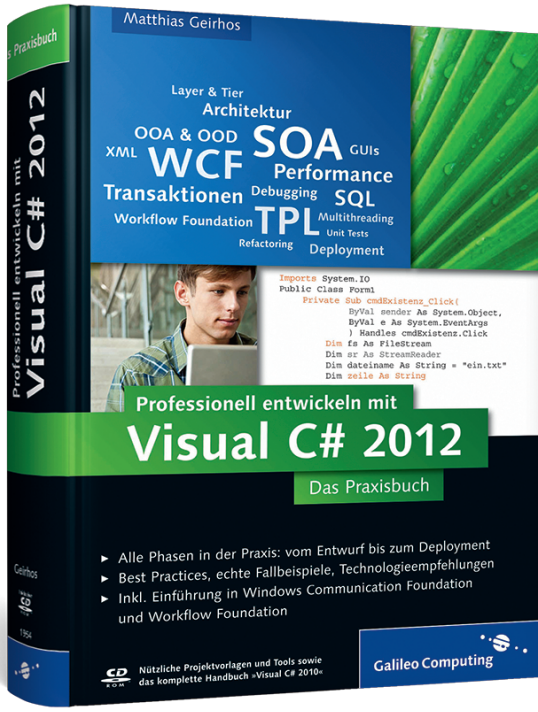
Matthias Geirhos

Professionell entwickeln mit Visual C# 2012

Das Praxisbuch



Galileo Press

Auf einen Blick

1 Einführung 25

2 Softwarearchitektur 37

3 Softwaredesign 131

4 .NET für Fortgeschrittene 241

5 Professionell codieren 361

6 Windows Communication Foundation 433

7 Datenbank und Datenzugriff 625

8 Workflow Foundation 731

9 Windows 8 und WinRT 913

10 Softwaretests 987

11 Softwarepflege 1077

Inhalt

Vorwort zur zweiten Auflage 21
Vorwort zur ersten Auflage 23

1 Einführung 25

1.1 Lehre und Praxis – der Unterschied 25
1.1.1 Gute Software, schlechte Software 26
1.1.2 Wege zur Lösung 27
1.2 Das Fallbeispiel 29
1.3 Die einzelnen Kapitel 30

2 Softwarearchitektur 37

2.1 Einführung 38
2.1.1 Das Problem 38
2.1.2 Gute Softwarearchitektur, schlechte Softwarearchitektur 39
2.1.3 Aufgaben 40
2.1.4 Anwendungstypen 41
2.1.5 Der Architekt 43
2.2 Anforderungen 44
2.2.1 Arten von Anforderungen 45
2.2.2 Anatomie einer Anforderung 48
2.2.3 Das richtige Maß 49
2.3 Komponenten 51
2.3.1 Komponenten identifizieren 51
2.3.2 Beziehungen 53
2.4 Prozesse 55
2.4.1 Was ist ein Prozess? 55
2.4.2 Geschäftsprozessmodellierung 56
2.4.3 Auswirkungen auf die Architektur 57
2.5 Layer (Schichten) 59
2.5.1 Grundlagen 59
2.5.2 Layer vs. Tier 63

2.5.3 Die Fassade ... 63
2.5.4 Presentation Layer ... 64
2.5.5 Business Layer ... 67
2.5.6 Data Layer ... 71

2.6 Tier und verteilte Software ... 76
2.6.1 Gründe für oder gegen Verteilung ... 76
2.6.2 Designmerkmale verteilter Architekturen ... 78
2.6.3 Ebenen für die Verteilung ... 79
2.6.4 Die wichtigsten Fragen für Architekten ... 80

2.7 Designmerkmale ... 81
2.7.1 Kopplung ... 82
2.7.2 Ausfallsicherheit ... 83
2.7.3 Performance ... 86
2.7.4 Sicherheit ... 91
2.7.5 Validierung ... 94
2.7.6 Lokalisierung ... 97
2.7.7 Statusinformationen ... 98
2.7.8 Interoperabilität und Integration ... 98
2.7.9 Die Admin-Sichtweise ... 102
2.7.10 Transaktionen und Gleichzeitigkeit (Concurrency) ... 106
2.7.11 Fehlerbehandlung ... 112

2.8 Architekturmodelle ... 114
2.8.1 Monolithische Anwendungen ... 114
2.8.2 Client-Server-Architektur ... 115
2.8.3 Mehrschichtige Anwendungen ... 115
2.8.4 Serviceorientierte Architekturen (SOA) ... 115

2.9 Vorgehensweise ... 119
2.9.1 Schritt 1: Architekturziele definieren ... 120
2.9.2 Schritt 2: Umfeld analysieren ... 120
2.9.3 Schritt 3: Entwurf, Review und Iteration ... 121
2.9.4 Schritt 4: Technologieentscheidung ... 121
2.9.5 Schritt 5: Dokumentation ... 121

2.10 Dokumentation ... 121
2.10.1 Was eine gute Dokumentation auszeichnet ... 122
2.10.2 Modelle ... 123
2.10.3 Inhalt ... 126

2.11 Was noch zu sagen wäre ... 129

3 Softwaredesign 131

3.1 Grundlegende Designziele 132
3.1.1 Erfüllung der Funktion 132
3.1.2 Zuverlässigkeit und Robustheit 133
3.1.3 Wartbarkeit 133
3.1.4 Erweiterbarkeit 134

3.2 Objektorientierte Analyse und Design 135
3.2.1 Gründe und Aufgaben der OOA/OOD 136
3.2.2 Das Fallbeispiel 137
3.2.3 Abstraktion und Hierarchie 138
3.2.4 Objekte und Klassen 140
3.2.5 Beziehungen 151
3.2.6 War es das? 159

3.3 Designentscheidungen 160
3.3.1 Gutes Design, schlechtes Design 160
3.3.2 Exception-Handling 167
3.3.3 Logging 169
3.3.4 Datenmodell 169

3.4 Schnittstellen und Integration 169
3.4.1 Integration 172
3.4.2 Unterscheidungsmöglichkeiten 173
3.4.3 Mindeststandards 176

3.5 Benutzeroberflächen 180
3.5.1 Die richtige Technologie 182
3.5.2 Anforderungen an eine gute grafische Benutzeroberfläche 183
3.5.3 Fallstudie: Entwicklung des Kalimba.KeyAccount Systems 192
3.5.4 Der »Rest« 196

3.6 Konfiguration 197
3.6.1 Grundlegendes zu Konfigurationen 198
3.6.2 .NET-Konfigurationsdateien 201
3.6.3 Datenbankkonfiguration 210

3.7 Vom Umgang mit der Zeit in Anwendungen 212
3.7.1 Dauer und Wahrnehmung 214
3.7.2 Anwenderfreundliche Informationsdialoge 219
3.7.3 Sonstige Tipps und Tricks 221

3.8 Tutorial: Die Enterprise Library 224
3.8.1 Der Logging Application Block 225
3.8.2 Der Exception Handling Application Block 233

4 .NET für Fortgeschrittene 241

4.1 Parallele Verarbeitung 241
4.1.1 Wann lohnt sich parallele Verarbeitung überhaupt? 242
4.1.2 Parallelität in der Praxis 244
4.1.3 Was sind Threads? 247
4.1.4 Multithreading in .NET 249
4.1.5 Klassische Threads 250
4.1.6 Thread Pools 257
4.1.7 Timer 258
4.1.8 Task Parallel Library 259
4.1.9 Async und Await 272
4.1.10 Thread-Synchronisierung 281

4.2 Fehlerbehandlung 291
4.2.1 Was ist eine Exception? 292
4.2.2 Der Status 293
4.2.3 Try 293
4.2.4 Catch 294
4.2.5 Finally 302
4.2.6 System.Exception 303
4.2.7 Eigene Exception-Klassen 304
4.2.8 Zum Schluss 305

4.3 Reguläre Ausdrücke 307
4.3.1 Reguläre Ausdrücke in .NET 308
4.3.2 Alternative und Gruppierung 310
4.3.3 Nach reservierten Zeichen suchen 310
4.3.4 Zeichenauswahl 311
4.3.5 Quantifizierer 312
4.3.6 Kontextsensitive Bedingungen 313
4.3.7 Suchoptionen 315
4.3.8 Gruppen 315

4.4 Lambda-Ausdrücke 317
4.4.1 Delegaten 317
4.4.2 Lambda-Ausdrücke 319
4.4.3 Func- und Action-Delegaten 320
4.4.4 Lambda-Ausdrücke vs. anonyme Methoden 321
4.4.5 Expression Tree 322
4.4.6 Babylon revisited 323

4.5 Transaktionen ... 327
4.5.1 Wozu Transaktionen? ... 327
4.5.2 Merkmale von Transaktionen ... 328
4.5.3 TransactionScope ... 329
4.5.4 Committable Transaction ... 332
4.5.5 Tracing ... 334

4.6 Erweiterungsmethoden ... 335
4.6.1 Erweiterungsmethoden schreiben ... 336
4.6.2 Der Compiler ... 337
4.6.3 Vorteile, Nachteile, Empfehlungen ... 338

4.7 Serialisierung ... 339
4.7.1 Ein Beispiel ... 340
4.7.2 Serialisierer ... 342
4.7.3 BinaryFormatter ... 343
4.7.4 XmlSerializer ... 345

4.8 Automatische Speicherverwaltung ... 348
4.8.1 Speicherzuteilung ... 348
4.8.2 Garbage Collection ... 349
4.8.3 Finalisierung ... 355
4.8.4 Monitoring ... 358

5 Professionell codieren 361

5.1 Was ist sauber und strukturiert? ... 362
5.1.1 Die grundlegenden Probleme ... 362
5.1.2 Was zeichnet guten Code aus? ... 365

5.2 Code-Styleguides ... 367

5.3 Gut benennen ... 368
5.3.1 Deutsch oder Englisch ... 369
5.3.2 Aussagekräftig ... 369
5.3.3 Einzahl oder Mehrzahl ... 372
5.3.4 camelCase ... 373
5.3.5 Leerwörter, Weasle-Words und reservierte Wörter ... 373
5.3.6 Feste Namenskonventionen ... 374
5.3.7 C#-Konventionen ... 374

5.4 Sauber formatieren ... 376
5.4.1 Struktur ... 376

5.4.2 Formatierung ... 380

5.5 Sinnvoll kommentieren ... 386
5.5.1 Selbstverständliches ... 388
5.5.2 Kürze und Prägnanz ... 388
5.5.3 // vs. /* */ ... 388
5.5.4 //todo ... 389
5.5.5 Kommentare in Visual Studio ... 389
5.5.6 Ort ... 391
5.5.7 Die richtige Zeit ... 391
5.5.8 Aktualität ... 391

5.6 Klassen und Klassenhierarchien ... 391
5.6.1 Klasse oder Schnittstelle? ... 391
5.6.2 Klasse oder struct? ... 393
5.6.3 Klassengröße ... 394
5.6.4 Zuständigkeit ... 394
5.6.5 Erweiterbarkeit ... 395
5.6.6 Abstrakte Klassen ... 398
5.6.7 Statische Klassen ... 398

5.7 Funktionen ... 398
5.7.1 Funktionsgröße ... 399
5.7.2 Zuständigkeit ... 399
5.7.3 Konstruktoren ... 399
5.7.4 Eigenschaft oder Funktion? ... 400
5.7.5 Parameter ... 402
5.7.6 Erweiterungsmethoden ... 403

5.8 Schnittstellen ... 404

5.9 Enums ... 405

5.10 Eigenschaften ... 406

5.11 Exceptions ... 407
5.11.1 Wann? ... 408
5.11.2 Wo? ... 408
5.11.3 Wie? ... 409

5.12 Refactoring ... 411
5.12.1 Gründe ... 411
5.12.2 Code-Smells ... 412
5.12.3 Der Prozess ... 413
5.12.4 Tools ... 414
5.12.5 Refactoring-Muster ... 415

5.13 Aus der Praxis: Codeanalyse in Visual Studio 2012 ... 420

5.14 Tutorial: Snippet Designer ... 425

6 Windows Communication Foundation 433

6.1 Services im Überblick ... 434

6.2 Der erste WCF-Service ... 440

6.2.1 Das Projekt ... 440

6.2.2 Der Service ... 442

6.2.3 Die Konfigurationsdatei web.config ... 445

6.2.4 Der Client ... 446

6.3 Anatomie eines WCF-Service ... 451

6.3.1 Endpunkte ... 451

6.3.2 Adressen ... 459

6.3.3 Binding ... 460

6.3.4 Contracts ... 463

6.3.5 Konfiguration ... 464

6.3.6 Transportsitzung ... 470

6.4 Hosting ... 471

6.4.1 Merkmale eines gutes Hosts ... 472

6.4.2 Visual Studio 2012 ... 473

6.4.3 Selfhosting ... 474

6.4.4 NT-Services ... 478

6.4.5 IIS ... 483

6.4.6 WAS ... 490

6.4.7 AppFabric ... 495

6.4.8 Entscheidungshilfe ... 498

6.5 Clients ... 499

6.5.1 Proxy erzeugen ... 499

6.5.2 Details zum erzeugten Proxy ... 504

6.5.3 Proxys verwenden ... 507

6.5.4 ChannelFactory ... 514

6.6 Services im Detail ... 516

6.6.1 Service Contracts ... 516

6.6.2 Data Contracts ... 523

6.6.3 Kommunikationsmuster ... 538

6.7 Fehlerbehandlung ... 546
6.7.1 Grundlagen ... 546
6.7.2 FaultException ... 549
6.7.3 SOAP Faults ... 550

6.8 Transaktionen ... 554
6.8.1 Verteilte Transaktionen ... 555
6.8.2 Transaktionen in WCF ... 560
6.8.3 Wegweiser ... 566
6.8.4 Ressourcenkonflikte ... 568

6.9 Instanzen ... 569
6.9.1 Instanziierungsmodi ... 570
6.9.2 Lastbegrenzung ... 579

6.10 Sicherheit ... 581
6.10.1 Einführung ... 582
6.10.2 Transportweg und Nachricht sichern ... 586
6.10.3 Detailkonfiguration ... 588
6.10.4 Identität ... 592
6.10.5 Autorisierung ... 595

6.11 Aus der Praxis: WCF erweitern ... 598
6.11.1 Schritt 1: Das Projekt einrichten ... 599
6.11.2 Schritt 2: Das Transferobjekt ... 599
6.11.3 Schritt 3: Anhängen des Transferobjekts an eine Nachricht ... 602
6.11.4 Schritt 4: Der Client ... 604
6.11.5 Schritt 5: Der Service ... 605
6.11.6 Schritt 6: Konfiguration ... 605
6.11.7 Schritt 7: Test ... 606

6.12 Tutorial: Message Queuing ... 607
6.12.1 Einführung ... 607
6.12.2 Schritt 1: MSMQ installieren ... 609
6.12.3 Schritt 2: Queues anlegen ... 609
6.12.4 Schritt 3: Projekte einrichten ... 611
6.12.5 Schritt 4: CustomerPortalService ... 611
6.12.6 Schritt 5: SalesPortalService ... 612
6.12.7 Schritt 6: Implementierungen ... 613
6.12.8 Schritt 7: Hosting ... 616
6.12.9 Schritt 8: Konfiguration ... 619
6.12.10 Schritt 9: Tests ... 621

7 Datenbank und Datenzugriff 625

7.1 .NET im SQL Server 627
7.1.1 Vorbereitungen 627
7.1.2 Benutzerdefinierte Datentypen 628
7.1.3 Sicherheit 635
7.1.4 Stored Procedures 636
7.1.5 Benutzerdefinierte Funktionen 638
7.1.6 Trigger 639
7.1.7 Benutzerdefinierte Aggregatfunktionen 642
7.1.8 Einsatz in der Praxis 645

7.2 XML in der Datenbank 647
7.2.1 Tabelle mit XML-Daten erzeugen 648
7.2.2 Daten hinzufügen 648
7.2.3 Daten auslesen 649
7.2.4 Indizes anlegen 651
7.2.5 Daten abfragen 651
7.2.6 Daten modifizieren 654

7.3 Volltextsuche 656
7.3.1 Installation 658
7.3.2 Volltextkatalog anlegen 659
7.3.3 Daten abfragen 662

7.4 Filestream 667
7.4.1 Filestream installieren 668
7.4.2 Filestream aktivieren 669
7.4.3 Datenbank für Filestream einrichten 669
7.4.4 Tabellen um Filestream-Spalten erweitern 671
7.4.5 Dateien ablegen mit SqlFileStream 672
7.4.6 Die Verwaltung der Filestream-Dateien 674
7.4.7 Dateien abrufen 675
7.4.8 Volltext und Filestream 676
7.4.9 Aus der Praxis 677

7.5 Das ADO.NET Entity Framework 678
7.5.1 Einführung 679
7.5.2 Projekt einrichten 685
7.5.3 Das Modell erweitern 689
7.5.4 Daten abfragen 691
7.5.5 Daten hinzufügen und ändern 698
7.5.6 SaveChanges und Gleichzeitigkeit 701
7.5.7 Was noch zu sagen wäre 704

7.6 WCF Data Services ... 704
7.6.1 Übersicht ... 705
7.6.2 Einfachen WCF Data Service erstellen ... 706
7.6.3 WCF Data Service testen ... 708
7.6.4 Zugriff aus einer .NET-Anwendung ... 710
7.6.5 Empfehlungen für den Einsatz ... 716

7.7 LINQ to XML ... 719
7.7.1 Statische XML-Dateien erstellen ... 719
7.7.2 XML-Dateien aus vorhandenen Strukturen erstellen ... 722
7.7.3 Erweiterungsmethoden ... 723
7.7.4 XML-Dateien laden ... 724
7.7.5 Abfragen ... 725
7.7.6 XML-Daten verändern ... 727
7.7.7 Anwendung in der Praxis ... 728

7.8 Was noch zu sagen wäre ... 728

8 Workflow Foundation 731

8.1 Einführung ... 731
8.1.1 Warum Workflows? ... 732
8.1.2 Der Workflow ... 738
8.1.3 Workflow Designer ... 742
8.1.4 Windows Workflow Foundation im Überblick ... 748

8.2 Fallbeispiel ... 751

8.3 Der erste sequenzielle Workflow ... 752
8.3.1 Das Projekt einrichten ... 752
8.3.2 Den Workflow gestalten ... 754
8.3.3 Der weitere Ausbau ... 761

8.4 Der erste Flowchart-Workflow ... 763
8.4.1 Wareneingang reloaded ... 763
8.4.2 Den Wareneingangs-Workflow umbauen ... 764

8.5 Workflows laden und ausführen ... 768
8.5.1 Workflows in XAML ausführen ... 769
8.5.2 Workflows in Code ausführen ... 769
8.5.3 WorkflowApplication ... 770

8.6 Eingebaute Aktivitäten verwenden ... 772
8.6.1 Auflistung ... 772

8.6.2 Parallele Verarbeitung ... 778
8.6.3 Fehlerbehandlung ... 783
8.6.4 Ausführungssteuerung ... 786
8.6.5 Ereignissteuerung ... 793
8.6.6 TerminateWorkflow ... 799
8.6.7 Sonstige Aktivitäten ... 800

8.7 Eigene Aktivitäten entwickeln ... 800
8.7.1 Aktivitäten im Überblick ... 801
8.7.2 Lebenszyklus ... 805
8.7.3 CodeActivity ... 806
8.7.4 CodeActivity mit Rückgabewert ... 810
8.7.5 CodeActivity mit Validierung ... 811
8.7.6 NativeActivity ... 815
8.7.7 ActivityDesigner ... 820
8.7.8 Bookmarks ... 825
8.7.9 Was noch zu sagen wäre 827

8.8 Transaktionen ... 827
8.8.1 TransactionScope ... 828
8.8.2 Kompensationen ... 831

8.9 Persistenz ... 838
8.9.1 InstanceStore ... 839
8.9.2 SQL Server einrichten ... 839
8.9.3 Änderungen an der Workflow-Anwendung ... 840
8.9.4 Speichern im Code ... 842
8.9.5 Persistenzschutz ... 842

8.10 Tracking und Tracing ... 843
8.10.1 Tracking-Grundlagen ... 843
8.10.2 Tracking-Objekte ... 846
8.10.3 Fallbeispiel ... 847
8.10.4 Tracing ... 854

8.11 Workflow Services ... 857
8.11.1 Grundlagen ... 857
8.11.2 Aktivitäten ... 863
8.11.3 Fallbeispiel – Teil 1: Der Laborservice ... 865
8.11.4 Fallbeispiel – Teil 2: WF ruft WCF ... 873
8.11.5 Fallbeispiel – Teil 3: Der Laborclient ... 876
8.11.6 Fallbeispiel – Teil 4: WCF ruft WF ... 877
8.11.7 Fallbeispiel – Teil 5: Persistence ... 888
8.11.8 Correlation ... 890
8.11.9 Zum Schluss ... 892

8.12 State Machine Workflows ... 893
8.12.1 Anfangszustand ... 895
8.12.2 Endzustand ... 897
8.12.3 Zustände dazwischen ... 898
8.12.4 Zustandsübergänge ... 899
8.12.5 Übungsempfehlung ... 903

8.13 Designer Rehosting ... 904
8.13.1 Fallbeispiel ... 905
8.13.2 Den Designer einbinden ... 905
8.13.3 Die Toolbox bestücken ... 909

9 Windows 8 und WinRT 913

9.1 Einführung ... 915
9.1.1 Laufzeitvoraussetzungen ... 915
9.1.2 Das Windows 8-Design ... 917
9.1.3 Deployment und der Windows 8 App Store ... 922
9.1.4 Prozesse in WinRT und das Windows Application Model ... 923

9.2 Fallbeispiel ... 925

9.3 Projekt einrichten ... 926
9.3.1 Voraussetzungen ... 926
9.3.2 Templates ... 926
9.3.3 Projekt anlegen und einrichten ... 929

9.4 Seiten hinzufügen ... 933
9.4.1 Das Navigationskonzept ... 934
9.4.2 Seiten hinzufügen ... 934
9.4.3 Startseite festlegen ... 936
9.4.4 Anwendung starten ... 937

9.5 Daten hinzufügen ... 937
9.5.1 Klassenmodell ... 938
9.5.2 Von XML in Klassenhierarchie laden ... 940
9.5.3 Ressource hinzufügen ... 941
9.5.4 Daten beim Aufruf der App laden ... 942
9.5.5 Daten an Steuerelement binden ... 943

