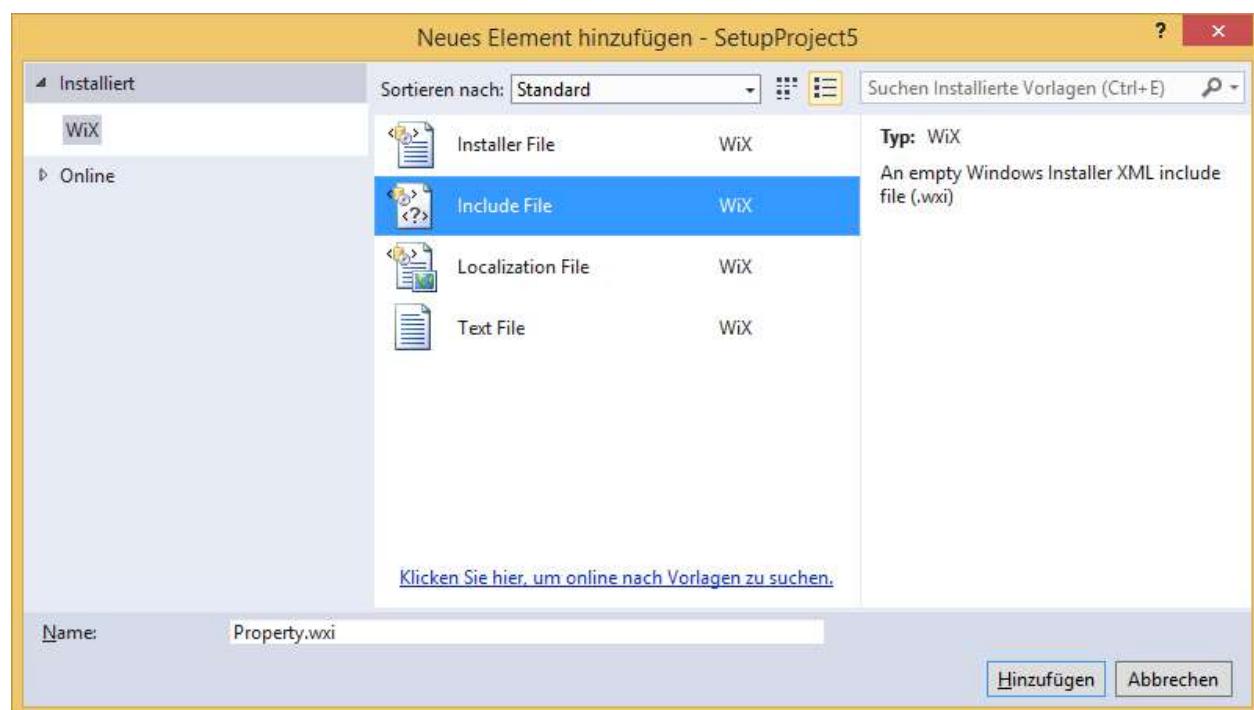
8 Include-Dateien

Dadurch dass man Include-Dateien einsetzt, kann man Programmteile wie eine Art **Makro** verwenden. An der Stelle, an der eine **Include**-Anweisung steht, wird die angegebene Datei durch den Präprozessor in das WXS-Skript geladen und anschließend kompiliert. Die Parameter für das Makro werden in der Regel über Präprozessordirektiven angegeben.

Gehen wir einmal davon aus, dass wir mehrere Properties mit RegSearch-Anteil erstellen müssten. Ein Property würde dann in etwa so aussehen:

```
<Property Id="MYPROPERTY" Secure="yes">
  <RegistrySearch Id="MyProperty_Search" Root="HKLM" Key="Software\[ProductName]"
    Name="MYPROPERTY" Type="raw"/>
</Property>
```

Genau diesen Programmteil wollen wir nun als Makro definieren. Dazu erstellen wir uns eine neue Include-Datei, indem wir ein neues Element auf dem Projekt hinzufügen:



In die neu erstellte Include-Datei tragen wir nun das Makro wie folgt ein:

```
<Include>
  <?ifndef PropertyName?>
    <?error Variable $var.PropertyName is required but not set.?>
  <?endif?>

  <Property Id="$(var.PropertyName)" Secure="yes">
    <RegistrySearch Id="$(var.PropertyName)_Search" Type="raw" Root="HKLM"
      Key="Software\[ProductName]" Name="$(var.PropertyName)" />
  </Property>
</Include>
```

Das Error-Element stellt sicher, dass alle Parameter für das Makro vorher angegeben wurden. Nun kann die Include-Datei in das Projekt mit folgenden Zeilen aufgerufen werden:

```
<?define PropertyName="PROPERTY1"?>
<?include Property.wxi ?>
```

Der Clou dabei ist, dass wir die Include-Dateien nun mehrmals hintereinander einbinden können:

```
<?define PropertyName="PROPERTY1"?>
<?include Property.wxi ?>

<?define PropertyName="PROPERTY2"?>
<?include Property.wxi ?>
```

Möchte man in Zukunft die Properties von einer anderen Stelle aus der Registry entnehmen, müssen wir die Änderung nur einmal in der Include-Datei vornehmen. Man kann sich sicher vorstellen, dass, wenn der Code in der Include-Dateien umfangreicher ist, wir einiges an Übersichtlichkeit gewinnen können.

9 Neben den Dateien

Im wirklichen Leben braucht es wahrscheinlich mehr, als Dateien zu kopieren. Oft müssen zu den Dateien Registry-Einträge, Dateiverknüpfungen und INI-Dateien beschrieben werden. Und genau darum werden wir uns nun kümmern.

9.1 Registry

Wie Dateien werden Registry-Einträge bestimmten Komponenten zugeordnet. Registry-Einträge werden über die Elemente **RegistryKey** und **RegistryValue** definiert. Für jedes Setup sollte sinnvollerweise zumindest der Installationsort eines Setups in der Registry abgelegt werden. So kann man bei einem Update sofort erkennen, wohin ältere Anwendungen installiert wurden, um dann gegebenenfalls vor der Aktualisierung noch benutzerspezifische Daten zu retten.

Die Registry-Keys werden wir hier in eine neue Komponente packen. Wenn diese Einträge zu einer Anwendung gehören, so kann oder muss man diese Einträge in dieselbe Komponente packen, in der die Anwendung sich befindet:

```
<Component Id='MyRegistry' Guid='YOURGUID' Directory='INSTALLDIR'>
  <RegistryKey Id='Hkml_MyFirstSetup' Root='HKLM' Key='Software\[ProductName]'
    ForceDeleteOnUninstall='yes'>
    <RegistryValue Type='string' Name='InstallDir' Value='[INSTALLDIR]' KeyPath='yes' />
    <RegistryValue Type='string' Name='Version' Value='[ProductVersion]' />
  </RegistryKey>
</Component>
```

Den Inhalt von Properties oder Directory-Variablen trägt man in die Registry ein, indem man eckige Klammern um die Variablen setzt. So wird also der Inhalt von ProductName, INSTALLDIR und ProductVersion in die Registry geschrieben. ForceDeleteOnUninstall='yes' bewirkt, dass bei der Deinstallation der Registry-Key samt aller Sub-Keys gelöscht wird.

Bei RegistryValue kann als Typ *string*, *integer*, *binary*, *expandable* und *multiString* angegeben werden. Bei *multiString* trägt man mehrere RegistryValue-Einträge mit demselben Namen ein und setzt das Attribut *Action* auf *append*:

```
<RegistryKey Root='HKLM' Key='Software\[ProductName]'>
  <RegistryValue Type='multiString' Name='InstallDir' Value='[INSTALLDIR]' />
  <RegistryValue Type='multiString' Name='InstallDir' Value='[AcrobatPlugIn]' Action='append' />
</RegistryKey>
```

► **Hinweis:** Wie wir in dem obigen Beispiel sehen, kann auch ein Registry-Eintrag der KeyPath der Komponente sein.

9.2 Dateiendungen registrieren

Wenn eine Anwendung Dateien mit einer bestimmten Dateiendung erstellt, ist es sicher wünschenswert, dass diese durch einen Doppelklick im Explorer automatisch geöffnet werden. Möchten wir z. B. die Dateiendung .ISD registrieren, dann müssen wir das über die Elemente **ProgId**, **Extension** und **Verb** machen:

```
<Component Id="MyExtension" Guid="YOURGUID" Directory="INSTALLDIR">
  <File Id="IpwiSample.exe" Source=".\\IpwiSample.exe" KeyPath="yes" />
  <ProgId Id="IsdFile" Description="WiX Sample File">
    <Extension Id="isd">
      <Verb Id="open" Command="Open" TargetFile="IpwiSample.exe" Argument='"%1"' />
    </Extension>
  </ProgId>
</Component>
```

► **Hinweis:** Das Attribut TargetFile im Element Verb verweist auf die ID des File-Elementes, nicht auf den Dateinamen selbst!

Die oben beschriebene Methode registriert die Dateiendung so, dass die auszuführende EXE-Datei direkt aufgerufen wird. Falls diese beim Doppelklick auf die Datendatei nicht vorhanden ist, wird Windows das mit einer Fehlermeldung quittieren. Besser wäre es, wenn wir die Anwendung vorher vom Windows Installer prüfen lassen, und zwar indem wir bei der ProgId das Attribut Advertise auf yes setzen:

```
<ProgId Id="IsdFile" Description="WiX Sample File" Advertise="yes">
  <Extension Id="isd">
    <Verb Id="open" Command="Open" Argument='"%1"' />
  </Extension>
</ProgId>
```

Voraussetzung dafür ist, dass sich die ProgId in derselben Komponente befindet wie die zu startende Anwendung. Falls für die Datei auch noch ein Icon definiert werden soll, dann kann das über das Attribut Icon und IconIndex bei der ProgId angegeben werden:

```
<ProgId Id="IsdFile" Advertise="yes" Icon="IpwiSample.exe" IconIndex="0">
```

9.3 INI-Dateien

Falls Sie in einer INI-Datei dynamische Parameter, wie z. B. das Installationsverzeichnis, beschreiben wollen, dann brauchen Sie folgenden Code in einer Komponente:

```
<Component Id="Sample.ini" Guid="YOURGUID" Directory="INSTALLDIR">
  <File Id="Sample.ini" Source=".\\SourceDir\\INI-File\\Sample.ini" KeyPath="yes"/>
  <IniFile Id="Sample.ini" Name="Sample.ini" Directory="INSTALLDIR"
    Section="Installation" Key="Installdir" Value="[INSTALLDIR]"
    Action="addLine" />
</Component>
```

Die oben dargestellten Zeilen installieren die Sample.ini nach INSTALLDIR und verändern diese über das **IniFile**-Element. Die INI-Datei und das IniFile-Element sollten immer in derselben Komponente sein, da dann die Anpassung immer dann durchgeführt wird, wenn man die Datei auch auf das Zielsystem installiert. Über die Reihenfolge im Skript müssen wir uns keine Gedanken machen, da zuerst die Datei installiert und danach die Anpassung erledigt wird.

Folgende Werte sind für das Attribut Action möglich:

Action	Funktion
addLine	Erstellt oder ändert einen INI-File-Eintrag
addTag	Erstellt einen neuen Eintrag, falls er noch nicht existiert, oder fügt den neuen Wert kommagetrennt an den bestehenden Wert an
createLine	Erstellt einen neuen Eintrag nur dann, wenn er noch nicht existiert
removeLine	Löscht einen Eintrag aus einer INI-Datei
removeTag	Löscht den angegebenen Wert aus dem Eintrag

9.4 HTML-Link erstellen

Möchte man auf dem Desktop eine Verknüpfung auf ein HTML-Dokument erstellen, dann kann man das mit einer URL-Datei machen. Die URL-Datei ist vom Aufbau her eine INI-Datei und wird direkt in das Verzeichnis DesktopFolder installiert. Eine URL-Datei hat folgenden Aufbau:

```
[InternetShortcut]
URL=C:\Program Files\MyCompany\MyFirstSetup\Tutorial\index.html
IconIndex=0
IconFile=c:\MyFolder\MyIcon.exe
```

Da wir in der INI-Datei die Pfade angeben, können wir die URL-Datei nicht statisch kopieren, sondern erstellen diese am einfachsten über das IniFile-Attribut:

```
<Component Id="Tutor.ico" Directory="INSTALLDIR" Guid="YOURGUID">
  <File Id="Tutor.ico" Source=".\\Tutor.ico" KeyPath="yes" />
</Component>

<Component Id="LaunchTutorial" Guid="YOURGUID" Directory="INSTALLDIR" KeyPath="yes">
  <IniFile Id="LaunchTutorial" Action="addLine" Directory="DesktopFolder"
    Name="Tutorial.url" Section="InternetShortcut"
    Key="URL" Value="[TUTORIAL]index.html" />
  <IniFile Id="IconIndex" Action="addLine" Directory="DesktopFolder"
    Name=" Tutorial.url" Section="InternetShortcut"
    Key="IconIndex" Value="0"/>
  <IniFile Id="IconFile" Action="addLine" Directory="DesktopFolder"
    Name=" Tutorial.url" Section="InternetShortcut"
    Key="IconFile" Value="[INSTALLDIR]Tutor.ico"/>
</Component>
```

► **Hinweis:** Wird IconFile und IconIndex weggelassen, wird der Shortcut mit dem Icon des Default-Browsers angezeigt.

9.5 ODBC-Eintrag erstellen

ODBC (Open Database Connectivity) ist eine standardisierte Datenbankschnittstelle, über die man auf Datenbanken mittels der SQL Syntax zugriffen kann. ODBC bietet eine Programmierschnittstelle (API), über die man unabhängig vom verwendeten Datenbanksystem auf Datenbanken zugreifen kann. Dafür werden ODBC-Treiber geliefert, über die eine Anwendung auf die Datenbank zugreift. Der Treiber ist einer Art Übersetzer, der zwischen ODBC (Anwendungsseite) und der eigentlichen Datenbanksprache (Datenbankseite) vermittelt.

Über ODBC kann man die Credentials der Datenbank, wie den Namen des Datenbanktreibers, den Namen des Datenbankservers, den Datenbanknamen, den Benutzernamen und das Passwort und weitere Details einrichten und diese über einen logischen Namen, den sogenannten **DSN** (Data Source Name) verwenden. Die Anwendung selbst greift über den logischen Namen auf die Datenbank zu, ohne selbst über genauere Informationen der Datenbank-Credentials zu verfügen.

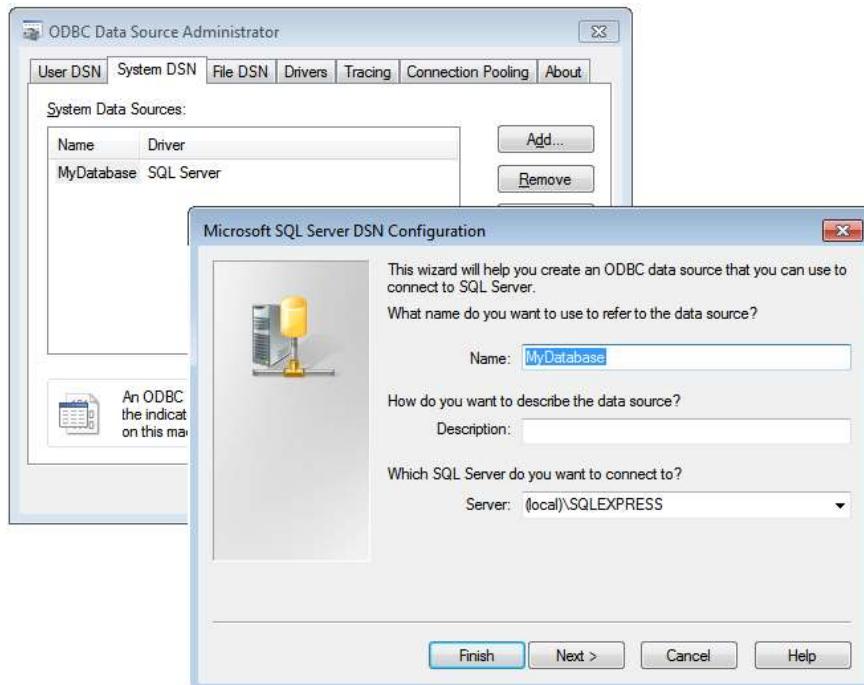
Welche Credentials für den Zugriff auf eine Datenbank benötigt werden, hängt sehr stark von der verwendeten Datenbanktechnologie ab. So braucht man für eine Access-Datenbank z. B. nur den Pfad zur MDB-Datei und ihren Dateinamen. Ein Microsoft-SQL-Server hingegen benötigt den Server- und Instanz-Namen, den Datenbanknamen, den Benutzernamen und das Passwort, um auf die Datenbank zuzugreifen. Da die Credentials von der Datenbanktechnologie abhängen, bringt jeder ODBC-Treiber ein Setup mit, über die die benötigten Credentials über einen Wizard erfragt und abgespeichert werden.

ODBC wird auf Windows-Systemen über den ODBC-Datenquellen-Administrator **odbcad32.exe** bzw. Systemsteuerung►Datenquellen (ODBC) eingerichtet.

► **Hinweis:** Achtung: 64-Bit-Betriebssystem! Ruft man auf einem 64-Bit-Windows den ODBC-Datenquellen-Administrator über die Systemsteuerung auf, so wird die 64-Bit-Variante gestartet. DSNs, die mit dieser Version eingerichtet werden, können von 32-Bit-Programmen nicht erkannt werden (sie werden in der 64-Bit-Registry abgelegt). Soll ein DSN für 32-Bit-Anwendungen eingerichtet werden, muss die odbcad32.exe aus dem Verzeichnis C:\Windows\SysWOW64 verwendet werden.

9.5.1 ODBC-DSN erstellen

Da die Credentials von der verwendeten Datenbanktechnologie abhängen, richten wir den DSN über den ODBC-Datenquellen-Administrator ein:

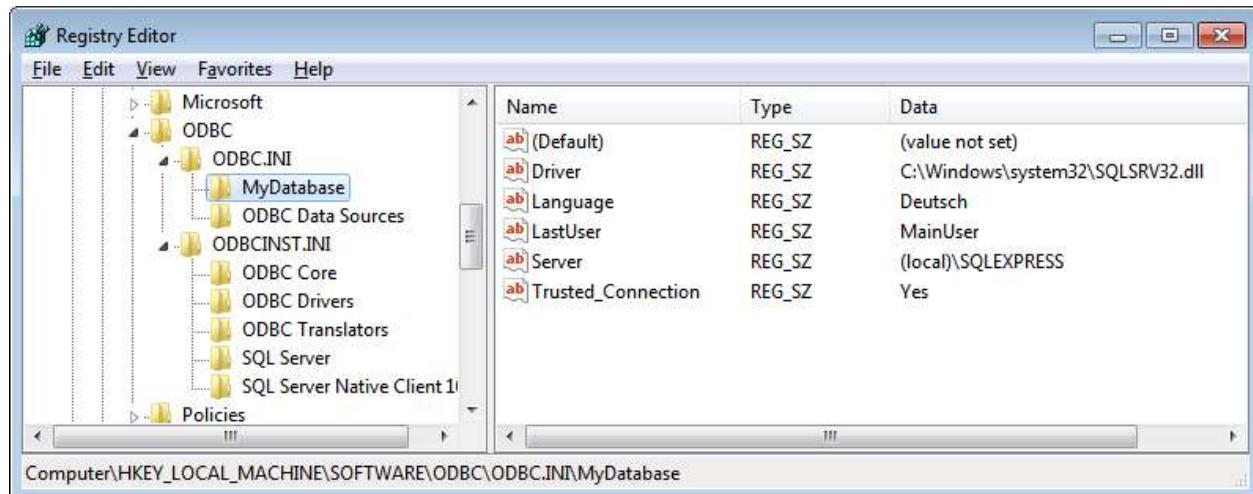


Die einzelnen DSN-Parameter können jetzt aus der Registry unter:

HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI

ermittelt werden.

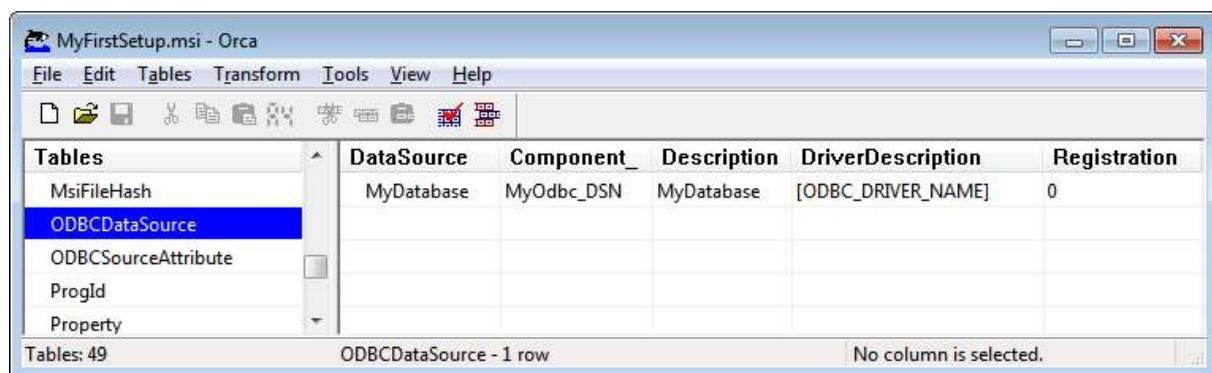
Hier sehen wir die Parameter einer SQL-Server-Verbindung:



In WiX legen wir den DSN über das Element **ODBCDataSource** an. Dort geben wir neben dem DSN-Namen den Treibernamen an. Anhand des Treibernamens weiß dann der Windows Installer, mit welcher Treiber-DLL der DSN verbunden werden soll. Alle anderen Parameter (wie *Language*, *Server* oder *Trusted Connection*) werden über das Element **Property** unterhalb von ODBCDataSource angegeben:

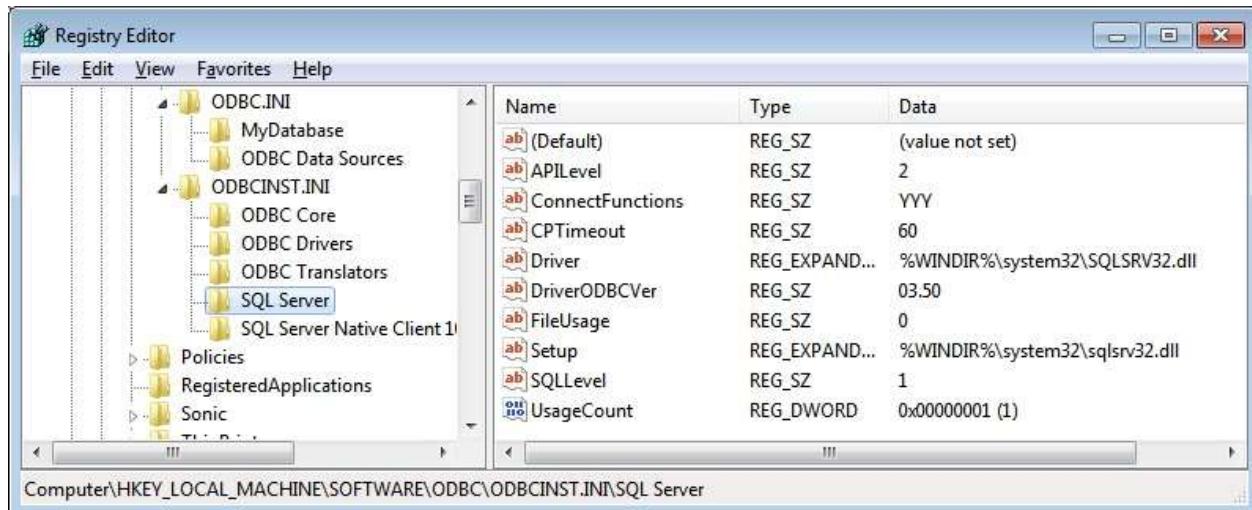
```
<Component Id="MyOdbc_DSN" Guid="YOURGUID" Directory="INSTALLDIR">
  <ODBCDataSource Id="MyDatabase" Name="MyDatabase" DriverName="SQL Server"
    Registration="machine" KeyPath="yes">
    <Property Id="Language" Value="Deutsch" />
    <Property Id="Server" Value="[SQL_SERVER_NAME]" />
    <Property Id="Trusted_Connection" Value="Yes" />
  </ODBCDataSource>
</Component>
```

Über das Attribut *Registration* legen wir fest, ob der DSN für den aktuellen Benutzer oder für alle Benutzer erstellt wird. Alle Werte, die über das ODBCDataSource-Element angegeben werden landen in der Windows Installer Tabelle **ODBCDataSource**, alle Werte der Property-Elemente in der **ODBCSourceAttribute** Tabelle:



9.5.2 ODBC-Treiber installieren

Wie die DSNs stehen auch die Treiber bei ODBC in der Registry:



ODBC-Treiber können wir ebenfalls über WiX installieren. Hierzu erstellt man zuerst eine Komponente, fügt dort die Treiber-DLL hinzu und richtet den Treiber über das WiX-Element **ODBCDriver** ein.

ODBCDriver erstellt hierbei die Registry-Einträge *Driver*, *Setup* und *UsageCount*. Alle weiteren Parameter müssen über das **Property**-Element unterhalb von **ODBCDriver** erstellt werden:

```
<Component Id="SqlDriver" Guid="YOURGUID" Directory="SystemFolder">
  <File Id="Mysqlsrv.dll" Source=".\\SourceDir\\OdbcDriver\\Mysqlsrv.dll"
    KeyPath="yes">
    <ODBCDriver Id="MySQLDriver" Name="My SQL Server">
      <Property Id="SQLLevel" Value="1" />
      <Property Id="FileUsage" Value="0" />
      <Property Id="DriverODBCVer" Value="03.50" />
      <Property Id="ConnectFunctions" Value="YYY" />
      <Property Id="APILevel" Value="2" />
      <Property Id="CPTimeout" Value="60" />
    </ODBCDriver>
  </File>
</Component>
```

Das **ODBCDriver**-Element füllt die Windows Installer Tabelle **ODBCDriver** und die **Property**-Elemente landen in der **ODBCAttribute** Tabelle:

Tables	Driver	Compon...	Description	File_	File_Setup
ODBCAttribute ODBCDataSource ODBCDriver ODBCSourceAttribute ProgId	MySQLDriver	SqlDriver	My SQL Server	Mysqlsrv.dll	Mysqlsrv.dll

Tables: 51 ODBCDriver - 1 row No column is selected.

9.6 Dateien löschen

Viele Anwendungen erstellen zur Laufzeit temporäre Dateien (z. B. Log-Dateien), die bei der Deinstallation der Anwendung auch mit entfernt werden sollen. Da der Windows Installer nur diejenigen Dateien löscht, die er auch installiert hat, müssen wir dem Windows Installer sagen, dass hier noch mehr getan werden muss. Und das macht man über das **RemoveFile**-Element:

```
<Component Id="RemoveFiles" Guid="YOURGUID" Directory="INSTALLDIR" KeyPath="yes">
    <RemoveFile Id="RemoveLogFiles" Name="*.log" Directory="INSTALLDIR" On="uninstall"/>
</Component>
```

Mit dem Attribut **On** können wir bestimmen, ob die Datei bei der Installation, Deinstallation oder immer gelöscht werden soll. Wie wir im obigen Beispiel sehen, können wir auch Wildcards (*,?) verwenden.

Man kann statt des Directoys auch ein Property angeben. Das Property kann zur Laufzeit z. B. per Custom-Action gesetzt werden. So kann man Dateien aus einem Ordner entfernen, den wir zum Zeitpunkt der Erstellung des Installationspaketes noch nicht kennen.

Über das Element RemoveFolder können auch Ordner gelöscht werden:

```
<Component>
    ...
    <RemoveFolder Id="LogFolder" Directory="LOGFOLDER" On="uninstall" />
</Component>
```

Allerdings muss der angegebenen Ordner leer sein – im Zweifelsfall müssen die Dateien und Ordner, die sich im angegebenen Ordner befinden, durch RemoveFile bzw. RemoveFolder-Elemente geleert werden.

10 Dateien in den Global Assembly Cache installieren

Das Installieren von .NET-Assemblies in den **Global Assembly Cache** (GAC) ist mit dem WiX-Toolset sehr einfach. Zu Beginn erstellt man das Zielverzeichnis über das Directory-Element:

```
<Directory Id="TARGETDIR">
  <Directory Id="GlobalAssemblyCache"/>
</Directory>
```

Der Windows Installer setzt das Property **GlobalAssemblyCache** auf den Global Assembly Cache des lokalen PCs (typischerweise ist das C:\Windows\Assembly) und überträgt dieses dann auf die gleichnamige Directory-Variable. Dieses Vorgehen kennen wir ja bereits von anderen Systemverzeichnissen.

Da im Global Assembly Cache mehrere Versionen desselben Assemblies abgelegt werden können, müssen die Assemblies einen sogenannten „strong name“ haben. Dieser setzt sich aus einem öffentlichen und einem privaten Key zusammen. Dieses Key-Paar kann man z. B. über das .NET Framework SDK Tool SN.exe erzeugen und auf das Assembly übertragen.

Um ein Key-Paar zu erstellen, ruft man SN.exe wie folgt auf:

```
SN -k MyKeyValuePair.snk
```

Die SN.exe erstellt die binäre Datei MyKeyValuePair.snk, in welcher der öffentliche und der private Key enthalten sind. Diese Datei gibt man in C# der Assembly-Datei über das Attribute AssemblyKeyFile an:

```
[assembly: AssemblyKeyFile("MyKeyValuePair.snk")]
```

In Visual Basic.NET sieht das dann in etwa so aus:

```
<Assembly: AssemblyKeyFile("MyKeyValuePair.snk")>
```

Nachdem das Assembly erstellt ist, kann dieses in den Global Assembly Cache installiert werden. Dazu erstellen wir eine Komponente, die nach GlobalAssemblyCache installiert wird, fügen das Assembly ein und setzen beim File-Element das Assembly-Attribut auf .net:

```
<Component Id="MyAssembly.dll" Directory="GlobalAssemblyCache" Guid="YOURGUID">
  <File Id=" MyAssembly.dll " KeyPath="yes" Source=".\\SourceDir\\MyAssembly.dll"
        Assembly=".net" />
</Component>
```

Durch das Setzen des Assembly-Attributs, werden zwei neue MSI-Tabellen, **MsiAssembly** and **MsiAssemblyName**, sowie die zugehörige Action **MsiPublishAssemblies** in das MSI eingebunden. Die Action MsiPublishAssemblies macht die ganze Arbeit für uns. Mehr ist nicht nötig.

11 Benutzerinterface

In einer Lektion zuvor haben wir gelernt, wie man Dateien in das Setup mit aufnimmt. Bisher lief alles ohne das sonst übliche Benutzerinterface, die es dem Anwender ermöglicht, das Installationsverzeichnis oder ähnliche Angaben vorzunehmen. Um dieses Thema wollen wir uns in diesem Abschnitt beschäftigen.

Der Windows Installer selbst hat keine eingebauten Dialoge – außer einer einfachen Fortschrittsanzeige, die wir bereits gesehen haben und ein paar Message-Boxen die auftauchen, um den Benutzer über verschiedene Fehler zu informieren. Es obliegt also alleine dem MSI-Setup, das Benutzerinterface zu definieren. Und das macht man über das **UI-Element**.

Wenn wir das gesamte Benutzerinterface selbst entwickeln müssten, hätten wir einiges an Arbeit vor uns. Glücklicherweise ist dies nicht notwendig, da das WiX Toolset mit einer Library von Standard-Dialogen ausgestattet ist, die in Form einer **WiX-Extension** vorliegt. Hier stellt sich sofort die Frage: Was ist eine WiX-Extension überhaupt?

11.1 WiX-Extension

Die einfachste Form einer WiX-Extension besteht aus einer WiX-Library (vorkompilierter WiX-Code) in Form von Fragmenten, die beim Linken des Setups mit in das MSI aufgenommen werden, falls diese referenziert wurden.

Zusätzlich kann eine WiX-Extension aber auch den Sprachumfang unseres WiX-Skripts durch weitere WiX-Elemente erweitern. Diese Extensions enthalten neben der WiX-Library auch noch eine Compiler-Erweiterung in Form von C# Klassen, die entweder von der Wix Basisklasse **PreprocessorExtension** oder **WixExtension** abgeleitet sind. In diesen WiX-Extensions befinden sich dann oft auch noch weitere Dateien wie die eine DLL mit Custom Actions, eine XSD-Datei für die Spracherweiterung und Tabellendefinitionen in Form von INI-Dateien.

Ein prominenter Vertreter dieser komplexeren WiX-Extensions ist z.B. die **WiXUtilExtension**, die weiter unten beschrieben wird.

11.2 Benutzerinterface einbinden

Wir werden unser letztes Beispiel um ein schönes Benutzerinterface erweitern. Das WiX Toolset enthält die **WixUIExtension**, welche eine WiX-Library mit mehreren Dialog-Reihen enthält. Die WixUIExtension hat fünf unterschiedliche „Geschmacksrichtungen“: **WixUI_Mondo**, **WixUI_FeatureTree**, **WixUI_InstallDir**, **WixUI_Minimal** und **WixUI_Advanced**. Welche Dialoge in den entsprechenden Geschmacksrichtungen enthalten sind, können wir im Anhang sehen.

Um ein Benutzerinterface zu bekommen, müssen wir eine Referenz zu das entsprechende UI-Element hinzufügen:

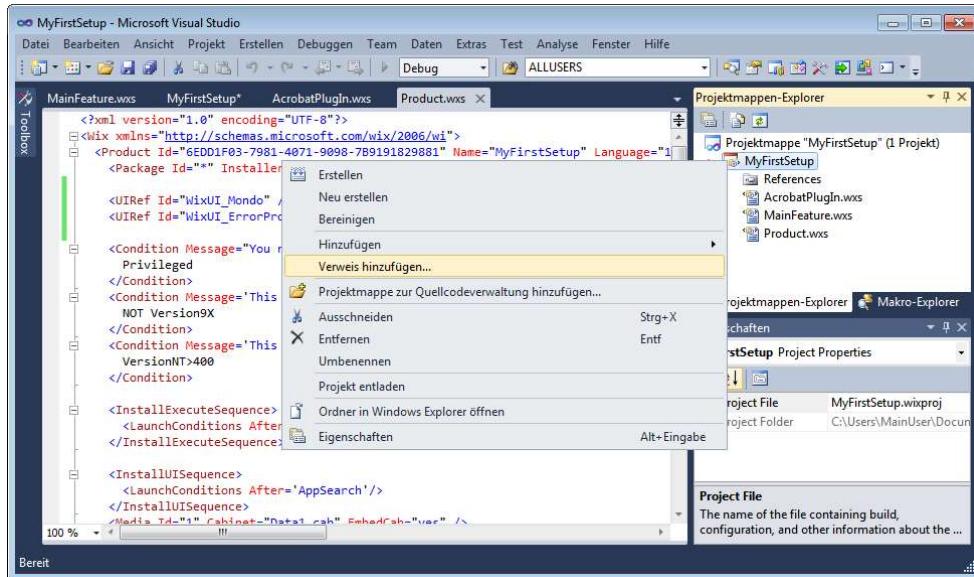
```
<UIRef Id="WixUI_Mondo" />
```

Möchten wir auch noch Fehlermeldungen und aussagekräftige Texte auf dem Fortschrittsbalken-Dialog haben, dann Referenzieren wir auch noch das Benutzerinterface **WixUI_ErrorProgressText**.

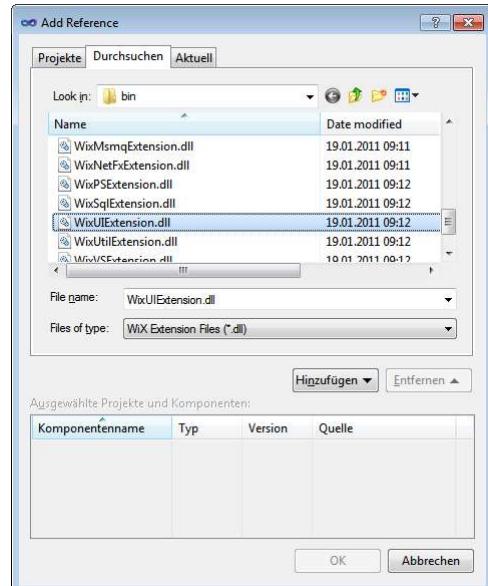
```
<UIRef Id="WixUI_ErrorProgressText" />
```

Das ist aber noch nicht alles. Wir müssen der light.exe auch noch mitteilen, in das beim Binden die WiX-Library aus der WixUIExtension berücksichtigt werden sollen. Dafür gehen wir in den Projektmappen-Explorer und klicken mit der rechten Maustaste auf References.

Im daraufhin erscheinenden Menü wählen wir den Menüpunkt „Verweis hinzufügen ...“ an:



Im sich nun öffnenden Fenster wählen wir die WixUIExtension.dll aus:



Wenn wir unser Setup jetzt neu erstellen, haben wir ein Setup mit einem schönen Benutzerinterface. Soll das Setup über die Kommandozeile erstellt werden, so müssen wir bei light.exe einen Verweis auf die WixUIExtension angeben:

```
candle.exe MySample.wxs
light.exe -ext WixUIExtension MySample.wixobj
```

► Hinweis: Im Anhang sind alle Dialogreihen der WixUIExtension dargestellt.

11.3 Jetzt wird's bunt

In dieser Lektion werden wir auf den Dialogen die Lizenzvereinbarung und ein paar Bitmaps anpassen.

11.3.1 Bitmaps austauschen

Das Benutzerinterface von WiX definiert alle auf den Dialogen dargestellten Bitmaps als WiX-Variablen (mit dem Attribut Overridable auf „yes“ gesetzt), sie können also in unserem WiX-Skript bzw. mit light.exe überschrieben werden.

Die Variable **WixUIBannerBmp** definiert das Banner-Bitmap (493 auf 58 Pixel) und die Variable **WixUIDialogBmp** (493 auf 312 Pixel) gibt ein Bitmap an, das sich über den gesamten Bereich des ersten und letzten Dialoges legt:

```
<!-- Bitmaps austauschen -->
<WixVariable Id="WixUIBannerBmp" Value=".\\SourceDir\\Dialog Bitmaps\\Banner.bmp" />
<WixVariable Id="WixUIDialogBmp" Value=".\\SourceDir\\Dialog Bitmaps\\Dialog.bmp" />
```

Es gibt noch weitere Variablen, über die Icons auf Dialogen geändert werden können:

- WixUIExclamationIco
- WixUIInfoIco
- WixUINewIco
- WixUIUpIco

Diese werden aber nur in seltenen Fällen ausgetauscht.

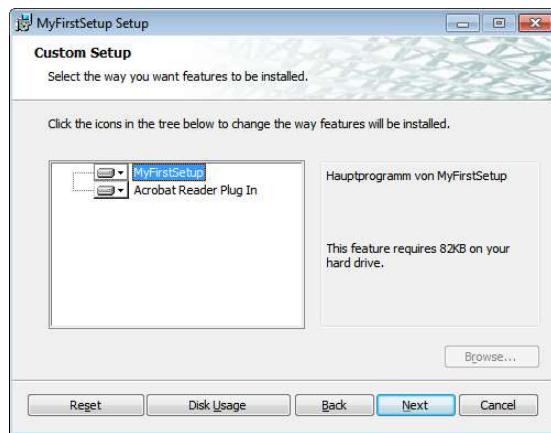
11.3.2 Lizenzvereinbarung

Auf dem zweiten Dialog wird die Lizenzvereinbarung ausgegeben. Diese wollen wir durch unsere eigene ersetzen. Natürlich ist hier auch eine WiX-Variable definiert. Diese Variable heißt **WixUILicenseRtf**. Die Lizenzvereinbarung ist eine RTF-Datei, die über WordPad o. Ä. formatiert und editiert werden kann:

```
<WixVariable Id="WixUILicenseRtf" Value=".\\SourceDir\\Licence\\Licence.rtf" />
```

11.4 Einstellungen beim Feature

Über unser Benutzerinterface können die Benutzer die zur Verfügung stehenden Feature an- und abwählen:



Das Feature-Attribut **Title** ist hier der Anzeigenname, das Attribut **Description** wird bei der Auswahl des entsprechenden Features auf der rechten Seite als Beschreibung des Features angezeigt.

Das **Display**-Attribut (mögliche Werte sind *collapse*, *expand* und *hidden*) bestimmt, ob das Feature ein- oder ausgeklappt ist - also, ob die Subfeatures angezeigt werden oder nicht - und ob das Feature überhaupt für den Endbenutzer sichtbar ist. Das Display-Attribut gibt aber auch die Reihenfolge vor, in der die Features dargestellt werden. Möchte man diese Reihenfolge ändern, so gibt man im Display-Attribut einfach einen Integer-Wert ein. Je höher der Wert ist, desto weiter unten steht das Feature.

Da die Einstellungen *collapse* und *expand* ebenfalls über Display festgelegt werden, hat Microsoft den Wert von Display folgendermaßen definiert:

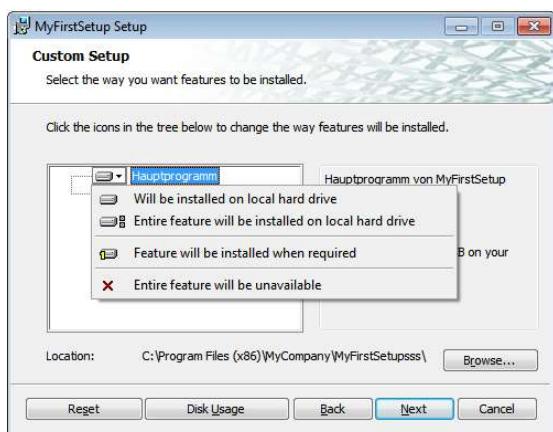
- Ist der Wert in Display gerade, dann ist das Feature eingeklappt
- Ist der Wert in Display ungerade, dann ist das Feature ausgeklappt
- Ist der Wert in Display = 0, dann ist das Feature unsichtbar

Da unser Feature *ProductFeature* ein Subfeature hat und wir keine Reihenfolge vorgeben wollen, setzen wir das Display-Attribut auf *expand*.

Im oben dargestellten Dialog sehen wir, dass der Browse-Button (rechts unten) deaktiviert ist. Das liegt daran, dass das Attribut **ConfigurableDirectory** des angewählten Features nicht gesetzt ist. Dieses Attribut ist ein Verweis auf ein Directory, welches über diesen Button verändert werden kann, wenn das Feature angewählt wird.

Wir ändern also unsere Feature wie folgt ab:

```
<Feature Id="ProductFeature" Title="Hauptprogramm" Display="expand" Absent="disallow"
    Level="1" ConfigurableDirectory="INSTALLDIR" >
    <ComponentGroupRef Id ="MainFeature" />
</Feature>
```



Klickt man mit der linken Maustaste auf ein Feature, so können mehrere Einstellungen am Feature vorgenommen werden:

Mit verschiedenen Attributen des **Features** kann das Aussehen des Menüs beeinflusst werden. In der folgenden Tabelle wird genauer beschrieben, welche Attribute auf welche Werte gesetzt werden:

Attribut und Wert	Beschreibung
AllowAdvertise='no'	Der Menüpunkt <i>Feature will be installed when required</i> wird entfernt. Dem Benutzer ist es nicht erlaubt, dieses Feature auf Anforderung zu installieren.
AllowAdvertise='yes'	<i>Feature will be installed when required</i> wird im Menü dargestellt. Dem Benutzer ist es erlaubt, dieses Feature auf Anforderung zu installieren.
AllowAdvertise='system'	<i>Feature will be installed when required</i> wird nur im Menü dargestellt, falls das Betriebssystem dies unterstützt (ab Windows 98 und installiertem Active Desktop).
InstallDefault='local'	<i>"Will be installed on local hard drive"</i> wird im Menü dargestellt.
InstallDefault='source'	<i>"Will be installed to run from CD"</i> wird im Menü dargestellt.
InstallDefault='followParent'	Der aktuelle Zustand (entweder <i>local</i> oder <i>source</i>) wird von der Einstellung des übergeordneten Features übernommen.
Absent='allow'	<i>Entire feature will be unavailable</i> wird im Menü dargestellt. Der Benutzer kann entscheiden, ob er dieses Feature installieren möchte oder nicht.
Absent='disallow'	Das Feature kann vom Benutzer nicht abgewählt werden.

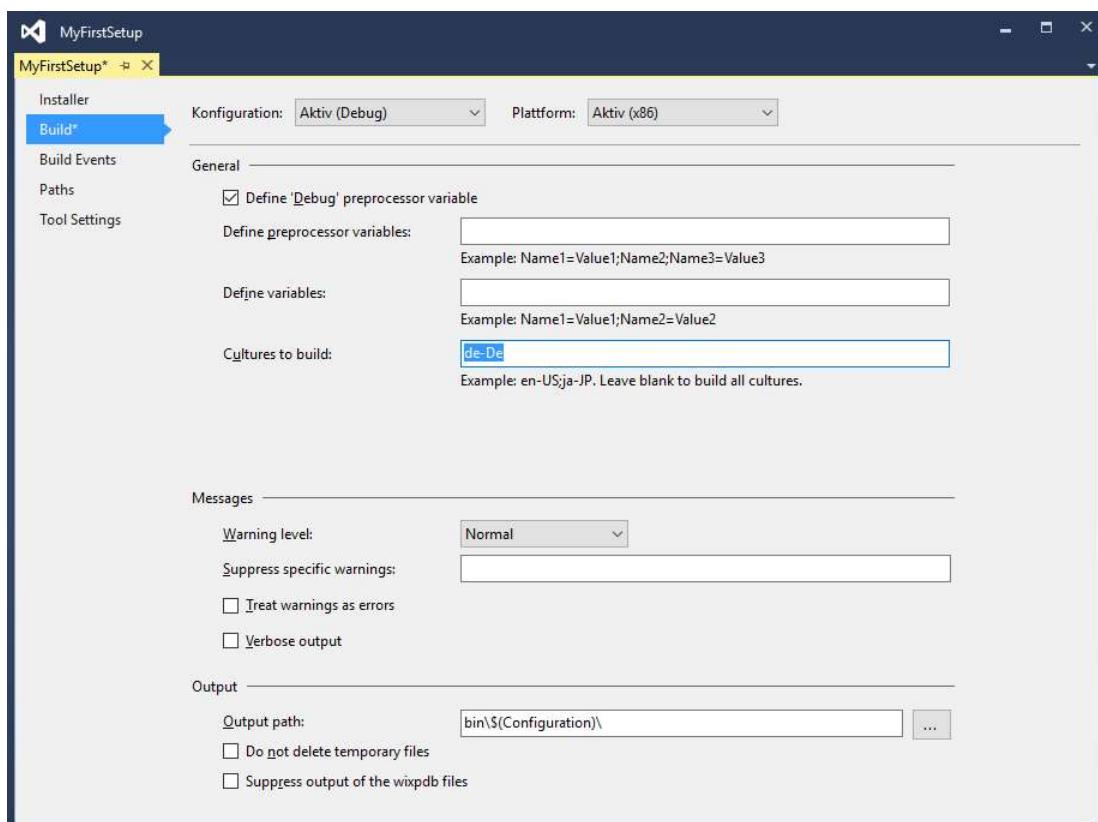
12 Lokalisierung

12.1 WixUI-Extension-Library

Das Standardsetup von WiX zeigt die Dialoge in englischer Sprache an. Das kann man ändern, WiX unterstützt derzeit eine ganz beachtliche Anzahl unterschiedlicher Sprachen. Die zu erstellenden Sprachen wählt man entweder, indem man dem Setup ein WiX-Localization-File in der entsprechenden Sprache hinzufügt (hierzu später noch mehr) oder über den Cultures-Schalter in der Kommandozeile der Light.exe:

Light.exe MySetup.wixobj -cultures:de-de -ext WixUIExtension.dll

In der gezeigten Kommandozeile wird das Setup in deutscher Sprache erstellt. Aus dem Visual Studio heraus setzen wir die zu erstellenden Sprachen über die Einstellungen des Setup-Projektes unter **Build▶Cultures to build**:



Wenn wir eine andere Sprache hinzufügen, sollten wir die Language-ID und die Codepage im **Product**- und **Package**-Element angeben. Ohne die richtigen Spracheneinstellungen kann es vorkommen, dass unerwartet Fehlermeldungen ausgegeben werden oder nichtlateinische Buchstaben gar nicht angezeigt werden. Für unser deutsches Setup geben wir den Language-Code 1252 an:

```
<Product ...="" Language="1031" ...="">
<Package ...="" Languages="1031" SummaryCodepage="1252" ...="" />
```

Die unten angegebenen Sprachen werden zum Zeitpunkt der Fertigstellung dieses Dokumentes unterstützt (um den Status anderer Sprachen zu prüfen, gehen Sie bitte auf folgende Internetseite: <http://wix.tramontana.co.hu/tutorial/localization>):

Sprache	Culture	Lokalisierungsdatei	ID	Codepage	
	Saudi Arabia	ar-SA	WixUI_ar-SA.wxl	1025	1256
	Bulgarian	bg-BG	WixUI_bg-BG.wxl	1026	1251
	Catalan	ca-ES	WixUI_ca-ES.wxl	1027	1252
	Croatian	hr-HR	WixUI_hr-HR.wxl	1050	1250
	Czech	cs-CZ	WixUI_cs-cz.wxl	1029	1250
	Danish	da-DK	WixUI_da-dk.wxl	1030	1252
	Dutch	nl-NL	WixUI_nl-nl.wxl	1043	1252
	English	en-US	WixUI_en-us.wxl	1033	1252
	Estonian	et-EE	WixUI_et-EE.wxl	1061	1257
	Finnish	fi-FI	WixUI_fi-FI.wxl	1035	1252
	French	fr-FR	WixUI_fr-fr.wxl	1036	1252
	German	de-DE	WixExt_de-de.wxl	1031	1252
	Greek	el-GR	WixUI_el-GR.wxl	1032	1253
	Hebrew	he-IL	WixUI_he-IL.wxl	1037	1255
	Hindi	hi-IN	WixUI_hi-IN.wxl	1081	65001
	Hungarian	hu-HU	WixUI_hu-hu.wxl	1038	1250
	Italian	it-IT	WixUI_it-it.wxl	1040	1252
	Japanese	ja-JP	WixUI_ja-jp.wxl	1041	932
	Kazakh	kk-KZ	WixUI_kk-KZ.wxl	1087	1251
	Korean	ko-KR	WixUI_ko-KR.wxl	1042	949
	Latvian	lv-LV	WixUI_lv-LV.wxl	1062	1257
	Lithuanian	lt-LT	WixUI_lv-LV.wxl	1063	1257
	Norwegian (Bokmål)	nb-NO	WixUI_nb-NO.wxl	1044	1252
	Polish	pl-PL	WixUI_pl-pl.wxl	1045	1250
	Portuguese Brazil	pt-BR	WixUI_pt-BR.wxl	1046	1252
	Portuguese Portugal	pt-PT	WixUI_pt-PT.wxl	2070	1252
	Romanian	ro-RO	WixUI_ro-ro.wxl	1048	1250
	Russian	ru-RU	WixUI_ru-ru.wxl	1049	1251
	Serbian (Latin)	sr-Latn-CS	WixUI_sr-Latn-CS.wxl	2074	1250
	Simplified Chinese	zh-CN	WixUI_zh-CN.wxl	2052	936
	Slovak	sk-SK	WixUI_sk-sk.wxl	1051	1250
	Slovenian	sl-SI	WixUI_sl-SI.wxl	1060	1250

Sprache	Culture	Lokalisierungsdatei	ID	Codepage
Spanish	es-ES	WixExt_es-es.wxl	3082	1252
Swedish	sv-SE	WixUI_sv-SE.wxl	1053	1252
Thai	th-TH	WixUI_th-TH.wxl	1054	874
Trad. Chin. Hong Kong	zh-HK	WixUI_zh-HK.wxl	3076	950
Traditional Chinese Taiwan	zh-TW	WixUI_zh-tw.wxl	1028	950
Turkish	tr-TR	WixUI_tr-TR.wxl	1055	1254
Ukrainian	uk-UK	WixExt_uk-ua.wxl	1058	1251

Viele Sprachen sind in der WixUI-Extension-Library kompiliert. Ist das nicht der Fall, dann entstehen beim Kompilieren mehrere Fehler. In diesem Fall muss die oben genannte Lokalisierungsdatei mit ins Setup aufgenommen werden. Die Lokalisierungsdatei findet man im Source-Code von WiX im Verzeichnis: „\src\ext\UIExtension\Wixlib“.

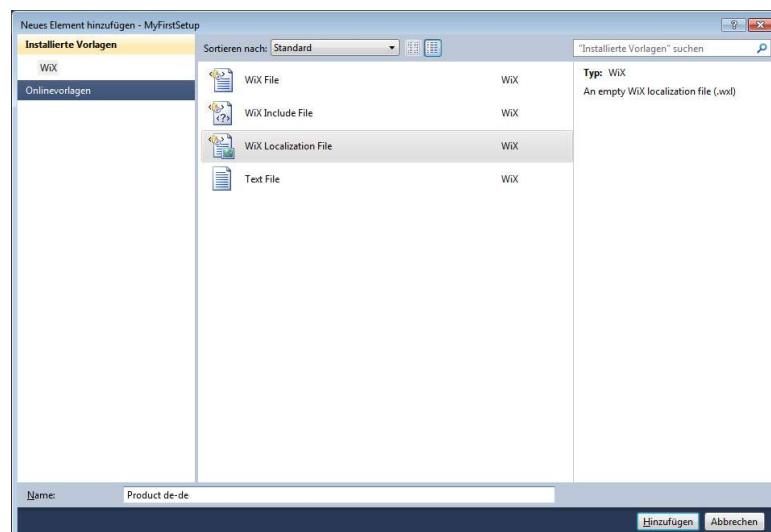
Möchte man die in der WixUI-Extension-Library eingebundenen Texte ändern, so ist das auch kein Problem. Die Strings in der WixUI-Extension-Library sind so definiert, dass sie überschrieben werden können. Wie man das macht bzw. wie Localisation-Files aufgebaut sind, sehen wir im nächsten Abschnitt.

12.2 String-Verweise und WiX-Localisation-Files

In einem internationalen Entwicklungsprojekt möchte man das Setup nicht nur in einer, sondern in mehreren Sprachen ausgeben. Da es derzeit schon einige Texte (z. B. Title und Description beim Feature) in unserem Skript gibt, brauchen wir eine Möglichkeit, WiX zu sagen, wie der Text in der jeweiligen Setupsprache auszusehen hat.

Hierzu gibt es **WiX-Localisation-Files** mit der Dateiendung WXL. In den WiX-Localisation-Files kann man für jede beliebige Setupsprache über das Element **WixLocalization** einen eigenen Bereich erstellen. Jeder String bekommt hierbei eine eindeutige ID, die wir dann in unserem WiX-Skript referenzieren. In den meisten Fällen ist es sinnvoll, dass wir für jede Sprache eine eigene Localisation-Datei erstellen – das muss nicht sein, ist aber für eine spätere Übersetzung durchaus hilfreich.

Zum Beispiel erstellen wir für unser Setup zwei Localisation-Files, „Product de-de.wxl“ für Deutsch und „Product en-us.wxl“ für Englisch. Hierzu gehen wir in unser Projekt und fügen eine neue Datei vom Typ „WiX Localisation File“ hinzu:



Im Localisation-File gibt man im WixLocalization-Element über das Culture-Attribut die Sprache der Stringtabelle an und kann dann über das Child-Element **String** die Texte angeben:

```
<WixLocalization Culture="de-de"
    xmlns="http://schemas.microsoft.com/wix/2006/localization">
    <String Id="LangID">1031</String>
    <String Id="FeatureTitleProductFeature">Hauptprogramm</String>
    <String Id="FeatureDescriptionProductFeature">Hauptprogramm von MyFirstSetup</String>
</WixLocalization>
```

Die Datei „Product en-us.wxl“ sieht dann so aus:

```
<WixLocalization Culture="en-us"
    xmlns="http://schemas.microsoft.com/wix/2006/localization">
    <String Id="LangID">1033</String>
    <String Id="FeatureTitleProductFeature">Mainprogram</String>
    <String Id="FeatureDescriptionProductFeature">Mainprogram from MyFirstSetup</String>
</WixLocalization>
```

Strings aus einer Lokalisation referenziert man im WiX-Code durch: **!(loc.StringID)**. Damit sieht das Features wie folgt aus:

```
<Feature Id="ProductFeature" Title="!(loc.FeatureTitleProductFeature)" Display="expand"
    Absent="disallow" Description="!(loc.FeatureDescriptionProductFeature)"
    ConfigurableDirectory="INSTALLDIR" Level="1">
    <ComponentGroupRef Id ="MainFeature" />
</Feature>
```

In den Elementen Product und Package muss man die Sprache beim Language-Attribut als Language-ID angeben. Das macht man über eine Variable im Localisation-File:

```
<Product ... Language="!(loc.LangID)" ... >
<Package ... Languages="!(loc.LangID)" ... />
```

Solange die String-IDs eindeutig sind, darf ein Setup-Projekt beliebig viele Localisation-Files derselben Sprache besitzen.

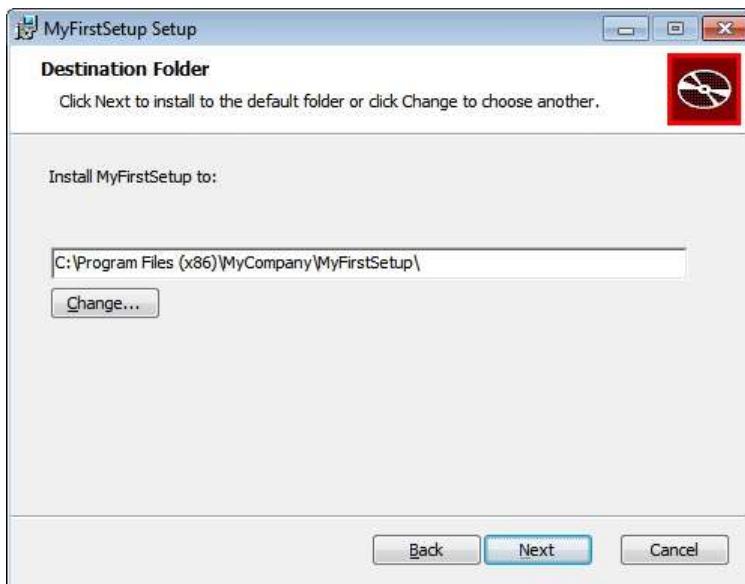
13 Ein neuer Dialog entsteht

Obwohl die WiXUI-Interface-Libraries mit den meisten üblichen Setupszenarien umgehen können, sind Modifizierungen oder Ergänzungen manchmal erforderlich. Um diese Fälle abwickeln zu können, benötigen wir den WiX-Source-Code (den Sie auf unserer Seminar-CD oder im Internet unter <http://wix.codeplex.com/SourceControl/list/changesets> finden).

In unserem Beispiel werden wir die WixUI_Mondo-Library um einen neuen Dialog für die Seriennummerringabe erweitern. Dieser Dialog soll nach der Lizenzvereinbarung und vor dem Setup-Typ-Dialog erscheinen.

Dafür erstellen wir ein neues Fragment namens SerialNumberDlg.wxs und die zugehörigen Localisation-Dateien SerialNumberDlg de-de.wxl und SerialNumberDlg en-us.wxl. Um einen neuen Dialog zu erstellen, haben wir zwei Möglichkeiten: Entweder wir erstellen den Dialog ganz neu oder wir kopieren einfach einen existierenden Dialog und ändern ihn ab. Zweiteres ist wohl fast immer die einfachere Variante.

Wir suchen uns also den Dialog, der für unsere Zwecke am passendsten ist. Der DestinationFolder-Dialog hat eigentlich schon alles, was wir brauchen:



Der Dialog hat die Kopfzeile mit den Bitmaps, einen Back-, Next- und Cancel-Button und ein Eingabefeld.

Dieser Dialog ist in der Library **WixUI_Advanced** unter dem Namen **InstallDirDlg** zu finden und ist in der WiX-Quelldatei InstallDirDlg.wxs im Verzeichnis `src\ext\UI\Extension\WiXlib` definiert.

Wenn der Quellcode geöffnet wird, sehen wir, dass ein Dialog immer aus dem Element **Dialog** und den Elementen **Control** besteht. Das Element **Dialog** definiert den Namen, die Größe und die Ausrichtung des Dialogs. Die **Control**-Elemente definieren, welche Elemente an welcher Position auf dem Dialog zu finden sind.

Folgende Controls können auf einem Dialog vorhanden sein: Billboard, Bitmap, CheckBox, ComboBox, DirectoryCombo, DirectoryList, Edit, GroupBox, Hyperlink, Icon, Line, ListBox, ListView, MaskedEdit, PathEdit, ProgressBar, PushButton, RadioButtonGroup, ScrollableText, SelectionTree, Text, VolumeCostList und VolumeSelectCombo.

Den Inhalt der WiX-Quelldatei *InstallDirDlg.wxs* kopieren wir in unsere Datei *SerialNumberDlg.wxs* und ändern diese wie folgt ab:

```
<?xml version="1.0" encoding="UTF-8"?>
<WiX xmlns="http://schemas.microsoft.com/WiX/2006/wi">
  <Fragment>
    <UI>
      <Dialog Id="SerialNumberDlg" Width="370" Height="270"
        Title="!(loc.SerialNumberDlg)">
        <Control Id="Next" Type="PushButton" X="236" Y="243" Width="56" Height="17"
          Default="yes" Text="!(loc.WiXUINext)" />
        <Control Id="Back" Type="PushButton" X="180" Y="243" Width="56" Height="17"
          Text="!(loc.WiXUIBack)" />
        <Control Id="Cancel" Type="PushButton" X="304" Y="243" Width="56" Height="17"
          Cancel="yes" Text="!(loc.WiXUICancel)">
          <Publish Event="SpawnDialog" Value="CancelDlg">1</Publish>
        </Control>

        <Control Id="Description" Type="Text" X="25" Y="23" Width="280" Height="15"
          Transparent="yes" NoPrefix="yes" Text="!(loc.SerialNumberDlgDescr)" />
        <Control Id="Title" Type="Text" X="15" Y="6" Width="200" Height="15"
          Transparent="yes" NoPrefix="yes" Text="!(loc.SerialNumberDlgTitle)" />
        <Control Id="BannerBitmap" Type="Bitmap" X="0" Y="0" Width="370" Height="44"
          TabSkip="no" Text="!(loc.SerialNumberDlgBannerBitmap)" />
        <Control Id="BannerLine" Type="Line" X="0" Y="44" Width="370" Height="0" />
        <Control Id="BottomLine" Type="Line" X="0" Y="234" Width="370" Height="0" />

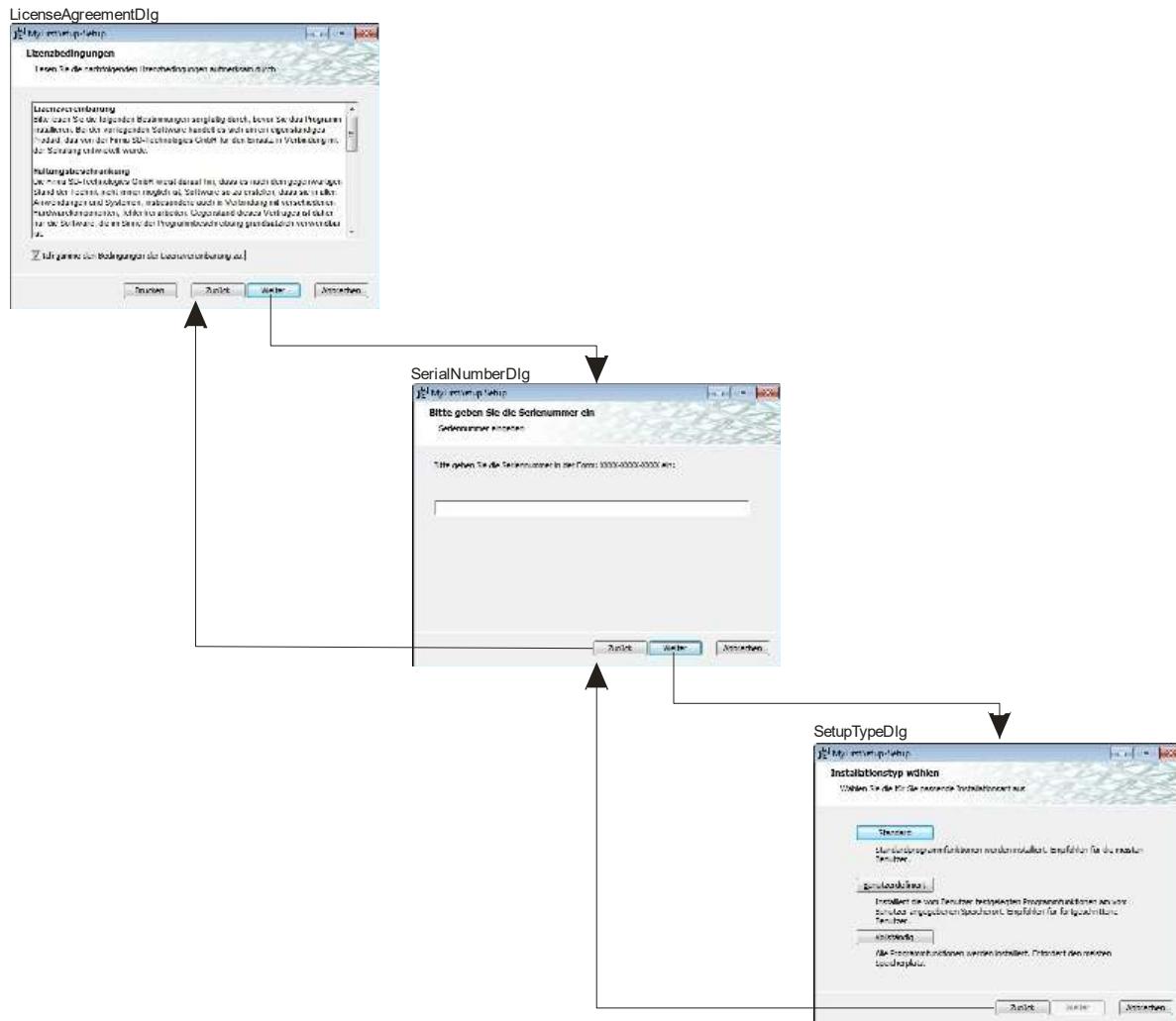
        <Control Id="Label" Type="Text" X="20" Y="60" Width="290" Height="30"
          NoPrefix="yes" Text="!(loc.SerialNumberDlgLabel)" />
        <Control Id="SerialNumberEdit" Type="Edit" X="20" Y="100" Width="320"
          Height="18" Property="SERIALNUMBER"/>
      </Dialog>
    </UI>
  </Fragment>
</WiX>
```

► Hinweis: Wir sehen: Beim Control *SerialNumberEdit* verbinden wir das Eingabefeld mit dem Property *SERIALNUMBER*. Beim Öffnen des Dialogs wird der Wert aus dem Property *SERIALNUMBER* in das Eingabefeld geschrieben. Nach dem Schließen des Dialogs wird der Wert wieder aus dem Eingabefeld ausgelesen und in das Property zurückgeschrieben. Und das passiert auch bei anderen Controls, wie z. B. bei Checkboxen, Radiobuttons, List- und Comboboxen.

Die Texte definieren wir in der Datei *SerialNumberDlg de-de.wxl*:

```
<?xml version="1.0" encoding="utf-8"?>
<WiXLocalization Culture="de-de"
  xmlns="http://schemas.microsoft.com/WiX/2006/localization">
  <String Id="SerialNumberDlg">[ProductName] Setup</String>
  <String Id="SerialNumberDlgDesc">Seriennummer eingeben</String>
  <String Id="SerialNumberDlgTitle">
    {\WixUI_Font_Title}Bitte geben Sie die Seriennummer ein
  </String>
  <String Id="SerialNumberDlgBannerBitmap">WixUI_Bmp_Banner</String>
  <String Id="SerialNumberDlgLabel">
    Bitte geben Sie die Seriennummer in der Form: XXXX-XXXX-XXXX ein:
  </String>
</WiXLocalization>
```

Bevor wir nun unseren Dialog einbinden, müssen wir generell verstehen, wie sich die Dialoge untereinander aufrufen. Folgendes Schaubild macht dies deutlich:



Wir sehen: Der „Weiter“-Button vom LicenseAgreementDlg ruft SerialNumberDlg auf. Um den Benutzer nicht zu verwirren, sollte der „Zurück“-Button von SerialNumberDlg wieder LicenseAgreementDlg aufrufen.

Unseren neuen Dialog müssen wir in die Dialogkette einbinden. Dazu erweitern wir den „Weiter“- bzw. den „Zurück“-Button unseres Dialoges:

```

<Control Id="Next" Type="PushButton" X="236" Y="243" Width="56" Height="17"
    Default="yes" Text="!(loc.WiXUINext)">
    <Publish Event="NewDialog" Value="SetupTypeDlg">1</Publish>
</Control>

<Control Id="Back" Type="PushButton" X="180" Y="243" Width="56" Height="17"
    Text="!(loc.WiXUIBack)">
    <Publish Event="NewDialog" Value="LicenseAgreementDlg">1</Publish>
</Control>

```

Nun müssen wir noch den „Weiter“- und den „Zurück“-Button aus der Dialogkette von **WixUI_Mondo** eintragen.

Hierzu kopieren wir uns folgenden Teil aus der Datei `src\ext\UIExtension\WiXlib\WixUI_Mondo.wxs` in unsere WSX-Datei:

```

<UI Id="WixUI_Mondo">
  <TextStyle Id="WixUI_Font_Normal" FaceName="Tahoma" Size="8" />
  <TextStyle Id="WixUI_Font_Bigger" FaceName="Tahoma" Size="12" />
  <TextStyle Id="WixUI_Font_Title" FaceName="Tahoma" Size="9" Bold="yes" />

  <Property Id="DefaultUIFont" Value="WixUI_Font_Normal" />
  <Property Id="WixUI_Mode" Value="Mondo" />

  <DialogRef Id="ErrorDlg" />
  <DialogRef Id="FatalError" />
  <DialogRef Id="FilesInUse" />
  <DialogRef Id="MsiRMFFilesInUse" />
  <DialogRef Id="PrepareDlg" />
  <DialogRef Id="ProgressDlg" />
  <DialogRef Id="ResumeDlg" />
  <DialogRef Id="UserExit" />

  <Publish Dialog="ExitDialog" Control="Finish" Event="EndDialog" Value="Return"
    Order="999">1</Publish>

  <Publish Dialog="WelcomeDlg" Control="Next" Event="NewDialog"
    Value="LicenseAgreementDlg">NOT Installed AND NOT PATCH</Publish>
  <Publish Dialog="WelcomeDlg" Control="Next" Event="NewDialog"
    Value="VerifyReadyDlg">Installed AND PATCH</Publish>

  <Publish Dialog="LicenseAgreementDlg" Control="Back" Event="NewDialog"
    Value="WelcomeDlg">1</Publish>
  <Publish Dialog="LicenseAgreementDlg" Control="Next" Event="NewDialog"
    Value="SetupTypeDlg" Order="2">LicenseAccepted = "1"</Publish>

  <Publish Dialog="SetupTypeDlg" Control="Back" Event="NewDialog"
    Value="LicenseAgreementDlg">1</Publish>
  <Publish Dialog="SetupTypeDlg" Control="TypicalButton" Event="NewDialog"
    Value="VerifyReadyDlg">1</Publish>
  <Publish Dialog="SetupTypeDlg" Control="CustomButton" Event="NewDialog"
    Value="CustomizeDlg">1</Publish>
  <Publish Dialog="SetupTypeDlg" Control="CompleteButton" Event="NewDialog"
    Value="VerifyReadyDlg">1</Publish>

  <Publish Dialog="CustomizeDlg" Control="Back" Event="NewDialog"
    Value="MaintenanceTypeDlg" Order="1">WixUI_InstallMode = "Change"</Publish>
  <Publish Dialog="CustomizeDlg" Control="Back" Event="NewDialog" Value="SetupTypeDlg"
    Order="2">WixUI_InstallMode = "InstallCustom"</Publish>
  <Publish Dialog="CustomizeDlg" Control="Next" Event="NewDialog"
    Value="VerifyReadyDlg">1</Publish>

  <Publish Dialog="VerifyReadyDlg" Control="Back" Event="NewDialog"
    Value="CustomizeDlg" Order="1">WixUI_InstallMode = "InstallCustom"</Publish>
  <Publish Dialog="VerifyReadyDlg" Control="Back" Event="NewDialog"
    Value="SetupTypeDlg" Order="2">
    WixUI_InstallMode = "InstallTypical" OR WixUI_InstallMode = "InstallComplete"
  </Publish>
  <Publish Dialog="VerifyReadyDlg" Control="Back" Event="NewDialog"
    Value="CustomizeDlg" Order="3">WixUI_InstallMode = "Change"</Publish>
  <Publish Dialog="VerifyReadyDlg" Control="Back" Event="NewDialog"
    Value="MaintenanceTypeDlg" Order="4">
    WixUI_InstallMode = "Repair" OR WixUI_InstallMode = "Remove"
  </Publish>

```

```
<Publish Dialog="VerifyReadyDlg" Control="Back" Event="NewDialog" Value="WelcomeDlg"
    Order="2">
    WixUI_InstallMode = "Update"
</Publish>

<Publish Dialog="MaintenanceWelcomeDlg" Control="Next" Event="NewDialog"
    Value="MaintenanceTypeDlg">1</Publish>

<Publish Dialog="MaintenanceTypeDlg" Control="ChangeButton" Event="NewDialog"
    Value="CustomizeDlg">1</Publish>
<Publish Dialog="MaintenanceTypeDlg" Control="RepairButton" Event="NewDialog"
    Value="VerifyReadyDlg">1</Publish>
<Publish Dialog="MaintenanceTypeDlg" Control="RemoveButton" Event="NewDialog"
    Value="VerifyReadyDlg">1</Publish>
<Publish Dialog="MaintenanceTypeDlg" Control="Back" Event="NewDialog"
    Value="MaintenanceWelcomeDlg">1</Publish>
</UI>

<UIRef Id="WixUI_Common" />
```

Nun können wir den „Weiter“-Button von Dialog LicenseAgreementDlg auf unseren Dialog setzen:

```
<Publish Dialog="LicenseAgreementDlg" Control="Next" Event="NewDialog"
    Value="SerialNumberDlg" Order="2">LicenseAccepted = "1"</Publish>
```

Interessant hier ist das Order-Attribut. Das Order-Attribut bestimmt die Ausführungsreihenfolge von Kommandos. Wir können also auf Events – hier das Drücken des „Next“-Buttons – mehrere Aktionen ausführen.

Nun noch den „Zurück“-Button von SetupTypeDlg auf unseren Dialog umleiten:

```
<Publish Dialog="SetupTypeDlg" Control="Back" Event="NewDialog"
    Value="SerialNumberDlg">1</Publish>
```

Dem Userinterface geben wir eine neue ID namens MyWiXUi, um dann in der Project-Datei auf das Userinterface verweisen zu können:

```
<UI Id="MyWiXUi">
    ...
</UI>
```

In der Datei Product.wxs ändern wir nun die Referenz des UserInterfaces von WixUI_Mondo auf MyWiXUi:

```
<UIRef Id="MyWiXUi" />
<UIRef Id="WixUI_ErrorProgressText" />
```

► **Hinweis:** Der letzte Wert beim **Publish**-Element ist die Bedingung für das Ausführen des Kommandos. Ist dieser Wert FALSE, so wird das Kommando nicht ausgeführt.

Als letzten Teil wollen wir die Seriennummer noch in die Registry ablegen. Hierzu definieren wir eine Komponente:

```
<Component Id='SerialNumber' Guid='YOURGUID' Directory='INSTALLDIR'>
  <RegistryKey Id='Hk1m_SerialNumber' Root='HKLM' Key='Software\[ProductName]' 
    ForceDeleteOnUninstall='yes'>
    <RegistryValue Type='string' Name='SerialNumber' Value='[SERIALNUMBER]' />
  </RegistryKey>
</Component>
```

Die Komponente fügen wir der ComponentGroup MainFeature zu:

```
<ComponentGroup Id ='MainFeature'>
  ...
  <ComponentRef Id='SerialNumber' />
</ComponentGroup>
```

Wenn wir das fertige Skript kompilieren, haben wir einen schönen neuen Dialog eingebaut. Natürlich wird die Seriennummer bis jetzt noch nicht überprüft. Das werden wir in einer späteren Lektion noch nachholen.

14 Billboards

Billboards sind Grafiken, die dem Benutzer bei der Installation nützliche Informationen übermitteln und den Installationsvorgang etwas kurzweiliger erscheinen lassen. Billboards werden in der Regel auf dem Dialog dargestellt, auf dem auch der Fortschritt der Installation mittels Fortschrittsbalken angezeigt wird. Wir können im Setup mehrere Billboards definieren, die dann mit dem Fortschritt der Installation wechseln.

Billboards werden über Billboard-Control auf dem Dialog platziert. Da das erste Billboard-Bild erst dann aktiviert wird, wenn die zugehörige Aktion startet, sollte man an derselben Stelle ein Anfangsbild als Bitmap-Control platzieren:

```
<Binary Id="MyStartPicture" src=".\\SourceDir\\MyStartPicture.bmp">

<Dialog Id="ProgressDlg" X="50" Y="50" Width="510" Height="481" Modeless="yes">
    ...
    <Control Id="BbCtrl" Type="Billboard" X="0" Y="0" Width="510" Height="39"
        TabSkip="no" Disabled="yes">
        <Subscribe Event="SetProgress" Attribute="Progress" />
    </Control>
    <Control Id="StartBitmap" Type="Bitmap" X="0" Y="0" Width="510" Height="390"
        Text="MyStartPicture" TabSkip="no" Disabled="yes" />
    ...
</Dialog>
```

Damit das Billboard-Control Nachrichten (Events) empfangen kann, muss das Control dies per **Subscribe**-Element anmelden. Das Attribut Event definiert, dass das Billboard auf die Nachricht SetProgress reagiert. SetProgress wird z. B. von der Aktion InstallFiles über MsiProcessMessage losgeschickt.

Zunächst laden wir die einzelnen Bilder in die Binär-Tabelle:

```
<Binary Id="BB1" SourceFile=".\\SourceDir\\bbrd1.bmp"/>
<Binary Id="BB2" SourceFile=".\\SourceDir\\bbrd2.bmp"/>
<Binary Id="BB3" SourceFile=".\\SourceDir\\bbrd3.bmp"/>
```

Über das **BillboardAction**-Element definieren wir, dass die Bilder mit dem Fortschritt der InstallFiles-Aktion wechseln sollen. Damit bestimmte Bilder nur dann angezeigt werden, wenn ein bestimmtes Feature installiert wird, geben wir den Namen des Features über das **Billboard**-Element an:

```
<UI>
    <BillboardAction Id="InstallFiles">
        <Billboard Id="Billboard1" Feature="MainFeature">
            <Control Id="BbCtrl" X="0" Y="0" Width="510" Height="39" Type="Bitmap" Text="BB1"/>
        </Billboard>
        <Billboard Id="Billboard2" Feature=" MainFeature ">
            <Control Id="BbCtrl" X="0" Y="0" Width="510" Height="39" Type="Bitmap" Text="BB2"/>
        </Billboard>
        <Billboard Id="Billboard3" Feature=" MainFeature ">
            <Control Id="BbCtrl" X="0" Y="0" Width="510" Height="39" Type="Bitmap" Text="BB3"/>
        </Billboard>
    </BillboardAction>
```

Innerhalb des Billboard-Elements definieren wir das zugehörige Bild über das Control-Element. Die ID des Control-Elements bezieht sich auf die ID des Billboard-Controls auf dem Dialog. Der Text vom Control-Element stellt eine Referenz auf die Binär-Tabelle dar.

15 Sequenzen

In den Lektionen zuvor haben wir gelernt, wie Dateien, Registry-Einträge und das Userinterface definiert werden. Dort wird definiert, was an welche Stelle geschrieben werden soll. In dieser Lektion werden wir uns darum kümmern, in welcher Reihenfolge die Aktionen ausgeführt werden.

Bevor wir uns mit den Sequenzen beschäftigen, wollen wir zuerst einmal klären, was eine Sequenz überhaupt ist. Eine Sequenz kann man sich wie ein Skript vorstellen, in dem einzelne Befehle (die sogenannten Actions) hintereinander aufgerufen werden. Die Aktionen in der Sequenz sind durchnummieriert und werden vor der Ausführung entsprechend sortiert. Aber: Es besteht ein kleiner Unterschied zwischen Skript und Sequenz. In der Sequenz gibt es keine Sprünge und keine Schleifen. Die Abarbeitung erfolgt also streng sequenziell.

Jede Aktion hat eine Bedingung. Man kann sich das so vorstellen, als wäre eine If-Anwendung um jede Aktion. Die Aktion wird nur dann ausgeführt, wenn die Bedingung erfüllt ist.

Grundsätzlich kennt der Windows Installer sechs Sequenzen:

Name	Beschreibung
InstallUISequence	Userinterface der Installation/Deinstallation und Reparatur
InstallExecuteSequence	Ausführen der Installation/Deinstallation und Reparatur
AdminUISequence	Userinterface der administrativen Installation
AdminExecuteSequence	Ausführen der administrativen Installation
AdvtUISequence	Userinterface einer Ankündigung (Advertisement)
AdvtExecuteSequence	Ausführen einer Ankündigung (Advertisement)

Beim Windows Installer ist grundsätzlich das Userinterface von der Ausführungs-Sequenz getrennt. Im Userinterface wird bestimmt, was wohin installiert werden soll und im Ausführenteil findet die tatsächliche Änderung am Zielrechner statt. Und das hat auch einen bestimmten Grund: Der Windows Installer unterstützt die Silent-Installation (also eine Installation, bei der der Anwender entweder nur den Fortschrittsbalken oder sogar überhaupt nichts von der Installation sieht). Wird das Setup nun silent gestartet, wird die Userinterface-Sequenz einfach übersprungen und gleich der Ausführenteil gestartet.

15.1.1 Installationssequenz

Die Sequenz **InstallUISequence** und **InstallExecuteSequence** werden immer bei einer Installation und der Deinstallation mit Benutzeroberfläche, also nicht im Silent Mode ausgeführt. Sie ist primär dafür zuständig, Eingaben der Benutzer über bestimmte Dialoge entgegenzunehmen.

Wurde die zu installierende Konfiguration festgelegt, so wird die Aktion **ExecuteAction** aufgerufen. Diese Aktion startet dann in einem andern Prozess die Sequenz **InstallExecuteSequence**.

15.1.2 Advertisement-Installation

Eine Advertisement-Installation wird auch angekündigte Installation genannt. Wird ein Softwarepaket angekündigt – hierbei werden nur die Einträge im Startmenü oder auf dem Desktop erstellt, die zugehörigen Dateien werden erst dann nachinstalliert, wenn sie auch tatsächlich benötigt werden – werden die Sequenzen **AdvtUISequence** und **AdvtExecuteSequence** ausgeführt.

Eine angekündigte Installation wird über folgende Kommandozeile gestartet:

```
msiexec.exe /jm <MSI-Datei>
```

15.1.3 Administrative Installation

Wird eine administrative Installation ausgeführt, werden die Sequenzen **AdminUISequence** und **AdminExecuteSequence** ausgeführt. Eine administrative Installation legt die Installationsdateien – also MSI-Datei und die zu installierenden Dateien, die in einer CAB-Datei gespeichert sind – in unkomprimierter Form ab. Eine administrative Installation benötigt man entweder dazu, Patches zu erstellen, oder dazu, eine Basisversion und ein Patch zu einer Installation zu vereinigen.

Eine administrative Installation wird über die Kommandozeile folgendermaßen gestartet:

```
msiexec.exe /a <MSI-Datei> TARGETDIR=<Ziel-Pfad>
```

Das Property **TARGETDIR** gibt hier den Zielpfad zu dem Ort an, an dem das Setup entpackt werden soll. Wenn wir uns das Directory-Element einmal genauer ansehen, erkennen wir, an welcher Stelle TARGETDIR verlinkt ist:

```
<Directory Id="TARGETDIR" Name="SourceDir">
  <Directory Id="ProgramFilesFolder">
    <Directory Id="INSTALLFOLDER" Name="MySample" />
  </Directory>
</Directory>
```

Wie wir sehen, ist TARGETDIR die Wurzel aller Verzeichnisse. Die Directory-Variablen geben also nicht nur die Zielverzeichnisse vor, sondern auch die Quellverzeichnisse. Am Beispiel von INSTALLFOLDER können wir das sehr schön ablesen. Alle Dateien, die als Zielverzeichnis INSTALLFOLDER haben, sind nach der administrativen Installation im Unterverzeichnis MySample zu finden. Den Namen dieses Unterverzeichnisses kann man über das Directory-Attribut SourceName ändern. Im nachfolgenden Beispiel werden die Dateien nicht mehr nach MySample, sondern nach *InstallFolder* entpackt. Am Zielverzeichnis ändert sich hierbei nichts.

```
<Directory Id="INSTALLFOLDER" Name="MySample" SourceName="InstallFolder" />
```

Eine Basisversion wird mit einem oder mehreren Patches über folgende Kommandozeile vereinigt:

```
msiexec.exe /a <MSI-Datei> /p <MSP-Datei>[;<MSP2-Datei>...]/ TARGETDIR=<Ziel-Pfad>
```

Im Zielpfad liegt nach dieser administrativen Installation ein Setup, das sowohl die Basis-Version sowie die angegebenen Patches installiert. Diese Vorgehensweise wird vor allem bei einer automatisierten Installation bevorzugt, da hier nur ein Setup gestartet werden muss.

15.1.4 Silent-Installation

Eine Silent-Installation ist hauptsächlich für eine automatisierte Installation interessant. Hier möchte man nicht, dass der Anwender in die laufende Installation durch Eingaben eingreifen kann. Um ein MSI-Setup im Silent-Mode zu starten, müssen wir msiexec.exe mit folgender Kommandozeile starten:

```
msiexec.exe /i <MSI-Datei> /qn
```

Die /q Option bestimmt den Level des Userinterfaces:

Option	Property UILevel	Beschreibung
/qf (Full UI)	5	Mit diesem Aufruf werden alle Dialoge dargestellt. Diese Option ist der Standard.
/qr (Reduced UI)	4	Mit diesem Aufruf werden zwar noch alle Dialoge angezeigt, es wird aber auf keine Eingabe mehr gewartet.
/qb (Basic UI)	3	Mit diesem Aufruf wird nur ein im Funktionsumfang beschränkter Fortschrittsbalken angezeigt. Die UI-Sequenz wird nicht ausgeführt.
/qb! (Basic UI)	3	Wie /qb, jedoch ohne Abbruchtaste.
/qn (None)	2	Mit diesem Aufruf werden gar keine Dialoge dargestellt. Die UI-Sequenz wird auch nicht ausgeführt. Dieser Parameter erfordert administrative Rechte, da die UAC hier nicht angezeigt wird.

15.1.5 InstallUISequence

Wenn wir unser Setup mit Orca öffnen und zur **InstallUISequence**-Tabelle gehen, dann sehen wir Folgendes:

Action	Condition	Sequence
FatalError		-3
UserExit		-2
ExitDialog		-1
PrepareDlg		49
AppSearch		50
LaunchConditions		51
ValidateProductID		700
CostInitialize		800
FileCost		900
CostFinalize		1000
MaintenanceWelcomeDlg	Installed AND NOT RESUME AND NOT Preselected AND NOT PATCH	1296
ResumeDlg	Installed AND (RESUME OR Preselected)	1297
WelcomeDlg	NOT Installed OR PATCH	1298
ProgressDialog		1299
ExecuteAction		1300

In der Spalte Action wird das Kommando an den Windows Installer gegeben, in Condition steht die Bedingung für die Ausführung und in Sequence steht die Reihenfolge der Befehle.

In der Spalte Sequence fällt einem gleich auf, dass es negative und positive Werte gibt. Alle Kommandos mit positiven Werten werden in der dort abgelegten Reihenfolge abgearbeitet. Negative Sequenznummern sind Events, die bei einem bestimmten Ereignis ausgeführt werden.

Folgende Events sind vom Windows Installer definiert:

Wert	Beschreibung
-1	Installation erfolgreich beendet.
-2	Installation von Benutzer abgebrochen.
-3	Installation wurde durch einen Fehler abgebrochen.
-4	Installation wurde ausgesetzt.

Als Kommandos können Standardaktionen (Aktionen, die vom Windows Installer definiert sind), Dialoge (sie werden im MSI-Setup über die Dialogtabelle u. A. definiert) und Custom-Actions (Aktionen, die im MSI über die **CustomAction**-Tabelle definiert sind) eingetragen werden.

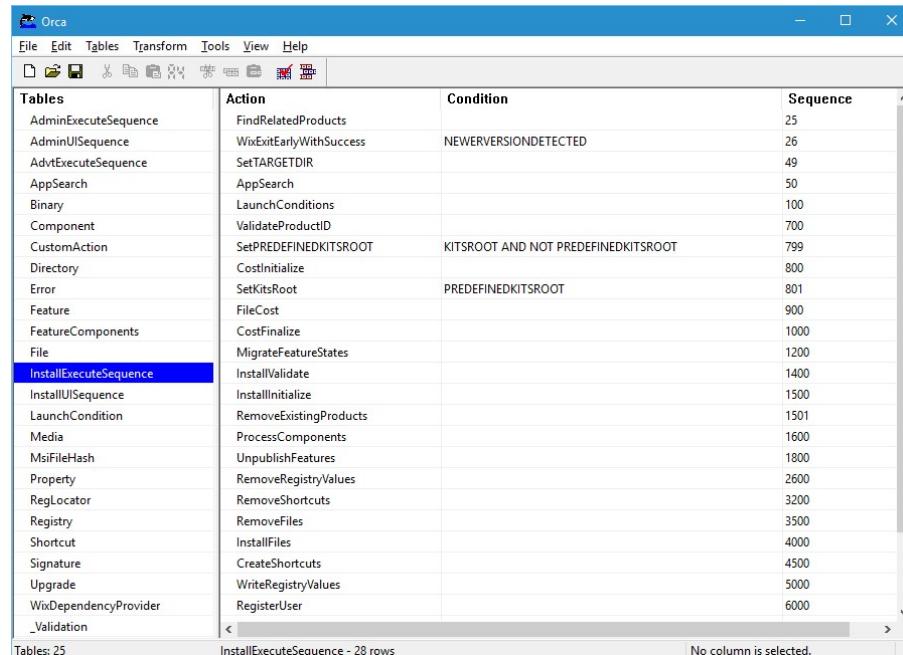
Nach der Berechnung des benötigten Speicherplatzes (Costing) sind vier Dialoge eingetragen:

Dialog	Bedeutung
MaintenanceWelcomeDlg	Wird aufgerufen, wenn das Setup bereits installiert ist und bietet dann die Reparatur und Deinstallation an
ResumeDlg	Wird aufgerufen, nachdem das Setup während der Installation einen Neustart durchgeführt hat
WelcomeDlg	Wird bei der Erstinstallation aufgerufen
ProgressDlg	Zeigt den Fortschrittsbalken für die Installation an

Nach den Dialogen wird die Aktion **ExecuteAction** aufgerufen. ExecuteAction führt dann die **InstallExecuteSequence** aus.

15.1.6 InstallExecuteSequence

Wenn wir mit Orca die Tabelle **InstallExecuteSequence** betrachten, dann sehen wir beispielweise folgende Einträge:



The screenshot shows the Orca tool interface with the 'Tables' tab selected. The 'InstallExecuteSequence' table is highlighted with a blue selection bar. The table has four columns: Action, Condition, Sequence, and a header row. The data rows show various actions like FindRelatedProducts, WinExitEarlyWithSuccess, SetTARGETDIR, AppSearch, LaunchConditions, ValidateProductID, SetPREDEFINEDKITSROOT, CostInitialize, SetKitsRoot, FileCost, CostFinalize, MigrateFeatureStates, InstallValidate, InstallInitialize, RemoveExistingProducts, ProcessComponents, UnpublishFeatures, RemoveRegistryValues, RemoveShortcuts, RemoveFiles, InstallFiles, CreateShortcuts, WriteRegistryValues, and RegisterUser, each with their respective sequence numbers.

Table	Action	Condition	Sequence
AdminExecuteSequence	FindRelatedProducts		25
AdminUISequence	WinExitEarlyWithSuccess	NEVERVERSIONDETECTED	26
AdvtExecuteSequence	SetTARGETDIR		49
AppSearch	AppSearch		50
Binary	LaunchConditions		100
Component	ValidateProductID		700
CustomAction	SetPREDEFINEDKITSROOT	KITSROOT AND NOT PREDEFINEDKITSROOT	799
Directory	CostInitialize		800
Error	SetKitsRoot	PREDEFINEDKITSROOT	801
Feature	FileCost		900
FeatureComponents	CostFinalize		1000
File	MigrateFeatureStates		1200
InstallExecuteSequence	InstallValidate		1400
InstallUISequence	InstallInitialize		1500
LaunchCondition	RemoveExistingProducts		1501
Media	ProcessComponents		1600
MsiFileHash	UnpublishFeatures		1800
Property	RemoveRegistryValues		2600
RegLocator	RemoveShortcuts		3200
Registry	RemoveFiles		3500
Shortcut	InstallFiles		4000
Signature	CreateShortcuts		4500
Upgrade	WriteRegistryValues		5000
WixDependencyProvider	RegisterUser		6000
_Validation			

In dieser Sequenz finden wir Aktionen wie RemoveShortcuts, RemoveFiles und RemoveFolders. Der Name verrät bereits, dass diese Aktionen Programmverknüpfungen, Dateien und Ordner löschen. Diese Aktionen werden in der Regel für die Deinstallation des Setups benötigt. Danach kommen Aktionen wie CreateFolders, InstallFiles und CreateShortcuts. Diese Aktionen sind hauptsächlich für die Installation und Reparatur zuständig.

Man würde erwarten, dass Aktionen für die Deinstallation eine Bedingung wie Remove=1 fordern und die Aktionen für die Installation so etwas wie Install=1 AND Repair=1. Das ist aber nicht der Fall. Warum das nicht so gemacht wurde, sehen wir im nächsten Abschnitt.

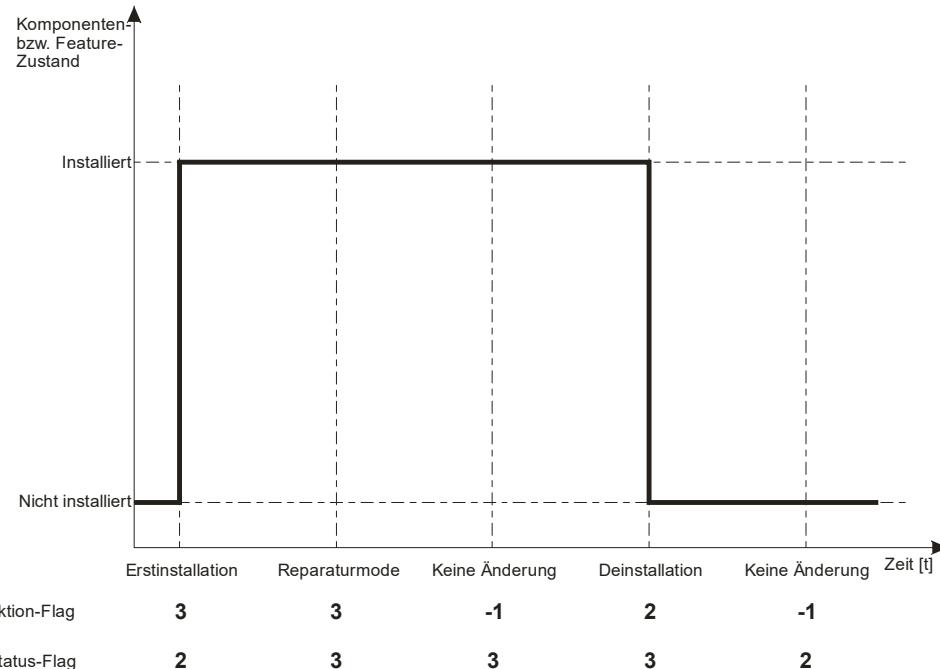
15.2 Status- und Aktion-Flag einer Komponente

Jeder Komponente (und auch jedes Feature) hat zwei Flags: Eine Status-Flag, die den Zustand der Komponente vor der Installation enthält (also quasi den Istzustand) und eine Aktion-Flag, die den gewünschten Zustand der Komponente nach der Installation anzeigt (also den Sollzustand).

Jede Flag kann folgende Zustände annehmen:

Installations-Zustand	Wert	Bedeutung
INSTALLSTATE_UNKNOWN	-1	Keine Aktion
INSTALLSTATE_ABSENT	2	Komponente ist nicht installiert bzw. soll deinstalliert werden.
INSTALLSTATE_LOCAL	3	Komponente ist installiert bzw. soll installiert werden.

Somit ergibt sich folgendes Schaubild:



Wir sehen, dass für jeden Zustand „Installation“, „Reparatur“ und „Deinstallation“ die Kombination mit Status- und Aktion-Flag unterschiedlich ist. Und genau das wird bei den Aktionen ausgewertet. InstallFiles sucht zuerst alle Komponenten, deren Aktion-Flag den Wert 3 (also „installieren“) hat. Danach werden alle Dateien, die mit diesen Komponenten verbunden sind, installiert. RemoveFiles hingegen sucht alle Komponenten, deren Aktion-Flag den Wert 2 (also „deinstallieren“) hat und löscht alle Dateien, die mit diesen Komponenten verbunden sind.

15.3 Die Sequenzen im WiX-Skript

Mit dem Element **InstallExecuteSequence** bzw. **InstallUISequence** können Aktionen in die Installations- bzw. Userinterface-Sequenz aufgenommen werden. Wollen wir die Aktion *MyAction* in diese Sequenzen einfügen, dann sieht das folgendermaßen aus:

```
<InstallExecuteSequence>
  <Custom Action='MyAction' After='AppSearch' />
</InstallExecuteSequence>

<InstallUISequence>
  <Custom Action='MyAction' After='AppSearch' />
</InstallUISequence>
```

Andere mögliche Attribute sind: **Before** (Aktion wird vor der genannten Aktion eingefügt) oder **Suppress = yes** (Aktion wird aus der Sequenz entfernt).

Es gibt noch viele Standardaktionen, die wir in unsere Sequenzen einbinden können. Zum Beispiel können wir über die Standardaktion **ScheduleReboot** das System am Ende der Installation neu starten lassen:

```
<InstallExecuteSequence>
  <ScheduleReboot After='InstallFinalize' />
</InstallExecuteSequence>
```

Es könnte jetzt auch sein, dass wir den Reboot nur dann durchführen wollen, wenn ein bestimmtes Feature oder eine bestimmte Komponente installiert wird. In diesem Fall müssen wir die Status- und Action-Flag der Komponente (des Features) in der Bedingung abfragen.

Hierzu gibt es Präfixes zum Komponenten- bzw. Feature-Namen, die uns genau diese Flags zurückliefern:

Präfix	Funktion
?	Status-Flag einer Komponente
\$	Action-Flag einer Komponente
&	Action-Flag eines Features
!	Status-Flag eines Features

► **Hinweis:** Die oben dargestellten Operatoren dürfen erst nach der Aktion **CostFinalize** benutzt werden, da die Flags erst zu diesem Zeitpunkt valide sind.

Wollen wir z. B. immer dann einen Neustart des Systems initiieren, wenn die Komponente *MyRebootComponent* das erste Mal installiert wird, dann sieht das in etwa so aus:

```
<InstallExecuteSequence>
  <ScheduleReboot After='InstallFinalize'>
    ?MyRebootComponent=2 AND $MyRebootComponent=3
  </ScheduleReboot>
</InstallExecuteSequence>
```

16 Eigene Custom-Action erstellen

Manchmal benötigen wir Funktionen, die im Windows Installer Standard nicht definiert sind. Wir müssen also die Funktionalität des Windows Installers erweitern – und das macht man mit einer sogenannten Custom-Action. Über Custom-Actions können Properties und Directory-Variablen gesetzt werden, EXE-Dateien, DLL-Funktionen, VB-Skript oder JavaScript Code ausgeführt und sogar Managed-Code-DLLs gestartet werden. Wie das geht, sehen wir in diesem Kapitel.

16.1 Property in Custom-Action setzen

Eine der einfachsten Arten einer Custom-Action ist es, Properties zu setzen. Als Beispiel wollen wir das Property SERIAL_NUMBER auf einen Default-Wert setzen, wenn das Property Evaluation (das könnte z. B. über einen Dialog gesetzt werden) den Wert 1 hat.

Dafür definieren wir zunächst über das Element **CustomAction** eine Custom-Action, die wir dann anschließend in die InstallUISequenz einbinden. Damit das Property SERIAL_NUMBER von der UI- in die Execute-Sequenz übernommen wird, setzen wir das Attribut Secure auf yes.:.

```
<Property Id="SERIAL_NUMBER" Secure="yes"/>

<CustomAction Id="SetEvalSerialNo" Property="SERIAL_NUMBER" Value="1234-ABCD-[Date]"/>
<InstallUISequence>
  <Custom Action="SetEvalSerialNo" Before="ExecuteAction">
    Evaluation=1
  </Custom>
</InstallUISequence>
```

Wie wir im obigen Beispiel sehen, können wir im Wert des Properties auch andere Properties (hier das Property Date, welches das aktuelle Datum enthält) über eckige Klammern zu referenzieren. Vor dem Setzen werden diese Ausdrücke aufgelöst, sodass im Property so etwas wie 1234-ABCD-29.06.2015 steht.

Das WiX-Toolset erlaubt uns aber auch, Properties in einer Kurzschreibweise zu setzen. Folgende Zeilen erreichen genau dasselbe wie das Beispiel zuvor:

```
<Property Id="SERIAL_NUMBER" Secure="yes"/>
<SetProperty Id="SERIAL_NUMBER" Before="ExecuteAction" Sequence="ui"
  Value="1234-ABCD-[Date]">
  Evaluation=1
</ SetProperty>
```

16.2 Directory in Custom-Action setzen

Ähnlich wie das Setzen von Properties sieht das Ganze von Directory-Variablen aus. Hier muss man jedoch beachten, dass das Setzen der Directory-Variablen erst nach der Action **CostFinalize** stattfinden darf da diese ja erst mit CostFinalize initialisiert werden:

```
<CustomAction Id="SetInstallfolder" Directory="INSTALLFOLDER" Execute="firstSequence"
  Value="[ProgramFilesFolder]MyNewFolder" />

<!-- Insert custom action -->
<InstallUISequence>
  <Custom Action="SetInstallfolder" After="CostFinalize">
    Not Installed AND SetNewFolder=1
  </Custom>
</InstallUISequence>
```

```
<InstallExecuteSequence>
  <Custom Action="SetInstallfolder" After="CostFinalize">
    Not Installed AND SetNewFolder=1
  </Custom>
</InstallExecuteSequence>
```

Im obigen Beispiel wird das neue Zielverzeichnis nur dann gesetzt, wenn das Property SetNewFolder den Wert 1 hat.

Da das Zielverzeichnis noch vor den Dialogen gesetzt wird, kann das Zielverzeichnis in den Dialogen eventuell geändert werden. Damit das Verzeichnis in der Execute-Sequenz nicht wieder auf den Default zurückgesetzt wird, haben wir bei der Custom-Action das Attribut Execute auf *firstSequence* gesetzt. Execute auf *firstSequence* bewirkt, dass die Custom-Action beim ersten Aufruf (UI-Sequenz) ausgeführt wird, beim zweiten Aufruf (Execute-Sequenz) aber unterdrückt wird. Nun könnte man denken, dass man den zweiten Aufruf ja auch ganz weglassen könnte. Das wäre allerdings nicht sinnvoll: Denn bei einer Silent-Installation wird die UI-Sequenz gar nicht durchlaufen – und daher würde das Zielverzeichnis überhaupt nicht geändert.

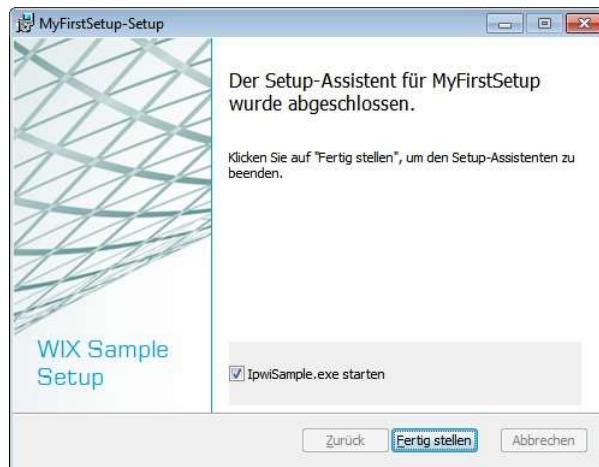
Auch für das Ändern von Directory-Variablen gibt es beim WiX-Toolset eine Kurzform in Form des **SetDirectory** Elements:

```
<SetDirectory Id="INSTALLFOLDER" Value="[ProgramFilesFolder]MyNewFolder"
  Sequence="first">
  Not Installed AND SetNewFolder=1
</SetDirectory>
```

Einen kleinen Unterschied gibt es da aber doch: Die obigen Zeilen setzen in Wirklichkeit keine Verzeichnisvariable, sondern ein gleichnamiges Property. Die Action wird deshalb automatisch vor CostFinalize eingebunden. Wie wir ja schon wissen, werden Verzeichnisvariablen, die ein gleichnamiges Property haben, automatisch bei CostFinalize auf den Wert des Properties gesetzt – somit ist der Effekt derselbe. Allerdings kann die Position von SetDirectory nicht explizit angegeben bzw. geändert werden.

16.3 EXE in Custom-Action aufrufen

In dieser Lektion wollen wir den Anwender in die Lage setzen, auf dem Dialog **ExitDialog** (der letzte Dialog bei einer erfolgreichen Installation) unsere Anwendung zu starten:



Glücklicherweise bietet uns die WixUI_Mondo-Library schon diese Möglichkeit. Sehen wir uns die Quelldatei ExitDialog.wxs aus dem WiX-Quellverzeichnis \src\ext\UIExtension\WiXlib an, so treffen wir auf folgende Definition:

```
<Dialog Id="ExitDialog" Width="370" Height="270" Title="!(loc.ExitDialog_Title)">
    <Control Id="Finish" Type="PushButton" X="236" Y="243" Width="56" Height="17"
        Default="yes" Cancel="yes" Text="!(loc.WiXUIFinish)" />
    <Control Id="Cancel" Type="PushButton" X="304" Y="243" Width="56" Height="17"
        Disabled="yes" Text="!(loc.WiXUICancel)" />
    <Control Id="Bitmap" Type="Bitmap" X="0" Y="0" Width="370" Height="234" TabSkip="no"
        Text="!(loc.ExitDialogBitmap)" />
    <Control Id="Back" Type="PushButton" X="180" Y="243" Width="56" Height="17"
        Disabled="yes" Text="!(loc.WiXUIBack)" />
    <Control Id="BottomLine" Type="Line" X="0" Y="234" Width="370" Height="0" />
    <Control Id="Description" Type="Text" X="135" Y="70" Width="220" Height="40"
        Transparent="yes" NoPrefix="yes" Text="!(loc.ExitDialogDescription)" />
    <Control Id="Title" Type="Text" X="135" Y="20" Width="220" Height="60"
        Transparent="yes" NoPrefix="yes" Text="!(loc.ExitDialogTitle)" />
    <Control Id="OptionalText" Type="Text" X="135" Y="110" Width="220" Height="80"
        Transparent="yes" NoPrefix="yes" Hidden="yes"
        Text="[WIXUI_EXITDIALOGOPTIONALTEXT]">
        <Condition Action="show">
            WIXUI_EXITDIALOGOPTIONALTEXT AND NOT Installed
        </Condition>
    </Control>
    <Control Id="OptionalCheckBox" Type="CheckBox" X="135" Y="190" Width="220"
        Height="40" Hidden="yes" Property="WIXUI_EXITDIALOGOPTIONALCHECKBOX"
        CheckBoxValue="1" Text="[WIXUI_EXITDIALOGOPTIONALCHECKBOXTXT]">
        <Condition Action="show">
            WIXUI_EXITDIALOGOPTIONALCHECKBOXTXT AND NOT Installed
        </Condition>
    </Control>
</Dialog>
```

Interessant für uns ist vor allem der Abschnitt:

```
<Control Id="OptionalCheckBox" Type="CheckBox" X="135" Y="190" Width="220"
    Height="40" Hidden="yes" Property="WIXUI_EXITDIALOGOPTIONALCHECKBOX"
    CheckBoxValue="1" Text="[WIXUI_EXITDIALOGOPTIONALCHECKBOXTXT]">
    <Condition Action="show">
        WIXUI_EXITDIALOGOPTIONALCHECKBOXTXT AND NOT Installed
    </Condition>
</Control>
```

Dieser Abschnitt sagt uns, dass es bereits eine CheckBox gibt, deren Attribut Hidden auf yes gesetzt ist und die mit einer Bedingung versehen ist. Die Bedingung *show* wird aktiviert, wenn das Property **WIXUI_EXITDIALOGOPTIONALCHECKBOXTXT** definiert ist. Dieses Property wird auch als Text der Checkbox verwendet. Wir müssen also nur dieses Property mit einem Text versehen. Da wir die IpwiSample.exe starten wollen, machen wir das in Product.wxs:

```
<Property Id="WIXUI_EXITDIALOGOPTIONALCHECKBOXTXT" Value="!(loc.RunApplication)" />
```

Da es sich um einen Ausgabetext handelt, verweisen wir selbstverständlich auf unser Localisation-File. Der deutsche Text sieht dann so aus:

```
<String Id="RunApplication">IpwiSample.exe starten</String>
```

Die Checkbox selbst ist mit dem Property WIXUI_EXITDIALOGOPTIONALCHECKBOX verbunden. Wenn wir also wollen, dass die Checkbox standardmäßig gesetzt ist, müssen wir in dieses Property den Wert 1 eintragen:

```
<Property Id="WIXUI_EXITDIALOGOPTIONALCHECKBOX" Value="1" />
```

So, jetzt ist alles dafür bereit, dass wir unsere Anwendung starten können. Dazu schreiben wir nun unsere erste Custom-Action:

```
<CustomAction Id="LaunchApp" FileKey="IpwiSample.exe" ExeCommand="" Return="asyncNoWait"/>
```

Eine Custom-Action wird über das Element **CustomAction** erstellt – o magisches WiX. Diese Custom-Action können wir dann über die angegebene Id aufrufen.

Das Attribut **FileKey** verweist auf das **File**-Element, in der sich die IpwiSample.exe selbst befindet – also irgendwo in einer Komponente. Deshalb brauchen wir hier auch keinen Pfad angeben – der Windows Installer weiß ja, wohin er die Datei installiert hat. Das Attribut **ExeCommand** kann eine optionale Kommandozeile enthalten, muss aber auf jeden Fall hier definiert sein. Das Attribut **Return** besagt hier, dass die Anwendung asynchron – also parallel zum Setup – aufgerufen werden soll und dass das Setup auch nicht auf einen Rückgabewert der Exe wartet.

Jetzt müssen wir die Custom-Action LaunchApp noch aufrufen. Das machen wir am einfachsten direkt über den „Fertigstellen“-Button auf dem Dialog:

```
<UI>
  <Publish Dialog="ExitDialog" Control="Finish" Event="DoAction" Value="LaunchApp" Order="1">
    Not Installed AND WIXUI_EXITDIALOGOPTIONALCHECKBOX=1
  </Publish>
</UI>
```

Die Bedingung „Not Installed“ besagt, dass die Custom-Action nur bei der Erstinstallation aufgerufen wird und die Checkbox angewählt ist (WIXUI_EXITDIALOGOPTIONALCHECKBOX=1).

Der Vollständigkeit halber wollen wir hier noch zeigen, wie eine externe Anwendung aufgerufen wird, also eine Anwendung, die nicht mit dem Setup installiert wurde. Wir wollen Notepad.exe aufrufen und wissen, dass diese Anwendung im Windows-Verzeichnis abgelegt wird. Auf das Windows-Verzeichnis gibt es ein Windows Installer Property namens **WindowsFolder**. So können wir einfach ein entsprechendes Directory-Element definieren:

```
<DirectoryRef Id="TARGETDIR">
  <Directory Id="WindowsFolder" Name="Windows Folder" />
</DirectoryRef>
```

und so anschließend die Custom-Action definieren:

```
<CustomAction Id="LaunchNotepad" Directory="WindowsFolder" ExeCommand="Notepad.exe" Return="asyncNoWait" />
```

Damit wir die Notepad.exe ausführen können, müssen wir die Custom-Action LaunchNotepad über den „Fertigstellen“-Button auf dem Dialog aufrufen:

```
<UI Id="MyWixUI_Mondo">
  ...
  <Publish Dialog="ExitDialog" Control="Finish" Event="DoAction" Value="LaunchNotepad" Order="1">
    Not Installed
  </Publish>
</UI>
```

Alternativ könnten wir auch ein Property mit dem Namen der Anwendung definieren und Notepad über das Property aufrufen. Notepad würde dann über den Standardsuchpfad von Windows gefunden:

```
<Property Id="NOTEPAD">Notepad.exe</Property>
<CustomAction Id="LaunchNotepad" Property="NOTEPAD" ExeCommand=""
    Return="asyncNoWait" />
```

Manchmal braucht man die aufzurufende Anwendung nur für das Setup selbst. Die ausführbare Datei soll jedoch dem Endbenutzer nicht zugänglich sein. In diesem Fall können wir die EXE in die Binary-Tabelle ablegen und diese über das Id-Attribut des Binary-Elements aufrufen:

```
<Binary Id="DongleDriver" SourceFile="DongleSetup.exe" />
<CustomAction Id="LaunchNotepad" BinaryKey="DongleDriver" ExeCommand="/s"
    Return="ignore" />
```

Wenn wir die Custom-Action in eine Sequenz einbinden wollen, dann machen wir das über das **Custom**-Element:

```
<InstallExecuteSequence>
    <Custom Action="LaunchNotepad" After="InstallFinalize">NOT Installed</Custom>
</InstallExecuteSequence>
```

16.4 C#-Custom-Action

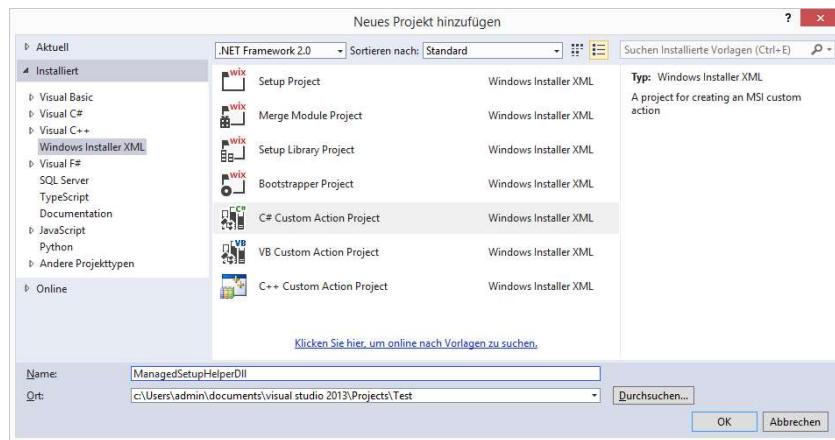
Viele Entwickler arbeiten mit einer .NET-Programmiersprache und möchten deshalb eine Custom-Action in C# erstellen. Mit der Einführung der **Deployment Tools Foundation (DTF)**, das eine umfangreiche .NET-Klassenbibliothek für die Setup-Entwicklung umfasst, wird es auch für den Windows Installer möglich, Managed Code in Custom-Actions auszuführen.

Das war die gute Nachricht. Die schlechte Nachricht ist: Um Managed Code auszuführen, muss natürlich das .NET-Framework auf dem Zielrechner vorhanden sein. Wir müssen also immer sicherstellen, dass sich zur Zeit der Installation (und auch zur Zeit der Deinstallation, falls die Deinstallation auch Custom-Actions als Managed Code ausführt) das .NET-Framework auf dem Zielrechner befindet. Ist das nicht der Fall, so wird die Custom-Action und eventuell sogar das gesamte Setup nicht richtig arbeiten.

16.4.1 Ein neue C#-Klassenbibliothek erstellen

Wir wollen nun eine Custom-Action in C# erstellen. Die Custom-Action soll nicht viel machen, sie soll nur das Zielverzeichnis auslesen und in einer MessageBox ausgeben.

Als ersten Schritt erstellen wir ein neues Projekt in der Solution. Hierzu gehen wir in Visual Studio auf die Projektmappe und fügen ein neues Projekt hinzu.



Als Vorlage nehmen wir aus den Windows Installer XML-Vorlagen das **C# Custom Action- Project** und nennen das Projekt z. B. ManagedSetupHelperDll.

Da unsere Custom-Action auf .NET Framework 2.0 aufbauen soll, wählen wir das in der oberen Leiste des Dialogs aus:

► **Hinweis:** Man sollte sich gut überlegen, welches .NET-Framework man für die Custom-Action zugrunde legt. Es sollte immer das Framework gewählt werden, das auf dem Zielsystem am häufigsten angetroffen wird bzw. mit dem die Anwendung, die man installieren möchte, auch arbeitet.

Nachdem das Projekt erstellt worden ist, finden wir in der erstellten Quelldatei bereits folgenden Funktionsrumpf:

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Deployment.WindowsInstaller;

namespace ManagedSetupHelperDll
{
    public class CustomActions
    {
        [CustomAction]
        public static ActionResult CustomAction1(Session session)
        {
            session.Log("Begin CustomAction1");

            return ActionResult.Success;
        }
    }
}
```

Über das Attribut *[CustomAction]* (oberhalb der Funktion) definieren wir, dass die nachfolgende Funktion als MSI-Custom-Action aufgerufen werden soll. Als Übergabeargument bekommen wir ein Session-Objekt der Deployment Tools Foundation zur Verfügung gestellt. Eine Beschreibung der Session-Klasse findet man in der DTF-Dokumentation, die mit dem WiX-Toolset installiert wird.

► **Hinweis:** Das Session-Objekt der Deployment Tools Foundation unterscheidet sich grundlegend vom Session-Objekt, das bei einer VBScript- bzw. Java-Script-Custom-Action zur Verfügung gestellt wird.

16.4.1.1 Funktion anpassen

Wir nennen die Funktion CustomAction1 in ShowInstallDir um und lesen die Directory-Variable INSTALLFOLDER mit der Funktion **GetTargetPath** aus:

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Deployment.WindowsInstaller;
using System.Windows.Forms;

namespace ManagedSetupHelperDll
{
    public class CustomActions
    {
```

```
[CustomAction]
public static ActionResult ShowInstallDir(Session session)
{
    string InstallDir;

    session.Log("Begin ShowInstallDir");
    InstallDir = session.GetTargetPath("INSTALLFOLDER");
    MessageBox.Show("INSTALLFOLDER: " + InstallDir);

    return ActionResult.Success;
}
}
```

Da wir das Installationsverzeichnis über eine MessageBox ausgeben, müssen wir einen Verweis auf System.Windows.Forms hinzufügen.

16.4.1.2 Build-Vorgang

Wenn wir unser Projekt kompilieren, bekommen wir im Ausgabeverzeichnis folgende Dateien:

Documents library				
Debug				
Name	Date modified	Type	Size	
ManagedSetupHelperDll.CA.dll	15.11.2012 08:31	Application extens...	205 KB	
ManagedSetupHelperDll.dll	15.11.2012 08:30	Application extens...	5 KB	
ManagedSetupHelperDll.pdb	15.11.2012 08:30	Program Debug D...	12 KB	
Microsoft.Deployment.WindowsInstaller.dll	03.09.2012 07:44	Application extens...	176 KB	
Microsoft.Deployment.WindowsInstaller.xml	03.09.2012 07:44	XML Document	475 KB	

Die Datei Microsoft.Deployment.WindowsInstaller.dll enthält die Klassenbibliothek der Deployment Tools Foundation. Die Datei ManagedSetupHelperDll.dll stellt die DLL mit Managed Code dar. Aber es wird noch eine weitere DLL erstellt: die ManagedSetupHelperDll.CA.dll. Diese ist deutlich größer als ManagedSetupHelperDll.dll und bindet die Datei Microsoft.Deployment.WindowsInstaller.dll mit der Datei ManagedSetupHelperDll.dll zusammen. Da der Windows Installer nur Win32-DLLs als Custom-Action aufrufen kann, wird in die CA-DLL auch noch Code eingebunden, der den Brückenschlag von einem Win32-DLL-Aufruf zu Managed Code herstellt.

Die CA-DLL wird über das Deployment-Tools-Foundation-Tool **MakeSfxCA.exe** erstellt.

16.4.2 Custom-Action einbinden

Um die Manage-Code-DLL einzubinden, erstellen wir zunächst eine Referenz auf das C#-Projekt. Damit können wir mit Projekt-Referenz-Variablen die DLL in die Binärtabelle einbinden:

```
<Binary Id="CSharpDll" SourceFile="$(var.ManagedSetupHelperDll.TargetDir)
$(var.ManagedSetupHelperDll.TargetName).CA$(var.ManagedSetupHelperDll.TargetExt)" />
```

Da die Projektvariable *TargetPath* auf die Managed-Code- und nicht auf die Win32-DLL zeigt, können wir diese hier nicht verwenden. Stattdessen nehmen wir den Pfad zur DLL über die Variable *Targetdir* und fügen ihr den Dateinamen über *TargetName* und die Extension über *TargetExt* ein. Zwischen Dateiname und Extension fügen wir den Text .CA ein. Der Rest sieht genauso aus wie bei einer Win32-DLL:

```
<CustomAction Id="SimpleCSharpFunction" BinaryKey="CSharpDll" DllEntry="ShowInstallDir"
Return="check" />

<InstallUISequence>
    <Custom Action="SimpleCSharpFunction" Before="ExecuteAction" />
</InstallUISequence>
```

16.5 VB.NET-Custom-Action

Das WiX-Toolset unterstützt auch über die **Deployment Tools Foundation (DTF)** den Aufruf von **VB.NET-Custom-Actions**.

16.5.1 Ein neue VB.NET-Klassenbibliothek erstellen

Wir wollen nun eine Custom-Action in VB.NET erstellen. Die Action soll nicht viel machen, sie soll nur ein Property lesen und ein anderes Property setzen.

Für die Bereitstellung geht man genauso vor wie bei C++-Custom-Actions und C#-Custom-Actions. Das WiX-Toolset hat auch für VB.NET eine Vorlage namens *VB Custom Action Project*, die wir verwenden können.

Nachdem das Projekt erstellt worden ist, finden wir in der erstellten Quelldatei bereits folgenden Funktionsrumpf:

```
Public Class CustomActions
    <CustomAction()> _
    Public Shared Function CustomAction1(ByVal session As Session) As ActionResult
        session.Log("Begin CustomAction1")

        Return ActionResult.Success
    End Function
End Class
```

16.5.1.1 Funktionen anpassen

Das Element <CustomAction()> über der Funktion *CustomAction1* zeigt dem Compiler, dass die Funktion als Custom-Action aufgerufen werden soll. Die Funktion erhält als Schnittstelle ein Session-Objekt der Deployment Tools Foundation zur Verfügung gestellt. In der DTF-Dokumentation, die mit dem WiX-Toolset installiert wird, findet man eine Beschreibung der Session-Klasse.

Die Funktion *CustomAction1* nennen wir um in *VBCustomAction* und passen diese wie folgt an:

```
Public Class CustomActions
    <CustomAction()> _
    Public Shared Function VBCustomAction(ByVal session As Session) As ActionResult
        session.Log("Begin VBCustomAction")

        Dim sLogonUser As String
        sLogonUser = session("LogonUser")

        'Username ausgeben in MessageBox
        MsgBox("Usernamen: " + sLogonUser, vbOKOnly, "VB.NET Custom Action")

        Return ActionResult.Success
    End Function
End Class
```

16.5.1.2 Build-Vorgang

Wenn wir unser Projekt kompilieren, bekommen wir im Ausgabeverzeichnis folgende Dateien:

Name	Typ	Größe
Microsoft.Deployment.WindowsInstaller.dll	Anwendungserweiterung	180 KB
Microsoft.Deployment.WindowsInstaller.xml	XML-Datei	476 KB
VBCustomAction.CA.dll	Anwendungserweiterung	218 KB
VBCustomAction.dll	Anwendungserweiterung	23 KB
VBCustomAction.pdb	Program Debug Database	32 KB

Die Datei Microsoft.Deployment.WindowsInstaller.dll enthält die Klassenbibliothek der Deployment Tools Foundation. Die Datei VBCustomAction.dll stellt die DLL mit VB.NET-Code dar. Aber es wird noch eine weitere DLL erstellt: die VBCustomAction.CA.dll. Diese ist deutlich größer als VBCustomAction.dll und bindet die Datei Microsoft.Deployment.WindowsInstaller.dll mit der VBCustomAction.dll zusammen. Da der Windows Installer nur Win32-DLLs als Custom-Action aufrufen kann, wird in die CA-DLL auch noch Code eingebunden, der den Brückenschlag von einem Win32-DLL-Aufruf zu Managed Code herstellt.

16.5.2 Custom-Action einbinden

Nun können wir die DLL im WiX-Code einbinden und als Custom-Action aufrufen. Um die DLL über Projekt-Referenz-Variablen einbinden zu können, fügen wir das VB.NET-Projekt als Referenz in unser WiX-Projekt ein. Danach können wir die DLL in die Binär-Tabelle einbinden, die Custom-Action erstellen und in die Sequenz einbauen:

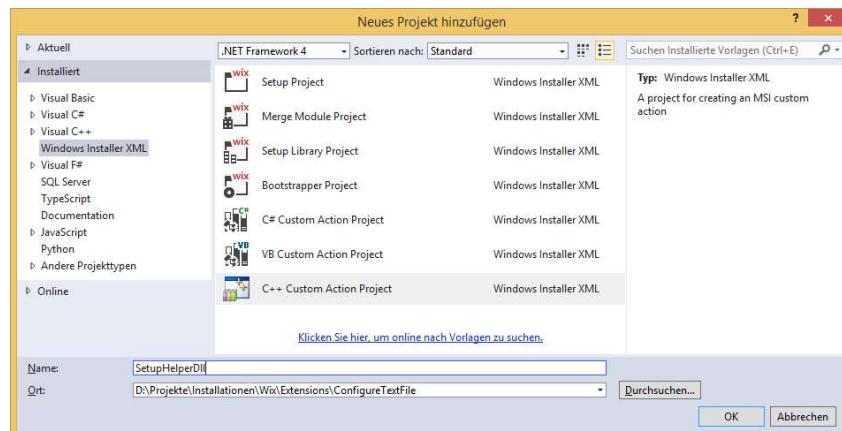
```
<Binary Id="VbCa" SourceFile="$(var.VBCustomAction.TargetDir)\VBCustomAction.CA.DLL" />
<CustomAction Id="VBCa" BinaryKey="VbCa" DllEntry="VBCustomAction" />
<InstallUISequence>
  <Custom Action="VBCa" After="CostFinalize" />
</InstallUISequence>
```

16.6 C++-Custom-Action

C++ stellt eine sehr interessante Alternative zu den vorher genannten Programmiersprachen dar. Als Beispiel wollen wir eine simple Custom-Action erstellen, die den Installationspfad in einer MessageBox ausgibt.

16.6.1 Eine neue C++-DLL erstellen

Als Erstes müssen wir ein neues Projekt zu unserer Projektmappe hinzufügen. Dazu gehen wir mit der rechten Maustaste auf die Projektmappe und wählen den Menüpunkt **Hinzufügen▶ Neues Projekt**.



Nun wählen wir in den Windows Installer XML-Vorlagen den Projekttyp **C++ Custom Action Project** aus und geben als Namen **SetupHelperDll** ein:

Visual Studio erstellt uns nun ein Programmgerüst, über das wir unsere Custom-Action erstellen können:

```
UINT __stdcall CustomAction1(MSIHANDLE hInstall)
{
    HRESULT hr = S_OK;
    UINT er = ERROR_SUCCESS;

    hr = WcaInitialize(hInstall, "CustomAction1");
    ExitOnFailure(hr, "Failed to initialize");

    WcaLog(LOGMSG_STANDARD, "Initialized.");

    // TODO: Add your custom action code here.
LExit:
    er = SUCCEEDED(hr) ? ERROR_SUCCESS : ERROR_INSTALL_FAILURE;;
    return WcaFinalize(er);
}
```

Ganz oben sehen wir eine für den Windows Installer typische Funktionsdeklaration. Als einziges Übergabeargument bekommen wir eine Variable vom Typ MSIHANDLE. Diese Variable stellt das Fenster zu unserem Setup dar. Über dieses Handle können wir Windows Installer API-Funktionen aufrufen, um z.B. Properties zu lesen.

Im Programmgerüst finden wir ein paar Funktionen, die mit **Wca** beginnen (Wca steht für WiX-Custom-Action). Diese Funktionen kommen aus der Library **dutil.lib**, die eine Bibliothek aus getesteten Funktionen darstellt, über die auf das MSI-Setup zugegriffen werden kann. Zum Beispiel können wir über die Funktion **WcaLog** in das Logfile des Setups schreiben.

16.6.2 Funktion anpassen

Als ersten Schritt passen wir den Namen der Funktion an. Da wir das Installationsverzeichnis ausgeben wollen, nennen wir die Funktion ShowInstallDir. Diesen Namen ändern wir nicht nur in der Cpp-Datei, auch in der Def-Datei (in unserem Beispiel also die CustomAction.def) müssen wir ihn anpassen. Die Def-Datei bestimmt, welche Funktionen in die Exporttabelle der DLL eingetragen werden – welche also von außen als Custom-Action aufgerufen werden können.

Die Def-Datei sieht dann wie folgt aus:

```
LIBRARY "SetupHelperDll"

EXPORTS

ShowInstallDir
```

Doch nun zum eigentlichen Code: Über die Funktion **WcaGetTargetPath** ermitteln wir das Verzeichnis, auf das die Directory-Variable INSTALLDIR zeigt:

```
LPWSTR pInstallDir = NULL;
WcaGetTargetPath(L"INSTALLDIR", &pInstallDir);
WcaLog(LOGMSG_STANDARD, "INSTALLDIR: %ls", pInstallDir);
```

WcaGetTargetPath liest den Pfad über die Windows Installer API-Funktion **MsiGetTargetPath** und reserviert, falls der übergebene Pointer den Wert *NULL* hat, automatisch einen genügend großen String zur Aufnahme des Pfades.

Die Funktion **WcaLog** schreibt den String dann in die Logdatei weg. Wir sehen, dass diese Funktion wie *sprintf* aufgerufen wird. Da die Strings Unicode-Strings sind, müssen wir statt *%s* den Ausdruck *%ls* angeben.

Das Property *ProductName* lesen wir über **MsiGetProperty** aus. Dieser soll später als Fenstertitel der MessageBox ausgegeben werden:

```
LPWSTR pProductName = NULL;
WcaGetProperty(L"ProductName", &pProductName);
WcaLog(LOGMSG_STANDARD, "ProductName: %ls", pProductName);
```

Nun können wir den String per MessageBox ausgeben und können den Speicher mit **ReleaseMem** wieder freigeben (ReleaseMem ist in der Datei *memutil.h* deklariert und muss deshalb mit #include eingebunden werden).

Die gesamte Funktion sieht dann so aus:

```
#include "stdafx.h"
#include "memutil.h"

UINT __stdcall ShowInstallDir(MSIHANDLE hInstall)
{
    HRESULT hr = S_OK;
    UINT er = ERROR_SUCCESS;
    LPWSTR pInstallDir = NULL;
    LPWSTR pProductName = NULL;

    hr = WcaInitialize(hInstall, "ShowInstallDir");
    ExitOnFailure(hr, "Failed to initialize");

    WcaLog(LOGMSG_STANDARD, "Initialized.");

    WcaGetTargetPath(L"INSTALLDIR", &pInstallDir);
    WcaLog(LOGMSG_STANDARD, "INSTALLDIR: %ls", pInstallDir);

    WcaGetProperty(L"ProductName", &pProductName);
    WcaLog(LOGMSG_STANDARD, "ProductName: %ls", pProductName);

    MessageBox(NULL, pInstallDir, pProductName, MB_OK);

    ReleaseMem(pInstallDir);
    ReleaseMem(pProductName);

LExit:
    er = SUCCEEDED(hr) ? ERROR_SUCCESS : ERROR_INSTALL_FAILURE;
    return WcaFinalize(er);
}
```

Über den Rückgabewert **ERROR_SUCCESS** (das ist der Wert 0) teilen wir dem Setup mit, dass die Funktion fehlerfrei durchlaufen wurde. Mit einem Rückgabewert **ERROR_INSTALL_FAILURE** melden wir dem Windows Installer einen Fehler zurück, der dann das Setup abbrechen würde.

16.6.3 Custom-Action einbinden

Nachdem wir unsere DLL erfolgreich erstellt haben, können wir die DLL in unser Setup einbinden. Die DLL selbst legen wir in der Binär-Tabelle ab, da sie nur zur Ausführungszeit des Setups benötigt wird. Am einfachsten verweisen wir in der Binär-Tabelle auf den Ausgabepfad vom Projekt **SetupHelperDII**. Dazu müssen wir zuerst eine Referenz auf dieses Projekt zu unserem WiX-Projekt hinzufügen:

Das machen wir, indem wir im WiX-Projekt mit der rechten Maustaste auf **References** klicken und den Menüpunkt „Verweis hinzufügen“ anklicken.

Nun können wir über die Projektvariable TargetPath auf die DLL verweisen und können die Custom-Action mit dem Element CustomAction definieren und in die Userface-Sequenz vor ExecuteAction einbinden.

```
<Binary Id="CppDll" SourceFile="$(var.SetupHelperDll.TargetPath)" />

<CustomAction Id="ShowInstallDir" BinaryKey="CppDll" DllEntry="ShowInstallDir" />

<InstallUISequence>
    <Custom Action="ShowInstallDir" Before="ExecuteAction" />
</InstallUISequence>
```

16.6.4 Die Wca-Library

Derzeit gibt es noch keine umfassende Hilfe, welche die Funktionen in dieser Bibliothek beschreibt. Da wir es aber mit einem Open-Source-Produkt zu tun haben, hilft uns ein kurzer Blick in den Quellcode, um uns einen umfassenden Eindruck über die Funktionen zu verschaffen. Wir finden den Quellcode im Verzeichnis `.\src\libs\wcautil` im WiX-Source-Code.

Hier eine kurze Auflistung, welche Funktionen in welcher Datei zu finden sind:

Dateiname	Beschreibung
qtxec.cpp	In dieser Datei befindet sich die Funktion QuietExec , die eine ausführbare Datei im Silent-Mode startet und die Ausgaben in das Logfile umleitet.
wcalog.cpp	Hier finden wir Funktionen, um Einträge ins Windows Installer Logfile zu schreiben.
wcascript.cpp	Für die Rollback-Funktionalität ist es manchmal notwendig, dass man sich irgendwo merkt, welche Aktionen rückgängig gemacht werden müssen. Dies kann man auf einfache Art und Weise in einer temporären Datei erledigen. Und genau solche Funktionen befinden sich in der Datei wcascript.cpp.
wcautil.cpp	Hier sind Funktionen zur Initialisierung (wie z.B. <code>WcaGlobalInitialize</code>) der Wca-Library zu finden.
wcawow64.cpp	In dieser Datei befinden sich Funktionen, über die man das File System Redirection in einem 64Bit-Betriebssystem ausschalten kann.
wcawrap.cpp	Hier finden wir die wohl am häufigsten benutzten Funktionen der Wca-Library. Es gibt z. B. Funktionen zum Lesen und Schreiben von Properties (<code>WcaGetProperty</code> bzw. <code>Wca SetProperty</code>), zum Lesen von Verzeichnisvariablen (<code>WcaGetTargetPath</code>) oder zum temporären Schreiben in die Windows Installer Tabellen (<code>WcaAddTempRecord</code>).
wcawrapquery.cpp	Manchmal muss man ganze Tabellen in eine temporäre Datei schreiben, um diese in einen anderen (System-)Kontext zu übertragen. Und genau das kann man mit den Funktionen in dieser Datei.

16.7 VBScript-Custom-Action

Neben dem Starten von EXEs können wir auch **VBScript** als Custom-Action einbinden.

Das Skript wird nicht vom **Windows Scripting Host** (WHS), sondern vom Windows Installer selbst abgearbeitet. Wird der Windows Scripting Host vom Virensucher geblockt oder ist er über Systemrichtlinien abgeschaltet, dann funktionieren unsere Custom-Actions in der Regel trotzdem.

Das Skript kann entweder in einer VBS-Datei oder direkt in der Custom-Action abgelegt werden. Das Ablegen des VBScripts in der Custom-Action ist nur für kurze Skripte sinnvoll, die keinen Rückgabewert an den Windows Installer zurückgeben sollen. Entscheiden wir uns für eine VBS-Datei, so kann diese entweder mit dem Produkt installiert oder in der Binärdatei abgelegt werden.

Der Windows Installer definiert beim Aufruf der VBScript-Funktion ein Objekt namens **Session**. Das Session-Objekt stellt quasi das Fenster zu unserem Setup dar. So kann z. B. über die Session-Eigenschaft **Property** eine Property ausgelesen werden.

16.7.1 VBScript direct in Custom-Action erstellen

Als Erstes wollen wir zeigen, wie das Skript direkt in der Custom-Action abgelegt wird. Wie bereits erwähnt, ist das nur für kurze Skripte sinnvoll, die keinen Rückgabewert an den Windows Installer zurückgeben sollen.

Die Custom-Action in dieser Form sieht folgendermaßen aus:

```
<CustomAction Id="ShowUserName" Script="VBScript" Return="check">
  <![CDATA[MsgBox "UserName: " & Session.Property("LogonUser")]]>
</CustomAction>
```

Das CDATA-Element sorgt dafür, dass die im Skript eingegebenen Zeichen (wie z. B. > oder <) als VBScript interpretiert werden und nicht als XML-Anweisungen. Obwohl das Skript keinen Rückgabewert zurückgibt, ist das Attribut *Return="check"* durchaus angebracht. Wenn ein Syntaxfehler vorhanden ist oder eine nicht behandelte Exception geworfen wird, bricht der Windows Installer das Setup mit einem Fehler ab. Setzt man hingegen *Return* auf „*ignore*“, dann wird das Skript an der entsprechenden Stelle abgebrochen und der Fehler wird ignoriert.

16.7.2 Custom-Action über VBS-Datei erstellen

Wenn das VBScript in einer VBS-Datei abgelegt wird, dann sieht das Skript folgendermaßen aus:

```
function MyVBScriptFunc
  MsgBox "Das ist eine VBScript Custom Action"
  MyVBScriptFunc = 0
end function
```

Über folgende Ziele wird es dann in das Setup eingebunden:

```
<Binary Id="MyVbScript" SourceFile="MyScript.vbs" />
<CustomAction Id="RunVbScript" BinaryKey="MyVbScript" VBScriptCall="MyVbScriptFunc"/>
```

► **Hinweis:** Das VBScript muss als ANSI-Datei abgelegt werden. Erstellt man die VBS-Datei über Visual Studio (z. B. als „Text File“), so wird die erstellte Datei mit UTF8 codiert und kann vom Skript-Compiler nicht verarbeitet werden.

16.7.3 Ausgabe in die Logdatei des Windows Installers

Wenn wir aus einem VBScript heraus einen Eintrag in die Logdatei machen wollen, dann können wir das über die Message-Methode machen:

```
sLogFileValue = "*** VBScript erzeugter Logdateieintrag"
Set oRecord = Session.Installer.CreateRecord(1)
oRecord.StringData(1) = sLogFileValue
Session.Message &H04000000, oRecord
```

Die CreateRecord-Methode vom **Installer-Objekt** erstellt ein Objekt vom Type Record. Der Record wird über die Methode StringData gefüllt und dann über die Methode Message in die Logdatei geschrieben.

Falls Konstrukte wie [INSTALLDIR] in das Logfile geschrieben werden sollen, kann der Ausabestring vorher mit folgender Funktion aufgelöst werden, so dass der Inhalt von INSTALLDIR ausgegeben wird:

```
function FormatMessage(strMessage)
    Dim oMsgRecord

    Set oMsgRecord = Session.Installer.CreateRecord(1)
    oMsgRecord.StringData(1) = strMessage
    FormatMessage = Session.FormatRecord(oMsgRecord)
end function
```

16.8 JavaScript-Custom-Action

Im Kapitel zuvor haben wir gesehen, dass wir VBScript als Custom-Action einbinden können. Arbeitet man lieber mit **JavaScript**, so ist das auch kein Problem – auch das wird vom Windows Installer unterstützt.

Das Skript wird nicht vom **Windows Scripting Host** (WHS), sondern vom Windows Installer selbst abgearbeitet. Wird der Windows Scripting Host vom Virenschanner geblockt oder ist er über Systemrichtlinien abgeschaltet, dann funktionieren unsere Custom-Actions in der Regel trotzdem.

Wie beim VBScript kann das Skript entweder in einer JS-Datei oder direkt in der Custom-Action abgelegt werden. Zur Kommunikation mit dem Setup steht auch das **Session-Objekt** zur Verfügung.

16.8.1 JavaScript in Custom-Action

Als Erstes wollen wir zeigen, wie das Skript direkt in der Custom-Action abgelegt wird. Das ist nur für kurze Skripte sinnvoll, die keinen Rückgabewert an den Windows Installer zurückgeben sollen.

Die Custom-Action sieht folgendermaßen aus:

```
<CustomAction Id="ShowLogonUser" Script="jscript" Return="check">
    <![CDATA[
        var msiMessageTypeUser      = 0x03000000;
        var msiMessageTypeOk       = 1;

        var options =     msiMessageTypeUser + msiMessageTypeOk;

        var objRecord = Session.Installer.CreateRecord(1);
        objRecord.StringData(0) = "Username: [1]";
        objRecord.StringData(1) = Session.Property("LogonUser");

        var response = Session.Message(options, objRecord);
    ]]>
</CustomAction>

<InstallUISequence>
    <Custom Action="ShowLogonUser" After="CostFinalize" />
</InstallUISequence>
```

Auch hier sorgt das CDATA-Element dafür, dass die im Skript eingegebenen XML-Zeichen als JavaScript interpretiert werden und nicht als XML-Anweisungen.

16.8.2 JavaScript in JS-Datei

Wenn das JavaScript in einer JS-Datei abgelegt wird, kann über den Rückgabewert gesteuert werden, ob das Setup abbricht oder nicht. In unserem Beispiel wollen wir eine MessageBox ausgeben, die den Anwender fragt, ob das Setup abgebrochen werden soll oder nicht. Je nach Wahl des Benutzers geben wir dann an den Windows Installer einen entsprechenden Rückgabewert zurück.

Hierzu legen wir zunächst eine JS-Datei im ANSI-Format an.

```
function AbortSetup( )
{
    // Return values
    var errorSuccess      = 0;
    var errorUserAbort    = 2367;

    // Constants for Session.Message
    var msiMessageTypeUser = 0x03000000;
    var msiMessageTypeYesNo = 4;

    // Return values from Session.Message
    var msiMessageStatusYes = 6;
    var msiMessageStatusNo  = 7;

    var options = msiMessageTypeUser + msiMessageTypeYesNo;

    var objRecord = Session.Installer.CreateRecord(1);
    objRecord.StringData(0) = "[1]";
    objRecord.StringData(1) = "Do you want to abort the setup?";

    var response = Session.Message(options, objRecord);

    // Check if we want to abort setup
    if(response == msiMessageStatusYes)
    {
        return errorUserAbort;
    }

    return errorSuccess;
};
```

Diese Datei wird dann wie folgt im WiX-Skript eingebunden:

```
<Binary Id="AbortSetup.js" SourceFile="AbortSetup.js"/>
<CustomAction Id="AbortSetup" JScriptCall="AbortSetup" BinaryKey="AbortSetup.js"
               Return="check"/>

<InstallUISequence>
    <Custom Action="AbortSetup" After="CostFinalize" />
</InstallUISequence>
```

16.8.3 Ausgabe in die Logdatei des Windows Installers

Wenn wir aus einem JavaScript heraus einen Eintrag in die Logdatei machen wollen, dann können wir das über die Message-Methode machen:

```
var msiMessageTypeInfo = 0x04000000;

// Create the record and fill it with data
var objRecord = Session.Installer.CreateRecord(1);
objRecord.StringData(1) = "This message goes to the log file";

// Write the record to the log file
Session.Message(msiMessageTypeInfo, objRecord)
```

16.9 PowerShell als Custom-Action aufrufen

Seit Windows 7 ist PowerShell fester Bestandteil des Betriebssystems und kann auf XP, Server 2003 und Vista per Windows-Update nachinstalliert werden. Über PowerShell können viele administrative Aufgaben über eine objektorientierte Skriptsprache abgehandelt werden.

Der Windows Installer unterstützt das Starten von PowerShell-Skripten nicht direkt. Es gibt also keine Custom-Action vom Typ „Start PowerShell“. Allerdings kann mit ein paar Kniffen die PowerShell.exe mit einer Custom-Action als EXE aufgerufen werden.

Zunächst prüfen wir, ob PowerShell überhaupt verfügbar ist und in welches Verzeichnis es installiert wurde:

```
<!--Get path to powershell.exe -->
<Property Id="POWERSHELLEXE" Secure="yes">
    <RegistrySearch Id="POWERSHELL" Root="HKLM"
        Key="SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"
        Name="Path" Type="raw"/>
</Property>
```

Ist PowerShell nicht installiert, könnten wir PowerShell im Setup installieren oder das Setup über eine Installationsbedingung abbrechen:

```
<Condition Message="This application requires Windows PowerShell.">
    <![CDATA[Installed OR POWERSHELL]]>
</Condition>
```

Wir haben nun den Pfad zur PowerShell.exe. Damit ist es ein Leichtes, PowerShell aufzurufen. Im ersten Beispiel wollen wir den „Background Intelligent Transfer Service BITS neu starten. Das machen wir, indem wir das Restart-Service-Kommando direkt an PowerShell übergeben:

```
<CustomAction Id="RestartService" Property="POWERSHELL" Execute="deferred"
    Impersonate="no" ExeCommand='"[POWERSHELL]" Restart-Service BITS' />
<InstallExecuteSequence>
    <Custom Action="RestartService" After="InstallFiles">Not Installed</Custom>
</InstallExecuteSequence>
```

16.9.1 PowerShell-Skript ausführen

Etwas aufwendiger sieht es aus, wenn ein PowerShell-Skript gestartet werden soll. Das Starten von PowerShell-Skripten ist nämlich in Windows per Gruppenrichtlinie abgeschaltet und muss zuerst über PowerShell mit folgendem Aufruf erlaubt werden:

PowerShell.exe Set-ExecutionPolicy Unrestricted

Da das Erlauben von PowerShell-Skripten ein Eingriff in die Sicherheitsrichtlinien darstellt, sollten wir die Gruppenrichtlinie nach dem Starten unserer Skripte wieder auf den Anfangszustand zurücksetzen. Da diese Einstellung in der Registry zu finden ist, ermitteln wir zunächst, wie der aktuelle Zustand ist:

```
<Property Id="PS_EXECUTION_POLICY" Value="Restricted" Secure="yes">
    <RegistrySearch Id="PsExecutionPolicy" Root="HKLM"
        Key="SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"
        Name="ExecutionPolicy" Type="raw"/>
</Property>
```

Da unser Setup ein 32-Bit-Setup ist, prüfen wir auch nur den 32-Bit-Bereich in der Registry. Für ein 64-Bit-Setup müssten wir noch das Attribut Win64 auf yes setzen. Da u. U. noch kein Wert in der Registry zu finden ist (in diesem Fall ist die Policy auf restricted gesetzt), geben wir als Default-Wert *Restricted* an.

Eventuell wollen wir Parameter aus dem Setup in das PowerShell-Skript einfügen. Das lässt sich sehr einfach mit Übergabeargumenten an das Skript realisieren. Wollen wir z. B. die Property USERNAME und PASSWORD an das Skript übergeben, dann rufen wir PowerShell wie folgt auf:

```
PowerShell.exe –File "c:\Skript.ps1" –Username "[USERNAME]" –Password "[PASSWORD]"
```

Im Skript definieren wir in der ersten Zeile entsprechende Variablen, die über die Param-Funktion angegeben werden. Diese werden dann in unserem Beispielskript einfach per MessageBox ausgegeben:

```
# Define parameter variables
param(
    [string]$Username,
    [string]$Password)

# Load forms assembly and show MessageBox
[System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
[System.Windows.Forms.MessageBox]::Show("Name=" + $Username +
    " Password=" + $Password,"PowerShell CA",0)
```

► Hinweis: Als Variabletype ist nicht nur [string] möglich, sondern u. a. [byte], [int], [long], [bool], [decimal] und [DateTime]. Eine vollständige Liste findet man in der Dokumentation von PowerShell.

Nun haben wir alles, um das PowerShell-Skript in unser Setup einzubauen. Wir fügen das Skript in unser Setup als Datei ein, installieren dieses und rufen es dann als Custom-Action auf. In WiX sieht das dann in etwa so aus:

```
<!-- Component with script file -->
<Component Id="PsScript" Guid="YOURGUID" Directory="INSTALLFOLDER">
    <File Id="PowerShell.ps1" Source=".\\PowerShell.ps1" KeyPath="yes"/>
</Component>
<!-- Get path to powershell.exe -->
<Property Id="POWERSHELL">
    <RegistrySearch Id="POWERSHELL" Root="HKLM"
        Key="SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"
        Name="Path" Type="raw"/>
</Property>

<!--No powershell - no show -->
<Condition Message="This application requires Windows PowerShell.">
    <![CDATA[Installed OR POWERSHELL]]>
</Condition>

<!-- Get execution policy from powershell -->
<Property Id="PS_EXECUTION_POLICY" Value="Restricted" Secure="yes">
    <RegistrySearch Id="PsExecutionPolicy" Root="HKLM"
        Key="SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"
        Name="ExecutionPolicy" Type="raw" Win64="yes"/>
</Property>
```

```
<!-- Custom actions -->
<CustomAction Id="EnablePsExecutionPolicy" Property="POWERSHELL"
    ExeCommand='"[POWERSHELL]" Set-ExecutionPolicy Unrestricted'
    Execute="deferred" Impersonate="no"/>
<CustomAction Id="RestorePsExecutionPolicy" Property="POWERSHELL"
    ExeCommand='"[POWERSHELL]" Set-ExecutionPolicy [PS_EXECUTION_POLICY]'
    Execute="deferred" Impersonate="no"/>
<CustomAction Id="StartPowerShellScript" Directory="INSTALLFOLDER"
ExeCommand='"[POWERSHELL]" -File PowerShell.ps1 -Username "[USERNAME]"
>Password "[PASSWORD]"' Execute="deferred" Impersonate="no"/>

<!-- Schedule custom actions -->
<InstallExecuteSequence>
    <Custom Action="EnablePsExecutionPolicy" After="InstallFiles">
<![CDATA[PS_EXECUTION_POLICY~<>"Unrestricted" AND $PsScript=3]]></Custom>
        <Custom Action="StartPowerShellScript" After="EnablePsExecutionPolicy">
$PsScript=3
</Custom>
        <Custom Action="RestorePsExecutionPolicy" After="StartPowerShellScript">
<![CDATA[PS_EXECUTION_POLICY~<>"Unrestricted" AND $ PsScript =3]]>
</Custom>
    </InstallExecuteSequence>
```

EnablePsExecutionPolicy und RestorePsExecutionPolicy werden nur aufgerufen, wenn die Gruppenrichtlinie keine PowerShell-Skript-Aufrufe erlaubt. Zwischen diesen Befehlen starten wir das eigentliche PowerShell-Skript. Alle drei Funktionen werden über die Action-Flag der PsScript-Komponente gesteuert (\$PsScript=3), da das Skript natürlich nur dann gestartet werden kann, wenn die Datei auch tatsächlich auf das Zielsystem kopiert wurde.