9.6 Die Lexikonseite ... 945
9.6.1 Allgemeines zur Navigation ... 945
9.6.2 Navigation zur Lexikonseite ... 946

9.6.3 Lexikonseite: Produkte anzeigen ... 947
9.6.4 Lexikonseite: Lexikoneintrag anzeigen ... 950

9.7 Die Bestellseite und die App Bar ... 954
9.7.1 App Bars in eigenen Anwendungen ... 954
9.7.2 Eine App Bar hinzufügen ... 955

9.8 Die Warenkorbseite ... 957
9.8.1 Die Anzeige ... 957
9.8.2 Die App Bar ... 958
9.8.3 Änderungen am Datenmodell ... 959

9.9 Die Bestellbestätigungsseite ... 961

9.10 Lebenszyklus- und Zustandsmanagement ... 962
9.10.1 Einführung ... 963
9.10.2 Anwendungsdaten ... 963
9.10.3 Sitzungsdaten – Framenavigation ... 964
9.10.4 Sitzungsdaten – Zustand der Seiten ... 968

9.11 Contracts ... 969
9.11.1 Die Suche ... 970
9.11.2 Die Suche implementieren ... 971
9.11.3 Testen ... 976

9.12 Für verschiedene Layouts entwickeln ... 977
9.12.1 Der Simulator ... 977
9.12.2 Verschiedene Formate ... 979
9.12.3 Funktionsweise ... 983
9.12.4 Empfehlungen ... 984

10 Softwaretests ... 987

10.1 Grundlagen ... 989
10.1.1 Ziele und Aufgaben ... 989
10.1.2 Übersicht und Einteilung der Tests ... 992
10.1.3 Vom richtigen Zeitpunkt ... 996
10.1.4 Der Tester und sein Team ... 999
10.1.5 Der Testablauf ... 1003
10.1.6 Kleine Fehlerkunde ... 1011

10.2 Testplanung und -organisation ... 1017
10.2.1 Release-Management ... 1018

10.2.2 Das Testteam ... 1019
10.2.3 Testfälle ... 1024

10.3 Testumgebung ... 1028
10.3.1 Voraussetzungen ... 1028
10.3.2 Die zu testende Software ... 1029
10.3.3 Daten ... 1030
10.3.4 Rechner und Betriebssystem ... 1032
10.3.5 Server- und Zusatzkomponenten ... 1034
10.3.6 Tools ... 1034

10.4 Testverfahren und -werkzeuge ... 1034
10.4.1 Exploratives Testen ... 1035
10.4.2 Test-to-pass vs. test-to-fail ... 1035
10.4.3 Äquivalenzklassenbildung ... 1036
10.4.4 Grenzwerte ... 1037
10.4.5 Sinnlose Daten ... 1039
10.4.6 Programmzustände ... 1040
10.4.7 Entscheidungstabellen ... 1041
10.4.8 Ablaufpläne ... 1042
10.4.9 Geschäftsprozessmodelle ... 1043
10.4.10 Continuous Delivery ... 1043

10.5 Testarten ... 1048
10.5.1 Test der Spezifikation ... 1048
10.5.2 Unit-Test ... 1050
10.5.3 Komponententest ... 1053
10.5.4 Usability-Test ... 1054
10.5.5 Systemtest ... 1057
10.5.6 Feldtest ... 1058
10.5.7 Abnahmetest ... 1060
10.5.8 Codereview ... 1061
10.5.9 Der Rest ... 1064

10.6 Workshop: Unit-Tests mit Visual Studio ... 1066
10.6.1 Anlegen eines Testprojekts ... 1066
10.6.2 Hinzufügen der Unit-Tests ... 1068
10.6.3 Codeabdeckung ... 1071
10.6.4 Praktische Empfehlungen ... 1073

11 Softwarepflege 1077

11.1 Release Management ... 1078
11.1.1 Begriffe ... 1078
11.1.2 Der Release-Prozess ... 1079

11.2 Anforderungen ... 1087
11.2.1 Einführung ... 1088
11.2.2 Die verschiedenen Sichtweisen ... 1090
11.2.3 Anforderungen an eine Anforderung ... 1092

11.3 Zeitschätzung ... 1096
11.3.1 Was ist eine Zeitschätzung? ... 1097
11.3.2 Herausforderungen einer Zeitschätzung ... 1098
11.3.3 Die lernende Organisation ... 1106
11.3.4 Woher kommen Zeitüberschreitungen? ... 1108
11.3.5 Methoden der Zeitschätzung ... 1112

Zum Schluss ... 1123
Index ... 1125

Kapitel 1
Einführung

Wo die Praxis des Lebens fehlt, ist das Studium immer nur eine halbtätige Arbeit. (August Graf von Platen Hallermund)

In diesem ersten Kapitel erfahren Sie etwas zur Motivation für dieses Buch, über das verwendete Fallbeispiel, und ich stelle Ihnen die einzelnen Kapitel vor.

1.1 Lehre und Praxis – der Unterschied

Erinnern Sie sich noch an die erste Generation der sogenannten *RAD-Tools*? Programme sollten zusammengeklickt statt programmiert werden. Für alle vorgefertigten Probleme wären fertige Bausteine in der Toolbox, und an die Stelle von individuellem Code sollten gut lesbare, hübsche Diagramme treten. Mit einem Satz an Regeln sollten sich alle denkbaren Probleme lösen lassen, schnell, fehlerfrei und ohne großen Einarbeitungsaufwand, sogar durch die Fachabteilung selbst.

Was theoretisch gut klang, scheiterte an der Praxis, an dem Variantenreichtum der Aufgabenstellungen, der Vielzahl an Produkten auf dem Markt und, nicht zuletzt, an der Phantasie der Produktmanager. Softwareentwicklung ist eben nicht mit anderen Ingenieurdisziplinen vergleichbar, in denen es feste Regeln und Normen gibt, die oft über lange Zeiträume entwickelt wurden. Die Herangehensweisen an ein Problem sind höchst unterschiedlich, und der daraus resultierende Code ist bei keinen zwei Entwicklern identisch.

Softwareentwicklung ist jedoch kein wissenschaftsfreies Gebiet – im Gegenteil, die Kenntnis der theoretischen Zusammenhänge erleichtert die Entwicklung ungemein. Aber alle Theorie muss sich in den Kontext der realen Aufgabenstellung einfügen. Was die Statik in der Architektur der Gebäude verbietet, muss die Softwarearchitektur trotzdem möglich machen. Der Markt bestimmt Aussehen und Funktionalität einer Software, und er nimmt wenig Rücksicht auf die Sachzwänge der eingesetzten Technologien.

Anstelle fester Lösungswege gibt es Best Practices und Patterns, Handlungsempfehlungen und Lösungsskizzen, die sich wie Bausteine in eigenen Projekten verwenden lassen, aber bei Bedarf noch zurechtgeklopft werden müssen.

Inzwischen gibt es ein neues RAD-Tool von Microsoft, Visual Studio LightSwitch, das sowohl als eigenständiges Werkzeug zu erwerben als auch in höheren Versionen von Visual Studio enthalten ist. Interessant an diesem Produkt ist dessen modularer Charakter, und es ist zu erwarten, dass es für immer mehr praktische Aufgabenstellungen fertige Module geben wird. Und die nahtlose Integration von eigenem Code in eigene LightSwitch-Projekte, die genau das akzeptiert, was unvermeidbar ist: Je komplexer die Anforderungen werden, desto komplexer werden auch die Konfigurationen solcher RAD-Tools – bis zu einem Punkt, an dem es einfacher ist, eigenen Code zu erstellen, als sich durch Dutzende von Dialogen mit unzähligen Optionen zu klicken.

1.1.1 Gute Software, schlechte Software

Vor gut zehn Jahren hatte ich den ersten Kontakt mit einem damals neuartigen Online-CMS (Content Management System). Es war kurz vorher bei einem Unternehmen eingeführt worden, in dem ich als Leiter der Online-Entwicklung eingestellt wurde. Es stammte von einem renommierten Unternehmen, hatte einen klangvollen Namen, und die Verkäufer konnten mit einer langen Referenzliste aufwarten. Es war gruppenweit für einen stattlichen sechsstelligen Betrag erworben worden. Meine Aufgabe sollte es sein, auf Basis dieses Systems zahlreiche neue Online-Auftritte zu entwickeln, gemeinsam mit eigenen und freiberuflichen Mitarbeitern.

Für den Einkäufer dieses CMS war das eine sichere Sache, sollte man meinen. Im praktischen Einsatz wurden sehr bald die Mängel deutlich, und die waren gravierend: Das System lahmte selbst auf den schnellsten Servern, es war instabil, ganz und gar kontra-intuitiv zu bedienen, schwer zu administrieren und mit allerlei eigenen, proprietären Lösungen für ganz alltägliche Probleme ausgestattet.

Hotline, Entwicklung und Geschäftsleitung des Softwareherstellers hatten das Problem längst ausgemacht: Der Kunde sei das Problem, das System liefe schließlich in großen Installationen zuverlässig und, soweit man wisse, problemlos. Das wollte ich nicht recht glauben, und so begann ich, mit diesen Referenzkunden zu telefonieren und sie zu besuchen. Und wie es zu erwarten war, glichen sich die Probleme. Viele Unternehmen berichteten über genau die Schwierigkeiten, die auch wir hatten.

Und so war das Ende unausweichlich: Nach einigen erfolglosen Releases tauschten wir das System gegen ein CMS aus, das auf offenen Standards wie PHP und MySQL aufbaute, kaum ein Zwanzigstel so teuer und wenigstens zehnmal so schnell war. Dieses System läuft auch heute noch.

Seitdem habe ich mich in zahlreichen Projekten immer wieder gefragt: Was macht gute Software aus, und was unterscheidet sie von schlechter Software? Warum lässt sich die eine Software elegant und preiswert erweitern, während sich andere Programme vehement gegen jede Form der Anpassung sträuben? Warum gibt es Lösun-

gen, mit denen sich flüssig arbeiten lässt, während bei anderen jeder Klick zum Geduldsspiel wird, und warum gibt es so viele gute Benutzeroberflächen, aber warum auch so viele schlechte? Wieso kann der Aufwand für das Customizing einen Bruchteil des Kaufpreises ausmachen, aber auch leicht ein Mehrfaches, und warum müssen manche Softwareprodukte bei jedem neuen Release in großen Teilen neu geschrieben werden, während andere über viele Jahre hinweg scheinbar mühelos erweitert werden können?

1.1.2 Wege zur Lösung

Ich behaupte nicht, die Antwort auf all diese Fragen zu kennen. Aber ich habe über die Jahre zahlreiche Muster entdeckt, Fallstricke, wenn Sie es so nennen möchten, aus den eigenen Projekten heraus, aber vor allem in der Zusammenarbeit mit meinen Mitarbeitern, externen Programmierern, Auszubildenden und Consultants. Aus meinen und ihren Erfahrungen ist dieses Buch entstanden. Es verfolgt drei Ziele:

Wissen erweitern

Gerade in kleineren Unternehmen müssen Entwickler oft wahre Allrounder sein. Denn neben fundierten Fertigkeiten in verschiedenen Programmiersprachen benötigen Sie noch viele weitere Kenntnisse, beispielsweise in Fragen der Softwarearchitektur, im Softwaredesign, in ihrer Entwicklungsumgebung, in den verschiedensten eingesetzten Technologien, in Softwaretests, Projektmanagement und Datenbankentwurf.

Nicht jeder Entwickler hat die Zeit, Lust oder Gelegenheit, sich durch viele Regalmeter Fachliteratur zu arbeiten. Dieses Buch macht den durchaus gewagten Versuch, die wichtigsten Bereiche der Softwareentwicklung in einem einzigen Werk zu behandeln, von der richtigen Architektur über die Umsetzung bis hin zur Softwarepflege nach deren Einführung. Damit ist es ein Buch für »Aufsteiger«, also Entwickler, die dazulernen möchten und die über Bekanntes aus einem anderen Blickwinkel neu nachdenken wollen.

Probleme vermeiden helfen

In diesem Buch finden Sie immer wieder Kästen mit dem Titel »Aus der Praxis«. In diesen Kästen finden Sie das eine oder andere Problem, das ich in der Vergangenheit selbst erlebt habe, und meist einen Lösungsansatz dazu. Vielleicht erkennen Sie bisweilen Ihr eigenes Projekt darin wieder, dann betrachte ich Sie als Leidensgenossen, wenn Sie erlauben. Ansonsten täten Sie mir einen großen Gefallen, wenn Sie diese Probleme vermeiden würden.

Ich habe beim Schreiben bisweilen selbst gestaunt, wie viele Fehler mir während der Jahre unterlaufen sind. Viele wären leicht zu vermeiden gewesen, bei anderen lagen ihre Ursachen etwas tiefer.

Best Practices vermitteln

Der Begriff *Best Practices* stammt eigentlich aus der Betriebswirtschaft und bezeichnet bewährte Verfahren, die sich für gleiche oder ähnliche Aufgabenstellungen eignen. *Entwurfsmuster* oder *Design Patterns* sind fertige Lösungen oder Lösungsschablonen für häufig wiederkehrende Probleme in der Softwareentwicklung. Sie können damit ebenfalls im weiteren Sinne den Best Practices zugerechnet werden.

Myriaden von Problemen entstehen durch die falsche oder unzureichende Implementierung häufig wiederkehrender Aufgabenstellungen. Nicht selten hat jeder Entwickler seine eigene Lösung entwickelt. Dabei sind viele dieser Aufgabenstellungen im Detail komplex, die Lösungen aber meist zu einfach, wie das folgende Beispiel zeigt.

Aus der Praxis

Um mit Services zu kommunizieren, die mit *WCF (Windows Communication Foundation)* entwickelt wurden, benötigt man einen Proxy, ein Objekt also, das die Kommunikation mit dem Service kapselt. In vielen Lehrbüchern findet man dazu Beispiele wie:

```
MyServiceClient client = new MyServiceClient();
client.Open();
try
{
   client.DoSomething();
}
catch(Exception ex)
{
   MessageBox(...);
}
client.Close();
```

Dieses Beispiel funktioniert und ist daher in vielen Programmen verwirklicht. Es ist aber wenig praxistauglich, denn Verbindungsabbrüche, Timeouts, Sicherheitsprobleme und einige andere auftretende Ausnahmen verlangen nach jeweils speziellen Behandlungen. Das ist unverzichtbar für ein robustes und fehlertolerantes Programm.

Im Grunde genommen müsste sich nun jeder Entwickler in die Tiefen der WCF-Kommunikation begeben, um selbst eine adäquate Lösung zu entwickeln, oder er greift auf das Client-Pattern in Kapitel 6, »Windows Communication Foundation«, zurück.

Leider kommen Best Practices in Studium und Ausbildung meist viel zu kurz. Im Studium mag der mangelnde wissenschaftliche Bezug der Grund dafür sein, in der Ausbildung fehlt oft schlicht die Zeit. Und so weiß ich, da ich seit vielen Jahren ausbilde, dass viele Auszubildende nach ihrer Ausbildung oft das Gefühl haben, erst am Anfang ihrer Entwicklung zu stehen.

1.2 Das Fallbeispiel

An der einen oder anderen Stelle verwende ich ein Fallbeispiel, gewissermaßen als Kontrastprogramm zu den allerorts beliebten »Hallo Welt!«-Beispielen. Herzlich willkommen also in der Welt der *Kalimba Sunfood GmbH*, dem Premium-Importeur für sonnenverwöhnte Früchte aus aller Welt mit Sitz in Hamburg.

Abbildung 1.1 Firmenlogo

Unsere Firma importiert Waren aus aller Welt und benötigt hierfür eine *Enterprise-Resource-Planning*-(*ERP*)-Software zur Steuerung aller betrieblichen Funktionen, also

- Warenwirtschaft,
- Kundenverwaltung,
- Finanz- und Rechnungswesen,
- Verkauf und Marketing,
- Controlling und
- Personalwirtschaft.

Es werden etwa 800 Mitarbeiter beschäftigt, die Hälfte am Stammsitz in Hamburg, die andere Hälfte verteilt auf mehrere Standorte weltweit. Kalimba Sunfood kauft die Früchte vor Ort ein, beispielsweise Orangen aus Brasilien oder Mangos aus Indien,

exportiert sie nach Deutschland und vertreibt sie dort an Großhändler und die Getränkeindustrie. Darüber hinaus betreibt die Gesellschaft einen Onlineshop für Cocktail-Fruchtsäfte, in dem Privatkunden direkt bestellen können.

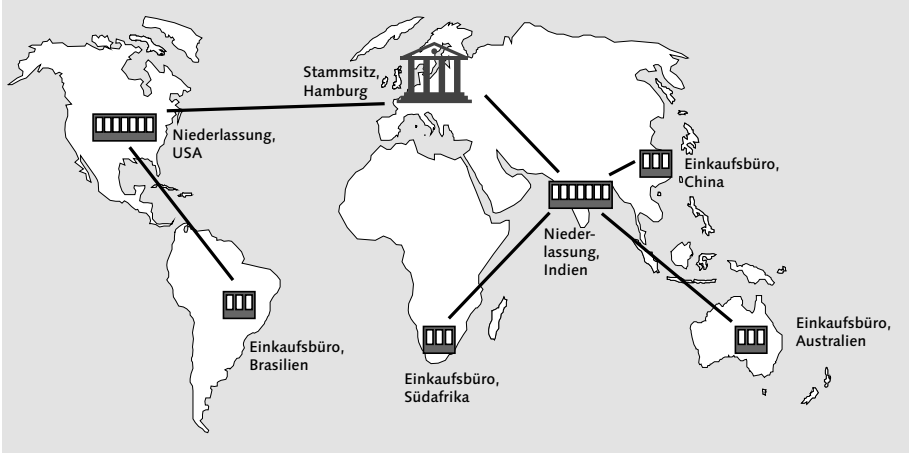


Abbildung 1.2 Standorte (Quelle: CIA Factbook)

Die beiden Niederlassungen in den USA und Indien sind breitbandig an die Zentrale angeschlossen, ihre Mitarbeiter nutzen Rich-Client-Anwendungen. Die Büros der Einkäufer sind schmalbandig angeschlossen, daher greifen die dortigen Mitarbeiter über das Internet auf Webanwendungen zu. Natürlich setzt die hausinterne Entwicklungsabteilung auf .NET (natürlich in der aktuellen Version 4.5) als Kerntechnologie und dazu noch auf

- WinForms für die Rich-Client-Anwendungen,
- ASP.NET für die Webanwendungen,
- WCF für die serviceorientierte Mittelschicht,
- WF für die Verarbeitung der Bestellungs-Workflows,
- Windows Server 2012 als Server-Betriebssystem,
- Visual Studio 2012 als Entwicklungsumgebung und auf
- SQL Server 2012 als Datenbanksystem.

Das zu entwickelnde ERP-System nennt unsere Firma das *Kalimba.ERP*.

1.3 Die einzelnen Kapitel

Das Buch orientiert sich an den Entwicklungsphasen eines Projekts. Da immer mehr Unternehmen agile Prozesse in der Softwareentwicklung einsetzen, beispielsweise Scrum, sind diese Phasen nicht mehr so starr getrennt wie früher. Sie gehen ineinan-

der über und wiederholen sich häufiger in den Teilschritten eines Projekts. Dennoch erfordern alle Phasen spezielles Wissen und ihre jeweils eigenen Fertigkeiten, in größeren Projekten werden sie oft von verschiedenen Personen oder Abteilungen geleitet. Was erwartet Sie auf den folgenden Seiten?

Kapitel 2, »Softwarearchitektur«

Um die *Architekturfindung* geht es im zweiten Kapitel. Architekturthemen sind immer besonders wichtige Themen, da die dort getroffenen Entscheidungen meist weittragend und nur mit hohem Aufwand während oder nach der Implementierung zu ändern sind, wenn das überhaupt möglich ist.

In der Softwarearchitektur geht es darum, die Einheiten oder Komponenten einer Software zu identifizieren und sie zueinander in Bezug zu setzen. Dazu muss das Problem geistig zerlegt, die Strukturen müssen erkannt und Hierarchien gebildet werden. Die *UML* kann hierbei gute Dienste leisten, weswegen wir sie uns aus praktischer Sicht näher ansehen.

Architektur findet auf verschiedenen Ebenen statt. Auf einer unteren Ebene geht es um Komponenten und deren Wiederverwendbarkeit, auf einer höheren Ebene um die innere Struktur des Codes, beispielsweise die Bildung von Layer und Tier. Auf einer der oberen Ebenen schließlich betrachten wir grundlegende Architekturparadigmen, beispielsweise die heute so beliebten serviceorientierten Architekturen.

Gerade beim Thema Softwarearchitektur ist es von großer Bedeutung, von der Theorie zur Praxis zu gelangen. Sie wollen schließlich keine Schaubilder entwickeln, sondern konkrete Software. Die einzelnen Ebenen sollen ineinander übergreifen und eine jede Schicht einen Beitrag zum Ganzen leisten. Mein Fallbeispiel soll dies illustrieren, im Kleinen wie im Großen. Dabei geht es mir weniger um die Darstellung möglichst vieler Technologien als darum, Ihnen die verschiedenen Perspektiven näherzubringen. Und das so, dass Sie die richtigen Fragen stellen können. Deshalb gehe ich zunächst auf Anforderungen ein und weswegen sie bereits in dieser Phase eines Softwareprojekts unverzichtbar sind.

Kapitel 3, »Softwaredesign«

Das dritte Kapitel beschäftigt sich mit wichtigen Designfragen. Es führt das Thema Architektur weiter aus, jedoch auf den unteren Ebenen, den Ebenen der Klassen, Dateien, Assemblys und Projekte. Auch hier gibt es wieder wichtige Ziele, die den Kontext des folgenden Kapitels bilden sollen. Danach geht es um die objektorientierte Analyse und um objektorientiertes Design (OOA/OOD), mit deren Hilfe Klassen gebildet und zueinander in Beziehung gesetzt werden.

Kaum eine Software kommt ohne Schnittstellen aus. Doch gerade hier lauern besonders viele Fallstricke, gilt es doch, meist zwei oder mehrere an sich inkompatible

Systeme miteinander zu verbinden. Leider wird dieser Bereich in der Praxis häufig vollkommen unterschätzt. Dabei ist er doch für so viele Fehlschläge von Softwareprojekten verantwortlich.