17 Custom-Action debuggen

Schreiben wir größere Custom-Actions, dann ist es natürlich sehr hilfreich, wenn man durch den Code mit dem Debugger steppen kann. In diesem Kapitel werden wir dieses Thema einmal näher beleuchten.

17.1 C#-Action debuggen

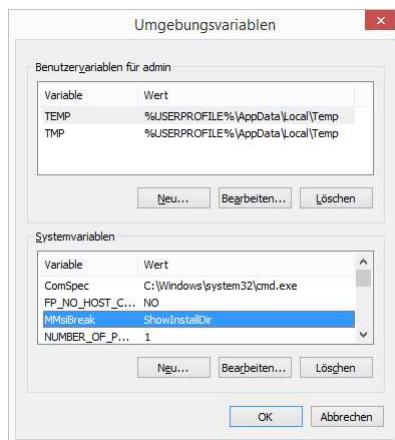
Es gibt drei Möglichkeiten, eine Custom-Action mit Managed Code zu debuggen:

- Über die Umgebungsvariable MMsiBreak
- Über eine MessageBox und das Anhängen des Debuggers an den Prozess
- Über das Assembly System.Diagnostics.Debugger

Alle Methoden wollen wir hier vorstellen.

17.1.1 Debugger über Umgebungsvariable aufrufen

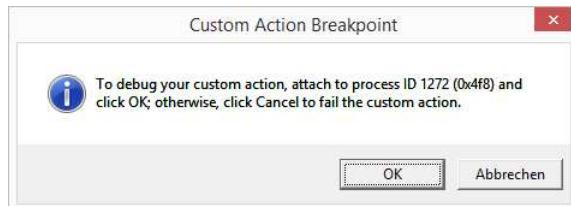
Um die Managed-Code-Custom-Action über eine Umgebungsvariable **MMsiBreak** zu debuggen, müssen wir die Umgebungsvariable als Systemvariable einrichten. Das machen wir über den Button *Umgebungsvariablen* unter Systemsteuerung▶System▶Erweiterte Einstellungen:



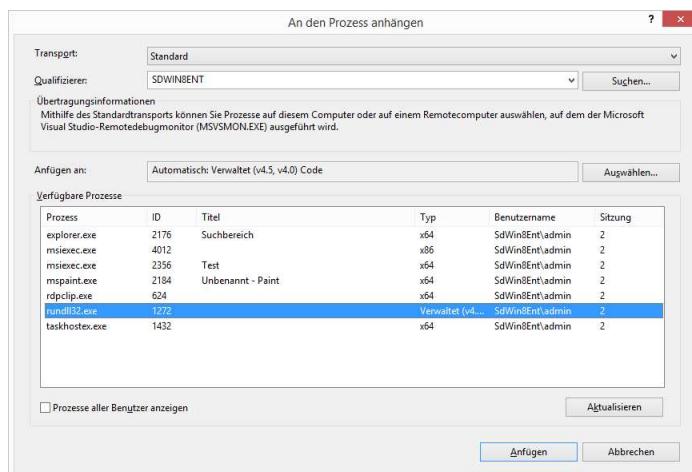
Als Wert der Umgebungsvariable MMsiBreak geben wir den **Namen der C#-Funktion** an. Der Custom-Action-Name ist in diesem Fall nicht relevant.

► Hinweis: Es reicht nicht, dass die Umgebungsvariable in der CMD-Box per SET-Kommando eingerichtet wird.

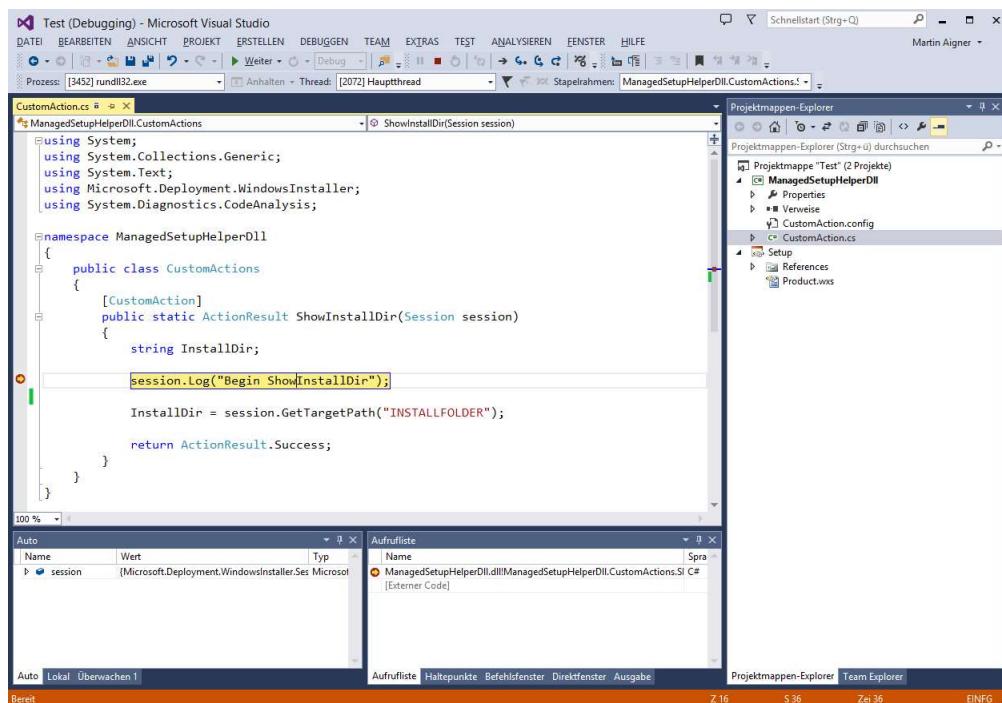
Nun starten wir das Setup (z. B. per Doppelklick). Sobald das Setup die Custom-Action aufruft, kommt folgende Meldung vom DTF-Framework:



Im Visual Studio können wir dann uns mit geladenem Projekt an den Prozess rundll32.exe anhängen. Hierzu rufen wir den Menüpunkt **Debuggen ► An den Prozess anhängen** auf. Auf dem daraufhin erscheinenden Dialog wählen wir den Prozess mit der angegebenen ID aus:



Bevor wir die MessageBox mit OK bestätigen, setzen wir im Code an der entsprechenden Stelle einen Breakpoint und können dann das Programm debuggen:



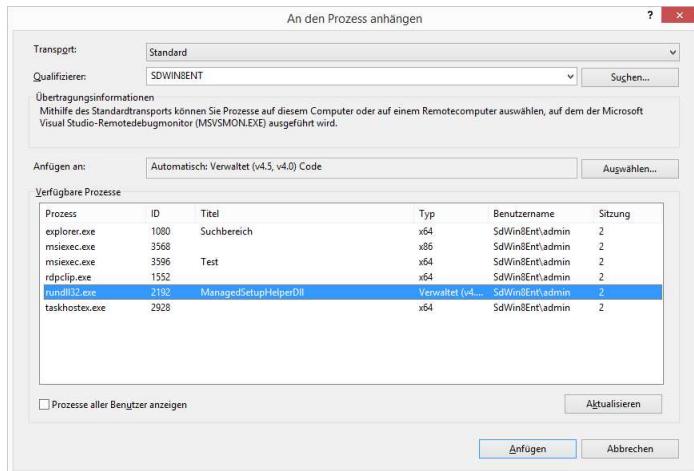
17.1.2 Debugger über MessageBox an Prozess anhängen

Auch wenn es sich etwas antiquiert anhört: Die Möglichkeit, den Debugger über eine MessageBox aufzurufen, ist manchmal sehr hilfreich und schneller als die Methode über die Umgebungsvariable. Der einzige Nachteil ist, dass die Custom-Action-DLL vor der finalen Auslieferung neu kompiliert werden muss, da wir die MessageBox ja nicht an unsere Kunden ausliefern wollen.

Doch hier die Schritt für Schritt Anleitung: Zuerst erstellen wir im Code an der interessanten Stelle eine MessageBox:

```
MessageBox.Show("Debug here", "ManagedSetupHelperDll");
```

Wird die MessageBox nach dem Starten des Setups angezeigt, setzen wir im Visual Studio einen Breakpoint (natürlich nach der MessageBox) und hängen uns an den Prozess rundll32.exe an. Da wir bei der MessageBox die Caption mit angegeben haben, können wir den entsprechenden Prozess leicht identifizieren – die Caption wird in der Prozessliste in der Spalte Titel ausgegeben:



17.1.3 Debugger über System.Diagnostics.Debugger aufrufen

Alternativ zu den anderen Methoden können wir im Code auch direkt den Debugger aufrufen. Das machen wir über das **Debugger**-Assembly im Namespace System.Diagnostics.

```
using System.Diagnostics;

[CustomAction]
public static ActionResult ShowInstallDir(Session session)
{
    // To Debug the custom action!
    Debugger.Launch();
    Debugger.Break();

    ...
}
```

Sobald die Methode ShowInstallDir aufgerufen wird, wird eine Exception geworfen und der Debugger aufgerufen.

17.2 VB.NET-Action debuggen

Es gibt drei Möglichkeiten, die VB.NET-Custom-Action zu debuggen:

- Über die Umgebungsvariable MMsiBreak
- Über eine MessageBox und das Anhängen des Debuggers an den Prozess
- Über das Assembly System.Diagnostics.Debugger

Alle Methoden wurden bereits im Kapitel mit C# erklärt, worauf wir hier verweisen möchten.

17.3 C++-Action debuggen

Es gibt zwei Möglichkeiten, eine C++-Custom-Action zu debuggen:

- Über die Umgebungsvariable **MsiBreak**
- Über eine MessageBox und das Anhängen des Debuggers an den Prozess

Alle Methoden wollen wir hier vorstellen.

17.3.1 Debugger über Umgebungsvariable aufrufen

Über die Umgebungsvariable **MsiBreak** kann der Debugger für eine C++-DLL aufgerufen werden. Hierzu kompiliert man die DLL mit Debug-Informationen und bindet diese DLL in das Setup mit ein. Danach startet man eine Cmd-Box als Administrator.

► **Hinweis:** Es wichtig, dass die Cmd-Box als **Administrator** ausgeführt wird.

Über die Kommandozeile setzen wir die Umgebungsvariable **MsiBreak** auf den Namen der Custom-Action, die wir debuggen wollen:

```
set MsiBreak=ShowInstallDir
```

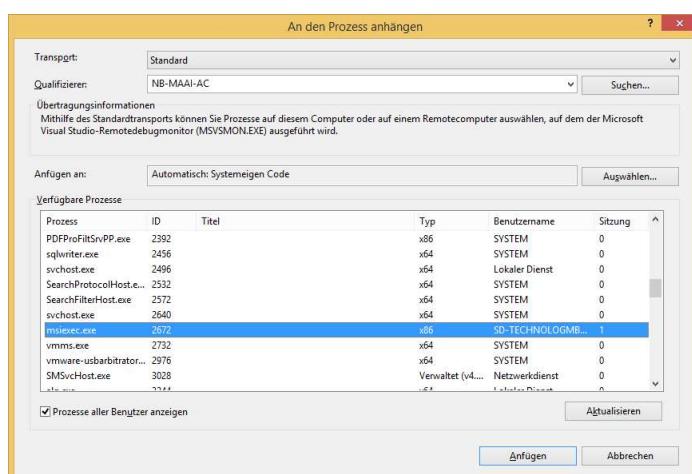
Anschließend starten wir das Setup über die Kommandozeile:

```
msiexec.exe /i <MySetup.msi>
```

Sobald das Setup die Custom-Action aufruft, kommt eine Meldung vom Windows Installer:



Im Visual Studio können wir uns dann mit geladenem Projekt an den msiexec.exe-Prozess anhängen. Hierzu rufen wir den Menüpunkt **Debuggen▶An den Prozess anhängen** auf. Auf dem daraufhin erscheinenden Dialog wählen wir den Prozess mit der angegebenen ID aus:



Bevor wir die MessageBox mit OK bestätigen, setzen wir im Code an der entsprechenden Stelle einen Breakpoint und können dann das Programm debuggen.

17.3.2 Debugger über MessageBox an Prozess anhängen

Auch wenn es sich etwas antiquiert anhört: Die Möglichkeit, den Debugger über eine MessageBox aufzurufen, ist manchmal sehr hilfreich und schneller als die Methode über die Umgebungsvariable. Der einzige Nachteil ist, dass die Custom-Action-DLL vor der finalen Auslieferung neu kompiliert werden muss, da wir die MessageBox ja nicht an unsere Kunden ausliefern wollen.

Doch hier die Schritt für Schritt Anleitung: Zuerst erstellen wir im Code an der interessanten Stelle eine MessageBox:

```
MessageBox(NULL, L"DebugMe", L>ShowInstallDir", MB_OK);
```

Wird die MessageBox nach dem Starten des Setups angezeigt, setzen wir im Visual Studio einen Breakpoint (natürlich nach der MessageBox) und hängen uns an den Msiexec.exe Prozess an. Da wir bei der MessageBox die Caption `ShowInstallDir` angegeben haben, können wir den entsprechenden Prozess leicht identifizieren – die Caption wird in der Prozessliste in der Spalte Titel ausgegeben.

17.4 VBScript-Action debuggen

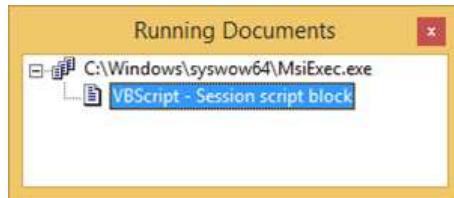
Arbeitet man lieber mit VBScript, dann muss man auch nicht auf das Debuggen verzichten. Wir benötigen hierfür den **Script-Debugger** von Microsoft, den man von der Microsoft-Webseite unter der URL: <http://www.microsoft.com/en-us/download/details.aspx?id=22185> herunterladen kann.

Nun müssen wir noch den Just-in-time-Debugger aktivieren, in dem wir folgenden Registry-Wert auf den Wert 1 setzen:

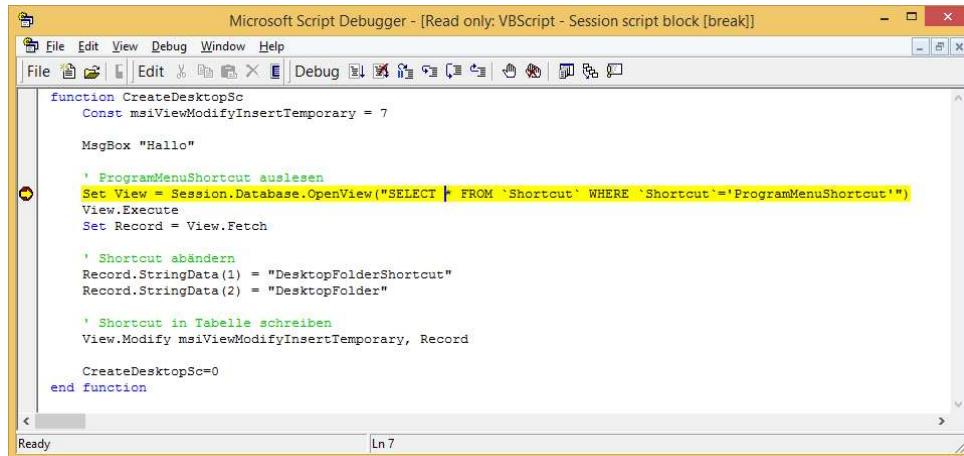
```
HKEY_CURRENT_USER\Software\Microsoft\Windows Script\Settings\JITDebug
```

Im VBScript tragen wir nun an der interessanten Stelle eine MessageBox über die MsgBox-Funktion ein. Wenn das Setup läuft und die MessageBox dargestellt wird, starten wir den Script-Debugger msscrdbg.exe (dieser liegt normalerweise im Verzeichnis c:\Program Files (x86)\Microsoft Script Debugger\ und gehen dort auf den Menüpunkt **View▶Running Documents**.

In der Liste der „Running Documents“ finden wir den MsiExec-Prozess samt dem *Session script block*:



Nun können wir im Script-Debugger einen Breakpoint setzen und das Skript debuggen:



Möchte man die Werte der einzelnen Variablen sehen, dann öffnet man das Command-Window über **View > Command Window** und gibt in der Kommandozeile die zu ermittelnde Variable ein und drückt Return. Im Command-Window wird dann der Wert der Variable ausgegeben:



17.5 JavaScript-Action debuggen

Das Debuggen von JavaScript funktioniert wie beim VBScript, nur dass die MessageBox hier nicht über MsgBox, sondern über Session.Message ausgegeben werden muss:

```
var msiMessageTypeUser = 0x03000000;  
var msiMessageTypeOk = 1;  
  
// MessageBox ausgeben  
var options = msiMessageTypeUser + msiMessageTypeOk;  
var objRecord = Session.Installer.CreateRecord(1);  
objRecord.StringData(0) = "Debug Me";  
Session.Message(options, objRecord);
```

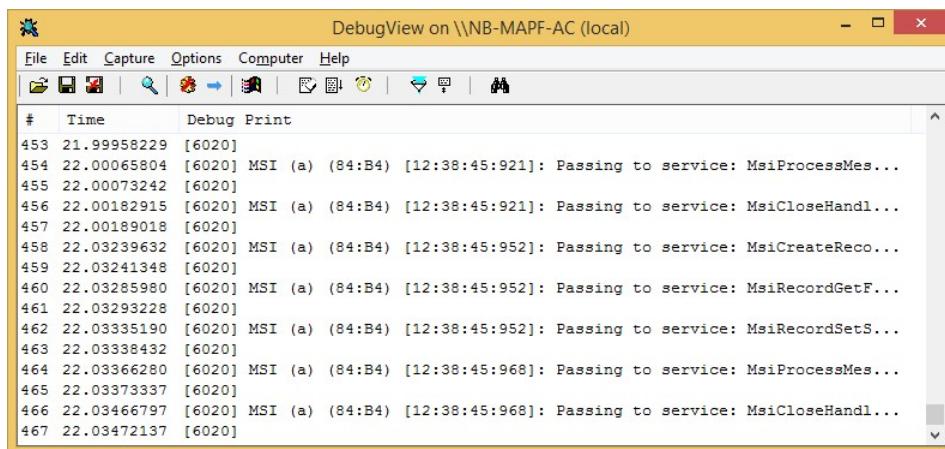
17.6 Debug-Informationen mit DebugView

Der Windows Installer gibt neben den Einträgen in der Logdatei auch noch Debug-Informationen mittels **OutputDebugString** (Win32) bzw. **DbgPrint** (Kernel-Mode) aus. Diese können mit dem Microsoft-Tool DebugView ausgeben werden. DebugView kann man von der Microsoft-Webseite unter der URL <https://technet.microsoft.com/en-us/library/bb896647.aspx> herunterladen.

DebugView ist vor allem für das Testen von Patches interessant, da die Patch-Validierung (welche Transformation im Patch wird verwendet und auf welches MSI wird der Patch angewandt?) detailliert über diesen Mechanismus ausgegeben wird.

Bevor man DebugView verwendet, muss man noch einen Schlüssel in der Registry anlegen. Unter **HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\Installer** wird der Schlüssel Debug vom Typ DWORD erzeugt und der Wert auf 2 gesetzt. Ist der Key noch nicht vorhanden, so muss er neu angelegt werden.

Nun kann DebugView und anschließend das Setup mit MsiSetup.exe gestartet werden. DebugView gibt die Debug-Ausgaben in folgender Form aus:



Möchte man selbst Ausgaben in DebugView machen, kann man folgende Befehle verwenden:

Sprache	Quellcode
C++	<code>OutputDebugString(L"Debug Ausgabe");</code>
C#	<code>System.Diagnostics.Debug.WriteLine("Debug Ausgabe");</code>
VB.NET	<code>System.Diagnostics.Debug.WriteLine("Debug Ausgabe")</code>

18 InstallExecute-Sequenz näher betrachtet

In diesem Abschnitt werden wir die InstallExecute-Sequenz etwas näher betrachten. Bisher sind wir davon ausgegangen, dass alle Aktionen anhand der Sequenznummer sortiert und in dieser Reihenfolge abgearbeitet werden. Trägt man aber folgende Zeilen in das Setup ein, so wirft uns das Ergebnis dieses „Weltbild“ durcheinander:

```
<CustomAction Id='Action1' Script='VB-Skript' Return='ignore' Execute='commit'>
  <![CDATA[MsgBox "Action1"]]>
</CustomAction>
<CustomAction Id='Action2' Script='VB-Skript' Return='ignore' Execute='deferred'>
  <![CDATA[MsgBox "Action2"]]>
</CustomAction>
<CustomAction Id='Action3' Script='VB-Skript' Return='ignore' Execute='immediate'>
  <![CDATA[MsgBox "Action3"]]>
</CustomAction>
<CustomAction Id='Action4' Script='VB-Skript' Return='ignore' Execute='rollback'>
  <![CDATA[MsgBox "Action4"]]>
</CustomAction>

<InstallExecuteSequence>
  <Custom Action='Action1' After='InstallInitialize' />
  <Custom Action='Action2' After='Action1' />
  <Custom Action='Action3' After='Action2' />
  <Custom Action='Action4' After='Action3' />
</InstallExecuteSequence>
```

Normalerweise würden wir davon ausgehen, dass zuerst die Nachricht „Action1“, dann „Action2“ usw. erscheint. Wenn wir das Setup aber ausführen, sehen wir, dass zuerst „Action3“, dann „Action2“ und dann „Action1“ auftritt. Um dies zu verstehen, müssen wir uns erst einmal ansehen, wie der Windows Installer die InstallExecute-Sequenz abarbeitet.

18.1 Installationsskripte

In der InstallExecute-Sequenz finden wir in jedem Windows Installer Paket zwei Standardaktionen: **InstallInitialize** und **InstallFinalize**. Die Standardaktion **InstallInitialize** erstellt drei Installationsskripte. Ein Deferred-Skript, ein Commit-Skript und ein Rollback-Skript. Alle Aktionen deren Execute-Attribut auf *immediate* gesetzt ist, werden sofort ausgeführt. Aktionen, deren Execute-Attribut auf *deferred*, *commit* oder *rollback* gesetzt ist, werden nicht sofort ausgeführt, sondern in das entsprechende Skript (*defered*, *commit* oder *rollback*) geschrieben.

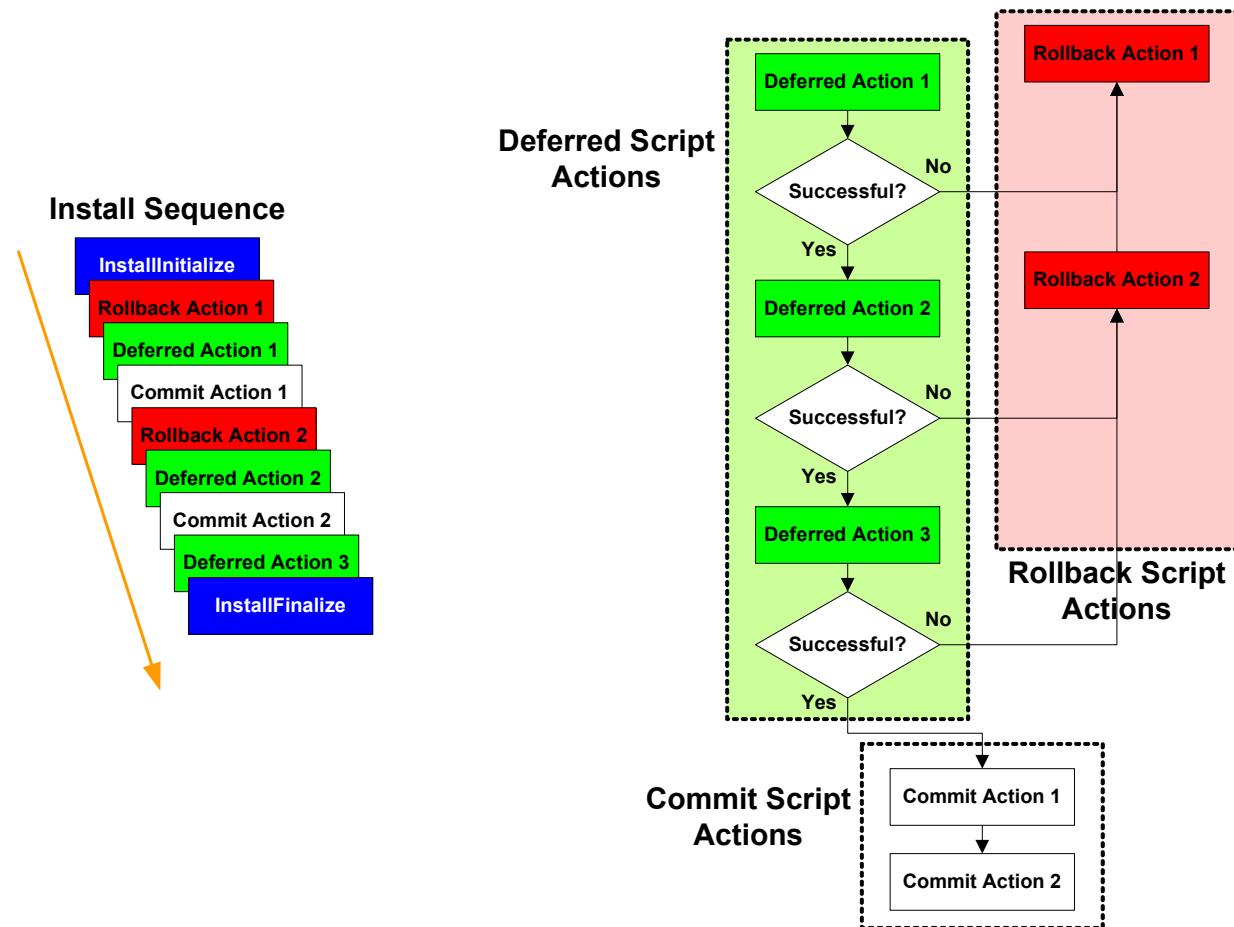
Erst die Standardaktion **InstallFinalize** führt dann die einzelnen Skripte aus. Zuerst wird mit dem Deferred-Skript begonnen. War die Abarbeitung erfolgreich, so wird das Commit-Skript abgearbeitet. Gibt eine Aktion im Deferred-Skript einen Fehler zurück, wird das Rollback-Skript ausgeführt. Aus dem Rollback-Skript werden nur diejenigen Aktionen ausgeführt, die oberhalb der abbruchverursachenden Deferred-Aktion standen – die Ausführungsreihenfolge im Rollback-Skript ist hierbei von unten nach oben.

Da die Skripte über zwei eigenständige MsiExec-Prozesse (ein Prozess läuft mit Benutzerrechten, der andere mit Systemrechten) abgearbeitet werden, haben die Aktionen keinen gültigen Session-Handle. Deshalb kann eine Deferred-, Commit- oder Rollback-Aktion auf keine Tabellen und nur auf eine Handvoll ausgesuchter Properties zugreifen.

Die folgende Tabelle zeigt, welche Properties in den Skripten zur Verfügung stehen:

Property	Beschreibung
CustomActionData	Enthält den für die Funktion übergebenen String
ProductCode	Produkt-Code (GUID)
UserSID	Die Security-ID des Benutzers, der die Installation ausführt

Das Schaubild macht diesen Prozess noch einmal grafisch deutlich:



Aber: Wofür eigentlich so viel Aufwand und wofür vier Installationsmodi (immediate, deferred, commit, rollback)? Die Antwort ist relativ einfach: Das Stichwort ist Rollback. Wenn wir die Standardaktion `InstallFiles` einmal näher betrachten, dann wird das Ganze etwas klarer.

Die Standardaktion `InstallFiles` besteht eigentlich aus vier Aktionen: `InstallFiles(immediate)`, `InstallFiles(deferred)`, `InstallFiles(commit)` und `InstallFiles(rollback)`. Die Aktion, die in der Sequenz eingetragen ist, ist die Aktion `InstallFiles(immediate)`. Da `InstallFiles(deferred)`, `InstallFiles(commit)` und `InstallFiles(rollback)` keine Tabelle lesen können, sind diese auf `InstallFiles(immediate)` angewiesen.

`InstallFiles(immediate)` liest die File-Tabelle aus, ermittelt, welche Dateien installiert werden müssen, übergibt diese Informationen an `InstallFiles(deferred)`, `InstallFiles(commit)` und `InstallFiles(rollback)` und startet diese Aktionen über die Windows Installer API-Funktion **MsiDoAction**. Wird eine Aktion über `MsiDoAction` gestartet, so schaut sich der Windows Installer auch das Execute-Attribut an und verteilt die Aktionen in die entsprechenden Skripte.

Wenn das Deferred-Skript ausgeführt wird, ruft der Windows Installer an der entsprechenden Stelle die Funktion `InstallFiles(deferred)` auf. `InstallFiles(deferred)` liest die von `InstallFiles(immediate)` übergebenen Informationen und weiß, welche Dateien installiert werden müssen. `InstallFiles(deferred)` prüft für jede Datei daraufhin, ob sich im Zielordner bereits eine gleichnamige Datei befindet. Ist dies der Fall, so wird diese Datei nicht einfach überschrieben, sondern für den Rollback-Fall weggesichert.

Wenn das Deferred-Skript ohne Fehler abgearbeitet wurde, wird das Commit-Skript ausgeführt. An entsprechender Stelle wird dann `InstallFiles(commit)` aufgerufen. Das Commit-Skript ist für das Löschen der in `InstallFiles(deferred)` angelegten Kopien zuständig.

Wenn eine Funktion im Deferred-Skript einen Fehler meldet, wird das Rollback-Skript ausgeführt. An entsprechender Stelle wird dann `InstallFiles(rollback)` aufgerufen. `InstallFiles(rollback)` löscht alle installierten Dateien und kopiert die angelegten Kopien wieder zurück. Damit sind alle Änderungen, die `InstallFiles(deferred)` am System durchgeführt hat, rückgängig gemacht.

19 Parameterübergabe an InScript-Custom-Actions

Custom-Actions laufen *InScript*, wenn diese den Installations-Mode *deferred*, *commit* oder *rollback* haben. Wie bereits weiter oben erwähnt, werden diese Aktionen erst mit *InstallFinalize* über eigenständige MsiExec-Prozesse abgearbeitet. Der MsiExe-Prozess, mit dem diese **InScript-Custom-Actions** abgearbeitet werden, hat kein gültiges Session-Handle. Deshalb können InScript-Custom-Actions nicht auf Tabellen und nur auf eine sehr kleine Auswahl an Properties zugreifen.

Doch ein kleines Fensterchen haben uns die Softwaredesigner von Microsoft zur Verfügung gestellt. Dieses kleine Fenster ist ein Property namens **CustomActionData**. Über CustomActionData haben wir die Möglichkeit, jeder InScript-Custom-Action genau einen Parameter-String zu übergeben. Sollen mehrere Parameter übergeben werden, so müssen diese durch ein Trennzeichen getrennt in einen String geschrieben werden. Dieser String muss dann von der InScript-Custom-Action ausgelesen und in die einzelnen Bestandteile auseinandergeparst werden.

Die Parameterübergabe erfolgt in vier Schritten:

1. Eine Custom-Action im Installations-Mode *immediate* ermittelt die erforderlichen Parameter (z. B. aus einer MSI-Tabelle bzw. aus Properties) und schreibt diese durch Trennzeichen getrennt in einen String.
2. Die *Immediate*-Custom-Action hinterlegt nun den String für die InScript-Action, indem sie ein Property setzt, das genauso heißt wie die InScript-Action. Der String wird hierbei in alle Installationsskripte geschrieben
3. Als nächster Schritt wird die InScript-Action aufgerufen. Da die InScript-Action den Installations-Mode *deferred*, *commit* oder *rollback* hat, wird diese nicht sofort ausgeführt, sondern zuerst in das entsprechende Skript geschrieben
4. Wird *InstallFinalize* ausgeführt, dann wird an entsprechender Stelle die InScript-Action ausgeführt. Die Action liest nun das Property *CustomActionData* aus und bekommt somit den für sie hinterlegten String wieder zurück.

19.1 Parameterübergabe in C#

19.1.1 Immediate-Action

Bei der C#-Custom-Action bekommen wir einige Hilfe von den **Development-Tools-Foundation** (DTF)-Klassen. Hier ist vor allem das **CustomActionData-Objekt** interessant. Über dieses Objekt können mehrere Übergabeparameter als Name / Value Paar an die Deferred-Action übergeben werden. Das machen wir, in dem wir für alle Übergabeparameterpaare die Funktion *Add* aufrufen:

```
[CustomAction]
public static ActionResult MyImmediateAction(Session session)
{
    session.Log("Begin MyImmediateAction");

    // Variablen auslesen
    String logonUser = session["LogonUser"];
    String systemFolder = session["SystemFolder"];

    // In Parameterstring schreiben
    session.CustomActionData.Add("LogonUser", logonUser);
    session.CustomActionData.Add("SystemFolder", systemFolder);

    // Action MyDeferredAction aufrufen
    session.DoAction("MyDeferredAction", session.CustomActionData);

    return ActionResult.Success;
}
```

Die Funktion *DoAction* fügt alle Parameterpaare zu einem String zusammen, schreibt diesen String in das Property *MyDeferredAction* und ruft dann mit *MsiDoAction* die Action *MyDeferredAction* auf.

Wenn wir von unserem MSI ein Logfile erstellen, sehen wir am Ende des Logfiles, dass ein Property namens MyDeferredAction erstellt wurde. Dessen Inhalt sieht dann etwa so aus:

Property(S): MyDeferredAction = LogonUser=martin.aigner;SystemFolder=C:\Windows\SysWOW64

In der Immediate-Action wird in der Regel nicht nur die Deferred-Action, sondern auch die Commit- und Rollback-Action mit DoAction aufgerufen. Im obigen Beispiel verwenden wir das CustomActionData-Objekt, das als Member-Variable vom Session-Objekt implementiert ist. Wenn aber jede Aktion unterschiedliche Parameter bekommen soll, kann man auch für jede Aktion ein eigenes CustomActionData-Objekt erstellen:

```
CustomActionData customActionDataDeferred = new CustomActionData();
CustomActionData customActionDataCommit = new CustomActionData();
CustomActionData customActionDataRollback = new CustomActionData();

// Werte für deferred Action übergeben
customActionDataDeferred.Add(...);

...
// Werte für commit Action übergeben
customActionDataCommit.Add(...);

...
// Werte für rollback Action übergeben
customActionDataRollback.Add(...);

...
// Alle Actions aufrufen
session.DoAction("MyRollbackAction", customActionDataRollback);
session.DoAction("MyCommitAction", customActionDataCommit);
session.DoAction("MyDeferredAction", customActionDataDeferred);
```

19.1.2 InScript-Action

In der InScript-Action lesen wir die Parameter über das CustomActionData-Objekt, welches eine Member-Variable des Session-Objekts ist, zurück. Für das CustomActionData-Objekt gibt es eine Überladung vom Operator []. Geben wir bei dieser Überladung den Parameternamen an, so bekommen wir den Parameterwert zurück:

```
[CustomAction]
public static ActionResult MyDeferredAction(Session session)
{
    session.Log("Begin MyDeferredAction");

    // Variablen auslesen
    String logonUser = session.CustomActionData["LogonUser"];
    String systemFolder = session.CustomActionData["SystemFolder"];
    ...

    return ActionResult.Success;
}
```

19.1.3 WiX-Skript

Im WiX-Skript binden wir dann beide Actions wie folgt ein:

```
<CustomAction Id="MyImmediateAction" BinaryKey="CsCa" DllEntry="MyImmediateAction"
              Execute="immediate" />
<CustomAction Id="MyDeferredAction" BinaryKey="CsCa" DllEntry="MyDeferredAction"
              Execute="deferred" Impersonate="no"/>

<InstallExecuteSequence>
    <Custom Action="MyImmediateAction" After="InstallInitialize" />
</InstallExecuteSequence>
```

Da die InScript-Action von der Immediate-Action aufgerufen wird, muss in die InstallExecuteSequenz nur die Immediate-Action eingetragen werden. Da diese eine InScript-Action (in diesem Fall eine Deferred-Action) aufruft, muss diese zwischen InstallInitialize und InstallFinalize stehen.

19.2 Parameterübergabe in VB.NET

19.2.1 Immediate-Action

Es wird nicht sehr verwundern, dass der VB.NET Code dem C#-Code sehr ähnelt – verwenden wir hier doch dieselben Klassen der Development Tools Foundation:

```
<CustomAction()> _
Public Shared Function MyImmediateAction(ByVal session As Session) As ActionResult
    Dim logonUser As String
    Dim systemFolder As String
    Dim customActionData As CustomActionData = New CustomActionData()

    ' Get values
    logonUser = session("LogonUser")
    systemFolder = session("SystemFolder")

    ' Add name/value pair to CustomActionData object
    customActionData.Add("LogonUser", logonUser)
    customActionData.Add("SystemFolder", systemFolder)

    ' Call deferred custom action
    session.DoAction("MyDeferredAction", customActionData)

    Return ActionResult.Success
End Function
```

Wir sehen hier, dass für die Parameterübergabe ein neues CustomActionData-Objekt erstellt wird. Hier könnte natürlich auch die Member-Variable vom Session-Objekt verwendet werden (siehe C#-Code).

19.2.2 InScript-Action

Die Action *MyDeferredAction* liest die Parameter über das die Member-Variable CustomActionData vom Session-Objekt wieder zurück:

```
<CustomAction()> _
Public Shared Function MyDeferredAction(ByVal session As Session) As ActionResult
    Dim logonUser As String
    Dim systemFolder As String

    ' Get values
    logonUser = session.CustomActionData("LogonUser")
    systemFolder = session.CustomActionData("SystemFolder")
    ...

    Return ActionResult.Success
End Function
```

19.2.3 WiX-Skript

Im Wix-Skript binden wir dann beide Actions wieder ein:

```
<CustomAction Id="MyImmediateAction" BinaryKey="VbCa" DllEntry="MyImmediateAction"
              Execute="immediate" />
<CustomAction Id="MyDeferredAction" BinaryKey="VbCa" DllEntry="MyDeferredAction"
              Execute="deferred" Impersonate="no"/>

<InstallExecuteSequence>
    <Custom Action="MyImmediateAction" After="InstallInitialize" />
```

```
</InstallExecuteSequence>
```

19.3 Parameterübergabe in C++

19.3.1 Immediate-Action

In C++ haben wir zwar keine Development-Tools-Foundation-Klassen, aber wir haben ja die **Wca-Library**, die uns hier einige Funktionen zur Verfügung stellt. Zuerst lesen wir alle Parameter aus den Properties aus und fügen diese dann über die Funktion **WcaWriteStringToCaData** zu einem Parameterstring (hier pCustomActionData) aneinander.

Über **WcaWriteIntegerToCaData** bzw. **WcaWriteStreamToCaData** können auch Integer bzw. ganze Byte-Streams dem Parameterstring angehängt werden.

Wenn alle Parameter im Parameterstring sind, rufen wir die Funktion **WcaDoDeferredAction** auf, die den Parameterstring in das Property MyDeferredAction schreibt und mit MsiDoAction die Aktion MyDeferredAction aufruft:

```
UINT __stdcall MyImmediateAction(MSIHANDLE hInstall)
{
    HRESULT hr = S_OK;
    UINT er = ERROR_SUCCESS;

    LPWSTR pCustomActionData = NULL;
    LPWSTR logonUser = NULL;
    LPWSTR systemFolder = NULL;

    // Initialisierung
    hr = WcaInitialize(hInstall, "MyImmediateAction");
    ExitOnFailure(hr, "Failed to initialize");
    WcaLog(LOGMSG_STANDARD, "Initialized.");

    // Parameter ermitteln
    WcaGetProperty(L"LogonUser", &logonUser);
    WcaGetProperty(L"SystemFolder", &systemFolder);

    // Parameter übergeben und Aktion aufrufen
    WcaWriteStringToCaData(logonUser, &pCustomActionData);
    WcaWriteStringToCaData(systemFolder, &pCustomActionData);
    WcaDoDeferredAction(L"MyDeferredAction", pCustomActionData, 0);

    // Speicher aufräumen
    ReleaseMem(logonUser);
    ReleaseMem(systemFolder);
    ReleaseMem(pCustomActionData);

    LExit:
    er = SUCCEEDED(hr) ? ERROR_SUCCESS : ERROR_INSTALL_FAILURE;
    return WcaFinalize(er);
}
```

19.3.2 InScript-Action

Dafür, den Parameterstring auseinanderzuparsen stehen natürlich auch entsprechende Wca-Funktionen zur Verfügung. Zunächst lesen wir das Property **CustomActionData** aus, das den zusammengefügten Parameterstring enthält. Nun können wir aus dem Parameterstring über die Funktion **WcaReadStringFromCaData** die einzelnen Parameter zurücklesen:

```
UINT __stdcall MyDeferredAction(MSIHANDLE hInstall)
{
    HRESULT hr = S_OK;
    UINT er = ERROR_SUCCESS;

    LPWSTR pCustomActionData = NULL;
    LPWSTR logonUser = NULL;
    LPWSTR systemFolder = NULL;

    // Initialisierung
    hr = WcaInitialize(hInstall, "MyImmediateAction");
    ExitOnFailure(hr, "Failed to initialize");
    WcaLog(LOGMSG_STANDARD, "Initialized.");

    // Parameter ermitteln
    WcaGetProperty(L"CustomActionData", &pCustomActionData);
    WcaReadStringFromCaData(&pCustomActionData, &logonUser);
    WcaReadStringFromCaData(&pCustomActionData, &systemFolder);

    // Speicher aufräumen
    ReleaseMem(pCustomActionData);
    ReleaseMem(logonUser);
    ReleaseMem(systemFolder);

    LExit:
    er = SUCCEEDED(hr) ? ERROR_SUCCESS : ERROR_INSTALL_FAILURE;
    return WcaFinalize(er);
}
```

► **Hinweis:** Über **WcaReadIntegerFromCaData** bzw. **WcaReadStreamFromCaData** können Integer und Streams aus dem Parameterstring zurückgelesen werden.

19.3.3 WiX-Skript

Zum WiX-Skript gibt es nicht viel zu bemerken. Wie bei C# wird hier nur die Immediate-Action in die InstallExecuteSequenz eingebunden:

```
<Binary Id="Ca.dll" SourceFile="$(var.SetupHelperDll.TargetPath)"/>
<CustomAction Id="MyImmediateAction" DllEntry="MyImmediateAction" BinaryKey="Ca.dll"
Return="check" Execute="immediate"/>
<CustomAction Id="MyDeferredAction" DllEntry="MyDeferredAction" BinaryKey="Ca.dll"
Return="check" Execute="deferred" Impersonate="no"/>

<InstallExecuteSequence>
    <Custom Action="MyImmediateAction" After="InstallInitialize" />
</InstallExecuteSequence>
```

19.4 Parameterübergabe in VBScript

19.4.1 Immediate-Action

Da uns beim VBScript keine Unterstützung von irgendwelchen Klassenbibliotheken zur Verfügung steht, machen wir uns die Sache einfach. Wir hängen alle Parameter, durch ein Pipe-Zeichen getrennt, aneinander. Wir verwenden das Pipe-Zeichen als Trennzeichen, da dieses weder im Benutzernamen noch im Pfadnamen vorkommen darf:

```
function MyImmediateAction
    Dim logonUser
    Dim systemFolder

    ' Parameter ermitteln
    logonUser = Session.Property("LogonUser")
    systemFolder = Session.Property("SystemFolder")

    ' Parameter übergeben und Aktion aufrufen
    Session.Property("MyDeferredAction") = logonUser + " | " + systemFolder
    Session.DoAction("MyDeferredAction")

    MyImmediateAction=0
end function
```

19.4.2 InScript-Action

In VBScript gibt es die Funktion *split*, die aus dem Parameterstring ein String-Array erstellt. Da wir die Reihenfolge kennen, in der die Parameter in die Parameterliste eingefügt worden sind, ist es ein Einfaches, diese wieder zurückzulesen:

```
function MyDeferredAction
    Dim params
    Dim logonUser
    Dim systemFolder

    ' Parameter ermitteln
    params = Split(Session.Property("CustomActionData"), " | ")
    logonUser = params(0)
    systemFolder = params(1)

    ...

    MyDeferredAction=0
end function
```

19.4.3 WiX-Skript

Das WiX-Skript sieht so ähnlich wie der C++-Code aus:

```
<Binary Id="Ca.vbs" SourceFile="Ca.vbs"/>
<CustomAction Id="MyImmediateAction" VBScriptCall="MyImmediateAction"
               BinaryKey="Ca.vbs" Return="check" Execute="immediate"/>
<CustomAction Id="MyDeferredAction" VBScriptCall="MyDeferredAction"
               BinaryKey="Ca.vbs" Return="check" Execute="deferred"
               Impersonate="no"/>

<InstallExecuteSequence>
    <Custom Action="MyImmediateAction" After="InstallInitialize" />
</InstallExecuteSequence>
```

19.5 Parameterübergabe in JavaScript

19.5.1 Immediate-Action

Auch im JavaScript machen wir uns die Sache einfach, indem wir alle Parameter aneinanderhängen:

```
function MyImmediateAction() {
    var errorSuccess = 0;

    // Parameter ermitteln
    var logonUser = Session.Property("LogonUser");
    var systemFolder = Session.Property("SystemFolder");

    // Parameter übergeben und Aktion aufrufen
    Session.Property("MyDeferredAction") = logonUser + " | " + systemFolder;
    Session.DoAction("MyDeferredAction");

    return errorSuccess;
};
```

19.5.2 InScript-Action

In JavaScript verwenden wir die Methode *split* der String-Klasse, um aus dem Parameterstring einen String-Array zu erstellen. Da wir die Reihenfolge kennen, in der die Parameter in die Parameterliste eingefügt worden sind, ist es ein Einfaches, diese wieder zurückzulesen:

```
function MyDeferredAction() {
    var errorSuccess = 0;

    // Parameter ermitteln
    var params = Session.Property("CustomActionData").split("|");
    var logonUser = params[0];
    var systemFolder = params[1];

    ...

    return errorSuccess;
};
```

19.5.3 WiX-Skript

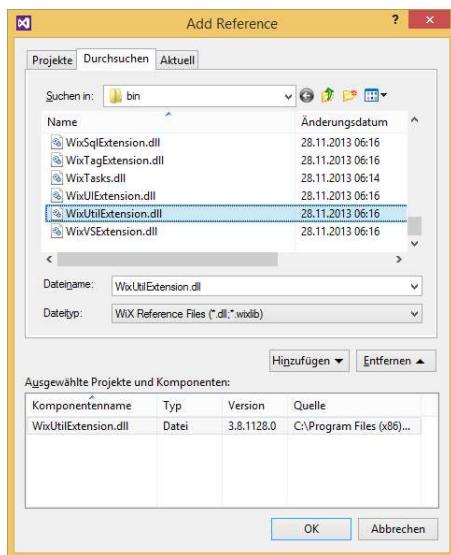
Das WiX-Skript bringt hier auch keine großen Überraschungen und folgt den weiter oben beschriebenen Abläufen:

```
<Binary Id="Ca.js" SourceFile="Ca.js"/>
<CustomAction Id="MyImmediateAction" JScriptCall="MyImmediateAction"
    BinaryKey="Ca.js" Return="check" Execute="immediate"/>
<CustomAction Id="MyDeferredAction" JScriptCall="MyDeferredAction"
    BinaryKey="Ca.js" Return="check" Execute="deferred"
    Impersonate="no"/>

<InstallExecuteSequence>
    <Custom Action="MyImmediateAction" After="InstallInitialize" />
</InstallExecuteSequence>
```

20 Anwendung im Hintergrund ausführen

Manchmal kommt man nicht umhin, Kommandozeilenanwendungen oder PowerShell-Skripte als Custom-Action zu starten. Hässlich schwarze Boxen trüben das professionelle Aussehen unseres Setups. Deshalb wäre es sicher schöner, wenn es eine Möglichkeit gäbe, diese Boxen zu unterdrücken. Dabei hilft uns die WixUtilExtension mit der Custom-Action CAQuietExec bzw. CAQuietExec64i.



Als Erstes binden wir die WixUtilExtension.dll als Abhängigkeit in unser Projekt ein.

Nun können wir die Custom-Action erstellen. Die Action ruft die Funktion `CAQuietExec` bzw. `CAQuietExec64` aus einer DLL auf, die in der Binär-Tabelle unter dem Namen `WixCA` eingetragen wurde. Die Funktion prüft beim Aufruf, ob die Action *immediate* oder *deferred* aufgerufen wurde.

20.1 Immediate Execution

Startet man die Action als *Immediate Execution*, so übergibt man die zu startende Anwendung samt Kommandozeile über das Property `QtExecCmdLine`:

```
<Property Id="QtExecCmdLine"
  Value="&quot;[SystemFolder]taskkill.exe&quot; /F /IM Notepad.exe"/>
<CustomAction Id="KillNotepad" BinaryKey="WixCA" DllEntry="CAQuietExec"
  Execute="immediate" Return="check"/>

<InstallExecuteSequence>
  <Custom Action="KillNotepad" Before="InstallValidate"/>
</InstallExecuteSequence>
```

Im dargestellten Beispiel beenden wir den Prozess namens `Notepad.exe`. Da wir den Rückgabewert prüfen, wird unsere Installation mit einem Fehler abgebrochen, wenn die Anwendung `taskkill.exe` einen Rückgabewert ungleich 0 zurückgibt.

Da der Pfad zur `taskkill.exe` Leerzeichen enthalten könnte, setzen wir den Pfad samt EXE-Name in Anführungszeichen – das Schlüsselwort `"` macht genau das.

20.2 Deferred Execution

Benötigt die zu startende Anwendung Admin-Rechte, so muss die Action verzögert ausgeführt (*Execute="deferred"*) und *Impersonate* muss auf "no" gesetzt werden. Die Übergabeparameter werden in diesem Fall über CustomActionData übergeben, wir müssen also die Kommandozeile in einem Property übergeben, das genauso heißt wie die Action selbst:

```
<Property Id="KillNotepad"
    Value=""[SystemFolder]taskkill.exe" /F /IM Notepad.exe"/>
<CustomAction Id="KillNotepad" BinaryKey="WixCA" DllEntry="CAQuietExec"
    Execute="deferred" Impersonate="no" Return="check"/>

<InstallExecuteSequence>
    <Custom Action="KillNotepad" After="InstallInitialize"/>
</InstallExecuteSequence>
```

20.3 64-Bit-Anwendung silent starten

Im oben aufgeführten Beispiel wird von einer 32-Bit-Anwendung ausgegangen. Soll eine 64-Bit-Anwendung gestartet werden, dann machen wir das mit der Funktion CAQuietExec64:

```
<Property Id="QtExecCmdLine"
    Value=""[SystemFolder]taskkill.exe" /F /IM Notepad.exe"/>
<CustomAction Id="KillNotepad" BinaryKey="WixCA" DllEntry="CAQuietExec64"
    Execute="immediate" Return="check"/>

<InstallExecuteSequence>
    <Custom Action="KillNotepad" Before="InstallValidate"/>
</InstallExecuteSequence>
```

21 Custom Table: Eigene MSI-Tabelle erstellen

In dieser Lektion lernen wir, wie eigene MSI-Tabellen erstellt werden. Doch bevor wir das tun, müssen wir noch klären, warum wir das überhaupt machen sollten bzw. wofür eigene Tabellen überhaupt sinnvoll sind.

Stellen wir uns vor, dass wir einen neuen Benutzer auf dem Zielsystem einrichten müssten. Relativ schnell finden wir heraus, dass Benutzer über die WMI-Schnittstelle eingerichtet werden können. Ein kurzes Beispielskript ist dank Google schnell gefunden. Das Problem scheint auf den ersten Blick einfach gelöst zu sein.

Wir ändern das Beispielskript entsprechend ab und starten die Custom-Action als Deferred-Action im Systemkontext, da wir ja Administratorrechte zum Anlegen des Benutzers benötigen. Wenn der Benutzername vom Anwender vorgegeben werden soll, brauchen wir noch eine zweite Custom-Action, die im Mode *immediate* arbeitet. Diese muss ja den Benutzernamen an die Deferred-Action per CustomActionData übergeben.

Wenn aber auch Rollback und Deinstallation berücksichtigt werden sollen, ist man schnell bei vier bis acht Custom-Actions, die eine entsprechende Komplexität mit sich bringen. Die fünf Zeilen Beispielskript werden dann schnell zu mehreren Seiten Code.

Wenn wir später einen zweiten Benutzer anlesen wollen oder die Funktionalität auch in einem anderen Projekt benötigen, müssen wir uns zuerst wieder in das Skript einlesen und das Ganze entsprechend anpassen.

Viel sinnvoller wäre es, sich die Arbeitsweise des Windows Installers zum Vorbild zu nehmen und die Aufgabe über eine **Custom-Table** zu lösen. In der Tabelle tragen wir ein, welche Benutzer einzurichten sind. Die Custom-Action holt sich alle nötigen Informationen aus der Tabelle und legt diese an. Wenn wir die zugehörigen Custom-Actions dann in eine Wix-Library auslagern, entsteht auf diese Art und Weise ein wiederverwendbarer Code, der sehr einfach in anderen Projekten verwendet werden kann.

21.1 Tabelle definieren

In diesem Kapitel sehen wir, wie eigene Tabellen über das Element **CustomTable** angelegt werden:

```
<CustomTable Id="CreateUser">
    ...
</CustomTable>
```

Das Id-Attribut gibt an, wie die Tabelle heißen soll. Über das **Column**-Element können dann die einzelnen Spalten definiert werden:

```
<CustomTable Id="CreateUser">
    <Column Id="UserName" Type="string" Width="255" Category="Text" PrimaryKey="yes"/>
    <Column Id="Password" Type="string" Width="255" Category="Text" Nullable="no"/>
    <Column Id="Component" Type="string" Width="72" Category="Identifier"
        KeyTable="Component" KeyColumn="1" Modularize="Column" />
</CustomTable>
```

Das Attribut **Type** gibt an, ob die Spalte einen String, einen Integer oder binäre Daten enthalten soll. Das Attribut **Category** hingegen ist für die Typenprüfung beim Kompilieren zuständig. Die Kategorie wird zum Teil auch in die Tabelle **_Validation** geschrieben, die zur Validierung des MSI-Paketes verwendet wird – Sichwort **Internal Consistency Evaluators (ICEs)**.

Wie wir in der Spalte *Component* sehen, können wir hier auch Referenzen auf andere Tabellen erstellen. Die Referenztabelle wird mit dem Attribut *KeyTable* bekanntgegeben und die Spaltennummer, beginnend mit 1 als erster Spalte, über *KeyColumn*.

Da die Spalte *Component* auf den Primary-Key der Component-Tabelle zeigt, muss das Attribut *Modularize* für die **Modularisierung** auf *Column* gesetzt werden. Die Modularisierung ist vor allem für **Merge Module** von Bedeutung. Merge Module kann man sich als kleine Setups vorstellen die beim Linken mit der light.exe in eine MSI-Datenbank zusammenkopiert werden.

21.2 Modularisierung

Um z.B. Dateien zu installieren, enthält das Merge Modul eine Component- und eine File-Tabelle. Beim Linken des Setups kopiert die light.exe einfach alle Tabellen des Merge Modules 1:1 in das zu erstellende MSI. Dieser Vorgang wird auch als *mergen* bezeichnet. Da die Primary-Keys einen eindeutigen Namen haben müssen, wird über die Modularisierung verhindert, dass zwischen Merge Module und dem WiX-Setup Namenskonflikte entstehen.

Stellen wir uns vor, wir haben ein Merge Module, dass die Datei test.txt installieren soll. Wir fügen im WiX-Code des Merge Modules also folgende Zeilen ein:

```
<Component Id="Test" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="test.txt" Source=".\\test.txt" KeyPath="yes"/>
</Component>
```

Wird das Merge Module kompiliert, dann hat der Primary-Key der File-Tabelle nicht den Eintrag *test.txt* sondern *test.txt.F0EA1205_F5FD_44F9_9F1E_ABBC6BFD755A*. Der an den Namen angehängte GUID-Schlüssel ist die ID des Merge Modules und natürlich für jedes Merge Module anders. Mit dem Attribut *Modularize="Column"* sagen wir also dem Kompiler, dass den Werten in der entsprechenden Spalte immer einen GUID angehängt wird, falls sich die Tabelle in einem Merge Module befindet.

Das Attribut *Modularize* kann noch weitere Werte annehmen:

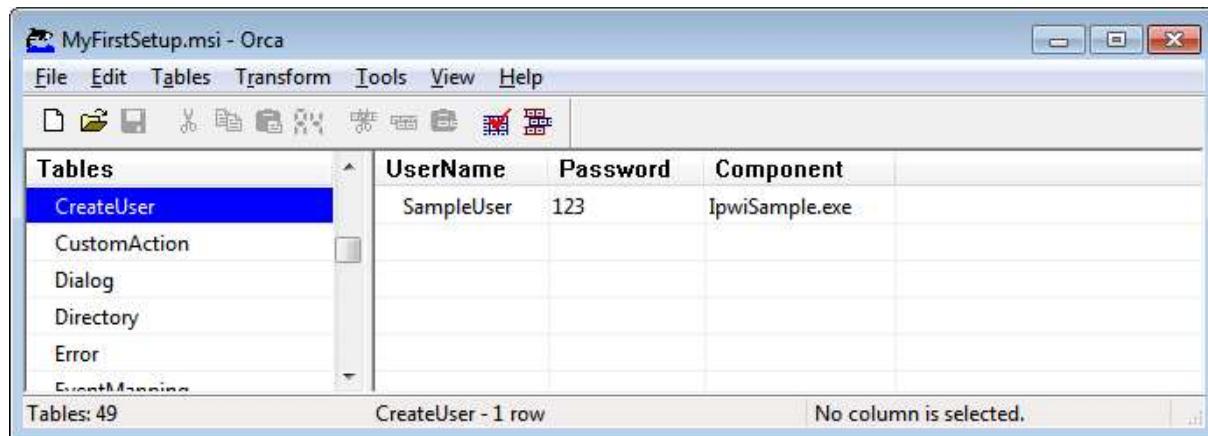
Wert	Bedeutung
None	Keine Modularisierung wird vorgenommen
Column	Dem Wert in der Spalte wird ein GUID-Schlüssel angehängt
Condition	In der Spalte ist eine Bedingung definiert. Allen angegebenen Properties wird ein GUID-Schlüssel angehängt
Icon	Wenn die Spalte ein Primary Key auf die Icon-Tabelle ist, muss diese speziell modularisiert werden.
Property	Allen Properties, die über die eckige Klammer referenziert werden, bekommen den GUID-Schlüssel angehängt
SemicolonDelimited	Die Spalte enthält eine Liste von Werten, die durch ein Semikolon getrennt sind. Alle Werte bekommen einen GUID-Schlüssel angehängt

21.3 Werte in die Tabelle schreibe

Doch was wäre eine Tabelle ohne Inhalt? Beim CustomTable-Element kann man auch über **Row** und **Data** die Tabelle füllen:

```
<CustomTable Id="CreateUser">
  <Row>
    <Data Column="UserName">SampleUser</Data>
    <Data Column="Password">123</Data>
    <Data Column="Component">IpwiSample.exe</Data>
  </Row>
</CustomTable>
```

WiX erlaubt mehrere CustomTable-Abschnitte mit derselben Id (hier CreateUser). Allerdings darf die Definition mittels Column nur einmal vorkommen. In Orca sieht die Tabelle dann folgendermaßen aus:



22 Windows Installer SQL-Syntax

Doch bevor wir uns ansehen, wie MSI-Tabellen gelesen werden können, müssen wir uns noch mit der SQL-Syntax beschäftigen.

Wenn wir lesend auf die Datenbank zugreifen wollen, machen wir das mit einem **SELECT**-Befehl:

SELECT fields FROM tables

Möchte man zum Beispiel alle Felder der FeatureComponents-Tabelle auslesen, dann sieht der Select-Befehl wie folgt aus:

*SELECT * FROM `FeatureComponents`*

Das Zeichen “*” bedeutet, dass alle Felder ausgelesen werden sollen. Man kann aber auch nur ganz spezifische Felder auswählen:

SELECT Feature_ ; Component_ FROM `FeatureComponents`

Um Konflikte zwischen Tabellennamen, Feldnamen oder reservierten SQL-Keywörtern zu vermeiden, können die Feld- und Tabellennamen in Backquotes (`) eingeschlossen werden:

SELECT `Feature_`, `Component_` FROM `FeatureComponents`

Wir können das Ergebnis auch auf nur ganz bestimmte Felder limitieren, indem wir eine **WHERE**- Clause angeben. So können wir z. B. alle Komponenten auslesen, die mit Feature MainFeature verbunden sind:

*SELECT * FROM `FeatureComponents` WHERE `Feature_`='MainFeature'*

Wenn mehrere Tabellen selektiert werden und die Feldnamen nicht eindeutig einer Tabelle zugeordnet werden können, kann man dem Feldnamen den Tabellennamen voranstellen:

*SELECT * FROM `FeatureComponents` WHERE `FeatureComponents`.`Feature_`='MainFeature'*

Wie wir im dargestellten Beispiel sehen, wird ein String (hier der Text MainFeature) in einfache Anführungszeichen gesetzt. Wenn alle Komponenten ermittelt werden sollten, die nicht im Feature MainFeature enthalten sind, dann sähe der Select-Befehl wie folgt aus:

*SELECT * FROM `FeatureComponents` WHERE `Feature_`<>'MainFeature'*

Ist der gesuchte Wert kein String, sondern ein Integer, dann verwendet man den Wert ohne Anführungszeichen. Das Größer- (>) und Kleiner-Zeichen (<) könnten natürlich auch verwendet werden. Manchmal möchte man Einträge suchen, bei denen in einem bestimmten Feld nichts eingegeben ist. Das kann man über folgenden Befehl erreichen:

*SELECT * FROM `Registry` WHERE `Value` IS NULL*

Oder man möchte nur Einträge haben, die keine leeren Einträge haben:

*SELECT * FROM `Registry` WHERE `Value` IS NOT NULL*

Die WHERE-Clause kann auch mit AND bzw. OR verknüpft werden:

*SELECT * FROM `Property` WHERE (`Property`='ProductName') OR (`Property`='ProductVersion')*

Sollen die Ergebniszellen auch noch sortiert werden, dann kann man das über **ORDER BY** machen:

```
SELECT `Action` FROM `InstallUISequence` ORDER BY `Sequence`
```

Wie bei anderen Datenbanken auch, kann man auf die MSI-Datenbank nicht nur lesend, sondern auch schreibend zugreifen. Um z. B. Zeilen aus einer Tabelle zu löschen, verwendet man den **DELETE**-Befehl:

```
DELETE FROM `Property` WHERE `Property`='TestProperty'
```

Über den **UPDATE**-Befehl können bestehende Zeilen auch geändert werden. Das kann allerdings nur mit Spalten gemacht werden, die nicht Primärschlüssel sind:

```
UPDATE `Property` SET `Value`='newValue' WHERE `Property`='TestProperty'
```

Will man mehrere Spalten gleichzeitig ändern, dann werden die Werte durch Komma getrennt angegeben:

```
UPDATE `Registry` SET `Name`='NewName', `Value`='newValue' WHERE `Registry`='MyEntry'
```

Über den **INSERT**-Befehl können auch neue Einträge in die Datenbank geschrieben werden:

```
INSERT INTO `FeatureComponents` (`Feature_`, `Component_`) VALUES ('MyFeat', 'MyComp')
```

Tabellen können mit **CREATE TABLE** neu erstellt, mit **ALTER TABLE** geändert und mit **DROP TABLE** gelöscht werden.

23 MSI-Tabelle über Custom-Action auslesen

Wie bereits angekündigt, wollen wir uns nun ansehen, wie MSI-Tabellen ausgelesen werden können. Als Erstes müssen wir einen **SELECT**-Befehl erstellen. Möchten wir z. B. die CustomTable CreateUser auslesen, so verwenden wir folgenden Select-Befehl:

```
SELECT * FROM `CreateUser`
```

Brauchen wir nur bestimmte Spalten (hier die Spalte UserName und Password), dann könnte das auch so aussehen:

```
SELECT `UserName`, `Password` FROM `CreateUser`
```

Im zweiten Schritt erstellen wir eine sogenannte View (Sicht) auf die Datenbank und führen den Select-Befehl aus. Danach können wir in einer Schleife Zeile für Zeile auslesen und verarbeiten.

23.1 MSI-Tabelle mit C# auslesen

Hier sehen wir, wie die Tabelle CreateUser ausgelesen wird:

```
public static ActionResult UserName(Session session)
{
    // View erstellen aus ausführen
    View view = session.Database.OpenView("SELECT * FROM `CreateUser`");
    view.Execute();

    // Zeile auslesen
    Record record = view.Fetch();

    // Alle Zeilen abarbeiten
    while (record != null)
    {
        string userName = record.GetString(1);
        string password = record.GetString(2);
        string component = record.GetString(3);
        ...

        // Nächste Zeile holen
        record = view.Fetch();
    }

    return ActionResult.Success;
}
```

Zu beachten ist, dass das erste Feld des Records bei GetString mit dem Index 1 ausgelesen wird. Anstatt der While-Schleife können wir auch eine Foreach-Schleife verwenden:

```
public static ActionResult UserName(Session session)
{
    View view = session.Database.OpenView("SELECT * FROM `CreateUser`");
    view.Execute();

    foreach(Record record in view)
    {
        string userName = record.GetString(1);
        string password = record.GetString(2);
        string component = record.GetString(3);
        ...

    }
    return ActionResult.Success;
}
```

23.2 MSI-Tabelle mit VB.NET auslesen

In VB.NET sieht das Ganze dann so aus:

```
Public Shared Function UserName(ByVal session As Session) As ActionResult
    ' View erstellen aus ausführen
    Dim view As View = session.Database.OpenView("SELECT * FROM `CreateUser`")
    view.Execute()

    ' Alle Zeilen abarbeiten
    For Each record In view
        Dim Name As String = record.GetString(1)
        Dim password As String = record.GetString(2)
        Dim component As String = record.GetString(3)
        ...
    Next

    Return ActionResult.Success
End Function
```

23.3 MSI-Tabelle mit C++ auslesen

Der Code in C++ ist um einige Zeilen länger – und wir haben hier gänzlich auf die Fehlerbehandlung verzichtet:

```
UINT __stdcall UserName(MSIHANDLE hInstall)
{
    HRESULT hr = S_OK;
    UINT er = ERROR_SUCCESS;
    MSIHANDLE hView, hRecord;

    hr = WcaInitialize(hInstall, "UserName");
    ExitOnFailure(hr, "Failed to initialize");

    // View erstellen
    WcaOpenView(L"SELECT * FROM `CreateUser`", &hView);
    WcaExecuteView(hView, NULL);

    // Alle Zeilen abarbeiten
    while (S_OK == WcaFetchRecord(hView, &hRecord))
    {
        LPWSTR sUserName = NULL, sPassword = NULL, sComponent = NULL;

        WcaGetString(hRecord, 1, &sUserName);
        WcaGetString(hRecord, 1, &sPassword);
        WcaGetString(hRecord, 1, &sComponent);
        ...

        // Speicher aufräumen
        ReleaseMem(sUserName);
        ReleaseMem(sPassword);
        ReleaseMem(sComponent);
        MsiCloseHandle(hRecord);
    }
    MsiViewClose(hView);
    MsiCloseHandle(hView);

    LExit:
    er = SUCCEEDED(hr) ? ERROR_SUCCESS : ERROR_INSTALL_FAILURE;
    return WcaFinalize(er);
}
```

23.4 MSI-Tabelle mit VBScript auslesen

Auch das Session-Objekt in den VBScript-Actions unterstützt über das **Database-Objekt** das Auslesen der MSI-Tabellen:

```
function UserName
    ' View erstellen aus ausführen
    Set View = Session.Database.OpenView("SELECT * FROM `CreateUser`")
    View.Execute

    ' Alle Zeilen abarbeiten
    Set Record = View.Fetch
    While Not Record Is Nothing
        UserName = Record.StringData(1)
        Password = Record.StringData(2)
        Component = Record.StringData(3)
        ...

        ' Nächste Zeile holen
        Set Record = View.Fetch
    Wend

    UserName=0
end function
```

23.5 MSI-Tabelle mit JavaScript auslesen

Und hier noch das Ganze in JavaScript:

```
function UserName() {
    var errorSuccess = 0;

    // View erstellen aus ausführen
    var params = View = Session.Database.OpenView("SELECT * FROM `CreateUser`");
    View.Execute();

    // Alle Zeilen abarbeiten
    var Record = View.Fetch();
    while(Record) {
        var userName = Record.StringData(1);
        var password = Record.StringData(2);
        var component = Record.StringData(3);
        ...

        // Nächste Zeilen holen
        Record = View.Fetch();
    }

    return errorSuccess;
};
```

24 In MSI-Tabelle mit Custom-Action schreiben

In diesem Kapitel sehen wir, wie wir über eine Custom-Action in MSI-Tabellen schreiben können. Wir können also zur Laufzeit aktiv auf die Installation einwirken und Änderungen vornehmen.

Doch zuvor müssen wir wohl klären, warum wir das überhaupt machen sollten bzw. wo die Anwendungsfälle zu finden sind:

a) Anwendungsfall: ListBox- bzw. ComboBox-Tabelle

Das Dialog-Control ListBox bzw. ComboBox wird über die **ListBox- bzw. ComboBox-Tabelle** gefüllt. Soll der Anwender eine Liste von Elementen bekommen, die erst zur Laufzeit auf dem Zielsystem ermittelt werden kann (wie z. B. eine Liste aller SQL-Server), dann müssen wir eine Custom-Action erstellen, die in die ListBox- bzw. ComboBox-Tabelle schreibt.

Dieser Anwendungsfall wird weiter hinten im Kapitel der SQL-Datenbanken betrachtet.

b) Anwendungsfall: dynamischer Shortcut

Der Anwender wird über einen Dialog gefragt, ob ein Desktop-Icon erstellt werden soll. Entscheidet sich der Anwender für ein Desktop-Icon, könnte dieser über einen dynamisch erstellten Eintrag in der Shortcut-Tabelle erstellt werden.

Diesen Anwendungsfall wollen wir anhand eines Beispiels hier aufzeigen.

► **Hinweis:** Über das Session-Handle können Änderungen grundsätzlich nur temporär durchgeführt werden. Das bedeutet: Änderungen gelten nur für den einzelnen Setupdurchlauf. Daher müssen Änderungen eventuell nicht nur bei der Erstinstallation durchgeführt werden, sondern auch bei der Reparatur und der Deinstallation.

24.1 Vorbereitung

Wie bereits erwähnt, wollen wir als Beispiel einen Shortcut auf den Desktop erstellen. Hierzu erstellen wir zuerst das Property CREATE_DESKTOP_SHORTCUT, das wir für die Reparatur bzw. Deinstallation in die Registry schreiben und dort wieder über RegistrySearch zurücklesen:

```
<Property Id="CREATE_DESKTOP_SHORTCUT" Value="1" Secure="yes">
  <RegistrySearch Id="DesktopShortcut" Root="HKLM" Key="Software\[ProductName]"
    Name="DesktopShortcut" Type="raw" />
</Property>

<ComponentGroup Id="DesktopShortcut">
  <Component Id="DesktopShortcut" Guid="YOURGUID" Directory="INSTALLFOLDER">
    <RegistryValue Id="DesktopShorctut" Root="HKLM" Key="Software\[ProductName]"
      Name="DesktopShortcut" Value="[CREATE_DESKTOP_SHORTCUT]"
      Type="string" KeyPath="yes"/>
  </Component>
</ComponentGroup>
```

Um über ein Template für den Shortcut zu verfügen, legen wir einen Programm-Menü-Shortcut unter dem Namen ProgramMenuShortcut an. Diesen Shortcut lesen wir im Skript aus und ändern den Primary-Key sowie das Zielverzeichnis entsprechend ab:

```
<Component Id="Component1" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="File1" Source=".\\Source\\Program V1.0\\IpwiSample1.exe" KeyPath="yes">
    <Shortcut Id="ProgramMenuShortcut" Advertise="yes" Name="MySample"
      Directory="ProgramMenuFolder" Icon="IpwiSample.exe"/>
  </File>
</Component>
```

Natürlich müssen die Directory-Variablen ProgramMenuShortcut und DesktopFolder im Directory-Element vorhanden sein. Da das Einbinden der Custom-Action unabhängig von der verwendeten Programmiersprache ist, binden wir diese gleich in die InstallExecute-Sequenz ein:

```
<InstallExecuteSequence>
  <Custom Action="CreateDesktopSc" After="CostFinalize">
    CREATE_DESKTOP_SHORTCUT=1
  </Custom>
</InstallExecuteSequence>
```

24.2 In MSI-Tabelle mit C# schreiben

Im Code sehen wir, dass zuerst der vorhandene Shortcut über einen Select-Befehl und der **WHERE**-Clause ausgelen werden. Danach wird der Record entsprechend abgeändert und über InsertTemporary in die Tabelle zurückgeschrieben:

```
public static ActionResult CreateDesktopSc(Session session)
{
  Record record = new Record();

  // ProgramMenuShortcut auslesen
  View view = session.Database.OpenView(
    "SELECT * FROM `Shortcut` WHERE `Shortcut`='ProgramMenuShortcut'");
  view.Execute();
  record = view.Fetch();

  // Shortcut ändern und in Tabelle schreiben
  record.SetString("Shortcut", "DesktopFolderShortcut");
  record.SetString("Directory_", "DesktopFolder");
  view.InsertTemporary(record);

  return ActionResult.Success;
}
```

► **Hinweis:** Um innerhalb der Tabelle Werte hinzuzufügen, benötigen wir immer eine View, die über einen SELECT-Befehl erstellt wurde. Das ist auch dann der Fall, wenn kein vorhandener Eintrag ausgelesen, sondern der Record komplett von Hand aufbaut wird.

24.3 In MSI-Tabelle mit VB.NET schreiben

In VB.NET sieht das Ganze dann so aus:

```
Public Shared Function CreateDesktopSc(ByVal session As Session) As ActionResult
  ' ProgramMenuShortcut auslesen
  Dim view As View = session.Database.OpenView(
    "SELECT * FROM `Shortcut` WHERE `Shortcut`='ProgramMenuShortcut'")
  view.Execute()
  Dim record = view.Fetch()

  ' Shortcut ändern
  record.SetString("Shortcut", "DesktopFolderShortcut")
  record.SetString("Directory_", "DesktopFolder")

  ' Shortcut in Tabelle schreiben
  view.InsertTemporary(record)

  Return ActionResult.Success
End Function
```

24.4 In MSI-Tabelle mit C++ schreiben

Trotz WCA-Unterstützung ist der C++-Code etwas länger als der Managed Code. Den Record schreiben wir hier mit dem Windows Installer API-Befehl **MsiViewModify** in die Tabelle.

```
UINT __stdcall CreateDesktopSc(MSIHANDLE hInstall)
{
    HRESULT hr = S_OK;
    UINT er = ERROR_SUCCESS;
    MSIHANDLE hView, hRecord;

    // Initialisierung
    hr = WcaInitialize(hInstall, "UserName");
    ExitOnFailure(hr, "Failed to initialize");

    // View erstellen
    WcaOpenView(L"SELECT * FROM `Shortcut` WHERE `Shortcut`='ProgramMenuShortcut'", &hView);
    WcaExecuteView(hView, NULL);
    WcaFetchRecord(hView, &hRecord);

    // Shortcut abändern
    WcaSetRecordString(hRecord, 1, L"DesktopFolderShortcut");
    WcaSetRecordString(hRecord, 2, L"DesktopFolder");

    // Shortcut in Tabelle schreiben
    MsiViewModify(hView, MSIMODIFY_INSERT_TEMPORARY, hRecord);

    MsiCloseHandle(hRecord);
    MsiViewClose(hView);
    MsiCloseHandle(hView);

    LExit:
    er = SUCCEEDED(hr) ? ERROR_SUCCESS : ERROR_INSTALL_FAILURE;
    return WcaFinalize(er);
}
```

24.5 In MSI-Tabelle mit VBScript schreiben

Über die Methode *Modify* vom View-Objekt werden Daten in die MSI-Tabellen geschrieben:

```
function CreateDesktopSc
    Const msiViewModifyInsertTemporary = 7

    ' ProgramMenuShortcut auslesen
    Set View = Session.Database.OpenView("SELECT * FROM `Shortcut` WHERE `Shortcut`='ProgramMenuShortcut'")
    View.Execute
    Set Record = View.Fetch

    ' Shortcut abändern
    Record.StringData(1) = "DesktopFolderShortcut"
    Record.StringData(2) = "DesktopFolder"

    ' Shortcut in Tabelle schreiben
    View.Modify msiViewModifyInsertTemporary, Record

    CreateDesktopSc=0
end function
```

24.6 In MSI-Tabelle mit JavaScript schreiben

Und hier noch das Ganze in JavaScript:

```
function CreateDesktopSc() {
    var errorSuccess = 0;
    var msiViewModifyInsertTemporary = 7;

    // ProgramMenuShortcut auslesen
    var params = View = Session.Database.OpenView(
        "SELECT * FROM `Shortcut` WHERE `Shortcut`='ProgramMenuShortcut'");
    View.Execute();
    var Record = View.Fetch();

    // Shortcut abändern
    Record.StringData(1) = "DesktopFolderShortcut";
    Record.StringData(2) = "DesktopFolder";

    // Shortcut in Tabelle schreiben
    View.Modify(msiViewModifyInsertTemporary, Record);

    return errorSuccess;
}
```

25 WixNetFxExtension

Wenn wir Managed Code als Custom-Action einbinden, sind wir auf das .NET-Framework angewiesen. Wenn wir ein Setup mit Managed-Code-Custom-Actions auf einem Rechner starten, auf dem kein .NET-Framework installiert ist, dann wird das Setup abbrechen, weil die Custom-Actions nicht aufgeführt werden können.

Die feine englische Art wäre es, wenn wir den Benutzer am Anfang unseres Setups schon drauf hinweisen würden, dass das .NET-Framework benötigt wird. Wir sollten also Installationsbedingungen einfügen.

Doch wie finden wir heraus, welches .NET-Framework installiert ist? Natürlich gibt es dafür Registry-Einträge. Wenn man aber sieht, in wie viel Versionen und Sprachen das .NET-Framework verfügbar ist, ahnt man: Das wäre sehr viel Arbeit. Aber WiX wäre nicht WiX, wenn da nicht schon ein schlauer Kopf Pionierarbeit für uns geleistet hätte. Die Ergebnisse der Arbeit sind in der WiX-Erweiterung **WixNetfxExtention** abgelegt.

WixNetfxExtention ist eine große Sammlung von Properties, aus der wir alle derzeit verfügbaren Framework-Versionen, deren Servicepacks sowie die Spracherweiterungen herausfinden können.

Bevor wir nun auf die einzelnen Properties zugreifen, müssen wir die Erweiterung zuerst in unsere Liste der Abhängigkeiten aufnehmen. Aber das wissen Sie ja bereits.

Da jedes Property ein separates Fragment besitzt, müssen wir das zu benutzenden Property über **PropertyRef** referenzieren:

```
<PropertyRef Id="NETFRAMEWORK20"/>
```

Danach können wir die Installationsbedingung erstellen:

```
<Condition Message="!(loc.InstallConditionNetFx2)">
    Installed OR NETFRAMEWORK20
</Condition>
```

Wenn wir das Setup über die Kommandozeile erstellen wollen, müssen wir der light.exe die Erweiterung mit geben:

```
candle.exe MySample.wxs
light.exe -ext WixNetFxExtension MySample.wixobj
```

Die wichtigsten Properties in **WixNetfxExtention** sind:

Property	Beschreibung
NETFRAMEWORKxx	Gibt an, ob das entsprechende Framework installiert ist (außer Framework 4.0)
NETFRAMEWORK40FULL	Gibt an, ob das Framework 4.0 voll installiert ist
NETFRAMEWORKxx_SP_LEVEL	Service-Pack-Level der einzelnen Frameworks
NETFRAMEWORKxxINSTALLROOTDIR	Installationsordner der einzelnen Frameworks

Die Zeichen xx müssen durch die Versionsnummer des .Net-Frameworks ersetzt werden. Möchte man wissen, ob das .Net-Framework 4.5 installiert ist, dann heißt zugehörige Property NETFRAMEWORK45. Die Anzahl der Properties ist einfach zu groß und deren Darstellung würde den Rahmen dieses Buches sprengen. Eine komplette Referenz aller Properties findet man in der Hilfe des WiX-Toolsets.

26 Hier kommt die Sonne – die Anwendung Heat

In den vergangenen Lektionen haben wir gesehen, dass es unter Umständen sehr mühsam sein kann, wenn unsere Anwendung hunderte oder gar tausende von Dateien mitbringt. Wenn wir diese Dateien alle von Hand eintragen müssen, dann sind wir mehrere Tage beschäftigt und danach wahrscheinlich reif für das Narrenhaus.

Wahrscheinlich hat sich der Entwickler von **Heat.exe** (Bestandteil des WiX Toolsets) auch vor diesem Schritt bewahren wollen. Heat ist ein Tool zum Ernten bzw. Sammeln von Daten. Dies können Dateien, DLLs, ActiveX-Controls, Performance Counter, Webseiten und Verzeichnisse sein. Wenn man beabsichtigt, Minor-Updates oder Patches zu erstellen, sollte Heat, vor allem wegen der Komponenten-Regel, nicht direkt in den Build-Prozess eingebaut zu werden. In diesem Fall sollte Heat eher dafür verwendet werden, einmalig bestimmte Dateien oder Verzeichnisse abzuscanen und alle weiteren Änderungen von Hand durchzuführen. Möchte man nur Major-Updates verteilen, dann steht der Verwendung von Heat direkt im Build-Prozess nichts im Wege.

26.1 Ordnerstruktur mit vielen Dateien abscannen

In diesem Abschnitt wollen wir uns ansehen, wie eine ganze Ordnerstruktur durchsucht und in ein WiX-Fragment geschrieben werden kann. Der Aufruf von Heat ist wie folgt:

```
heat dir <Ordner> -cg MyComponentGroup -gg -out SampleGroup.wxs
```

Der Suchtyp *dir* gibt an, dass ein ganzes Verzeichnis gescannt werden soll. Mit dem Namen *-cg* geben wir die **ComponentGroup** an und *-gg* bedeutet, dass Heat uns auch gleich neue Komponenten-GUIDs erstellen soll. Geben wir *-gg* nicht an, so wird der GUID der Komponente so eingetragen Guid="PUT-GUID-HERE".

Die von Heat erstellte ComponentGroup wird nach dem Aufruf genauso viele Komponenten enthalten, wie es Dateien im abzuscannden Ordner gibt – es wird also für jede Datei genau eine Komponente erstellt. Zusätzlich zu den Komponenten werden auch alle Verzeichnisse referenziert. Standardmäßig geht Heat vom Rootverzeichnis TARGETDIR aus. Wenn die Dateien ausgehend vom angegebenen Verzeichnis unterhalb von INSTALLDIR liegen sollen, dann müssen wir das Verzeichnis über das Argument *-dr* angeben. Der Aufruf sieht dann also so aus:

```
heat dir <Ordner> -dr INSTALLDIR -cg SampleGroup -gg -out SampleGroup.wxs
```

Heat legt automatisch unterhalb des mit *-dr* übergebenen Verzeichnisses ein Unterverzeichnis an. Soll dies nicht geschehen, so muss noch das Attribut *-srd* mit übergeben werden.

26.1.1 Tutorial einbinden

Genug der Theorie, jetzt zum praktischen Teil. Wir wollen ein Tutorial einbauen, welches aus einer Webseite mit ca. 100 Dateien besteht. Diese wollen wir mittels Heat in unser Setup mit aufnehmen. Hierzu erstellen wir zuerst ein Feature:

```
<Feature Id="Tutorial" Title="!(loc.FeatureTitleTutorial)"  
        Description="!(loc.FeatureDescriptionTutorial)" Level="1">  
    <ComponentGroupRef Id="TutorialGroup"/>  
</Feature>
```

Die Dateien sollen in das Verzeichnis Tutorial unterhalb von INSTALLDIR abgelegt werden. Wir legen also dieses Verzeichnis von Hand an und ausgehend von diesem Verzeichnis sollen dann die anderen Unterverzeichnisse von Heat erstellt werden.

Die Directory-Definition sieht dann so aus:

```
<Directory Id="TARGETDIR" Name="SourceDir">
    <Directory Id="ProgramFilesFolder">
        <Directory Id="CompynaName" Name="MyCompany">
            <Directory Id="INSTALLDIR" Name="MyFirstSetup">
                <Directory Id="TUTORIAL" Name="Tutorial"/>
            </Directory>
        </Directory>
    </Directory>
</Directory>
```

Die Dateien des Tutorials liegen im Unterverzeichnis Tutorial unserer Beispieldateien. Wir gehen in das Verzeichnis, in dem die anderen WXS-Dateien liegen und rufen Heat mit folgender Befehlszeile auf:

```
heat dir SourceDir\Tutorial -dr TUTORIAL -srd -cg TutorialGroup -gg -out Tutorial.wxs
```

Das Attribut –srd geben wir an, da wir das Verzeichnis TUTORIAL bereits weiter oben definiert haben – sonst hätten wir auf dem Zielsystem zwei gleichnamige Verzeichnisse untereinander.

Das Erstellen der WXS-Datei geht erstaunlich schnell. Nachdem wir Tutorial.wxs in unser Visual Studio Projekt durch *Hinzufügen▶ Vorhandenes Element...* aufgenommen haben, sehen wir uns diese Datei doch etwas genauer an:

```
<ComponentGroup Id="TutorialGroup">
    <Component Id="cmp559B" Directory="dir0A1D" Guid="YOURGUID">
        <File Id="filA456" KeyPath="yes" Source="SourceDir\index.html" />
    </Component>
    ...
</ComponentGroup>
```

Uns fällt auf, dass der Pfad zu den Dateien noch nicht richtig stimmt. Bei Source steht „SourceDir\index.html“, dort müsste aber „SourceDir\Tutorial\index.html“ stehen. Also müssen wir das noch per *Suchen und Ersetzen* ändern.

Alternativ können wir mit dem Parameter –var Heat sagen, dass er den Quellpfad über eine Präprozessor Variable ersetzen soll:

```
heat dir SourceDir\Tutorial -dr TUTORIAL -srd -cg TutorialGr -gg -var var.SrcTutor -out Tutorial.wxs
```

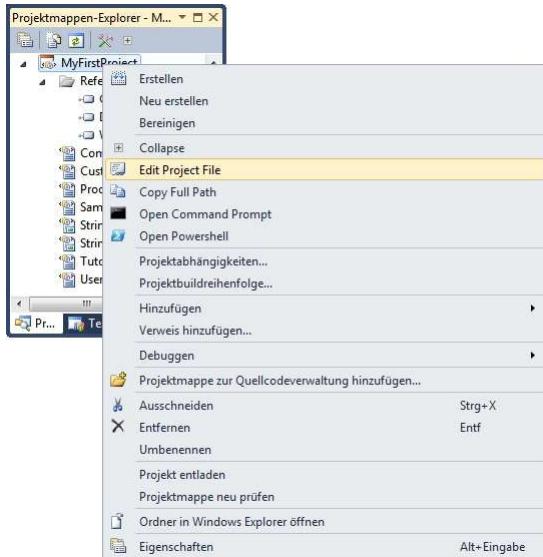
Der erstellte Code sieht dann in etwa so aus:

```
<ComponentGroup Id="TutorialGroup">
    <Component Id="cmp559B" Directory="dir0A1D" Guid="YOURGUID">
        <File Id="filA456" KeyPath="yes" Source="=$(var.SrcTutor)\index.html" />
    </Component>
    ...
</ComponentGroup>
```

Die Präprozessor-Variable SrcTutor muss dann beim Komplizieren mit Candle.exe entsprechend übergeben werden.

26.2 Abscannen der Ordnerstruktur im Build-Prozess

Wie bereits erwähnt kann Heat auch in den Build-Prozess integriert werden. Wenn man keine Minor-Updates und Patches erstellen möchte, dann ist das eine sehr interessante Alternative.



Der Aufruf von Heat.w wird direkt in die Projekt-Datei eingetragen. Hierzu müssen wir das Projekt-File editieren. Das macht man über den Menüpunkt *Edit Project File* der sichtbar wird, wenn man auf das WiX-Projekt mit der rechten Maustaste klickt.

Unsere Projektdatei sieht dann etwa so aus:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    ...
  </PropertyGroup>
  <ItemGroup>
    ...
  </ItemGroup>
  <Import Project="$(WiXTargetsPath)" />
  <!-- To modify your build process, add your task inside one of the targets below.
  <Target Name="BeforeBuild">
    </Target>
    <Target Name="AfterBuild">
      </Target>
    -->
  </Project>
```

Nun können wir unterhalb des Import-Elements als Prebuild-Event das **HeatDirectory** Element eintragen:

```
<Target Name="BeforeBuild">
  <HeatDirectory ToolPath="$(WiXToolPath)"
    Directory=".\\SourceDir\\Tutorial"
    OutputFile="Tutorial.wxs"
    ComponentGroupName="TutorialGroup"
    DirectoryRefId="TUTORIAL"
    GenerateGuidsNow="true"
    PreprocessorVariable="var.TutorialDir" />
</Target>
```

Somit wird der Tutorial-Ordner bei jedem Build-Prozess abgescannt und ins Setup aufgenommen.

26.3 COM-Server und ActiveX-Controls registrieren

26.3.1 Was sind COM-Server bzw. ActiveX-Controls?

COM-Server und ActiveX-Controls sind DLLs, OCX-Dateien oder EXEs, die ein Automatisierungs-Interface zur Verfügung stellen. Das Automatisierungs-Interface wird in die Registry (unter HKEY_CLASSES_ROOT) mit der ProgID, der ClassID und der Typenbibliothek eingetragen.

In konventionellen Setups wurde die Registrierung von DLLs und OCX-Dateien über die Kommandozeilen-Anwendung **RegSvr32.exe** durchgeführt, die EXEs wurden mit der Kommandozeile /REGSERVER aufgerufen. Diese Registrierungsart nennt sich *Selbstregistrierung* und bringt ein paar entscheidende Nachteile mit sich. Wenn die zu registrierende Datei statische Abhängigkeiten zu anderen DLLs hat und die abhängigen DLLs sind zur Installationszeit noch nicht vorhanden, dann scheitert die Registrierung – das Automatisierungs-Interface kann nicht aufgerufen werden. Deshalb hat Microsoft die Strategie zur Installation dieser Dateien mit dem Windows Installer grundlegend geändert.

Mit den **Windows Installer Best Practices** wird u.A. definiert, dass die Registrierung über entsprechende Tabellen (**Class**, **ProgID**, **TypeLib** sowie **Registry**) durchgeführt werden soll.

Das Problem das uns als Setup-Ersteller nun stellt ist, welche Einträge müssen wir in die entsprechenden Tabellen eintragen. Eine ClassID ist ein GUID-Schlüssel - und ein COM-Server kann mitunter viele duzend, ja sogar hunderte davon haben. Mit der ProgID sieht dies ähnlich aus. Es wäre eine Sisyphusarbeit, das Ganze von Hand vornehmen zu müssen.

Aber keine Angst, Heat hilft uns bei dieser Aufgabe. Die Registrierungs-Informationen können über folgende Kommandozeile ermittelt und in ein Fragment geschrieben werden:

```
heat file <Datei> -cg MyComponentGroup -out SampleGroup.wxs
```

26.3.2 ActiveX-Control hinzufügen

Nun zu unserem Beispiel. Wir wollen das ActiveX-Control IpwiActiveX.ocx installieren. Die Erste Frage die sich stellt ist das Zielverzeichnis. Wir wissen jetzt, dass ActiveX-Controls registriert werden. Aufgerufen wird ein ActiveX-Control über die ProgID, nicht über den Dateinamen. Deshalb ist es eigentlich egal, wohin das Control installiert wird. Aber ganz egal ist es nicht! Wird das ActiveX-Control noch von einer anderen Anwendung installiert, dann müssen wir auf jeden Fall sicherstellen, dass beide Anwendungen das Control in dasselbe Verzeichnis installieren. Ansonsten wird immer das Control, welches zuletzt installiert wurde, von beiden Anwendungen aufgerufen.

Die hieraus resultierenden Probleme sind zum einen, dass die zweite Anwendung eine ältere Version des Controls installieren könnte die dann das erwartete Interface der neueren Version überhaupt nicht unterstützt und zum anderen, dass die zweite Anwendung das Control bei der Deinstallation einfach entfernen und deregistrieren würde. Die erste Anwendung würde auf ein Control zugreifen, welches nicht mehr vorhanden ist. Diese Probleme sind unter anderem auch als **DLL-Hell** bekannt.

Deshalb hier eine Faustregel: Wenn wir nicht 100% sicher sind, dass das Control nicht auch noch mit einer anderen Anwendung installiert wird, installieren wir das Control ins System-Verzeichnis. Dort sollten in der Win32-Welt alle „gemeinsam benutzen“ DLLs abgelegt werden.

Um also unser Control ins System-Verzeichnis ablegen zu können, müssen wir dieses Verzeichnis zuerst definieren:

```
<Directory Id="TARGETDIR" Name="SourceDir">
    <Directory Id="SystemFolder" Name="SystemFolder" />
</Directory>
```

Der Windows Installer definiert uns bereits ein Property namens **SystemFolder**. Deshalb heißt unsere Directory-Variable genauso und wird automatisch auf den Wert des Windows Installer Properties gesetzt.

Nun können wir Heat aufrufen, um das neue Fragment zu erstellen:

```
heat file "IpwiActiveX.ocx" -srd -cg IpwiActiveX -gg -dr SystemFolder -out IpwiActiveX.wxs
```

Wir erstellen eine neue ComponentGroup Namens IpwiActiveX und lassen durch –gg auch gleich einen neuen GUID-Schlüssel erstellen. Das von Heat erstellte Fragment sieht dann wie folgt aus:

```
<?xml version="1.0" encoding="utf-8"?>
<WiX xmlns="http://schemas.microsoft.com/WiX/2006/wi">
    <Fragment>
        <DirectoryRef Id="SystemFolder" />
    </Fragment>
    <Fragment>
        <ComponentGroup Id="IpwiActiveX">
            <Component Id="cmp7117364CDEF1D17216047F05A90D814E"
                Directory="SystemFolder" Guid="YOURGUID">
                <File Id="fil1EB6950C7060D58B580C616676E1E674" KeyPath="yes"
                    Source="SourceDir\ActiveX V1.00\IpwiActiveX.ocx">
                    <Class Id="{41BDCB0F-B1C7-44F0-93A4-C917471F3505}"
                        Context="InprocServer32"
                        Description="IpwiActiveX Property Page" />
                    <TypeLib Id="{3B70D823-6994-4AF8-84A0-093DCBE47C62}" Control="yes"
                        Description="IpwiActiveX ActiveX-Steuerelement-Modul"
                        HelpDirectory="SystemFolder" Language="0" MajorVersion="1"
                        MinorVersion="0">
                        <Class Id="{0BBFA4A4-0750-49D0-A5C4-33BB317C5C54}"
                            Context="InprocServer32" Description="IpwiActiveX Control"
                            ThreadingModel="apartment" Version="1.0" Insertable="yes"
                            Control="yes">
                            <ProgId Id="IPWIACTIVEX.IpwiActiveXCtr1.1"
                                Description="IpwiActiveX Control" />
                        </Class>
                        <Interface Id="{67878CDD-9D51-43DB-A16B-E1CF6BE41F58}"
                            Name="_DIpwiActiveXEvents" ProxyStubClassId32=
                            "{00020420-0000-0000-C000-00000000046}" />
                        <Interface Id="{DA175B59-B376-42EA-ADFE-456A4FFB1BC5}"
                            Name="_DIpwiActiveX" ProxyStubClassId32=
                            "{00020420-0000-0000-C000-00000000046}" />
                    </TypeLib>
                </File>
            <RegistryValue Root="HKCR"
                Key="CLSID\{0BBFA4A4-0750-49D0-A5C4-33BB317C5C54}\MiscStatus\1"
                Value="197009" Type="string" Action="write" />
            <RegistryValue Root="HKCR"
                Key="CLSID\{0BBFA4A4-0750-49D0-A5C4-33BB317C5C54}\MiscStatus"
                Value="0" Type="string" Action="write" />
            <RegistryValue Root="HKCR" Key=
                "CLSID\{0BBFA4A4-0750-49D0-A5C4-33BB317C5C54}\ToolboxBitmap32"
                Value="![fil1EB6950C7060D58B580C616676E1E674], 1" Type="string"
                Action="write" />
            <RegistryValue Root="HKCR"
                Key="IPWIACTIVEX.IpwiActiveXCtr1.1\Insertable" Value=""
                Type="string" Action="write" />
        </Component>
    </ComponentGroup>
    </Fragment>
</WiX>
```

Um sicher zu gehen, dass Windows bei der Deinstallation aufpasst, dass die Datei nur dann deinstalliert wird, wenn Sie von keiner anderen Anwendung benötigt wird, setzen wir in der Komponente des Controls das Flag **SharedDllRefCount** auf den Wert yes:

```
<Component Id="cmp7117364CDEF1D17216047F05A90D814E" Directory="SystemFolder"
    Guid "YOURGUID" SharedDllRefCount="yes">
```

Durch das Setzen von SharedDllRefCount, erhöht der Windows Installer bei der Installation einen Zähler in der Registry (unter `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\SharedDLLs`). Dieser gibt darüber Aufschluss, von wie vielen Anwendungen diese Datei installiert wurde.

Bei der Deinstallation wird der Referenzzähler um eins dekrementiert. Die Datei wird erst dann deinstalliert, wenn der Referenzzähler den Wert 0 (null) angenommen hat.

► **Hinweis:** Das Flag SharedDllRefCount wirkt ausschließlich auf die Datei, die KeyPath der Komponente ist.

Jetzt brauchen wir nur noch das erstellte Fragment in Visual Studio mit einfügen und einen Verweis auf die ComponentGroup im entsprechenden Feature ablegen:

```
<ComponentGroup Id ="MainFeature">
    <ComponentGroupRef Id="IpwiActiveX" />
</ComponentGroup>
```

Wie wir sehen, können wir eine ComponentGroup auch in einer anderen ComponentGroup referenzieren.

26.3.3 COM-Server per Selbstregistrierung registrieren

In seltenen Fällen erkennt Heat nicht, dass eine übergebene DLL einen COM-Server beinhaltet. In diesen Fällen kann der COM-Server auch über den Selbstregistrierungs-Mechanismus installiert werden. Dies geschieht, in dem man beim File-Element im Attribut SelfRegCost einen Wert angibt. Der angegebene Wert muss ein positiver Wert sein und gibt die Anzahl der zu reservierenden Bytes an.

Durch die Angabe eines Wertes in SelfRegCost wird der File-Key in die **SelfReg**-Tabelle geschrieben. Diese Datei wird dann über die Actions **SelfRegModules** bzw. **SelfUnregModules** registriert bzw. deregistriert:

```
<File Id="fileEB6950C7060D58B580C616676E1E674" KeyPath="yes" SelfRegCost="20"
    Source=".\\SourceDir\ActiveX V1.00\IpwiActiveX.ocx">
```

► **Hinweis:** Von der Selbstregistrierung wird von Microsoft ausdrücklich abgeraten, da evtl. vorhandene Abhängigkeiten zu anderen DLLs den Registrierungsvorgang verhindern kann.

26.3.4 .NET Assembly als COM Interop registrieren

Möchte man ein.NET Assembly in der Win32-Welt benützen, muss man das .NET Assembly als COM-Server registrieren. Diesen Vorgang nennt man **COM Interop**. Ähnlich wie Win32 COM-Server, die mit RegSvr32.exe registriert werden, gibt es beim NET Framework die **RegAsm.exe**, über die man.NET Assemblies per COM Interop registrieren kann. Wir könnten also eine Custom Action erstellen, die bei der Installation RegAsm.exe aufruft. Für die Deinstallation müsste man dann aber eine zweite Custom Action machen, die das .NET Assembly wieder deregistriert. Rollback und der gleichen wäre mit dieser Lösung nicht berücksichtigt.

Aber es gibt einen viel einfacheren Weg. Der Windows Installer und die Heat.exe helfen uns dabei. Zuerst müssen wir mit RegAsm.exe eine Type Library (TLB-Datei) erstellen:

RegAsm.exe InteropAssembly.dll /tlb

Danach rufen wir die Heat.exe zwei Mal auf und erstellen ein Fragment für die DLL und für die Type Library:

*Heat.exe file InteropAssembly.dll /out asm_fragment.wxs
Heat.exe file InteropAssembly.tlb /out tlb_fragment.wxs*

Wenn wir die beiden Fragmente einbinden wird das .NET Assembly als COM-Server registriert.

26.4 IIS Webseite abscannen

Heat.exe kann auch dazu benutzt werden, eine IIS Webseite abzuscanen und als WXS-Datei abzulegen.

Heat.exe website "Default Web Site" -out website.wxs

Die oben angegebene Zeile scannt die Default-Webseite ab und schreibt alle Angaben in die Datei website.wxs.

► **Hinweis:** Um eine Website abzescannen zu können muss die „IIS 6 Management Compatibility“ auf dem Webserver installiert sein.

26.5 Visual Studio Projekt

Die dritte Anwendungsmöglichkeit von Heat ist es, ein Visual Studio Projekt anzugeben. Alle dort referenzierten Dateien würden dadurch automatisch in das Fragment mit aufgenommen. Heat ruft man in diesem Fall wie folgt auf:

heat project <PROJEKTDATEI> -pog:Binaries -cg SampleGroup -out SampleGroup.wxs

27 Weitere Systemeinstellungen

In diesem Kapitel wollen wir ein paar Spezialitäten aus der Windows Installer bzw. der WiX-Welt kennenlernen.

27.1 Einträge in der Systemsteuerung

Der Windows Installer bietet eine ganze Sammlung von Properties, die die Darstellung unserer Software in der **Systemsteuerung**▶**Programme**▶**Programme und Funktionen** verändern können. Hier können wir Anwendungen, Kommentare, Kontaktinformationen, Internetlinks usw. definieren. Alle Property-Namen beginnen mit ARP was *Add or remove program* (die englische Bezeichnung für diesen Ordner).

Folgende Properties stehen zur Verfügung:

Property	Beschreibung
ARPCOMMENTS	Property kann einen beliebigen Kommentar enthalten.
ARPCONTACT	Kontaktdaten zur Firma, welche die Software herstellt bzw. vertreibt.
ARPINSTALLLOCATION	Qualifizierter Pfad, in der die Hauptanwendung installiert ist.
ARPNOMODIFY	Mit dem Wert 1 steht die Schaltfläche <i>Ändern</i> nicht zur Verfügung.
ARPNOREMOVE	Mit dem Wert 1 steht die Schaltfläche <i>Deinstallieren</i> nicht zur Verfügung.
ARPNOREPAIR	Mit dem Wert 1 steht die Schaltfläche <i>Reparieren</i> nicht zur Verfügung.
ARPPRODUCTICON	Property definiert das Icon, mit dem die Software dargestellt wird.
ARPREADME	Property kann einen Link zu einer Readme-Datei enthalten.
ARPSIZE	Geschätzte Größe der Anwendung in Kilobytes.
ARPSYSTEMCOMPONENT	Property verhindert, dass die Software überhaupt in der Softwareliste angezeigt wird.
ARPURLINFOABOUT	URL zur Homepage der Anwendung.

Wir werden nun ein paar Einstellungen für unsere Software vorsehen:

```
<!-- Properties für Add or remove programs -->
<Property Id='ARPCOMMENTS'>WiX Toolset Beispiel</Property>
<Property Id='ARPHELPLINK'>www.sd-technologies.de</Property>
<Property Id='ARPPRODUCTICON'>IpwiSample.exe</Property>
```

Das Property ARPPRODUCTICON zeigt auf das Id-Attribut unseres bereits definierten Icons:

```
<Icon Id='IpwiSample.exe' SourceFile='.\SourceDir\Program V1.0\IpwiSample.exe' />
```

27.2 Umgebungsvariablen definieren

Der Windows Installer unterstützt über die **Environment**-Tabelle die Erstellung von Umgebungsvariablen. Deshalb ist es nicht verwunderlich, dass WiX dies auch unterstützt. Über das Element **Environment** können Umgebungsvariablen erstellt, erweitert und entfernt werden.

Ein typischer Anwendungsfall ist die Erweiterung der Path-Variable. Die Path-Variable enthält durch Semikolon getrennte Suchpfade. Wenn eine Anwendung eine DLL laden will, sucht Windows diese DLL

zuerst im Systemverzeichnis, dann in den Pfaden, die in der Path-Variablen eingetragen sind und erst zuletzt im Verzeichnis, in der die anfordernde Anwendung selbst installiert ist.

Das Environment-Element muss innerhalb einer Komponente definiert werden, die dann bestimmt, wann die Umgebungsvariable gesetzt wird:

```
<Component Id='IpwiSample.exe' Guid='YOURGUID' Directory='INSTALLDIR'>
  ...
  <Environment Id='MyPath' Name='PATH' Action='set' System='yes' Part='last'
    Value='[INSTALLDIR]' />
</Component>
```

Das Attribut **Action** definiert, was gemacht werden soll. Mögliche Werte sind:

Create	Umgebungsvariable erstellen, wenn es sie noch nicht gibt; nichts machen, wenn es sie schon gibt
Set	Umgebungsvariable erstellen, wenn es sie noch nicht gibt; verändern, wenn es sie schon gibt
Remove	Umgebungsvariable bei der Installation löschen

Das **Part**-Attribut regelt die Art und Weise, wie ein neuer Wert gesetzt wird:

All	ersetzt den bestehenden Wert durch den neuen Wert
First	stellt den angegebenen Wert vor den bestehenden Wert der Umgebungsvariablen, wobei der neue und der alte Wert durch ein Semikolon getrennt werden
Last	hängt den angegebenen Wert an den bestehenden Wert der Umgebungsvariablen an, wobei der alte und der neue Wert durch ein Semikolon getrennt werden

Das Attribut **System** definiert, ob die Path-Variable für alle oder nur für den derzeit angemeldeten Benutzer definiert wird. Da unser Setup für alle Benutzer installiert wird, ist es nur konsequent, dass wir die Umgebungsvariable für alle Benutzer ändern.

Als Value geben wir den Wert (die eckigen Klammern bedeuten den Wert der eingeschlossenen Directory-Variablen) von INSTALLDIR an.

► **Hinweis:** Das benutzen der Path-Variable ist etwas problematisch. Umgebungsvariablen können maximal 2.048 Zeichen lang sein. Wenn in der Path-Variablen mehrere Pfade eingetragen sind, kommt man relativ schnell auf diese Größe. Um dies zu umgehen, definiert man besser nur für die Anwendung geltende Pfade, die man unter HKLM\Software\Microsoft\Windows\CurrentVersion\App Paths einträgt.

27.3 Schriftarten installieren

In diesem Abschnitt werden wir zeigen, wie Windows-Schriftarten installiert werden können. Schriftarten werden in der Regel in das Fonts-Verzeichnis unterhalb des Windows-Verzeichnisses installiert. Hierfür bietet der Windows Installer uns ein Property namens **FontsFolder** an. Dieses Property wird wie z. B. SystemFolder beim Starten des Windows Installer Setups gesetzt. Wenn wir also unsere TTF-Datei dorthin installieren, brauchen wir zunächst einen Directory-Eintrag:

```
<DirectoryRef Id="TARGETDIR">
  <Directory Id="FontsFolder" Name="FontsFolder" />
</DirectoryRef>
```

Nun kann eine neue Komponente erstellt werden, welche die zu installierende Datei enthält. Wenn wir, wie in unserem Beispiel, eine TrueType-Schriftart verwenden, dann muss nur das Attribut **TrueType** auf den Wert yes gesetzt werden.

```
<Component Id="Andyb.ttf" Guid="YOURGUID" Directory="FontsFolder">
  <File Id="Andyb.ttf" Name="Andyb.ttf" Source=".\\SourceDir\\Fonts\\Andyb.ttf"
    TrueType="yes" KeyPath="yes"/>
```

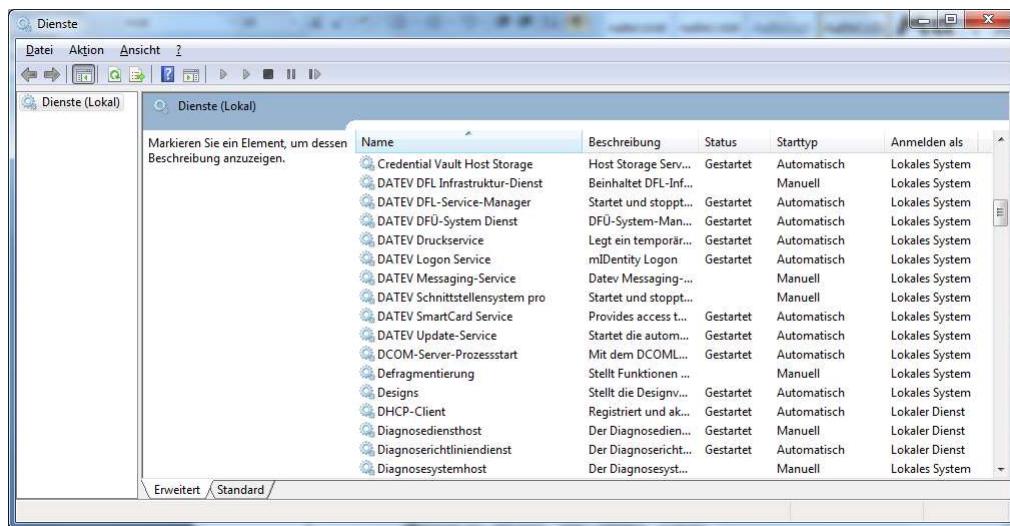
```
</Component>
```

Wenn es keine TrueType-Schriftart ist (z. B. eine FON-Datei), muss zusätzlich der Name der Schriftart über das Attribut **FontTitle** gesetzt werden. Bei TrueType-Schriftarten macht man das nicht, da bei TrueType-Schriftarten der Name im Header der Datei eingetragen ist.

27.4 Dienst installieren

Ein **Dienst** ist eine Anwendung, die im Hintergrund läuft und verschiedene Systemaufgaben übernimmt. Dienste werden von Windows im Hintergrund gestartet und arbeiten auch, wenn kein Benutzer am Rechner angemeldet ist. Damit Windows weiß, welche Dienste mit welchen Benutzerrechten gestartet werden sollen, müssen wir die Dienste entsprechend anmelden.

Alle auf unserem System installierten Dienste können wir über Systemsteuerung ► Verwaltung ► Dienste sehen:



Die Einstellungen des Dienstes können wir uns über einen Doppelklick mit der Maus ansehen. Der Windows Installer und somit auch WiX unterstützen sowohl das Einrichten als auch das Steuern – also das Starten und Stoppen – von Diensten.

Wir werden in unserem Beispiel ein neues Feature erstellen, den Dienst installieren und auch gleich starten. Die Definition des Features sieht dann etwa so aus:

```
<Feature Id="Service" Title="Windows Dienst" Level="101">
    <ComponentGroupRef Id="Service"/>
</Feature>
```

In einer ComponentGroup erstellen wir eine neue Komponente und weisen dieser den Dienst MyService.exe zu:

```
<ComponentGroup Id="Service">
    <Component Id="MyService.exe" Guid="YOURGUID" Directory="INSTALLDIR" >
        <File Id="MyService.exe" Source=".\\SourceDir\\MyService.exe" KeyPath="yes"/>
    </Component>
</ComponentGroup>
```

Das war nichts Neues, das kennen wir bereits. Nun wollen wir den Dienst registrieren. Hierzu gibt es das **ServiceInstall**-Element, das ein Child-Element der Komponente ist:

```
<ServiceInstall Id="MyServiceInstall" Name="WiXService" DisplayName="WiX Dienst"
    Description="WiX Toolset Dienst" Start="auto" Type="ownProcess"
    Vital="yes" ErrorControl="ignore" />
```

Das Attribut **Name** definiert den Namen des Dienstes. Mit diesem Namen wird der Dienst in die Registry geschrieben. Für unseren Anwender sind die Attribute **DisplayName** sowie **Description** interessanter. Diese werden in der Systemsteuerung ► Verwaltung ► Dienste dargestellt. Mit dem Attribut **Start** definieren wir, ob der Dienst automatisch – also beim Starten des Betriebssystems – oder bei Bedarf gestartet wird. Mit **Vital** geben wir an, ob das Setup abgebrochen werden soll, wenn der Dienst nicht installiert werden konnte. **ErrorControl** definiert schließlich, wie Windows reagieren soll, wenn der Dienst bei einem Neustart nicht gestartet werden kann.

27.5 Dienst starten/stoppen

Wenn bei der Installation des Dienstes das Attribut Start auf *auto* gesetzt wird, ist es durchaus angebracht, den Dienst auch gleich mit der Installation zu starten. Selbst wenn das nicht der Fall ist, müssen wir den Dienst bei der Deinstallation anhalten und löschen. Das macht man in WiX über das **ServiceControl**-Element, das ebenfalls ein Child-Element der Komponente ist:

```
<ServiceControl Id="StartService" Name="WiXService" Start="install" Wait="no"/>
<ServiceControl Id="StopService" Name="WiXService" Stop="uninstall" Wait="yes"
    Remove="uninstall" />
```

Mit Wait geben wir an, ob das Setup warten soll, bis die Aktion (starten oder stoppen) durchgeführt worden ist, oder ob das Setup parallel weiterarbeiten soll. Bei der Deinstallation sollten wir auf jeden Fall warten, da wir uns sonst einen Reboot einhandeln könnten.

28 Berechtigungen setzen

Der Windows Installer besitzt von Haus aus Möglichkeiten, Berechtigungen auf Verzeichnisse, Dateien und Registry-Einträge zu setzen. Seit Windows Installer 5.0 können Berechtigungen auch über **SDDL** (Security Descriptor Definition Language) gesetzt werden. Die Windows Installer Funktionen zum Setzen der Berechtigungen setzen die angegebenen Berechtigungen absolut – es bestand also keine Möglichkeit, die vorhandene Berechtigung so zu verändern, dass Rechte zu den bestehenden Berechtigungen hinzugefügt werden. Somit ist das Setzen der MSI-Berechtigungen nur dort sinnvoll, wo keine Rücksicht auf die bestehende Berechtigungsstruktur genommen werden muss.

Soll auf die bestehende Berechtigungsstruktur Rücksicht genommen werden, so hilft uns die WixUtilExtension weiter.

28.1 Setzen der Berechtigung über das Permission-Element

Über das **Permission**-Element können Berechtigungen auf Verzeichnisse absolut gesetzt werden (diese Variante ist für alle Windows Installer Versionen verfügbar). Die Berechtigungen werden hier in die **LockPermission**-Tabelle gesetzt, wobei der zu berechtigende Benutzer bzw. die Benutzergruppe explizit per Namen angegeben wird.

Bei der Angabe des Namens muss man etwas aufpassen: Die Standardgruppen wie *Jeder*, *Hauptbenutzer* oder *Administratoren* heißen in jeder Betriebssystemsprache anders – der Benutzer *Jeder* heißt im englischen Betriebssystem *Everyone*. Der Windows Installer hilft uns hier etwas. MSI definiert zwei sprachunabhängige Benutzernamen: *Everyone* (das ist die Gruppe *Jeder*) und *Administrators* (Gruppe der Administratoren).

Was aber, wenn wir z. B. für die Gruppe *Benutzer* Berechtigungen definieren wollen. Hier hilft uns die WixUtilExtension und die Custom-Action **WixQueryOsWellKnownSID** vom WiX-Toolset weiter. Diese Custom-Action ermittelt mit der sogenannten *Well Known SID* den Namen der Benutzergruppe. Folgende Properties sind definiert: ###

Property-Name	Beschreibung
WIX_ACCOUNT_LOCALSYSTEM	Name des Local-System-Accounts
WIX_ACCOUNT_LOCALSERVICE	Name des Local-Service-Accounts
WIX_ACCOUNT_NETWORKSERVICE	Name des Netzwerk-Service-Accounts
WIX_ACCOUNT_ADMINISTRATORS	Name der Gruppe der Administratoren
WIX_ACCOUNT_USERS	Name des Gruppe der Benutzer
WIX_ACCOUNT_GUESTS	Name des Gastgruppe

Die Custom-Action selbst wird durch das Referenzieren eines dieser Properties eingebunden. Er reicht also folgende Ziel aus, um die Gruppe der Benutzer zu ermitteln:

```
<PropertyRef Id="WIX_ACCOUNT_USERS"/>
```

28.1.1 Verzeichnisberechtigung setzen

Das Permission-Element wird als Child-Element von CreateFolder gesetzt. Im Attribut User gibt man den Benutzernamen an, für den die Berechtigung gesetzt werden soll.

Berechtigungen auf ein Verzeichnis setzt man wie folgt:

```
<Component Id="DirPerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <CreateFolder Directory="INSTALLFOLDER">
    <Permission User="Everyone" CreateChild="yes" CreateFile="yes" DeleteChild="yes"
      Traverse="yes" ReadAttributes="yes" Read="yes" ReadPermission="yes"
      ReadExtendedAttributes="yes" GenericRead="yes" GenericWrite="yes" />
    <Permission User="Administrators" GenericAll="yes"/>
  </CreateFolder>
```

```
</Component>
```

Wie das Beispiel zeigt, setzen wir Berechtigungen auf das Verzeichnis INSTALLFOLDER so, dass jeder Benutzer Schreibrechte bekommt. Die einzigen Rechte, die einem Nichtadministrator verwehrt werden, sind die, das Verzeichnis zu löschen, die Berechtigungen zu setzen sowie den Besitz zu übernehmen. Das überlassen wir dem Administrator bzw. der Gruppe der Administratoren.

Da das Permission-Element die Berechtigungen absolut setzt, sollten wir den Administrator nie vergessen. Das Einzige, was der Windows Installer automatisch setzt ist der Vollzugriff auf das System – sonst könnte der Windows Installer dieses Verzeichnis gar nicht mehr deinstallieren. Soll die Berechtigung auf einen Domänenbenutzer gesetzt werden, so kann die Domäne über das Domain-Attribut angegeben werden. Wird dieses Attribut weggelassen, dann wird die Berechtigung für den lokalen PC gesetzt.

Folgende Rechte können für ein Verzeichnis definiert werden:

Attribut	Beschreibung
GenericAll	Vollzugriff
GenericRead	Ordner auflisten, Attribute lesen, erweiterte Attribute lesen, Berechtigungen lesen
GenericWrite	Dateien erstellen, Ordner erstellen, Attribute schreiben, erweiterte Attribute schreiben, Berechtigungen lesen
GenericExecute	Ordner durchsuchen, Attribute lesen und Berechtigungen lesen
Traverse	Ordner durchsuchen
Read	Ordner auflisten
ReadAttributes	Attribute lesen
ReadExtendedAttributes	Erweiterte Attribute lesen
CreateFile	Dateien erstellen
CreateChild	Ordner erstellen
WriteAttributes	Attribute schreiben
WriteExtendedAttributes	Erweiterte Attribute schreiben
DeleteChild	Unterordner und Dateien löschen
Delete	Löschen
ReadPermission	Berechtigungen lesen
ChangePermission	Berechtigungen ändern
TakeOwnership	Besitz übernehmen

Soll die Gruppe *Benutzer* berechtigt werden, so gibt man das Property WIX_ACCOUNT_USERS im Attribut User in eckigen Klammern an:

```
<PropertyRef Id="WIX_ACCOUNT_USERS"/>

<Component Id="DirPerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <CreateFolder Directory="INSTALLFOLDER">
    <Permission User="[WIX_ACCOUNT_USERS]" GenericRead="yes" GenericWrite="yes" />
    <Permission User="Administrators" GenericAll="yes"/>
  </CreateFolder>
</Component>
```

28.1.2 Dateiberechtigung setzen

Das Permission-Element kann auch als Child vom File-Element definiert werden. Damit werden die Rechte auf die Datei im File-Element gesetzt:

```
<Component Id="FilePerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="Test" Source=".\\Test.txt" KeyPath="yes">
    <Permission User="Everyone" Read="yes" Write="yes" Append="yes"
      ReadAttributes="yes" ReadExtendedAttributes="yes" ReadPermission="yes" />
    <Permission User="Administrators" GenericAll="yes" />
  </File>
</Component>
```

Auch hier sollte man darauf achten, dass man dem Administrator bzw. der Gruppe der Administratoren Vollzugriff gewährt. Wie beim Setzen der Ordnerberechtigung wird das System automatisch als Vollzugriff eingetragen.

Folgende Rechte können für eine Datei definiert werden:

Attribut	Beschreibung
GenericAll	Vollzugriff
GenericRead	Daten lesen, Attribute lesen, erweiterte Attribute lesen und Berechtigungen lesen. Hinweis: GenericRead kann nicht als einziges Recht vergeben werden
GenericWrite	In die Datei schreiben, erweiterte Attribute schreiben und Berechtigungen lesen
GenericExecute	Dateien ausführen, Attribute lesen und Berechtigungen lesen
SpecificRightsAll	Dateien ausführen, Daten schreiben, Daten lesen, Daten anhängen, erweiterte Attribute lesen und erweiterte Attribute schreiben
Synchronize	Keine grundlegenden oder erweiterten Berechtigungen
FileAllRights	Schaltet alle Rechte auf die Datei ein
Execute	Dateien ausführen
Read	Daten lesen
ReadAttributes	Attribute lesen
ReadExtendedAttributes	Erweiterte Attribute lesen
Write	Daten schreiben
Append	Daten anhängen
WriteAttributes	Attribute schreiben
WriteExtendedAttributes	Erweiterte Attribute schreiben
Delete	Löschen
ReadPermission	Berechtigungen lesen
ChangePermission	Berechtigungen ändern
TakeOwnership	Besitz übernehmen

28.1.3 Registry-Berechtigung setzen

Last but not least könnten Berechtigungen auch auf Registry-Einträge gesetzt werden. Das machen wir auch hier über das Permission-Element:

```
<Component Id="RegPerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <RegistryKey Root="HKLM" Key="Software\Sd\Testkey">
    <RegistryValue Name="Path" Value="[INSTALLFOLDER]" Type="string" KeyPath="yes">
      <Permission User="Everyone" GenericRead="yes" GenericWrite="yes" />
      <Permission User="Administrators" GenericAll="yes" />
    </RegistryValue>
    <RegistryValue Name="Path" Value="[INSTALLFOLDER]" Type="string" KeyPath="yes">
    </RegistryValue>
  </RegistryKey>
</Component>
```

Auch wenn das schon bemerkt wurde: Bitte nicht den Administrator vergessen!

Folgende Rechte können für die Registry definiert werden:

Attribut	Beschreibung
GenericAll	Vollzugriff
GenericRead	Daten lesen, Attribute lesen, erweiterte Attribute lesen und Berechtigungen lesen. Hinweis: GenericRead kann nicht als einziges Recht vergeben werden
GenericWrite	In die Datei schreiben, erweiterte Attribute schreiben und Berechtigungen lesen
GenericExecute	Dateien ausführen, Attribute lesen und Berechtigungen lesen
SpecificRightsAll	Dateien ausführen, Daten schreiben, Daten lesen, Daten anhängen, erweiterte Attribute lesen und erweiterte Attribute schreiben
Synchronize	Keine grundlegenden oder erweiterten Berechtigungen
FileAllRights	Schaltet alle Rechte auf die Datei ein
Execute	Dateien ausführen
Read	Daten lesen
ReadAttributes	Attribute lesen
ReadExtendedAttributes	Erweiterte Attribute lesen
Write	Daten schreiben
Append	Daten anhängen
WriteAttributes	Attribute schreiben
WriteExtendedAttributes	Erweiterte Attribute schreiben
Delete	Löschen
ReadPermission	Berechtigungen lesen
ChangePermission	Berechtigungen ändern
TakeOwnership	Besitz übernehmen

► **Hinweis:** Wenn mehrere Werte in die Registry geschrieben werden, sollten wir immer prüfen, ob die Berechtigungen auch tatsächlich übernommen wurden. Im Zweifelsfall muss die Berechtigung für jeden Wert angegeben werden.

28.2 Setzen der Berechtigung über das PermissionEx-Element

Das **PermissionEx**-Element ist die neue Art, Berechtigungen zu setzen. Diese Variante ist ab Windows Installer 5.0 (also ab Windows 7) verfügbar und bietet die Möglichkeit, Berechtigungen mittels der **Security Descriptor Definition Language (SDDL)** zu definieren.

► **Hinweis:** Der Windows Installer lässt es nicht zu, Berechtigungen mit Permission (LockPermission-Tabelle) und PermissionEx (MsiLockPermissionsEx Tabelle) zu setzen. Versucht man das, wirft der Windows Installer den Fehler 1941.

28.2.1 Die Security Descriptor Definition Language (SDDL)

In SDDL gibt man den Security-Descriptor als Zeichenkette an:

`O:ownerG:groupD:dacl_flags(string_ace1)...(string_acen)S:sacl_flags(string_ace1)...(string_acen)`

Der Security-Descriptor besteht aus 4 primären Teilen: Dem **DACL** (Discretionary Access Control List) [**D**], dem **SACL** (System Access Control List) [**S**], der Gruppe [**G**] und dem Besitzer [**O**]. Jeder dieser Teile wird über das Präfix gekennzeichnet. Der Besitzer und die Gruppe werden durch den **SID** (Security Descriptor String) angegeben. SACL und DACL können beliebig viele **ACEs** (Access Control Entries), also Berechtigungsstrings, enthalten.

Die ACEs selbst bestehen aus sechs Feldern die in runden Klammern zusammengefasst und durch Semikolon getrennt werden. Die Felder sind folgendermaßen definiert: ACE-Type (erlauben/verbieten/prüfen), ACE-Flags (Vererbung oder Prüfung), die Berechtigung (Liste der Berechtigungen), Objekttyp (GUID), vererbter Objekttyp (GUID) und die Zielgruppe als SID.

Ein Beispiel eines SDDL-Strings für ein Verzeichnis:

`O:BAG:SYD:(A;CIOI;GA;;SY)(A;CIOI;GA;;;BU)`

Dieser String setzt den Besitzer auf den Administrator (BA=Built-in administrators), und definiert Rechte für das System (SY = System). In der ersten ACL (`A;CIOI;GA;;SY`) wird der Zugriff für die Gruppe *System* erlaubt (A = allow und der letzte Parameter SY = System). Die Berechtigen CIOI geben an, dass alle Unterverzeichnisse (CI) und Dateien (OI) die Berechtigungen erben. GA gibt dann noch die Berechtigung an, wobei GA Generic All bedeutet. Dieselbe Berechtigung wird in der zweiten ACL (`A;CIOI;GA;;;BU`) gesetzt, allerdings hier für die Gruppe *Benutzer* (BA = build-in users).

► **Hinweis:** Über das Tool cacls.exe kann man sich über den Parameter /S die bestehende Berechtigung (z. B. die Dateiberechtigung) im SDDL-Format zurückgeben lassen.

Das Kommando sieht dann so aus: `cacls.exe c:\Test.txt /S`

28.2.2 Verzeichnisberechtigung setzen

Das PermissionEx-Element wird wie das Permission-Element als Child von CreateFolder gesetzt:

```
<Component Id="DirPermEx" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <CreateFolder>
    <PermissionEx Id="Dir" Sddl="O:BAG:SYD:(A;CIOI;GA;;SY)(A;CIOI;GA;;;BA)"/>
  </CreateFolder>
</Component>
```

Das dargestellte Beispiel setzt für das Systemkonto sowie für die Gruppe der Administratoren den Vollzugriff und vererbt die Rechte an alle Dateien und Unterverzeichnisse. Anders als Permission wird das Systemkonto nicht implizit berechtigt. Vergessen wir das, können wir das Verzeichnis u. U. nicht mehr deinstallieren.

28.2.3 Dateiberechtigung setzen

Das PermissionEx-Element kann auch als Child vom File-Element definiert werden. Damit werden die Rechte auf die Datei im File-Element gesetzt:

```
<Component Id="FilePerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="Test" Source=".\\Test.txt" KeyPath="yes">
    <PermissionEx Id="File" Sddl="O:BAG:SYD:(A;;GA;;;SY)(A;;GA;;;BA)"/>
  </File>
</Component>
```

Das dargestellte Beispiel setzt für das Systemkonto sowie für die Gruppe der Administratoren den Vollzugriff. Auch wenn weitere Berechtigungen gesetzt werden, sollte man dem Administrator und dem Systemkonto Vollzugriff gewähren.

28.2.4 Registry-Berechtigung setzen

Hier noch das Vorgehen, wie Berechtigungen auch auf die Registry gesetzt werden:

```
<Component Id="RegPerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <RegistryKey Root="HKLM" Key="Software\\Sd\\Testkey">
    <RegistryValue Name="Path" Value="[INSTALLFOLDER]" Type="string">
      <PermissionEx Id="Reg" Sddl="O:BAG:SYD:(A;CIOI;GA;;;SY)(A;CIOI;GA;;;BU)"/>
    </RegistryValue>
  </RegistryKey>
</Component>
```

28.2.5 Berechtigung für Dienst setzen

PermissionEx kann auch zum Setzen von Dienstberechtigungen verwendet werden. Das PermissionEx-Element wird hierbei als Child vom ServiceInstall-Element abgelegt.

28.3 Berechtigung über PermissionEx von der WixUtilExtension

Im folgenden Abschnitt wollen wir uns noch das **PermissionEx**-Element der **WixUtilExtension** ansehen. Diese Erweiterung ist sofern interessant, als dass man damit auch Berechtigungen additiv setzen kann.

Um die Erweiterung verwenden zu können, fügen wir zunächst einen Verweis auf die Datei WixUtilExtension.dll hinzu und erweitern den Namensraum:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
  xmlns:util="http://schemas.microsoft.com/wix/UtilExtension">
```

28.3.1 Verzeichnisberechtigung setzen

Das PermissionEx-Element wird als Child von CreateFolder gesetzt:

```
<PropertyRef Id="WIX_ACCOUNT_USERS"/>

<Component Id="DirPerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <CreateFolder>
    <util:PermissionEx User="[WIX_ACCOUNT_USERS]" GenericAll="yes" />
  </CreateFolder>
</Component>
```

Anders als bei den Standard-WiX-Lösungen wird die Berechtigung hier den bestehenden Berechtigungen hinzugefügt. Administrator bzw. Systemkonto zu berechtigen, ist hier nicht erforderlich.

28.3.2 Dateiberechtigung setzen

Bei der Dateiberechtigung sieht das Ganze ähnlich aus:

```
<PropertyRef Id="WIX_ACCOUNT_USERS"/>

<Component Id="FilePerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="Test" Source=".\\Test.txt" KeyPath="yes">
    <util:PermissionEx User="[WIX_ACCOUNT_USERS]" GenericAll="yes" />
  </File>
</Component>
```

28.3.3 Registry-Berechtigung setzen

Hier noch das Vorgehen, wie Berechtigungen auch auf die Registry gesetzt werden:

```
<PropertyRef Id="WIX_ACCOUNT_USERS"/>

<Component Id="RegPerm" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <RegistryKey Id="Key" Root="HKLM" Key="Software\\Sd\\Testkey"
    ForceCreateOnInstall="yes">
    <util:PermissionEx User="[WIX_ACCOUNT_USERS]" GenericAll="yes" />
    <RegistryValue Id="Path" Name="Path" Value="[INSTALLFOLDER]" Type="string"/>
  </RegistryKey>
</Component>
```

28.3.4 Berechtigung für Dienst setzen

Über das PermissionEx-Element können auch Berechtigungen für Dienste gesetzt werden.

```
<Component Id="InstallService" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="Service" Source=".\\ MyService.exe" KeyPath="yes"/>
  <ServiceInstall Id="MyService" Type="ownProcess" Name="MyService" Start="demand"
    Account="LocalSystem" ErrorControl="ignore" Interactive="no">
    <util:PermissionEx User="Everyone" GenericAll="yes" ServiceStart="yes"
      ServiceStop="yes" ServiceEnumerateDependents="yes" ServicePauseContinue="yes"
      ServiceQueryConfig="yes" ServiceQueryStatus="yes"/>
  </ServiceInstall>
</Component>
```

Das oben gezeigte Beispiel installiert den Dienst so, dass er von jedem Benutzer gestartet und gestoppt werden kann.

29 Bootstrapping mit Burn

Falls wir Managed Code als Custom-Action verwenden, müssen wir (wie bereits erwähnt) sicherstellen, dass das .NET-Framework auch tatsächlich installiert ist, bevor das Setup gestartet wird. Aber auch wenn es keine Managed-Code-Custom-Action im Setup gibt und wir nur eine .NET-Anwendung installieren, sollte das Setup das Framework mitbringen.

Die erste Idee, die einem hier vielleicht kommen mag ist: Warum installieren wir das .NET-Framework eigentlich nicht per Custom-Action? Das Setup vom .NET-Framework kann man als ausführbare Datei bei Microsoft herunterladen. Dieses könnte man in das Setup integrieren und an entsprechender Stelle installieren.

Die ganze Sache hat aber einen Haken. Die Rollback-Fähigkeit des Windows Installers bedingt, dass zur gleichen Zeit nur ein MSI aktiv sein darf. Aufgrund dieser Einschränkung können keine anderen MSI-Sets in der InstallExecuteSequence gestartet werden (keine ist hier nicht ganz richtig, es gibt eine Custom-Action vom Type 7 „concurrent installations“, davon wird aber von Microsoft – und nicht zuletzt auch von mir – abgeraten). Da das .NET-Framework aber ein MSI ist, bleibt uns dieser Weg verschlossen.

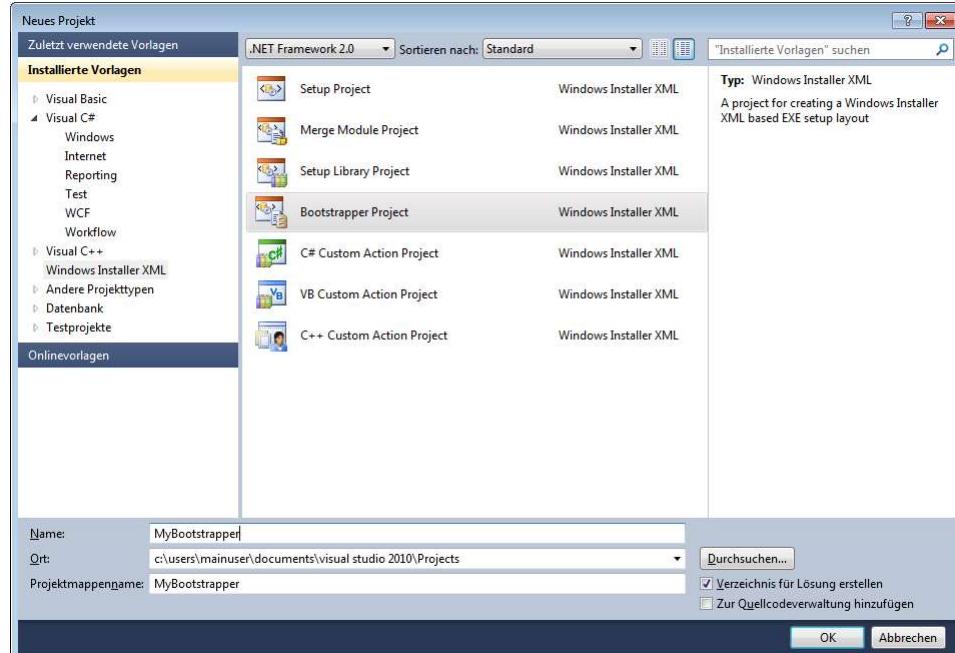
Wenn wir also das .NET Framework mit installieren wollen, dann geht das nur aus dem Userinterface heraus. Da dieses bei einer Silent-Installation nicht läuft, brauchen wir, falls dieser Umstand nicht akzeptabel ist, eine andere Möglichkeit: Der Helfer in der Not heißt: **Bootstrapper**.

Ein Bootsrapper ist eine Art Rahmensetup (z. B. Setup.exe), die vor der Installation der Anwendung die notwendige Laufzeitumgebung installiert. Das muss nicht unbedingt das .NET-Framework sein, es kann auch der Windows Installer selbst oder ein SQL-Server sein.

Der Bootstrapper von WiX heißt **Burn**, der seit WiX 3.6 den alten Microsoft-Bootstrapper ersetzt hat. In Burn beschreiben wir, wer hätte es gedacht, die Installationsreihenfolge der einzelnen Setsups per XML. Wie das geht, sehen wir im nächsten Abschnitt.

29.1 Das Burn-Projekt

Ein Burn-Projekt wird über den Projekttyp Bootstraper Project in den Windows Installer XML-Vorlagen erstellt:



Die Projektvorlage erstellt folgenden Code:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Bundle Name="MyBootstrapper" Version="1.0.0.0" Manufacturer="SD-Technologies"
    UpgradeCode="YOURGUID">
    <BootstrapperApplicationRef Id="WixStandardBootstrapperApplication.RtfLicense"/>
    <Chain>
      <!-- TODO: Define the list of chained packages. -->
      <!-- <MsiPackage SourceFile="path\to\your.msi" /> -->
    </Chain>
  </Bundle>
</Wix>
```

Das Root-Element von Burn wird **Bundle** genannt und in ihm werden die zu installierenden Setup-Pakete im **Chain**-Element definiert.

Burn dient nicht nur dazu, bei der Erstinstallation die Setupvoraussetzungen zu installieren, sondern steuert vielmehr über die komplette Lebenszeit der Installation alle Aktivitäten wie Reparatur und Deinstallation. Burn trägt sich hierfür in die **Systemsteuerung▶Programme▶Programme und Funktionen** ein.

Deshalb ist es nicht verwunderlich, dass wir im Bundle-Element neben den Eigenschaften *Name*, *Version* und *Manufacturer* Eigenschaften wie *HelpTelephone*, *HelpUrl*, *UpdateUrl* und *AboutUrl* finden. Fast selbstverständlich gibt es zum Abschalten der Reparatur bzw. Deinstallation auch entsprechende Eigenschaften wie *DisableModify*, *DisableRepair* und *DisableRemove*.

Normalerweise werden alle Setup-Pakete, die vom Bootstrapper gesteuert werden, in eine einzelne EXE gepackt. Wenn die Pakete extern neben der EXE liegen sollen, dann kann man das mit *Compressed="no"* beim Bundle-Element angeben.

Jedes Bundle benötigt eine sogenannte **Bootstrapper-Applikation**. Eine Bootstrapper-Applikation ist eine DLL, die das Userinterface des Setups darstellt und über das **BootstrapperApplication**- Element definiert wird. Das neu erstellte Projekt bindet gleich zu Beginn die **WixBalExtension** ein, in der eine Bootstrapper-Applikation mit dem Namen WixStandardBootstrapperApplication.RtfLicense definiert ist. Diese wird im Bundle über das **BootstrapperApplicationRef**-Element referenziert. In der WixBalExtension findet man in der Wix-Library etwa folgenden Code:

```
<Fragment>
  <BootstrapperApplication Id="WixStandardBootstrapperApplication.RtfLicense"
    SourceFile="$(var.wixextba.TargetDir)WixStdBA.dll">
    <Payload Name="thm.xml" Compressed="yes" SourceFile=".\\RtfTheme.xml" />
    <Payload Name="thm.wxl" Compressed="yes" SourceFile=".\\RtfTheme.wxl" />
    <Payload Name="Logo.png" Compressed="yes" SourceFile=".\\Logo.png" />
    <Payload Name="!(wix.WixExtbalicenseRtfName=license.rtf)" Compressed="yes"
      SourceFile=".\\LoremIpsumLicense.rtf" />
  </BootstrapperApplication>
</Fragment>
```

► Hinweis: Wie bei Setup-Projekten kann man in Burn auch Lokalisierungsdateien (.WXL Dateien) verwenden. Auch hier werden die einzelnen Strings über *!(loc.STRING_ID)* eingebunden.

29.2 Burn-Variablen

Variablen sind, wie die Properties bei MSI, vor allem für Bedingungen und zur Speicherung bzw. Weitergabe von Benutzereingaben wichtig. Wie bei MSI gibt es bei Burn eine ganze Reihe an **Built-in Variables**, über die wir viel über das Zielbetriebssystem erfahren können.

Anders als bei MSI gibt es bei Burn-Variablen Datentypen. Variablen in Burn können vom Typ String, Numeric oder Version sein.

29.2.1 Built-in Variables

Burn stellt folgende **Burn Built-in Variables** zur Verfügung:

Variablenname	Beschreibung
AdminToolsFolder	Well-known-Ordner CSDL_ADMINTOOLS
AppDataFolder	Well-known-Ordner CSDL_APPDATA
CommonAppDataFolder	Well-known-Ordner CSDL_COMMON_APPDATA
CommonFilesFolder	Well-known-Ordner CSIDL_PROGRAM_FILES_COMMONX86
CommonFiles64Folder	Well-known-Ordner CSDL_PROGRAM_FILES_COMMON
CommonFiles6432Folder	Well-known-Ordner CSDL_PROGRAM_FILES_COMMON bei 64-Bit-Windows und CSIDL_PROGRAM_FILES_COMMONX86 bei 32-Bit- Windows
CompatibilityMode	Ungleich 0, wenn der Bootstrapper im "compatibility mode" gestartet wurde
ComputerName	Name des Computers
Date	Das aktuelle Datum im „short date“-Format
DesktopFolder	Well-known-Ordner CSDL_DESKTOP
FavoritesFolder	Well-known-Ordner CSDL_FAVORITES
FontsFolder	Well-known-Ordner CSDL_FONTS
InstallerName	Gibt den Namen der Installer-Engine ("WiX Burn") zurück.
InstallerVersion	Version der Installer-Engine
LocalAppDataFolder	Well-known-Ordner CSDL_LOCAL_APPDATA
LogonUser	Ermittelt den Namen des angemeldeten Benutzers
MyPicturesFolder	Well-known-Ordner CSDL_MYPICTURES
NTProductType	Numerischer Produkttyp des Betriebssystems
NTSuiteBackOffice	Ungleich 0, wenn das Betriebssystem eine Backoffice- Version ist
NTSuiteDataCenter	Ungleich 0, wenn das Betriebssystem ein Datacenter ist
NTSuiteEnterprise	Ungleich 0, wenn das Betriebssystem eine Enterprise- Version ist

Variablenname	Beschreibung
NTSuitePersonal	Ungleich 0, wenn das Betriebssystem eine Personal Version ist
NTSuiteSmallBusiness	Ungleich 0, wenn das Betriebssystem eine Small-Business-Version ist
NTSuiteSmallBusinessRestricted	Ungleich 0, wenn das Betriebssystem eine beschränkte Small-Business-Version ist
NTSuiteWebServer	Ungleich 0, wenn das Betriebssystem eine Web-Server ist
PersonalFolder	Well-known-Ordner CSDL_PERSONAL
ProcessorArchitecture	Gibt den Wert von SYSTEM_INFO.wProcessorArchitecture zurück
Privileged	Ungleich 0, wenn der Prozess mit erhobenen Rechten arbeiten kann (ab Windows Vista) oder als Administrator gestartet wurde (Windows XP)
ProgramFilesFolder	Well-known-Ordner CSDL_PROGRAM_FILESX86
ProgramFiles64Folder	Well-known-Ordner CSDL_PROGRAM_FILES
ProgramFiles6432Folder	Well-known-Ordner CSDL_PROGRAM_FILES beim 64-Bit-Windows und CSDL_PROGRAM_FILESX86 beim 32-Bit-Windows
ProgramMenuFolder	Well-known-Ordner CSDL_PROGRAMS
RebootPending	Ungleich 0, wenn ein ausstehender Neustart erkannt wurde. Diese Variable reflektiert den Neustartstatus des Systems zum Zeitpunkt des ersten Abfragens der Variable
SendToFolder	Well-known-Ordner CSDL_SENDTO
ServicePackLevel	Numerischer Wert des installierten Service-Packs
StartMenuFolder	Well-known-Ordner CSDL_STARTMENU
StartupFolder	Well-known-Ordner CSDL_STARTUP
SystemFolder	Well-known-Ordner CSDL_SYSTEMX86
SystemLanguageID	Gibt die Sprach-ID des Systems wider
TempFolder	Well-known-Ordner temp location
TemplateFolder	Well-known-Ordner CSDL_TEMPLATES
TerminalServer	Ungleich 0, wenn das System remote als Terminal-Server läuft
UserLanguageID	Sprach-ID des angemeldeten Benutzers
VersionMsi	Versionsnummer des installierten Windows Installers
VersionNT	Version des Betriebssystems. Diese Variable ist vom Datentyp <i>Version</i> . Die Variable hat einen Wert mit der Formatierung v#.#.#. Windows 7 gibt den Wert v6.1 zurück, somit sieht die Abfrage so aus: VersionNT >= v6.1"

Variablenname	Beschreibung
VersionNT64	Version des 64-Bit-Betriebssystems. Bei einem 32-Bit-Betriebssystem ist diese Variable nicht definiert. Diese Variable ist vom Datentyp <i>Version</i> . Die Variable hat einen Wert mit der Formatierung v#.#.#.#. Zum Beispiel gibt Windows 7 einen Wert v6.1 zurück. Eine Abfrage sieht dann in etwa so aus: VersionNT64 >= v6.1"
WindowsFolder	Well-known-Ordner CSDL_WINDOWS
WindowsVolume	Well-known-Ordner the windows volume
WixBundleAction	Enthält den numerischen Wert von enum BOOTSTRAPPER_ACTION (siehe IBootstrapperEngine.h) und wird beim Aufruf von IBootstrapperEngine::Plan() geändert
WixBundleDirectoryLayout	Setzt den Pfad, wohin heruntergeladene Pakete abgelegt werden (siehe –layout-Schalter aus dem Kapitel "Packages vom Internet herunterladen"). Dieser Ordner wird standardmäßig auf den Pfad der Bootstrapper-EXE gelegt. Diese Variable kann auch innerhalb des Bootstrappers geändert werden
WixBundleElevated	Wird auf den Wert 1 gesetzt, wenn das Bundle mit erhobenen Rechten gestartet wurde. Diese Variable kann dazu verwendet werden, auf der Benutzeroberfläche das "elevation shield" auf dem Install-Now-Button zu zeigen
WixBundleExecutePackageCacheFolder	Gibt den absoluten Pfad zum Package-Cache des gestarteten Packages an. Diese Variable ist nur verfügbar, wenn das Package ausgeführt wird
WixBundleForcedRestartPackage	Gibt die ID des Packages, das einen Reboot angefordert hat, zurück
WixBundleInstalled	Gibt an, ob das Bundle bereits installiert ist oder nicht. Dieser Wert wird bei der Initialisierung des Bundles gesetzt
WixBundleLastUsedSource	Gibt den Pfad zur letzten erfolgreichen Pfadermittlung eines Containers bzw. Payloads zurück
WixBundleName	Gibt den Namen des Bundles zurück. Diese Variable kann zur Laufzeit verändert werden
WixBundleManufacturer	Gibt den Hersteller des Bundles zurück (siehe Manufacturer-Attribut vom Bundle)
WixBundleOriginalSource	Gibt den Dateinamen samt Pfad des originalen Bundles zurück
WixBundleOriginalSourceFolder	Gibt den Pfadnamen des originalen Bundles zurück
WixBundleProviderKey	Gibt den Dependency-Provider-Key des Bundles zurück
WixBundleTag	Gibt den Tagnamen (Attribut Tag vom Bundle) zurück
WixBundleVersion	Gibt die Bundle-Version (Attribut-Version vom Bundle) zurück

29.2.2 Eigene Variablen definieren

Über das **Variable**-Element kann man eigene Variablen definieren. Möchte man z. B. eine Seriennummer für die Evaluation-Version vorgeben, dann definiert man diese Variable wie folgt:

```
<Variable Name="SerialNumber" Type="string" Value="A20C-7HC1-111D" Hidden="yes" />
```

Das Attribut **Hidden** definiert, ob der Variablenwert ins Logfile geschrieben wird oder nicht. Dies ist vor allem bei Passwörtern ein nützliches Feature. Das Attribut **Persisted** definiert, ob ein Variablenwert von Burn für den Maintenance-Mode (Reparatur oder Deinstallation) gespeichert werden soll oder nicht.

Hat man die **WixBalExtension** im Projekt eingefügt, dann ist das Attribut **bal:Overridable** verfügbar. Ist dieses Attribut auf **yes** gesetzt, kann die Variable beim Starten von Burn über die Kommandozeile angegeben werden.

Die Variable **SerialNumer** wird beim Starten des Setups wie folgt angegeben:

```
Bootstrapper.exe SerialNumber="ACDD-30CE-X200"
```

► Hinweis: Das **bal:Overridable**-Attribut funktioniert nur dann automatisch, wenn die **WixStdBA** als Userinterface verwendet wird. Erstellt man sein eigenes Userinterface, dann muss man diese Funktionalität selbst erstellen.

29.3 Installationsbedingung

29.3.1 Installationsbedingung über das Bundle definieren

Im **Bundle**-Element wird eine Installationsbedingung über das **Condition**-Attribut angegeben.

```
<Bundle Name="MyBootstrapper" Condition="VersionNT >= v6.1" Version="1.0.0.0" ...>
```

Ist die angegebene Bedingung nicht erfüllt, wird folgender Dialog ausgegeben:



Wie wir sehen, prüfen wir mit der Variable **VersionNT** nicht in der vom Windows Installer bekannten Form, sondern es vielmehr wird eine Version-Variablen in der Form **v#.#.#** zurückgegeben. Burn definiert eine ganze Reihe von **Built-in Variablen**, die in Bedingungen Verwendung finden. Eine komplette Auflistung findet man im Internet unter der URL:

http://wixtoolset.org/documentation/manual/v3/bundle/bundle_builtin_variables.html.

Da der Text der Fehlermeldung nicht änderbar ist, sollte an dieser Stelle nur tatsächlich die Versionsnummer des Betriebssystems geprüft werden. Was aber, wenn wir noch weitere Installationsbedingungen haben? In diesem Fall hilft uns die BAL-Erweiterung **WixBalExtension** weiter.

29.3.2 Installationsbedingung über die BAL-Erweiterung erstellen

Wenn ein Burn-Projekt erstellt wird, bindet das Template automatisch die BAL-Erweiterung ein. Wir müssen also nur das Schema entsprechend anpassen:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
      xmlns:bal="http://schemas.microsoft.com/wix/BalExtension">
```

Nun können wir im Bundle beliebige Installationsbedingungen angeben:

```
<Bundle Name="BurnSample" Version="1.0.0.0"...>
  <bal:Condition Message="!(loc.NO_ADMIN)"><![CDATA[Privileged>0]]></bal:Condition>
</Bundle>
```

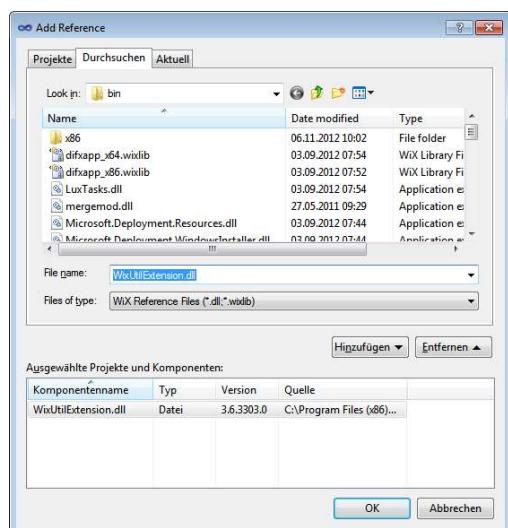
Wird das Setup nun ohne Administrationsrechte gestartet, erscheint folgende Fehlermeldung:



29.3.3 Registry auslesen für die Installationsbedingung

Oft reichen die angebotenen Standardvariablen nicht aus, um festzustellen, ob eine Installation erlaubt ist oder nicht. Wollen wir z. B. ermitteln, ob ein externes Programm wie das .NET-Framework installiert ist und möchten wir dafür eine Installationsbedingung definieren, dann müssen wir Werte aus der Registry lesen. Hier hilft uns die Burn-Extension **WixUtilExtension** weiter.