Danach geht es um die Benutzeroberfläche. Keine Angst, es erwartet Sie kein Designkurs, und Sie müssen auch nicht zu Papier und Bleistift greifen. Von den vielen Facetten konzentrieren wir uns hier auf diejenigen, die Entwicklern erfahrungsgemäß am meisten Probleme bereiten. Ein häufig unterschätztes Thema ist beispielsweise der Umgang mit der Zeit, also mit Aktionen, die nicht innerhalb der Wahrnehmungsspanne abgeschlossen werden können, sondern ein wenig länger dauern, wie beispielsweise eine Kopieraktion. Hier gilt, wie überhaupt beim GUI-Design, dass ein guter Entwurf kaum mehr Arbeit macht als ein schlechter Entwurf, er erfordert lediglich Wissen und die Bereitschaft, sich darauf einzulassen.

Die Frage nach dem richtigen Speicherort und der richtigen Behandlung von Konfigurationsdaten ist ebenfalls eine ganz praktische und verdient einen eigenen Abschnitt. Zum Schluss zeige ich Ihnen anhand der *Enterprise Library* (*EL*), wie gut gemachte Bibliotheken und Tools gerade diese Phase vereinfachen können.

Kapitel 4, ».NET für Fortgeschrittene«

Ich gehe in diesem Buch davon aus, dass Sie bereits über C#-Kenntnisse verfügen. Dennoch begegne ich oft Entwicklern, denen einige der neueren Sprachkonstrukte und Technologien noch nicht geläufig sind, beispielsweise *LINQ* oder parallele Verarbeitung (*Multithreading*).

Einige der hier behandelten Technologien sind wahre Beschleuniger, beispielsweise reguläre Ausdrücke. Andere wiederum machen erst richtig Spaß, wenn man sie einmal etwas gründlicher betrachtet. Transaktionen sind ein gutes Beispiel dafür. Wiederum andere Technologien sind notwendig (oder sinnvoll) wenn Sie spezielle Arten von Anwendungen entwickeln wollen, zum Beispiel Metro-Apps.

Zu zahlreichen Themen gibt es eigene, manchmal sehr umfangreiche Bücher. Mein Anspruch ist es also nicht, diese Themen umfassend zu behandeln; aber vielleicht ist das gerade der Grund dafür, warum Sie sich bisher nicht damit beschäftigen konnten, und meine Praxis-Einführungen regen Ihr Interesse an. Dann freue ich mich mit Ihnen.

Themenauswahl und Tiefe der behandelten Themen sind mir hier besonders schwergefallen, und von mir aus hätte dieses Kapitel auch noch ein-, zweihundert Seiten mehr haben können. Aber das Konzept des Buches ist ja gerade nicht absolute Vollständigkeit, sondern das Setzen thematischer Schwerpunkte. Daher tröste ich mich also mit dem Gedanken, dass ich die Seitenzahl ohnehin bereits überschritten habe, und hoffe, dass Sie auf der Suche nach C#-Neuigkeiten fündig werden.

Kapitel 5, »Professionell codieren«

Studenten, Schüler und Auszubildende erfahren oft mehr über die *syntaktisch richtige* Benennung von Variablen, als sie behalten können, aber oft wenig bis gar nichts über deren *sinnvolle* Benennung. Sie kennen die Möglichkeiten, Kommentare als solche auszuzeichnen, niemand erklärt ihnen jedoch, wann Kommentare notwendig sind und wie diese aussehen sollten. Auch zur Formatierung des Quellcodes liest und hört man erstaunlich wenig. Das kommt wie gerufen für ein Praxisbuch, und deswegen habe ich in diesem Kapitel viele Tipps und Empfehlungen dazu für Sie gesammelt.

Einen weiteren Schwerpunkt bilden verschiedene Konstrukte der Sprache C#. Erfahren Sie hier beispielsweise, wann Sie statt auf Schnittstellen lieber auf eine Klassenhierarchie setzen sollten oder was es bei Aufzählungen zu berücksichtigen gilt. Das Kapitel liegt mir besonders am Herzen, weil sich schnell Gewohnheiten einschleichen, die nicht immer dem Ziel dienlich sind, qualitativ hochwertigen Code zu erzeugen.

Der wachsenden Bedeutung von *Refactoring* widmet sich ein eigener Abschnitt, denn auf Anhieb »richtig machen« ist nahezu ein Ding der Unmöglichkeit, das Lernen entlang des Weges ein Teil der praktischen Entwicklung. Zum Schluss stelle ich Ihnen mit *FxCop* ein nettes Werkzeug vor, das über die Einhaltung konfigurierbarer Regeln wacht und eigentlich unentbehrlich ist, wenn Ihr Code sauber und wartbar sein soll.

Kapitel 6, »Windows Communication Foundation«

Die Antwort auf die Webservices der Java-Welt nennt Microsoft *Windows Communication Foundation* oder kurz *WCF*. Wer in der .NET-Welt serviceorientiert entwickeln möchte, kommt daran nicht vorbei und muss es auch nicht: WCF bietet eine Menge und lässt sich gut programmieren, erfordert jedoch auch eine andere Herangehensweise von Entwickler und Architekt.

Wenn Sie bisher ausschließlich zweischichtig entwickelt haben, darunter verstehe ich die Anwendung und eine Datenbank (z. B. SQL Server), dann helfen Ihnen vielleicht meine Ratschläge für Neueinsteiger, die ich aus der Lernpraxis meines eigenen Teams gewonnen habe. Dennoch ist Kapitel 6 nicht nur für Einsteiger interessant, sondern behandelt auch fortgeschrittene Themen wie das Hosting über WAS oder die Verwendung von Message Queuing in eigenen WCF-Anwendungen.

Kapitel 7, »Datenbank und Datenzugriff«

In Kapitel 7 geht es um einige fortschrittliche Technologien, die überaus nützlich sind, aber noch relativ selten in realen Projekten eingesetzt werden, wie zum Beispiel die Möglichkeit, .NET-Assemblys in den SQL Server zu laden und dort auszuführen.

Das ist eine praktikable Methode, den Wirkungsbereich von .NET auf das Datenbank-Backend-System auszudehnen. Diese sogenannte CLR-Integration eröffnet ganz neue Möglichkeiten, um beispielsweise bestehenden T-SQL-Code zu migrieren oder datenbanknahe Schnittstellen damit umzusetzen.

Gleiches gilt für XML-Daten, also vorstrukturierte Daten innerhalb der Datenbank, wofür der SQL Server geeignete Werkzeuge zur Abfrage und zur Manipulation bietet. Das ist besonders für Integrationen nützlich. Um XML geht es auch bei LINQ to XML, einer eleganten Möglichkeit, XML-Daten zu schreiben und abzufragen, ohne dass dafür viel Code notwendig wäre.

Weitere Abschnitte beschäftigen sich mit (relativ) neuen Technologien bzw. mit Technologien, die gerade erst erwachsen geworden sind, z. B. das Entity Framework und WCF Data Services. Aber auch die neuen Datenbank-Tools in Visual Studio 2012 sind nicht nur hübsch, sondern auch praktisch. Schon länger gibt es hingegen die Volltextsuche im SQL Server, eine Technologie, die ihre Zeit dennoch noch vor sich hat. Ein Blick darauf lohnt sich auf alle Fälle; sie ist sowohl vielseitig als auch leistungsfähig.

Den Schluss bilden einige ganz praktische Empfehlungen, die dem Umstand geschuldet sind, dass sie (zu) häufig nicht beachtet werden; der richtige Umgang mit Indizes zum Beispiel oder die Bedeutung guter Testdaten.

Kapitel 8, »Workflow Foundation«

Der wesentliche Unterschied zwischen manuellen Prozessen und solchen, die in Software abgebildet sind, liegt in der Synchronisierung. Prozesse in Software sind meist synchron. Eine Aktion wird ausgelöst, und die Verarbeitung wird so lange unterbrochen, bis das Ergebnis vorliegt (*Request/Reply-Pattern*). Manuelle Prozesse hingegen sind oft asynchron, ein Prozess wird gestartet, aber in diesem Moment nicht abgeschlossen, beispielsweise die Genehmigung eines Einkaufs oder der Eingang einer Bestellung (*Fire-and-Forget-Pattern*).

Seit der Version 3.0 gibt es dafür eine Unterstützung in .NET, *Workflow Foundation* genannt. Damit lassen sich solche langlaufenden Workflows abbilden. Ein Workflow kann jederzeit persistiert und später wieder abgerufen und fortgeführt werden.

Workflows in WF lassen sich programmieren oder, praktischer, in XAML beschreiben. Dies hat den Vorteil, den Workflow später verändern zu können, ohne die Anwendung verändern zu müssen. In .NET 4.0 hat diese Technologie ein gründliches Redesign erfahren oder sagen wir lieber, es wurde von Grund auf neu programmiert. Damit hat Microsoft allerdings viele Entwickler verprellt, weil der bislang erstellte Code aufs Abstellgleis manövriert wurde. Mit .NET 4.5 wurden wir wieder ein wenig versöhnt und die Features der Workflow Foundation sind wieder nahezu komplett.

Mit WF 4.5 lernen Sie eine völlig neue Art der Programmierung kennen: elegant, effizient, visuell und erweiterbar. Wenn das keine guten Gründe sind, um WF zu erlernen?

Kapitel 9, »Windows 8 und Windows RT«

Die Kachel-Oberfläche ist völlig neu in Windows 8 und .NET 4.5 und Gegenstand von Kapitel 9. Sie werden sehen, es macht Spaß, damit zu arbeiten; erst recht, wenn Sie sich den Luxus eines eigenen Windows-Tablets gönnen und sich nach Herzenslust durch Ihre neu entwickelten Anwendungen scrollen – intuitiv, bunt und butterweich.

Doch vor den Erfolg hat Microsoft die Lernkurve gesetzt, oder genauer: zwei Lernkurven. Zum einen gibt es eine neue API, Windows Runtime (WinRT) genannt, eine API, die direkt auf dem Windows-Kernel aufbaut und die neue Namensräume, Methoden und Möglichkeiten mitbringt. Zum anderen sind Windows-Apps aber auch viel standardisierter als herkömmliche Anwendungen, sollen sie doch auch jenseits von Intel auf ARM-Geräten laufen und sich einfach und schnell über den Microsoft Store installieren lassen.

Der vielleicht beste Weg, das alles zu lernen, führt über ein Fallbeispiel, das daher ein wenig umfangreicher ausgefallen ist. Aber, keine Angst, die einzelnen Abschnitte bauen aufeinander auf, wie immer.

Kapitel 10, »Softwaretests«

Das Testen von Software ist oft eine ungeliebte Aufgabe, dabei ist gerade dieser Teil der Softwareentwicklung für das Endresultat besonders wichtig und oft genauso erfolgskritisch. Im Kern geht es darum, Softwaretests als Teil der Entwicklung zu verstehen und zu integrieren und sie nicht alleine an deren Ende zu stellen. Dabei kann das Testen von Software auch eine sehr interessante Tätigkeit sein, mit den richtigen Werkzeugen und jeder Menge Know-how.

Zum Glück hat sich in .NET 4.5 und Visual Studio 2012 eine Menge getan, zum Beispiel hat sich das Produkt für andere Unit-Test-Werkzeuge geöffnet und die Integration in den (ebenfalls neuen) Team Foundation Server (TFS) wurde weiter verbessert. Softwaretests sind vor allem aber auch praktisch durchzuführen, also gerade recht für ein Praxisbuch der Softwareentwicklung. Und so erfahren Sie in diesem Kapitel nicht nur etwas über die Grundlagen, sondern auch, wie Sie am besten vorgehen und wie Sie Ihre Tests praktisch gestalten können.

Kapitel 11, »Softwarepflege«

Ein Kapitel über Softwarepflege in einem Buch über Softwareentwicklung? Passt das zusammen? Und ob, Programme sind schließlich keine Eintagsfliegen, und nach

dem Produktiveinsatz wartet der größere Teil der Arbeit, jedenfalls über die Zeit gesehen.

Spätestens jetzt ist es höchste Zeit, sich über Release-Zyklen, Versionsplanung und den Umgang mit laufenden Anforderungen Gedanken zu machen. Nicht minder wichtig und interessant sind die verschiedenen Möglichkeiten, Aufwände zu schätzen, gerade dann, wenn Sie agile Methoden einsetzen, womit Kapitel und Buch schließen.

Kapitel 3
Softwaredesign

Im Entwurf, da zeigt sich das Talent, in der Ausführung die Kunst.
(Marie Freifrau von Ebner-Eschenbach)

Softwaredesign beginnt dort, wo die Architektur endet, in der Theorie. Praktisch gesehen sind die beiden Themenkomplexe selten scharf zu trennen; viele Architekturentscheidungen haben Auswirkungen auf Fragen des Designs, beispielsweise indem sie dessen Auswahlmöglichkeiten einschränken. Wiederum praktisch gesehen macht das keinen großen Unterschied, denn Architektur und Design sind beide gleich wichtig.

Erfahrungsgemäß haben viele Entwickler Schwierigkeiten anzufangen, also die ersten Schritte zu finden und zu gehen. Anhand eines Fallbeispiels erarbeiten wir in diesem Kapitel daher zunächst einmal ein Klassenmodell, das oft als Ausgangspunkt für die Codierung dient. Im Anschluss wenden wir uns Designentscheidungen zu. Das sind wichtige Themen, die vor der Umsetzung eines Projekts zu klären sind.

Daneben stellen sich für nahezu jedes Projekt immer wieder dieselben Fragen: Wo und wie speichere ich meine Konfigurationsdaten, oder wie kopple ich meine Anwendung mit bestehenden Systemen? Diesen Fragen sind jeweils eigene Abschnitte gewidmet.

Bestimmt kennen auch Sie Anwendungen, die kaum zu bedienen sind, weil sie vom Anwender viel implizites Wissen verlangen (das er nicht besitzt), zum Beispiel weil jede Maske anders aussieht und anders bedient werden muss. Die Anwendung gängiger Windows-Styleguides genügt aber nicht, denn noch wichtiger als die Anordnung von Bedienelementen ist es, dem Anwender zu jeder Zeit ein logisches und in sich schlüssiges Bedienkonzept zu präsentieren. In Abschnitt 3.6, »Konfiguration«, gehe ich näher darauf ein.

Der nächste Abschnitt beschäftigt sich mit der Zeit, oder genauer mit der Wartezeit des Anwenders, einem oft vernachlässigten Thema – wobei dieses Thema gerade rasant an Fahrt gewinnt, weil es für die Erstellung von Windows-8-Anwendungen, die auf einem Tablet laufen können, nun einmal von zentraler Bedeutung ist.

Zur professionellen Entwicklung gehört auch untrennbar die Verwendung entsprechender Werkzeuge und Bibliotheken. Die Enterprise Library ist so eine (freie) Biblio-

thek, die Ihnen vieles leichter machen kann. Den Schluss dieses Kapitels bildet daher eine Einführung in die Enterprise Library mit zwei Tutorials.

Dieses Kapitel präsentiert sich Ihnen als ein Lesekapitel, aus dem Sie nach Lust und Laune einzelne Abschnitte herauspicken können. Ich hoffe, Sie sind mit meiner Auswahl der Themen zufrieden. Und nun viel Freude damit.

3.1 Grundlegende Designziele

Bevor wir uns der praktischen Seite zuwenden und mit Hilfe der objektorientierten Analyse und des objektorientierten Designs ein Klassenmodell entwerfen, möchte ich auf einige allgemeingültige Designziele eingehen, die Richtschnur für jede Designentscheidung sein sollten. Einige dieser Ziele wiederholen sich in Abschnitt 3.3.1, »Gutes Design, schlechtes Design«, dann mit konkreten Empfehlungen für die Modellierung.

3.1.1 Erfüllung der Funktion

Die Software soll das machen, was spezifiziert war – aus Sicht des Unternehmers wirklich nur das, aus Sicht des Anwenders auch gerne mehr, aus Sicht des Entwicklers manchmal lieber etwas anderes.

Dass Software die an sie gestellten Grundanforderungen nicht erfüllt, kommt gar nicht so selten vor, wie man zuerst denken möchte. Ich kann mich beispielsweise an eine Standardsoftware erinnern, bei der in einer Stammdatenmaske nicht gespeichert werden konnte, weil der Speichern-Button vergessen wurde. Und ich erinnere mich an eine Archivlösung, bei der Dokumente zwar als archiviert gekennzeichnet, aber eben nicht archiviert wurden; ein vergessener Parameter war die Ursache. Ich habe auch schon Adresseingabemasken gesehen, in denen das Feld »Postleitzahl« vergessen wurde. Andererseits: Wie schön, dass bei aller Automatisierung noch Menschen am Werk sind. Verlieren Sie also nicht die Grundfunktionen aus dem Blick, denn es gilt: Erst die Pflicht, dann die Kür.

Anwender haben ihr eigenes Qualitätsempfinden, für sie liegt das Problem immer hinter dem Bildschirm. Sie erwarten nicht nur, dass die Funktion erfüllt wird, sondern auch

- dass die Arbeitsschritte so angeordnet und bedienbar sind, wie es ihrer eigenen Denkweise entspricht. Das nennen sie dann benutzerfreundlich.
- geringe oder gar keine Wartezeiten (Wartezeiten sind für Anwender auch dann nicht akzeptabel, wenn sie lange genug für eine Kaffeepause wären).

- Sicherheit, beispielsweise den Schutz persönlicher Daten oder das Vorhandensein einer Benutzerverwaltung.
- Außerdem wollen sie, wenn schon nicht unterhalten, dann aber auch nicht gelangweilt werden, ein gefälliges Erscheinungsbild nehmen sie dankbar an.

3.1.2 Zuverlässigkeit und Robustheit

Software soll zuverlässig sein. Darunter versteht natürlich jeder etwas anderes. Für unsere Zwecke meinen wir damit, dass ein Anwender einer Software darauf vertrauen kann,

- dass sie jeden Tag auf dieselbe Art und Weise funktioniert,
- sie keine Daten verliert und
- das Ergebnis einer Funktion korrekt ist.

Anwender unterscheiden nicht immer (also selten) zwischen Abstürzen aufgrund mangelnder Zuverlässigkeit und aufgrund fehlerhafter Eingaben. Sie erwarten Zuverlässigkeit auch dann, wenn ihre Eingaben falsch oder unvollständig waren. Wir nennen das *Fehlertoleranz*, also die Unempfindlichkeit des Programms gegenüber fehlerhafter Bedienung.

Zuverlässigkeit beinhaltet auch *Robustheit*. Robust ist eine Software, wenn sie einerseits fehlertolerant ist, andererseits aber auch mit großen Datenmengen dauerhaft zuverlässig umgehen kann. Das ist etwas ganz anderes und in der Praxis verläuft das Verhältnis meist umgekehrt proportional: Je größer die Datenmengen, desto mehr Sand verirrt sich ins Getriebe.

3.1.3 Wartbarkeit

Wartbarkeit ist die Grundlage für *Erweiterbarkeit* und selbst wiederum ein facettenreicher Begriff. Wartbar ist eine Software, wenn

- sie verständlich und vollständig dokumentiert ist,
- sie vom inneren und äußeren Aufbau her gut gegliedert ist,
- es auch Entwickler gibt, die die Software noch verstehen,
- sie nicht bereits über viele Softwaregenerationen gealtert ist,
- die verwendete Technologie selbst Wartbarkeit unterstützt,
- allgemein anerkannte Patterns verwendet werden, die von den beteiligten Entwicklern wiedererkannt werden,
- der Quellcode überhaupt noch kompilierbar vorhanden ist,
- was auch für den Quellcode (oder Binaries) von verwendeten Komponenten gilt,

- Code, Dateien und Projektumfeld ordentlich strukturiert und aufgeräumt sind,
- Redundanz vermieden wurde und stattdessen Code und Komponenten wiederverwendbar gestaltet sind und wenn
- kein `goto` verwendet wird. (Ok, das war nicht ernst gemeint.)

Wartbarkeit ist aber kein binärer Zustand, sondern oft eine Frage von Zeit und Geld. Es ist aber auf jeden Fall sinnvoll, diesen Aspekt bereits beim Design einer Software zu berücksichtigen.

In den letzten Jahren ist ein Trend zu beobachten, der Sorge bereitet: Die allzu sorglose und vorschnelle Verwendung von freien Bibliotheken und sogar Sprachen in Projekten. Als Unternehmer kann Ihnen das einen Wettbewerbsvorteil bieten, verhindert es doch, dass versierte Kunden und deren Dienstleister Hand anlegen. Vor allem, wenn exotische Script-Engines verwendet werden, ist auch für das gewöhnliche Customizing schnell Expertenwissen vonnöten.

Viele dieser Projekte werden nicht mehr weiterentwickelt und lassen sich dann in Folgeversionen von Visual Studio und C# nicht mehr fehlerfrei einsetzen. Und fehlerarm sind sie ohnehin nur selten.

Beispiel

F# ist eine tolle Sprache, weil sie allzwecktauglich ist und dennoch durch ihren funktionalen Ansatz für eine Vielzahl an Problemen einfache und elegante Lösungen erlaubt. Kurz: Ich mag sie.

Ein Blick auf den aktuellen TIOBE-Index, ein Index für die Beliebtheit von Programmiersprachen, verrät, dass F# aktuell auf Platz 66 verweilt, zwischen Emacs Lisp und Factor. Und das, obwohl F# schon in der Version 3 verfügbar ist und eine breite Unterstützung seitens Microsoft erfährt.

Das Rating, ein Grad für die Verfügbarkeit an qualifizierten Fachkräften, Kursen, Toolanbietern etc., weist für den 50. Platz (als letzten Platz in der Liste) einen Wert von 0,143 % aus, gegenüber 6,53 % von C#, das schon einige Zeit auf Platz 5 liegt.

Ob man den TIOBE-Index nun als tauglich ansieht oder nicht: Fakt bleibt, dass, relativ zu C#, C++ oder Java gesehen, nur wenige Entwickler F# beherrschen. So leid es mir für diese Sprache tut, in meinen Programmen, für die ich wenigstens eine Laufzeit von 10 Jahren annehme, kann ich sie guten Gewissens noch nicht einsetzen.

3.1.4 Erweiterbarkeit

Auch unter diesem Begriff, manchmal auch mit dem Begriff *Evolvierbarkeit* synonym, verbergen sich mehrere Bedeutungen. Einerseits geht es um *Änderbarkeit*, beispielsweise wenn zur Identifizierung von Büchern statt der alten ISBN 10 die neuere

ISBN 13 verwendet werden soll. Solche Änderungen können tiefgreifend sein, wenn z. B. das Datenmodell betroffen ist und die Änderung durch alle Schichten einer Anwendung umgesetzt werden muss. Andererseits geht es um *Ergänzbarkeit*, also das Umsetzen neuer Anforderungen, beispielsweise durch Ergänzung neuer Module, durch Hinzufügen neuer GUI-Elemente oder neuer Stammdaten.

Wenn Sie selbst Software einkaufen, dann ärgern Sie sich vielleicht über die teilweise unverschämt hohen Preise für kundenspezifische Anpassungen, meist *Customizing* genannt. Einige Unternehmen leben von diesen Einnahmen, und das gar nicht schlecht. Andere Unternehmen haben aber gar keine Wahl, sie müssen viel verlangen, weil ihre Software sich gegen jede Form von Erweiterung sträubt.

Aus der Praxis

Vor einigen Jahren habe ich eine Faxsoftware gekauft und eingeführt. Diese Faxsoftware ermöglicht den Empfang von Faxen über den Exchange Server und das Versenden von Faxen, unter anderem auch über eine Dateischnittstelle.

Um Mahnungen zu faxen, ist es aber nicht nur wichtig, den Versand zu starten, man muss auch überprüfen können, ob das Fax ankam. Wenn der Versand fehlschlägt, soll die dahinter liegende Software die Mahnung ausdrucken, damit sie anschließend mit der Briefpost versendet werden kann.

Die neue Faxsoftware unterstützte diese Anforderung entgegen der ursprünglichen Zusage nicht, und so musste die Funktion nachträglich eingebaut werden. Der Hersteller der Software verwendete dafür eine Skriptsprache, die nur wenige Menschen dieser Erde beherrschen, sodass ein Spezialist nötig war. Die Kosten für die Anpassung beliefen sich schließlich auf 2.500 EUR. Die Anzahl der Codezeilen: weniger als 30.

Ich kenne allerdings auch die andere Seite, nämlich die des Softwareanbieters, der sich über Einnahmen durch Customizing freut. Je nachdem, auf welcher Seite Sie stehen: Halten Sie Ihre Software erweiterbar. Und wenn als Konsequenz die Entwicklung neuer Anforderungen nicht viel Zeit kostet, dann verkaufen Sie doch einige Tage Beratung, Planung und Projektmanagement.

Soviel zu den Grundlagen, nun aber zum eigentlichen Design.

3.2 Objektorientierte Analyse und Design

Das Konzept der Software ist fertig, das Entwicklerteam steht, und die Vorstellungen über den Fertigstellungstermin sind konkreter, als es einem lieb ist. Kurzum: Es beginnt die heiße Phase der Umsetzung. In Kapitel 2, »Softwarearchitektur«, haben

wir über die Komponenten gesprochen und deren Beziehungen zueinander sowie über Architekturstile. Nun geht es darum, die Abstraktionsleiter weiter hinabzusteigen und konkreter zu werden.

Erfahrungsgemäß fällt dies nicht immer leicht. Gerade die Frage: »Womit fange ich an?« wird häufig gestellt. Das Erkennen von Objekten und ihren Beziehungen zueinander ist ein guter Start, denn aus dem Ergebnis leiten sich Code und Datenbank sowie Klassen und Entitäten gleichermaßen ab. Die folgenden Abschnitte beschäftigen sich daher damit, die Architektur »auf den Boden zu bringen«, also von den Diagrammen und Vorüberlegungen zu realem Code zu gelangen.

3.2.1 Gründe und Aufgaben der OOA/OOD

Die eigentliche Aufgabe der Phase von OOA und OOD ist es, die oft enorme Komplexität von Softwareprojekten beherrschbar zu machen. *Divide et impera*, also »Teile und herrsche«, ist ihr Motto.

Das wird an einem Beispiel deutlich: Hatte Windows NT noch 6 Millionen Codezeilen, an denen rund 200 Entwickler arbeiteten, so bringt es Windows XP schon auf mehr als 10-mal so viele Codezeilen. Und einige Unix-Derivate beinhalten schon annähernd 300 Millionen Codezeilen. Hinzu kommen eine höhere Komplexität durch mehr Möglichkeiten und neue Technologien, gesteigerte Anforderungen und weit stärker vernetzte Systemlandschaften.

Definition

In der *objektorientierten Analyse (OOA)* geht es darum, ein System von Objekten zu bilden und diese Objekte so miteinander zu verbinden, dass die Aufgabenstellung des Fachkonzepts dadurch abgebildet wird. Wird das so gebildete Modell umgesetzt, so sprechen wir von *objektorientiertem Design (OOD)*. Die Umsetzung erfolgt dann mithilfe der *objektorientierten Programmierung (OOP)*.

Für unsere Zwecke werden wir OOA/OOD zusammenfassen, denn es geht uns ja gerade darum, die Objekte so zu definieren, dass wir sie mittels .NET und C# umsetzen können. Dennoch ist es nicht dasselbe, denn im Design schränkt die verwendete Technologie die Möglichkeiten ein. Beispielsweise unterstützt C# keine Mehrfachvererbung, C++ hingegen schon.

Zentrales Werkzeug ist die *Dekomposition*, die Zerlegung einer Aufgabenstellung in kleinere, beherrschbare Einheiten, eben »Teile und herrsche«. In der Literatur gibt es noch viele ähnliche, aber im Detail doch abweichende Definitionen. Manchmal wird der Objektbegriff viel weiter gefasst und umfasst dann z. B. auch die Benutzeroberfläche. In diesem Buch fassen wir den Objektbegriff enger und meinen damit .NET-

Klassen. Die abstrakteren Ebenen wurden ja teilweise schon in Kapitel 2, »Softwarearchitektur«, behandelt.

Noch ein Hinweis zum Schluss: Es gibt Zeiten, in denen OOA/OOD nicht das Mittel der Wahl ist. Nicht jede Aufgabenstellung muss, soll oder kann gar mittels Objekten gelöst werden. Immer wieder sieht man Programme, in denen der inflationäre Einsatz von Objekten der Performance und dem Speicherverbrauch abträglich sind. Ein Beispiel hierfür sind Massendatenoperationen. Hier können manchmal ein `SqlDataReader` und eine einfache prozedurale Vorgehensweise das Problem schneller und effizienter lösen. Dennoch: Für die meisten Anwendungen ist OOA/OOD ein guter Weg.

Was OOA/OOD zu leisten vermag, bemerkt man so richtig erst dann, wenn diese Organisationsprinzipien einmal fehlen. JavaScript ist das vielleicht wichtigste Beispiel. Viele Entwickler klagen, dass sich damit kaum größere Anwendungen entwickeln lassen. Und daher gibt es dort Precompiler, die diese Features »nachrüsten«, mit *TypeScript* von Microsoft sogar einen recht populären. Das brauchen wir für C# zum Glück nicht, weil es nahezu alles mitbringt, was man sich nur wünschen kann.

3.2.2 Das Fallbeispiel

Beginnen wir mit der Anforderung an eine neu zu erstellende Software, oft ein guter Ausgangspunkt für OOA/OOD, egal, ob Sie aus dem daraus entstehenden Lastenheft direkt in die Entwicklung einsteigen oder vorab ein Pflichtenheft erstellen wollen: Eine Klassenhierarchie ist in beiden Fällen notwendig.

Anforderungen Projekt »CRM«

Die Vertriebsmannschaft der Kalimba Sunfood setzt noch auf Outlook, um mit den Kunden zu kommunizieren. Dadurch haben die Kollegen keinen Zugriff auf die Kontakthistorie und andere wichtige Kontaktdaten des Vertriebs. Das soll sich nun ändern, und zwar mithilfe einer *CRM-Software* (CRM: *Customer Relationship Management*). Alle Anrufe, E-Mails und Briefe sollen darin erfasst und gespeichert werden, auch solche, die im Rahmen einer Vertriebsaktion einem Kunden zugesendet werden.

Ein Wiedervorlagesystem erlaubt es, Termine festzulegen, an denen ein Kunde wieder kontaktiert werden soll. Darüber hinaus gibt es noch Ereignisse, die bei einem Kunden gespeichert werden können, zum Beispiel Geburtstage oder Jubiläen, jeweils mit eigenen Feldern. Diese Ereignisse lösen Benachrichtigungen aus, und zwar automatisch, ohne dass sich der Vertriebsbeauftragte darum kümmern müsste.

3.2.3 Abstraktion und Hierarchie

Um der Komplexität Herr zu werden, gibt es in der objektorientierten Analyse zwei wesentliche Konzepte, die Abstraktion und die Hierarchie. Wir werden sie beide immer wieder anwenden, zum Beispiel, wenn wir die Objekte suchen, die einer Anwendung zugrunde liegen.

Abstraktion

In der Naturwissenschaft ist die Abstraktion oft die einzige Möglichkeit, der Fülle von Informationen und Daten überhaupt einen Sinn zu entlocken.

Definition

Unter Abstraktion (lat. entfernen, trennen) verstehen wir das Weglassen von Details, und zwar dergestalt, dass die wesentlichen Eigenschaften eines Systems immer noch beibehalten werden.

Es geht also darum, den Wesenskern zu entdecken und damit sowohl Anzahl als auch Komplexität von Objekten zu reduzieren. Wir Menschen tun dies automatisch, es ist sozusagen in uns einprogrammiert. Wenn wir eine Metallkonstruktion auf vier Rädern erkennen, die sich mithilfe eines Motors auf einer Straße fortbewegt, so reihen wir dieses Objekt in die bekannte Klasse `Auto` ein. Damit liegen wir nicht immer richtig, aber doch so häufig, dass wir in der Praxis einen Nutzen daraus ziehen können. Wir wenden die Attribute und Methoden der Klasse `Auto` dann auf das beobachtete Objekt an: Wir sollten besser aus dem Weg gehen, denn das Objekt ist schwer, und es ist in der Lage, seine Richtung zu ändern, denn es besitzt eine Lenkung und so weiter.

Das Wetter ist ein höchst komplexes Phänomen, viel zu komplex, um es auch nur annähernd mithilfe heutiger und vermutlich auch kommender Technologie exakt berechnen zu können. Die Wetterforscher überziehen daher den Planeten mit einem dreidimensionalen Gitter. Für jede dieser Gitterzellen ermitteln sie zu einem Startzeitpunkt einige Parameter wie Luftdruck, Windstärke oder Temperatur. Die Abstraktion besteht nun darin, dass sie eine Gitterzelle als eine Einheit betrachten. Überall in dieser Zelle herrschen also, im Denkmodell der Meteorologen, dieselben Bedingungen. Mithilfe dieser Vereinfachungen wird das Wetter berechenbar, jedenfalls für einige Zeit im Voraus. Dieses Gitter und die Verfahren zur Berechnung nennen sie dann ein *Modell*, eine Nachbildung der Wirklichkeit – dieses Modell ist aber nicht die Wirklichkeit selbst, aber so gut, dass sich damit arbeiten lässt.

In der Softwareentwicklung ergeht es uns nicht viel anders. Aus der abstrahierten Welt der Anforderungen entsteht ebenfalls ein Modell, oft dargestellt mittels UML.

Dennoch besteht ein Unterschied: Unser Modell ist die Grundlage für die detailgetreue Umsetzung in der Programmierung, wir können Details nicht einfach »wegabstrahieren«. Unser Modell ist die Basis für die Umsetzung, nicht die Umsetzung selbst. Die eigentliche Abstraktion findet in der Softwareentwicklung zudem auf zwei Ebenen statt: Neben der Ebene der OOA ist bereits die Spezifikation, also z. B. das Erstellen des Lastenhefts, ein Prozess, in dem Abstraktion eine wichtige Rolle spielt. Wir wollen eben nicht alle Details eines Kunden speichern und verarbeiten, sondern nur diejenigen, die für unser Geschäft von Interesse sind. Während jeder Kunde biologisch unterschiedlich ist, fassen wir alle Kunden zusammen und bilden allenfalls noch Kategorien, um sie zu unterscheiden.

Auch auf der Ebene der Programmierung bedienen wir uns der Abstraktion, wenn auch etwas anders. Wenn wir beispielsweise LINQ verwenden, so beschreiben wir das zu erwartende Ergebnis, ebenso in SQL. Wir könnten stattdessen auch durch eine Liste iterieren und die benötigten Einträge manuell selektieren. Wir abstrahieren also die Details der Umsetzung, das Programm wird kürzer und eleganter. Dennoch gilt auch hier: Die Umsetzung selbst ist vollständig, nur das Programmiermodell erlaubt uns einen höheren Zugang zum Problem.

Ein höherer Zugang bedeutet allerdings auch in den allermeisten Fällen den Verlust von Kontrolle. Während wir ein SQL-Statement von Hand zu wahrer Größe verhelfen können, übernimmt in LINQ to Entities dies eine Engine.

Für die OOA ist die Abstraktion die wichtigste Grundlage, denn wir müssen Objekte suchen und finden, diese Objekte mithilfe von Attributen beschreiben, sie mittels Methoden zugänglich machen und die Beziehungen untereinander definieren. Wir destillieren diese Informationen aus den Spezifikationen, die aber auch oft redundant sind und viele Informationen enthalten, die für die OOA nicht von unmittelbarem Interesse sind.

Hierarchie

Ein zweites Hilfsmittel ist die Hierarchie. Sie begegnet uns in vielen verschiedenen Formen. Vererbung ist eine davon. Wenn wir beispielsweise erkennen, dass eine E-Mail eine spezialisierte Form einer Nachricht ist wie auch der Brief oder das Fax, so haben wir einerseits abstrahiert (Nachricht), andererseits Nachricht, E-Mail, Brief und Fax in eine besondere Hierarchie, in eine Vererbungshierarchie, eingereiht.

Auch Eigenschaften können eine Hierarchie aufweisen. In unserem Beispiel könnten das die Branchen sein. Man spricht dann oft von einem Baum, in unserem Beispiel vom Branchenbaum. Es gibt bereits viele vordefinierte Hierarchien, für die Branchen beispielsweise den NACE-Code, die hierarchische Klassifikation der Branchen, die weltweit Anwendung findet. Und natürlich beinhaltet das .NET Framework selbst schon unzählige Klassen in ebenso unzähligen Vererbungshierarchien.

Für Anfänger oft schwierig ist die Unterscheidung beider Kategorien, denn niemand hindert einen daran, eine Klasse `Bau` zu definieren und eine Ableitung davon, beispielsweise `Architekturbüro`. Näheres hierzu erfahren Sie im folgenden Abschnitt.

3.2.4 Objekte und Klassen

Vielleicht habe ich Sie bisher schon ein wenig mit der Verwendung dieser Begriffe verwirrt, das möchte ich nun auflösen. Zur Wiederholung: Von Objekten sprechen wir in zweierlei Hinsicht. Einerseits sind Objekte einfach »Dinge« aus der realen Welt oder aus der Welt der Spezifikation, zum Beispiel *Kunde* oder *Nachricht*. Andererseits gibt es Objekte aber auch in der Programmierung; dort handelt es sich um Instanzen von Klassen, sie benötigen also Arbeitsspeicher und existieren zu einer bestimmten Zeit im Laufzeitsystem von .NET.

Klassen wiederum sind Blaupausen, also Baupläne für Objekte. Nach ihnen können beliebig viele Objekte desselben Typs erzeugt werden. Aber Klassen und Objekte in .NET kennen Sie natürlich bereits, konzentrieren wir uns also darauf, wie Sie sie identifizieren können.

Objekte suchen und finden

Um Objekte zu identifizieren, ist es erst einmal hilfreich, die wesentlichen Hauptwörter aus dem Text der Spezifikation herauszusuchen und in der Einzahl niederzuschreiben. In unserem Fall sind dies:

- Vertriebsmannschaft
- Outlook
- Kollege
- Kontakthistorie
- CRM
- E-Mail
- Wiedervorlagesystem
- Ereignis
- Jubiläum
- Benachrichtigung
- Vertrieb
- Vertriebsaktion
- Kalimba Sunfood
- Kunde
- Zugriff
- Kontaktdaten
- Anruf
- Brief
- Termin
- Geburtstag
- Vertriebsbeauftragter
- Mitarbeiter
- Feld

Wir können nun schon einige Objekte identifizieren, die wir ganz offensichtlich nicht für das Programm benötigen. *Outlook* beispielsweise, denn es dient im Text ja nur

dazu, das bisherige Verfahren zu beschreiben. Offensichtlich keine Objekte sind ebenfalls:

- *Kalimba Sunfood* (beschreibt den Auftraggeber)
- *CRM* (beschreibt das ganze System)

Für die Identifizierung der restlichen Nicht-Objekte müssen wir uns anschauen, welche Anforderungen an ein Objekt zu stellen sind. Die Kunst besteht darin, die Objekte weder zu grob noch zu fein zu fassen und sie so zu entwerfen, dass alle Anforderungen abgedeckt werden. Aber auch die zukünftigen Anforderungen, sofern absehbar, sollten bereits berücksichtigt werden. Konkreter gesagt: Objekte sollten

- einen Bezug zur Praxis haben, für die Aufgabenstellung also relevant sein. Was relevant ist, hängt vom Kontext ab. Für einen Ersatzteilkatalog wäre eine Schraube nicht als Objekt von Interesse, sie wäre dann im Objekt *Ersatzteil* enthalten und eher ein Fall für die Datenbank als für das Objektmodell. Für ein CAD-Programm ist hingegen eine Schraube sehr wohl ein Objekt, vor allem dann, wenn die Schraube für das CAD-Modell wichtig ist, weil beispielsweise Spannungen im Werkstück berechnet werden sollen.
- durch Attribute näher beschrieben werden, also einen Zustand besitzen. Ob und, wenn ja, welche Attribute es gibt, sollte im Pflichtenheft stehen. Wie immer gilt auch hier das Prinzip des Minimalismus: Sie sollten nichts modellieren, was Sie nicht sicher benötigen. Das dahinterstehende Prinzip, YAGNI (You ain't gonna need it) wird Ihnen in diesem Buch immer wieder begegnen.
- eine Schnittstelle nach außen anbieten, also Methoden und eventuell weitere Member besitzen.
- eindeutig unterscheidbar sein. Es gibt auch Fälle, in denen es später nur eine Instanz des Objekts geben wird. Dann ist diese Unterscheidung natürlich unwichtig. Factories, Engines und allgemein statische Objekte fallen oft in diese Kategorie.
- möglichst eine Implementierung besitzen. Manchmal kommen im Laufe der Analyse noch Objekte hinzu, die Oberklassen sind und in C# dann manchmal als abstrakte Klassen umgesetzt werden; diese Objekte finden Sie aber meist nicht im Pflichtenheft. Wir kommen später darauf zurück.
- redundanzfrei sein, denn häufig finden wir in Spezifikationen verschiedene Begriffe für ein und dieselbe Sache.
- weder zu grob noch zu fein sein. Zu grob wäre ein Objekt dann, wenn es mehrere Anforderungen im Pflichtenheft gäbe, die mithilfe dieses einen Objekts abgedeckt würden. Zu fein hingegen wäre ein Objekt, wenn man nichts Konkretes damit anfangen könnte und der Bezug zu den Anforderungen unklar wäre. Solche Objekte können später immer noch dem Modell hinzugefügt werden, wenn dies nötig werden sollte. Lassen Sie sie also im Zweifel weg.

Betrachten wir nun abermals unsere Objektliste, und untersuchen wir unsere Kandidaten anhand dieser Forderungen. Wir können dann folgende Kandidaten streichen:

- *Vertriebsmannschaft* – in der Anforderung findet sich kein Hinweis, dass die gesamte Mannschaft für eine Aktion benötigt wird.
- *Kollege*, *Mitarbeiter* – sind redundant, weil bereits der *Vertriebsbeauftragte* in der Liste steht.
- *Zugriff* – ist nicht konkret implementierbar, wir können das Objekt hinsichtlich seiner Attribute und Methoden nicht spezifizieren.
- *Vertrieb* – ist zu allgemein und zu abstrakt, um einen Platz im Modell zu finden.
- *Feld* – wäre zu fein und zu wenig bestimmt.
- *Marketingaktion* – es gibt bereits den Kandidaten *Vertriebsaktion*, die beiden Begriffe sind synonym.

Es bleiben also die folgenden Kandidaten übrig, die wir nun als Objekte identifiziert haben:

- Kunde
- Kontaktdaten
- E-Mail
- Wiedervorlagesystem
- Ereignis
- Jubiläum
- Benachrichtigung
- Vertriebsaktion
- Kontakthistorie
- Anruf
- Brief
- Termin
- Geburtstag
- Vertriebsbeauftragter
- Mitarbeiter

Das bedeutet nun aber nicht, dass unser Modell schon fertig wäre. Wir müssen unsere Objekte später noch weiteren Tests unterziehen. So wissen wir zum Beispiel noch nicht, ob wir das Objekt *Vertriebsbeauftragter* benötigen.

Kommen wir nun von der *objektorientierten Analyse* zum *objektorientierten Design*, indem wir aus den identifizierten Objekten Klassen entwerfen. Ich habe bereits erwähnt, dass ich OOA/OOD als Werkzeug der Umsetzung betrachte und nicht als Werkzeug des Systementwurfs, auch nicht als zwei getrennte Phasen, jedenfalls nicht in der Praxis. Denn es geht uns ja darum, zu einer fertigen Anwendung zu gelangen und nicht lediglich darum, Diagramme zu zeichnen. Hinzu kommt, dass sich in der Praxis oft der Entwickler einer Anwendung selbst diese Gedanken macht. Das unterscheidet diese Phase von der Architekturfindung, denn dort trifft man in der Praxis häufiger auf Spezialisten, die sich hauptberuflich mit diesem Thema beschäftigen.

Bevor wir nun weitermachen können, muss ich ein wenig auf UML eingehen, genauer genommen auf die Klassendiagramme.

Klassendiagramme

Das Klassendiagramm ist die vielleicht wichtigste Diagrammart in UML, denn sie beschreibt Klassen, ihre Attribute, Operationen (Methoden) sowie ihre Beziehungen untereinander. Es ist daher zustandsbehaftet und lässt die Dynamik außer Acht, dafür gibt es andere Diagramme in UML, zum Beispiel das Sequenzdiagramm.

Seit Visual Studio 2010 können nun endlich einige UML-Diagramme erzeugt werden, unter anderem auch Klassendiagramme. Leider sind diese Möglichkeiten in ihrer Gänze nur in der teuren Ultimate Edition enthalten. Am Markt befinden sich aber viele kommerzielle (und einige freie) AddOns, die im Wesentlichen dasselbe leisten. Abbildung 3.1 zeigt eine in Visual Studio erzeugte Klasse.