Um die Extension einzubinden, fügen wir im Burn-Projekt unter References einen neuen Verweis hinzu:



Übrigens: Wenn wir uns die Referenzen genau ansehen, werden wir bemerken, dass bereits die **WixBalExtension** eingebunden ist. Diese erweitert unser Userinterface um den Lizenzvereinbarungsdialog.

Nun erweitern wir das XML-Schema mit einem neuen Namensraum (hier util). Wir tragen in unsere WXS-Datei folgende Zeilen ein:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
      xmlns:util="http://schemas.microsoft.com/wix/UtilExtension"
      xmlns:bal="http://schemas.microsoft.com/wix/BalExtension">
...
</Wix>
```

Nun können wir innerhalb des Bundles einen Registry-Wert mittels **util:RegSearch** auslesen und in eine Variable abspeichern:

```
<Bundle Name="BurnSample" ...>
  <Util:RegistrySearch Id="DotNet4Search" Root="HKLM"
    Key="SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full" Value="Install"
    Variable="DotNet4Found" Format="raw"/>
  <bal:Condition Message="!(loc.NoDotNet)"><![CDATA[DotNet4Found=1]]></bal:Condition>
...
</Bundle>
```

► **Hinweis:** Die WixUtilExtension erlaubt uns nicht nur, die Registry zu lesen, sondern auch Dateien (**FileSearch**) suchen und Verzeichnisse (**DirectorySearch**) oder Komponenten (**ComponentSearch**) zu ermitteln.

29.4 Bundles updaten

Jedes Bundle bekommt beim Kompilieren automatisch eine neue **Bundle-Id**, die nicht verändert werden kann. Zusätzlich zur Bundle-Id geben wir beim Bundle einen **UpgradeCode** an. Wie wir bestimmt schon vermuten, spielt der UpgradeCode beim Update eine entscheidende Rolle.

Burn prüft bei der Installation, ob ein Bundle mit demselben Upgrade-Code und einer anderen Bundle-Id bereits installiert ist. Ist das der Fall, dann prüft Burn die Versionsnummer des bereits installierten Bundles.

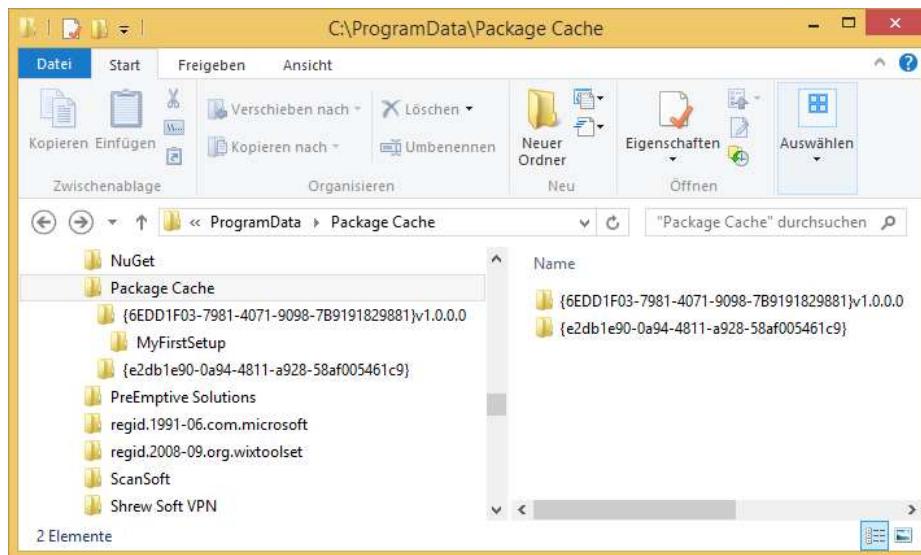
Ist die Versionsnummer gleich, dann wird das zweite Bundle parallel zum ersten installiert. Das kann man sehr einfach sehen, indem man ein Bundle installiert, nochmal kompiliert (dann bekommt das Bundle eine neue Bundle-Id) und dann nochmal installiert. In der *Systemsteuerung▶Programme und Features* sehen wir dann, dass zwei Setups eingetragen sind.

Ist die Versionsnummer des zweiten Bundles größer, dann wird das alte Bundle entfernt und durch das neue Bundle ersetzt. Ist in den Bundles ein MSI-Setup enthalten, prüft Burn bei der Deinstallation, ob im neuen Bundle ein neueres MSI enthalten ist (die Prüfung wird über die Versionsnummer des MSIs gemacht). Ist im neuen Bundle ein neueres MSI enthalten, wird das alte MSI durch das neuere ersetzt. Ist dasselbe MSI-Setup enthalten, wird das MSI nicht deinstalliert.

► **Hinweis:** Über das **RelatedBundle**-Element kann man auch Beziehungen zu anderen Bundles herstellen, die einen anderen UpgradeCode als das aktuelle Bundle haben.

29.5 Der PackageCache von Burn

Im Verzeichnis „%ProgramData%\Package Cache“ (wenn das Setup für die Maschine installiert wurde) bzw. „%AppData%\Package Cache“ (wenn das Setup für den angemeldeten Benutzer installiert wurde) befindet sich der **PackageCache** von Burn:



Den PackageCache braucht man vor allem für den Maintenance-Mode (Reparaturen bzw. Deinstallation). Burn erstellt im PackageCache einen Ordner mit der Bundle-Id. Dort wird die Bootstrapper-EXE abgelegt. Waren im Bundle MSI-Setups enthalten, so speichert Burn diese ebenfalls im PackageCache ab. Der Name des Unterverzeichnisses, in dem die MSI-Setups gespeichert werden, setzt sich aus dem ProductCode und der Versionsnummer des MSI-Setups zusammen.

29.6 Setuppakete einbinden

Die Setuppakete, die vom Bootstrapper installiert werden sollen, werden über das **Chain**-Element definiert. Im Chain-Element kann man MSI-Pakete über das **MsiPackage**-Element definieren, Setups in Form einer EXE über das **ExePackage**-Element, Patches über **MspPackage** und Betriebssystem-Updates in Form von MSU-Dateien über **MsuPackage**.

29.6.1 MSI-Setups einbinden

Möchte man ein MSI-Pakete (z. B. Orca) einbinden, dann sieht das in etwa so aus:

```
<Chain>
  <MsiPackage SourceFile=".\\Source\\Orca.msi" />
</Chain>
```

Manchmal möchte man dem MSI-Setup noch Properties übergeben. Hier hilft das **MsiProperty**-Element weiter, das als Child-Element von MsiPackage eingebunden wird:

```
<MsiPackage SourceFile=".\\Source\\MSI2\\Orca.msi" Compressed="no" >
  <MsiProperty Name="ORCADIRECTORY" Value="C:\\Orca\\"    />
</MsiPackage>
```

Werden mehrere Packages angegeben, so kann man über das *After*-Attribut die Reihenfolge der Installation festlegen:

```
<Chain>
  <MsiPackage Id="Package1" SourceFile=".\\Source\\Package1.msi" />
  <MsiPackage Id="Package2" SourceFile=".\\Source\\Package2.msi" After="Package1"/>
</Chain>
```

Nach der Installation unseres Setups finden wir in **Systemsteuerung ► Programme und Features** zwei Einträge: einen Eintrag vom Bootstrapper und einen Eintrag vom eingebundenen MSI-Setup. Möchte man nicht, dass auch das MSI-Setup dort eingetragen wird, kann man das unterdrücken, indem man das Attribut **Visible** auf *no* setzt.

Wie bereits erwähnt installiert Burn die eingebundenen MSI-Pakete nicht nur auf dem Zielsystem, sondern legt diese auch in den **PackageCache** ab. Dieses Verhalten kann über das Attribut **Cache** geändert werden.

Bei den MSI-Paketen haben wir keine Installationsbedingung angegeben. Da es sich um ein MSI-Paket handelt, kann Burn vor dem Start des Packages selbst prüfen, ob dieses bereits installiert ist oder nicht. Ist das MSI bereits installiert, wird das Package einfach übersprungen. Das Condition-Attribut kommt erst dann ins Spiel, wenn das MSI nur unter einer bestimmten Bedingung installiert werden soll.

Im Registry-Eintrag HKCR\Installer\Dependencies\<ProductCode> merkt sich Burn, welches Package mit welchem Burn-Bundle installiert worden ist. Wird z. B. SetupXY mit zwei oder mehr Bundles installiert, dann wird SetupXY erst deinstalliert, wenn das letzte Bundle vom System genommen wird.

Alle Aktionen, die von Burn ausgeführt werden, werden in einem **LogFile** im Temp-Verzeichnis abgelegt. Der Name des Logfiles ist folgendermaßen aufgebaut: MyBootstrapper_YYYYMMDDhhmmss.log. Wenn SetupXY mit mehreren Packages installiert wurde, dann wird die Deinstallation mit folgenden Zeilen übersprungen:

```
Will not uninstall package: SetupXY.msi, found dependents: 1
Found dependent: {566e703c-187d-4886-83d5-159214822663}, name: MyBootstrapper
```

29.6.2 Eine EXE aus dem Bundle starten

Das **ExePackage**-Element fügt eine EXE-Datei dem Bundle hinzu. Wenn wir z. B. das Java Runtime Environment (JRE) dem Bundle hinzufügen wollen, dann sieht das in etwa so aus:

```
<Chain>
  <ExePackage Id="JreX86" SourceFile=".\\jre-7u45-windows-i586.exe"/>
</Chain>
```

Da es vom JRE eine Version für 32- und 64-Bit gibt, müssen wir entsprechende Fallunterscheidungen hinzufügen:

```
<Chain>
  <ExePackage Id="JreX86" SourceFile=".\\jre-7u45-windows-i586.exe"
    InstallCondition="NOT VersionNT64"/>
  <ExePackage Id="JreX64" SourceFile=".\\jre-7u45-windows-x64.exe"
    InstallCondition="VersionNT64"/>
</Chain>
```

Die Variable **VersionNT64** ist nur dann definiert, wenn es sich um ein 64-Bit Betriebssystem handelt.

```
<Chain>
  <ExePackage Id="JreX86" SourceFile=".\\jre-7u45-windows-i586.exe"
    InstallCommand="/s" UninstallCommand="/s /uninstall" ... />
</Chain>
```

Damit die JRE im Hintergrund installiert wird, geben wir über das Attribut **InstallCommand** den Parameter **/s** an. Da wir das Setup bei der Deinstallation des Bundles auch wieder deinstallieren wollen, geben wir im Attribut **UninstallCommand** den Parameter für die Deinstallation an. Alternativ könnten wir auch das Attribut **Permanent** auf *yes* setzen. In diesem Fall würden die JRE nicht wieder deinstalliert werden.

Doch woran erkennt Burn, ob die JRE bereits installiert ist? Das läuft über das DetectCondition-Attribut. Ist die Bedingung in DetectCondition erfüllt, weiß Burn, dass das Setup bereits installiert ist und die EXE bei der Installation nicht aufgerufen werden muss. Ist die Bedingung in DetectCondition nicht erfüllt, wird die EXE bei der Deinstallation nicht aufgerufen: Was nicht da ist, muss auch nicht deinstalliert werden.

Die Variablen der DetectCondition werden in der Regel durch RegistrySearch- bzw. FileSearch-Elemente der **WixUtilExtension** ermittelt. Die Installation vom JRE sieht dann vollständig etwa so aus:

```
<!-- Check if JRE is installed on the machine -->
<Util:RegistrySearch Root="HKLM" Key="SOFTWARE\JavaSoft\Java Runtime
    Environment\1.7" Result="exists" Variable="JavaInstalled_x64" Win64="yes"/>
<Util:RegistrySearch Root="HKLM" Key="SOFTWARE\JavaSoft\Java Runtime
    Environment\1.7" Result="exists" Variable="JavaInstalled_x86" />

<Chain>
    <!-- Install JRE -->
    <ExePackage Id="JreX86" SourceFile=".\\jre-7u45-windows-i586.exe"
        InstallCommand="/s" UninstallCommand="/s /uninstall"
        InstallCondition="NOT VersionNT64" DetectCondition="JavaInstalled_x86" />
    <ExePackage Id="JreX64" SourceFile=".\\jre-7u45-windows-x64.exe"
        InstallCommand="/s" UninstallCommand="/s /uninstall"
        InstallCondition="VersionNT64" DetectCondition="JavaInstalled_x64"/>
</Chain>
```

► **Hinweis:** Neben dem InstallCommand- und dem UninstallCommand-Attribut gibt es auch noch das RepairCommand. Die EXE wird bei einer Reparatur mit genau diesen Parametern aufgerufen – das natürlich nur, wenn über die DetectCondition erkannt wird, dass die Anwendung installiert ist. Ansonsten würde Burn die Anwendung mit dem InstallCommand installieren.

Viele Setups, die wir als EXE installieren, geben über den Rückgabewert bekannt, ob das Setup erfolgreich installiert wurde und ob ein Neustart des Systems notwendig ist.

Wir können auf Rückgabewerte über entsprechende **ExitCode** Elemente, die als Child-Elemente von ExePackage definiert werden, reagieren:

```
<Chain>
    <!-- Install JRE -->
    <ExePackage ...>
        <ExitCode Value="3010" Behavior="forceReboot"/>
    </ExePackage ...>
</Chain>
```

Die oben dargestellte Anwendung würde einen Reboot durchführen, wenn ein Rückgabewert von 3010 zurückgegeben wird. Weitere Werte von Behavior sind *success*, *error* und *scheduleReboot*.

29.6.3 Patches installieren

Über das **MspPackage** können wir Patches dem Bundle hinzufügen:

```
<Chain>
    ...
    <MspPackage Id="Patch" SourceFile=".\\Patch1.msp"/>
    ...
</Chain>
```

Wie beim MSI erkennt Burn automatisch, ob die zum Patch gehörige Anwendung installiert ist oder nicht. Eine Installationsbedingung muss also nicht zwingend angegeben werden. Doch was macht man, wenn die Anwendung selbst als MSI mit dem Produkt installiert wird und anschließend gleich auf die aktuellste Version hochgepatcht werden soll? Diese Methode nennt man übrigens **Slipstreaming**.

Das kann man mit folgenden Zeilen machen:

```
<Chain>
  <MspPackage Id="Main" SourceFile=".\\MyProduct.msi"/>
  <MspPackage Id="Patch1" SourceFile=".\\Patch1.msp" Slipstream="yes"/>
  <MspPackage Id="Patch2" SourceFile=".\\Patch2.msp" Slipstream="yes"/>
</Chain>
```

Setzen wir das Slipstream-Attribut auf yes, ermittelt Burn beim Komplilieren, zu welchem MSI es gehört, und ordnet das Patch automatisch dem MSI zu. Um diesen Vorgang besser lesbarer zu machen, kann man auch alternativ folgenden Code verwenden:

```
<Chain>
  <MspPackage Id="Main" SourceFile=".\\MyProduct.msi">
    <SlipstreamMsp Id="Patch1"/>
    <SlipstreamMsp Id="Patch2"/>
  </MspPackage>

  <MspPackage Id="Patch1" SourceFile=".\\Patch1.msp" Slipstream="yes"/>
  <MspPackage Id="Patch2" SourceFile=".\\Patch2.msp" Slipstream="yes"/>
</Chain>
```

29.6.4 Betriebssystem-Updates installieren

MSU-Dateien sind Microsoft-Update-Standalone-Installer-Dateien, über die seit Windows Vista das Betriebssystem oder andere Tools, wie z. B. das .NET-Framework, aktualisiert werden können. Möchte man z. B. ein Security-Update vom .NET Framework 3.5 installieren, dann sieht das in etwa so aus:

```
<Chain>
  <MsuPackage SourceFile=".\\Windows6.1-KB2656373-v2-x64.msu" KB="KB2656373"
              DetectCondition="KB2656373Found" />
</Chain>
```

Soll der Patch deinstallierbar sein, gibt man über das KB-Attribut die Knowledge-Base-Artikelnummer an. Eine DetectContion ist bei einer MSU-Datei notwendig, da Burn nicht erkennen kann, ob das Update bereits installiert ist oder nicht.

29.7 Packages vom Internet herunterladen

Burn bietet die Möglichkeit, alle Package-Typen (MSI, MSP, EXE und MSU) von einer bekannten URL herunterzuladen. Unser Setup wird also kleiner, da diese Packages nicht mitgeliefert werden müssen. Da das .NET Framework 4.0 bereits von den neuen Betriebssystemen mitgeliefert wird und in den wenigsten Fällen noch installiert werden muss, könnte man mit dem Gedanken spielen, dieses Setup nicht mitzuliefern, sondern vom Internet zu beziehen.

Wenn eine EXE vom Internet geladen werden soll, geben wir kein SourceFile an, sondern definieren die Attribute *Name* und *DownloadUrl*. Das Attribut *Compressed* wird auf *no* gesetzt, da ja nichts im Bootstrapper-Setup eingebunden werden soll. Hier ein Beispiel wie man den SQL-Server angeben würde:

```
<ExePackage Id="SQLSERVER" DownloadUrl="$(var.SqlDownloadUrl)" Compressed="no"
            Name="SQLEXPR_x64_ENU.exe" DetectCondition="SqlInstanceFound"
            InstallCommand="$(var.SqlInstallCmd)" UninstallCommand="$(var.SqlUninstallCmd)"
            RepairCommand="$(var.SqlRepairCmd)">
  <RemotePayload Description="Microsoft SQL Server 2012 Express Edition"
                  ProductName="Microsoft SQL Server 2012 Express Edition" Version="11.0.2100.60"
                  Size="138412032" Hash="e4561d5caa761a5d1daa0d305f4fecedc6a0d39c" />
</ExePackage>
```

Das **RemotePayload**- Attribut ist eine Checksummenprüfung. Es wird geprüft, ob das, was heruntergeladen wurde, auch dem gewünschten Package entspricht. Der Hash-Wert ist ein SHA1-Hash und kann über das Microsoft-Kommandozeilen-Tool fciv.exe (File Checksum Integrity Verifier) (siehe <http://www.microsoft.com/en-us/download/details.aspx?id=11533>) ermittelt werden. Die Checksumme muss bei einer EXE nur dann angegeben werden, wenn die EXE unsigniert ist oder die Checksummenprüfung über das Attribut *SuppressSignatureVerification="no"* ausgeschaltet wurde.

Das Ganze wird noch einfacher, wenn wir ein MSI als Download-Package einbinden:

```
<MsiPackage Id="PYTHON" SourceFile="python-2.7.3.msi" Compressed="no"
DownloadUrl="http://www.python.org/ftp/python/2.7.3/python-2.7.3.msi" />
```

Bei einem MsiPackage geben wir das MSI über SourceFile an. In diesem Fall muss das MSI zum Zeitpunkt des Bauens lokal vorhanden sein. Aus dem MSI werden nämlich beim Erstellen viele Informationen wie ProductCode und Versionsnummer entnommen. Lediglich die Attribute Compressed und DownloadUrl definieren, dass es sich hier um ein Download-Package handelt.

► **Hinweis:** Ruft man das Burn-Setup mit der Kommandozeilenoption /layout auf, lädt Burn alle Pakete vom Internet herunter und erstellt ein Offline-Package. Dieses Offline-Package kann dann auf Computern ausgeführt werden, die keine Internetverbindung besitzen.

29.8 RollbackBoundary

Wir haben im vorherigen Abschnitt alle zur Installation vorgesehenen Elemente betrachtet. Stellen wir uns vor, wir hätten drei MSI-Setups eingebunden. Die ersten zwei MSIs werden erfolgreich installiert, beim dritten MSI kommt es zu einem Fehler. Da standardmäßig alle MSIs in einer Transaktion über **Chaining** installiert werden, bewirkt der Fehler des dritten MSIs, dass ein Rollback über alle MSIs durchgeführt wird.

Dieses Verhalten kann man durch das Element **RollbackBoundary** ändern. Möchte man die ersten zwei MSI nicht zurückrollen, dann können wir über Rollback-Boundarys definieren, dass zuerst die ersten zwei MSI in einer Transaktion installiert und abgeschlossen werden und erst danach das dritte MSI installiert wird:

```
<Chain>
  <MsiPackage .../>
  <MsiPackage ...="" />

  <RollbackBoundary>
    <MsiPackage ...="" />
  </RollbackBoundary>
</Chain>
```

Bei einem Fehler wird nur das dritte MSI zurückgerollt, die zwei ersten MSI bleiben auf den Rechner. Und: Behandelt Burn das gesamte Bundle als fehlerhafte oder erfolgreiche Installation, wenn das dritte MSI durch einen Fehler abgebrochen wurde?

Wenn wir das Bundle wie oben definieren, dann würde Burn das Bundle als nicht erfolgreich werten. Dieses Verhalten können wir ändern, wenn wir bei der Rollback-Boundary das *Vital* Attribut auf *no* setzen:

```
<Chain>
  <MsiPackage .../>
  <MsiPackage ...="" />

  <RollbackBoundary Vital="no">
    <MsiPackage ...="" />
  </RollbackBoundary>
</Chain>
```

Ein weiterer Anwendungsfall kann z. B. so aussehen:

```
<Chain>
  <MsiPackage .../>"

  <RollbackBoundary Vital="no"/>
  <MsiPackage ...="" />"

  <RollbackBoundary />
  <MsiPackage ...="" />"

</Chain>
```

Wirft das zweite MSI in diesem Beispiel einen Fehler, wird es zurückgerollt und das Setup wird bei der nächsten Boundary weiter ausgeführt. Sind hinter dem fehlerhaften MSI noch weitere MSI eingebunden, die sich in derselben Boundary befinden, dann werden diese MSI übersprungen.

► **Hinweis:** Das MSIpackage-Element hat auch ein Vital-Attribut. Wird dieses auf *no* gesetzt, wird bei einem Fehler dieses MSI zurückgerollt und beim nächsten Package weiterinstalliert. Der Unterschied bei der RollbackBoundary ist, dass bei einem Fehler zur nächsten Boundary und nicht zum nächsten Package gesprungen wird. Zusätzlich werden erfolgreich installierte MSI, die zur selben Boundary gehören, ebenfalls zurückgerollt.

29.9 Die Setupstruktur organisieren

Die Struktur unseres Setups kann über mehrere Parameter maßgeblich beeinflusst werden. Lassen wir beim Bundle das Attribut *Compressed* auf dem Wert *yes*, wird im Ausgabeverzeichnis unseres Setups eine selbst extrahierende EXE erstellt, die nach dem Starten auf dem Zielsystem entpackt und installiert wird.

Soll das Setup über ein Speichermedium wie eine CD-ROM verteilt werden, möchte man dem Anwender unter Umständen noch zusätzliche Informationen in Form von Readme-Dateien o. Ä. mitgeben. In diesem Fall setzen wir das Bundle-Attribut *Compressed* auf den Wert *no*. Die Zusatzdateien, die aus Sicht von Burn nichts mit dem Setup zu tun haben, werden innerhalb des Bundles in sogenannten **PayloadGroups** verwaltet. Innerhalb der PayloadGroup werden die Dateien dann über **Payload**-Elemente angegeben:

```
<PayloadGroup Id="ExternalFiles">
  <Payload Id="Readme.txt_Common" SourceFile=".\\SourceDir\\Readme.txt"/>
</PayloadGroup>
```

Soll die Datei nicht im Root-Verzeichnis bei der Bootstrapper-EXE liegen, dann kann das über das Attribut *Name* angegeben werden. Im unteren Beispiel wird ein Unterverzeichnis namens *Documentation* erstellt in den die Readme-Datei abgelegt wird:

```
<PayloadGroup Id="ExternalFiles">
  <Payload Id="Readme.txt_Common" SourceFile=".\\SourceDir\\Readme.txt"
    Name="Documentation\\Readme.txt"/>
</PayloadGroup>
```

Das Payload-Element kann auch als Child-Element von MsiPackage, ExePackage, MspPackage, und MsuPackage angelegt werden. Beim MsiPackage könnte das z. B. eine Sprachtransformation sein, die bei der Installation des MSI berücksichtigt werden soll.

29.10 Das PackageGroup-Element

Package-Groups helfen uns, den Code etwas übersichtlicher zu halten. Package-Groups werden in Fragmenten definiert und enthalten EXE-, MSI-, MSP- oder MSU-Elemente:

```
<Fragment>
  <PackageGroup Id="SQL-Server">
    <ExePackage .../>
    <ExePackage .../>
  </PackageGroup>
</Fragment>
```

Die Package-Groups werden dann innerhalb des Chain-Elements referenziert:

```
<Chain>
  ...
  <PackageGroupRef Id="Sql-Server"/>
  ...
</Chain>
```

► **Hinweis:** Die **WixNetFxExtension** enthält mehrere Package-Groups, über die unterschiedliche Versionen des .NET-Frameworks installiert werden können. Um das .NET Framework 4 einzubinden, muss man die PackageGroup NetFx4Web referenzieren.

30 Burn-Userinterface mit C# erstellen

Das Layout des Standard-Userinterfaces **WixStdBA** der BAL-Erweiterung lässt sehr schnell den Wunsch nach einer etwas schöneren und individuell gestalteten Oberfläche aufkommen. Das WiX-Toolset liefert im SDK-Verzeichnis das Assembly **BootstrapperCode.dll** mit.

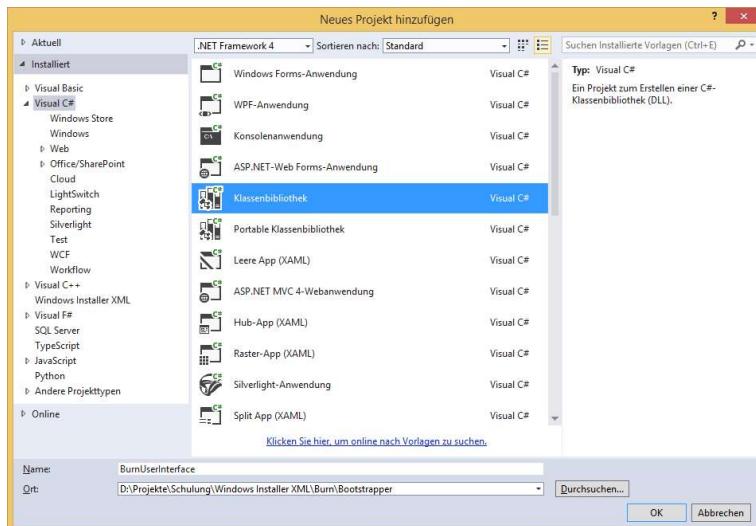
Diese Bibliothek liefert alles, was nötig ist, um ein neues Benutzerinterface in den Installationsablauf einzuhängen. Das Assembly enthält eine Klasse namens **BootstrapperApplication**. Wir leiten unsere neue Klasse von dieser Klasse ab und können dann mit unserem eigenen Code Events abfangen und Methoden überladen.

Grundsätzlich ist es möglich, die **Userinterface-Erweiterung** mit C++, C# Forms oder C# **WPF** (**Windows Presentation Foundation**) zu gestalten. Da WPF die modernste Art ist, in C# Oberflächen zu gestalten, werden wir diese Form hier erklären.

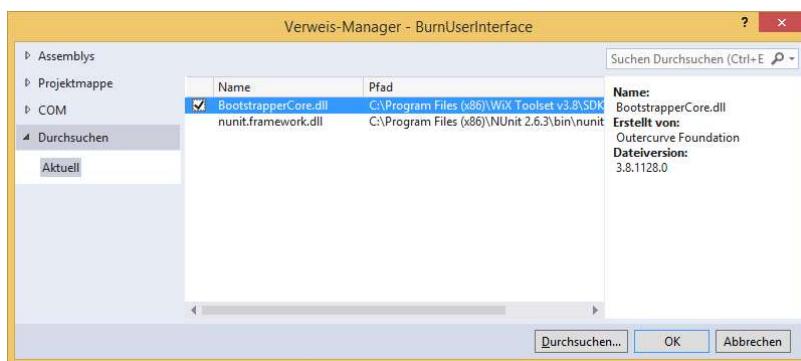
► Hinweis: Ein schönes Beispiel findet man auch im Source-Code des WiX-Toolsets im Verzeichnis `.\src\Setup\WixBA\`.

30.1 Benutzerinterface-Extension für Burn erstellen

Um eine Benutzerinterface-Extension für Burn zu entwickeln, erstellen wir zunächst ein neues Projekt. Dazu verwenden wir die Projektvorlage Klassenbibliothek:



Damit wir auf die BootstrapperCore.dll zugreifen können, fügen wir diese gleich als Verweis hinzu (sie befindet sich im SDK-Verzeichnis des WiX-Toolsets) und binden den Namensraum `Microsoft.Tools.WindowsInstallerXml.Bootstrapper` in unserer neu erstellen Hauptklasse ein:



Nun können wir unsere neue Klasse von BootstrapperApplication ableiten:

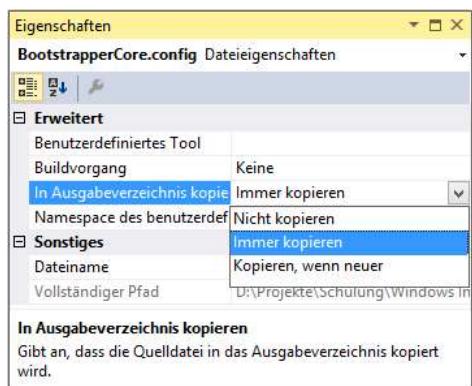
```
using System;
using Microsoft.Tools.WindowsInstallerXml.Bootstrapper;

namespace BurnUserInterface
{
    public class CustomBootstrapperApplication : BootstrapperApplication
    {
        protected override void Run()
        {
            // Hier ist der Einstiegspunkt
        }
    }
}
```

In unserer neuen Klasse überladen wir die Methode Run. Diese Methode stellt den Einstiegspunkt des Bootstrappers dar. Damit Burn weiß, welche Klasse aufgerufen werden muss, tragen wir in die Datei \Properties\AssemblyInfo.cs folgende Zeile am Ende ein:

```
[assembly:BootstrapperApplication(typeof(CustomBootstrapperApplication))]
```

Nun brauchen wir noch eine Datei namens BootstrapperCore.config. Diese Datei sagt Burn, mit welcher Version des .NET-Frameworks die Extension arbeitet und in welchem Namensraum die überschriebene Klasse CustomBootstrapperApplication zu finden ist. Eine Vorlage für diese Datei finden wir im WiX-Toolset-SDK-Verzeichnis. Wenn wir diese Datei in Visual Studio eingefügt haben, setzen wir die Eigenschaften der Datei „In Ausgabeverzeichnis kopieren“ auf „Immer kopieren“.



Damit wird die Datei BootstrapperCore.config beim Build-Vorgang in das Ausgabeverzeichnis kopiert.

Die Datei BootstrapperCore.config ändern wir wie folgt ab:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="wix.bootstrapper"
            type="Microsoft.Tools.WindowsInstallerXml.Bootstrapper.BootstrapperSectionGroup,
            BootstrapperCore">
            <section name="host"
                type="Microsoft.Tools.WindowsInstallerXml.Bootstrapper.HostSection,
                BootstrapperCore" />
        </sectionGroup>
    </configSections>
    <startup useLegacyV2RuntimeActivationPolicy="true">
        <supportedRuntime version="v4.0" />
    </startup>
    <wix.bootstrapper>
        <host assemblyName="BurnUserInterface" />
    </wix.bootstrapper>
</configuration>
```

30.2 Das Benutzerinterface im Bundle referenzieren

Nun ist es an der Zeit, die neu erstellte BurnUserInterface.dll im Bundle zu referenzieren. Damit wir auf die Projektvariablen des Benutzerinterfaces zugreifen können, tragen wir eine Referenz auf das BurnUserInterface im Burn-Projekt ein. Das Element **BootstrapperApplicationRef** bekommt die neue ID *ManagedBootstrapperApplicationHost*. An dieser ID erkennt Burn oder besser gesagt die BAL-Erweiterung, dass ein Benutzerinterface als Managed-Code geladen werden soll und fügt die BootstrapperCode.dll Library automatisch als Payload hinzu. Wir müssen nun noch alle für das Interface notwendigen DLLs sowie die Datei BootstrapperCore.config per Payload hinzufügen:

```
<BootstrapperApplicationRef Id="ManagedBootstrapperApplicationHost">
  <Payload SourceFile="$(var.BurnUserInterface.TargetPath)"/>
  <Payload SourceFile="$(var.BurnUserInterface.TargetDir)\BootstrapperCore.config"/>
</BootstrapperApplicationRef>
```

Nun wird das neue Interface beim Starten des Setups automatisch angetriggert.

30.3 Grundlegende Strukturen erstellen

Da wir das Model-View-View-Model (MVVM)-Pattern verwenden, über das die Präsentations- und die Businesslogik sauber getrennt werden, müssen wir eine Klasse für die View, eine für das Model und eine verbindende Klasse View-Model erstellen. Diese instanziieren wir direkt in der Run-Methode unserer neuen Basisklasse.

In der Run-Methode erstellen wir zunächst ein neues Dispatcher-Objekt. Dieses Objekt empfängt und sendet Nachrichten zwischen dem Interface-Thread und dem Burn-Backend-Thread. Die Dispatcher-Klasse ist im Namensraum *System.Windows.Threading* zuhause und in der Klassenbibliothek WindowsBase.dll enthalten.

```
Dispatcher = Dispatcher.CurrentDispatcher;
```

Da sich das Userinterface bei Erst- und Maintenance-Installation unterscheidet, instanziieren wir hier für den jeweiligen Fall die richtige View. Das Model und die ModelView-Objekte sind so allgemein aufgebaut, dass wir für beide Fälle dieselben Objekte benutzen können (mir ist bewusst, dass das verwenden des selben ModelView-Objekts für mehrere unterschiedliche Views nicht ganz im Sinne von MVVM ist – der Einfachheit halber soll mir diese Sünde vergeben sein).

Wir erstellen folgende Klassen: BurnUiModel, BurnUiViewModel, InstallWelcomeView und MaintlWelcomeView. Ob wir uns im Erstinstallations- oder im Mainteance-Mode befinden, können wir über die Variable **WixBundleInstalled** herausfinden. Da die Klassen InstallWelcomeView und MaintlWelcomeView von der Klasse Window abgeleitet sind, erstellen wir eine Variable von diesem Typ und weisen ihr bei der Instanziierung die entsprechende Klasse zu:

```
Window view;

Dispatcher = Dispatcher.CurrentDispatcher;

Debugger.Launch();

var model = new BurnUiModel(this);
var viewModel = new BurnUiViewModel(model);
var Installed = this.Engine.StringVariables["WixBundleInstalled"];

// Check if first installation or maintenance mode
if (this.Engine.StringVariables["WixBundleInstalled"] == "0")
{
    view = new InstallWelcomeView(viewModel);
}
else
{
    view = new MaintlWelcomeView(viewModel);
}
```

Nun übergeben wir dem Model das Fensterhandle der View. Dieser Vorgang ist eigentlich gegen die Idee der Trennung zwischen Präsentations- und Businesslogik, aber es ist ein notwendiges Übel der BootstrapperCore Library:

```
model.SetWindowHandle(view);
```

Nun können wir den Ball ins Rollen bringen, indem wir die Detect-Methode der Basisklasse aufrufen. Die Detect-Methode prüft nun alle im Bundle enthaltenen Packages daraufhin, ob diese bereits installiert sind oder nicht. Hierzu werden mehrere Events abgefeuert, die wir abfangen können:

```
this.Engine.Detect();
```

Nun können wir die View darstellen und den Dispatcher starten. Der Dispatcher dreht nun so lange seine Runden bis die Methode InvokeShutdown des Dispatcher-Objekts aufgerufen wird. Das Model enthält eine Variable, die dem Bootstrapper sagt, ob die Installation erfolgreich war oder nicht. Dieser Wert wird dann über die Engine.Quit-Methode an den Bootstrapper zurückgegeben:

```
view.Show();
Dispatcher.Run();
this.Engine.Quit(model.FinalResult);
```

Hier noch einmal der gesamte Code im Überblick:

```
using System.Windows;
using System.Windows.Threading;
using Microsoft.Tools.WindowsInstallerXml.Bootstrapper;
using BurnUserInterface.Views;

namespace BurnUserInterface
{
    public class CustomBootstrapperApplication : BootstrapperApplication
    {
        public static Dispatcher Dispatcher { get; set; }

        protected override void Run()
        {
            Window view;

            Dispatcher = Dispatcher.CurrentDispatcher;

            var model = new BurnUiModel(this);
            var viewModel = new BurnUiViewModel(model);
            var Installed = this.Engine.StringVariables["WixBundleInstalled"];

            // Check if first installation or maintenance mode
            if (this.Engine.StringVariables["WixBundleInstalled"] == "0")
                view = new InstallWelcomeView(viewModel);
            else
                view = new MaintlWelcomeView(viewModel);

            model.SetWindowHandle(view);

            this.Engine.Detect();

            view.Show();

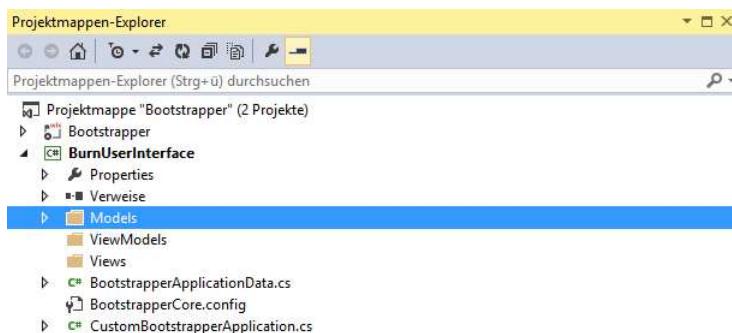
            Dispatcher.Run();

            this.Engine.Quit(model.FinalResult);
        }
    }
}
```

30.4 Die Model-Klasse

30.4.1 Die Model-Klasse erstellen

Da die **Model-Klasse** nur Events der Oberfläche von und zum Bootstrapper durchreicht, können wir diese Klasse für den Erstinstallations- und Maintenance-Mode allgemein halten. Dazu erstellen wir eine neue Model-Klasse. In WPF ist es üblich, für die Model, die Views und die ModelViews einen separaten Ordner zu erstellen:



Die Model-Klasse wird dann im Ordner Models erstellt:

```
using System;
using System.Windows;
using System.Windows.Interop;
using Microsoft.Tools.WindowsInstallerXml.Bootstrapper;

namespace BurnUserInterface
{
    public class BurnUiModel
    {
    }
}
```

30.4.2 Der Konstruktor

Der Konstruktor der Klasse erwartet ein Objekt vom Typ BootstrapperApplication. Wie wir oben gesehen haben, übergeben wir hier die Instanz unserer CustomBootstrapperApplication-Klasse. Da wir für unsere ApplyAction-Methode einen Pointer auf unser WPF-Fenster brauchen und dieser erst mit dem Aufruf von SetWindowHandle erzeugt wird, initialisieren wir diesen zunächst mit einem Null-Pointer:

```
private IntPtr hwnd;
public BootstrapperApplication BootstrapperApplication { get; private set; }

public BurnUiModel(BootstrapperApplication bootstrapperApplication)
{
    this.BootstrapperApplication = bootstrapperApplication;
    this(hwnd = IntPtr.Zero;
}
```

30.4.3 Verbindungsmethoden

Die Funktion SetWindowHandle ist sehr übersichtlich und speichert nur den Zeiger auf die View:

```
public void SetWindowHandle(Window view)
{
    this(hwnd = new WindowInteropHelper(view).Handle;
}
```

Den Auftrag zur Installation bzw. Deinstallation geben wir Burn in zwei Schritten. Zuerst rufen wir die **PlanAction**-Funktion auf und sagen damit dem Bootstrapper, was gemacht werden soll. Ist der Bootstrapper fertig mit der Planung, dann feuert er das Event **PlanComplete** ab. Dieses Event fangen wir ab und rufen die Funktion **ApplyAction** aus, welche die eigentliche Installation durchführt. In unserer Model-Klasse definieren wir dazu zwei Methoden, die den Aufruf einfach an Burn weiterreichen:

```
public void PlanAction(LaunchAction action)
{
    this.BootstrapperApplication.Engine.Plan(action);
}

public void ApplyAction()
{
    this.BootstrapperApplication.Engine.Apply(this.hwnd);
}
```

Funktionen zum Schreiben in das Logfile bzw. zum Schreiben und Lesen von Burn-Variablen schließen die Methoden ab:

```
public void LogMessage(string message)
{
    this.BootstrapperApplication.Engine.Log(LogLevel.Standard, message);
}

public void SetBurnVariable(string variableName, string value)
{
    this.BootstrapperApplication.Engine.StringVariables[variableName] = value;
}

public string GetBurnVariable(string variableName)
{
    return this.BootstrapperApplication.Engine.StringVariables[variableName];
}
```

30.4.4 Code der BurnUiModel-Klasse

Hier noch einmal die gesamte Klasse als Übersicht:

```

using System;
using System.Windows;
using System.Windows.Interop;
using Microsoft.Tools.WindowsInstallerXml.Bootstrapper;

namespace BurnUserInterface
{
    public class BurnUiModel
    {
        private IntPtr hwnd;
        public BootstrapperApplication BootstrapperApplication { get; private set; }
        public int FinalResult { get; set; }

        public BurnUiModel(BootstrapperApplication bootstrapperApplication)
        {
            this.BootstrapperApplication = bootstrapperApplication;
            this(hwnd = IntPtr.Zero;
        }

        public void SetWindowHandle(Window view)
        {
            this(hwnd = new WindowInteropHelper(view).Handle;
        }

        public void PlanAction(LaunchAction action)
        {
            this.BootstrapperApplication.Engine.Plan(action);
        }

        public void ApplyAction()
        {
            this.BootstrapperApplication.Engine.Apply(this(hwnd));
        }

        public void LogMessage(string message)
        {
            this.BootstrapperApplication.Engine.Log(LogLevel.Standard, message);
        }

        public void SetBurnVariable(string name, string value)
        {
            this.BootstrapperApplication.Engine.StringVariables[name] = value;
        }

        public string GetBurnVariable(string name)
        {
            return this.BootstrapperApplication.Engine.StringVariables[name];
        }
    }
}

```

30.5 Die ViewModel-Klasse

30.5.1 Hilfreiche Libraries einbinden

Nun geht es weiter bei der ViewModel-Klasse. Damit wir es uns beim Verweisen auf Ereignismethoden (dem Delegate) etwas einfacher tun, binden wir die Microsoft **Prism Library for WPF** hinzu. Die Prism Library kann von folgender Webseite heruntergeladen werden:

<http://compositewpf.codeplex.com>

Alternativ kann Prism auch über den NuGet-Package-Manager bezogen werden. Von Prism brauchen wir das Assembly Microsoft.Practices.Prism.dll, das wir als Referenz in unser Projekt mit aufnehmen. Dieses Assembly müssen wir dann auch noch als Payload zu unserem Bundle hinzufügen. Wir gehen also in die WXS-Datei zum Bundle und erweitern unser Payload mit folgenden Zeilen:

```
<BootstrapperApplicationRef Id="ManagedBootstrapperApplicationHost">
  <Payload SourceFile="$(var.BurnUserInterface.TargetPath)"/>
  <Payload SourceFile="$(var.BurnUserInterface.TargetDir)\BootstrapperCore.config"/>
  <Payload SourceFile="$(var.BurnUserInterface.TargetDir)\Microsoft.Practices.Prism.dll" />
</BootstrapperApplicationRef>
```

30.5.2 Die ViewModel-Klasse erstellen

Jetzt ist es an der Zeit die ViewModel-Klasse im ViewModel-Ordner zu erstellen. Dort erweitern wir den Namensraum mittels Using auf die Prism-Library-Klassen:

```
using Microsoft.Practices.Prism.Commands;
using Microsoft.Practices.Prism.ViewModel;
using Microsoft.Tools.WindowsInstallerXml.Bootstrapper;
using System;
using System.Windows.Input;

namespace BurnUserInterface
{
    public class BurnUiViewModel : NotificationObject
    {
    }
}
```

30.5.3 Deklaration der Variablen

Wie wir in der Klassendefinition sehen, ist unsere ViewModel-Klasse von der Prism-Klasse **NotificationObject** abgeleitet. Diese hilft uns, die View anzupassen, wenn eine Eigenschaft im ViewModel geändert wurde. In der Klasse definieren wir ein Enum namens InstallState:

```
public enum InstallState
{
    Initializing,
    Present,
    NotPresent,
    Applying,
    Cancelled
}
```

Diese Variable hilft uns herauszufinden, in welcher Phase des Bootstrappings wir uns gerade befinden. Die folgenden Variablen vom Typ ICommand helfen uns, das Installations-, Reparatur-, Deinstallations- und Abbruchkommando an den Bootstrapper zu senden:

```
public ICommand InstallCommand { get; private set; }
public ICommand RepairCommand { get; private set; }
public ICommand UninstallCommand { get; private set; }
public ICommand CancelCommand { get; private set; }
```

Als Nächstes erstellen wir zwei weitere Variablen: Message und State. Diese Variablen werden mit einem Control auf der Oberfläche verbunden welches jedes Mal, wenn sich der Wert der Variable ändert, über die RaisePropertyChanged Methode einen Refresh der View aufruft.

Die InstallState-Variable ruft auch die Refresh-Methode auf. So kann ein Control abhängig vom Installationsstatus seinen eigenen Status von sichtbar auf unsichtbar bzw. von enabled auf disabled ändern:

```
private string message;
public string Message
{
    get
    {
        return this.message;
    }
    set
    {
        if (this.message != value)
        {
            this.message = value;
            this.RaisePropertyChanged(() => this.Message);
        }
    }
}

private InstallState state;
public InstallState State
{
    get
    {
        return this.state;
    }
    set
    {
        if (this.state != value)
        {
            this.state = value;
            this.Message = this.state.ToString();
            this.RaisePropertyChanged(() => this.State);
        }
    }
}
```

30.5.4 Der Konstruktor

Der Konstruktor bekommt als einzigen Parameter BurnUiModel, also unser allgemeingültiges Model, überreicht. Das Model wird in der Member-Variablen *model* gespeichert und wird immer dann benötigt, wenn von der View ein Wert an das Model übergeben wird. Im Konstruktor wird auch noch die Variable State initialisiert und die Installations-, Reparatur-, Deinstallations- und Abbruchkommandos an den Bootstrapper gebunden.

Immer wenn eines der Installations-, Reparatur- oder Deinstallationskommandos aufgerufen wird, wird automatisch die **PlanAction**-Methode des Bootstrappers aufgerufen. Das wird im Konstruktor über die Klasse DeligateCommando von Prism gemacht. Dem Konstruktor werden die aufzurufenden Kommandos über eine **Lambda-Expression** übergeben.

Wird das Cancel-Kommando aufgerufen, wird geprüft, ob die Installation bereits gestartet wurde. Ist das der Fall, wird der Status entsprechend gesetzt und die Bootstrapper-Methode **InvokeShutdown** aufgerufen. Die Methode InvokeShutdown sagt dem Bootstrapper, dass eine laufende Installation abgebrochen werden soll und unter Umständen der Rollback durchgeführt werden muss.

Hier der Code des Konstruktors:

```
public BurnUiViewModel(BurnUiModel model)
{
    this.model = model;
    this.State = InstallState.Initializing;

    this.WireUpEventHandlers();

    this.InstallCommand = new DelegateCommand(() =>
        this.model.PlanAction(LaunchAction.Install));

    this.RepairCommand = new DelegateCommand(() =>
        this.model.PlanAction(LaunchAction.Repair));

    this.UninstallCommand = new DelegateCommand(() =>
        this.model.PlanAction(LaunchAction.Uninstall));

    this.CancelCommand = new DelegateCommand(() =>
    {
        System.Windows.MessageBoxResult Result;
        Result = System.Windows.MessageBox.Show("Cancel?", "Cancel",
            System.Windows.MessageBoxButton.YesNo);

        if (Result == System.Windows.MessageBoxResult.Yes)
        {
            this.model.LogMessage("Cancelling...");
            if (this.State == InstallState.Applying)
            {
                this.State = InstallState.Cancelled;
            }
            else
            {
                CustomBootstrapperApplication.Dispatcher.InvokeShutdown();
            }
        }
    }, () => this.State != InstallState.Cancelled);

    Message = "Waiting for action.";
}
```

30.5.5 Definition der Event-Handler

Die Event-Handler werden in der Methode `WireUpEventHandlers` delegiert. Immer wenn der Bootstrapper ein bestimmtes Event auslöst, werden die hier verdrahteten Methoden aufgerufen:

```
private void WireUpEventHandlers()
{
    this.model.BootstrapperApplication.PlanComplete += this.PlanComplete;
    this.model.BootstrapperApplication.ApplyComplete += this.ApplyComplete;

    this.model.BootstrapperApplication.ApplyBegin += this.ApplyBegin;

    this.model.BootstrapperApplication.ExecutePackageBegin += this.ExecutePackageBegin;
    this.model.BootstrapperApplication.ExecutePackageComplete +=
        this.ExecutePackageComplete;
}
```

Die erste delegierte Funktion, die wir anschauen wollen, ist **PlanComplete**:

```
protected void PlanComplete(object sender, PlanCompleteEventArgs e)
{
    if (this.State == InstallState.Cancelled)
    {
        CustomBootstrapperApplication.Dispatcher.InvokeShutdown();
        return;
    }

    this.model.ApplyAction();
}
```

Schicken wir an den Bootstrapper ein **PlanAction**-Kommando, beginnt der Bootstrapper mit der Planungsphase. Ist diese abgeschlossen, löst dieser das Event **PlanComplete** aus. Dieses Event fangen wir ab, prüfen, ob der Anwender während der Planungsphase den Cancel-Button gedrückt hat und rufen die Bootstrapper-Methode **ApplyAction** auf, wenn dies nicht der Fall war. Somit wird die eigentliche Installation der Pakete eingeleitet.

Die **ApplyAction**-Methode ist sehr einfach aufgebaut:

```
protected void ApplyBegin(object sender, ApplyBeginEventArgs e)
{
    this.State = InstallState.Applying;
}
```

Hier setzen wir nur den Status auf `InstallState.Applying`. Das ist wichtig, damit beim Abbruch durch den Benutzer der Bootstrapper nicht nur geschlossen wird, sondern auch noch entsprechend benachrichtigt wird, dass ein Rollback durchgeführt werden soll.

Das Event **ExecutePackageBegin** ruft der Bootstrapper immer dann auf, wenn begonnen wird, ein Package zu bearbeiten. Welches Package das ist, können wir anhand des zweiten Überabeparameters ermitteln:

```
protected void ExecutePackageBegin(object sender, ExecutePackageBeginEventArgs e)
{
    if (this.State == InstallState.Cancelled)
    {
        e.Result = Result.Cancel;
    }
}
```

In unserem Beispiel prüfen wir nur, ob der Benutzer das Setup über den Cancel-Button abgebrochen hat und reagieren entsprechend.

ExecutePackageComplete ist quasi das Pendant zu ExecutePackageBegin und wird am Ende einer Package-Aktion aufgerufen:

```
protected void ExecutePackageComplete(object sender, ExecutePackageCompleteEventArgs e)
{
    if (this.State == InstallState.Cancelled)
    {
        e.Result = Result.Cancel;
    }
}
```

Das letzte Event, das wir uns hier ansehen wollen, ist **ApplyComplete**. ApplyComplete wird am Ende des Setups aufgerufen und beendet den Installationsvorgang:

```
protected void ApplyComplete(object sender, ApplyCompleteEventArgs e)
{
    this.model.FinalResult = e.Status;
    CustomBootstrapperApplication.Dispatcher.InvokeShutdown();
}
```

30.5.6 Code der BurnUiViewModel-Klasse

Hier noch einmal die gesamte Klasse zur Übersicht:

```
using Microsoft.Practices.Prism.Commands;
using Microsoft.Practices.Prism.ViewModel;
using Microsoft.Tools.WindowsInstallerXml.Bootstrapper;
using System;
using System.Windows.Input;

namespace BurnUserInterface
{
    public class BurnUiViewModel : NotificationObject
    {
        public enum InstallState
        {
            Initializing,
            Present,
            NotPresent,
            Applying,
            Cancelled
        }

        private BurnUiModel model;

        // Properties to execute commands
        public ICommand InstallCommand { get; private set; }
        public ICommand RepairCommand { get; private set; }
        public ICommand UninstallCommand { get; private set; }
        public ICommand CancelCommand { get; private set; }

        private string message;
        public string Message
        {
            get
            {
                return this.message;
            }
            set
            {
                if (this.message != value)
                {
                    this.message = value;
                    this.RaisePropertyChanged(() => this.Message);
                }
            }
        }
    }
}
```

```

        }
    }

    private InstallState state;
    public InstallState State
    {
        get
        {
            return this.state;
        }
        set
        {
            if (this.state != value)
            {
                this.state = value;
                this.Message = this.state.ToString();
                this.RaisePropertyChanged(() => this.State);
            }
        }
    }

    public BurnUiViewModel(BurnUiModel model)
    {
        this.model = model;
        this.State = InstallState.Initializing;

        this.WireUpEventHandlers();

        this.InstallCommand = new DelegateCommand(() =>
            this.model.PlanAction(LaunchAction.Install));

        this.RepairCommand = new DelegateCommand(() =>
            this.model.PlanAction(LaunchAction.Repair));

        this.UninstallCommand = new DelegateCommand(() =>
            this.model.PlanAction(LaunchAction.Uninstall));

        this.CancelCommand = new DelegateCommand(() =>
        {
            System.Windows.MessageBoxResult Result;
            Result = System.Windows.MessageBox.Show("Cancel?", "Cancel",
                System.Windows.MessageBoxButton.YesNo);
            if (Result == System.Windows.MessageBoxResult.Yes)
            {
                this.model.LogMessage("Cancelling...");
                if (this.State == InstallState.Applying)
                {
                    this.State = InstallState.Cancelled;
                }
                else
                {
                    CustomBootstrapperApplication.Dispatcher.InvokeShutdown();
                }
            }
        }, () => this.State != InstallState.Cancelled);

        Message = "Waiting for action.";
    }
}

```

```
protected void PlanComplete(object sender, PlanCompleteEventArgs e)
{
    if (this.State == InstallState.Cancelled)
    {
        CustomBootstrapperApplication.Dispatcher.InvokeShutdown();
        return;
    }

    this.model.ApplyAction();
}

protected void ApplyBegin(object sender, ApplyBeginEventArgs e)
{
    this.State = InstallState.Applying;
}

protected void ExecutePackageBegin(object sender,
                                  ExecutePackageBeginEventArgs e)
{
    if (this.State == InstallState.Cancelled)
    {
        e.Result = Result.Cancel;
    }
}

protected void ExecutePackageComplete(object sender,
                                      ExecutePackageCompleteEventArgs e)
{
    if (this.State == InstallState.Cancelled)
    {
        e.Result = Result.Cancel;
    }
}

protected void ApplyComplete(object sender, ApplyCompleteEventArgs e)
{
    this.model.FinalResult = e.Status;
    CustomBootstrapperApplication.Dispatcher.InvokeShutdown();
}

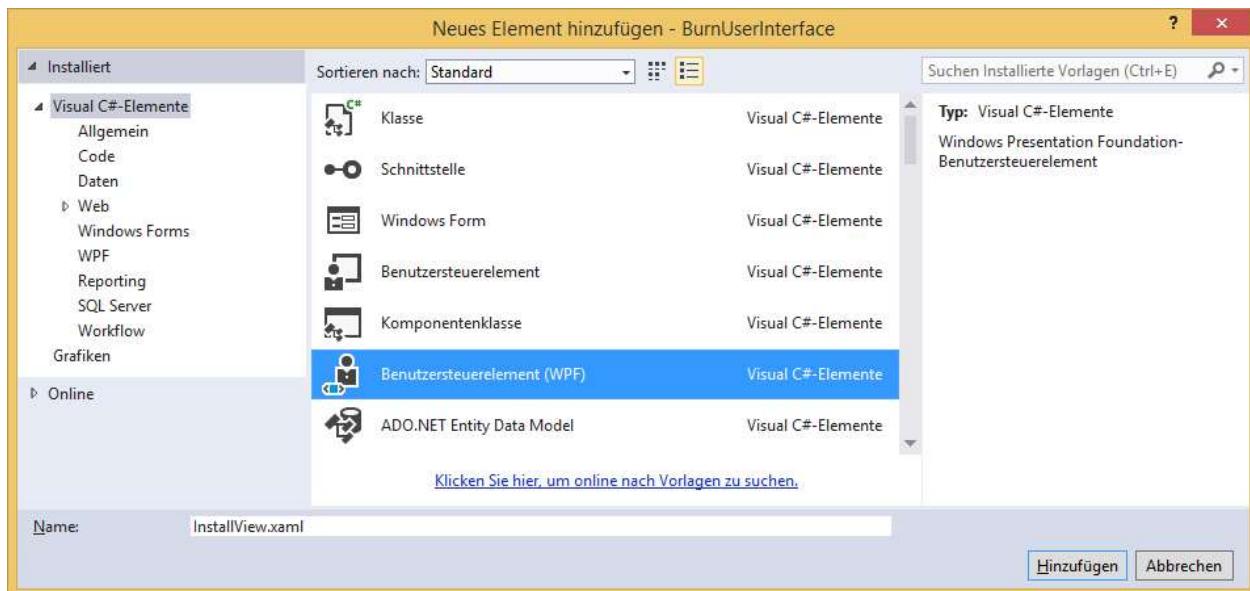
private void WireUpEventHandlers()
{
    this.model.BootstrapperApplication.PlanComplete += this.PlanComplete;
    this.model.BootstrapperApplication.ApplyComplete += this.ApplyComplete;

    this.model.BootstrapperApplication.ApplyBegin += this.ApplyBegin;

    this.model.BootstrapperApplication.ExecutePackageBegin +=
        this.ExecutePackageBegin;
    this.model.BootstrapperApplication.ExecutePackageComplete +=
        this.ExecutePackageComplete;
}
}
```

30.6 Die View-Klasse für die Erstinstallation

Ganz WPF-typisch müssen wir noch die **View-Klasse** mit der **XAML**-Datei erstellen. Wir erstellen die View, indem wir im Unterverzeichnis Views ein neues Element mit der Vorlage **Benutzersteuerelement (WPF)** erstellen. Diese Vorlage erstellt die XAML- samt zugehöriger Code-Behind-Datei:



Da wir sowohl für die Erstinstallation als auch für den Maintenance-Mode zwei unterschiedliche Dialoge und damit zwei unterschiedliche Views brauchen, erstellen wir zunächst die InstallView.

30.6.1 Der Aufbau der XAML-Datei

Die XAML-Datei ist eine XML-Datei und beschreibt das Aussehen des Dialoges:

```
<Window x:Class="BurnUserInterface.Views.InstallWelcomeView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expressionsblend/2008"
        ResizeMode="NoResize" mc:Ignorable="d" Height="400" Width="391"
        Background="AliceBlue">
    <Grid>
        <StackPanel Margin="10,0,0,0">
            <Label VerticalAlignment="Center" Margin="10,10,10,0">
                Wellcome to the custom bootstrapper user interface</Label>
            <Label Content="{Binding Message}" Margin="10"/>
            <Button Command="{Binding InstallCommand}"
                    Margin="10,20,10,5">Install</Button>
            <Button Command="{Binding CancelCommand}"
                    Margin="10,10,10,5">Cancel</Button>
        </StackPanel>
    </Grid>
</Window>
```

Das erste Attribut von Windows ist x:Class. Dieses Attribut verweist auf den Namen (inklusive Namensraum) der zugeordneten View-Klasse. Das ist also quasi das Verbindungselement zwischen der View-Klasse und XAML-Code. Danach werden alle Namensräume bekannt gegeben, mit denen wir in der XAML-Datei arbeiten.

Innerhalb des Grid-Elements definieren wir unsere Steuerelemente, die auf dem Dialog erscheinen sollen. Zunächst tragen wir ein Label ein, um eine Überschrift zu definieren. Danach wird ein weiteres Label eingefügt, das den Installationsstatus anzeigen soll. Deshalb ist dieses Label über das Command-Attribut mit der View-Variablen Message verbunden.

Danach kommen zwei Buttons – ein Installations-Button und ein Cancel-Button. Über die DelegateCommand-Klasse von Prism, das eine Verbesserung von RoutedCommand ist, binden wir in der ViewModel-Klasse das über den Button ausgelöste Event an unsere Methode.

30.6.2 Die Code-Behind-Datei

Die Code-Behind-Datei stellt den C# Code der View dar und ist recht übersichtlich:

```
using System.Windows;

namespace BurnUserInterface.Views
{
    /// <summary>
    /// Interaktionslogik für InstallView.xaml
    /// </summary>
    public partial class InstallWelcomeView : Window
    {
        public InstallWelcomeView(BurnUiViewModel viewModel)
        {
            InitializeComponent();
            this.DataContext = viewModel;
            this.Closed += (sender, e) => viewModel.CancelCommand.Execute(this);
        }
    }
}
```

In der erstellten Code-Behind-Datei ändern wir zunächst die Klassenableitung und ersetzen die UserControl-Klasse durch die Window-Klasse. Im Konstruktor der Klasse binden wir das beim Erstellen der View übergebene ModelView-Object an den DataContext. Nun sind die Label und Button der XAML-Datei an eine konkrete Implementierung der ModelView-Klasse gebunden.

Die letzte Zeile im Konstruktor leitet das Close-Event des Close-Buttons (oben rechts im Fenster) mit dem CancelCommand weiter.

30.7 Die View-Klasse für die Maintenance-Installation

Für unseren zweiten Dialog, der die Deinstalation bzw. Reparatur anbieten soll, erstellen wir ein weiteres Element über die Vorlage *Benutzersteuerelement (WPF)*.

30.7.1 Der Aufbau der XAML-Datei

Die XAML-Datei sieht der XAML-Datei von der Erstinstallation sehr ähnlich:

```
<Window x:Class="BurnUserInterface.Views.MaintlWelcomeView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
        ResizeMode="NoResize" mc:Ignorable="d" Height="400" Width="391"
        Background="AliceBlue">
    <Grid>
        <StackPanel Margin="10,0,0,0">
            <Label VerticalAlignment="Center" Margin="10,10,10,0">
                Wellcome to the custom bootstrapper user interface
            </Label>
        </StackPanel>
    </Grid>

```

```

<Button Command="{Binding RepairCommand}" Margin="10,20,10,5">
    Repair
</Button>
<Button Command="{Binding UninstallCommand}" Margin="10,10,10,5">
    Uninstall
</Button>
<Button Command="{Binding CancelCommand}" Margin="10,10,10,5">
    Cancel
</Button>
</StackPanel>
</Grid>
</Window>

```

30.7.2 Die Code-Behind-Datei

Die Code-Behind-Datei ist – bis auf den Klassennamen – sogar identisch mit der der Erstinstallation:

```

using System.Windows;

namespace BurnUserInterface.Views
{
    /// <summary>
    /// Interaktionslogik für InstallView.xaml
    /// </summary>
    public partial class MaintlWelcomeView : Window
    {
        public MaintlWelcomeView(BurnUiViewModel viewModel)
        {
            InitializeComponent();
            this.DataContext = viewModel;

            this.Closed += (sender, e) => viewModel.CancelCommand.Execute(this);
        }
    }
}

```

30.8 Userinterface von Burn erweitern

30.8.1 Eingaben an das MSI weitergeben

In diesem Kapitel wollen wir uns ansehen, wie Eingaben vom Dialog an ein MSI-Package weitergeleitet werden können. Dafür erstellen wir in der View zuerst ein Eingabefeld. Wir wollen auf dem Dialog die Seriennummer angeben lassen. Wir fügen zu unserer XAML-Datei folgende Zeilen hinzu:

```

<Grid>
    ...
    <Label VerticalAlignment="Center" Margin="10,10,10,0">
        Bitte geben sie die Seriennummer ein:
    </Label>
    <TextBox Margin="10,0,10,10" Text="{Binding SerialNumber}" ></TextBox>
    ...
</Grid>

```

Wie wir sehen, binden wir unsere TextBox an die Eigenschaft `SerialNumber`. Diese müssen wir nun im ViewModel entsprechend definieren. Dafür erstellen wir eine String-Variable mit zugehörigem Get- und Set-Kommando. Im Set-Kommando prüfen wir, ob die Seriennummer verändert wurde. Ist das der Fall, wird die Funktion `SetBurnVariable` des Models aufgerufen, das wiederum die Funktion `StringVariables` vom Bootstrapper aufruft:

```
private string serialNumber;
public string SerialNumber
{
    get
    {
        return this.serialNumber;
    }
    set
    {
        if(this.serialNumber != value)
        {
            this.serialNumber = value;
            this.model.SetBurnVariable("SerialNumber", this.serialNumber);
        }
    }
}
```

Nun müssen wir die Variable SerialNumber im Bundle an das MSI übergeben. Das MSI erwartet die Seriennummer im Property SERIALNUMBER. Damit sieht der Aufruf im Bundle so aus:

```
<MsiPackage Id="Test" Name=" MyFirstSetup.msi" ... >
    <MsiProperty Name="SERIALNUMBER" Value="[SerialNumber]" />
</MsiPackage>
```

30.9 Setupvoraussetzungen installieren

Wir haben nun gesehen, wie ein Userinterface mit Managed Code eingebunden wird. Was passiert aber, wenn beim Starten des Bootstrappers kein .NET-Framework installiert ist? Wir benötigen also noch einen Bootstrapper vor dem Bootstrapper, der die nötigen Setupvoraussetzungen schafft. Dieser Vorgang wird als **Burn Prerequisite** bezeichnet und funktioniert wie folgt:

Der Bootstrapper liest aus der Datei BootstrapperCore.config, welches Framework vom Userinterface benötigt wird. Wird das benötigte Framework nicht gefunden, werden die Setupvoraussetzungen installiert. Hierzu wird folgender Dialog geöffnet:



Drückt der Anwender auf „Akzeptieren und installieren“, wird das.NET Framework installiert. Welches Framework installiert wird, definiert man im Bundle. Dazu erstellt man eine **PackageGroup**, in der man die notwendigen Setupvoraussetzungen definiert.

In unserem Beispiel definieren wir eine PackageGroup namens Netfx4Full und installieren damit das.NET Framework 4.

Um herauszufinden, ob .NET Framework 4 installiert ist, lesen wir zunächst die Registry per RegistrySearch aus. Im ExePackage definieren wir dann eine entsprechende DetectCondition:

```
<Fragment>
    <Util:RegistrySearch Root="HKLM" Variable="Netfx4FullVersion"
        Key="SOFTWARE\Microsoft\Net Framework Setup\NDP\v4\Full" Value="Version" />

    <PackageGroup Id="Netfx4Full">
        <ExePackage Id="Netfx4Full" Cache="no" Compressed="no" PerMachine="yes"
            Permanent="yes" Vital="yes" DetectCondition="Netfx4FullVersion"
            SourceFile=".\\SourceDir\\ Prerequisite\\dotNetFx40_Full_x86_x64.exe"
            DownloadUrl="http://go.microsoft.com/fwlink/?LinkId=164193" >
            <ExitCode Value="3010" Behavior="scheduleReboot" />
        </ExePackage>
    </PackageGroup>
</Fragment>
```

Nun müssen wir noch mitteilen, dass die hier definierte PackageGroup die Setupvoraussetzungen darstellt. Das machen wir über die WiX-Variable **WixMbaPrereqPackageId**:

```
<!-- Define PackageGroup Netfx4Full as burn prerequisite package -->
<WixVariable Id="WixMbaPrereqPackageId" Value="Netfx4Full" />
```

Da wir die PackageGroup in einem Fragment definiert ist, müssen wir das Fragment im Chain-Element noch referenzieren:

```
<Chain>
  ...
  <PackageGroupRef Id="Netfx4Full"/>
  ...
</Chain>
```

Mehr müssen wir gar nicht machen.

30.10 Burn-Prerequisite-Dialog anpassen

Das Aussehen des Burn-Prerequisite-Dialoges kann vollständig über ein paar WiX-Variablen und Zusatzdateien angepasst werden. Der Dialog selbst setzt sich aus einer Theme-Ressource (THM-Datei), einer PNG-Datei (diese enthält das dargestellte Icon) sowie mehreren WXL-Dateien für die Texte zusammen, die beim **BootstrapperApplication**-Element als Payload mitgegeben werden.

Möchte man z. B. das Logo ändern, dann gibt man in der WiX-Variablen **PreqbaLogo** den Pfad zum Logo an. Das Aussehen wird über die Variable **PreqbaThemeXml** geändert werden. In der deutschen Sprache ist der Button zur Installation zu klein, den können wir einfach in dieser Datei anpassen. Genauso können wir über die Variable **PreqbaThemeWxl1031** eine Lokalisierungsdatei angeben, die die dargestellten Texte ändert. Eine Vorlage der Theme bzw. Lokalisierungsdatei findet man im Source-Code von WiX im Verzeichnis `.\src\xext\Bal\Extension\wixstdba\Resources`.

Wir definieren folgende Variablen:

```
<WixVariable Id="PreqbaThemeWxl1031" Value=".\\SourceDir\\Prerequisite\\mbapreq.wxl" />
<WixVariable Id="PreqbaThemeXml" Value=".\\SourceDir\\Prerequisite\\mbapreq.thm" />
<WixVariable Id="PreqbaLogo" Value=".\\SourceDir\\Prerequisite\\logo.png" />
```

und schon sieht der Dialog so aus:



► **Hinweis:** Im Source-Code findet man in der PayloadGroup **MbaPreqStandard** von Datei `.\src\xext\Bal\Extension\wixlib\Mba.wxs` eine vollständige Liste aller zur Verfügung stehenden WiX-Variablen.

31 Signieren von Windows Installer Setups

Warum sollten wir unser Setup überhaupt digital signieren? Tatsächlich gibt es dafür zwei gute Gründe: Die Stichworte hierzu sind Datensicherheit und **User-Account-Control(UAC)-Patching**.

31.1 Signieren des Setups

Ab Windows Installer Version 2.0 kann das Setup über die **MsiDigitalCertificate** und **MsiDigitalSignature** Tabelle prüfen, ob externe CAB-Dateien nach dem Signierungsvorgang verändert wurden. Das ist vor allem dann wichtig, wenn das Setup über das Internet vertrieben wird. Wird die digitale Signatur gebrochen (oder ist die CAB-Datei überhaupt nicht signiert), kommt bei der Installation folgende Fehlermeldung:



Die MsiDigitalCertificate-Tabelle enthält den Namen des digitalen Zertifikates (auf diesen wird in der Spalte *DigitalCertificate_* der MsiDigitalSignature-Tabelle verwiesen) und ein X.509-Zertifikat, das als CER-kodiertes Zertifikat (cer-Datei) in einer Binärtabelle gespeichert wird.

In WiX können wir diese Tabellen über das Element **DigitalSignature** sowie **DigitalCertificate** setzen. Das DigitalSignature-Element ist ein Child-Element des **Media**-Elements:

```
<Media Id="1" Cabinet="Data1.cab">
  <DigitalSignature>
    <DigitalCertificate Id="MyCertificate"
      SourceFile=".\\SourceDir\\Certificate\\Wix Certificate.cer"/>
  </DigitalSignature>
</Media>
```

Die oben dargestellten Zeilen sagen dem Windows Installer, dass die CAB-Datei mit dem angegebenen Zertifikat signiert sein muss. Die Signierung der CAB-Datei bzw. des MSI-Setups wird durch das WiX-Toolset nicht unterstützt. Das muss mit einem externen Tool, wie z. B. **SignTool.exe** (Bestandteil des Microsoft-Windows-Software-Development-Kits (SDK)) gemacht werden. SignTool.exe ruft man mit folgenden Parametern auf:

SignTool.exe sign /f "Wix Certificate.pfx" Data1.cab

Den Aufruf von SignTool.exe trägt man am besten als Postbuild-Event ein.

31.2 Das Tool Insignia

Eine Alternative zum Füllen der DigitalSignature-Elemente ist das Tool **Insignia**, das Teil des WiX-Toolsets ist. Insignia wird nach dem Erstellen des MSI und signieren der Cabinet-Dateien aufgerufen:

Insignia.exe -im setup.msi

Insignia liest aus den Cabinet-Dateien das Zertifikat aus und trägt es in die DigitalCertificate-Tabelle ein.

31.3 Zertifikat zu Testzwecken erstellen

Zertifikate sind eine Art von Identifikation, über die sichergestellt wird, dass Webdienste, Programme, Setups oder Dateien von einem bestimmten Herausgeber sind und diese nach dem Signierungsvorgang

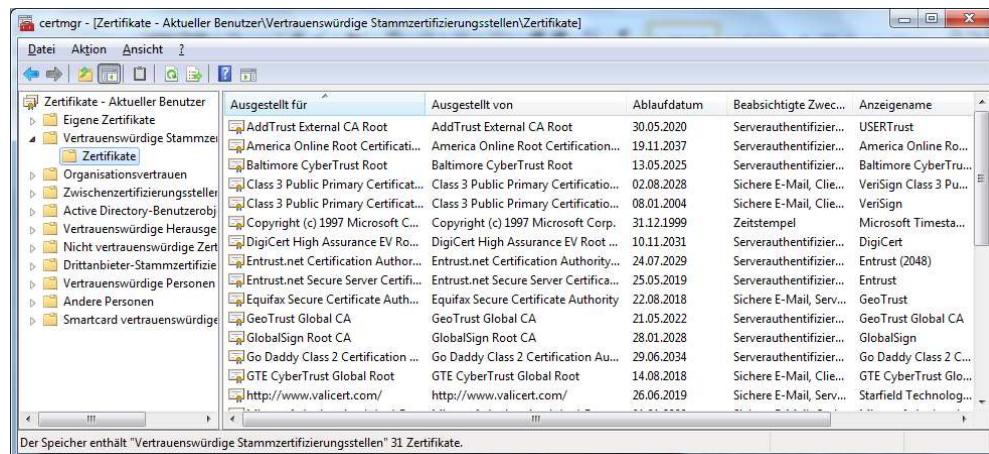
nicht mehr verändert wurden. Jedes Zertifikat muss von einer Zertifizierungsstelle (auch Stammzertifizierungsstelle oder Certificate Authority genannt) ausgestellt werden. Die Zertifizierungsstelle selbst besitzt ebenfalls ein Zertifikat, das **Stammzertifikat** genannt wird.

Die Zertifizierungsstelle stellt natürlich nicht blind jedem ein Zertifikat aus, sondern ist für die Integritäts sicherung, also für die Informationssicherheit, verantwortlich. Da dieser Service nicht kostenlos ist, verlangt die Zertifizierungsstelle einen bestimmten, in der Regel jährlich anfallenden Obolus.

Jedes von einer Zertifizierungsstelle ausgestellte Zertifikat wird über das Stammzertifikat signiert. Wird einem PC das Stammzertifikat bekanntgegeben, so wird jedem Zertifikat, das durch dieses Zertifikat signiert ist, vertraut. Diesen Vorgang nennt man Chain Trust.

Microsoft registriert bei der Installation von Windows bereits einige Stammzertifikate, die über Betriebssystem-Patches aktuell gehalten werden. Wenn man sehen will, welche Stammzertifikate auf dem PC installiert sind, dann startet man die Windows-Management-Konsole **certmgr.msc**.

Im Pfad **Vertrauenswürdige Stammzertifizierungsstellen ► Zertifikate** sind alle Stammzertifikate aufgelistet:



Wie bereits erwähnt, sind Zertifikate kostenpflichtig. Für Entwicklungs- und Testzwecken kann man sich auch selbst Zertifikate erstellen. Das kann man z. B. mit dem Microsoft-Tool **MakeCert.exe** machen. MakeCert.exe ist Bestandteil der .NET-Framework-Tools und kann von der Microsoft-Webseite heruntergeladen werden.

Um ein eigenes Zertifikat zu erstellen, brauchen wir zunächst einmal ein Stammzertifikat. Dazu rufen wir MakeCert.exe mit folgenden Parametern auf:

```
makecert.exe -n "CN=My Wix Certificate Authority" -cy authority -a sha1
-sv "My Wix Certificate Authority Private Key.pvk" -r "My Wix Certificate Authority.cer"
```

Parameter	Bedeutung
-n	Kennzeichnet den Zertifikatnamen. CN steht für Common Name und kennzeichnet den Namen der Person bzw. der Firma, für die das Zertifikat ausgestellt wurde
-cy authority	Erstellt ein Stammzertifikat
-a sha1	SHA1-Algorithmus soll verwendet werden
-sv	Name des privaten Schlüssels, der erstellt bzw. benutzt werden soll
-r	Es soll ein „self-signed“ Zertifikat – also ein Rootzertifikat erstellt werden

Da es die pvk-Datei noch nicht gibt, erstellt der -sv-Parameter eine neue Datei und fragt nach dem Passwort des privaten Schlüssels. Danach wird nach nochmaliger Nachfrage des Passworts das Zertifikat erstellt.