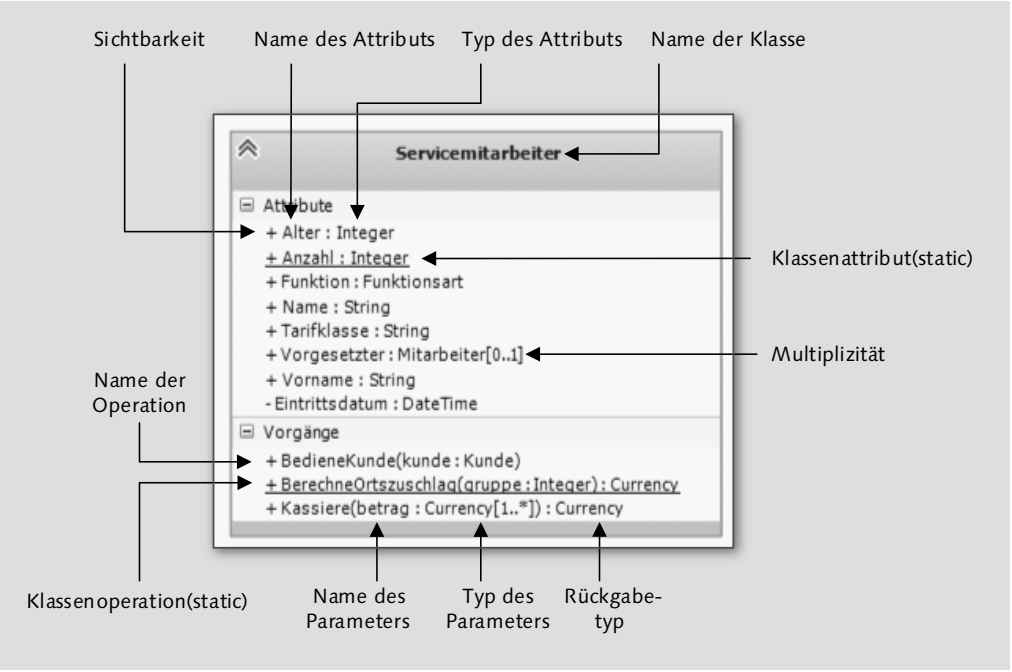


Abbildung 3.1 Klasse in Visual Studio

Im oberen Teil des Fensters in Abbildung 3.1 sind die Attribute abgebildet, im unteren Teil die Operationen, von Visual Studio etwas unglücklich, wie ich finde, »Vorgänge« genannt. Die folgende Tabelle zeigt die Elemente der UML-Klasse im Detail.

Bezeichnung	Beschreibung
Name des Attributs	Der Name des Attributs wird in den meisten Fällen den späteren Variablen bzw. Eigenschaftsnamen entsprechen.
Typ des Attributs	Als mögliche Typen kommen alle Typen von .NET infrage, sowohl Verweis- als auch Referenztypen. Wenn Sie die Diagramme unabhängig von der Implementierung entwerfen möchten, dann sollten Sie aber auf spezielle .NET-Typen verzichten.
Sichtbarkeit	Auch die Sichtbarkeitsebene kann und sollte angegeben werden. Zur Wahl stehen: ▸ + (`public`) ▸ # (`protected`) ▸ - (`private`) ▸ ~ (`paket`) Diesen Typ benötigen Sie dann, wenn Sie Diagramme in Paketen ablegen möchten. Pakete können Sie ebenfalls in Visual Studio 2012 erzeugen, wie im folgenden Beispiel zu sehen ist.  **Abbildung 3.2** Eine Klasse innerhalb eines Pakets
Klassenattribut	Standardmäßig sind alle Attribute Instanzattribute, jedes instanziierte Objekt besitzt also eine eigene Kopie des Attributs. Klassenattribute hingegen existieren für alle Objekte nur ein einziges Mal und werden in UML unterstrichen dargestellt.
Vorgabewert	Ein Attribut kann auch einen Vorgabewert nach der Instanziierung haben, der standardmäßig im Diagramm von Visual Studio aber nicht angezeigt wird.

Tabelle 3.1 Eine Klasse in UML

Bezeichnung	Beschreibung
Multiplizität	Die Multiplizität wird in eckige Klammern gesetzt. Gemeint ist damit, wie viele Ausprägungen es für das Attribut geben soll. Die Werte können als Menge angegeben werden (z. B. [1..3]) oder als eine genau festgelegte Anzahl (z. B. [1]). Einige Beispiele, die auch in Visual Studio standardmäßig zur Auswahl geboten werden: ▸ [0..1]: Entweder ist es nicht vorhanden oder genau einmal. ▸ [1]: Es ist genau einmal vorhanden; dies ist der Standardwert und muss daher nicht eigens angegeben werden. ▸ *: Das Attribut darf beliebig oft vorkommen, also auch gar nicht. ▸ [1..*]: Das Attribut darf beliebig oft vorkommen, jedoch mindestens einmal. Wenn Sie Multiplizitäten verwenden möchten, dann sollten Sie vorher überlegen, ob Sie nicht vielleicht eine eigene Klasse entwerfen wollen, gerade dann, wenn das Attribut einen Referenztyp haben soll.
Name der Operation	Der Name der Operation entspricht dem Methodennamen Ihrer Klasse, jedenfalls in den meisten Fällen.
Name des Parameters	Bitte beachten Sie die Regeln für die Vergabe von Parameternamen in C# oder der Sprache Ihrer Wahl.
Typ des Parameters	Auch hier gilt wieder: Sie können jeden Typ verwenden, sowohl Wert- als auch Referenztypen.
Klassenoperation	Per Standard sind alle Operationen Instanzoperationen, lassen sich also nur für ein instanziiertes Objekt ausführen. Sie können aber auch Klassenoperationen definieren, die in C# dann mit dem Schlüsselwort `static` definiert werden.
Eigenschaft	Sie können beliebige Eigenschaften selbst definieren, wenn Sie ein Attribut oder eine Methode näher bestimmen möchten. Ein Beispiel hierfür wäre `readonly`.
Rückgabetyp	Der Typ des Rückgabewertes, sofern vorhanden, wird bestimmt.

Tabelle 3.1 Eine Klasse in UML (Forts.)

Und so sieht die Klasse in Visio aus, das allerdings kaum Funktionen für UML-Diagramme enthält (siehe Abbildung 3.3). Ich will da gar nichts beschönigen: Mit Visual Studio 2012 können Sie zwar UML-Diagramme erstellen, genauer die folgenden Typen:

- Klassendiagramme
- Sequenzdiagramme
- Anwendungsfalldiagramme (Use Case)
- Aktivitätsdiagramme
- Komponentendiagramme

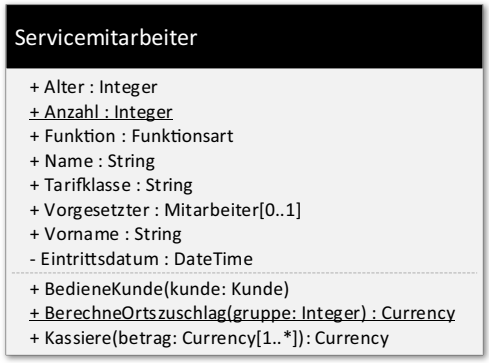


Abbildung 3.3 Die Klasse Servicemitarbeiter in Visio

Darüber hinaus gibt es noch weitere Diagrammtypen, beispielsweise die recht praktischen Ebenendiagramme. Alle hier dargestellten Diagramme sind auf diese Weise entstanden. Bin ich damit glücklich? Nein, nicht wirklich, dafür sind mir die Möglichkeiten insgesamt doch zu rudimentär.

Wenn Sie die englischsprachige Wikipedia konsultieren, dann finden Sie dort eine ganze Reihe von UML-Tools, auch bewertet nach ihren Funktionen, kommerzielle wie auch freie.

Wenn Sie Klassendiagramme einsetzen möchten, dann werden Sie sich immer wieder fragen, inwieweit Besonderheiten der Sprache darin umgesetzt sein können und sollen. Seit C# 4.0 gibt es beispielsweise den Datentyp `Tupel`, Sie könnten ihn als Rückgabewert also verwenden. Damit opfern Sie allerdings die Unabhängigkeit Ihres Diagramms gegenüber der späteren Implementierung. Ich persönlich halte die Diagramme immer dann unabhängig, wenn ich noch nicht weiß, ob und wie ein Projekt umgesetzt werden soll, also zum Beispiel dann, wenn auch PHP zum Einsatz kommen könnte. Je konkreter die Planung der Umsetzung bekannt ist, desto eher verwende ich die Spezialitäten der jeweiligen Sprache, dann aber auch konsequent.

In Visual Studio 2012 ist es nun auch möglich, den Code für die Klasse zu generieren, einfach durch Auswahl von Code generieren im Kontextmenü des Diagramms (bzw. der Klasse). Ein richtig komfortables Two-way-Tool, das Änderungen in beiden Systemen intelligent und automatisch generiert, ist das aber nicht.

Leider unterstützt Visual Studio keine automatische Codegenerierung aus dem Modell. Das ist wirklich schade. Wünschenswert wäre ein Two-way-Werkzeug, das aus einem Modell den Code erzeugen, aber auch aus dem Code das Modell rekonstruieren kann. Wie auch immer: In der letzten Auflage zu Visual Studio 2010 hatte ich noch moniert, dass überhaupt kein Code generiert werden kann.

Der generierte Code ist übrigens durchaus durchdacht:

```
...
private DateTime Eintrittsdatum
{
  get;
  set;
}
public virtual void BedieneKunde(Kunde kunde)
{
  throw new System.NotImplementedException();
}

public static Currency BerechneOrtszuschlag(int gruppe)
{
  throw new System.NotImplementedException();
}

public virtual Currency Kassiere(IEnumerable<Currency> betrag)
{
  throw new System.NotImplementedException();
}
```

Listing 3.1 Der vom Codegenerator aus dem Klassendiagramm generierte Code

Visual Studio hat hier die Multiplizität des Parameters `betrag` [1..*] mit Hilfe einer generischen List umgesetzt. Praktisch, wenn auch nicht perfekt, denn natürlich könnte die Liste auch leer sein, entgegen der Vorgabe, die mindestens ein Element darin verlangt. Das ist jedoch nicht weiter schlimm, denn Sie können die Templates auch selbst anpassen und zwar mit Hilfe der T4-Sprache. Unter Einstellungen • Standardeinstellung für die Codegenerierung konfigurieren erhalten Sie für jedes hinterlegte UML-Konstrukt den Verweis auf ein T4-Template (siehe Abbildung 3.4).

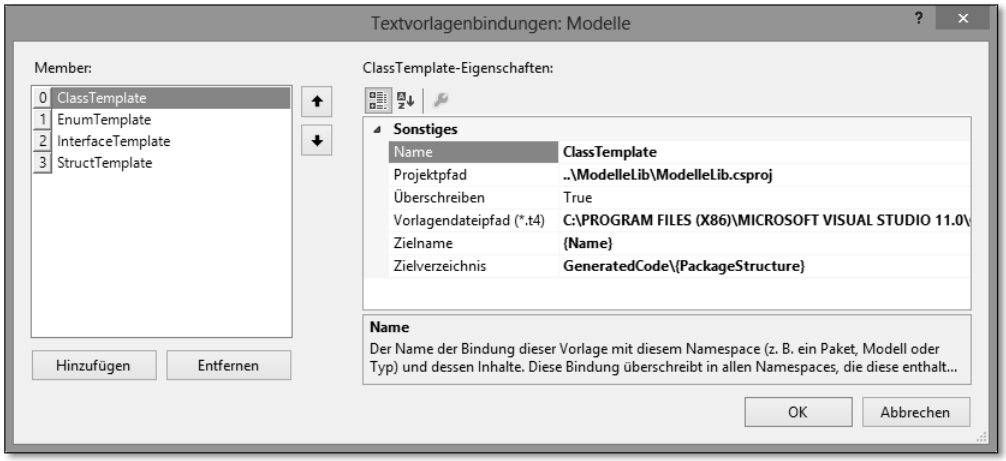


Abbildung 3.4 Die Codegenerierung aus dem Diagramm konfigurieren

Zum Schluss möchte ich noch kurz auf Schnittstellen und Enumerations eingehen, die beide im Visual-Studio-Designer entworfen werden können.

Sie können den Typ der Enumeration natürlich auch als Typ eines Attributs verwenden. Sprachbedingt gibt es für Schnittstellen in C# einige Einschränkungen, die jedoch in Visual Studio nicht logisch umgesetzt sind, zum Beispiel bei der Sichtbarkeit der Operationen, die im obigen Beispiel mit `public` angegeben wurde, in der Implementierung aber eben gerade nicht `public` sein darf. UML ist eben nicht C#. Aber natürlich wird der richtige Code daraus generiert.

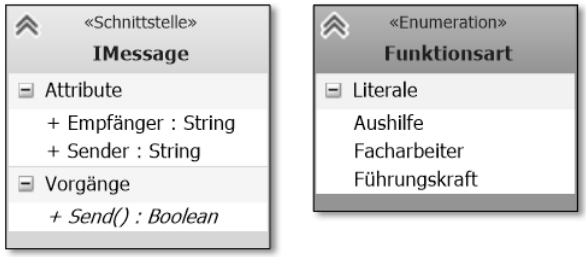


Abbildung 3.5 Schnittstelle und Enumerations

Attribute

Nun können wir die Klassen in UML modellieren und die Attribute identifizieren und hinzufügen. Sie brauchen zum jetzigen Zeitpunkt noch nicht alle Attribute angeben. Während der späteren Umsetzung werden bestimmt noch zahlreiche Attribute ergänzt oder der Typ bestehender Attribute geändert. Das ist normal und Ausdruck der Dynamik in der Entwicklung und in aller Regel kein Mangel in der Planung. Wichtig ist es vor allem, dass Sie hier die expliziten Attribute angeben, also Attribute, die

ein Objekt näher beschreiben. Daneben gibt es noch die impliziten Attribute, die sich aus der Umsetzung ergeben und nur dafür gebraucht werden.

Es liegt nahe, dass Sie den Typ `object` oder den impliziten Typ `var` vermeiden sollten. Die Klasse `EMail` aus unserem Beispiel könnte wie in Abbildung 3.6 dargestellt aussehen, nachdem wir ihr die Attribute hinzugefügt haben.

Das Hinzufügen von Attributen führt oft zum Hinzufügen weiterer Klassen. In unserem Beispiel haben wir die Klasse `EMail` definiert. In der Auflistung der bisherigen Klassen gibt es jedoch bislang keine Klasse, die diese E-Mail versenden könnte. Eine Möglichkeit wäre es nun, der Klasse `EMail` eine statische Methode hinzuzufügen, `sendEmail (emailToSend EMail)`. Oder aber wir erzeugen eine neue Klasse, die dann auch die zu versendenden E-Mails abrufen und verarbeiten könnte.

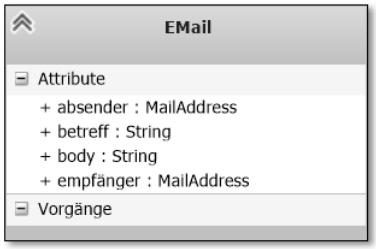


Abbildung 3.6 Die Klasse »EMail«

Sie sollten auch darüber nachdenken, ob sich bereits jetzt Vererbung anbietet. Wenn Sie beispielsweise die E-Mails an einen Kunden beim Kunden speichern möchten, dann gibt es dafür zwei Möglichkeiten, wie Abbildung 3.7 zeigt.

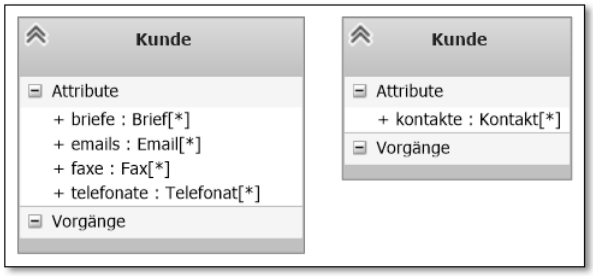


Abbildung 3.7 Klasse »Kunde«

Das rechte Beispiel macht sich Vererbung zunutze, die Typen der linken Klasse können, müssen aber nicht, von einem gemeinsamen Vorfahren erben. Die Attribute ergeben sich direkt aus der Spezifikation. Bei der rechten Definition benötigen wir eine weitere Klasse, `Kontakt`, die so, wie bereits gesagt, nicht in der Spezifikation zu finden ist.

Operationen

Als Nächstes fügen wir die Operationen hinzu, die später in C# zu Methoden werden. Wenn Sie bereits über Erfahrung im Design eines Objektmodells verfügen, dann werden Sie vermutlich nicht strikt nach diesen Phasen vorgehen. Vielmehr werden Sie die Attribute, die Operationen und die Beziehungen der Klassen parallel entwickeln. Wenn Sie unsicher sind, dann empfehle ich Ihnen aber, zunächst mit der hier vorgestellten Vorgehensweise zu beginnen. Sie werden dann von ganz alleine die für Sie richtige Reihenfolge finden.

Um die Operationen für Ihre Klassen zu bestimmen, können Sie sich an dieser Liste möglicher Operationsarten orientieren (jeweils mit Beispiel). Eine Operation kann

- den Zustand eines Objekts verändern, indem sie Attributwerte verändert,

  ```
  void SetCustomerInactive()
  ```

- den Zustand eines Objekts abrufen, ohne ihn zu verändern,

  ```
  bool IsCustomerActive()
  ```

- einen Iterator bzw. einen Indexer implementieren,

  ```
  public Customer this[int customerNo]
  ```

- ein Konstruktor oder

  ```
  public Customer(string name, ...)
  ```

- ein Destruktor sein (bzw. ein Finalizer in C#).

  ```
  ~Customer() { Dispose (false); }
  ```

Die schwierigste Aufgabe ist es, die Zuständigkeiten zu definieren, also festzulegen, welches Objekt welche Operationen durchführen soll. Auch in dieser Phase kann es zur Bildung neuer Klassen kommen, die sich nur indirekt aus der Spezifikation ergeben.

Auch hier stellt sich wiederum die Frage nach der Vererbung und damit die Frage nach Generalisierung und Spezialisierung. Das Vorhandensein von Operationen gleichen Namens deutet oft darauf hin, dass eine Basisklasse gebildet werden kann. Das Vorhandensein von Operationen ähnlichen Namens kann ein Indiz für Polymorphie sein.

Am Ende dieser Phase haben Sie also die ersten Klassen modelliert, mit ihren jeweiligen Attributen und Operationen, soweit dies jetzt schon möglich ist. Sie werden vermutlich weder völlig danebenliegen noch wird das Klassenmodell dem endgültigen Ergebnis bereits entsprechen. Bereits der nächste Schritt wird es sehr wahrscheinlich an der einen oder anderen Stelle ergänzen oder modifizieren. Lassen Sie mich das bitte noch einmal wiederholen: Das ist normal, für Anfänger genauso wie für hauptberufliche Klassendesigner.

3.2.5 Beziehungen

Widmen wir uns nun dem Kitt unserer Klassen zu, den Beziehungen. Erst mit ihren Beziehungen wird das Modell zu mehr als der bloßen Summe aller Klassen. Um die richtige Art der Beziehung zu finden, hilft die Sprache. Im Folgenden stelle ich Ihnen einige Sprachmuster vor, die wir in den Detailabschnitten dann jeweils anhand eines konkreten sprachlichen Beispiels und mithilfe von UML abbilden.

- Muster A: Ein Objekt ist eine Spezialform eines anderen Objekts, aber beide Objekte sind direkt verwendbar.
- Muster B: Zwei oder mehr Objekte ordnen sich begrifflich einem übergeordneten Objekt unter, dieses übergeordnete Objekt ist jedoch abstrakt. Erst die abgeleiteten Objekte können konkret implementiert werden.
- Muster C: Objekte stehen in Beziehung zueinander, sind jedoch auch offen für andere Beziehungen oder aber auch eigenständig verwendbar.
- Muster D: Ein Objekt nimmt Bezug auf sich selbst.
- Muster E: Ein Objekt ist Teil eines anderen Objekts, ohne dass es seine Selbständigkeit dadurch verliert.
- Muster F: Ein Objekt ist untrennbar Teil eines anderen Objekts; es kann ohne das Ganze nicht überleben und auch nicht Teil eines zweiten Objekts sein.

Generalisierung/Spezialisierung und abstrakte Klassen

Beispiel A: »Vertriebsbeauftragte sind spezielle Mitarbeiter, es gibt aber auch Mitarbeiter ohne spezielle Eigenschaften«.

Beispiel B: »Anruf, E-Mail, Brief und Fax sind Kontakte, der Begriff *Kontakt* ist aber abstrakt. Wenn wir jemanden kontaktieren möchten, dann immer über die oben genannten Wege«.

Es gibt also immer eine allgemeine Oberklasse und eine spezifische Unterklasse. Als alter Programmierhase erkennen Sie darin natürlich eines der Grundprinzipien der Objektorientierung, die Vererbung. In UML nennen wir dies *Generalisierung/Spezialisierung* und meinen dasselbe. Die Unterklasse erbt alle Eigenschaften und Methoden ihrer Oberklasse und kann durch weitere Eigenschaften und Methoden erweitert werden. Methoden in der Unterklasse können Methoden der Oberklasse gleichen Namens überschreiben; welche Operation ausgeführt wird, hängt vom konkreten Typ der Variable ab. In der OO-Welt spricht man dann von *Polymorphie* und C# kennt dafür die Schlüsselwörter `virtual` und `override`.

Der Unterschied zwischen Muster A und Muster B liegt darin, dass im ersten Fall die Oberklasse konkret ist, also selbst instanziiert und verwendet werden kann, während sie im zweiten Fall abstrakt ist. Davon lassen sich also keine Objekte instanziieren. Abstrakte Klassen sind aus drei Gründen wichtig:

- Sie begründen eine Vererbungshierarchie, stehen damit an oberster Stelle.
- Variablen können dann vom Typ der abstrakten Oberklasse sein. Es wird immer die Methode des konkreten Objekttyps verwendet.
- Manchmal ist es einfach nicht möglich, in der Oberklasse etwas zu implementieren, das in einer Unterklasse verwendet werden kann, sondern die Methoden müssen immer wieder neu überschrieben werden. Dennoch ist die abstrakte Oberklasse der gemeinsame Vorfahr aller Unterklassen. Man spricht daher auch häufig von einer *Ist-eine-* oder *Is-a*-Beziehung.

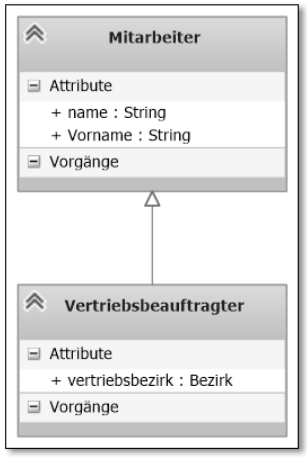


Abbildung 3.8 Beispiel A in UML

Nicht nur Klassen, sondern auch Schnittstellen und andere UML-Elemente können diese Art der Beziehung haben. In C# setzen wir das Beispiel aus Abbildung 3.8 wie folgt um:

```
class Mitarbeiter
{
  public string Name;
  public string Vorname;
}
class Vertriebsbeauftragter : Mitarbeiter
{
  public Bezirk Vertriebsbezirk;
}
```

Listing 3.2 Beispiel A in C#

Sie können, wenn Sie möchten, wieder die Codegenerierung anwerfen. Auch hier ist wieder nicht alles so, wie Sie das vermutlich gerne hätten, weil Visual Studio 2012 alle Eigenschaften beider Klassen als virtuell deklariert. Übrigens: Einmal generiert

markiert Visual Studio die Klassen mit dem UML-Stereotyp *C#-Klasse*. Stereotypen in UML erweitern vorhandene Modelle, vor allem – wie im Beispiel –, um die Verwendung näher zu spezifizieren. Zurück zum Beispiel:

In Beispiel B ist die Oberklasse abstrakt, die Unterklassen müssen also alle in der Oberklasse spezifizierten Methoden erst implementieren.

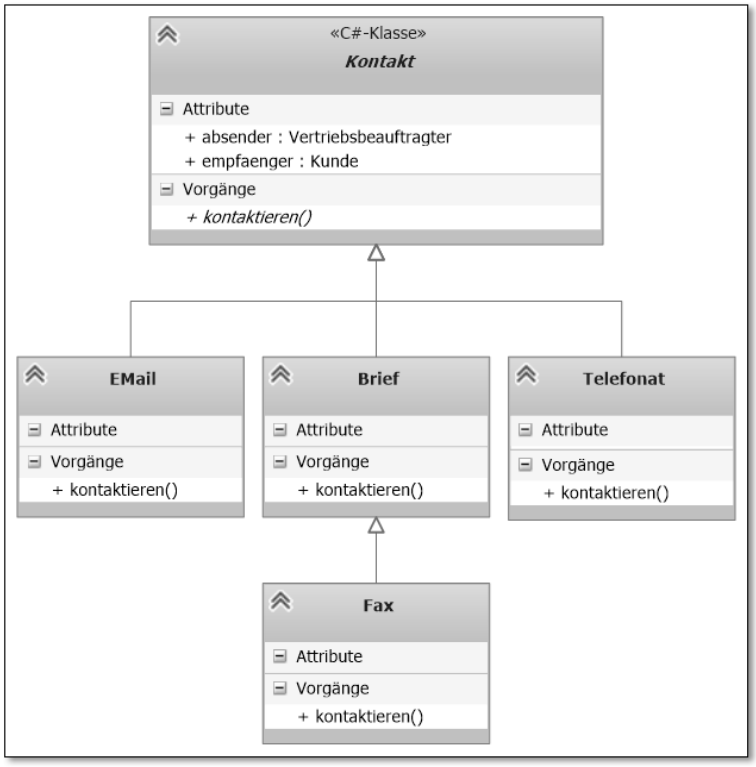


Abbildung 3.9 Beispiel B in UML

Wie Sie in Abbildung 3.9 sehen, sind abstrakte Klassen und abstrakte Methoden kursiv dargestellt. Die Umsetzung in C# könnte so aussehen:

```
abstract class Kontakt
{
  protected Kunde empfaenger;
  protected Vertriebsbeauftragter absender;
  public Kontakt(Kunde kunde, Vertriebsbeauftragter
                 vertriebsbeauftragter)
  {
    empfaenger = kunde;
    absender = vertriebsbeauftragter;
  }
```

```
  public abstract void Kontaktieren();
}

class Telefonat : Kontakt
{
  public string Telefonnummer;
  public Telefonat(Kunde kunde, Vertriebsbeauftragter
                   vertriebsbeauftragter)
                 : base(kunde, vertriebsbeauftragter)
  {
  }

  public override void Kontaktieren()
  {
    //Telefonverbindung aufbauen
  }
}
```

Listing 3.3 Muster B in C#

Assoziation

Beispiel C: »Ein Kunde nimmt an einer Vertriebsaktion teil«.

Wenn eine Klasse mit einer anderen Klasse assoziiert ist, so bedeutet dies, dass die beiden Klassen einerseits unabhängig sind, andererseits dennoch miteinander »zu tun haben«. Im Beispiel in Abbildung 3.10 sind Kunde und Vertriebsaktion dergestalt verbunden, dass ein Kunde Werbung erhält und eine Vertriebsaktion Werbung an einen Kunden aussendet. Dennoch können beide Klassen noch an vielen weiteren Verbindungen teilnehmen; die Verbindung Kunde zu Vertriebsaktion ist gewissermaßen zweckgebunden.

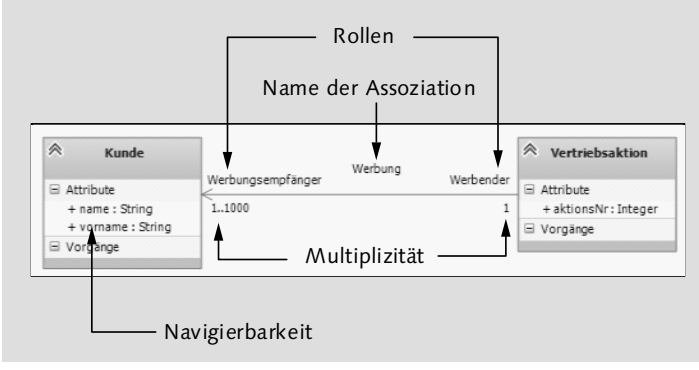


Abbildung 3.10 Muster C in UML, binäre Assoziation

Weil zwei Klassen daran beteiligt sind, heißt diese Assoziation *binär*. Es gibt einige Merkmale einer solchen Assoziation.

Bezeichnung	Beschreibung
Name der Assoziation	Der Name gibt umgangssprachlich an, welcher Tatbestand beide Klassen verbindet. In unserem Fall ist das die Werbung, die ein Kunde erhält und die aufgrund einer Vertriebsaktion versendet wird.
Rollen	In einer Assoziation können alle beteiligten Klassen unterschiedliche Rollen ausüben.
Multiplizität	In unserem Fall ist jeder Kunde (zu einer Zeit) nur an einer Vertriebsaktion beteiligt, während jede Vertriebsaktion mindestens einen und höchstens 1.000 Kunden bewirbt.
Navigierbarkeit	Wenn angegeben wird, ob und, wenn ja, wie navigiert werden kann, dann spricht man von einer *gerichteten Assoziation*. Im Beispiel gelangt man von der Vertriebsaktion zum Kunden, es ist also bekannt, welcher Kunde beworben wurde. Aus der Sicht des Kunden ist jedoch nicht klar, ob er weiß, aufgrund welcher Aktion er Werbung erhält. Es gibt drei Möglichkeiten: ▸ erlaubte Navigation (Kunde) ▸ nicht erlaubte Navigation (nicht im Beispiel enthalten, durch ein X gekennzeichnet) ▸ keine Aussage möglich (Vertriebsaktion)

Tabelle 3.2 Merkmale der Assoziation

Sehen wir uns nun die Klasse `Vertriebsaktion` im Code an.

```
class Vertriebsaktion
{
  public List<Kunde> Kunden;

  public Vertriebsaktion(List<Kunde> kunden)
  {
    Kunden = new List<Kunde>();
    if (kunden.Count == 0 || kunden.Count > 1000)
                throw new ArgumentException("Die Anzahl muss
                   zwischen 1 und 1000 liegen");
    Kunden.AddRange(kunden);
  }
}
```

Ein Sonderfall ist die *reflexive Assoziation*, also die Assoziation einer Klasse mit sich selbst.

Beispiel D: »Mitarbeiter können sowohl Vorgesetzte sein als auch Untergebene«.

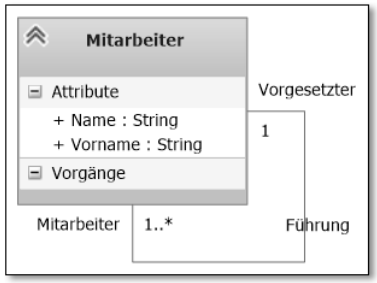


Abbildung 3.11 Muster D in UML, reflexive Assoziation

Die Umsetzung in C# birgt keine Überraschungen:

```
class Mitarbeiter
{
  public string Name;
  public string Vorname;
  public Mitarbeiter Vorgesetzter;
  public List<Mitarbeiter> Untergebene;

  public Mitarbeiter()
  {
    Untergebene = new List<Mitarbeiter>();
  }
}
```

Die praktische Umsetzung ist wieder eine Frage der Navigierbarkeit. Wenn lediglich vom Mitarbeiter zum Vorgesetzten navigiert werden können soll, dann braucht es die Liste der einer Führungskraft zugeordneten Mitarbeiter nicht. Unser Beispiel sieht jedoch eine ungerichtete Assoziation vor, die also in beide Richtungen funktioniert. Visual Studio berücksichtigt die Navigierbarkeit bei der Generierung von Code übrigens.

UML kennt darüber hinaus auch noch Assoziationen, an denen mehr als zwei Klassen direkt beteiligt sind, sowie Assoziationsklassen. Beide Konstruktionen sind in C# so nicht direkt umzusetzen und lassen sich zudem auf die hier besprochenen Assoziationen zurückführen, weswegen ich ihre Beschreibung auslasse.

Aggregation

Beispiel E: »Ein Kontakt ist Teil einer Vertriebsaktion, kann jedoch unabhängig davon auch beim Kunden gespeichert sein«.

Die Aggregation beschreibt eine etwas engere Assoziation. Im obigen Beispiel ist eine Vertriebsaktion denkbar, die (noch) keine Kontakte aufweist. Andererseits bleibt ein Kontakt auch dann bestehen, wenn er aus einer Vertriebsaktion entfernt wird. Das Objekt könnte dann zum Beispiel in eine andere Vertriebsaktion aufgenommen werden. In UML spricht man auch von einer *Teile-Ganzes-Beziehung* und meint damit immer genau zwei Klassen, die wie folgt miteinander in Verbindung stehen.

Worin liegt nun der eigentliche Unterschied zu einer bloßen Assoziation? Die UML gibt uns darauf keine wirklich befriedigende Antwort, und die Umsetzung in C# ist ähnlich bis gleich. Einige Modellierwerkzeuge unterscheiden bei ihrer Code- oder Modellgenerierung gar nicht mehr zwischen diesen beiden Typen.

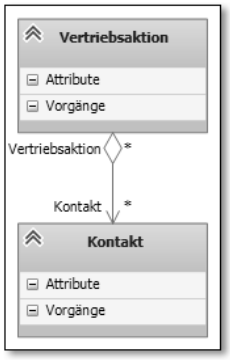


Abbildung 3.12 Beispiel E in UML, Aggregation

Die Aggregationsbeziehung ist etwas stärker als die Assoziation. »Ein Kunde nimmt an einer Vertriebsaktion teil« war unser Beispiel für eine Assoziation. Beide Klassen sind hier gleichwertig. Wenn eine Kunde zum Beispiel aus einer Vertriebsaktion entfernt wird, so ist das ohne Probleme möglich. Beide Objekte werden davon nicht besonders stark berührt.

Wenn wir hingegen einen Kontakt, zum Beispiel einen versendeten Brief, aus einer Vertriebsaktion entfernen möchten, so müssen wir im Code vielleicht entscheiden, was mit dem Objekt Kontakt geschehen soll, denn der Brief wurde ja schon an den Kunden versendet. Das ist ein Indiz für eine Aggregationsbeziehung, denn die Vertriebsaktion ist unabhängiger als die Klasse `Kontakt`.

Komposition

Beispiel F: »Ein Ereignis, zum Beispiel ein Jubiläum, ist untrennbar mit einem Kunden verbunden, ohne den Kunden ist es nicht mehr zu gebrauchen«.

Die *Komposition* ist also noch stärker als die Aggregation und ebenfalls eine Teile-Ganzes-Beziehung. Wird das Ganze gelöscht, so werden auch die Teile gelöscht. Ein Teil kann immer nur Bestandteil eines einzigen Ganzen sein, im Gegensatz zur Aggregation. Die Multiplizität auf Seiten des Kunden ist immer 1, denn ein Ereignis muss ja immer genau einem Kunden zugeordnet sein, obgleich der Designer im Visual Studio auch anderes zulässt. Im obigen Beispiel ist das Ereignis selbst zudem noch die Oberklasse für die Klasse `Jubiläum`.

Die Komposition hat zur Folge, dass Objekte automatisch gelöscht werden müssen, wenn das Ganze gelöscht wird. In C# ist das einfach zu lösen, beispielsweise durch eine Liste innerhalb der Klasse `Ereignis`. Der *Garbage Collector* sorgt dann schon dafür, dass auch Objekte des Typs `Ereignis` zerstört werden. Auf Seiten der Datenbank kann die Löschregel *OnDeleteCascade* angegeben werden, um auch die Datensätze der abhängigen Tabelle zu löschen.

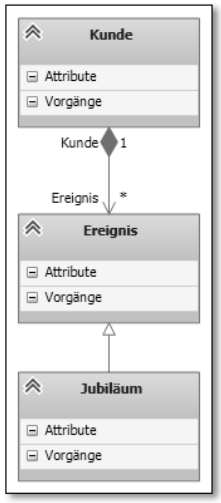


Abbildung 3.13 Beispiel F in UML, Komposition

Schnittstellen

Natürlich lassen sich auch Schnittstellen in UML abbilden, auch mithilfe von Visual Studio 2012. In unserem Fall wäre es z. B. möglich, die Kontaktarten nicht über eine abstrakte Klasse miteinander zu verbinden, sondern mithilfe einer Schnittstelle.

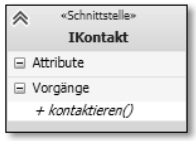


Abbildung 3.14 Schnittstelle in UML

3.2.6 War es das?

Was haben wir bisher erreicht? Wir haben aus dem echten Leben bzw. unserer Spezifikation ein Modell abstrahiert. Dazu haben wir die Objekte identifiziert, modelliert und zueinander in Beziehung gesetzt. Das Klassengeflecht spiegelt somit die Wirklichkeit wider. Unser Modell ist aber noch unvollständig, denn wir haben das Wiedervorlagesystem noch nicht modelliert. Doch das sollte Ihnen nun leichtfallen.

Für ein vollständiges Modell ist unser »Lastenheft« doch ein wenig zu dünn, aber im Grunde ändert sich auch bei umfangreichen Spezifikationen wenig bis gar nichts an dieser Vorgehensweise. Außer vielleicht:

- Sie werden vielleicht mehrere Diagramme anlegen, die jeweils einen anderen Teilaspekt der Software abdecken.
- Mit Paketen ist ein weiteres Ordnungsprinzip innerhalb von UML-Diagrammen machbar, das die Übersichtlichkeit weiter erhöht und das Modell ein weiteres Stück näher an die Wirklichkeit befördert. Pakete werden vom VS-Designer übrigens als Ordner innerhalb des Projekts angelegt, nebst passender Namespaces.
- Sie können in praktisch allen UML-Designern einzelne Klassen, Schnittstellen oder Packages auch einklappen. Außerdem stehen meist viel mehr Komfortfunktionen zur Verfügung, die Sie in größeren Projekten nutzen werden.

Das war die ganz praktische Seite von UML, die Einführung deckt aber nicht alle Funktionen ab. Allerdings bringt mich das zu einigen Tipps und Warnungen im Umgang mit der UML.

- UML ist Mittel zum Zweck, wie alles andere auch. Welch höchst liebevoll gestaltete Grafiken habe ich schon gesehen, deren Erstellung einen Gutteil der gesamten Entwicklungszeit gekostet haben muss.
- Hüten Sie sich auch vor dem Gegenteil: Diagramme auf Kindergarten-Niveau. Wenn ein einziger Satz genügt, um einen Use Case zu beschreiben, dann sollten Sie das tun und dafür kein Use-Case-Diagramm anfertigen.
- Sie sollten in die UML nichts hineininterpretieren, was nicht da ist. UML bietet für viele praktische Probleme in der Entwicklung, erst recht, wenn es um .NET und C#-Spezifika geht, keine (brauchbare) Antwort. Muss sie aber auch nicht, sie soll ja generisch sein.
- Verwenden Sie einige Zeit in die Suche nach einem geeigneten Tool.
- Wenn dieses Tool keinen vernünftigen Code erzeugt und handgemachte Änderungen am Code überschreibt oder nicht in das Modell übernehmen kann, dann verzichten Sie besser gleich auf die Codegenerierung. Daran scheitern leider die meisten Tools, sofern sie überhaupt Code erzeugen können.

So viel zu OOA, OOD und UML. Für eine gute Software war dies ein erster, wichtiger Schritt, aber nicht das Ende. Der folgende Abschnitt beschäftigt sich daher mit einigen wichtigen Designentscheidungen, die bereits vor der Codierung getroffen sein sollten.

3.3 Designentscheidungen

Wir Autoren wählen unsere Fallbeispiele oft so, dass sie möglichst eindeutig sind und wenig Fragen offenlassen. Ich habe mich bemüht, in diesem Buch möglichst realistische Beispiele zu verwenden, aber auch ich musste die Beispiele klein genug halten, damit sie in den Rahmen eines Buchs passen.

Vermutlich werden Ihre Produktmanager und andere Ideengeber nicht so nett sein, sie werden Sie mit all den komplexen, widersprüchlichen und bisweilen unrealistischen Anforderungen konfrontieren, die nun einmal Ihr Geschäft ausmachen. Eine Schnellstraße gibt es nicht – Software zu entwickeln ist und bleibt eine anspruchsvolle Aufgabe; Wissen und Erfahrung sind Ihre wichtigsten Werkzeuge.

Zwar müssen wir heute weniger über Implementierungsdetails nachdenken, und die umfangreiche .NET-Klassenbibliothek und viele, viele Frameworks nehmen uns einiges an Arbeit ab. Dafür sind aber die Anforderungen gestiegen, und die Entwicklung hat sich ganz allgemein auf eine höhere Ebene verlagert, sie ist abstrakter geworden. Schon 1991 hatte ein Compiler-Hersteller für das brandneue Betriebssystem OS/2 2.0 versprochen, Entwicklung wäre demnächst ganz einfach: Man müsse sich nur eine Anwendung im Designer zusammenklicken, den Rest erledige der Compiler. Ich warte immer noch darauf, wenn auch die Hersteller in der Zwischenzeit nichts unversucht gelassen haben und mit *LightSwitch* gerade ein neuer Stern am RAD(Rapid Application Development)-Himmel aufgezogen ist. Aber während ich noch weiter warte, sehen wir uns doch einige Erleichterungen in Form von Designempfehlungen an.

3.3.1 Gutes Design, schlechtes Design

Ich habe schon an verschiedenen Stellen über die Merkmale guter Softwaresysteme gesprochen. In diesem Abschnitt gehe ich auf, wie ich meine, besonders wichtige Designgrundsätze ein und zwar im Hinblick auf die Umsetzung und die dafür nötigen Designentscheidungen, also weniger auf die Grundsätze. Einige davon widersprechen sich allerdings, so dass ein guter Kompromiss gefragt ist.

Zukunftsorientiert

Software wirkt auch immer in die Zukunft, denn sie bleibt meist nicht lange so, wie sie ist. Sie entwickelt sich weiter, wird gelegentlich einem Redesign unterzogen und letztendlich durch eine andere Software ersetzt. Eine gute Spezifikation enthält daher nicht nur die heutigen Anforderungen, sondern blickt auch in die Zukunft:

- Welche Anforderungen werden nicht umgesetzt, weil Zeit und Budget nicht ausreichen?
- Welche Änderungen am Markt werden vermutlich eintreten, und welche Auswirkungen werden sie haben?
- Wie entwickelt sich der gesamte Bereich weiter?
- Welche geplanten organisatorischen Änderungen gibt es?
- Welche Verbindungen mit anderer Software wären wünschenswert, beispielsweise in Form von Schnittstellen?

Sie sollten diese Fragen in Ihre Spezifikation mit aufnehmen, beispielsweise in einem Kapitel, das Sie »Ausblick« nennen. Natürlich müssen derartige Anforderungen nicht sofort umgesetzt werden, aber die Designentscheidungen sollten diese Dinge bereits jetzt berücksichtigen, wo immer möglich.

Immer wieder treffe ich auf die beiden Extreme, die selbstredend zu vermeiden sind:

1. Stures Festhalten an der Spezifikation, auch wenn dort erkennbar Lücken vorhanden sind. Weite Teile sind fest codiert.
2. Der Versuch, das System so zukunftssicher zu machen, dass die Software entweder nicht mehr zu bedienen ist, weil viel zu generisch, und darüber hinaus nie fertig wird.

Als Designer und Entwickler sind wir natürlich keine Fachanwender. Allerdings besitzen wir eine Fähigkeit, die für die Fachanforderungen unerlässlich ist: zu erkennen, wann eine Anforderung unzureichend, lückenhaft, nicht umsetzbar, zu allgemein oder zu spezifisch ist – und Präzision. Diese Fähigkeit zu nutzen, bedeutet während der Designphase, die Anforderungen zu hinterfragen und entstehende Fragen mit den Fachanwendern zu diskutieren.