Nun können wir das Root-Zertifikat importieren. Hierzu öffnen wir wieder certmgr.msc mit der Microsoft-Management-Konsole mmc.exe und importieren das Stammzertifikat indem wir bei **Trusted Root Certification Authorities** ► **Certificates** mit der rechten Maustaste auf **All Tasks** ► **Import** gehen und dort die Zertifikatdatei auswählen:



Wenn wir das Zertifikat importieren, erscheint noch eine Warnung, dass Windows nicht validieren konnte, ob das Zertifikat tatsächlich von *My Wix Certificate Authority* ist oder nicht.

Wenn wir diese Meldung bestätigen, erscheint unser Zertifikat in der Liste der Stammzertifikate. Nun erstellen wir unser eigentliches Zertifikat:

```
makecert.exe -n "CN=Wix Certificate" -ic "My Wix Certificate Authority.cer"
    -iv "My Wix Certificate Authority Private Key.pvk" -a sha1 -sky exchange -pe
    -sv "Wix Certificate.pvk" "Wix Certificate.cer"
```

Parameter	Bedeutung
-n	Zertifikatname
-ic	Zertifikatdatei des Stammzertifikates
-iv	Privater Schlüssel des Stammzertifikates
-a sha1	SHA1-Algorithmus soll verwendet werden
-sky exchange	Gibt den Schlüsseltyp an – hier <i>exchange</i>
-pe	Markiert den generierten privaten Schlüssel als exportierbar. Damit kann der private Schlüssel in das Zertifikat aufgenommen werden
-sv	Gibt die Datei für den privaten Schlüssel (.pvk) an
*.cer	Dateiname der Zertifikatdatei

Wollen wir den privaten Schlüssel (pvk-Datei) mit dem Zertifikat (cer-Datei) zu einer pfx-Datei kombinieren, müssen wir noch das Tool **Pfc2Pfx.exe** aufrufen:

```
pvc2pfx.exe -pvk "Wix Certificate.pvk" -spc "Wix Certificate.cer" -pfx "Wix Certificate.pfx"
```

31.4 Konvertierung der Zertifikatdateien

Liegt kein CER-kodiertes Zertifikat vor, sondern nur eine pfx(Personal Information Exchange)-Datei, ist das nicht schlimm. Da die pfx-Datei sowohl den privaten Schlüssel als auch das zugehörige Zertifikat enthält, kann man aus einer pfx-Datei eine cer-Datei erstellen. Das kann man z. B. mit dem Tool **openssl.exe** aus dem OpenSSL-Projekt (<http://www.openssl.org/>) machen. Dafür müssen wir zuerst das Tool herunterladen und installieren. Danach können wir über folgende Kommandozeile aus einer pfx-Datei eine crt-Datei erstellen:

```
Openssl.exe pkcs12 -in xxxx.pfx -out mycertificates.crt -nokeys -clcerts
```

Die crt-Datei stellt das Zertifikat im PEM-Format dar. Da wir aber eine cer-Datei und keine crt-Datei benötigen, müssen wir die crt-Datei nochmals konvertieren. Das machen wir mit folgender Kommandozeile:

```
Openssl.exe x509 -inform pem -in mycertificates.crt -outform der -out mycertificates.cer
```

31.5 Zertifikat installieren

Die WiX-Erweiterung **WixIIsExtension** bietet Funktionen, über die Zertifikate (u. A. auch Stammzertifikate) installiert werden können. Dazu muss man zuerst eine Referenz auf die WixIIsExtension.dll hinzufügen und das Schema entsprechend erweitern.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns= "http://schemas.microsoft.com/wix/2006/wi"
      xmlns:iis="http://schemas.microsoft.com/wix/IIsExtension">
...
</Wix>
```

Das zu installierende Zertifikat kann entweder in die Binär-Tabelle aufgenommen werden oder es kann auf ein externes Zertifikat verwiesen werden. In unserem Fall wollen wir unser Stammzertifikat installieren, das wir in die Binär-Tabelle aufnehmen:

```
<Binary Id="RootCertificate" SourceFile=".\\SourceDir\\Certificates\\My Wix Certificate Authority.cer" />
```

Installiert wird das Zertifikat mit dem WiX-Element **iis:Certificate** innerhalb einer Komponente:

```
<Component Id="InstallRootCer" Guid="YOURGUID" Directory="INSTALLDIR" KeyPath="yes">
  <iis:Certificate Id="RootCertificate" Name="My Wix Certificate Authority.cer" StoreLocation="localMachine" StoreName="root" Overwrite="yes" BinaryKey="RootCertificate"/>
</Component>
```

Da wir sicherstellen müssen, dass das Stammzertifikat bereit sein muss, wenn der Windows Installer das Zertifikat der CAB-Datei prüft, muss die Aktion **InstallCertificates** vor der Aktion **InstallFiles** aufgerufen werden.

```
<InstallExecuteSequence>
  <Custom Action="InstallCertificates" Before="InstallFiles" />
</InstallExecuteSequence>
```

So ziehen wir uns wie Münchhausen selbst am eigenen Schopf aus dem Sumpf.

32 Treiberinstallation mit DIFxApp

Das Microsoft® Windows® Driver Install Frameworks (**DIFx**) ist eine Weiterentwicklung der seit Windows 95 integrierten Fähigkeit des Plug & Play. Es ist die aktuellste von Microsoft zur Verfügung gestellte Technologie, um Treiberpakete in einer hohen Qualität zur Verfügung zu stellen. Mit DIFx werden wir in die Lage gesetzt, Treiberpakete mit einer Anwendung in einem MSI-Paket zu installieren. Dafür wird auf die Standard Windows Programming Interfaces (APIs) zugegriffen.

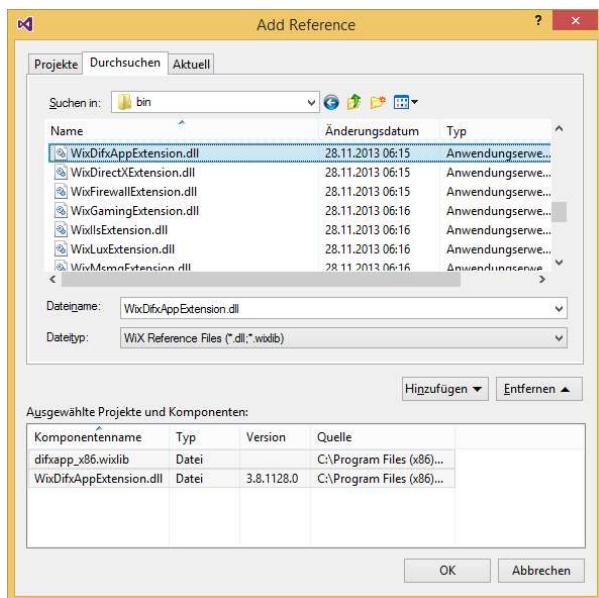
Das DIFx unterstützt die Installation von signierten Plug & Play-Treibern und signierten Class-Filter-Treibern ab dem Betriebssystem Windows 2000. DPInst kann auch im Legacy-Modus verwendet werden, dabei werden auch unsignede Treiber installiert. Selbst Treiber mit fehlenden Dateien können im Legacy-Modus installiert werden.

Eine komplette Liste der Anforderungen, die ein Treiber erfüllen muss, findet man in dem White-Paper "Requirements for Driver Packages That Are Used with the Driver Install Frameworks (DIFx) Version 2.0 Tools" (URL: http://www.microsoft.com/whdc/driver/install/DIFxtools_reqs.mspx). Das Driver-Install-Frameworks (DIFx) stellt die Struktur und die Tools zur Verfügung, mit deren Hilfe die Installation von Geräten und Treibern sowohl für Entwickler wie auch für Benutzer vereinfacht wird.

Die DIFx-Werkzeuge sind:

- Driver Package Installer (DPInst),
- Driver Installation Frameworks for Applications (**DIFxAPP**) und
- Driver Installation Frameworks Library (DIFxAPI).

Um mit DIFxAPP arbeiten zu können, binden wir die Extension **WixDifxAppExtension** ein:



► **Hinweis:** Anders als die anderen Erweiterungen bindet die WixDifxAppExtension.DLL nur die Erweiterung der MsiDriverPackages-Tabelle ein, nicht aber die zugehörigen Custom-Actions (MsiProcessDrivers, MsiInstallDrivers, MsiUninstallDrivers usw.). Diese müssen über die WiX-Librarys **difxapp_x86.wixlib** (für x86-Setups) bzw. **difxapp_x64.wixlib** (für 64-Bit-Setups) zusätzlich hinzugefügt werden. Tut man das nicht, so wird der Kompiliervorgang mit folgender Fehlermeldung quittiert:

Unresolved reference to symbol 'CustomAction:MSIProcessDrivers'

Damit das **Driver**-Element angesprochen werden kann, muss das Schema um die neuen Funktionalitäten erweitert werden:

```
<Wix xmlns='http://schemas.microsoft.com/wix/2006/wi'
      xmlns:difx='http://schemas.microsoft.com/wix/DifxAppExtension'>
```

Nun können die Treiber mithilfe des Driver-Elements installiert werden.

32.1 32-Bit- und 64-Bit-Treiber im selben Projekt installieren

Wie wir bereits an den Librarys gesehen haben, gibt es für x86- und x64-Systeme unterschiedliche Bibliotheken. Das liegt daran, dass die Custom-Actions als 32-Bit- und als 64-Bit-Actions ausgeführt sind. Unglücklicherweise werden beide Custom-Actions über dieselben Namen angesprochen. Daher kann man kein Setup erstellen, das sowohl x86- als auch x64-Treiber über DifxApp installiert.

Doch was, wenn wir dasselbe Projekt verwenden, um ein x86- und x64-Setup zu erstellen? In diesem Fall binden wir zunächst beide Librarys ein.

Wenn wir das Setup so erstellen, wird Light uns mit einer Fehlermeldung sagen, dass die Custom-Action MsiProcessDrivers doppelt definiert wurde (nämlich die x86- und x64-Variante). Damit MsBuild immer nur eine der beiden Librarys berücksichtigt, müssen wir die Projektdatei (.wixproj Datei) editieren. Relativ weit unten finden wir folgende Zeilen:

```
<ItemGroup>
    <WixLibrary Include="difxapp_x86">
        <HintPath>C:\Program Files\WiX Toolset v3.8\bin\difxapp_x86.wixlib</HintPath>
        <Name>difxapp_x86</Name>
    </WixLibrary>
    <WixLibrary Include="difxapp_x64">
        <HintPath>C:\Program Files\WiX Toolset v3.8\bin\difxapp_x64.wixlib</HintPath>
        <Name>difxapp_x64</Name>
    </WixLibrary>
</ItemGroup>
```

Wie wir sehen, werden beide Librarys über dieselbe ItemGroup eingebunden. Um das Problem zu lösen, erstellen wir für jede Library eine eigene ItemGroup und versehen diese mit einer Bedingung:

```
<ItemGroup Condition="$(Platform) == 'x86' ">
    <WixLibrary Include="difxapp_x86">
        <HintPath>C:\Program Files\WiX Toolset v3.8\bin\difxapp_x86.wixlib</HintPath>
        <Name>difxapp_x86</Name>
    </WixLibrary>
</ItemGroup>
<ItemGroup Condition="$(Platform) == 'x64' ">
    <WixLibrary Include="difxapp_x64">
        <HintPath>C:\Program Files\WiX Toolset v3.8\bin\difxapp_x64.wixlib</HintPath>
        <Name>difxapp_x64</Name>
    </WixLibrary>
</ItemGroup>
```

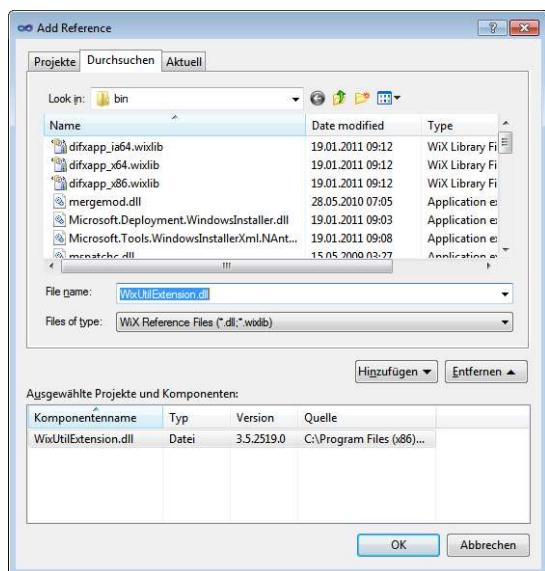
Somit wird immer nur die Library mit gebunden, die wir auch tatsächlich brauchen.

33 XML-Dateien ändern

Spätestens seit Microsoft die .NET-Zeit eingeläutet hat, benutzen immer mehr Programme XML-basierte Konfigurationsdateien anstatt der schon in die Jahre gekommenen INI-Dateien. Auch die Registry wird immer weniger als Ablageort der Konfiguration verwendet, da Anwendungen mit XML-Dateien leichter auf andere Systeme portiert werden können.

Aus einem nicht nachzuvollziehenden Grund unterstützt der Windows Installer nach seinem inzwischen mehr als zehnjährigem Dasein immer noch keine XML-Dateien. Glücklicherweise gibt es die Erweiterung **WixUtilExtension**, über die wir trotzdem XML-Dateien ändern können.

Als ersten Schritt müssen wir eine Referenz auf diese DLL eintragen:



Die WixUtilExtension.dll finden wir unter dem Installationsort vom WiX-Toolkit im Unterordner *bin*.

Im Wxs-Skript passen wir dann unser Schema entsprechend an:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
      xmlns:util="http://schemas.microsoft.com/wix/UtilExtension">
```

Grundsätzlich gibt es zwei Möglichkeiten, XML-Dateien zu verändern. Entweder man benutzt das Element **XmlFile** oder das Element **XmlConfig**. Alles was man mit XmlFile machen kann, unterstützt auch XMLConfig, ist aber beim Erstellen des zugehörigen Codes etwas komplexer.

XmlFile nimmt man in der Regel, wenn man eine XML-Datei mit dem Produkt installiert und danach modifizieren will. XMLConfig benutzt man, wenn man eine XML-Datei verändern muss, die von mehreren Anwendungen genutzt wird. XMLConfig kann auch bei der Deinstallation noch Modifikation machen.

33.1 Attribut-Werte in XML-Datei ändern

In diesem Abschnitt wollen wir uns ansehen, wie bestimmte Attribut-Werte in einer XML-Datei verändert werden können. Wir haben z. B. die Konfigurationsdatei Sample.config, die folgendes Aussehen hat:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- complete path to Sample.exe -->
    <add key="InstallationPath" value="[INSTALLFOLDER]Sample.exe" />
  </appSettings>
</configuration>
```

An der Stelle, an der derzeit [INSTALLFOLDER] steht, wollen wir den tatsächlichen Pfad der Sample.exe eintragen. Als Erstes erstellen wir eine Komponente und fügen die Sample.config dort ein:

```
<Component Id="Sample.config" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="Sample.config" Name="Sample.config"
    Source="SourceDir\XML-File\Sample.config" KeyPath="yes"/>
</Component>
```

Da wir nur Änderungen bei der Installation vornehmen, nehmen wir XmlFile. Das Element XmlFile wird – wie sollte es auch anders sein – unterhalb des Component-Elements eingefügt. Wir führen die Modifikation in derselben Komponente aus, mit der auch die XML-Datei installiert wird. Somit stellen wir sicher, dass die Modifikation immer dann gemacht wird, wenn die XML-Datei installiert wird.

XmlFile hat neben den üblichen Attributen, wie **Id** und **File**, die Attribute **Action**, **Name** und **Value**, die definieren, was in der XML-Datei gemachten werden soll. Das Attribut **ElementPath** gibt an, wo die Änderung gemacht werden soll. ElementPath benutzt die Standard **XPath spezification** (siehe <http://www.w3.org/TR/xpath/>) um den Ort der Änderung zu beschreiben. Über das Attribut **SelectionLanguage** kann man aber auch die **XSLPattern spezification** verwenden.

Folgende Aktionen können definiert werden:

Aktion	Name	Wert	Operation
createElement	name	—	Erstellt ein neues Element
setValue	—	value	Setzt einen Wert
setValue	name	value	Setzt den Wert des angegebenen Attributes
deleteValue	—	—	Löscht einen Wert
deleteValue	name	—	Löscht das angegebenen Attribut

Nun können wir in die erstellte Komponente das Element XmlFile einbauen:

```
<Component Id="Sample.config" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="Sample.config" Name="Sample.config"
    Source="SourceDir\XML-File\Sample.config" KeyPath="yes"/>
  <Util:XmlFile Id="XmlSetting1" File="[INSTALLFOLDER]Sample.config"
    Action="setValue" Name="value" Value="[INSTALLFOLDER]Sample.exe"
    ElementPath='//configuration/appSettings/add[\[]@key="InstallationPath"\[]]'
    Sequence="1" />
</Component>
```

Mit dem Attribut **Sequence** geben wir die Reihenfolge an. Da wir nur ein Element haben, ist die Reihenfolge nicht relevant. Wenn wir aber ein neues Element erstellen und danach die Attribute füllen, wäre die Reihenfolge durchaus wichtig.

Mit Elementpath geben wir zuerst die Elementkonten *configuration* und *appSettings* an. Danach kommt das übliche XPath-Format *node[@attr="value"]*. Da ElementPath einen formatierten String erwartet – Ausdrücke wie [INSTALLFOLDER] würden entsprechend ausgewertet und durch den Wert der Directory-Variablen ersetzt – und im ElementPath eckige Klammern angegeben werden, müssen wir das Zeichen [durch [/] und] durch /\] ersetzen. Dieser Vorgang heißt **Quoting**. Dadurch weiß der Windows Installer, dass hier tatsächlich die eckigen Klammern eingetragen werden sollen und kein Property referenziert wird.

33.2 Neue Elemente in XML-Datei eintragen

Wie wir im folgenden Beispiel sehen, müssen die Elemente bzw. Attribute nicht zwingend vorhanden sein. Wir haben z. B. folgende XML-Datei, die wir auch wieder mit dem File-Element installieren:

```
<settings>
</settings>
```

In diese Datei wollen wir nun ein neues Element, ein neues Attribut und im Element einen Inner-Text eintragen, sodass das Ergebnis dann so aussieht:

```
<settings>
  <NewElement path="C:\Program Files (x86)\Msi Sample\">
    inside_item
  </NewElement>
</settings>
```

Das Ändern dieser Datei erledigen wir mit folgenden Zeilen:

```
<Component Id="Demo.xml" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <File Id="Demo.xml" Source=".\\Source\\Demo.xml" KeyPath="yes"/>
  <util:XmlFile Id="AddNewElement" File="#[Demo.xml]" Action="createElement"
    Name="NewElement" ElementPath="//settings"
    PreserveModifiedDate="yes" Sequence="1"/>
  <util:XmlFile Id="AddPathAttribut" File="#[Demo.xml]" Action="setValue" Name="path"
    Value="[INSTALLFOLDER]" ElementPath="//settings/NewElement"
    PreserveModifiedDate="yes" Sequence="2" />
  <util:XmlFile Id="SetInnerItem" File="#[Demo.xml]" Action="setValue"
    Value="inside_item" ElementPath="//settings/NewElement"
    PreserveModifiedDate="yes" Sequence="3" />
</Component>
```

Wie wir sehen, wird mit Action createElement ein neues Element erstellt und mit Action setValue ein Attribut bzw. der Inner-Text gesetzt. Mit createElement müssen wir etwas vorsichtig sein. Setzen wir das Attribut PreserveModifiedDate nicht auf yes, verändert die Action das Änderungsdatum. Das hat zur Folge, dass diese Datei bei einer Reparatur nicht überschrieben wird (der Windows Installer überschreibt ja bekanntlich keine Dateien, bei denen das Änderungsdatum jünger ist als das Erstellungsdatum). Da das createElement aber trotzdem ausgeführt wird, wird zum bestehenden Element ein neues NewElement erstellt. Die Datei sieht dann so aus:

```
<settings>
  <NewElement path="C:\Program Files (x86)\Msi Sample\">
    inside_item
  </NewElement>
  <NewElement></NewElement>
</settings>
```

Bei jeder Reparatur wird dann ein neues NewElement hinzukommen. Deshalb sollte man grundsätzlich mit der createElement-Action vorsichtig sein und testen, was bei der Reparatur passiert.

34 WixUtilExtension näher betrachtet

Die WiX-Erweiterung **WixUtilExtension** stellt noch eine ganze Fülle an Erweiterungen bereit, die in dieser Lektion kurz aufgezeigt werden.

34.1 Benutzer erstellen

Über WixUtilExtension können neue Benutzer erstellt, geändert und bestimmten Gruppen zugewiesen werden. Wir verwenden dazu das Element **User**, das als Child-Element von Component abgelegt wird. Den Benutzer können wir mit **GroupRef** auch gleich einer Gruppe zuordnen. Die Gruppe muss allerdings vorher mit einem **Group**-Element ermittelt werden.

In unserem Beispiel werden wir für einen Dienst einen neuen Benutzer erstellen. Der Benutzer wird der Gruppe der Administratoren zugeordnet und der Dienst wird mit diesen Credentials gestartet.

Mit der Gruppe der Administratoren haben wir ein kleines Problem zu lösen. Der Name dieser Gruppe ist sprachabhängig. Das bedeutet, dass diese Gruppe bei jeder Betriebssystemssprache anders heißen kann. Wie beim Setzen der Berechtigungen hilft uns das WiX-Toolset mit den **WixQueryOsWellKnownSID**-Properties. Hier müssen wir nur das Property **WIX_ACCOUNT_ADMINISTRATORS** referenzieren:

```
<PropertyRef Id="WIX_ACCOUNT_ADMINISTRATORS"/>
```

Da wir unserem Benutzer auch gleich ein Passwort mitgeben, werden wir auch hierfür ein Property verwenden:

```
<Property Id="SERVICE_USER_NAME" Value="MyServiceUser" Secure="yes" />
<Property Id="SERVICE_USER_PASSWORD" Value="ABC@123" Secure="yes" Hidden="yes" />
```

Damit das Passwort nicht in das Logfile ausgegeben wird – das wäre ein kleines Sicherheitsloch - setzen wir das Attribut **Hidden** auf yes. Hierdurch wird der Name des Properties in das Windows Installer Property **MsiHiddenProperties** geschrieben und verhindert, dass das Property im Logfile erscheint. Eigentlich sollte das Passwort dann auch über eine Custom-Action und nicht über die Property-Tabelle definiert werden – aber das schenken wir uns an dieser Stelle.

Nun können wir die Gruppe der Administratoren ermitteln. Wie wir sehen, geht das ganz einfach:

```
<util:Group Id="Administrators" Name="[WIX_ACCOUNT_ADMINISTRATORS]"/>
```

Nun können wir den neuen Benutzer erstellen:

```
<Component Id="MyService.exe" Guid="YOURGUID" Directory="INSTALLFOLDER" >
  <util:User Id="ServiceUser" Name="[SERVICE_USER_NAME]" CreateUser="yes"
    LogonAsService="yes" Password="[SERVICE_USER_PASSWORD]" Disabled="no"
    PasswordNeverExpires="yes" UpdateIfExists="yes" RemoveOnUninstall="yes">
    <util:GroupRef Id="Administrators"/>
  </util:User>
  ...
</Component>
```

Da der Dienst mit diesem Benutzerkonto laufen soll, müssen wir bei **util:User** das Attribut **LogonAsService** auf yes setzen.

Nun können wir das Benutzerkonto beim Dienst angeben:

```
<ServiceInstall Id="MyServiceInstall" Name="WiXService" Start="auto"
    Type="ownProcess" DisplayName="!(loc.ServiceDisplayName)" Vital="yes"
    Description="!(loc.ServiceDescription)" ErrorControl="ignore"
    Account=".\[SERVICE_USER_NAME]" Password="[SERVICE_USER_PASSWORD]" />
```

Wenn wir das Setup nun so ausführen, dann sehen wir, dass der Dienst jetzt mit diesem Benutzerkonto gestartet wird und somit Administratorenrechte hat.

34.2 Ordnerfreigabe einrichten

Über WixUtilExtension können wir auch Ordnerfreigaben definieren. Um die Zugriffsberechtigung zu setzen, verwenden wir das Element **FileShare** und das Element **FileSharePermission**.

In unserem Beispiel legen wir einen neuen Ordner unterhalb von **CommonAppDataFolder** an, den wir für die Gruppe der Administratoren freigeben wollen. Hierzu müssen wir zuerst das Zielverzeichnis definieren:

```
<DirectoryRef Id="TARGETDIR">
    <Directory Id="CommonAppDataFolder" Name="AppData">
        <Directory Id="MyCommonAppDataFolder" Name="MyFirstSetup" />
    </Directory>
</DirectoryRef>
```

Nun können wir eine neue Komponente definieren und die Ordnerfreigabe einrichten:

```
<!-- Komponente für die Ordnerfreigabe erstellen -->
<Component Id="FileShare" Guid="YOURGUID" Directory="MyCommonAppDataFolder"
    KeyPath="yes">
    <CreateFolder Directory="MyCommonAppDataFolder" />
    <util:User Id="ExistingUser" Name="\[SERVICE_USER_NAME]" CreateUser="no"
        FailIfExists="no" />

    <util:FileShare Id="FileShare" Name="MYFILESHARE" Description="!(loc.FsDescr)">
        <util:FileSharePermission User="ExistingUser" GenericRead="yes" Read="yes"
            GenericWrite="yes" CreateFile="yes" Delete="yes" GenericAll="yes" />
    </util:FileShare>
</Component>
```

Damit der freizugebende Ordner erstellt wird, tragen wir in der Komponente ein **CreateFolder**-Element ein. Das **util:User**-Element benötigen wir für das Attribut **User** von **FileShare**. Da der Benutzer nicht in dieser Komponente erstellt werden soll, sondern nur ermittelt, setzen wir das Attribut **CreateUser** von Element **User** auf **no**. Genauso würde das dann auch aussehen, wenn wir in unserem Setup den Benutzer überhaupt nicht erstellen würden, sondern einen bestehenden Benutzer verwenden würden.

► **Hinweis:** CommonAppDataFolder ist der Ort, an dem Anwendungen, die für alle Benutzer installiert sind, ihre Daten ablegen sollten. Es unterhalb von INSTALLFOLDER zu tun, wäre eine schlechte Idee. Denn ab Windows Vista besitzt selbst ein Administrator dort keine Schreibrechte. Die User Access Control (UAC) verbietet dies bzw. virtualisiert diese Änderungen in das lokale Benutzerkonto.

34.3 Verzeichnisse rekursiv löschen

Der Windows Installer bietet zwar Funktionen an, um Dateien und Verzeichnisse zu löschen (siehe **RemoveFile** und **RemoveFolder**), allerdings gibt es keine Funktion, die ganze Verzeichnisbäume rekursiv löscht. In diesem Fall muss man auf die WixUtilExtension zurückgreifen. Diese bietet das Element **RemoveFolderEx**, über das man auch Verzeichnisse rekursiv löschen kann. Die Erweiterung durchsucht zur Laufzeit ein angegebenes Verzeichnis und trägt das angegebene Verzeichnis samt aller Unterverzeichnisse in die RemoveFolder-Tabelle ein.

Da das Löschen von Verzeichnissen einen erheblichen Einfluss auf das File-Costing hat, müssen die Einträge in der RemoveFile-Tabelle vor dem File-Costing (also vor CostInitialize) geschehen. Da die Verzeichnisvariablen aber erst mit CostFinalize initialisiert werden, gibt man das zu löschende Verzeichnis nicht über eine Directory-Variable, sondern über ein Property an. Das Property muss hierbei denselben Namen haben wie die korrespondierende Directory-Variable.

Wir müssen also sicherstellen, dass das Property genau denselben Wert hat wie die Directroy-Variable. Die einfachste Lösung ist, das entsprechende Verzeichnis in die Registry zu schreiben und dann bei der Deinstallation wieder per RegistrySearch aus der Registry zurückzulesen:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
      xmlns:util="http://schemas.microsoft.com/wix/UtilExtension">
  <Fragment>
    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="ProgramFilesFolder">
        <Directory Id="INSTALLFOLDER" Name="MyApp">
          <Directory Id="MYFOLDER" Name="MyFolder"/>
        </Directory>
      </Directory>
    </Directory>
  </Fragment>
</Wix>

<Property Id="MYFOLDER" Secure="yes">
  <RegistrySearch Id="Path" Type="raw" Root="HKLM"
    Key="SOFTWARE\[ProductName]" Name="Path"/>
</Property>

<Component Id="RegHkml" Guid="YOURGUID" Directory="INSTALLFOLDER">
  <RegistryKey Root="HKLM" Key="SOFTWARE\[ProductName]">
    <RegistryValue Name="Pfad" Value="[MYFOLDER]"
      Type="string"/>
  </RegistryKey>
</Component>

<Component Id="DelFolder" Guid="YOURGUID"
  Directory="INSTALLFOLDER">
  <util:RemoveFolderEx Id="DelMyFolder" Property="MYFOLDER"
    On="uninstall" />
</Component>
</Fragment>
</Wix>
```

Alternativ kann man das Property auch über eine Custom-Action setzen:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
      xmlns:util="http://schemas.microsoft.com/wix/UtilExtension">
  <Fragment>
    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="ProgramFilesFolder">
        <Directory Id="INSTALLFOLDER" Name="MyApp">
          <Directory Id="MyFolder" Name="MyFolder"/>
        </Directory>
      </Directory>
    </Directory>
  </Fragment>
</Wix>
```

```
<Property Id="INSTALLFOLDER" Secure="yes">
  <RegistrySearch Id="Path" Type="raw" Root="HKLM"
    Key="SOFTWARE\[ProductName]" Name="Path"/>
</Property>

<CustomAction Id="SetFolder" Property="MyFolder"
  Value="[INSTALLFOLDER]MyFolder\"/>
<InstallExecuteSequence>
  <Custom Action="SetFolder" After="AppSearch"/>
</InstallExecuteSequence>

<ComponentGroup Id="MyFiles">
  <Component Id="RegHkml" Guid="YOURGUID" Directory="INSTALLFOLDER">
    <RegistryKey Root="HKLM" Key="SOFTWARE\[ProductName]">
      <RegistryValue Name="Pfad" Value="[INSTALLFOLDER]"
        Type="string"/>
    </RegistryKey>
  </Component>

  <Component Id="Del" Guid="YOURGUID" Directory="INSTALLFOLDER">
    <util:RemoveFolderEx Id="DelMyFolder" Property="MyFolder"
      On="uninstall" />
  </Component>
</ComponentGroup>
</Fragment>
</Wix>
```

► **Vorsicht:** Die Designer des Windows Installers wussten, warum der Windows Installer Standard kein rekursives Löschen vorsieht. Rekursives Löschen ist sehr gefährlich und sollte nur auf bekannte Verzeichnissen angewandt werden, nicht auf Verzeichnisse, die z. B. vom Benutzer ausgewählt werden können. Wollen Sie z. B. das Root-Verzeichnis Ihrer Anwendung bei der Deinstallation aufräumen und löschen dieses rekursiv, dann erleben Sie ein Fiasko, wenn der Anwender als Zielverzeichnis c:\Programme\ oder gar auf ein Rootverzeichnis eines Laufwerkes ausgewählt hat. Besser ist es, die bekannten Unterverzeichnisse vom Installationsverzeichnis zu löschen.

35 Logging

35.1 Logdatei erstellen

Der Windows Installer bietet umfassende Möglichkeiten, Logdateien zu erstellen und den Detailgrad der darin abgelegten Meldungen einzustellen. Doch wie teilen wir dem Windows Installer überhaupt mit, dass Logdateien gewünscht sind? Dafür gibt es mehrere Möglichkeiten:

- über Parameter /L bei der Kommandozeile von MsiExec.exe
- über das Property **MsiLogging** (ab Windows Installer 4.0). In diesem Fall wird ein Logfile automatisch erstellt
- über die Windows Installer **Gruppenrichtlinie Logging**. In diesem Fall werden für alle MSI, die auf dem PC gestartet werden, Logdateien erstellt

Mit folgenden Parametern definieren wir den Detailgrad der Logdatei:

Wert	Beschreibung
i	Statusmeldungen
w	Warnungen
e	Fehlermeldungen
a	Start der Aktionen
r	aktionsspezifische Records
u	Nachfragen beim Benutzer
c	Initialisierungsparameter des Userinterfaces
m	Out of Memory und schwerwiegende Rückgabewerte
o	Meldungen bezüglich fehlenden Speicherplatzes
p	Terminal-Properties
v	verbose output – detaillierte Ausgaben
x	Zusätzliche Debug-Informationen (erst ab Windows Installer 3.0)
+	hängt die Logdatei an einer bestehenden Logdatei an
!	schreibt jede Zeile sofort in das Logfile (normalerweise wird das Logfile immer blockweise geschrieben)
*	Wildcard (alle Optionen außer v und x). Dieses Zeichen ist nur in der Kommandozeile von MsiExec.exe erlaubt

Bestimmt man beim Aufruf von MsiExec.exe, dass Logdateien erstellt werden sollen, dann kann das z. B. mit folgender Kommandozeile geschehen:

*MsiExec.exe /i Test.msi /L*v Test.log*

Bei der Kommandozeile von MsiExec.exe geben wir den Namen der Logdatei explizit an. Das ist anders, wenn die Logdatei über das Property **MsiLogging** bzw. über die Gruppenrichtlinien erstellt wird. In diesem Fall wird die Logdatei mit dem Namen MSIxxxx.log (xxxxx ist eine beliebige hexadezimale Zahl) abgelegt. Alleine am Inhalt der Logdatei können wir dann erkennen, zu welcher Installation die Logdatei gehört.

► **Hinweis:** Das Setup kann über das Property **MsiLogFileLocation** ermitteln, wohin und mit welchem Namen die Logdatei geschrieben wird.

35.2 Logdatei-Einträge richtig interpretieren

Da eine Logdatei, selbst bei kleineren Installationen, ohne weiteres mehrere hundert Zeilen enthalten kann, ist es wichtig, dass man die Ausgaben richtig lesen und interpretieren kann.

Die wohl interessanteste Frage bei einer abgebrochenen Installation ist: An welcher Stelle ist das Setup abgebrochen? Bzw.: An welcher Stelle im Logfile findet man Informationen über den Abbruch? Die Antwort ist relativ einfach: Wir suchen im Logfile nach dem Text „Rückgabewert 3“ (bei einem deutschen Windows Installer) bzw. „Return value 3“ (bei einem Windows Installer in englischer Sprache).

35.2.1 Client- und Server-Prozess

Die Installation wird durch einen Client- und einen Server-Prozess ausgeführt. Der Client-Prozess arbeitet die InstallUISequence-Sequenz ab, der Server-Prozess die InstallExecuteSequence. Der Server- und Client-Teil tragen als erste Buchstaben immer *MSI* (*c*) bzw. *MSI* (*s*) ein und kennzeichnen so, von welchem Prozess der Logeintrag erstellt wurde. Bei Client sieht die Zeile so aus:

```
MSI (c) (29:28) : Resetting cached policy values
```

Beim Server so:

```
MSI (s) (A0:EC): Original package ==> D:\TEMP\Default.msi
```

Die Nummer (29:28) bzw. (A0:EC) repräsentieren die letzten zwei Ziffern der Prozess- und Thread-ID. Beim Wechsel vom Client- zum Server-Prozess werden die öffentlichen Properties, die mit dem Attribut Secure="yes" gekennzeichnet sind, übergeben:

```
MSI (c) (29:28) : Switching to server: INSTALLDIR="C:\Programme\Mc"  
COMPANYNAME="SD-Technologies" USERNAME="Administrator" ...
```

Private Properties werden überhaupt an den Serverprozess übergeben.

35.2.2 Feature- und Komponenten-Status

Aus einer Logdatei ist ebenfalls ersichtlich, welche Features und Komponenten installiert bzw. entfernt wurden. Diese Informationen gibt die Aktion **InstallValidate** preis:

```
MSI (s) (A8:F0): Feature:Main; Installed:Absent; Request:Local; Action:Local  
MSI (s) (A8:F0): Feature:Help; Installed:Absent; Request:Null; Action:Null  
MSI (s) (A8:F0): Component:Test.dll; Installed:Absent; Request:Null; Action:Null  
MSI (s) (A8:F0): Component:Mfc42.dll; Installed:Absent; Request:Local; Action:Null  
...
```

So kann immer nachvollzogen werden, welche Komponenten installiert worden sind und welche nicht. Der Eintrag **Installed** gibt mit *Local* bzw. *Absent* an, ob das Item bereits installiert ist oder nicht. **Request** zeigt an, ob das Item an- bzw. abgewählt wurde und **Action** gibt schließlich an, ob das Item auch tatsächlich auf das Zielsystem übertragen werden muss oder nicht.

Wenn wir uns die Komponente *Mfc42.dll* genauer ansehen, dann sehen wir, dass **Request** auf *Local*, **Action** aber mit *Null* steht. Dies ist immer dann der Fall, wenn die Key-Datei der Komponente bereits in einer adäquaten Version auf dem Zielsystem existiert und somit nicht nochmal installiert werden muss.

35.2.3 Aktionen

Die Aktionen tragen sowohl den Start- als auch den Beendigungszeitpunkt in die Logdatei ein:

```
MSI (c) (24:50): Doing action: AppSearch
Action start 20:00:51: AppSearch.
MSI (c) (24:50): Note: 1: 2262 2: AppSearch 3: -2147287038
Action ended 20:00:51: AppSearch. Return value 1.
```

Zwischen Start und Stopp können noch zusätzlich individuelle Ausgaben erfolgen (Beispielsweise wird bei der Aktion InstallFiles genau angegeben, ob Dateien neu angelegt oder bestehende Dateien überschrieben wurden).

Viele Aktionen geben beim Beenden auch Rückgabewerte zurück, wobei die Bedeutung aus folgender Tabelle ersichtlich ist:

Wert	Beschreibung
0	Aktion wurde nicht ausgeführt
1	Aktion wurde erfolgreich ausgeführt
2	Aktion wurde vom Benutzer abgebrochen
3	Aktion wurde mit einem Fehler beendet
4	Aktion wurde angehalten und wird nach einem Reboot weitergeführt

35.2.4 Properties

Am Ende des Logfiles werden immer alle (sichtbaren) Properties ausgegeben:

```
Property(C): Date = 30.05.2002
Property(C): Time = 21:54:39
Property(C): COMPANYNAME = MicroConsulting
Property(C): USERNAME = Martin Aigner
```

Hierbei ist selbstverständlich ersichtlich, ob das Property vom Client (C) bzw. vom Server (S) ausgegeben wurde.

► **Hinweis:** Ist beim Property das Attribut *Hidden* auf yes gesetzt, so wird die Ausgabe in die Logdatei unterbunden. Das ist vor allem bei Passwörtern o. Ä. sinnvoll.

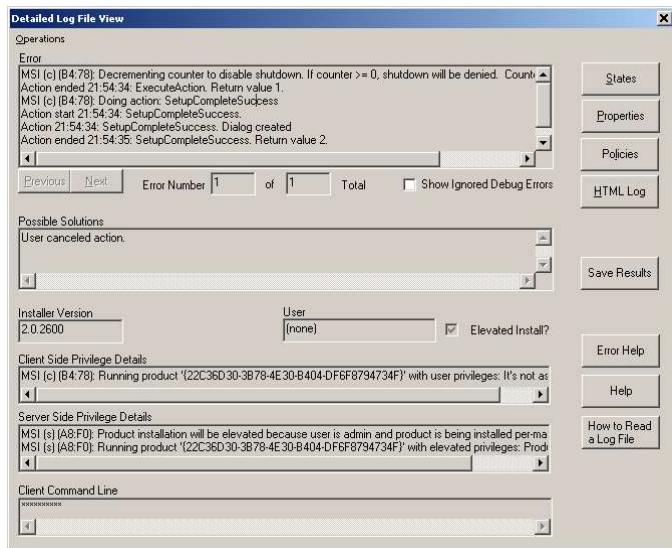
35.2.5 Gruppenrichtlinien

In die Logdatei werden auch alle für den Windows Installer relevanten Gruppenrichtlinien eingetragen:

```
MSI (c) (B4:78): Machine policy value 'DisableUserInstalls' is 0
MSI (c) (B4:78): Machine policy value 'TransformsSecure' is 0
MSI (c) (B4:78): User policy value 'TransformsAtSource' is 0
```

35.3 Analyse der Logdatei mit WiLogUtil

Das Analyse-Tool **WiLogUtil.exe** wird mit dem Microsoft Windows Installer SDK mitgeliefert. Über dieses Tool ist es sehr komfortabel möglich, Logdateien zu sichten. Mit diesem Tool können der Status der Komponenten und Feature, die Properties sowie die Gruppenrichtlinien gesichtet werden. Auch Fehlermeldungen und Warnungen können auf einfache Art und Weise eingesehen werden:



36 Transformationen

Eine **Transformation** ist eine Datei mit der Dateiendung **MST**, die man sich wie eine Art SQL-Skript vorstellen kann. Mit einer Transformation können Einträge in der MSI-Datenbank hinzugefügt, geändert und entfernt werden, ohne die MSI-Datenbank selbst zu verändern. Die Transformation ist also ein Werkzeug, mit dessen Hilfe eine Installation den individuellen Wünschen angepasst werden kann (Stichwort Customizing).

Eine Transformation wird entweder mit der zugehörigen MSI-Datenbank ausgeliefert (z. B. Sprachtransformationen, mit deren Hilfe die Setup-Sprache geändert werden kann) oder sie wird von Administratoren erstellt, um für eine Silent-Installation Beispielweise die Seriennummer und die zu installierenden Feature zu definieren. Somit ist ein Administrator in der Lage, Transformationen für verschiedene Abteilungen innerhalb eines Unternehmens zu erstellen: Die Buchhaltung benötigt andere Features als die Entwicklungsabteilung usw..

Eine Transformation wird beim Starten des MSI-Setups über die Kommandozeile angegeben:

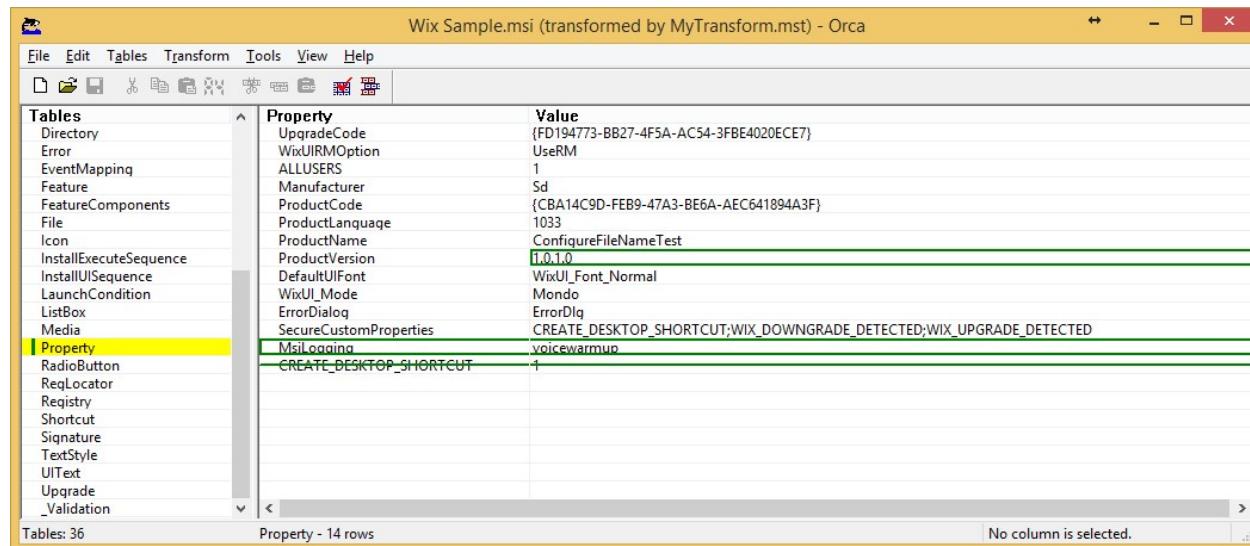
```
msiexec.exe /i MySetup.msi TRANSFORMS=MyTransform.mst
```

Die angegebene MSI-Datei wird beim Starten in den Speicher geladen, transformiert und abgearbeitet. Nach der Installation wird das MSI mit der zugehörigen MST-Datei in den Installer-Cache kopiert. In der Registry werden dann MSI und MST „verheiratet“. Damit wird die Transformation auch für die Reparatur bzw. Deinstallation berücksichtigt.

► Hinweis: Es können auch mehrere Transformationen hintereinander durch Semikolon getrennt angegeben werden. Die Transformationen werden dann von links nach rechts auf das zu installierende Setup angewandt.

36.1 Transformation mit Orca erstellen

Eine Transformation erstellt man in Orca, indem man die zu transformierende MSI-Datenbank öffnet und dann über den Menüpunkt **Transform**▶**New Transform** an wählt. Alle Änderungen, die nun am MSI vorgenommen werden, werden nun gesondert dargestellt:



Die Transformation muss dann über den Menüpunkt **Transform**▶**Generate Transform gespeichert werden**. Über **Transform**▶**Apply Transform** kann eine bestehende Transformation auch auf das MSI angewandt werden. Das geänderte MSI kann dann über **File**▶**Save As** gespeichert werden.

Über das **Windows Installer SDK** wird ein VBScript namens **wilstxfm.vbs** mitgeliefert, das die Änderungskommandos in der Transformation auf die Konsole ausgibt. Das Skript wird wie folgt gestartet:

```
cscript.exe wilstxfm.vbs WixSample.msi MyTransform.mst
```

Die oben angegebenen Änderungen werden dann folgendermaßen ausgegeben:

```
Microsoft (R) Windows Script Host, Version 5.8
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Property DELETE      [CREATE_DESKTOP_SHORTCUT]
Property Value       [ProductVersion] {1.0.0.0} -> {1.0.1.0}
Property Value       [MsiLogging] {} -> {voicewarmup}
Property INSERT     [MsiLogging]
```

36.2 Sprachtransformation erstellen

Ein sehr interessantes Anwendungsgebiet für Transformationen sind Sprachtransformationen. MSI-Setups können grundsätzlich nur in einer Sprache erstellt werden. Wenn wir aber wollen, dass das Userinterface in einer anderen Sprache dargestellt wird, so können wir für die gewünschte Sprache eine Sprache-Transformation zur Verfügung stellen.

Das Ganze hört sich jetzt vielleicht aufwendig an, ist es aber im Grunde genommen nicht. Wir erstellen einfach das Setup in deutscher und englischer Sprache. Über die Anwendung **torch.exe** können wir nun, quasi per Differenzbildung, eine Sprach-Transformation erstellen.

36.2.1 Setup in zwei Sprachen erstellen

Zuerst erstellen wir unser Setup in zwei Sprachen. Da wir unter dem Element Product die Sprache über das Attribut Language angeben, müssen wir diese durch eine Variable definieren. Das machen wir über eine String-ID im **WiX-Localisation-File**:

```
<String Id="Language">1031</String>
```

Und geben diese entsprechend an:

```
<Product Id="YOURGUID" Name="MyFirstSetup" Language="!(loc.Language)" ...>
```

36.2.2 Transformation über torch.exe erstellen

Nun kommen wir zur eigentlichen Magie: Stellen wir uns vor, wir wollen das Setup in englischer Sprache verteilen und wollen eine deutsche Sprachtransformation hinzufügen. Die deutsche Sprachtransformation nennen wir einfach 1031.mst (1031 ist ja der Language-Code von Deutschland).

Wir rufen hierzu die torch.exe mit folgender Kommandozeile auf:

```
torch.exe -p -t language en-US\MyFirstSetup.msi de-de\MyFirstSetup.msi -out 1031.mst
```

Mit dem Kommandozeilenargument -t language teilen wir torch.exe mit, dass die Sprachtransformation u.U. auch die Codepage umschalten muss. Die so erstellte Sprachtransformation kann man nun mit folgender Kommandozeile auf das englische Setup anwenden:

```
msiexec.exe /i MyFirstSetup.msi TRANSFORMS=1031.mst
```

36.2.3 Transformation einbetten

Wenn wir das Setup über das Internet verteilen wollen, dann ist es sicher hilfreich, wenn das Setup nur aus einer Datei besteht. Über das VB-Skript **wisubstg.vbs**, welches Teil vom Windows Installer SDK ist, kann die Sprachtransformation 1031.mst in das MSI mit dem Namen 1031 eingebettet werden:

```
wisubstg.vbs MyFirstSetup.msi 1031.mst 1031
```

Der Aufruf unterscheidet sich hier etwas zum Aufruf einer externen Transformation:

```
msiexec.exe /i MyFirstSetup.msi TRANSFORMS=:1031
```

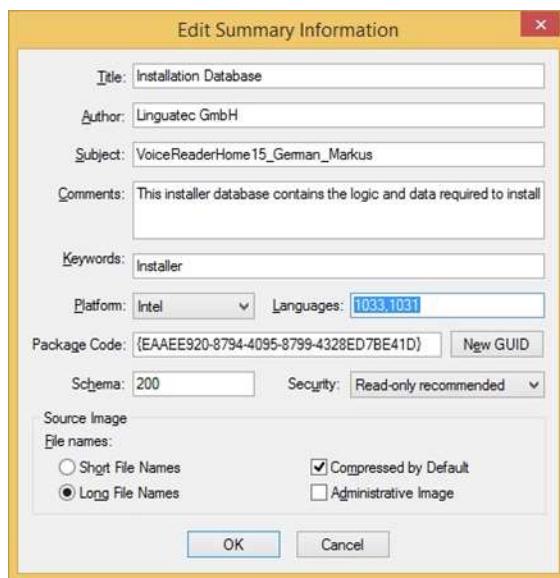
36.2.4 Automatische Wahl der MST anhand der Betriebssystemssprache

Der Windows Installer bietet die Möglichkeit, eine eingebettete Transformation automatisch anhand der eingestellten Betriebssystemssprache auszuwählen. Hierzu müssen alle Sprachtransformationen über wisubstg.vbs in das MSI eingebettet werden. Der Name der Transformation muss immer mit der Language-ID identisch sein.

Nun müssen wir nur noch die eingebetteten Transformationen über das Feld „Languages“ im Summary-Information-Stream des MSI-Setups bekannt geben. Das erreichen wir mit dem Skript heißt **WiLangId.vbs**, welches ebenfalls Bestandteil des Windows Installer SDKs ist:

```
WiLangId.vbs MyFirstSetup.msi Package 1033,1031
```

Die Language-IDs werden hierbei durch ein Komma getrennt angegeben, wobei originale Sprache des Setups an erster Stelle stehen muss. Öffnen wir das MSI mit Orca und rufen die Summary-Information-Stream-Einstellungen über das Menü View▶Summary Information auf, dann sehen die Einstellungen nun so aus:

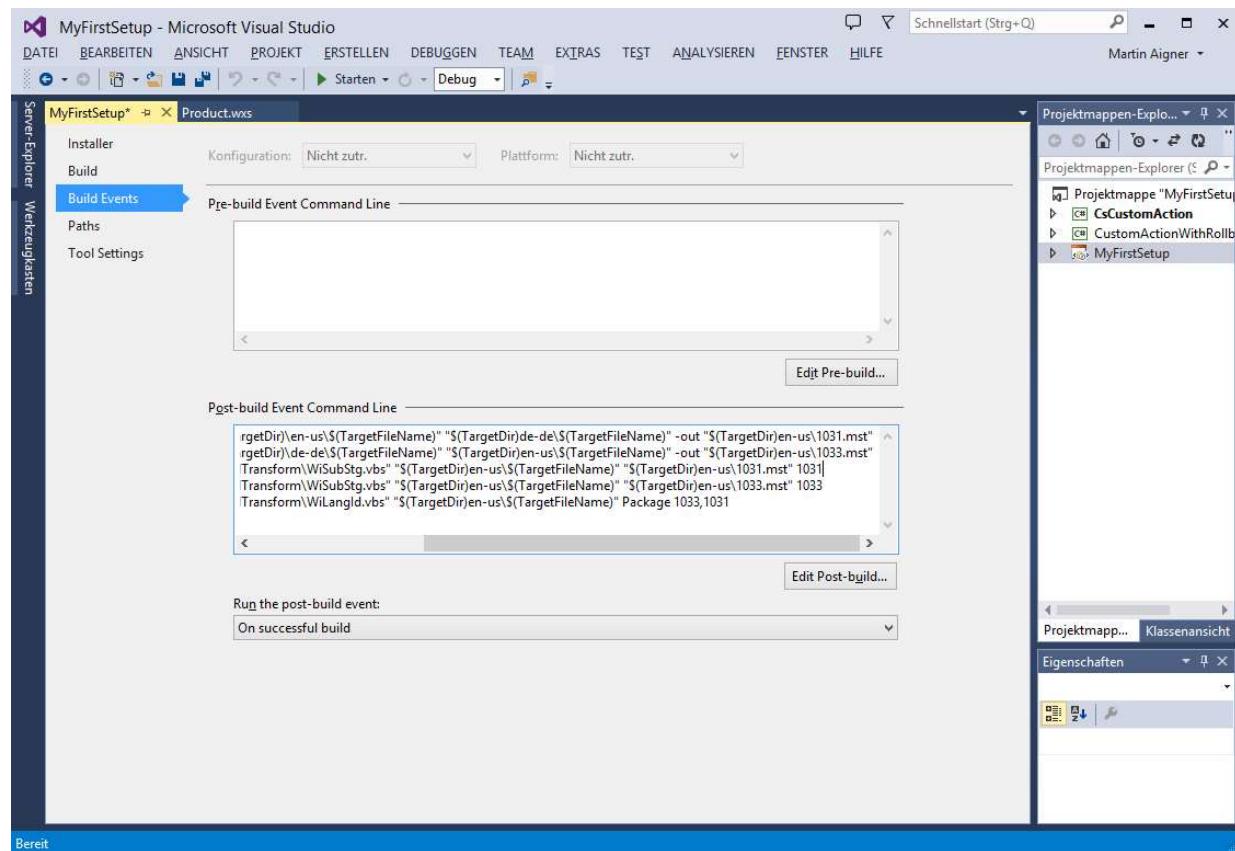


Startet man nun das MSI-Setup, wählt der Windows Installer automatisch die passende Sprachtransformation an. Ist keine passende Sprachtransformation vorhanden, wird keine Transformation angewandt.

► **Hinweis:** Zum Testen reicht es nicht aus, nur die Tastatureinstellung auf eine andere Sprache zu ändern. Wir müssen vielmehr das gesamte Betriebssystem auf eine andere Sprache umstellen. Das macht man in der Systemsteuerung unter Systemsteuerung▶Zeit, Sprache und Region▶Sprache▶Sprachoptionen.

36.2.5 Das Erstellen der Transformation in den Build-Prozess einbauen

Natürlich möchte man die oben angegebenen Kommandozeilen nach jedem Build-Prozess händisch ausführen. Vielmehr es wäre erstrebenswert, diese Zeilen über den Build-Prozess zu starten. Hierzu tragen wir die Aufrufe als Post-build Event ins Installer-Projekt ein:



Um die Aufrufe portabel zu halten, verwenden wir nach Möglichkeit Visual-Studio-Variablen:

```
"%WiX%bin\torch.exe" -p -t language "$(TargetDir)\en-us\$(TargetFileName)"  
"$(TargetDir)de-de\$(TargetFileName)" -out "$(TargetDir)en-us\1031.mst"  
cscript.exe "$(ProjectDir)\SourceDir\EmbedTransform\WiSubStg.vbs"  
"$(TargetDir)en-us\$(TargetFileName)" "$(TargetDir)en-us\1031.mst" 1031  
cscript.exe "$(ProjectDir)\SourceDir\EmbedTransform\WiLangId.vbs"  
"$(TargetDir)en-us\$(TargetFileName)" Package 1033,1031
```

37 Mehrere Instanzen eines Setups installieren

Seit der Windows Installer Version 2.0 kann man dasselbe MSI-Setup in mehreren Instanzen parallel installieren. Diesen Vorgang nennt man **Multiinstanz-Installation**. Wenn dasselbe Setup eine weitere Instanz installiert, muss dieses Setup sicherstellen, dass die neue Instanz nicht in dasselbe Verzeichnis einer früheren Instanz installiert wird.

Technisch funktioniert das ganze über **Instanztransformationen**. Die Instanztransformation ändert den Produktnamen (das ist für die Unterscheidung der Instanzen in der Systemsteuerung notwendig) und den ProductCode. Instanztransformationen können sehr einfach im WiX-Code über das Element **InstanceTransforms** erzeugt werden, das unterhalb von Element Product stehen muss.

Das InstanceTransforms-Element hat ein Attribut namens Property. In diesem Attribut gibt man den Namen des Properties an, dass beim Anwenden der Transformation die ID es **Instance**-Elements bekommt. Im Instance-Element selbst gibt man dann den ProductCode sowie den ProductName der Instance an:

```
<Property Id="MyInstance" Value="MainInstance" />

<InstanceTransforms Property="MyInstance">
    <Instance Id="SecondInstance" ProductCode="YOURGUID" ProductName="Instance 1"/>
    <Instance Id="ThirdInstance" ProductCode="YOURGUID2" ProductName="Instance 2"/>
</InstanceTransforms>
```

Anhand dem Property *MyInstance* kann nun über eine Custom-Action ein neues Zielverzeichnis für die entsprechende Instanz gesetzt werden:

```
<CustomAction Id="SetInstallDir" Directory="INSTALLDIR"
    Value="[ProgramFilesFolder]MyProduct\[MyInstance]" Execute="firstSequence" />

<InstallUISequence>
    <Custom Action="SetInstallDir" After="CostFinalize">Not Installed</Custom>
</InstallUISequence>

<InstallExecuteSequence>
    <Custom Action="SetInstallDir" After="CostFinalize">Not Installed</Custom>
</InstallExecuteSequence>
```

Light erstellt aus diesen Anweisungen zwei Transformationen namens SecondInstance und ThirdInstance (siehe Instance-ID) undbettet diese in das MSI ein. Die entsprechende Instanztransformation gibt man beim Starten des Setups über **TRANSFORMS** an:

Msiexec.exe /i Product.msi TRANSFORMS=:SecondInstance MSININSTANCE=1

Sollen noch weitere Transformationen (wie z. B. eine Sprachtransformation) berücksichtigt werden, dann muss die Instanztransformation als erste Transformation angegeben sein:

Msiexec.exe /i Product.msi TRANSFORMS=: SecondInstance;1031.mst MSININSTANCE=1

Da die Instanztransformation den ProduktCode ändert (normalerweise darf eine Transformation das nicht), müssen wir dem Windows Installer über das Property **MSININSTANCE** mitteilen, dass eine Instanztransformation angewandt werden soll. Somit erlaubt der Windows Installer das Ändern des ProductCodes bei der ersten angegebenen Transformation.

38 Updates

In den vorhergehenden Lektionen haben wir bereits gelernt, welche Installationsmöglichkeiten WiX uns bietet. Sogar wenn der Windows Installer keine direkte Lösung bietet, können wir einfach unseren guten alten Compiler hochfahren und die Lösung via Custom Action selbst schreiben.

Aber was sollen wir machen, wenn es darum geht, ein Update zu erstellen? Müssen wir den Benutzer darum bitten, die vorige Version zu deinstallieren und dann die neue wieder zu installieren? Und was würde dann mit den Eigenschaften passieren, die unser Benutzer in der Software eingestellt hat?

Selbstverständlich müssen wir das nicht erbitten. Also lasst uns schauen, wie der Windows Installer bei der Lösung derartiger Probleme helfen kann.

38.1 Update-Typen

Der Windows Installer unterstützt drei Typen von Updates: **Small Updates**, **Minor Updates** und **Major Updates**.

Ein Hauptunterscheidungsmerkmal ist, dass ein Small und Minor Update eine bestehende Installation so einspielt, dass das geachte MSI durch das neue MSI ersetzt wird und dann die geänderten Dateien durch eine Reparatur des neuen Setups aktualisiert werden. Die bereits in der richtigen Version vorliegenden Dateien werden nicht angefasst. Bei einem Major Update sieht das Ganze etwas anders aus. Ein Major Update deinstalliert zuerst die vorhandene Installation und installiert dann erst die neue Version.

Small und Minor Updates werden in der Regel bei kleineren Änderungen (z. B. dem Ersetzen einer DLL durch eine neuere Version) verwendet. Wenn grundlegende strukturelle Änderungen, wie z. B. das Löschen eines gesamten Verzeichnisses oder das Ändern der Feature-Struktur, durchgeführt werden, greift man besser zum Major Update.

Grundsätzlich (bis auf ein paar kleine Ausnahmen) ist es jedoch dem Entwickler überlassen, welchen Update-Typ er benutzt.

► **Hinweis:** Updates dürfen nicht mit Patches verwechselt werden. Ein Update kann auch als Erstinstallation verwendet werden. Es enthält immer alle zur Installation notwendigen Dateien. Ein Patch hingegen bringt nur die aktualisierten Dateien mit und ist daher auf eine bereits bestehende Version angewiesen.

Bevor wir uns mit den Einzelheiten der Update-Typen genauer beschäftigen, müssen wir uns mit ein paar GUID-Schlüsseln auseinandersetzen.

38.1.1 ProductCode, PackageCode und UpgradeCode

Der Windows Installer benötigt zur Repräsentation eines Setups drei GUID-Schlüsselelemente: den ProductCode, den PackageCode und den UpgradeCode.

Das Id-Attribut vom **Produkt**-Element nennt man **ProductCodes**. Anhand des ProductCodes erkennt der Windows Installer, ob ein Produkt bereits auf dem Rechner installiert ist oder nicht. Ein installiertes Produkt kann z. B. über den ProductCode deinstalliert werden:

```
msiexec.exe /x {GUID}
```

Der Windows Installer sucht in der Registry nach dem ProductCode, ermittelt den Namen des im Cache abgelegten MSIs und startet dieses für die Deinstallation.

Das Id-Attribut des **Package**-Elements nennt man **PackageCode**. Der PackageCode gibt verschiedene Ausprägungen desselben Produkts an. Haben wir z. B. ein Produkt, das sowohl in deutscher als auch englischer Sprache vertrieben wird, dann sollte zwar der ProductCode für beide Versionen derselbe sein, der PackageCode sollte sich aber unterscheiden.

Der **UpgradeCode** ist ebenfalls ein Attribut des Product-Elements und sollte für ein und dasselbe Produkt, unabhängig von Version und Ausprägung, immer gleich bleiben. Über den UpgradeCode kann das Setup nach einer Vorgängerversion suchen und diese gegebenenfalls durch das neue Produkt ersetzen.

38.2 Small Update

Ein Small Update unterscheidet sich vom Minor Update nur darin, dass beim Minor Update die Versionsnummer des Produktes erhöht wird, was beim Small Update nicht der Fall ist. Deshalb gilt alles, was hier über das Minor Update gesagt wird, auch für das Small Update.

38.3 Minor Updates

Da Minor Updates eine bestehende Installation so aktualisieren, dass nur die geänderten Dateien aktualisiert werden, sind ein paar Besonderheiten zu beachten, die in die Überlegung einfließen sollten, welcher Update-Typ zu wählen ist:

- Der Produktname und der Name der MSI-Datei darf nicht verändert werden.
- Die Feature-Struktur darf sich nicht marginal ändern. Es dürfen zwar neue Features erstellt werden, jedoch darf ein Kind-Feature nicht vom Vater-Feature getrennt werden.
- Es dürfen keine Komponenten gelöscht werden.
- Wenn Dateien aus Komponenten gelöscht werden, müssen diese über die RemoveFile-Tabelle explizit gelöscht werden. Ansonsten blieben diese Dateien als Leichen auf dem Rechner.

Trotz dieser Einschränkungen ist ein Update immer dann die erste Wahl, wenn nur kleine Änderungen am Setup durchgeführt werden sollen. Um ein Minor Update zu erstellen, müssen wir, neben den tatsächlichen Änderungen an den zu installierenden Dateien, Folgendes am Setup vornehmen:

Element	Beschreibung
UpgradeCode	Bleibt gleich bezüglich Vorversion
ProductCode	Bleibt gleich bezüglich Vorversion
PackageCode	Muss geändert werden bezüglich Vorversion
Version	Beim Minor Update wird die Versionsnummer geändert, beim Small Update nicht

Doch genug der Theorie, jetzt wollen wir das Ganze einmal praktisch ausprobieren. Da wir ein Minor Update erstellen wollen, ändern wir zunächst die Versionsnummer im Product-Element. Wenn die ID als GUID angegeben wurde, dann müssen wir dort auch eine neue GUID erstellen. Ist keine Id oder ein Asterix (*) angegeben, dann macht das WiX für uns:

```
<Product Id="6EDD1F03-7981-4071-9098-7B9191829881" Name="MyFirstSetup" Language="1031"
        Version="1.0.1.0" UpgradeCode="80B87725-54F2-429D-9F93-BB2C1B22A70D">
    <Package Id="*" InstallerVersion="200" Description="My very first setup">
```

Nun müssen wir nur noch die Änderungen am Produkt selbst machen. Wir werden hier einfach die EXE durch eine neuere Version austauschen. Hierzu ändern wir nur das Quellenverzeichnis von *Program V1.0* nach *Program V1.5*:

```
<ComponentGroup Id ="MainFeature">
  <Component Id="IpwiSample.exe" Guid="YOURGUID" Directory="INSTALLDIR">
    <File Id="IpwiSample.exe" Source=".\\SourceDir\\Program V1.5\\IpwiSample.exe"
      KeyPath="yes">
    </File>
  </Component>
</ComponentGroup>
```

Aus den Beispielen löschen wir auch noch eine Datei. Wie bereits erwähnt, genügt es nicht, dass wir diese Datei aus der Komponente löschen, vielmehr müssen wir diese Datei über einen **RemoveFile**-Element explizit löschen. Sind in der Komponente, aus der die Datei gelöscht wurde, noch weitere Dateien / Registry-Einträge vorhanden, dann legen wir das RemoveFile-Element in eine neuen Komponente. Somit wird sichergestellt, dass die Komponente beim Einspielen des Updates auch tatsächlich aktiviert wird:

```
<ComponentGroup Id="Samples">
  <Component Id="Samples" Guid="YOURGUID" Directory="SAMPLES">
    <File Id="Sample1.ISD" Source=".\\SourceDir\\Samples\\Sample1.ISD" />
    <File Id="Sample2.ISD" Source=".\\SourceDir\\Samples\\Sample2.ISD" />
    <File Id="Sample3.ISD" Source=".\\SourceDir\\Samples\\Sample3.ISD" />
  </Component>
  <Component Id="RemoveSamples" Guid="YOURGUID" Directory="SAMPLES">
    <RemoveFile Id="Sample4.ISD" Name="Sample4.ISD" On="install"/>
  </Component>
</ComponentGroup>
```

Wenn wir die Vorversion installiert haben und das Update starten, macht sich zunächst Enttäuschung breit. Es erscheint folgende Fehlermeldung:



Der Dialog besagt, dass ein MSI-Paket mit demselben ProductCode und anderen PackageCode installiert ist. Das lässt der Windows Installer so nicht zu. Es könnte ja auch der Fall sein, dass ein MSI in deutscher Sprache installiert ist und nun das Setup in englischer Sprache aufgerufen wird. In beiden Fällen hätte das Setup den selben ProduktCode, aber einen unterschiedlichen PackageCode. Wenn der Windows Installer diesen Aufruf ohne Meckern hinnehmen würde, dann hätten wir am Ende einen Mischmasch aus dem deutschen und englischen Setup auf dem PC. Dieser Zustand ist auf jeden Fall zu vermeiden.

Der langen Rede kurzer Sinn - wir müssen dem Windows Installer über die Kommandozeile mitteilen, dass es sich hier um ein Update handelt. Diese stimmt den Windows Installer milde und er spielt das Update ohne Murren ein. Die Kommandozeile lautet:

```
msiexec.exe /i Update.msi REINSTALL=ALL REINSTALLMODE=voums
```

Diese Kommandozeile kann man auch in einer Kurzform angeben:

```
msiexec.exe /fvoums Update.msi
```

Wenn wir das Setup über eine der angegebenen Kommandozeilen starten, erscheint der **ResumeDig**-Dialog und das Setup lässt sich installieren. **REINSTALL=ALL** besagt hierbei, dass alle bereits installierten Features erneut installiert werden sollen. Die Bedeutung von **REINSTALLMODE** ist aus der untenstehenden Tabelle ersichtlich:

REINSTALLMODE Beschreibung

- v Das Setup soll von der angegebenen Quelle gestartet werden und das MSI der Vorgängerversion im Installer-Cache ausgetauscht werden. Diese Option lässt den Windows Installer erkennen, dass es sich um ein Update handelt.
- o Es sollen alle fehlenden und neueren Dateien auf das Zielsystem übertragen werden.
- a Alternativ zu Mode o und e kann a angegeben werden. Dadurch werden Dateien kopiert, auch wenn die Versionsnummer im MSI und auf dem PC gleich sind.
- e Alternativ zu Mode o und a kann e angegeben werden. Dadurch werden Dateien immer (unabhängig von der Versionsnummer) auf das Zielsystem kopiert. Der Windows Installer führt also im Zweifelsfall auch ein Downgrade der DLL durch.
- u Es sollen alle Registry-Werte unter *HKEY_CURRENT_USER* und *HKEY_USERS* neu geschrieben werden.
- m Es sollen alle Registry-Werte unter *HKEY_LOCAL_MACHINE* und *HKEY_CLASSES_ROOT* neu geschrieben werden.
- s Es sollen alle Shortcuts (und Icons der Shortcuts) neu auf das Zielsystem übertragen werden.

Wurde mit dem Update ein neues Feature hinzugefügt, müssen wir die Installation des Features erzwingen. Das machen wir mit dem Kommando **ADDLOCAL**. Über ADDLOCAL geben wir alle Features, die installiert werden sollen, kommagetrennt an. In diesem Fall können wir aber nicht mehr **REINSTALL=ALL** angeben, sondern müssen auch dort alle Features kommagetrennt angeben, die bereits installiert waren und upgedatet werden sollen.

► **Hinweis:** Über die Properties **ADDLOCAL**, **ADDSOURCE**, **ADVERTISE**, **REMOVE**, **ADDDEFAULT** und **REINSTALL** kann über die Kommandozeile gesteuert werden, wie einzelne Features behandelt werden sollen. Weitere Informationen hierüber finden Sie in der MSI-Hilfe.

38.4 Major Update

Beim Minor Update wird die neue Version einfach über die bestehende Version „drüberkopiert“. Das hat den Vorteil, dass das Update sehr schnell eingespielt wird – es werden dann ja nur die geänderten Dateien ersetzt.

Beim Minor Update mussten wir Dateien, die jetzt nicht mehr Bestandteile der neuen Version sind, händisch löschen. Problematischer ist es, wenn die Versionsnummern der DLLs nicht angepasst wurden. Der Windows Installer vergleicht einfach die Versionsnummer der DLL im MSI-Paket mit der Versionsnummer auf dem Zielrechner. Ist die Versionsnummer gleich, dann sieht der Windows Installer keine Veranlassung, die bestehende DLL mit der neuen zu überschreiben (außer natürlich, man passt den REINSTALLMODE entsprechend an).

Beim Major Update ist die Vorgehensweise anders. Hier wird zuerst die alte Version deinstalliert und danach die neue installiert.

Um ein Major Update zu erstellen, müssen wir folgende Dinge am Setup ändern:

Element	Beschreibung
UpgradeCode	Bleibt gleich bezüglich Vorversion
ProductCode	Muss geändert werden bezüglich Vorversion
PackageCode	Muss geändert werden bezüglich Vorversion
Version	Muss geändert werden bezüglich Vorversion

Nun müssen wir unserem Setup noch sagen, welche Vorversionen gesucht und deinstalliert werden sollen. Das geschieht über die Windows Installer Tabelle **Upgrade**, die über das WiX-Element **MajorUpgrade** gesetzt werden kann.

Wenn das Projekt mit WiX 3.9 oder einer höheren Version erstellt wurde, enthält das Setup bereits folgenden Eintrag:

```
<MajorUpgrade DowngradeErrorMessage="!(loc.InstallConditionNewerProductFound)" />
```

Diese Zeile trägt in die Upgrade-Tabelle zwei Einträge ein:

UpgradeCode	VersionMin	VersionMax	Language	Attributes	Remove	ActionProperty
{80887725-54F2-429D-9F93-BB2C1B22A70D}	1.0.0.0		1			WIX_UPGRADE_DETECTED
{80887725-54F2-429D-9F93-BB2C1B22A70D}	1.0.0.0		2			WIX_DOWNGRADE_DETECTED

Die erste Zeile sucht alle älteren Versionen des Produktes und deinstalliert diese, die zweite Zeile ermittelt alle neueren Versionen, um dann im Anschluss den Benutzer zu informieren, dass bereits eine neuere Version installiert ist und dass das Setup nicht eingespielt werden kann (siehe DowngradeErrorMessage in MajorUpgrade).

Die Updated-Tabelle wird in den Sequenzen über zwei bzw. drei Aktionen abgearbeitet. Die Standardaktion **FindRelatedProducts** liest die Upgrade-Tabelle aus und sucht nach den angegebenen Sets. Wird ein Setup gefunden, wird der ProductCode der gefundenen Installation in das Property geschrieben, das in der letzten Spalte des Upgrades angegeben ist. Mehr macht FindRelatedProducts nicht.

Das Element MajorUpgrade definiert eine Installationsbedingung, die das Setup abbricht, wenn das Property WIX_DOWNGRADE_DETECTED gesetzt ist. Damit das funktioniert, wird FindrelatedProducts vor **LaunchConditions** in die Installationssequenzen eingetragen:

Tables	Action	Condition	Seque...
AppSearch	FindRelatedProducts	25	
Binary	AppSearch	50	
Certificate	LaunchConditions	100	
CertificateHash	ValidateProductID	700	
CheckBox	CostInitialize	800	
Class	FileCost	900	
ComboBox	CostFinalize	1000	
Component	SetInstallDir	1001	
Control	MigrateFeatureStates	1200	
ControlCondition	InstallValidate	1400	
ControlEvent	RemoveExistingProducts	1401	
CreateFolder	InstallInitialize	1500	

Die nächste Aktion, die für Major Updates zuständig ist, ist die Aktion **MigrateFeatureStates**.

MigrateFeatureStates überträgt den Installationsstatus aller gleichnamigen Features von der alten auf die neue Version. War z. B. bei einem Office-Setup Word installiert, Excel aber nicht, dann werden im neuen Setup die Features entsprechend an- bzw. abgewählt. Diesen Mechanismus kann man über das Attribut MigrateFeatures ausschalten.

Die eigentliche Deinstallation der Vorversion wird mit der Aktion **RemoveExistingProducts** durchgeführt. Diese Aktion kann an verschiedenen Stellen stehen und steht standardmäßig vor InstallInitialize. Die Position von RemoveExistingProducts kann über das Schedule Attribut des MajorUpgrade Elements angepasst werden.

Da bei einem Major Update die Vorversion deinstalliert wird, müssen eventuell vor der Deinstallation noch Benutzerdaten bzw. Konfigurationen gesichert werden. Das muss über eine Custom-Action erfolgen, die über das Property WIX_UPGRADE_DETECTED gesteuert werden kann.

Das Element MajorUpgrade ist immer dann angebracht, wenn wir ein Setup suchen, das denselben UpgradeCode hat wie das aktuelle Setup. Manchmal muss man aber noch Setups suchen, bei denen das nicht der Fall ist. In diesem Fall hilft uns das WiX-Element **Upgrade** weiter:

```
<Upgrade Id='80B87725-54F2-429D-9F93-BB2C1B22A70D'>
    <UpgradeVersion Maximum='1.0.5' MigrateFeatures='yes' Property='OLDVERSION' />
</Upgrade>
```

39 Patchwork

Ein Patch ist eine alternative Technologie, um Anwendungen zu aktualisieren. Anders als ein Minor oder Major Update ist ein Patch nur lebensfähig, wenn auf dem Rechner eine gültige Vorversion (auch Basisversion genannt) installiert ist. Ein Patch enthält nur die Differenz zwischen der Basis- und der Update-Version und wird deshalb immer dann bevorzugt, wenn die Größe des Updates eine Rolle spielt.

Im Patch sind die zu aktualisierenden Dateien entweder in einer Kabinettdatei oder als **Byte-Level-Patch** (Binärupdate) enthalten. Bei den Byte-Level-Patches handelt es sich um Dateifragmente, über die eine Datei aus der Basisversion so modifiziert wird, dass diese wie die Datei der Update-Version aussieht. Bei der Erstellung des Patches entscheidet der Patch-Generierungsvorgang, ob ein Byte-Level-Patch anwendbar und ökonomisch ist. Macht der Byte-Level-Patch keinen Sinn, wird die entsprechende Datei als Ganzes in die Kabinettdatei mit aufgenommen.

Patches können so erstellt werden, dass mehrere Vorversionen desselben Produktes auf die Update-Version angehoben werden. In diesem Fall spricht man von **Multi-Target-Patch**. Stellen wir uns folgendes Szenario vor:

Produkt	Version	Bemerkung
Basis	1.0.0	Wird als MSI geliefert
Update 1	1.1.0	Erster Patch
Update 2	1.2.0	Zweiter Patch

Der Multi-Target-Patch kann sowohl auf die Version 1.0.0 als auch auf 1.1.0 angewandt werden und aktualisiert beide Produkte auf Version 1.2.0.

Ein Patch kann auch mehrere unterschiedliche Produkte, die parallel auf einem Zielsystem installiert sein können, aktualisieren. Ein Anwendungsfall wäre beispielsweise, dass ein Programm über separate **Produktfamilien** sowohl als Einzelprodukt als auch als in einer Suite vertrieben wird (Microsoft Word und Office lassen grüßen). Damit alle Produktfamilien parallel existieren können, muss jedes Produkt einen eigenen Produkt-Code bekommen. Über einen sogenannten **Multi-SKU-Patch** (Stock Keeping Unit Patch), der eine Sonderform des Multi-Target-Patches ist, werden alle angegebenen Produktfamilien aktualisiert. Dies geschieht, indem der Windows Installer den Patch nacheinander auf alle verfügbaren Produktfamilien anwendet.

Ein Patch ist ein Verbunddokument mit der Dateiendung **MSP**. Im Verbunddokument befindet sich ein Summary-Information-Stream, der Informationen zu den Basisversionen enthält, für jede Produktfamilie eine Kabinettdatei mit den zu installierenden Daten und für jede Basisversion ein Transformationspaar: die Datenbank- und die Patch-Transformation. Unterstützt der Patch Funktionen des Windows Installers 3.0 oder höher, finden wir im Patch auch noch die **MsiPatchSequence**- und die **MsiPatchMediadata**-Tabelle.

39.1 Arbeitsweise des Patches

Zuerst sehen wir uns an, wie ein Patch überhaupt arbeitet. Über den Summary-Information-Stream des Patches ermittelt der Windows Installer zuerst, ob eine passende Produktlinie (also ein Produkt mit einem bestimmten ProductCode) installiert ist. Wenn kein passendes Produkt gefunden wird, wird der Patch mit Fehler **ERROR_PATCH_TARGET_NOT_FOUND** (1642) beendet. Wird eine passende Produktlinie gefunden, wird anhand des Summary-Information-Streams der Transformationen ermittelt, welche Transformation auf die gefundene Basisversion passt. Im Summary-Information-Stream der Transformation sind die Validierungsbedingungen abgelegt. Wird keine passende Basisversion gefunden, wird der Patch ebenfalls mit Fehler **ERROR_PATCH_TARGET_NOT_FOUND** beendet.

Da die Validierung vor dem eigentlichen Starten des Setups stattfindet, wird diese Prüfung nicht in der Protokolldatei des Setups abgelegt. Jedoch wird der Validierungsvorgang über die Funktion `OutputDebugString()` ausgegeben und kann so über **DebugView.exe** ausgelesen werden.

Wurde eine Basisversion gefunden und die passende Transformation ausgewählt, wird das MSI der Basisversion aus dem **Installer-Cache** (unter `c:\windows\installer`) geladen und transformiert. Die so veränderte Basisversion enthält nun alle Informationen, die die Update-Version auch enthalten würde. Die Transformation registriert in der Media-Tabelle die Patch-Kabinett-Datei und ändert die File-Tabelle so ab, dass alle neuen Dateien von der Patch-Kabinett-Datei installiert werden.

Mit der transformierten Basisversion wird nun, wie mit einem Minor Update, eine Reparatur durchgeführt. Nach der Installation wird die MSP-Datei im Installer-Cache abgelegt und über die Registry mit dem Setup verlinkt. Somit weiß der Installer bei einer Reparatur bzw. Deinstallation, dass der Patch berücksichtigt werden muss.

► Hinweis: Der Summary-Information-Stream sowie die Tabellen `MsiPatchSequence` und `MsiPatchMediadata` sind ersichtlich, wenn die MSP-Datei mit Orca geöffnet wird.

Über Orca können wir uns die Arbeitsweise des Patches noch einmal vor Augen führen. Hierzu öffnen wir in Orca zuerst die Basisversion und laden dann den Patch über Menüpunkt *Transform* ► *View Patch...*:

Component_	FileName	FileS...	Vers...	Langu...	Attribu...	Seque...
cmp6B5F6F7EBC...	tly5v3y.htm bey...	14621		512	85	
cmp49488E07B06...	gzt8gg0.zip Sa...	1695		512	86	
cmpDFC57D2D23...	fprfcihd.zip Sam...	2772		512	87	
cmp6C88AA49A...	hcfgzgga.htm jnt...	9369		512	88	
cmpF7D8E248288...	_kvcrck.htm tfel...	14612		512	89	
cmpAF9960220...	hgy_rqe.htm fra...	15711		512	90	
cmp0AE2CEA700...	be6wfrrx.htm jwe...	13048		512	91	
cmpDF1AED5AC...	yhu2oppq.htm jt...	14766		512	92	
cmpD8FDE682DF...	qrclyn8r.htm sile...	12249		512	93	
cmp40644CAF5B...	e681rucp.htm co...	13301		512	94	
cmp4B17962615A...	b3iz2r3rj.htm tran...	10741		512	95	
cmpFC8A49594...	qnd_dmd_.zip Sa...	1156		512	96	
cmpD6EAEB3BB...	p8fhjsas.htm mul...	13236		512	97	
cmp2126713A22F...	SampleCA.zip	3343		512	98	
cmp4A3C0A95A3...	GUID.vbs	228		512	99	
cmpB5064A77E83...	xz_ydsy.htm ver...	13635		512	100	
hdl.api	AcrobatPlugIn	204800		512	101	
Sample.ini	Sample.ini	75		512	103	
Sample1.ISD	Samples	Sample1.ISD	92	512	104	
Sample2.ISD	Samples	Sample2.ISD	188	512	105	
Sample3.ISD	Samples	Sample3.ISD	44	512	106	
Sample4.ISD	Samples	Sample4.ISD	36	512	107	
IpwiSample.exe	IpwiSample.exe	qmwgumxq.exe l...	139264 5.0.1 1031	4608 109		

Orca kennzeichnet alle durch den Patch geänderten Tabellen mit grüner Farbe. In der File-Tabelle sehen wir z. B., dass die Datei `Sample4.ISD` gelöscht und die Datei `IpwiSample.exe` geändert wurde (die Dateigröße, die Version, die Attribute und vor allem die Sequenznummer wurden verändert).

Die Sequenznummer gibt zum einen an, in welcher Reihenfolge die Dateien installiert werden, und zum anderen, aus welchem Medium diese kommen. Ein Blick in die Media-Tabelle verrät uns, dass die Datei `IpwiSample.exe` nicht mehr wie vorher aus der `Data1.cab` kommt, sondern jetzt aus der `Patch1.cab`.

	DiskId	LastSequence	DiskPro...	Cabinet	VolumeLa...	Source
Feature	1	107		#Data1.cab		
FeatureComponents	108	109		#Patch1.cab		_8FFFFFF006CE04372B8524950CE518477

Tables: 45 Media - 2 rows No column is selected.

Das WiX-Toolset bietet zwei Möglichkeiten, einen Patch zu erstellen. Entweder man erstellt den Patch aus den **wixpdb**-Dateien der Basis- und Update-Version – in diesem Fall werden nur native WiX-Tools verwendet – oder man erstellt den Patch über eine **Patch-Creation-Properties-Datei (PCP-Datei)** und dem Tool **msimsp.exe**, das Teil des **Windows Installer SDKs** ist.

► **Hinweis:** Die in diesem Buch beschriebene Vorgehensweise bezieht sich auf die Patch-Technologie, die ab Windows Installer 3.0 verfügbar ist. Deshalb werden nur Minor-Update-Patches besprochen und behandelt.

39.2 Vorbereitung des Patches

Egal, ob der Patch aus MSI-Setups oder aus der wixpdb-Dateien erstellt wird – die Vorbereitung für den Patch ist für beide Varianten immer die gleiche. Als ersten Schritt erstellen wir ein Minor Update und sind dann natürlich allen Einschränkungen eines Minor Updates unterworfen.

Seit Windows Installer Version 3.0 haben wir bei Patches auch die Möglichkeit, den Patch so zu gestalten, dass der Patch auch wieder **deinstalliert** werden kann. Die zu aktualisierenden Dateien werden in diesem Fall vor dem Überschreiben in den Installer-Cache abgelegt.

Soll der Patch deinstallierbar sein, dann dürfen keine Änderungen bezüglich COM-Server, Schriftarten, INI-Dateien, MIME-Types, Diensten und ODBC vorhanden sein. Außerdem dürfen keine neuen Ordner erstellt und Dateien über DuplicateFile oder MoveFile verschoben bzw. dupliziert werden.

Eine genaue Auflistung der MSI-Tabellen, die nicht verändert sein dürfen, findet man unter [https://msdn.microsoft.com/en-us/library/aa372102\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa372102(v=vs.85).aspx).

Ändert man trotzdem einer der genannten Dinge und gibt bei der Erstellung des Patches an, dass der Patch deinstallierbar sein soll, dann bekommt man bei der Erstellung des Patches eine entsprechende Warnung.

39.3 Patch installieren

Ein Patch wird über folgende Kommandozeile installiert:

```
msiexec.exe /p Patch.msp /L*v Patch.log
```

Soll ein vollständiges Logfile erstellt werden, dann geschieht das über den bereits bekannten Parameter `/L*vx`. Da bei einem Multi-SKU-Patch alle Zielproduktausprägungen nacheinander aktualisiert werden, findet man im Logfile mit diesem Parameter nur die letzte Aktualisierung. Deshalb ist es sinnvoller, hier den Parameter `/L*vx+` anzugeben.

Möchte man einen Patch auf eine **Multiinstanzinstallation** anwenden, dann muss beim Aufruf des Patches der ProductCode er entsprechenden Instanz mit angeben werden:

```
msiexec.exe /p mypatch.msp /n <ProductCode der Instanz> /qb
```

Es ist auch möglich, ein MSI zusammen mit einem Patch zu installieren. Die zugehörige Kommandozeile sieht dann so aus:

```
msiexec.exe /i mysetup.msi PATCH=mypatch.msp
```

39.4 Patch über Patch-Creation-Properties-Datei (PCP-Datei) erstellen

Wie bereits erwähnt, kann ein Patch entweder über eine PCP-Datei oder über die wixpdb-Dateien erstellt werden. Da der Weg zum Patch über eine PCP-Datei der von Microsoft vorgegebene Weg ist, wollen wir diese Methode hier genauer betrachten. Die PCP-Datei selbst ist eine Windows Installer Datenbank und kann somit auch mit Orca gesichtet und verändert werden.

39.4.1 Allgemeine Vorgehensweise

Das Tool **msimsp.exe** (Teil des **Windows Installer SDK**) entnimmt der PCP-Datei alle relevanten Parameter und vergleicht dann das Basis-MSI mit dem oder den Update-MSI. Das Ergebnis des Vergleiches wird in Transformationen und Kabinettdateien abgelegt, die dann zu einer MSP-Datei kombiniert werden. Für den Vergleich müssen alle Dateien vom Basis- und Update-MSI in entpackter Form vorliegen. Sind die Dateien in einer Kabinettdatei gepackt, dann müssen wir vor der Patch-Erstellung die MSIs über eine **administrative Installation** auspacken lassen:

```
msiexec.exe /a MySetup.msi TARGETDIR=c:\UnpackedLocation /qb
```

Durch das Property TARGETDIR geben wir das Verzeichnis an, in das das Setup entpackt abgelegt werden soll.

39.4.2 PCP-Datei erstellen

Das WiX-Toolset unterstützt den Build-Prozess insoweit, dass die PCP-Datei über WiX-Elemente beschrieben werden kann. Der Compiler erstellt dann beim Build-Prozess die PCP-Datei.

Die PCP-Datei beschreibt man in WiX über das **PatchCreation** Element:

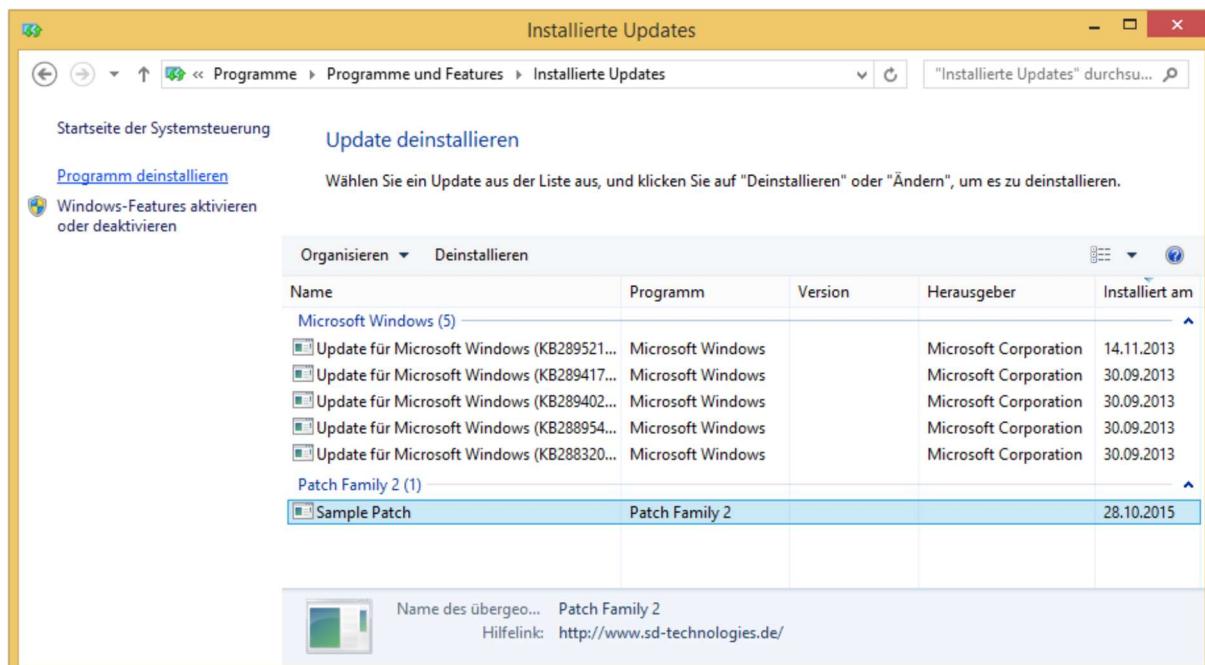
```
<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <PatchCreation Id="YOURGUID" OutputPath="patch.pcp" WholeFilesOnly="no">
    <PatchInformation Description="Patch" Comments="Sample" Manufacturer="Sd"/>

    <PatchMetadata AllowRemoval="yes" Description="Patch" ManufacturerName="Sd"
      TargetProductName="MySetup" Classification="Update"
      MoreInfoURL="http://www.sd-technologies.de/"
      DisplayName="Sample Patch"/>

    <Family Name="Patch" DiskId="2" SequenceStart="4" MediaSrcProp="SourceDirPath">
      <UpgradeImage Id="UpdateMsi" SourceFile=".\\UpdateMsi\\Sample.msi" >
        <TargetImage Id="BasisMsi" SourceFile=".\\BasisMsi\\Sample.msi" />
      </UpgradeImage>
    </Family>
    <PatchSequence PatchFamily="MyFamily" Sequence="1.0.0.0" Supersede="yes" />
  </PatchCreation>
</Wix>
```

Im PatchCreation Element sind allgemeine Einstellungen wie z. B. die Patch-ID (Patch Code) für den Patch definiert. Im Element **PatchInformation** sind die Informationen abgelegt, die im Summary-Information-Stream des PCP-Files landen. Diese Werte sind nur für die PCP-Datei, und somit für den Patch-Entwickler relevant und werden nicht auf dem Patch übertragen.

Die Informationen im **PatchMetadata**-Element landen in Tabelle **PatchMetadata** der PCP-Datei bzw. **MsiPatchMetadata** des Patches. Diese identifizieren dann den Patch auf dem Zielsystem unter **Systemsteuerung▶Programme hinzufügen oder entfernen▶Installierte Update anzeigen**:



Für das Update-MSI legen wir eine Produktfamilie an, die über das **Family**-Element definiert wird. Bei einem **Multi-SKU-Patch**, also einem Patch, der auf mehrere Basis-MSIs aufbaut und somit auch mehrere Update-MSIs besitzt, erstellen wir für jedes Update-MSI eine separate Produktfamilie.

Die Produktfamilie wird in der PCP-Datei in die **ImageFamilies**-Tabelle abgelegt, in der das Attribut *Name* den Primary-Key darstellt. Dieser Name kann nur Buchstaben und Unterstriche enthalten und darf maximal 8 Zeichen lang sein. Beim Generieren des Patches werden die zu installierenden Dateien in einer Kabinetdatei mit dem Namen PCW_CAB_<Name> abgelegt, die dann in den Patch gestreamt wird.

Das Attribut *DiskId* setzen wir auf einen Wert, der um eins höher ist als die zuletzt verwendete Disk-Id im Basis-MSI. Die zuletzt verwendete Disk-Id findet man in der Media-Tabelle. Ähnlich sieht das mit dem Attribut *SequenceStart* aus, wobei hier der Wert aus Spalte *LastSequence* um eins hoch gezählt wird. Im folgenden Bild sehen wir hierzu ein Beispiel:

Tables	DiskId	LastSequence	DiskPrompt	Cabinet	VolumeLabel	Source
InstallExecuteSequence	1	3		#cab1.cab		
InstallUISequence						
LaunchCondition						
ListBox						
Media						
MsiFileHash						
Property						
RadioButton						

Wir sehen hier die Media-Tabelle des Basis-MSIs. *DiskId* würden wir für unseren Patch auf 2 und *SequenceStart* auf 4 setzen. Erstellen wir einen auf einen anderen Patch aufsetzten Patch, dann muss die Disk-Id für jeden Patch um eins hochgezählt werden. *SequenceStart* zählen wir um die Anzahl der im letzten Patch hinzugekommenen Dateien hoch.

Am einfachsten ermittelt man die Anzahl der Dateien im Patch, indem man die Basis-MSI in Orca öffnet, den Patch über Menüpunkt *Transform▶View Patch* lädt und die Media-Tabelle sichtet.

Tables	DiskId	LastSequence	DiskPrompt	Cabinet	VolumeLa...	Source
LaunchCondition	1	3		#cab1.cab		
ListBox						
Media	2	5		#PCW_CAB_Patch1		SourceDirPath
MsiFileHash						
MsiPatchHeaders						
Patch						

In diesem Fall würden wir *DiskId* auf 3 und *SequenceStart* auf 6 setzen.

Das Attribut *MediaSrcProp* wird ebenfalls für die Media-Tabelle des Patches benötigt und wird in die Spalte *Source* eingetragen. Diese Spalte definiert den Namen eines Properties, das bei der Installation des Patches den Dateinamen und Speicherort der MSP-Datei enthält.

Im Family-Element geben wir über das **UpgradelImage**-Element das administrativ entpackte Update-MSI bekannt. Das bzw. die Basis-MSIs werden über **TargetImage** angegeben, das wiederum ein Child-Element von UpgradelImage ist.

Als letztes Element kommt noch **PatchSequence** hinzu, das im MSP-File in die **MsiPatchSequence**-Tabelle abgelegt wird. PatchSequence wird immer dann definiert, wenn ein Patch auf bereits vorhandene Patches angewandt werden soll. In PatchSequence definiert man über das Attribut PatchFamily die Patch-Familie. Die Patch-Familie ist ein beliebiger Name, dem wir über das Sequence-Attribut eine Versionsnummer zuweisen können. Sind beim Installieren eines Patches bereits Patches derselben Patch-Familie vorhanden, gibt die Versionsnummer der Patch-Familie die Reihenfolge der einzelnen Patch-Transformationen vor. Das ist vor allem bei Small-Update-Patches notwendig, da hier der Windows Installer nicht über die Produktversion ermitteln kann, in welcher Reihenfolge die Patches angewandt werden müssen.

Zusätzlich kann über das Attribut *Supersede* definiert werden, dass ältere Patches derselben Patch-Familie vor dem Anwenden des Patches entfernt werden sollen. Entfernt ist hier nicht ganz der treffende Ausdruck: Patches, die Supersede sind, werden nur ausgeblendet und bleiben weiterhin auf dem System. Deinstalliert man einen Patch, der ältere Mitglieder derselben Patch-Familie ausgeblendet hat, dann werden die ausgeblendeten Patches wieder aktiviert.

► **Hinweis:** Ein Patch kann die Zugehörigkeit zu mehreren Patch-Familien definieren. Das findet vor allem bei Multi-SKU-Patch Anwendung.

39.4.3 Properties der PCP-Datei

Microsoft hat für die PCP-Datei eine ganze Reihe weiterer Einstellungen in Form von Properties definiert. Diese kann man in der WXS-Datei unterhalb der Family-Elemente definieren. Das geschieht mit dem **PatchProperty**-Element.

39.4.4 Patch erstellen

Haben wir das Basis- und Update-MSI administrativ entpackt und die WXS-Datei für die PCP-Datei erstellt, dann können wir über folgende Kommandozeilen den Patch erstellen:

```
candle.exe "Patch.wxs"  
light.exe "Patch.wixobj" -out "patch.pcp"  
msimsp.exe -s "patch.pcp" -p "patch.msp" -l patch.log
```

39.5 Patch über Pyro.exe und MSI erstellen

Alternativ zur Erstellung der Patches über PCP-Dateien, können Patches auch über **Pyro.exe**, das Bestandteil des WiX-Toolsets ist, erstellt werden. Im Gegensatz zu msimsp.exe beschreibt man den Patch nicht über eine PCP-Datei, sondern über wixmst- und wixmsp-Dateien.

Dazu definiert man zuerst das grundsätzliche Aussehen des Patches in einer WXS-Datei über das **Patch**-Element:

```
<?xml version="1.0" encoding="utf-8"?>  
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">  
  <Patch AllowRemoval="yes" Manufacturer="SD" DisplayName="Sample Patch"  
        Description="Patch" Classification="Update">  
    <Media Id="2" Cabinet="Patch1.cab" Source="SourceDirPath" >  
      <PatchBaseline Id="UpdateBaseline"/>  
    </Media>  
  
    <PatchFamily Id="TestPatch" Version="1.0.0" Supersede="no" />  
  </Patch>  
</Wix>
```

Vergleicht man das Patch-Element mit dem PatchCreation-Element, das wir von der PCP-Datei her kennen, erkennt man viele Parallelen. Beide Elemente enthalten sogar viele gleichnamige Child-Elemente. Der größte Unterschied besteht vor allem bei der Definition von Basis- und Update-Version. Diese werden im PatchCreation-Element über das Family-Element und hier über eine externe Referenz in Form eines **PatchBaseline**-Elements definiert. Die PatchBaseline repräsentiert eine Patch-Transformation, die über **torch.exe** erstellt wird.

Um den Patch zu erstellen, müssen auch hier die Basis- und Update-MSIs über eine **administrative Installation** entpackt werden. Ist das geschehen, kann über die **torch.exe** eine Transformation in Form einer **wixmst**-Datei erstellt werden:

```
Torch.exe -p -ax ".\Temp" -t patch -xo ".\Basis\Setup.msi" ".\Update\Setup.msi" -out Patch.wixmst
```

Der Parameter **-ax** gibt an, dass alle Dateien extrahiert werden sollen, die in Binär-Tabellen gebunden sind. Das sind u. a. alle Dateien in der Binär- und der Icon-Tabelle. Mit **-t** geben wir an, dass die Transformation für Patches verwendet werden soll – hier ist eine sinnvolle Kombination der Error- und Validierungs-Flags für die Patch-Transformation vordefiniert. Parameter **-xo** schließlich gibt an, dass eine **wixmst**-Datei und keine **mst**-Datei erstellt werden soll.

Die **Torch.exe** rufen wir für jede PatchBaseline, also für jedes Basis-/Update-MSI-Paar, auf und erstellen die entsprechende Transformation.

Nun starten wir **Candle.exe** und **Light.exe** um eine **wixmsp**-Datei aus der **WXS**-Datei zu erstellen:

```
candle.exe Patch.wxs  
light.exe Patch.wixobj -out "Patch.wixmsp"
```

Letztendlich erstellen wir mit **Pyro.exe** den Patch:

```
pyro.exe Patch.wixmsp -out Patch.msp -t UpdateBaseline Patch.wixmst
```

Über den Parameter **-t** werden Patch-Transformation und Baseline verlinkt.

► Hinweis: Erstellen wir ein Patch, der auf mehrere Basisversionen angewandt werden soll, dann gibt man den Parameter **-t**, gefolgt von PatchBaseline und **wixmst**-Datei, mehrfach an.

39.6 Patch über Pyro.exe und wixpdb-Datei erstellen

Wie bereits erwähnt, kann die Patcherstellung auch eine **wixpdb**-Datei statt der MSI-Datei verwenden. Die **wixpdb**-Datei wird beim Linken von **Light.exe** erstellt und stellt das MSI im WiX-Format dar. Somit erspart man sich das Erstellen von administrativen Installationen.

Auch hier erstellen wir zunächst eine Transformation mittels **torch.exe**, nehmen aber jetzt zum Erstellen der Transformation nicht die MSIs, sondern die **wixpdb**-Dateien:

```
torch.exe -p -xi alt\ MyFirstSetup.wixpdb neu\ MyFirstSetup.wixpdb -out Patch.wixmst
```

Das Erstellen der **Patch.wixobj**-Datei sieht genauso aus wie bei der Erstellung des Patches mittels MSI:

```
candle.exe Patch.wxs  
light.exe Patch.wixobj -out "Patch.wixmsp"  
pyro.exe Patch.wixmsp -out Patch.msp -t MyBaseline UpdateBaseline.wixmst
```

39.7 Fallbeispiele PCP-Patch

In den folgenden Fallbeispielen sind ein paar typische Anwendungsfälle der Patch-Erstellung dargestellt. Alle Beispiele gehen davon aus, dass wir die Patches über PCP-Dateien erstellen.

39.7.1 Erster Patch

Im ersten Fallbeispiel betrachten wir den ersten Patch, den, der auf der Basisversion aufsetzt:

Version	Form	Beschreibung
1.0	MSI	Basis-MSI
1.1	MSP	Patch auf Basis-MSI

Wir erstellen für die Version 1.1 ein Minor Update, für das Basis- und Update-MSI eine administrative Installation und anschließend den Patch mit folgender WXS-Datei:

```
<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <PatchCreation Id="YOURGUID" OutputPath="patch.pcp" WholeFilesOnly="no">
        <PatchInformation Description="Patch" Comments="Sample" Manufacturer="Sd"/>

        <PatchMetadata AllowRemoval="yes" Description="Patch" ManufacturerName="Sd"
            TargetProductName="MySetup" Classification="Update"
            MoreInfoURL="http://www.sd-technologies.de/"
            DisplayName="Sample Patch"/>

        <Family Name="Patch" DiskId="2" SequenceStart="4" MediaSrcProp="SourceDirPath">
            <UpgradeImage Id="UpdateMsi" SourceFile=".\\Version1.1\\Sample.msi" >
                <TargetImage Id="BasisMsi" SourceFile=".\\Version1.0\\Sample.msi" />
            </UpgradeImage>
        </Family>

        <PatchSequence PatchFamily="MyFamily" Sequence="1.1.0.0"/>
    </PatchCreation>
</Wix>
```

39.7.2 Patch baut auf vorhergehendem Patch auf

Im nächsten Fallbeispiel wollen wir einen Patch erstellen, der auf den ersten Patch aufbaut. Um den zweiten Patch (Version 1.2) einspielen zu können, muss zwingend die Version 1.1 auf dem Zielsystem vorhanden sein.

Version	Form	Beschreibung
1.0	MSI	Basis-MSI
1.1	MSP	Patch auf Basis-MSI
1.2	MSP	Zweiter Patch, der auf den ersten Patch aufbaut

Wir erstellen ausgehend von der Version 1.1 ein Minor Update, installieren Version 1.1 und 1.2 administrativ und erstellen den Patch mit folgender WXS-Datei:

```

<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <PatchCreation Id="YOURGUID" OutputPath="patch.pcp" WholeFilesOnly="no">
    <PatchInformation Description="Patch" Comments="Sample" Manufacturer="Sd"/>

    <PatchMetadata AllowRemoval="yes" Description="Patch" ManufacturerName="Sd"
      TargetProductName="MySetup" Classification="Update"
      MoreInfoURL="http://www.sd-technologies.de/"
      DisplayName="Sample Patch"/>

    <Family Name="Patch" DiskId="3" SequenceStart="6" MediaSrcProp="SourceDirPath">
      <UpgradeImage Id="UpdateMsi" SourceFile=".\\Version1.2\\Sample.msi" >
        <TargetImage Id="BasisMsi" SourceFile=".\\Version1.1\\Sample.msi" />
      </UpgradeImage>
    </Family>

    <PatchSequence PatchFamily="MyFamily" Sequence="1.2.0.0" Supersede="no" />
  </PatchCreation>
</Wix>

```

Das Basis-MSI ist in diesem Fall die Version 1.1 und das Update-MSI ist Version 1.2. Bis auf die Attribute DiskId und SequenceStart des Family-Elements und der Sequenznummer im PatchSequence-Element unterscheidet sich die WXS-Datei nicht von der des ersten Patches. Um anzugeben, dass dieser Patch auf einem bereits vorhandenen Patch aufbaut, setzen wir das Attribut Supersede im PatchSequence-Element auf *no*. Das Attribut Supersede muss hier nicht zwingend angegeben werden, da *no* die Standardeinstellung ist.

39.7.3 Patch ersetzt vorhergehenden Patch (kumulativer Patch)

Das letzte Fallbeispiel hat vorausgesetzt, dass die Version 1.1 bereits eingespielt ist. Möchte man einen Patch erstellen, der sowohl auf Version 1.0 als auch auf Version 1.1 aufsetzen soll, sieht das Ganze etwas anders aus. Diese Art von Patch nennt man **kumulativen Patch**.

Version	Form	Beschreibung
1.0	MSI	Basis-MSI
1.1	MSP	Patch auf Basis-MSI
1.2	MSP	Zweiter Patch soll auf Version 1.0 und Version 1.1 aufsetzen

Wir erstellen ausgehend vom Minor Update Version 1.1 ein weiteres Minor Update (Version 1.2). Nun installieren wir Version 1.0 und die neu erstellte Version 1.2 administrativ und erstellen den Patch mit folgender WXS-Datei:

```

<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <PatchCreation Id="YOURGUID" OutputPath="patch.pcp" WholeFilesOnly="no">
    <PatchInformation Description="Patch" Comments="Sample" Manufacturer="Sd"/>

    <PatchMetadata AllowRemoval="yes" Description="Patch" ManufacturerName="Sd"
      TargetProductName="MySetup" Classification="Update"
      MoreInfoURL="http://www.sd-technologies.de/"
      DisplayName="Sample Patch"/>

```

```
<Family Name="Patch" DiskId="2" SequenceStart="6" MediaSrcProp="SourceDirPath">
    <UpgradeImage Id="UpdateMsi" SourceFile=".\\Version1.2\\Sample.msi" >
        <TargetImage Id="BasisMsi" SourceFile=".\\Version1.0\\Sample.msi" />
    </UpgradeImage>
</Family>

    <PatchSequence PatchFamily="MyFamily" Sequence="1.2.0.0" Supersede="yes" />
</PatchCreation>
</Wix>
```

Das Basis-MSI ist in diesem Fall die Version 1.0 und das Update-MSI ist Version 1.2. Das Attribut Supersede vom PatchSequence-Element ist die diesem Fall sehr wichtig. Dieses setzen wir auf yes und geben damit an, dass alle älteren Patches derselben Patch-Familie durch diesen Patch ersetzt werden. Da alle älteren Patches ersetzt werden, kann auch die Disk-Id im Family-Element wieder mit dem Wert 2 beginnen.

39.7.4 Patch setzt auf vorhergehendem Patch und Major-Update-MSI auf

In diesem Fallbeispiel gehen wir davon aus, dass das erste Update (Version 1.1) nicht nur als Patch, sondern auch als Vollupdate verteilt werden soll. Da das Einspielen eines Minor Updates eine spezielle Kommandozeile benötigt, soll das Vollupdate als Major Update verteilt werden.

Da wir zur Erstellung des Patches ein Minor Update benötigen, gehen wir wie folgt vor: Wir erstellen zuerst ein Minor Update, das die Basisversion auf die Update-Version hebt. Aus dem Minor Update erstellen wir den Patch. Durch Einfügen eines neuen Product- und Package-Codes machen wir anschließend aus dem Minor Update ein Major Update. Somit unterscheidet sich das Vollupdate vom gepatchten Setup nur in diesen zwei Punkten.

Der Werdegang des Patches sieht dann wie folgt aus:

Version	Form	Beschreibung
1.0	MSI	Basis-MSI
1.1	MSP	Patch auf Basis-MSI
1.1	MSI	Vollversion und Major Update von Version 1.0
1.2	MSP	Patch soll auf Version 1.0, auf Patch 1.1 und auf Major-Version 1.1 aufsetzen. Der Patch 1.1 soll durch den neuen Patch ersetzt werden.

Um den Patch Version 1.2 zu bekommen, erstellen wir ausgehend vom Minor Update Version 1.1 ein weiteres Minor Update Version 1.2. Da der Patch auch auf das Major Update Version 1.1 passen soll und wir nur Patches mit Minor Updates bauen können, kopieren wir den Produkt-Code aus dem Major Update 1.1 in das Minor Update Version 1.2. So haben wir zwei passende Paare:

Paar 1

Version 1.0 (ProductCode A)

Update 1.2 (ProductCode A)

Paar 2

Version 1.1 (ProductCode B)

Version 1.2 (ProductCode B)

Aus beiden Paaren können wir nun einen Patch erstellen. Die zugehörige PCP-Datei sieht dann wie folgt aus:

```

<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <PatchCreation Id="YOURGUID" OutputPath="patch.pcp" WholeFilesOnly="no">
    <PatchInformation Description="Patch" Comments="Sample" Manufacturer="Sd"/>

    <PatchMetadata AllowRemoval="yes" Description="Patch" ManufacturerName="Sd"
      TargetProductName="MySetup" Classification="Update"
      MoreInfoURL="http://www.sd-technologies.de/"
      DisplayName="Sample Patch"/>

    <Family Name="Patch" DiskId="2" SequenceStart="6" MediaSrcProp="SourceDirPath">
      <UpgradeImage Id="UpdateMsi" SourceFile=".\\MinorVersion1.2\\Sample.msi" >
        <TargetImage Id="BasisMsi" SourceFile=".\\MinorVersion1.0\\Sample.msi" />
      </UpgradeImage>
      <UpgradeImage Id="UpdateMsi" SourceFile=".\\MajorVersion1.2\\Sample.msi" >
        <TargetImage Id="BasisMsi" SourceFile=".\\MajorVersion1.1\\Sample.msi" />
      </UpgradeImage>
    </Family>

    <PatchSequence PatchFamily="MyFamily" Sequence="1.2.0.0" Supersede="yes" />
  </PatchCreation>
</Wix>

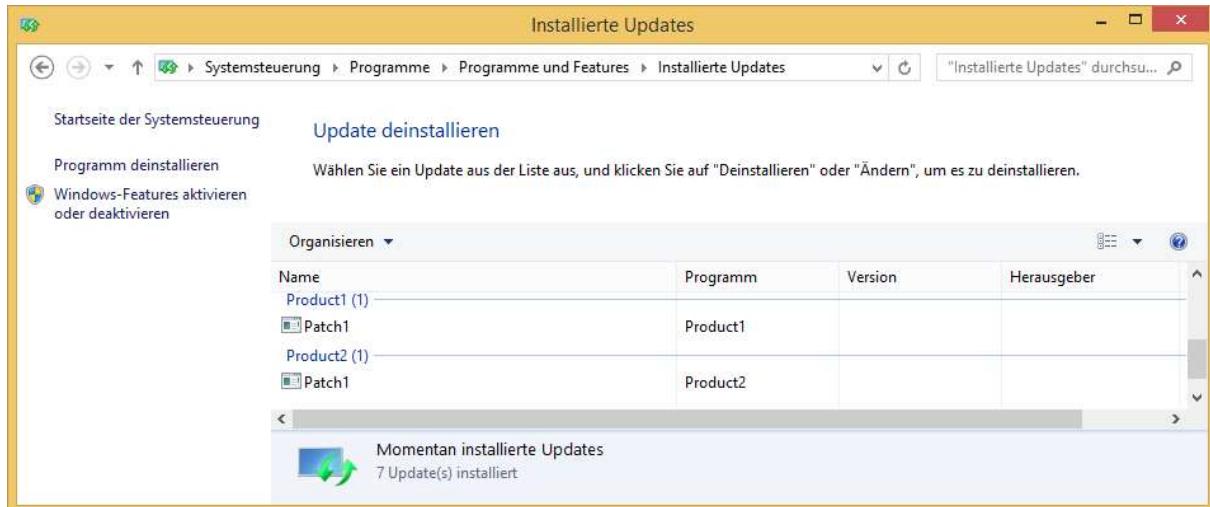
```

39.7.5 Erstellung eines Multi-SKU-Patches

Ein **Multi-SKU-Patch** wird immer dann eingesetzt, wenn unterschiedliche Produkte, die parallel auf einem PC koexistieren können, gemeinsame Dateien bzw. Komponenten besitzen. Stellen wir uns vor, wir hätten ein Textverarbeitungs- und ein Tabellenkalkulationsprogramm. Beide Programme werden über separate MSIs installiert und besitzen Dateien für die Rechtschreibprüfung. Um Speicherplatz zu sparen, installieren beide Produkte die Rechtschreibprüfung in dasselbe Zielverzeichnis.

Nun soll die Rechtschreibprüfung über einen Patch aktualisiert werden. Da sich bei der neuen Rechtschreibprüfung das Interface geändert hat, müssen beide Programme aktualisiert werden. Für diesen Anwendungsfall ist ein Multi-SKU-Patch genau die richtige Lösung. Ein Multi-SKU-Patch erkennt bei der Installation, dass beide Produkte installiert sind, und aktualisiert dann beide Produkte in einem Durchlauf.

In der Systemsteuerung wird dann derselbe Patch für beide Produkte separat aufgeführt:



Die PCP-Datei für den Patch wird hierbei mit folgender WXS-Datei erstellt:

```
<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <PatchCreation Id="YOURGUID" OutputPath="patch.pcp" WholeFilesOnly="no">
    <PatchInformation Description="Patch" Comments="Sample" Manufacturer="Sd" />

    <PatchMetadata AllowRemoval="yes" Description="Patch" ManufacturerName="Sd"
      TargetProductName="MySetup" Classification="Update"
      MoreInfoURL="http://www.sd-technologies.de/"
      DisplayName="Sample Patch"/>

    <Family Name="Product1" DiskId="2" SequenceStart="6" MediaSrcProp="SrcDirPath">
      <UpgradeImage Id="Update1" SourceFile=".\\Produkt1\\UpdateMsi\\Produkt1.msi" >
        <TargetImage Id="Basis1" SourceFile=".\\Produkt1\\BasisMsi\\Produkt1.msi" />
      </UpgradeImage>
    </Family>

    <Family Name="Product2" DiskId="2" SequenceStart="6" MediaSrcProp="SrcDirPath">
      <UpgradeImage Id="Update2" SourceFile=".\\Produkt2\\UpdateMsi\\Produkt2.msi" >
        <TargetImage Id="Basis2" SourceFile=".\\Produkt2\\BasisMsi\\Produkt2.msi" />
      </UpgradeImage>
    </Family>

    <PatchSequence PatchFamily="Produkt1" Sequence="2.1.0.0" />
    <PatchSequence PatchFamily="Produkt2" Sequence="1.2.0.0" />
  </PatchCreation>
</Wix>
```

Für jedes Produkt definieren wir eine separate Familie, in der wir dann die Basis- und Update-MSIs entsprechend angeben. Wie wir oben sehen, wurde auch für jedes Produkt eine eigene Patch-Sequenz definiert. Das muss man nicht machen, es wäre auch möglich, hier eine Patch-Sequenz nur für die Rechtschreibprüfung anzugeben.

39.8 User-Account-Control-Patching

Ab Windows Installer Version 3.0 können über das sogenannte **User-Account-Control-Patching** Patches ohne Administratorrechte eingespielt werden. Der Benutzer, der den Patch installiert, muss diesen nur aufrufen können. Alle anderen Rechte werden über den Windows Installer bezogen, der im Systemkonto ausgeführt wird und damit alle nötigen Rechte besitzt.

Für das User-Account-Control-Patching muss sowohl das Basis-MSI als auch der Patch mit demselben gültigen Zertifikat signiert sein. Zusätzlich muss es im Basis-MSI die Tabelle **MsiPatchCertificate** geben. In die MsiPatchCertificate wird die Zertifikatdatei eingetragen. Somit werden alle Patches, die mit diesem Zertifikat signiert wurden, ohne User-Account-Meldung installiert.

-
- **Hinweis:** Das User-Account-Control-Patching gilt nur für Workstation-Betriebssysteme und ist auf Server-Betriebssysteme nicht verfügbar.
-

Im WiX-Toolset wird die MsiPatchCertificate-Tabelle über das **PatchCertificates**-Element angegeben:

```
<!-- Enable UAC Patching -->
<PatchCertificates>
  <DigitalCertificate Id="MyCertificate"
    SourceFile=".\\SourceDir\\Certificate\\Wix Certificate.cer" />
</PatchCertificates>
```

Wie wir sehen, wird hier nur die cer-Datei angegeben. Damit das Zertifikat als gültiges Zertifikat angenommen wird, muss evtl. noch vorher das Stammzertifikat als *vertrauenswürdiges Stammzertifikat* installiert werden.

Der Patch kann, wie das MSI selbst, über **SignTool.exe** signiert werden:

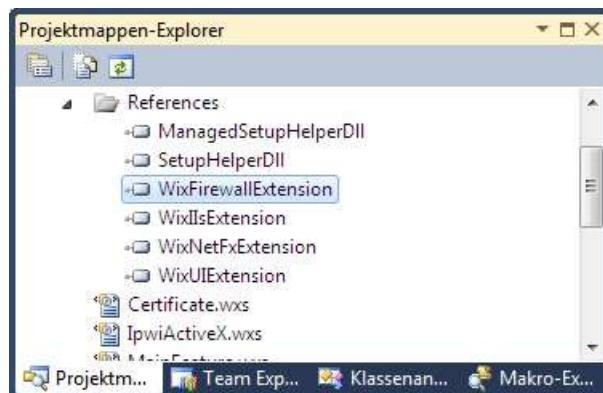
```
SignTool.exe sign /f "MyCertificate.pfx" Patch.msp
```

► **Hinweis:** Ein Administrator kann das User-Account-Control-Patching über die Gruppenrichtlinie DisableLUAPatching auf dem gesamten PC bzw. über das Windows Installer Property **MSIDISABLELUAPATCHING** für das MSI abschalten.

40 Windows-Firewall konfigurieren

Seit Windows XP hat Microsoft das Windows-Betriebssystem mit einer Firewall ausgestattet, die in den später erschienenen Betriebssystemversionen mit immer reichhaltigeren Funktionen aufwartet. Doch so gut Sicherheit auf der einen Seite auch ist, so hinderlich kann sie im Wege stehen. Will unsere Anwendung mit der Außenwelt kommunizieren, müssen wir bei der Firewall entsprechende Ausnahmen beantragen.

Ausnahmen können wir über die WiX-Erweiterung **WixFirewallExtension** einrichten. Dazu fügen wir eine Referenz auf die WixFilewallExtension.dll ein:



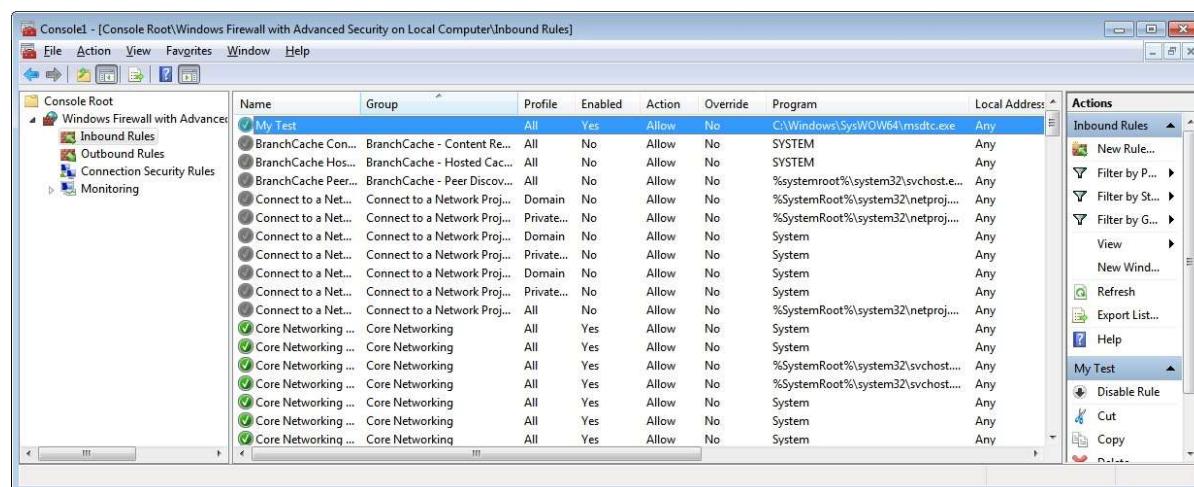
Nun müssen wir das Schema noch entsprechend erweitern, sodass die Elemente der Firewall-Extension über das Präfix **fw:** erreichbar sind:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
      xmlns:fw="http://schemas.microsoft.com/wix/FirewallExtension">
```

Über das Element **FirewallException**, das ein Child-Element von Component ist, können wir nun eine Ausnahme für eine Anwendung (hier für die msdtc.exe) erstellen:

```
<Component Id="Firewall" Guid="YOURGUID" Directory="INSTALLDIR" KeyPath="yes">
  <fw:FirewallException Id="Msdtc" Program="[SystemFolder]msdtc.exe" Name="My Test"
    Scope="any" IgnoreFailure="yes"/>
</Component>
```

Wird diese Komponente installiert, kann man die Ausnahme bei den eingehenden Regeln (Inbound Rules) der Firewall-Einstellungen sehen:



Benutzt man das Attribut *Program* sowie *Port* oder *Protokoll* in derselben Firewall-Exception, wird das Setzen der Ausnahmen bei den Betriebssystemen Windows XP und Windows Server 2003 mit einem Fehler und somit mit einem Rollback beendet. Das Setzen von Attribut *IgnoreFailure="yes"* verhindert den Abbruch – die Ausnahme wird jedoch bei diesen Betriebssystemen nicht gesetzt.

Besser wäre es, für die entsprechenden Betriebssysteme separate Einstellungen in separaten Komponenten zu erstellen. Über Bedingungen in den Komponenten stellt man sicher, dass die passende Komponente für das richtige Betriebssystem installiert wird.

Soll ein bestimmter Port, wie z. B. der Port 1234 für den Microsoft-SQL-Server, geöffnet werden, so kann man das mit folgenden Zeilen tun:

```
<Component Id="SqlException" Directory="INSTALLDIR" Guid="YOURGUID">
    <fw:FirewallException Id="SqlServerException" Name="Sql Exception" Port="1234"
        Protocol="tcp" Scope="any" IgnoreFailure="yes" />
</Component>
```

► **Hinweis:** Wenn der Firewall-Dienst deaktiviert ist, kann es vorkommen, dass das Setzen der Exception einen Fehler und so einen Rollback des Setups hervorruft. Deshalb ist es zu empfehlen, das Attribut *IgnoreFailure* immer auf *yes* zu setzen.

Wenn das Setup in einer anderen Sprache als in Englisch erstellt wird, kann es sein, dass der Compiler mit folgenden Fehlermeldungen abbricht:

Fehlerliste			
	7 Fehler	0 Warnungen	0 Meldungen
1	The localization variable !(loc.WixSchedFirewallExceptionsInstall) is unknown. Please ensure the variable is defined.	FirewallExtension_Platform.wxi	15
2	The localization variable !(loc.WixSchedFirewallExceptionsUninstall) is unknown. Please ensure the variable is defined.	FirewallExtension_Platform.wxi	16
3	The localization variable !(loc.WixRollbackFirewallExceptionsInstall) is unknown. Please ensure the variable is defined.	FirewallExtension_Platform.wxi	17
4	The localization variable !(loc.WixExecFirewallExceptionsInstall) is unknown. Please ensure the variable is defined.	FirewallExtension_Platform.wxi	18
5	The localization variable !(loc.WixRollbackFirewallExceptionsUninstall) is unknown. Please ensure the variable is defined.	FirewallExtension_Platform.wxi	19
6	The localization variable !(loc.WixExecFirewallExceptionsUninstall) is unknown. Please ensure the variable is defined.	FirewallExtension_Platform.wxi	20
7	The localization variable !(loc.msierrFirewallCannotConnect) is unknown. Please ensure the variable is defined.	FirewallExtension.wxi	14

Das liegt daran, dass bestimmte Texte nicht in den Localisation-Dateien gefunden werden. Diese Texte müssen in einer eigenen Datei deklariert werden:

```
<?xml version="1.0" encoding="utf-8"?>
<WixLocalization Culture="de-de"
    xmlns="http://schemas.microsoft.com/wix/2006/localization">
    <String Id="msierrFirewallCannotConnect">
        Fehler beim verbinden mit der WindowsFirewall. ([2] [3] [4] [5])
    </String>
    <String Id="WixSchedFirewallExceptionsInstall">
        Windows Firewall wird konfiguriert
    </String>
    <String Id="WixSchedFirewallExceptionsUninstall">
        Windows Firewall wird konfiguriert
    </String>
    <String Id="WixRollbackFirewallExceptionsInstall">
        Änderungen in der Windows Firewall werden rückgängig gemacht
    </String>
```

```
<String Id="WixExecFirewallExceptionsInstall">
    Installiere die Windows Firewall Konfiguration
</String>
<String Id="WixRollbackFirewallExceptionsUninstall">
    Änderungen in der Windows Firewall werden rückgängig gemacht
</String>
<String Id="WixExecFirewallExceptionsUninstall">
    Deinstalliere die Windows Firewall Konfiguration
</String>
</WixLocalization>
```

41 Merge-Module

Wie wir in den vergangenen Lektionen gesehen haben, helfen uns Fragmente dabei, ein großes Paket in überschaubare Einheiten zu verteilen und ermöglichen es, dass ein einmal erstellter Code wiederverwendet werden kann. Daher passen Fragmente am besten zur innerbetrieblichen Entwicklung.

Was aber, wenn Teile des Setups (z. B. Treiber oder Runtime-Umgebungen) von Dritten verwendet werden sollen? WiX-Fragmente sind hier wohl nicht der richtige Weg denn wir wissen nicht, ob die dritte Partei auch WiX zur Erstellung von Setup benützt.

Microsoft stand bei der Entwicklung vom Windows Installer vor demselben Problem – lange bevor es WiX überhaupt gab. Deshalb gibt es noch einen anderen Mechanismus – den Windows Installer Weg - der es zulässt, Module für den Gebrauch durch Dritte zu erstellen: Die Rede ist von **Merge-Modulen**.

Auch wenn wir selbst keine Bibliotheken an Dritte weitergeben ist es durchaus angebracht, sich mit Merge-Modulen zu beschäftigen. Fast jedes Programm wird heutzutage über eine höhere Programmiersprache programmiert. Die Programmiersprache selbst benötigt in den meisten Fällen selbst Runtime-Umgebungen, die wir natürlich mit in unserem Setup ausliefern wollen. Und fast alle Programmiersprachen stellen diese Runtime-Bibliotheken in Form von Merge-Modulen bereit. Die Merge-Module binden wir einfach in unserem Setup ein, wobei wir uns keine Gedanken darüber machen müssen, welche Dateien das nun genau sind und wie und wohin sie installiert werden.

Uns wird es jetzt nicht mehr wundern, dass Microsoft mit dem Visual Studio auch Runtime-Bibliotheken mitbringt. Diese Dateien sind im „Gemeinsame-Dateien-Verzeichnis“ im Unterverzeichnis „Merge Modules“ abgelegt. Das Internet bietet auch sehr viele Seiten, auf denen Merge-Module verfügbar sind. Neben der Microsoft-Download-Seite selbst findet man viele Module z. B. auch auf <http://www.installsite.org> (wir sagen „Danke“ an Stefan Krüger).

Doch was ist ein Merge-Module überhaupt? Ein Merge-Module ist eine vereinfachte MSI-Datenbank, die Setup-Informationen, Dateien, Registry-Einträge und andere Daten enthalten kann. Beim Build-Prozess werden die Tabellen des Merge-Moduls durch die Light.exe einfach in das MSI-Paket kopiert. Somit hat das MSI genau die Informationen, die im Merge-Module festgehalten sind.

► Hinweis: Da Merge-Module MSI-Datenbanken sind, können wir sie auch mit Orca öffnen.

41.1 Merge-Module einbinden

Als Beispiel wollen wir die Microsoft Foundation Classes von Visual Studio 6 ins Setup einbinden. Die Runtime-Bibliothek, die wir brauchen ist die Mfc42.dll und deren **Abhängigkeiten**. Und welche DLLs sind von der Mfc42.dll abhängig?

Ein Blick mit Orca in die **ModuleDependency**-Tabelle des Merge-Moduls mfc42.msm gibt hierüber Aufschluss:

ModuleID	ModuleLangu...	RequiredID
MFC42.51D569E2_8A28_11D2_B962_006097C4DE24	0	MSVCRT.51D569E0_8A28_11D2_B962_006097C4DE24
MFC42.51D569E2_8A28_11D2_B962_006097C4DE24	0	COMCAT.3207D1B0_80E5_11D2_B95D_006097C4DE24
MFC42.51D569E2_8A28_11D2_B962_006097C4DE24	0	OLEAUT32.8C0C59A0_7DC8_11D2_B95D_006097C4DE24

Wir sehen, dass die MFC42 DLL Abhängigkeiten von msrvct.dll, von comcat.dll und von oleaut32.dll aufweist. Für diese Dateien sind auch Merge-Module verfügbar. Eventuell könnten diese Merge-Module auch wieder Abhängigkeiten aufweisen – wir müssen also auch diese prüfen und so weiter.

Über das WiX-Element **Merge** binden wir die Merge-Module in unser WiX-Skript ein. Das Merge-Element hat als Elternelement entweder das Directory oder DirectoryRef Element. Der Elternteil bestimmt, wohin die Dateien des Merge-Moduls installiert werden: Das Verzeichnis wird also in das Merge-Module hineingereicht. Ob das Verzeichnis auch tatsächlich verwendet wird oder nicht, ist im Merge-Module selbst verankert. Manche Merge-Module installieren in fest vorgegebenen Verzeichnissen und ignorieren diese Vorgabe:

```
<DirectoryRef Id="TARGETDIR">
    <Merge Id="MFC42" SourceFile=".\\SourceDir\\MergeModule\\Mfc42.msm"
        DiskId="1" Language="0"/>
    <Merge Id="msvcrt" SourceFile=".\\SourceDir\\MergeModule\\msvcrt.msm"
        DiskId="1" Language="0"/>
    <Merge Id="comcat" SourceFile=".\\SourceDir\\MergeModule\\comcat.msm"
        DiskId="1" Language="0"/>
    <Merge Id="oleaut32" SourceFile=".\\SourceDir\\MergeModule\\oleaut32.msm"
        DiskId="1" Language="0"/>
</DirectoryRef>
```

Das Merge-Module fügen wir nun in das selbe Feature ein, in der sich auch die Anwendung befindet, die diese Dateien benötigt. Die Zuweisung geschieht über das **MergeRef** Element:

```
<Feature Id="ProductFeature" ...>
    <ComponentGroupRef Id ="MainFeature" />
    <MergeRef Id="MFC42"/>
    <MergeRef Id="msvcrt"/>
    <MergeRef Id="comcat"/>
    <MergeRef Id="oleaut32"/>
</Feature>
```

Wenn wir das Setup erstellen, haben wir alle benötigten DLLs eingebunden. Aber speziell bei diesen Merge-Modulen bekommen wir auch ein paar Fehlermeldungen:

Fehlerliste				
		3 Fehler	5 Warnungen	0 Meldungen
Beschreibung		Datei	Zeile	Spalte
		Projekt		
✖ 5	ICE03: Table: MIME Column: ContentType Missing specifications in _Validation Table (or Old Database)	MyFirstSetup.msi	0	1
✖ 6	ICE03: Table: MIME Column: Extension_Missing specifications in _Validation Table (or Old Database)	MyFirstSetup.msi	0	1
✖ 7	ICE03: Table: MIME Column: CLSID Missing specifications in _Validation Table (or Old Database)	MyFirstSetup.msi	0	1

Die Warnung ICE03 besagen, dass wir in unserem MSI-File die **MIME**-Tabelle eingebunden haben, aber es keine Beschreibung in der **_Validation**-Tabelle gibt. Die MIME-Tabelle wird als leere Tabelle in unser MSI-File eingefügt. Das daher, weil die Tabelle in dem Merge-Modul vorhanden ist und beim Linken einfach in unser MSI kopiert wird. Da das Merge-Modul aber keine Einträge in der **_Validation**-Tabelle hat, entsteht dieser Fehler.

Aber wofür ist die **_Validation**-Tabelle überhaupt gut? Die **_Validation**-Tabelle beschreibt die einzelnen Spalten einer Tabelle und definiert also, welche Spalten der Primary-Key einer Tabelle hat, welche Werte zulässig sind und welche Spalten auf andere Tabellen referenzieren. Diese Tabelle ist für die **Validierung** da, die automatisch nach dem Build-Prozess ausgeführt wird. Anhand dieser Tabelle kann dann die Validierung prüfen, ob alle Referenzen aufgelöst sind und ob nicht erlaubte Werte in der Tabelle stehen.

Über das Element **EnsureTable** können wir sicherstellen, dass diese Tabelle sowie die `_Validation` Einträge von WiX erstellt werden:

```
<EnsureTable Id="MIME"/>
```

Wenn wir diese Zeile in das WiX-Skript eintragen, haben wir in unserem MSI folgende Einträge in der `_Validation`-Tabelle:

Table	Column	Nullable	MinValue	MaxValue	KeyTable	KeyColu...	Category	Set	Description
ModuleDependency	MIME	Y					Guid		Optional associated CLSID.
ModuleSignature	MIME	N					Text		Primary key. Context identifier, typically "type/format"
MsfileHash	MIME	N			Extension	1	Text		Optional associated extension (without dot)
Progid	Media	Y					Property		The property defining the location of the cabinet file.
Property	Media	N	1	32767					Primary key, integer to determine sort order for table.
RadioButton	Media	N	0	2147483647					File sequence number for the last file for this media.
RegLocator	Media	Y					Text		Disk name: the visible text actually printed on the disk.
Registry	Media	Y					Cabinet		If some or all of the files stored on the media are comp
RemoveFile	Media	Y					Text		The label attributed to the volume.
Shortcut	ModuleAdmi...	Action	N				Identifier		Action to insert
Signature	ModuleAdmi...	Condition	Y				Condition		
TextStyle	ModuleAdmi...	Sequence	Y	-4	32767				Standard Sequence number
UIText	ModuleAdmi...	After	Y	0	1				Before (0) or After (1)
Verb	ModuleAdmi...	BaseAction	Y		ModuleAdm...	1	Identifier		Base action to determine insert location.
Validation									

41.2 Merge-Module erstellen

Im folgenden Abschnitt wollen wir sehen, wie selbst ein Merge-Module erstellen können. Dafür erstellen wir ein neues Projekt über die Windows Installer XML-Vorlage „Merge Module Project“.

Das Template von diesem Projekt sieht in etwa so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<WiX xmlns="http://schemas.microsoft.com/WiX/2006/wi">
  <Module Id="MyMergeModule" Language="1033" Version="1.0.0.0">
    <Package Id="YOURGUID" Manufacturer="MyMergeModule" InstallerVersion="200" />

    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="MergeRedirectFolder">
        </Directory>
      </Directory>
    </Module>
  </WiX>
```

Da die Sprache zwischen dem einbindenden Setup und dem einzubindenden Merge-Module gleich sein muss, definieren wir unser Merge-Module als sprachunabhängig. Dies machen wir, indem wir beim Attribut Language den Wert 0 angeben.

```
<Module Id="MergeModule1" Language="0" Version="1.0.0.0" Codepage="1252">
  <Package Id=" YOURGUID " Manufacturer="MergeModule1" InstallerVersion="200" />
```

41.2.1 Abhängigkeiten

Soll das Merge-Module Abhängigkeiten von anderen Merge-Modulen aufweisen, können wir dies über das Element **Dependency** erreichen:

```
<Dependency RequiredId="MFC42.51D569E2_8A28_11D2_B962_006097C4DE24"
  RequiredLanguage="0"/>
```

Als Required-Id geben wir hier die Modul-ID des Merge-Moduls an. Diese kann einfach über Orca ermittelt werden, indem das abhängige Merge-Modul geöffnet und die Modul-ID aus der Tabelle **ModuleSignature** entnommen wird:

The screenshot shows the Orca tool interface with the title bar "mfc42.msm - Orca". The menu bar includes File, Edit, Tables, Transform, Tools, View, and Help. The toolbar has icons for Open, Save, Print, and others. On the left, a tree view under "Tables" shows "ModuleSignature" selected, highlighted with a yellow background. The main pane displays a table with one row. The columns are "ModuleID", "Langu...", and "Version". The data row contains "MFC42.51D569E2_8A28_11D2_B962_006097C4DE24", "0", and "6.0.8665.0". At the bottom, it says "Tables: 14" and "ModuleSignature - 1 row". A status bar at the bottom right says "No column is selected."

ModuleID	Langu...	Version
MFC42.51D569E2_8A28_11D2_B962_006097C4DE24	0	6.0.8665.0

41.2.2 Festes Zielverzeichnis definieren

Den Installationsort, in dem ein Merge-Modul Dateien ablegt, können wir entweder im Merge-Modul selbst festlegen oder das einbindende Programm bestimmt diesen Zielort. Einen fest vorgegebenen Installationsort definieren wir über folgende Zeilen:

```
<!-- Über IgnoreModularization definieren wir, dass an die Directory Variable  
SystemFolder kein Modularisierungs-GUID angehängt werden soll -->  
<IgnoreModularization Name="SystemFolder" Type="Directory"/>  
  
<Directory Id="TARGETDIR" Name="SourceDir">  
    <Directory Id="SystemFolder" SourceName="SystemFolder" />  
</Directory>
```

Das Element **IgnoreModularization** bestimmt, dass für das in Name angegebene Element keine Modularisierung vorgenommen wird. Modularisierung bedeutet, dass jedem Primary-Key im Merge-Modul automatisch die Package-Id des Merge-Moduls angehängt wird. Aus MyKey wird somit MyKey.56FDDF1F_4BA1_4381_90DD_091A1CF897AF. Das ist auch sinnvoll, da ein Merge-Modul über Light.exe in das MSI kopiert wird. Die Modularisierung stellt hierbei sicher, dass die Primary-Keys auch tatsächlich eindeutig sind. Gäbe es im Hauptsetup bereits einen Eintrag mit dem Namen MyKey, würde der vorhandene Eintrag mit dem Eintrag aus dem Merge-Modul überschrieben.

Würden wir hier die Modularisierung nicht ausschalten, würden wir hier ein neues Directory-Element namens SystemFolder.56FDDF1F_4BA1_4381_90DD_091A1CF897AF erstellen. Da wir aber die Windows Installer Variable SystemFolder benutzen wollen, müssen wir die Modularisierung ausschalten.

Nun können wir eine Komponente erstellen, die ins Systemverzeichnis eine Datei installiert:

```
<!-- Das Ziel der Komponente ist das System-Verzeichnis -->  
<Component Id="MmDemo.dll" Guid="YOURGUID" Directory="SystemFolder"  
    SharedDllRefCount="yes">  
    <File Id="MmDemo.dll" Source=".\\SourceDir\\MmDemo.dll" KeyPath="yes" />  
</Component>
```

► **Hinweis:** IgnoreModularization ist eigentlich ein Ausdruck, der durch das Attribut SuppressModularization ersetzt wurde. Da das WiX-Element Directory dieses Attribut (noch) nicht kennt, muss es im WiX Toolset 3.9 noch über den veralteten Ausdruck IgnoreModularization definiert werden.

41.2.3 Dynamische Zielverzeichnisse definieren

Soll das Zielverzeichnis vom Hauptsetup definiert werden, sieht das Merge-Modul etwas einfacher aus:

```
<Directory Id="TARGETDIR" Name="SourceDir">
    <Directory Id="MergeRedirectFolder" />
</Directory>

<!-- Das Ziel der Komponente kann vom Hauptsetup bestimmt werden -->
<Component Id="MergeModule.txt" Guid="YOURGUID" Directory="MergeRedirectFolder">
    <File Id="MergeModule.txt" Source=".\\SourceDir\\MergeModule.txt" KeyPath="yes" />
</Component>
```

Im Setup wird dann dem Root-Verzeichnis TARGETDIR das tatsächliche Zielverzeichnis des Merge-Moduls zugeordnet (hier z. B. INSTALLDIR):

```
<DirectoryRef Id="INSTALLDIR">
    <Merge Id="MyMergeModule" SourceFile=".\\SourceDir\\MergeModule\\ MyMergeModule.msm"
        DiskId="1" Language="0"/>
</DirectoryRef>
```

41.2.4 Das Merge-Modul im Überblick

Hier nochmals das gesamte WiX-Skript im Überblick:

```
<?xml version="1.0" encoding="UTF-8"?>
<WiX xmlns="http://schemas.microsoft.com/WiX/2006/wi">
    <Module Id="MergeModule1" Language="0" Version="1.0.0.0" Codepage="1252">
        <Package Id="YOURGUID" Manufacturer="MergeModule1" InstallerVersion="200" />

        <!-- Über IgnoreModularization definieren wir, dass an die Directory Variable
            SystemFolder kein Modularisierungs-GUID angehängt werden soll -->
        <IgnoreModularization Name="SystemFolder" Type="Directory"/>

        <Directory Id="TARGETDIR" Name="SourceDir">
            <Directory Id="SystemFolder" SourceName="SystemFolder" />
            <Directory Id="MergeRedirectFolder" />
        </Directory>

        <!-- Das Ziel der Komponente kann vom Hauptsetup bestimmt werden -->
        <Component Id="MergeModule.txt" Guid="YOURGUID" Directory="MergeRedirectFolder">
            <File Id="MergeModule.txt" Source=".\\SourceDir\\MergeModule.txt" KeyPath="yes" />
        </Component>

        <!-- Das Ziel der Komponente ist das System-Verzeichnis -->
        <Component Id="MmDemo.dll" Guid="YOURGUID" Directory="SystemFolder"
            SharedDllRefCount="yes">
            <File Id="MmDemo.dll" Source=".\\SourceDir\\MmDemo.dll" KeyPath="yes" />
        </Component>
    </Module>
</WiX>
```

42 Das WiX-Projekt auf einem Build-Server erstellen

Soll das Setup auf einem **Build-Server** als **Daily-Build** integriert werden, braucht dort weder das Visual Studio noch das WiX-Toolset installiert sein. Durch ein paar Handgriffe kann die Solution-Datei (SLN-Datei) über **MsBuild.exe** erstellt werden. Da die MsBuild.exe Bestandteil des .NET-Frameworks ist, reicht die Installation von .NET auf dem Build-Server aus.

42.1 Buildprozess ohne installiertem WiX-Toolset

Möchte man auf dem Build-Server das WiX-Toolset nicht installieren, laden wir nur die Binaries des Toolsets herunter. Die Binaries entpacken wir entweder an einer zentralen Stelle oder in einem Verzeichnis unterhalb des Projektes. Eventuell ist es sogar sinnvoll, die Binaries in der Quellcode-Verwaltung einzuchecken. Somit ist ein reproduzierbares Ergebnis garantiert.

Damit MSBuild beim Kompilieren das WiX-Toolset findet, müssen wir ein paar Variablen im Projekt definieren. Diese sagen dem Buildprozess, wo die Binaries des Toolsets abgelegt sind. Das machen wir in der Projektdatei (.Wixproj-Datei) über eine neue PropertyGroup, die wie folgt aufgebaut ist:

```
<PropertyGroup Condition=" '$(Wix)' == '' ">
  <WixToolPath>.\wix_binaries\</WixToolPath>
  <WixTargetsPath>$(WixToolPath)Wix.targets</WixTargetsPath>
  <WixTasksPath>$(WixToolPath)wixtasks.dll</WixTasksPath>
</PropertyGroup>
```

Die oben angegebenen Zeilen sagen dem Compiler, dass die Binaries im Unterverzeichnis *wix_binaries* unterhalb der Projektdatei liegen. Die Bedingung '\$(Wix)' == "" prüft, ob die Umgebungsvariable Wix gesetzt ist (diese wird bei der Installation des WiX Toolsets gesetzt) und verhindert, dass die PropertyGroup bei installiertem WiX-Toolset berücksichtigt wird. Ein Absturz von Visual Studio wäre u. U. ansonsten die Folge.

42.2 Buildprozess über MsBuild.exe starten

Egal ob das WiX-Toolset installiert oder die Projektdatei über die oben angegebenen Zeilen angepasst wurden können wir nun über MsBuild.exe und der Solution das Setup erstellen. Wir verwenden hier die MsBuild.exe vom .NET Framework 4.0, bei einem anderen Framework muss der Pfad entsprechend angepasst werden:

```
"%windir%\Microsoft.NET\Framework\v4.0.30319\msbuild.exe" "MySetup.sln" /t:Rebuild
```

Sind mehrere Konfigurationen in der Solution enthalten, kann man diese beim Aufruf von MsBuild.exe über den Parameter */p:Configuration* angeben:

```
msbuild.exe "MySetup.sln" /p:Configuration="Release"
```

Soll auch noch die Plattform angegeben werden, dann sieht das so aus:

```
msbuild.exe "MySetup.sln" /p:Configuration="Release" /p:Platform="x86"
```

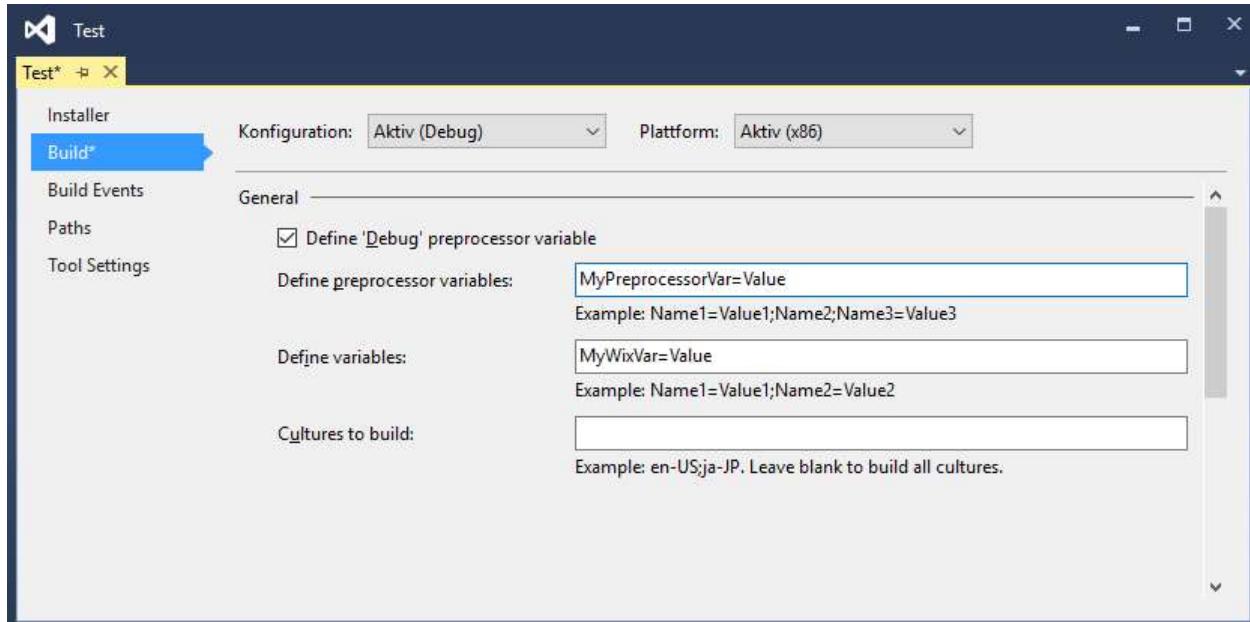
Alternativ zur Solution-Datei kann auch die Projektdatei angegeben werden:

```
msbuild.exe "MyProject.wixproj" /t:Rebuild
```

42.3 Variable dem Buildprozess übergeben

In manchen Fällen ist die Angabe der Konfiguration bzw. der Plattform nicht ausreichend. Möchte man noch weitere Parameter per Kommandozeile angeben können, dann greift man auf **Präprozessor-** bzw. auf **WixVariablen** zurück.

Diese Variablen werden im WiX-Projekt unter dem Reiter *Build* definiert:



Entlädt man das Projekt und schaut sich dieses mit dem XML-Editor an, dann sieht man, dass die Variablen in einer PropertyGroup angegeben werden:

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|x86' ">
  <OutputPath>bin\$(Configuration)\</OutputPath>
  <IntermediateOutputPath>obj\$(Configuration)\</IntermediateOutputPath>
  <DefineConstants>Debug;MyPrepVar=Value</DefineConstants>
  <WixVariables>MyWixVar=Value</WixVariables>
</PropertyGroup>
```

Präprozessorvariablen werden über *DefineConstants* und WixVariablen über *WixVariables* angegeben. Um nun diese Variablen über den Buildprozess zu steuern, verweisen wir an dieser Stelle auf die Build-Variablen MyPreVar bzw. MyWixVar:

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|x86' ">
  ...
  <DefineConstants>Debug;MyPreVar=$(MyPreVar)</DefineConstants>
  <WixVariables>MyWixVar=$(MyWixVar)</WixVariables>
</PropertyGroup>
```

Wie wir sehen verweist man auf Build-Variablen über ein vorangestelltes Dollarzeichen und dem in Klammern eingeschlossenen Variablennamen. Um sicher zu stellen, dass beide Build-Variablen immer einen Sinnvollen Wert haben, geben wir einen Defaultwert an. Den Defaultwert tragen wir in einer PropertyGroup ein, die keine Bedingung hat:

```
<PropertyGroup>
  <MyPreVar Condition=" '$(MyPreVar)' == '' ">MyPreDefault</MyPreVar>
  <MyWixVar Condition=" '$(MyWixVar)' == '' ">MyWixDefault</MyWixVar>
</PropertyGroup>
```

Da die Build-Variable selbst mit einer Bedingung versehen ist, wird der Defaultwert nur dann gesetzt, wenn über die Kommandozeile kein anderer Wert angegeben wurde. Die Build-Variablen MyPreVar und MyWixVar geben wir wie die Konfiguration bzw. Plattform über den Parameter /p bei MsBuild.exe an:

```
msbuild.exe "MySetup.sln" /p:MyPreVar="MyNewPreValue" /p:MyWixVar="MyNewWixValue"
```

Kombinationen mit Konfiguration bzw. Plattform sind natürlich jederzeit möglich.

43 MSI in WXS konvertieren

Bisher haben wir gesehen, wie man aus WXS-Skripten ein MSI (oder MSM) erstellt. Doch manchmal ist auch der umgekehrte Weg interessant. Auch hierfür bietet das WiX-Toolset etwas: Das Programm **dark.exe**. Dark wird über folgende Kommandozeile aufgerufen:

```
dark.exe Setup.msi Setup.wxs
```

Enthält das MSI leere Tabellen, so werden diese über das oben angegebene Kommando nicht mit in das WiX-Script übernommen. Sollen diese auch per **EnsureTable**-Element mit aufgenommen werden, so ruft man dark.exe mit dem Parameter **-sdet** auf:

```
dark.exe Setup.msi Setup.wxs -sdet
```

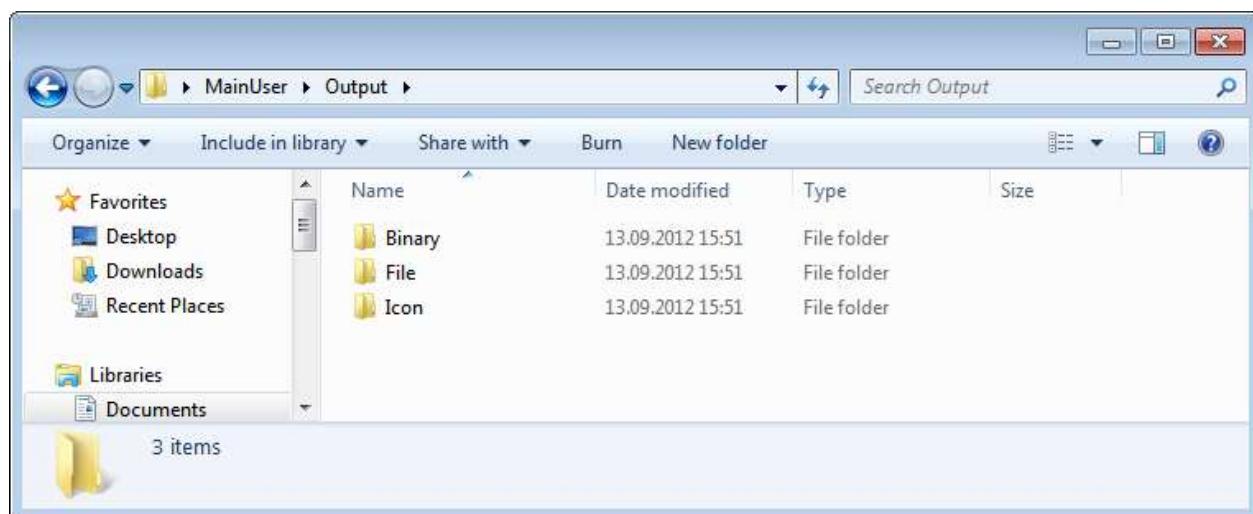
Ist man nur an den Komponenten mit den zugehörigen Dateien und Registry-Einträgen und nicht am Userinterface interessiert, dann hilft uns Parameter **-sui** weiter.