Nicht mehr und nicht weniger als das Geforderte

Gerade eben ging es eher um die Anforderungen, hier nun um die Umsetzung im Design. Gute Software erledigt genau das, was von ihr gefordert wird. Nicht weniger, aber auch nicht mehr, wobei ich davon ausgehe, dass Sie die erste Runde schon hinter sich haben – aus den Anforderungen ein fachlich vollständiges Lastenheft zu

erstellen. Beim Anfertigen eines Modells besteht auch immer die Gefahr, dass es unnötig umfangreich wird. Sie können dies vermeiden, indem Sie

- nur die Klassen modellieren, die sich aus der Anforderung ergeben,
- nur dann weitere Klassen abstrahieren, wenn sich dafür ein konkreter Nutzen für die Implementierung ergibt, der schon heute zum Tragen kommt oder der bereits absehbar ist,
- Basisklassen nur dann bilden, wenn die Umsetzung dadurch erleichtert wird, und
- nur Beziehungen modellieren, die in der Spezifikation gefordert sind.

Sollten Sie davon abweichen, dann hat das gute Gründe. Allerdings ist es dann unbedingt erforderlich, die Anforderungen ebenfalls zu erweitern, so dass Anforderungen und Software einander entsprechen. Es gibt nichts Unbrauchbareres als ein Lastenheft, das bereits nach der Einführung einer Software obsolet geworden ist – pure Zeitverschwendung!

Wartbarkeit

Gute Software ist wartbar. In Bezug auf die Modellierung der Objekte bedeutet dies:

- Die Benennung der Objekte ist eindeutig und selbsterklärend.
- Die Objekte sind klar voneinander getrennt, ein Objekt dient nur einer Aufgabe.
- Es gibt keine redundanten Elemente im Modell.
- Das gesamte Modell ist wohl dokumentiert.
- Attribute und Methodennamen sind ebenfalls eindeutig und selbsterklärend.
- Die Sichtbarkeit wurde gut durchdacht, es gibt nur wenige Elemente, die `public` sind.
- Die Objekte sind weder zu umfangreich noch zu klein, das Modell zersplittert nicht in viele Klassen und ist auch nicht trivial, mit nur sehr wenigen Klassen (es sei denn natürlich, die Anforderungen ließen dies zu).
- Es gibt ein eindeutiges und nachvollziehbares Konzept, wie die Objekte später persistiert werden, beispielsweise in einer Datenbank.
- Das Modell ist nicht zu umfangreich ausgestaltet, nicht alle Details müssen in einem Modell enthalten sein. Denken Sie daran: Ein Klassendiagramm ist eine Vereinfachung der Wirklichkeit, ein Modell eben.
- Das Modell ist jederzeit aktuell, ebenso die Anforderungen (s. o.).
- Nutzen Sie die Möglichkeit von Paketen, um komplexe UML-Diagramme logisch zu unterteilen, und die Möglichkeiten von .NET und C#, um den Code zu strukturieren.

Effizienz

Eines der größten Probleme eines strikten Top-down-Ansatzes ist, dass er zu Anwendungen mit schlechter Performance führen kann. Nicht selten werden Modelle entwickelt, die hunderte von Klassen enthalten und eher akademischen als praktischen Wert haben.

Ihren Anwendern ist es egal, wie viele Objekte Sie im Laufe der Programmbenutzung instanziieren und verwalten müssen, schlechte Performance oder Abbrüche wegen Speichermangel werden sie nicht akzeptieren. Vor der Erstellung eines Modells sollten Sie daher einige Grundsätze beherzigen:

- Hören Sie rechtzeitig auf, und widerstehen Sie der Versuchung, alle Details mit Hilfe von Objekten zu modellieren. Manche Dinge lassen sich leichter, schneller und performanter mithilfe einfacher Prozeduren lösen. Auch Stored Procedures oder SQL-Statements haben weiterhin ihre Berechtigung in der Entwicklung von Software.
- Dazu sollte vorab klar sein, welche Teile besonders speicher- oder performancesensitiv sind. Es sind dies oft die Teile, die besonders häufig aufgerufen werden und/oder viele Daten benötigen. Auch der Remote-Zugriff auf Services und Datenbanken zählt dazu. Optimieren Sie diese Stellen von Hand, das beste Modell kann Ihnen dabei nicht helfen.
- Bedenken Sie, dass Objekte Speicher auf dem Heap kosten. Dieser Speicher muss nicht nur allokiert, sondern auch beizeiten wieder freigegeben werden. Der Garbage Collector wurde in der Version 4.0 des .NET Frameworks zwar deutlich verbessert (und in der aktuellen Version abermals), nachdem er sich seit Version 1 nur wenig verändert hatte, aber es geschieht immer wieder, dass die Performance durch extensives Freiräumen von Ressourcen erheblich leidet.
- Wenn Sie objektrelationales Mapping einsetzen, dann sind Ihre Objekte vielleicht doppelt vorhanden: zum einen als *POCO* (*Plain Old CRL Object*), zum anderen als OR-Entities. Das erhöht den Speicherverbrauch und verlangsamt die Geschwindigkeit.
- Massendatenoperationen sollten nur dann mithilfe von Klassen realisiert werden, wenn die Auswirkungen absehbar und der Nutzen quantifizierbar sind. Ich habe schon erlebt, dass im Rahmen der Euro-Umstellung für Millionen von Datensätzen Objekte erzeugt wurden, nur um den Wert weniger Spalten umzurechnen.

Obwohl wenig davon zu lesen ist, halte ich dies für eines der größten Probleme in der IT. Durch die zunehmende Virtualisierung und durch immer mehr Zwischenschichten in der Softwareentwicklung wird jede noch so kleine Aufgabe zu einer Herkulesaufgabe für Laufzeitsystem und Betriebssystem.

Aus der Praxis

Noch vor wenigen Jahren konnte ich machen, was ich wollte: Die Datenbankserver waren eigentlich immer zu klein, die IO-Last zu groß, der Speicher zu klein und die Prozessoren chronisch überlastet. Aber immerhin konnte man durch weiteres Aufrüsten und Austauschen von Hardware die Grenzen beständig weiter verschieben.

Das hat sich verändert. Einer meiner aktuellen Datenbankserver hat 4 Prozessoren mit insgesamt 64 Kernen, 192 GB Arbeitsspeicher, und auch die IO-Bandbreite ist gestiegen, wenn auch weniger langsam als der Rest. Die CPUs sind nur selten mehr als zu 20 % ausgelastet. Und dennoch: Die Anwendungen sind nicht viel schneller geworden. Ein typisches Szenario, aber warum?

- Zusätzliche Schichten (s. o.) kosten zusätzliche Performance.
- Handoptimierter Code weicht generiertem Code, zum Beispiel auf den Server angepasste SQL-Statements werden durch OR-Mapper-generierte SQL-Statements ersetzt – und überdies oft die Grundregeln von Indizes verletzt.
- Immer mehr Verbindungen zwischen den Systemen erhöhen die Latenzzeiten, was sich in Wartezeiten bemerkbar macht.
- Aktuelle Software ist oft nicht in der Lage, viele verschiedene Kerne effizient auszulasten, vor allem wenn 64 physische Kerne im Spiel sind.
- Netzwerke und andere Technologien haben nicht in demselben Maß Schritt gehalten.

Kurz: Moderne Systeme sind heute häufig ineffizienter als alte Systeme und oft über deutlich mehr CPU-Zyklen mit Warten beschäftigt. Warten auf Sperren, Warten auf eingehenden Netzwerkverkehr oder Warten auf die Bestätigung anderer Prozesse.

Leistungsfähige Systeme auszulasten ist also eine Herausforderung. Ich empfehle Ihnen daher, immer diejenigen Prozesse und Module ausfindig zu machen, die effizient ablaufen müssen, und dort die gängigen Prinzipien der Softwareentwicklung auch einmal über Bord zu werfen, wenn es sich lohnt.

Hilft alles nichts, dann lässt sich manchmal viel dadurch gewinnen, indem man dem Anwender nur die Illusion von Geschwindigkeit gibt, durch asynchrone Aufrufe oder Berücksichtigung von Abschnitt 3.7, »Vom Umgang mit der Zeit in Anwendungen«.

Dokumentation

Das Problem fehlender Dokumentation ist meistens, dass nachfolgende Entwickler die Gedankengänge des Vorgängers nicht kennen und daher oft Lösungen verwerfen, die sie nicht verstehen. Eine Dokumentation kann dem vorbeugen.

Wir haben uns hier bislang auf die Möglichkeiten von Visual Studio konzentriert. Es gibt aber viele freie und kommerzielle Werkzeuge auf dem Markt. Diese Werkzeuge helfen dann auch bei der Dokumentation, die besseren erlauben sogar mehrere Revisionen von Modellen und Kommentaren.

Dokumentation gibt es während der drei Phasen in der Entwicklung:

1. im Vorhinein, durch gute Lasten- und Pflichtenhefte
2. während der Entwicklung, im Code oder bei der zeitnahen Anfertigung von technischen Dokumenten
3. im Nachhinein, zum Beispiel die nachträgliche Dokumentation eines Klassenmodells

Über die ersten beiden habe ich schon ausführlicher berichtet oder werde dies an verschiedenen Stellen noch tun.

Zu 3 möchte ich die stark verbesserten Fähigkeiten von Visual Studio 2012 nicht unerwähnt lassen, bestehende Klassenstrukturen transparent zu machen; wie ich finde, ein Meilenstein. Abhängigkeitsdiagramme nennt dies das deutschsprachige Visual Studio und, ich traue es mich fast nicht zu schreiben, es ist ein Feature der Ultimate-Version. Daher, und auch weil es das Thema sprengen würde, ein Screenshot als Appetizer, hier von der Enterprise Library:

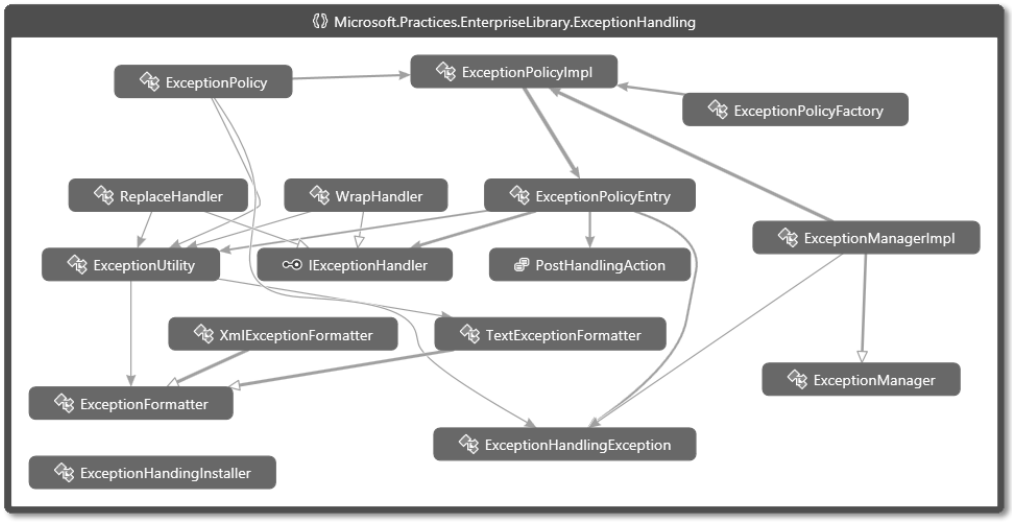


Abbildung 3.15 Abhängigkeitsdiagramm, Auszug aus der Enterprise Library

Meine Tipps dazu:

- Arbeiten Sie mit einem festen Set an Dokumentationstemplates, den Sie gut kennen und der Ihnen so lästige Dinge wie die Gliederungsnummerierung abnimmt.

- Sie haben kein Visual Studio Ultimate? Ebenfalls mehr als brauchbar ist NDepend (*www.ndepend.com*). Zwar auch nicht umsonst, aber nicht sehr teuer und sein Geld wirklich wert, zumal die Möglichkeiten zur Analyse bestehender Anwendungen äußerst umfangreich sind.
- Erstellen Sie in größeren Projekten ein Metadokument, in dem Sie auf die vielen anderen Dokumente, die im Laufe der Entwicklung entstehen werden, verlinken können.
- Wikis sind ebenfalls eine gute Sache, vor allem, wenn sie eine Volltextsuche mitbringen.
- Vergessen Sie bitte niemals wichtige Formalien, wie die Versionsnummer des aktuell gültigen Dokuments.

Erweiterbar

Ein gutes Design ist erweiterbar und damit auch die Programme, die daraus entstehen. Das ist nun so eine Sache, denn Prognosen sind schwierig, vor allem, wenn sie die Zukunft betreffen, das ist bekannt.

Unter dem Merkmal »zukunftsorientiert« habe ich bereits ausgeführt, dass eine gute Spezifikation die Zukunft schon ein wenig vorwegnehmen kann. Aus Sicht des Modells gibt es einige Empfehlungen, um Software und Modell erweiterbar zu halten:

- Suchen Sie nach gemeinsamen Vorfahren und nutzen Sie Generalisierung/Spezialisierung. Referenzieren Sie immer die oberste Klasse in einer Vererbungshierarchie, die Ihre Anforderungen gerade eben noch erfüllt.
- Schnittstellen sind eine Möglichkeit, einen Hauch von Mehrfachvererbung in C#-Programme zu bringen und Funktionalitäten zu implementieren, ohne bestehenden Code verändern zu müssen.
- Eventuell sind *Inversion of Control* (*IoC*) bzw. *Dependency Injection* (*DI*) Alternativen für Sie. Dabei wird zum Beispiel per Konfiguration festgelegt, welche Klasse zur Laufzeit verwendet wird. *Unity* ist ein leichtgewichtiger Vertreter eines IoC-Containers.
- Verwenden Sie eine Architektur, die dem Rechnung trägt.
- Vermeiden Sie allzu enge Kopplung, vor allen dann, wenn Klassen sehr spezielle Aufgaben erfüllen.
- Klassen, deren Vererbung mit dem Schlüsselwort `sealed` unterbunden wird, verhindern Erweiterbarkeit, ebenso wie Methoden, die nicht virtuell sind. Setzen Sie beides mit Bedacht ein.
- Events und Callbacks sind typische »Extension Points«, nicht nur für den GUI-Entwickler.

3.3.2 Exception-Handling

Wenn Ihnen auch die Zeit für alles andere fehlen sollte, unter allen Umständen sollten Sie eine gute und strukturierte Fehlerbehandlung einbauen. Ich gehe in Kapitel 4, ».NET für Fortgeschrittene«, näher darauf ein, daher behandle ich das Thema abermals nur aus der Blickrichtung des Anwendungsdesigners. Exceptions werden üblicherweise nicht modelliert, sie gehören zu den vielen Aspekten der Umsetzung, die oft nicht oder nicht klar geregelt sind. Ich empfehle Ihnen, die Verfahren zur Fehlerbehandlung vorab zu regeln. Vielleicht kennen Sie Software, die Sie dutzende Male starten mussten, nur um eine Aufgabe zu erledigen, weil sie immer wieder abstürzt. Das ist lästig oder gar gefährlich, genauso wie

- Fehler, die niemand mitbekommt, weil sie irgendwo geschluckt werden,
- lange, komplizierte Fehlermeldungen, die ein Anwender nicht versteht,
- läppische Fehlermeldungen, wie »Fehler aufgetreten«,
- Fehler, die in der Fehlerbehandlung auftreten,
- Fehler, die in Schleifen mitgeteilt werden, in denen der Anwender dann dutzende Male ein Popup-Fenster wegklicken muss.
- Weitere Beispiele finden Sie in Kapitel 4, ».NET für Fortgeschrittene«.

Exception-Hierarchie

Exception-Klassen, die von `Exception` oder einer anderen Basisklasse abgeleitet werden, lassen sich in UML natürlich genauso modellieren wie fachliche Klassen. Beim Entwurf eigener Exception-Klassen sollten Sie sich die Fehlerkategorien erarbeiten, die in Ihren Programmen vorkommen, zum Beispiel Fehler in der Kommunikation mit Ihrer Middleware, im Datenzugriff oder in der Validierung von Benutzereingaben.

Wenn Sie eigene Exception-Klassen entwerfen, so sollten Sie die ursprüngliche .NET-Exception nicht verwerfen. Anstatt also zu schreiben

```
try
{
   ValidateInput();
}
catch(Exception ex)
{
   throw new KalimbaValidationException(ex.Message);
}
```

sollten Sie lieber die `InnerException`-Eigenschaft verwenden und die Original-Exception dort zuweisen:

```
try
{
  int i = 5 / 0; //Unerwarteter Fehler
}
catch(Exception ex)
{
  throw new KalimbaValidationException ("Es ist ein Fehler beim
     Validieren Ihrer Eingaben aufgetreten", ex);
}
```

In Abschnitt 3.8.2, »Der Exception Handling Application Block«, finden Sie ein Tutorial zum Einsatz der Enterprise Library für das Behandeln von Exceptions in Anwendungen.

Logging

Definieren Sie, ob und wenn ja, welche Exceptions geloggt werden sollen. Dies ist aus mehreren Gründen wichtig: Einerseits dient es der Nachverfolgung von Fehlern, beispielsweise für das Debugging. Andererseits hilft es der Objektivierung einer Aussage, denn nicht selten übertreiben Anwender in der Schilderung der Fehler maßlos. Außerdem kann es hilfreich sein, im Fehlerfall die Stelle zu kennen, an der eine Operation wieder aufgesetzt werden soll.

Meldung

Welche Fehler soll der Anwender zu Gesicht bekommen, welche Fehler sollen ihm verborgen bleiben? Wie soll er die Fehler sehen, in einem eigenen Dialog oder über `MessageBox.Show()`? Möchten Sie, dass er die Original-Fehlermeldung sieht oder stattdessen eine selbst getextete Fehlermeldung?

In vielen Büchern werden Sie lesen, dass die `Exception`-Basisklasse niemals selbst behandelt werden sollte. Ich möchte Kapitel 4, ».NET für Fortgeschrittene«, nicht vorgreifen, aber ich halte das für falsch. Sie sollten Ihre Anwendung immer so entwerfen, dass das Programm niemals beendet wird, wenn es nicht absolut notwendig ist. Die meisten Fehlermeldungen, die in der Praxis auftreten, machen ein Weiterarbeiten absolut möglich. Was soll Ihr Anwender mit einer .NET-Fehlermeldung anfangen, gar noch mit einem StackTrace versehen, nach deren Anzeige sich das Programm beendet?

Ressourcen

Beachten Sie auch den Umgang mit zentralen Ressourcen. Oft wird der `finally`-Teil eines `try`-Blocks vergessen, und so bleiben zum Beispiel Scanner geblockt, obwohl es nicht nötig wäre, wenn in der TWAIN-Schnittstelle ein `finally`-Block dies regeln würde.

3.3.3 Logging

Auch beim Logging sollten Sie einige Designentscheidungen treffen. Am Ende des Kapitels gibt es hierzu ein Tutorial, sodass ich hier nicht näher darauf eingehe und mich mit der Feststellung begnüge, dass eine gute Anwendung umso mehr loggt, je umfangreicher sie ist.

3.3.4 Datenmodell

Das Erstellen des Datenmodells gehört zum Design einer Anwendung natürlich dazu. Näheres zu Datenbanken finden Sie in Kapitel 7, »Datenbank und Datenzugriff«.

3.4 Schnittstellen und Integration

Unternehmensanwendungen sind selten isolierte Systeme, sie sind meistens in eine IT-Landschaft eingebunden, in der es viele zugekaufte oder selbst entwickelte Programme gibt. Abbildung 3.16 zeigt ein Beispiel. In der Praxis sind Anzahl und Vernetzungsgrad der Systeme noch komplexer. Darüber hinaus sind die Technologien häufig unterschiedlich; so mögen sowohl .NET als auch native Anwendungen darunter sein, Windows- als auch Unix-Lösungen, Rich Clients als auch Thin Clients, Tablet- und Webanwendungen.

Die Welt wäre viel einfacher, wären wenigstens die Datenbanken kompatibel, aber das sind sie selten. Feldbezeichnungen, Feldlängen, Tabellenbeziehungen und oft genug sogar Feldtypen unterscheiden sich von Datenbank zu Datenbank meist deutlich. Informationen sind zudem häufig mehrfach vorhanden, Kundendatensätze beispielsweise im ERP-System, im CRM-System oder in den Datenbanken des Adressmanagements und vielleicht noch in einigen Auswertungsdatenbanken, von den vielen Exportdateien, die ihr Dasein in Dateiverzeichnissen fristen, ganz zu schweigen. Und für die Business Intelligence Software sollen alle Informationen zentral bereitstehen, für den einheitlichen Blick auf den Kunden und die Prozesse.

Von alledem sollen die Kunden nichts mitbekommen. Für sie soll der Vorgang, von der Anfrage bis zur Bestellabwicklung, harmonisch ablaufen. Doch wer einmal bei einem bekannten Telefonanbieter ein Problem mit seinem DSL-Anschluss hatte, der weiß, das ist nicht immer so. Die Illusion des einheitlichen Systems bricht dort zusammen, wo Schnittstellen nicht das leisten, was sie sollen, oder wo sie sogar fehlen. Grund genug, den Schnittstellen einen eigenen Abschnitt zu widmen.

Kapitel 8
Workflow Foundation

Das Ganze ist mehr als die Summe seiner Teile.
(Aristoteles)

Willkommen zu einer der großartigsten Neuerungen in der .NET-Klassenbibliothek: der *Workflow Foundation* (*WF*). Es gibt sie seit der Version 3.0, als Microsoft erstmals eine .NET-Version vorstellte, die eigentlich keine neue Version war, sondern vielmehr aus einer Reihe von Erweiterungen für die bestehende Version 2.0 bestand. Mit der Version 4.0 hat Microsoft die WF zu großen Teilen neu geschrieben und in der aktuellen Version 4.5 viele der Features nachgerüstet und Mängel beseitigt, für die in der Version 4 wohl keine Zeit mehr war; allen voran die State Machine Workflows, mit denen sich manche Workflows besonders elegant beschreiben lassen, und die es vor dem Redesign der WF schon einmal gab.

Viele Programme implementieren heute schon Workflows, denn kaum eine Business-Anwendung verkauft sich heute noch ohne. Workflows sind ein Synonym geworden für das Abbild von Prozessen in Software, aber auch für Konfigurierbarkeit, denn Workflows passen sich dem Anwender an, nicht umgekehrt. Sie sind aber keineswegs auf Geschäftssoftware beschränkt, und so gehen wir im ersten Abschnitt der Frage nach, was Workflows eigentlich sind, wann sich der Einsatz der WF lohnt und aus welchen Teilen ein Workflow besteht.