```
dark.exe Setup.msi Setup.wxs -sui
```

Enthält das zu dekomprimierende Setup Dateien in der Binärtabelle bzw. in Kabinettdateien, so können diese über den Parameter **-x** entpackt werden:

```
dark.exe Setup.msi Setup.wxs -x "<OUTPUTPATH>"
```

Im angegebenen Ausgabeverzeichnis werden dann mehrere Ordner angelegt, die dann die Dateien enthalten:



44 Anhang

A. Standard-Verzeichnisvariablen des Windows Installers

Um die Zielverzeichnisse mit den Standard-Verzeichnisvariablen des Windows Installers richtig setzen zu können, muss man zuerst wissen, wohin diese Variablen zeigen. Die unten dargestellten Pfade beziehen sich auf ein Windows-10-System:

Variable	Beschreibung
AdminToolsFolder	Zeigt auf das Verzeichnis, in dem administrative Tools abgelegt werden. Diese Variable wird entsprechend dem Property ALLUSERS auf „C:\users\[LogonUser]\Start Menu\Programs\Administrative Tools“ (Installation für den aktuell angemeldeten Benutzer) bzw. „C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Administrative Tools“ (Installation für alle Benutzer) gesetzt.
AppDataFolder	Dieses Property zeigt auf den Pfad, in dem Applikationsdaten des aktuell angemeldeten Benutzers gespeichert werden. Bei einem Windows-10-System ist dies in der Regel „C:\users\[LogonUser]\AppData\Roaming“.
CommonAppDataFolder	Dieses Property zeigt auf den Pfad, in dem allgemeine Applikationsdaten für alle Benutzer gespeichert werden. Bei einem Windows-10-System ist dies in der Regel „C:\ProgramData“.
CommonFilesFolder	Zeigt auf das Verzeichnis, in dem gemeinsam benutzte Anwendungsdateien gespeichert werden. Bei einem Windows-10-System ist dies in der Regel „C:\Program Files\Common Files“.
DesktopFolder	Dieses Property zeigt auf den Pfad, in dem der Desktop gespeichert ist. Diese Variable wird entsprechend dem Property ALLUSERS auf „C:\users\[LogonUser]\Desktop“ (Installation für den aktuell angemeldeten Benutzer) bzw. „C:\Users\Public\Desktop“ (Installation für alle Benutzer) gesetzt.
FavoritesFolder	Dieses Property zeigt auf den Pfad, in dem die Favoriten des aktuell angemeldeten Benutzers gespeichert werden. Bei einem Windows-10-System ist dies in der Regel „C:\users\[LogonUser]\Favorites“.
FontsFolder	In diesem Ordner werden die Schriftarten eines Windows-Systems gespeichert. Bei einem Windows-10-System ist dies „C:\Windows\Fonts“.
LocalAppDataFolder	Dieses Property zeigt auf den Pfad, in dem die benutzerbezogenen Applikationsdaten (für Non-roaming-Benutzer) gespeichert werden. Bei einem Windows-10-System ist dies „C:\Users\[LogonUser]\AppData\Local“.
MyPicturesFolder	In diesem Verzeichnis werden die Bilder des aktuell angemeldeten Benutzers gespeichert. Bei einem Windows-10-System ist dies in der Regel „C:\users\[LogonUser]\Pictures“.
NetHoodFolder	Dieses Property zeigt auf den Pfad, in dem die Netzwerkumgebung des aktuell angemeldeten Benutzers gespeichert ist. Bei einem Windows-10-System ist dies: „C:\Users\[LogonUser]\AppData\Roaming\Microsoft\Windows\Network Shortcuts“.

Variable	Beschreibung
PersonalFolder	Dieses Property zeigt auf den Pfad, in dem die eigenen Dateien des angemeldeten Benutzers gespeichert werden. Bei einem Windows-10-System ist dies: "C:\users\[LogonUser]\Documents\".
PrintHoodFolder	Dieses Property zeigt auf den Pfad, in dem die Druckumgebung des aktuell angemeldeten Benutzers gespeichert ist. Bei einem Windows-10-System ist dies: "C:\Users\[LogonUser]\AppData\Roaming\Microsoft\Windows\Printer Shortcuts\".
ProgramFilesFolder	Dieses Property zeigt auf den Pfad, in dem die Programme abgelegt werden. Bei einem Windows-10-System ist dies: "C:\Program Files\".
ProgramMenuFolder	Dieses Property zeigt auf den Pfad, in dem der Programme-Ordner des Startmenüs gespeichert ist. Diese Variable wird entsprechend dem Property ALLUSERS auf „C:\Users\[LogonUser]\AppData\Roaming\Microsoft\Windows\Start Menu\Programs“ (Installation für den aktuell angemeldeten Benutzer) bzw. „C:\ProgramData\Microsoft\Windows\Start Menu\Programs“ (Installation für alle Benutzer) gesetzt.
SendToFolder	Dieses Property zeigt auf den Pfad des „Senden an“-Ordners. Bei einem Windows-10-System ist dies: „C:\Users\[LogonUser]\AppData\Roaming\Microsoft\Windows\SendTo“.
StartMenuFolder	Dieses Property zeigt auf den Pfad, in dem das Startmenü gespeichert ist. Diese Variable wird entsprechend dem Property ALLUSERS auf „C:\Users\[LogonUser]\AppData\Roaming\Microsoft\Windows\Start Menu“ (Installation für den aktuell angemeldeten Benutzer) bzw. „C:\ProgramData\Microsoft\Windows\Start Menu“ (Installation für alle Benutzer) gesetzt.
StartupFolder	Dieses Property zeigt auf den Pfad, in dem das Startmenü gespeichert ist. Diese Variable wird entsprechend dem Property ALLUSERS auf „C:\Users\[LogonUser]\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Autostart“ (Installation für den aktuell angemeldeten Benutzer) bzw. „C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup“ (Installation für alle Benutzer) gesetzt.
SystemFolder	Dieses Property zeigt auf den Pfad, in dem 32-Bit-DLLs gespeichert sind. Bei einem Windows-10-System ist dies „C:\Windows\System32“ (32 Bit) bzw. „C:\Windows\SysWOW64“ (64 Bit).
TempFolder	Dieses Property zeigt auf den Pfad, in dem temporäre Dateien abgelegt werden. Bei einem Windows-10-System ist dies in der Regel: "C:\users\[LogonUser]\ AppData\Local\Temp\".
WindowsFolder	Dieses Property zeigt auf den Pfad, in dem Windows abgelegt ist. Bei einem Windows-10-System ist dies: "C:\Windows\".
WindowsVolume	Dieses Property zeigt auf das Laufwerk, auf dem Windows installiert ist. Bei einem Windows-10-System ist dies: „c:\“.

B. Projekt-Referenz-Variablen

Folgende Projekt-Referenz-Variablen stehen zur Verfügung:

Variablenname	Beispiel	Wert
var.ProjectName.Configuration	\$(var.MyProject.Configuration)	Debug oder Release
var.ProjectName.FullConfiguration	\$(var.MyProject.FullConfiguration)	Debug AnyCPU
var.ProjectName.Platform	\$(var.MyProject.Platform)	AnyCPU, Win32, x64 oder ia64
var.ProjectName.ProjectDir	\$(var.MyProject.ProjectDir)	C:\Project\
var.ProjectName.ProjectExt	\$(var.MyProject.ProjectExt)	.csproj
var.ProjectName.ProjectFileName	\$(var.MyProject.ProjectFileName)	MyProject.csproj
var.ProjectName.ProjectName	\$(var.MyProject.ProjectName)	MyProject
var.ProjectName.ProjectPath	\$(var.MyProject.ProjectPath)	C:\Project\MyProject.csproj
var.ProjectName.TargetDir	\$(var.MyProject.TargetDir)	C:\Project\bin\Debug\
var.ProjectName.TargetExt	\$(var.MyProject.TargetExt)	.exe
var.ProjectName.TargetFileName	\$(var.MyProject.TargetFileName)	MyProject.exe
var.ProjectName.TargetName	\$(var.MyProject.TargetName)	MyProject
var.ProjectName.TargetPath	\$(var.MyProject.TargetPath)	C:\Project\bin\Debug\MyProject.exe
var.ProjectName.Culture.TargetPath	\$(var.MyProject.en-US.TargetPath)	C:\Project\bin\Debug\MyProject.msm
var.SolutionDir	\$(var.SolutionDir)	C:\Solution\
var.SolutionExt	\$(var.SolutionExt)	.sln
var.SolutionFileName	\$(var.SolutionFileName)	MySolution.sln
var.SolutionName	\$(var.SolutionName)	MySolution
var.SolutionPath	\$(var.SolutionPath)	C:\Solution\MySolution.sln

C. Windows Installer Versionen

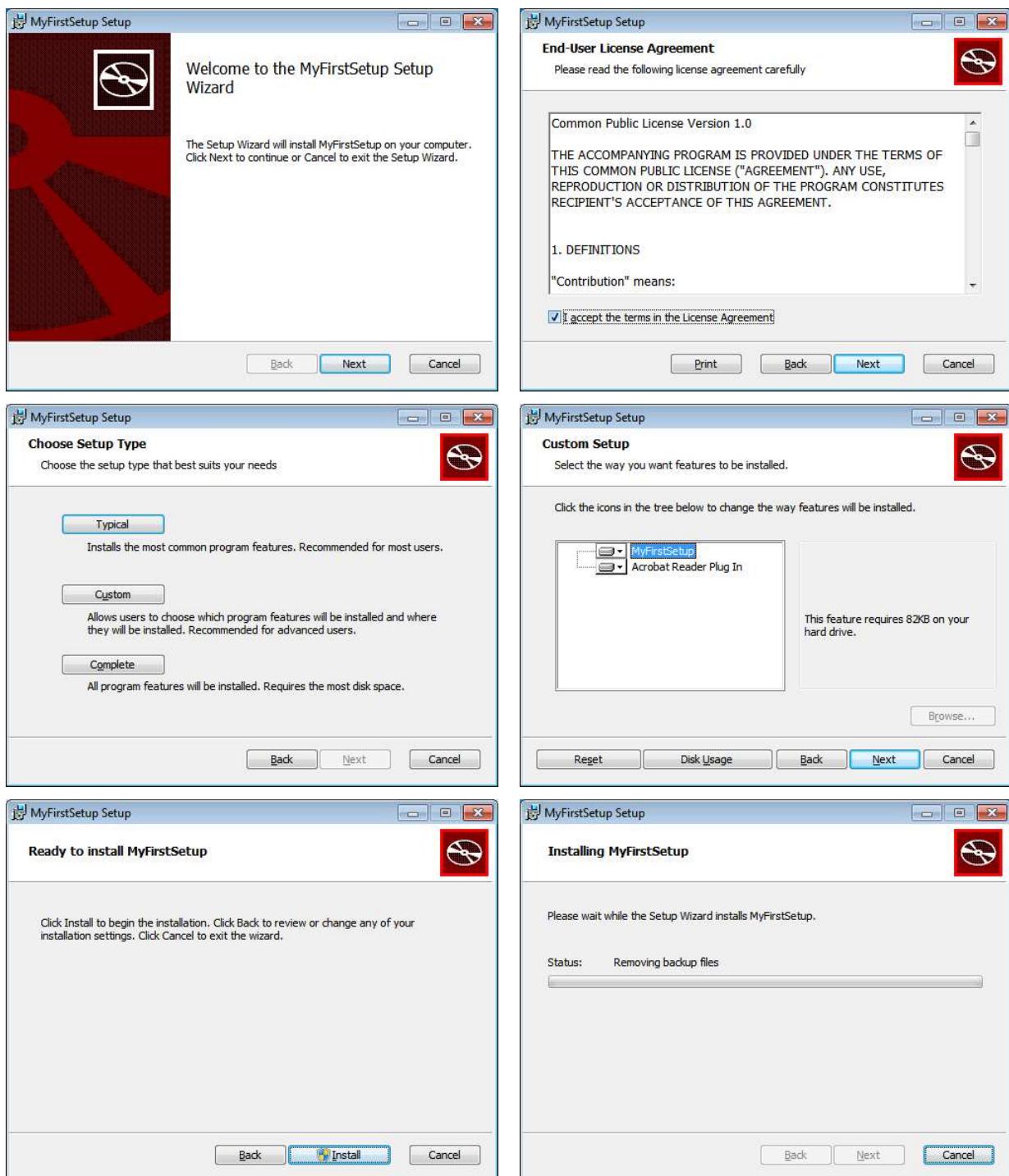
Der Windows Installer wurde mit Windows 2000 mit der Version 1.x ausgeliefert. Seither gibt es einige Erweiterungen, die in folgender Tabelle dargestellt sind:

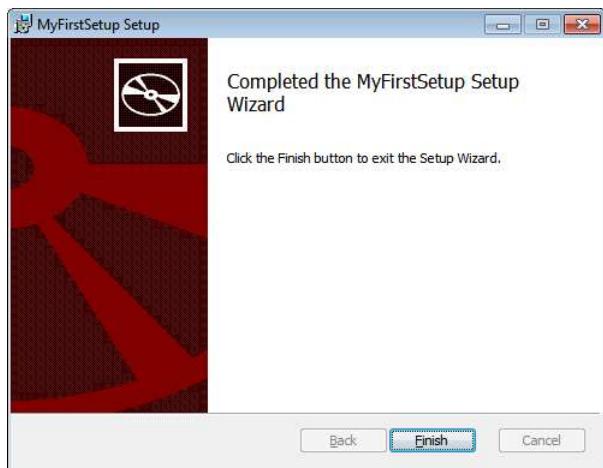
Version	Windows	Features
1.x	pre-XP	Basic MSI Support. Nur 32 Bit
2.0	XP, 2000 Server SP3	64-Bit-Support
3.0	XP SP2	Erweiterte Patch-Funktionalität
3.1	XP SP3, 2003 Server SP2	Erweiterung des Benutzerinterfaces
4.0	Vista, Server 2008	Unterstützung von User Access Control (UAC), Restart Manager und MSI Chaining
4.5	XP SP3, Vista and Server 2008 SP2	Erweiterte Patch-Funktionalität
5.0	Windows 7 bis Server 2008 R2 bis Server 2016	Erweiterung bei Berechtigungen, der Konfiguration von Diensten, dem Benutzerinterface, der Installation per User und per Maschine

Bei der Frage, welches Schema das zu erstellende Setup haben soll, gibt es keine allgemeingültige Antwort. Wenn wir alle Betriebssysteme unterstützen wollen, dann sollte unser Schema 200 (also Version 2.0) sein. Nur wenn wir Funktionalitäten benötigen, die erst in späteren Versionen des Windows Installers hinzugefügt wurden, sollten wir das Schema entsprechend anpassen.

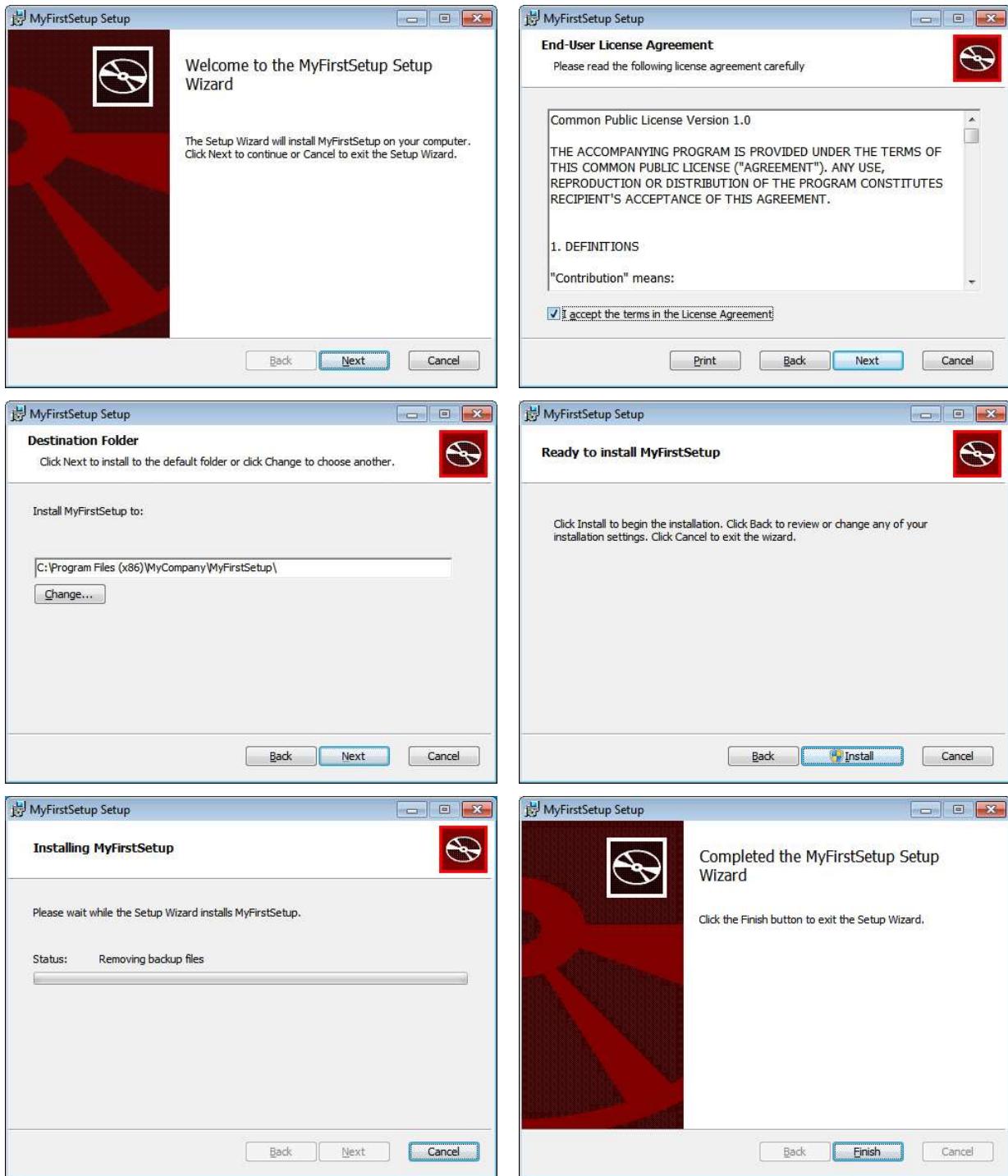
D. WiXUI-Dialoge

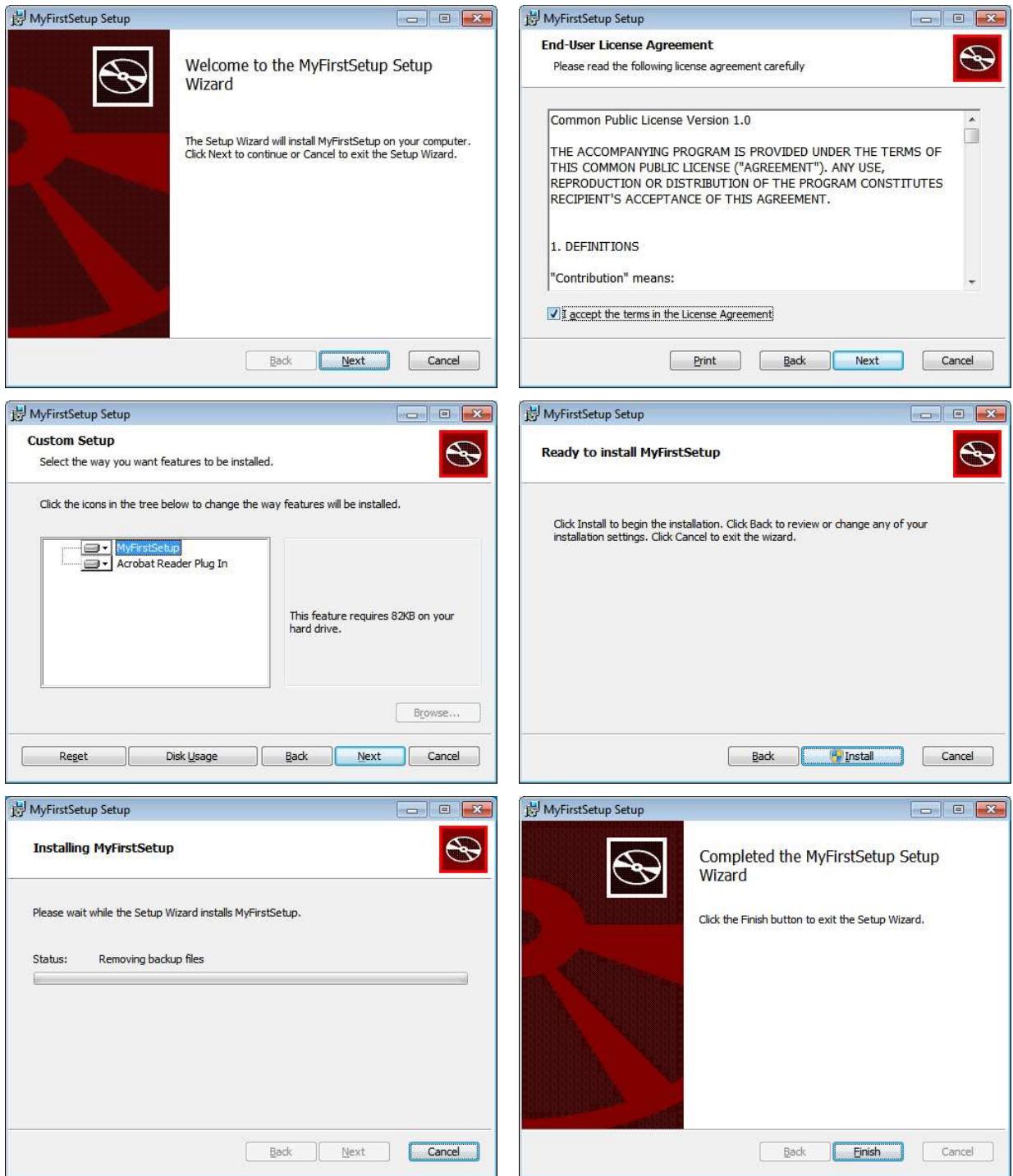
WixUI_Mondo:



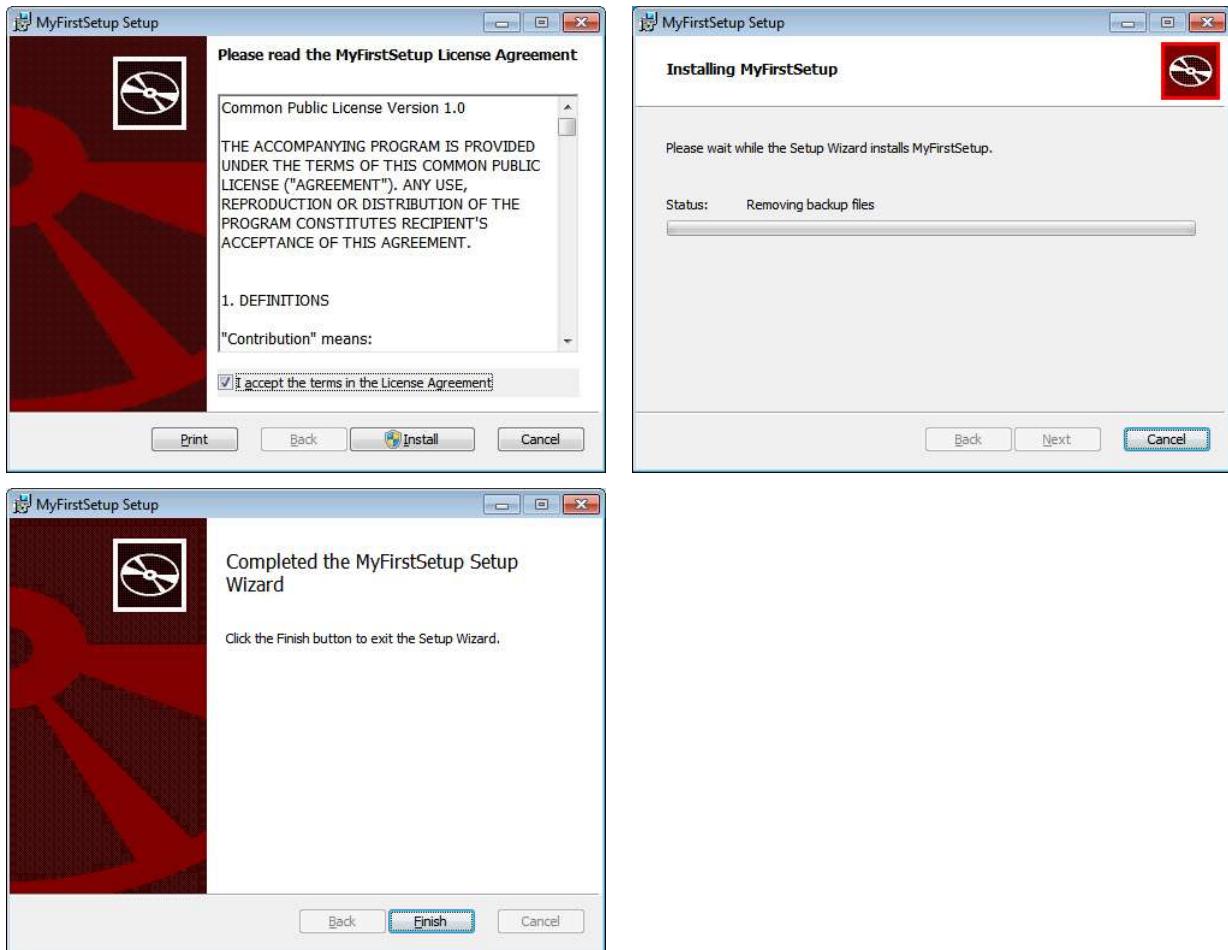


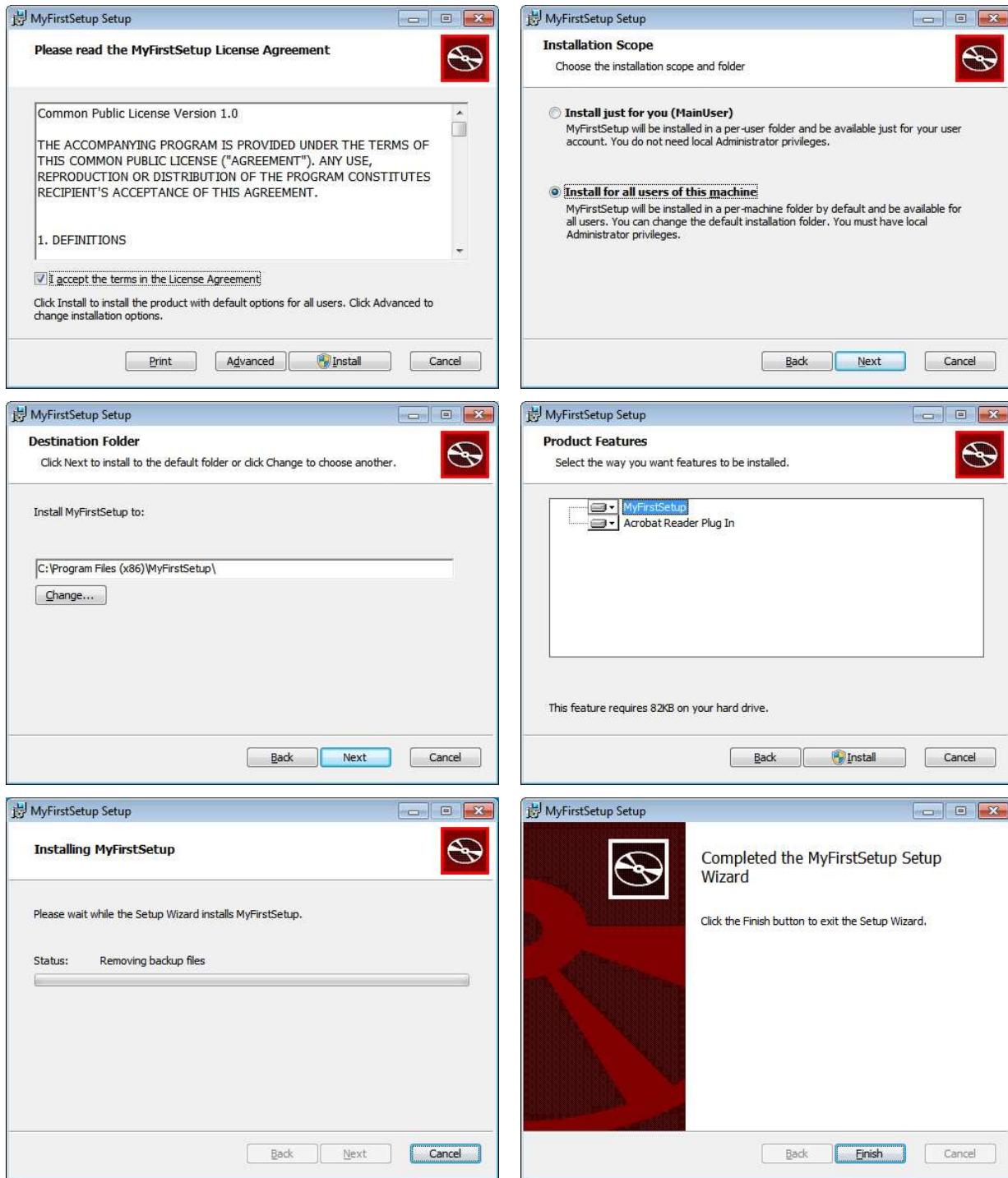
WixUI_InstallDir:



WixUI_FeatureTree:

WixUI_Minimal:



WixUI_Advanced:

E. Tools des WiX-Toolsets

Übersicht

Name	Beschreibung
Candle.exe	Präprozessor und Compiler von WiX-Quell-Dateien (wxs-Dateien). Als Ergebnis wird eine .wixobj-Datei erstellt, die im Grunde genommen auch eine XML-Datei darstellt. Die .wixobj werden mit Light zu einem Windows Installer Setup zusammengebunden.
Light.exe	Bindet eine oder mehrere wixobj-Dateien zu einem Windows Installer Setup (msi oder msm) zusammen. Wenn es nötig ist, erstellt light.exe auch CAB-Dateien undbettet diese in das MSI ein.
Lit.exe	Bindet mehrere wixobj-Dateien zu einer Library zusammen.
Dark.exe	Diskompliert eine Windows Installer Datenbank (msi Datei) und generiert daraus eine Wxs-Datei.
Heat.exe	Über Heat.exe können Dateien und Verzeichnisse abgescannt und in Wxs-Dateien eingetragen werden. Heat.exe ist vor allem bei der ersten Erstellung eines MSI-Setups ein nützliches Tool.
Insignia.exe	Schreibt Informationen vom digitalen Zertifikat, mit dem eine externe CAB-Datei signiert wurde, in die MSI-Datenbank.
Melt.exe	Konvertiert ein Merge-Modul in ein ComponentGroup einer Wxs-Datei.
Torch.exe	Erstellt aus zwei Datenbanken (WiXpdb, WiXout oder MSI) per Differenzbildung eine Transformation (entweder als WiXmst oder mst-Datei).
Smoke.exe	Führt eine Validierung von MSI- oder MSM-Dateien durch.
Pyro.exe	Erstellt aus einer WiXmsp-Datei und einer Transformation ein Windows Installer Patch (msp-Datei).
WiXCop.exe	Erzwingt Standards im WiX SourceCode. WinCop.exe kann auch zur Konvertierung von WiX-Code, der für eine älteren WiX-Version geschrieben wurden, in die aktuellste WiX-Version verwendet werden.
WiXUnit.exe	Führt Validierungen von WiX-Source-Dateien aus.
Lux.exe	Erstellt Modultests für Custom-Actions.
Nit.exe	Führt Modultests für Custom-Actions aus.

Candle.exe

Aufruf: candle.exe [-?] [-nologo] [-out outputFile] sourceFile [sourceFile ...] [@responseFile]

Parameter	Beschreibung
-arch	Gibt die Architektur des Kompilats an. Das ist wichtig, da einige Elemente im Setup (wie z. B. Komponenten) als 64 Bit markiert werden müssen.
-d	Über den Parameter -d können Präprozessorvariable in der Form: -dVariableName=VariableWert angegeben werden.
-ext	Über diesen Parameter können WiX-Extensions, wie z.B. WiXUIExtension, angegeben werden.
-fips	Dieser Parameter schaltet den Hashing-Algorithmus des Windows Installers von MD5 auf SHA1 um. FIPS bedeutet „Federal Information Processing Standard“ und ist die Bezeichnung für öffentlich bekanntgegebene Sicherheitsstandards der Vereinigten Staaten.
-I	Gibt den Suchpfad für WiX-Include-Dateien (.wxii) an.
-nologo	Candle.exe gibt normalerweise in der ersten Zeile die Versionsnummer und Copyright-Vermerke aus. Der Parameter -nologo überdrückt diese Ausgabe.
-o	Das –o- bzw. –out-Argument gibt den Namen und den Speicherort der erstellten wixobj-Datei an. Wird dieser nicht angegeben, so wird die Ausgabedatei im gleichen Pfad und mit demselben Dateinamen wie die zu kompilierende Wxs-Datei abgelegt.
-p	Als eines der ersten Schritte beim Kompilieren erstellt Candle aus der Wxs-Datei eine standardisiert geformte XML-Datei, die nach dem Kompilieren wieder gelöscht wird. Wenn diese XML-Datei nicht wieder gelöscht werden soll, kann der Parameter -p angegeben werden.
-pedantic	Wie der Name bereits sagt, wird Candle durch Angeben des Parameters -pedantic pedantisch. Wird z. B. bei einer Komponente kein Key angegeben, dann mahnt Candle diesem Umstand entsprechend an.
-sfdvital	Bei einem normalen Build werden alle Dateien als „vital“ markiert. Vital bedeutet, dass es sich bei der Datei um eine „lebensnotwendige“ Datei handelt. Kann diese Datei nicht installiert werden, bricht der Windows Installer die Installation ab. Dieses Standardverhalten kann durch diesen Parameter abgeschaltet werden.
-ss	Candle prüft den Aufbau des wxs-File anhand der XML-Schema-Datei wix.xsd. Diese Prüfung kann durch Angabe des Parameters -ss abgeschaltet werden.
-sw	Durch den Parameter -sw werden Kompilierwarnungen unterdrückt. Durch die explizite Angabe der Referenz-ID gelingt es auch, ausschließlich ganz bestimmte Warnungen zu unterdrücken (z. B. -sw1009). Eine Liste der Referenz-IDs können aus der Datei messages.xml im WiX-Quellcode ermittelt werden.
-trace	Fordert bei Warnungen und Fehler eine erweiterte Ausgabe an.
-v	Schaltet zusätzliche „verbose“ Ausgaben beim Kompilieren an.
-wx	Sollen alle Warnungen wie Fehler behandelt werden, so kann man den Parameter -wx setzen.

Light.exe

Aufruf: light.exe [-?] [-b basePath] [-nologo] [-out outputFile] objectFile [objectFile ...] [@responseFile]

Kommandozeilenargumente für das Verknüpfen:

Parameter	Beschreibung
-ai	Erlaubt gleiche Spalten. Gleiche Spalten werden als Warnung behandelt.
-b	Gibt light.exe an, in welchem Verzeichnis nach wixobj-Dateien gesucht werden soll. Dieser Parameter kann auch öfters angegeben werden, um mehrere Quellpfade zu definieren. Wird dieser Parameter nicht angegeben, sucht light.exe im aktuellen Verzeichnis.
-bf	Dies -bf-Flag wird immer im Zusammenhang mit der -xo-Flag benutzt. Die Flag sagt Light.exe, dass kein binäres MSI-File, sondern eine XML-Datei (mit der Dateiendung WiXout) erstellt werden soll.
-binder	Bei Light.exe kann eine benutzerdefinierte Binder-DLL angegeben werden. Über diesen Parameter kann die benutzerdefinierte Binder-Klasse angegeben. Wird -binder nicht angegeben, so wird die Klasse Microsoft.Tools.WindowsInstallerXml.Binder verwendet.
-cultures	Diese Flag sagt light.exe, welche WiX-Localisation-Dateien geladen werden sollen. Über diesen Parameter kann nur eine Sprache angegeben werden, weil das MSI-Setup immer monolingual ist.
-d	Über diesen Parameter kann eine WiX-Variable definiert (ohne Wert) bzw. angegeben werden. Die Syntax hierfür ist: -d<VariablenName>[=<VariablenWert>]
-dut	Der WiX-Compiler und -Linker erstellt bei der Erstellung von WiXout- bzw. WiXpdb-Dateien zusätzliche Tabellen, um dort Metadaten abzulegen. Diese Tabellen existieren in der MSI-Spezifikation nicht. Deshalb werden diese Tabellen „unreal tables“ genannt. Über den Parameter -dut verhindern wir, dass diese Tabellen in der WiXout- bzw. WiXpdb-Datei angelegt werden.
-ext	Über die -ext-Flag werden WiX-Extension-DLLs angegeben, die mit geladen werden sollen. So wird über die Kommandozeile -ext "%WiX%bin\WiXUIExtension.dll" die User-Interface-Extension geladen.
-loc	Wenn Texte in WiX-Localisation-Dateien abgelegt sind, können diese über diesen Parameter mit berücksichtigt werden. Es werden jedoch nur die Sprachteile berücksichtigt, die in Parameter -cultures angegeben wurden.
-nologo	Light.exe gibt normalerweise in der ersten Zeile die Versionsnummer und Copyright-Vermerke aus. Der Parameter -nologo überdrückt diese Ausgabe.
-notidy	Light.exe erstellt zur Laufzeit temporäre Dateien, die nach dem Binden der Dateien wieder gelöscht werden. Für die Fehlerbehandlung kann es nützlich sein, dass diese Dateien nicht wieder gelöscht werden. Das kann man über diesen Parameter definieren.
-o	Der Parameter -o oder -out gibt Light.exe den Namen der zu erstellenden MSI oder WiXout-Datei an.
-pedantic	Wie der Name bereits sagt, wird Light.exe durch Angeben des Parameters -pedantic pedantisch – es werden zusätzliche Nachrichten ausgegeben.
-sadmin	Dieser Parameter verhindert, dass Light.exe die AdminExecuteSequence- bzw. AdminUISequence- Tabellen erstellt. Ein Setup, das mit diesem Parameter erstellt wurde, kann nicht administrativ installiert werden.

Parameter	Beschreibung
-sadv	Dieser Parameter verhindert, dass Light.exe die AdvtExecuteSequence-Tabelle erstellt. Ein Setup, das mit diesem Parameter erstellt wurde, kann nicht als „advertised setup“ installiert werden.
-sloc	Dieser Parameter verhindert, dass Texte aus den WiX-Localisation-Dateien genommen werden. An Stelle des Textes wird der Verweis auf die Variable (z. B. as !(loc.myVariable)) angegeben. Dieser Parameter wird ignoriert, sobald entweder -loc oder -cultures angegeben wurde.
-sma	Durch das Setzen des Assembly-Attributes im File-Element wird die Datei in den Global-Assembly-Cache installiert. Das Setzen des Attributes bewirkt, dass zwei neue MSI-Tabellen MsiAssembly und MsiAssemblyName sowie die zugehörige Action MsiPublishAssemblies in das MSI eingebunden werden. Durch das Setzen des Attributes -sma wird dieser Vorgang unterdrückt.
-ss	Light.exe validiert normalerweise das XML-Schema, um zu prüfen, ob die Syntax der WiXout- bzw. WiXpdb-Dateien korrekt ist. Durch diesen Parameter wird die Validierung unterdrückt.
-sts	Light.exe benutzt GUIDs, um Zeilen in WiXout- bzw. WiXpdb-Dateien zu identifizieren. Dieser Parameter verhindert dies.
-sui	Durch Angabe dieses Parameters werden alle Userinterface-Sequenz-Tabellen (InstallUISequence und AdminUISequence) unterdrückt. Das MSI wird also ohne UserInterface erstellt.
-sv	Light.exe speichert in die WiXout- bzw. WiXpdb-Dateien die aktuelle Versionsnummer ab. Light.exe prüft vor der Verarbeitung von WiXout- bzw. WiXpdb-Dateien, ob diese auch mit der aktuellen Version von light.exe erstellt wurden. Ist das nicht der Fall, so wird der Bindevorgang mit einem Fehler abgebrochen. Der Parameter -sv verhindert, dass light.exe die Versionsnummern prüft.
-sw	Der Parameter -sw (oder auch -swall) schaltet alle Fehler und Warnungen ab. Durch die explizite Angabe der Referenz-ID gelingt es auch, ausschließlich ganz bestimmte Warnungen zu unterdrücken (z. B. -sw1009). Eine Liste der Referenz-IDs kann aus der Datei messages.xml im WiX-Quellcode ermittelt werden.
-usf	Durch Angabe des Parameters -usf, gefolgt vom Dateiname einer XML-Datei (z. B. -usf unrefSymbols.xml), werden alle nicht referenzierte Elemente in die XML-Datei geloggt.
-v	Schaltet zusätzliche „verbose“-Ausgaben beim Linken an.
-wx	Sollen alle Warnungen wie Fehler behandelt werden, so kann man den Parameter -wx setzen. Durch die explizite Angabe der Referenz-ID gelingt es auch, dass ausschließlich ganz bestimmte Warnungen als Fehler behandelt werden (z. B. -wx1009).
-xo	Dieser Parameter bewirkt, dass die Ausgabedatei von Light.exe kein MSI, sondern eine WiXout-Datei ist.

Kommandozeilenargumente für das Binden:

Parameter	Beschreibung
-bcgg	Abwärtskompatibilität für den GUID-Generierungsalgorithmus benutzen.
-cc <path>	Cache-Pfad, in dem die Kabinettdateien erstellt werden (Pfad wird nach dem Binden nicht gelöscht)
-ct <N>	Maximale Anzahl der Threads für das Erstellen der Kabinettdateien (Default ist die Anzahl der verfügbaren Prozessoren)
-cub <file.cub>	Zusätzliche Cub-Dateien für die Validierung am Ende des Build-Prozesses
-dcl:level	Gibt den Kompressionslevel bei der Erstellung der Cabint-Dateien an. Mögliche Werte sind: low, medium, high, none, mszip; mszip default.
-eav	Exakte .NET-Assembly-Versionsnummer (Standard ist .NET 1.1 RTM)
-fv	Fügt einen Eintrag <i>fileVersion</i> in die MsiAssemblyName-Tabelle hinzu.
-ice:<ICE>	Führt nur die angegebenen ICEs (Internal Consistency Evaluator) bei der Validierung aus.
-O1	Optimiert das Komprimieren der Dateien auf die kleinsten Kabinettdateien.
-O2	Optimiert das Komprimieren der Kabinettdateien auf die schnellste Installationszeit.
-pd bout <output >	Speichert die WiXPdb-Datei in den angegebenen Pfad und Dateinamen.
-reusecab	Gibt an, dass die Kabinettdatei aus dem Cache-Pfad (siehe Parameter -cc) verwendet werden soll.
-sa	“Suppress Assemblies”. Assembly-Informationen werden beim Bild-Prozess nicht aus den Dateien ausgelesen.
-sacl	ACLs werden nicht zurückgesetzt (ist sinnvoll, wenn das Ergebnis auf einem Netzlaufwerk abgelegt werden soll).
-sf	„Suppress file info“. Es werden keine File-Informationen gesammelt (siehe Parameter -sa und -sh).
-sh	Es wird kein Hash-Wert, keine Versionsnummer und auch keine Sprache aus den Dateien ausgelesen.
-sice:<ICE>	Unterdrückt die angegebenen ICEs (Internal Consistency Evaluator) bei der Validierung.
-sl	Suppress-Layout
-spdb	Unterdrückt das Erstellen der WiXPdb-Datei.
-sval	Unterdrückt die MSI-/MSM-Validierung.

Die Umgebungsvariable WIX_TEMP überschreibt das Verzeichnis, in dem temporäre Dateien beim Build-Prozess abgelegt werden.

Heat.exe

Aufruf: heat.exe harvestType harvestSource [harvester arguments] -o[ut] sourceFile.wxs

Unterstützte Harvesting-Typen:

Parameter	Beschreibung
Dir	Ein Verzeichnis wird abgescannt.
File	Eine Datei wird abgescannt.
perf	Ein Performance-Counter wird abgescannt.
project	Ein Visual-Studio-Projekt wird abgescannt.
Reg	Eine Reg-Datei (z. B. mit Regedit.exe exportiert) wird abgescannt.
website	Eine IIS-Webseite wird abgescannt.

Optionen:

Parameter	Beschreibung
-ag	Die GUID-Schlüssel der Komponenten werden automatisch generiert.
-cg <Name>	Definiert den Namen der erstellten ComponentGroup.
-configuration	Konfiguration, die beim Abscannen eines Projektes gesetzt werden soll
-directoryid	Wert, der beim Generieren der Directory-IDs verwendet wird
-dr <Name>	Verzeichnisname, der als Directory-Root-Verzeichnis verwendet werden soll
-ext <ext>	Über diesen Parameter kann eine Extension von Heat.exe angegeben werden.
-g1	Es werden Komponenten-GUIDs ohne Klammern erstellt.
-generate	Definiert, welche Elemente erstellt werden sollen. Mögliche Werte sind <i>components</i> , <i>container</i> , <i>payloadgroup</i> , <i>layout</i> , wobei die Standardeinstellung <i>components</i> ist.
-gg	Erstellt GUID-Schlüssel.
-indent <N>	Indentation multiple (overrides default of 4)
-ke	Leere Verzeichnisse werden auch mit in die Wxs-Datei aufgenommen.
-nologo	Unterdrückt die Heat-Logo-Informationen.
-out	Definiert die Ausgabedatei (wird standardmäßig im aktuellen Verzeichnis abgelegt).
-platform platform	Plattform, die gesetzt werden soll, wenn ein Projekt abgescannt wird
-pog	Spezifiziert die Ausgabegruppe von Visual-Studio-Projekten. Mögliche Werte: <i>Binaries</i> , <i>Symbols</i> , <i>Documents</i> , <i>Satellites</i> , <i>Sources</i> , <i>Content</i> . Diese Option kann für mehrere Ausgabegruppen öfters angegeben werden.
-projectname	Überschriebener Projektname, der in Variablen verwendet werden soll
-scom	COM-Server werden nicht über die Class-, ProgId- und Typelib-Tabelle installiert, sondern über die Registry-Tabelle. Möchte man überhaupt keine COM-Elemente abscheiden, so muss -scom in Kombination mit -sreg aufgerufen werden.
-sfrag	Fragmente unterdrücken.
-srd	Root-Verzeichnis soll nicht abgescannt werden (nur die Unterverzeichnisse).

Parameter	Beschreibung
-sreg	Registry-Einträge sollen nicht abgescannt werden.
-suid	Unterdrückt das Erstellen von eindeutigen IDs für den Primary-Key der File-, Component- und Direcotry-Tabelle.
-svb6	VB6-COM-Elemente sollen unterdrückt werden.
-sw<N>	Unterdrückt alle Warnungen oder eine best. Message ID (z. B. -sw1011 -sw1012).
-swall	Unterdrückt alle Warnungen (veraltet).
-t	Transformiert abgescanntes Ausgabeergebnis mit einer XSL-Datei.
-template	Template, das verwendet werden soll. Mögliche Werte: fragment,module,product.
-v	Schaltet die detaillierte Ausgabe ein.
-var <Name>	Ersetzt das Quellenverzeichnis der Dateien (Standardmäßig „SourceDir“) durch die angegebene Präprozessor WiX-Variable.
-wx[N]	Behandelt alle Warnungen (oder die angegebene Message-ID) als Fehler (z. B. -wx1011 -wx1012).
-wxall	Behandelt alle Warnungen als Fehler (veraltet).
-?	Zeigt die Kommandozeilenargumente von Heat.exe.

Torch.exe

Aufruf: torch.exe [-?] [Optionen] Ziel-Paket Upgrade-Paket-out outputFile [@responseFile]

Optionen:

Parameter	Beschreibung
-a	Erstellt ein administratives Image (entpackt alle Dateien aus der Cabinet-Datei).
-ax <path>	Erstellt ein administratives Image inklusive dem entpacken der binären Dateien (Kombination aus -a und -x).
-ext <extension>	Gibt eine Extension für Torch.exe an. Hierbei wird entweder nur die Assembly-DLL oder -Klasse und die Assembly-DLL angegeben.
-nologo	Unterdrückt die Torch-Logo Informationen.
-notidy	Temporär erstellte Dateien sollen nicht gelöscht werden (sinnvoll bei der Fehlersuche).
-o[ut]	Definiert die Ausgabedatei (wird standardmäßig im aktuellen Verzeichnis abgelegt).
-p	Belässt unveränderte Inhalte im Ausgabeverzeichnis.
-pedantic	Zeigt pedantisch Nachrichten.
-serr <L>	Wenn die erstellte Transformation angewandt wird, sollen die angegebenen Fehler (siehe Error-Flags) unterdrückt werden.
-sw<N>	Unterdrückt alle Warnungen oder eine best. Message ID (z. B. -sw1011 - sw1012).
-swall	Unterdrückt alle Warnungen (veraltet).
-t <type>	Gibt den Typ der Transformation an (siehe Transformation-Flag).
-v	Schaltet die detaillierte Ausgabe ein.
-val <L>	Gibt an, was beim Anwenden der erstellten Transformation geprüft werden soll (siehe Validierungs-Flags).
-wx[N]	Behandelt alle Warnungen (oder die angegebene Message-ID) als Fehler (z. B. -wx1011 -wx1012).
-wxall	Behandelt alle Warnungen als Fehler (veraltet).
-x <pfad>	Entpackt Dateien in der Binär-Tabelle an den angegebenen Pfad.
-xi	Gibt an, dass kein MSI, sondern Wix (.wixout oder .wixpdb) als Eingangsformat verwendet werden soll.
-xo	Gibt an, dass kein MST, sondern Wix (.wixout) als Ausgangsformat verwendet werden soll.
-?	Zeigt die Kommandozeilenargumente von Pyro.exe.

Error-Flags:

Parameter	Beschreibung
a	Ignoriert den Fehler, wenn eine vorhandene Zeile hinzugefügt werden soll.
b	Ignoriert den Fehler, wenn eine Zeile gelöscht werden soll, die nicht vorhanden ist.
c	Ignoriert den Fehler, wenn eine vorhandene Tabelle hinzugefügt werden soll.
d	Ignoriert den Fehler, wenn eine Tabelle gelöscht werden soll, die nicht vorhanden ist.
e	Ignoriert den Fehler, wenn eine Zeile geändert werden soll, die nicht vorhanden ist.
f	Ignoriert den Fehler, wenn die Codepage geändert wird.

Validierungs-Flags:

Parameter	Beschreibung
g	UpgradeCode muss übereinstimmen.
l	Sprache muss übereinstimmen.
r	ProductCode (Product-ID) muss übereinstimmen.
s	Es wird nur die Major-Versionsnummer geprüft.
t	Es werden die Major- und die Minor-Versionsnummer geprüft.
u	Es werden Major-, Minor- und Upgrade-Versionsnummer geprüft.
v	Die Versionsnummer des Upgrade-Pakets muss kleiner als die Versionsnummer des Zielpakets sein.
w	Die Versionsnummer des Upgrade-Pakets muss kleiner als die / gleich der Versionsnummer des Zielpakets sein.
x	Die Versionsnummer des Upgrade-Pakets muss gleich der Versionsnummer des Zielpakets sein.
y	Die Versionsnummer des Upgrade-Pakets muss größer als die Versionsnummer des Zielpakets sein.
z	Die Versionsnummer des Upgrade-Pakets muss größer als die / gleich der Versionsnummer des Zielpakets sein.

Validierungs-Flags:

Parameter	Beschreibung
language	Gibt Validation-Flags für eine Sprachtransformation an.
instance	Gibt Validation-Flags für eine Instance-Transformation an.
patch	Gibt Validation-Flags für eine Patch-Transformation an.

Pyro.exe

Aufruf: pyro.exe [-?] [-nologo] inputFile -out outputFile [-t baseline wixTransform] [@responseFile]

Optionen:

Parameter	Beschreibung
-bt <pfad>	Setzt den BindPath für das alte wixpdb. Es werden, wie bei Light.exe, zwei Formate akzeptiert. (Beispiel: -bt name1=c:\Test\Name\ -bt c:\MySources)
-bu <pfad>	Setzt den BindPath für das neue wixpdb. Es werden, wie bei Light.exe, zwei Formate akzeptiert. (Beispiel: -bt name1=c:\Test\Name\ -bt c:\MySources)
-cc <pfad>	Pfad zum Cache, in dem die Cabinet-Dateien erstellt werden
-delta	Erstellt einen Delta-Patch. Ohne diesen Parameter wird ein Full-File-Patch erstellt. Ein Delta-Patch ist in der Regel kleiner als ein Full-File-Patch, benötigt jedoch mindestens Windows Installer Version 3.0.
-ext <extension>	Gibt eine Extension für Pyro.exe an. Hierbei wird entweder nur die Assembly-DLL oder -Klasse und die Assembly-DLL angegeben.
-fv	Gibt an, dass die Versionseinträge in der MsiAssemblyName-Tabelle aktualisiert werden.
-nologo	Unterdrückt die Pyro-Logo-Informationen.
-notidy	Temporär erstellte Dateien sollen nicht gelöscht werden (sinnvoll bei der Fehlersuche).
-o[ut]	Definiert die Ausgabedatei (wird standardmäßig im aktuellen Verzeichnis abgelegt).
-pdabout <output.wixpdb>	Gibt an, dass die WixPdb-Datei in einem anderen Verzeichnis abgelegt werden soll (wird standardmäßig im aktuellen Verzeichnis abgelegt).
-reusecab	Cabinet-Dateien sollen, falls möglich, aus dem Cabinet-Cache verwendet werden.
-sa	Assemblies unterdrücken: Assembly-Informationen werden nicht ermittelt.
-sf	Dateien unterdrücken: Dateiinformationen werden nicht berücksichtigt (äquivalent zu -sa und -sh).
-sh	Dateiinformationen unterdrücken: Für Dateien werden Hash-Werte, Versionsnummern, Sprache usw. nicht berücksichtigt.
-spdb	Unterdrückt das Erstellen der WixPdb-Datei.
-sw[N]	Unterdrückt alle Warnungen oder eine spezifische Message-ID (z.B. -sw1011 – sw1012).
-swall	Behandle alle Warnungen als Fehler (veraltet).
-t baseline transform	Gibt eine oder mehrere Transformationen und die zugehörige Baseline an.
-v	Schaltet die detaillierte Ausgabe ein.
-wx[N]	Behandle alle Warnungen (oder die angegebene Message-ID) als Fehler (z.B. -wx1011 -wx1012).
-wxall	Behandelt alle Warnungen als Fehler (veraltet).
-?	Zeigt die Kommandozeilenargumente von Pyro.exe.

F. Index

&

" · 113

A

Access Control Entry · Siehe ACE

ACE · 143

Action

- CostFinalize · 39, 78, 79
 - CreateFolders · 76
 - CreateShortcuts · 76
 - ExecuteAction · 73, 76
 - FindRelatedProducts · 206
 - InstallFiles · 76
 - InstallFinalize · 104
 - InstallInitialize · 104
 - InstallValidate · 194
 - LaunchConditions · 207
 - MigrateFeatureStates · 207
 - MsiPublishAssemblies · 57
 - RemoveExistingProducts · 207
 - ScheduleReboot · 78
 - SelfRegModules · 133
 - SelfUnregModules · 133
- AdminExecuteSequence · 246
- Administrative Installation · 74, 211, 215
- AdminToolsFolder · 234
- AdminUISequence · 246
- Advertisement Installation · 73
- Advertising · 13
- AdvtExecuteSequence · 247
- ALTER TABLE · 119
- AppDataFolder · 234
- ApplyAction · 166, 171
- ApplyComplete · 172
- AutoVersion · 45

B

- Billboards · 72
- Binary · 83
- Binder Variable · 45
- BindPath · 253
- bindpath Variable · 32
- Bootstrapper · 146
- BootstrapperApplication Klasse · 161
- Bootstrapper-Applikation · 147
- BootstrapperCode.dll · 161, 163
- Build-Server · 230

- BundleId · 153
- Burn · 146
- Burn Built-in Variables · 148, 151
- Burn Extension
 - WixBalExtension · 147, 151, 152
 - WixUtilExtension · 152
- Burn Logfile · 155
- Burn Prerequisite · 178
- Burn-Attribut
 - RemotePayload · 158
- Burn-Element
 - BootstrapperApplication · 147, 179
 - BootstrapperApplicationRef · 147, 163
 - Bundle · 147
 - Chain · 147
 - Chain · 154
 - ComponentSearch · 153
 - DirectorySearch · 153
 - ExePackage · 154, 155
 - ExitCode · 156
 - FileSearch · 153
 - MsiPackage · 154
 - MsiProperty · 154
 - MspPackage · 154, 156
 - MsuPackage · 154
 - PackageGroup · 160, 178
 - Payload · 159
 - PayloadGroup · 159
 - RegSearch · 153
 - RelatedBundle · 153
 - RollbackBoundary · 158
 - Variable · 151
- Burn-Variable
 - WixBundleInstalled · 163
- Byte-Level-Patch · 208

C

- C# Custom Action · 83
- C# Custom Action Project · 83
- C++ Action debuggen · 100
- C++ Custom Action · 87
- candle.exe · 27
- Candle.exe · 244, 245
- CAQuietExec · 113
- CAQuietExec64 · 113
- CDATA · 91, 92
- CER-kodiertes Zertifikat · 180
- certmgr.msc · 181
- Chaining · 158
- COM Interop · 133
- CommonAppDataFolder · 234
- CommonFilesFolder · 234

Component · 22
 ComponentGroup · 128
 Conditions · 33
 ConfigurableDirectory · 61
 CREATE TABLE · 119
 CRLF in Message · 34
 Custom Action · 83
 C# · 83
 C++ · 87
 Direcotry · 79
 EXE · 80
 JavaScript · 92
 PowerShell · 93
 Property · 79
 VB.NET · 86
 VBScript · 90
 Custom Action debuggen · 97
 CustomActionData · 106, 110, 115
 CustomActionData Objekt · 106
 Custom-Table · 115

D

DACL · 143
 Daily-Build · 230
 dark.exe · 233
 Dark.exe · 244
 Data Source Name · *Siehe DSN*
 Database Objekt · 122
 Datei Überschreibung Regeln · 23
 Dateien einbinden · 23
 Datenbank-Transformation · 208
 DbgPrint · 102
 Debuggen
 C++ Action · 100
 JavaScript Action · 102
 VB.NET Action · 99
 VBScript Action · 101
 Debugger Assembly · 99
 DebugView.exe · 209
 Def-Datei · 88
 define · 44
 DELETE · 119
 Deployment Tools Foundation · 83, 86
 DesktopFolder · 234
 Development Tools Foundation · 106
 DFT · *Siehe Deployment Tools Foundation*,
Siehe Deployment Tools Foundation
 Dienst installieren · 137
 DIFx · 184
 DIFxAPP · 184
 difxapp_x64.wixlib · 184
 difxapp_x86.wixlib · 184
 Directory · 226
 DirectoryRef · 226
 Discretionary Access Control List · *Siehe DACL*
 DLL-Hell · 131
 Driver Install Frameworks · *DIFx*

Driver Installation Frameworks for Applications ·
DIFxAPP
 DROP TABLE · 119
 DSN · 53
 dutil.lib · 88

E

Entry Point · 21
 Escape-Squenz · 47
 ExecutePackageBegin · 171
 ExecutePackageComplete · 172
 ExitDialog · 80
 Extension
 WixDifxAppExtension · 184
 WixFirewallExtension · 222
 WixIISExtension · 183
 WixNetFxExtension · 160
 WixUtilExtension · 113, 139, 186, 189

F

FavoritesFolder · 234
 Feature · 25, 61
 Features · 20
 FontsFolder · 234
 foreach · 46
 Fragment · 40

G

GAC · *Siehe Global Assembly Cache*
 Global Assembly Cache · 57
 Global Unique Identifier · *Siehe GUID*
 Gruppenrichtlinie
 Logging · 193
 GUID · 17

H

Heat.exe · 128, 244
 HeatDirectory · 130

I

ICE · 21, 115
 ifdef · 44
 ifndef · 44
 IIS · 134
 include · 48
 Include Dateien · 48
 InScript-Custom Action · 106
 INSERT · 119
 Insignia · 180
 Insignia.exe · 244

Installationsbedingungen · 34
Installationssequenz · 73
Installations-Status · 77
InstallCertificates · 183
InstallDirDlg · 66
Installer Objekt · 91
Installer-Cache · 209
InstallExecuteSequence · 76, 78
InstallUISequence · 75, 78
InstallValidate · 194
Instanz-Transformation · 201
Internal Consistency Evaluators · Siehe ICE
InvokeShutdown · 170
Iteration · 46

J

JavaScript Action debuggen · 102
JavaScript Custom Action · 92

K

Kabinettdatei · 21
KeyPath · 21
Klasse
 PreprocessorExtension · 58
 WixExtension · 58
Komponenten · 20
Komponentenregel · 23
Kumulativer Patch · 217

L

Lambda-Expression · 170
light.exe · 27
Light.exe · 244, 246, 249
Lit.exe · 244
LocalAppDataFolder · 234
Logging · 193
Lokalisierung · 62
Lux.exe · 244

M

MakeCert.exe · 181
MakeSfxCA.exe · 85
Makro · 48
Melt.exe · 244
Merge Module · 116, 225
 Abhängigkeit · 225, 227
 Einbinden · 225
 Erstellen · 227
Microsoft.Practices.Prism.dll · 168
MMsiBreak · 97
Model-Klasse · 165
Modularisierung · 116

ModuleSignature · 228
MsBuild.exe · 230
MsiAssembly · 247
MsiAssemblyName · 247
MsiBreak · 100
MsiDigitalCertificate · 180
MsiDigitalSignature · 180
MsiDoAction · 105, 109
Msieexec.exe · 28
MsiGetProperty · 89
MsiGetTargetPath · 88
MsiLogging · 193
msimsp.exe · 210, 211
MsiPatchCertificate · 220
MsiViewModify · 125
MSP Datei · 208
msscrdbg.exe · 101
MST · Siehe Transformation
Multi Target Patch · 208
Multiinstanz-Installation · 201, 211
Multi-SKU-Patch · 208, 212, 219
MyPicturesFolder · 234

N

NetHoodFolder · 234
Nit.exe · 244
NotificationObject · 168

O

ODBC · 52
odbcad32.exe · 53
Open Database Connectivity · Siehe ODBC
openssl.exe · 183
Orca · 16, 197
Orca.exe · 28
ORDER BY · 118
OutputDebugString · 102

P

PackageCache · 154, 155
PackageCode · 203
Patch · 208
Patch auf Multiinstanz · 211
Patch Creation Properties Datei · 210, 211
Patch deinstallieren · 210
PatchProperty · 214
Patch-Transformation · 208
PCP-Datei · 210, 211
PersonalFolder · 235
Pfc2Pfx.exe · 182
PlanAction · 166, 170, 171
PlanComplete · 166, 171
Plug & Play · 184
PowerShell Custom Action · 93

Präprozessor
 define · 44
 error · 44
 ifdef · 44
 ifndef · 44
 include · 48
 Präprozessor-Variable · 44, 231
 PreqbLogo · 179
 PrintHoodFolder · 235
 Prism Library · 168
 ProductCode · 202
 ProgramFilesFolder · 235
 ProgramMenuFolder · 235
 Projektreferenz-Variable · 43
 Property · 33
 ADDDEFAULT · 205
 ADDLOCAL · 205
 ADDSOURCE · 205
 ADVERTISE · 205
 ALLUSERS · 234
 ARPCOMMENTS · 135
 ARPCONTACT · 135
 ARPINSTALLLOCATION · 135
 ARPNOMODIFY · 135
 ARPNOREMOVE · 135
 ARPNOREPAIR · 135
 ARPPRODUCTICON · 135
 ARPREADME · 135
 ARPSIZE · 135
 ARPSYSTEMCOMPONENT · 135
 ARPURLINFOABOUT · 135
 CommonAppDataFolder · 190
 CustomActionData · 104, 106, 110
 FontsFolder · 136
 GlobalAssemblyCache · 57
 INSTALLLEVEL · 39
 MSIDISABLELUAPATCHING · 221
 MsiHiddenProperties · 34, 189
 MsiLogFileLocation · 194
 MsiLogging · 193
 MSINWINSTANCE · 201
 ProductCode · 104, 201, 202
 ProduktCode · 28
 REINSTALL · 205
 REINSTALLMODE · 24, 205
 REMOVE · 205
 SystemFolder · 131
 TARGETDIR · 74
 TRANSFORMS · 201
 UpgradeCode · 203
 UserSID · 104
 WindowsFolder · 82
 WIXUI_EXITDIALOGOPTIONALCHECKBO
 XTTEXT · 81
 Publishing · Siehe Veröffentlichen
 Pyro.exe · 214, 215, 244, 253

Q

QuietExec · 90
 Quoting · 187

R

Redistributables · 146
 RegAsm.exe · 133
 RegSvr32.exe · 131
 ReleaseMem · 89
 ResumeDlg · 205
 Rollback · 13

S

SACL · 143
 Schriftart installieren · 136
 Script Debugger · 101
 SDDL · 143
 Security Descriptor Definition Language · 143
 Security Descriptor String · Siehe SID
 Selbstregistrierung · 133
 Selbstreparatur · 21
 SELECT · 118, 120
 SelfRegModules · 133
 SelfReg-Tabelle · 133
 SelfUnregModules · 133
 SendToFolder · 235
 Sequenz · 73
 Service installieren · Siehe Dienst installieren
 Session
 GetTargetPath · 84
 Session Objekt · 91, 92
 SharedDllRefCount · 133
 Shortcut · 24
 SID · 143
 SignTool.exe · 180, 221
 Silent-Installation · 74
 Slipstreaming · 156
 Small Update · 202
 Smoke.exe · 244
 Sprachtransformation · 198
 SQL Syntax · 118
 SQL-Befehle
 ALTER TABLE · 119
 CREATE TABLE · 119
 DELETE · 119
 DROP TABLE · 119
 INSERT · 119
 ORDER BY · 118
 SELECT · 118, 120
 UPDATE · 119
 WHERE · 118, 124
 Stammzertifikat · 181
 StartMenuFolder · 235
 StartupFolder · 235
 StringVariables · 177

System Access Control List · Siehe SACL
SystemFolder · 235

T

Tabelle

Validation · 115, 226
AdminExecuteSequence · 74
AdminUISequence · 74
AdvtExecuteSequence · 73
AdvtUISequence · 73
Class · 131
ComboBox · 123
CustomAction · 76
ImageFamilies · 213
InstallExecuteSequence · 73, 76
InstallUISequence · 75
InstallUiSequenz · 73
ListBox · 123
LockPermission · 139, 143
Media · 209, 213
MIME · 226
ModuleDependency · 225
ModuleSignature · 228
MsiAssembly · 57
MsiAssemblyName · 57
MsiLockPermissionsEx · 143
MsiPatchMediadata · 208
MsiPatchMetadata · 212
MsiPatchSequence · 208, 214
ODBCAttribute · 55
ODBCDataSource · 54
ODBCDriver · 55
ODBCSourceAttribute · 54
PatchMethadata · 212
ProgID · 131
Registry · 131
SelfReg · 133
TypeLib · 131
Upgrade · 206
TARGETDIR · 74
TempFolder · 235
torch.exe · 198, 215
Torch.exe · 215, 244, 251
Transformation · 197
Transformation erstellen · 197
TrueType Schriftart · 136

U

UAC · Siehe User-Account-Control
Umgebungs-Variablen · 44
Update · 202
 Major Update · 202, 206
 Minor Update · 202, 203
 Small Update · 203
UPDATE · 119
UpgradeCode · 203

UpgradeCode (Bundle) · 153
User Account Control Patching · 180, 220
Userinterface Erweiterung · 161
util
 XmlAttribute · 186
 XmlAttribute · 186

V

Validierung · 21, 226
VB.NET Action debuggen · 99
VB.NET Custom Action · 86
VBScript Action debuggen · 101
VBScript Custom Action · 90
Veröffentlichen einer Anwendung · 13
Verzeichnisvariablen · 234
View-Klasse · 175
ViewModel-Klasse · 168
Votive · 15

W

Wca · 88
WcaAddTempRecord · 90
WcaDoDeferredAction · 109
WcaGetProperty · 90
WcaGetTargetPath · 88, 90
WcaGlobalInitialize · 90
Wca-Library · 90, 109
WcaLog · 88
WcaReadIntegerFromCaData · 110
WcaReadStreamFromCaData · 110
WcaReadStringFromCaData · 110
Wca SetProperty · 90
WcaWriteIntegerToCaData · 109
WcaWriteStreamToCaData · 109
WcaWriteStringToCaData · 109
Webseite · 134
WHERE · 118, 124
WHS · Siehe Windows Scripting Host, Siehe
Windows Scripting Host
WiLangId.vbs · 199
WiLogUtil.exe · 196
wilstxfm.vbs · 198
Windows Installer Best Practices · 21, 131
Windows Installer SDK · 16, 198, 210, 211
Windows Presentation Foundation · WPF
Windows Scripting Host · 90, 92
WindowsFolder · 235
WindowsVolume · 235
wisubstg.vbs · 199
Wix Extension · 58
WixBalExtension · 152
WixCop.exe · 244
WixDifxAppExtension · 184
WixElement
 AdminExecuteSequence · 73
 AdminUISequence · 73

AdvtExecuteSequence · 73
 AdvtUISequence · 73
 Billboard · 72
 BillboardAction · 72
 CDATA · 91, 92
 Column · 115
 Component · 22
 ComponentGroup · 42, 137
 ComponentGroupRef · 42
 Condition · 34, 38
 CreateFolder · 190
 Custom · 83
 CustomAction · 79, 82, 90
 CustomTable · 115
 Data · 117
 Dependency · 227
 DigitalCertificate · 180
 DigitalSignature · 180
 Directory · 24
 DirectoryRef · 25, 41
 DirectorySearch · 38
 Driver · 185
 Element FirewallException · 222
 EnsureTable · 227, 233
 Environment · 135
 Extension · 51
 Family · 212
 Feature · 25
 File · 23
 FileSearch · 38
 FileShare · 190
 FileSharePermission · 190
 FirewallException · 222
 Fragment · 40
 Group · 189
 GroupRef · 189
 HeatDirectory · 130
 Icon · 24
 IgnoreModularization · 228
 iis
 Certificate · 183
 IniFile · 51
 IniFileSearch · 37
 InstallExecuteSequence · 73, 78
 InstallUISequence · 78
 Instance · 201
 InstanceTransforms · 201
 MajorUpgrade · 206
 Media · 22, 180
 MediaTemplate · 21
 Merge · 226
 MergeRef · 226
 ODBCDataSource · 54
 ODBCDriver · 55
 Package · 203
 Patch · 214
 PatchBaseline · 215
 PatchCertificates · 220
 PatchCreation · 211
 PatchInformation · 212
 PatchMetadata · 212
 PatchSequence · 214
 Permission · 139
 PermissionEx · 143, 144
 Product · 202
 ProgId · 51
 Property · 33, 54, 55
 PropertyRef · 127
 RegistryKey · 50
 RegistrySearch · 37
 RegistryValue · 50
 RemoveFile · 56, 191, 204
 RemoveFolder · 25, 191
 RemoveFolderEx · 191
 Row · 117
 ServiceControl · 138
 ServiceInstall · 137, 144
 SetDirectory · 80
 String · 65
 Subscribe · 72
 TargetImage · 213
 UI · 58
 Upgrade · 207
 UpgradelImage · 213
 User · 189
 Verb · 51
 WixLocalization · 64
 WixVariable · 43, 231
 WiX-Extension · 58
 WixFirewallExtension · 222
 WixIISExtension · 183
 WixMbaPrereqPackageId · 179
 wixmsp · 215
 WixNetFxExtension · 127, 160
 wixpdb Datei · 210, 215
 WixQueryOsWellKnownSID · 139, 189
 WixStdBA · 151, 161
 WixUI_Advanced · 58, 66, 243
 WixUI_FeatureTree · 58, 241
 WixUI_InstallDir · 58, 240
 WixUI_Minimal · 58, 242
 WixUI_Mondo · 58, 81, 238
 WixUIBannerBmp · 60
 WixUIDialogBmp · 60
 WixUIExclamationIcon · 60
 WixUIExtension · 58
 WixUIInfolico · 60
 WixUILicenseRtf · 60
 WixUINewIcon · 60
 WixUIUpIcon · 60
 WixUnit.exe · 244
 WixUtilExtension · 113, 139, 144, 152, 156, 186, 189
 WPF · 161

X

X.509-Zertifikat · 180
 XAML · 175

XPath · 187
XSLPattern · 187

Z

Zertifizierungsstelle · 181
Zuweisen von Anwendungen · 13