Die folgenden Abschnitte bauen dann aufeinander auf und führen Sie systematisch in die Workflow Foundation ein. Sie sind eine Mischung aus Lehrbuch und Tutorial und befassen sich jeweils mit einem Thema innerhalb der WF. Am Anfang finden Sie meistens ein kleines Fallbeispiel, das als Leitfaden für Erläuterungen und Umsetzung dient. Insgesamt ergibt sich so eine umfangreichere Anwendung, natürlich aus der Geschäftswelt der Kalimba Sunfood.

8.1 Einführung

Der Einsatz der WF unterscheidet sich stark von der Programmierung klassischer Anwendungen. Einerseits stellt WF eine neue API zur Verfügung, andererseits verlangt sie vom Entwickler, dass er seine Anwendungen mit einem grafischen Work-

flow Designer entwirft, eine zwar logische, aber keineswegs gewohnte Umgebung für die meisten Entwickler.

Hinzu kommt, dass die WF in den Vorgängerversionen (vor 4.0) nicht in allen Bereichen gelungen war. Viele Entwickler klagten über schlechte Performance, Bedienungsmängel im Designer, eine schlechte Integration mit anderen Produkten und eine bisweilen gewöhnungsbedürftige API. Die meisten dieser Mängel wurden allerdings in der Version 4.0 behoben.

8.1.1 Warum Workflows?

Die erste Frage ist vielleicht die wichtigste: Warum sollten Sie sich in eine neue Technologie einarbeiten, wo doch bereits mit dem .NET Framework und C# alle Arten von Anwendungen entwickelt werden können? Das scheint mir eine Frage zu sein, die öffentlich zu wenig beantwortet wird, vielleicht ein Grund dafür, warum die WF in der Vergangenheit nicht die Verbreitung gefunden hat, die sie verdient hätte. Um diese Frage zu beantworten, sollten wir uns einmal typische Anforderungen anschauen, die an moderne Software gestellt werden.

Prozessorientierung

Es verwundert mich immer wieder, wie sehr sich Unternehmen in ihren Prozessen unterscheiden, obwohl sie dasselbe Geschäftsmodell haben, dieselben Kunden bedienen, von vergleichbarer Größe sind und ähnliche Produkte vertreiben.

Die Bereitschaft, sich an die Strukturen einer Software anzupassen, sinkt beständig. Kunden erwarten heute, dass die Software so flexibel ist, dass sie nichts oder sehr wenig ändern müssen, um sie einzuführen. Andererseits möchten viele Unternehmen lieber auf Standardsoftware setzen. Ein Widerspruch, der in vielen Softwareprodukten so gelöst wird: In unzähligen Konfigurationseinstellungen lässt sich das System so flexibel wie möglich konfigurieren, jedenfalls so flexibel, dass die meisten Anforderungen dadurch abgedeckt werden. Weitergehende Anforderungen werden dann im Rahmen einer kundenspezifischen Programmierung abgedeckt. Der Nachteil dieses Verfahrens liegt auf der Hand: Der Code ist durch unzählige Fallunterscheidungen an unzähligen Code-Stellen zerfleddert. Kundenspezifische Anpassungen müssen über Folgeversionen hinweg weitergepflegt werden, und ein genereller Release-Wechsel wird mit jeder Version immer schwieriger. Und ist die Software nur alt genug, wird jede Code-Änderung zum Vabanquespiel.

Workflows helfen auf dreierlei Art, diese Probleme zu vermeiden:

- Zum Ersten bilden sie eine Einheit. Ein Entwickler muss sich einen Sachverhalt nicht in mehreren Code-Dateien zusammensuchen, sondern kann ihn mit einem Blick auf den Workflow erfassen. Dennoch ist ein Drilldown in den Workflow hinein möglich, denn Workflows sind hierarchisch organisiert.

- Zum Zweiten lassen sich Workflows aus vorgefertigten Bausteinen zusammenstellen. Sie sind damit nicht auf Entwickler festgelegt, auch ein versierter Anwender könnte einen Workflow erstellen oder verändern. Es findet eine Umkehr der Verantwortung statt, der Entwickler stellt Workflow-Elemente bereit, das .NET Framework eine Infrastruktur und der Anwender das zu lösende Problem nebst Lösungsvorschlägen, die er selbst umsetzen kann.
- Zum Dritten sind Workflows konfigurierbar, denn in WF sind sie in XML-Dateien gespeichert. Ein solcher Workflow kann sich über die Versionen hinweg weiterentwickeln, oder der Workflow kann in der Anwendung selbst verändert oder ausgewählt werden. Dem Customizing öffnen sich neue Wege, denn sie können Workflows auch kundenspezifisch oder fallweise anpassen.

Wartbarkeit

Eng mit dem vorherigen Punkt verbunden ist die Wartbarkeit. Workflows sind vor allem deshalb besser wartbar, weil sie einen zentralen Punkt darstellen, an dem die Geschäftslogik abgebildet ist. Darüber hinaus lassen sich viele, wenn auch nicht alle, Änderungen grafisch im Designer vornehmen. Aus der Praxis weiß ich aber auch: Wir sollten die Kirche im Dorf lassen. Es wird nicht jede Anforderung durch einen Anwender ohne Programmierkenntnisse umsetzbar sein. Je detaillierter die Anforderung ist, desto mehr Programmierwissen ist notwendig.

Workflows ersetzen daher nicht die Programmierung in einer Programmiersprache, sondern ergänzen sie. Sie helfen dabei, den Code zu organisieren, indem sie für die obersten Ebenen einer Anwendung eine grafische Modellierung ermöglichen. Ab einer gewissen Ebene, die für jede Applikation unterschiedlich ist, lohnt sich der Einsatz dann nicht mehr, weil der Aufwand für die Modellierung im Designer den Aufwand für die Umsetzung im Code übersteigt und solche Codebestandteile auf unteren Ebenen sich ohnehin nicht mehr so häufig ändern.

Die Trennung zwischen beiden Welten ist die *Aktivität*, das zentrale Element in Workflows, wie wir später noch sehen werden. Die Aktivität ist Bestandteil des Workflows, die Implementierung der Aktivität hingegen erfolgt im Code.

Skalierbarkeit

Unternehmensanwendungen sollen immer häufiger gut skalieren, also mit wachsendem Durchsatz Schritt halten können. Dies trifft vor allem auf Serveranwendungen zu, beispielsweise eine Webseite oder einen Service. Aber auch vonseiten des Clients wird zunehmend verlangt, dass die Software die steigende Anzahl von Kernen in Prozessoren zu beschäftigen vermag. Workflows unterstützen Skalierbarkeit in mehrfacher Hinsicht:

- Es können mehrere bis viele Workflows parallel ausgeführt werden. Denken Sie beispielsweise an die Bestellannahme eines Warenhauses. Dort können viele Bestellungen parallel eintreffen und sollten dann auch so weit wie möglich parallel verarbeitet werden. Die parallele Ausführung mehrerer Workflows ist eine Eigenschaft der Runtime und des Prozesses, in dem die Workflows laufen.
- Workflows können sich schlafen legen. Die Runtime weckt sie dann wieder auf, sobald ein Ereignis von außen die Weiterführung des Workflows erforderlich macht. Dadurch werden die Ressourcen des Servers geschont, es laufen immer nur die Workflows, die auch gerade benötigt werden.
- Irgendwann reichen die Ressourcen eines einzelnen Servers nicht mehr aus. In WF, vor allem in Verbindung mit *AppFabric*, können auf einfache Art und Weise Load-Balancing-Szenarien umgesetzt werden. Es ist sogar möglich, dass ein Workflow in einem Prozess eines Rechners gestartet und zu einem späteren Zeitpunkt in einem anderen Prozess weitergeführt wird, der sogar auf einem anderen Rechner laufen kann.
- Ein Workflow selbst kann Parallelität unterstützen, in WF beispielsweise durch die *Parallel*- oder *ParallelForEach*-Aktivität.

All dies erhalten Sie mit der WF out of the box – auch dann, wenn Sie dies im Augenblick noch gar nicht benötigen – für den Preis einiger Zeilen Code und etwas Arbeit an der Konfiguration. Die WF eignet sich damit vorzüglich für die Entwicklung verteilter Anwendungen, ohne dass sie jedoch für die Entwicklung lokaler Anwendungen ihren Charme verlieren würde.

Unterstützung von asynchronen Szenarien

Lassen wir das Telefon mal außer Acht, dann sind die meisten Prozesse im Geschäftsleben asynchroner Natur. Ein Urlaubsantrag wird gestellt, aber nicht sofort bearbeitet; eine Bestellung trifft ein, die zugehörige Lieferung verlässt aber erst einige Tage später das Lager.

Was uns Menschen spielend gelingt, ist für Software ein ernsthaftes Problem. Wenn der Urlaubsantrag nicht bearbeitet wird, dann fragen wir rechtzeitig vor Urlaubsantritt nach, wenn die Lieferung den Kunden nicht erreicht, dann hilft ein Mitarbeiter im Kundenservice bei der Lösung des Problems. Software hingegen wird ungleich komplizierter, wenn wir für asynchrone Vorgänge perfekte Lösungen entwickeln wollen, zum Beispiel für dieses Szenario:

Beispiel

Die Kalimba Sunfood GmbH betreibt einen Internetshop für Cocktail-Zubehör, in dem Endkunden über einen Warenkorb Produkte bestellen können.

Die Anbindung zum ERP-System ist asynchron gestaltet, um die Verfügbarkeit des Internet-Auftritts nicht an die Verfügbarkeit der ERP-Software zu koppeln. ERP-Software und Warenwirtschaft sind ebenfalls asynchron gekoppelt, da sich das Auslieferungslager in einer anderen Stadt befindet.

Synchrone Aufrufe laufen nach dem *Request/Reply*-Pattern ab, der Erfolg oder Misserfolg eines Aufrufs steht unmittelbar danach fest. Im Fehlerfall kann ein Entwickler im Exception Handler den Fehler »abfangen«, also angemessen darauf reagieren, zum Beispiel indem er den Vorgang erneut versucht oder eine Fehlermeldung ausgibt, die Verantwortung für das Problem also (letztlich) auf den Anwender überträgt.

Das Pattern für asynchrone Aufrufe heißt *Fire and Forget*, aber genau das soll die Software nicht tun, denn in unserem Beispiel würde das im Fehlerfall bedeuten:

- Der Lagerbestand wird dezimiert, obwohl die Bestellung abgebrochen wurde, oder der Lagerbestand wurde nicht verringert, z. B. weil die Kommunikation mit der Warenwirtschaftssoftware gestört ist.
- Auf der Internetseite wird die Bestellung als aufgenommen gekennzeichnet, obwohl sie das ERP-System nie erreicht hat.

Beide Fälle verlangen nach einer *Kompensation* im Fehlerfall, zum Beispiel, indem der Lagerbestand korrigiert wird. WF unterstützt solche Kompensationen mit einer eigenen Aktivität und ermöglicht darüber hinaus die Umsetzung eines Konzepts zur Fehlerbehandlung für asynchrone Szenarien.

Asynchrone Prozesse können langlaufende Prozesse sein, man spricht dann gerne von langlaufenden Transaktionen. Das Problem dabei ist, dass der aufrufende Teil nicht weiß, wie lange er auf eine Antwort warten muss oder ob sie überhaupt eintrifft. Was langlaufend ist, hängt vom Einsatzgebiet ab und kann sich von wenigen Sekunden bis hin zu einigen Monaten oder gar Jahren erstrecken. Das wirft einige Probleme auf:

- Der Prozess muss so lange warten, bis eine Antwort eintrifft. Dies führt zu vielen gleichzeitigen Prozessen oder Threads und damit zur Verschwendung von Ressourcen.
- Der Prozess könnte zwischenzeitlich auch beendet sein, die Antwort würde den Empfänger dann überhaupt nicht mehr erreichen.
- Ein Außenstehender könnte nicht sofort erkennen, an welchem Punkt der Prozess unterbrochen wurde und welche Bestandteile bereits erfolgreich durchlaufen wurden.

Natürlich gibt es auch konventionelle Lösungen für diese Probleme, zum Beispiel könnte eine Anwendung den Status in eine Datenbank schreiben und später, beim Eintreffen der Antwort, wieder daraus lesen. Ein Listener-Prozess, zum Beispiel der IIS, könnte auf unbestimmte Zeit auf eine Antwort warten. Und für die Nachvollziehbarkeit könnte eine Anwendung eine Log-Datei schreiben.

Dies setzt jedoch immer entwicklerseitigen Code voraus, der die Businesslogik wiederum zerstreut und erst einmal geschrieben werden muss. Viele solcher Problemstellungen zeigen ihr hässliches Gesicht erst, wenn man sie gründlich durchdenkt. Eine robuste Lösung für Kompensationen oder asynchrone Kommunikation über Prozessgrenzen hinweg ist keineswegs trivial und übersteigt oft die in der Praxis verfügbare Zeit.

Die WF bietet auch hierfür fertige Lösungen: Workflows werden automatisch persistiert, oder Sie können dies manuell tun. Im Zusammenspiel mit *AppFabric* und dem SQL Server ist das, nebenbei bemerkt, besonders komfortabel. Beim Persistieren wird der Status des Workflows geschrieben, der aktuelle Punkt der Ausführung und alle Inhalte von Variablen und Argumenten und der Kontext. Beim Eintreffen einer Antwort erweckt die WF den Workflow wieder zum Leben und führt ihn mit der nächsten Aktivität fort. Auch hier müssen Sie gar nichts selbst entwickeln, Sie können aber Einfluss darauf nehmen, wenn Sie möchten. Persistenz wird in Abschnitt 8.9, »Persistenz«, behandelt, und in Abschnitt 8.8, »Transaktionen«, finden Sie Informationen zu Transaktionen und Kompensationen.

Transparenz

Die Sorge vor Blackboxes wächst, also Software, der man ihre Funktionsweise weder ansieht noch in sie hineinsehen kann. Bei Workflows ist das aus naheliegenden Gründen besonders kritisch.

Der Markt verlangt daher nach Transparenz, die ein Entwickler durch den Einsatz der konfigurierbaren Tracking-Mechanismen herstellen kann. Dem Tracking widmen wir uns in Abschnitt 8.10, »Tracking und Tracing«.

Wiederverwendbarkeit

Workflows sind von Haus aus wiederverwendbar, denn sie bestehen aus Aktivitäten. Diese sind in etwa vergleichbar mit Komponenten und stellen die unterste Ebene der Wiederverwendbarkeit dar.

Da Aktivitäten auch andere Aktivitäten beinhalten können, man nennt sie dann *Composite Activities*, können auch Konglomerate auf höheren Ebenen als Ganzes wiederverwendet werden, bis hin zum gesamten Workflow.

Aktivitäten können auch in die Toolbox platziert werden, wenn Sie sie häufiger benötigen. Inzwischen finden Sie auf zahlreichen Seiten im Internet vorgefertigte Aktivitäten zum Download.

Workflowfreie Szenarien

Ein Workflow Designer ist keine Programmiersprache, auch wenn die Workflow Foundation Programmierkonstrukte, wie z. B. Schleifen, als Aktivitäten anbietet. Die WF ist daher vermutlich nicht die richtige Wahl, wenn Ihre Prozesse

- statisch sind, sich also wenig bis gar nicht verändern,
- nicht durch den Anwender verändert werden sollen,
- der Aufwand für die Erstellung der Aktivitäten durch den Vorteil der Konfigurierbarkeit nicht aufgewogen wird,
- hinsichtlich der Geschwindigkeit besonders optimierungsbedürftig sind, beispielsweise Anwendungen für das *Number Crunching*,
- viel speziellen Code enthalten, zum Beispiel für die Kommunikation mit Hardware, oder
- aus vielen Aktivitäten bestehen, die eigens entwickelt werden müssten, die aber jeweils nur an wenigen Stellen Verwendung fänden.

Auch wenn die Workflow Engine sehr viele Instanzen erzeugen und verarbeiten kann: In Schleifen mit hunderttausenden Durchläufen jeweils eigene Workflow-Instanzen anzulegen, kann einen Performance-Overhead verursachen, der den Einsatz ebenfalls infrage stellt.

Wenn Sie schließlich fremde Produkte integrieren müssen, womöglich noch Produkte, die nicht auf Basis von .NET entwickelt wurden, dann sollten Sie sich lieber nach fertigen Integrationswerkzeugen umsehen, die selbst auch oft einen Workflow Designer integriert haben, zum Beispiel *Microsoft BizTalk*.

Dennoch eignen sich die meisten Aufgabenstellungen gut bis ideal, um mit WF umgesetzt zu werden, oft auch solche, bei denen man es auf den ersten Blick gar nicht vermuten würde. Und wenn Sie unsicher sind: Beginnen Sie mit einem Workflow. Sie werden beim Modellieren dann schnell Klarheit gewinnen, ob sich eine Aufgabenstellung für die Workflow Foundation eignet.

Aus der Praxis

In den meisten Unternehmen gibt es viele gelebte Prozesse, die sich für die WF eignen, zum Beispiel die Bearbeitung eingehender Bewerbungen, die Genehmigung von Urlaubsanträgen, Prozesse rund um die Auslieferung und Lagerhaltung oder das Bestellwesen.

In vielen Unternehmen gibt es für einige, oder alle, dieser Prozesse Standardsoftware, die heutzutage nahezu allesamt Workflows beherrschen. Leider zeigt sich hier, dass Workflow eben nicht gleich Workflow ist, denn was einem in der Praxis häufig als Workflow verkauft wird, ist meist nichts anderes als einige Felder, die eine Reihe von Zuständen durchlaufen können.

»Echte« Workflows, so wie sie in der WF erstellt werden können, sind da ungleich flexibler, weil sie Kontrollstrukturen, wie Entscheidungen oder Schleifen kennen und verschiedenste Aktivitäten mitbringen, erweiterbar sind und asynchrone Vorgänge unterstützen. Achten Sie also am besten darauf, wenn ein Hersteller von Software wieder einmal behauptet: »Workflows? Klar, das können wir schon seit zehn Jahren!«

8.1.2 Der Workflow

So viel zum Einsatz der WF, wenden wir uns nun den Workflows selbst zu.

Anatomie

Definition

Ein *Workflow* ist ein Arbeitsablauf, der aus *Aktivitäten* besteht, die in einer bestimmten, vorher festgelegten Abfolge ausgeführt werden. Ein Workflow bildet daher immer einen bestimmten Prozess ab, meist einen Geschäftsprozess innerhalb einer Organisation.

Einige Aktivitäten können selbst wiederum Aktivitäten beinhalten, sodass sich eine Hierarchie ergibt. Im Grunde genommen ist in WF alles eine Aktivität. Die Aktivität auf der obersten Stufe nennen wir dann einen Workflow (siehe Abbildung 8.1).

Workflow und *Workflow-Instanz* verhalten sich zueinander wie Klasse und Objekt. Von einem Workflow lassen sich beliebig viele Instanzen starten, die dann in der Laufzeitumgebung der Workflow Foundation parallel ausgeführt werden. Jede dieser Instanzen besitzt ihren eigenen Status (nennen wir ihn Kontext) und kann in der Ausführung unterschiedlich weit fortgeschritten sein.

Workflows sind in Software nichts Neues, es gibt sie schon seit vielen Jahren. Und mit der *Business Process Modeling Language* (*BPML*) gibt es sogar eine auf XML basierende Metasprache zur Modellierung solcher Prozesse (= Workflows). Für die Darstellung von Workflows gibt es am Markt unzählige freie und kommerzielle Werkzeuge; viele davon nutzen die *Business Process Modeling Notation* (*BPMN*), einen Weg, Workflows grafisch darzustellen.

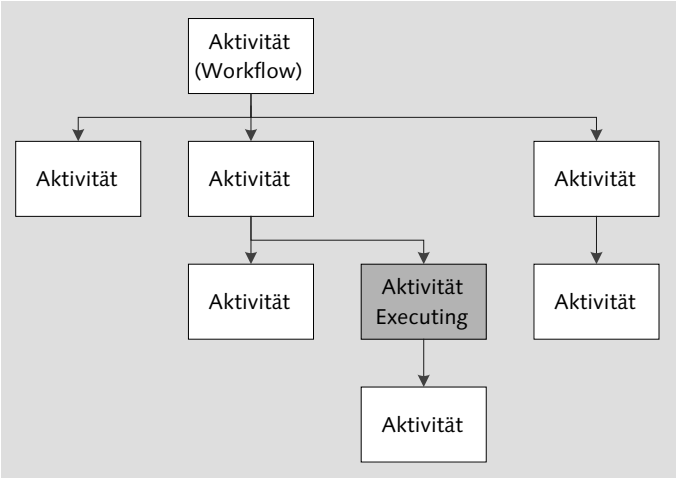


Abbildung 8.1 Ein Workflow als Hierarchie von Aktivitäten

Wie Sie schon in Abschnitt 8.1.1, »Warum Workflows?«, erfahren haben, werden Workflows *modelliert* und *konfiguriert*. Damit unterscheiden sie sich von der klassischen Programmierung, die natürlich ebenfalls Geschäftsprozesse abzubilden vermag.

Damit der versierte Endanwender selbst einen Workflow gestalten kann, steht ihm ein Satz an *Aktivitäten* zur Seite, den er in seinem Workflow mithilfe von *Kontrollstrukturen* nahezu beliebig verknüpfen kann.

Diese Aktivitäten sind vordefiniert und bestehen aus klassischem Code, denn wollte ein Anwender beliebige Programme mittels Workflows erzeugen, dann wäre der Workflow Designer letztendlich genauso komplex wie die Programmierumgebung selbst und damit für ihn unbrauchbar. Die WF gleicht also einem Baukasten, die einzelnen Teile sind ihre Aktivitäten und die Aktivitäten selbst bestehen letztendlich aus Code (siehe Abbildung 8.2).

Die WF bringt einen Satz fertiger Aktivitäten mit, die *Base Activity Library* (*BAL*). Sie sind keine Voraussetzung für Workflows. Wenn Sie wollten, könnten Sie darauf verzichten und alle Aktivitäten selbst entwickeln. Aber natürlich ist die BAL äußerst nützlich; wer möchte beispielsweise schon eine `for-each`-Schleife selbst entwickeln?

Innerer Zustand und Kommunikation mit außen

Ein Workflow kommuniziert mit der Außenwelt über *Argumente*, so jedenfalls nennt die WF die Variablen, die an eine Workflow-Instanz übergeben und an den aufrufenden Prozess zurückgeliefert werden können. Workflows kommen praktisch nie ohne Argumente aus, denn es geht ja gerade darum, aufgrund von Eingabewerten einen bestimmten Weg durch den Workflow zu nehmen.

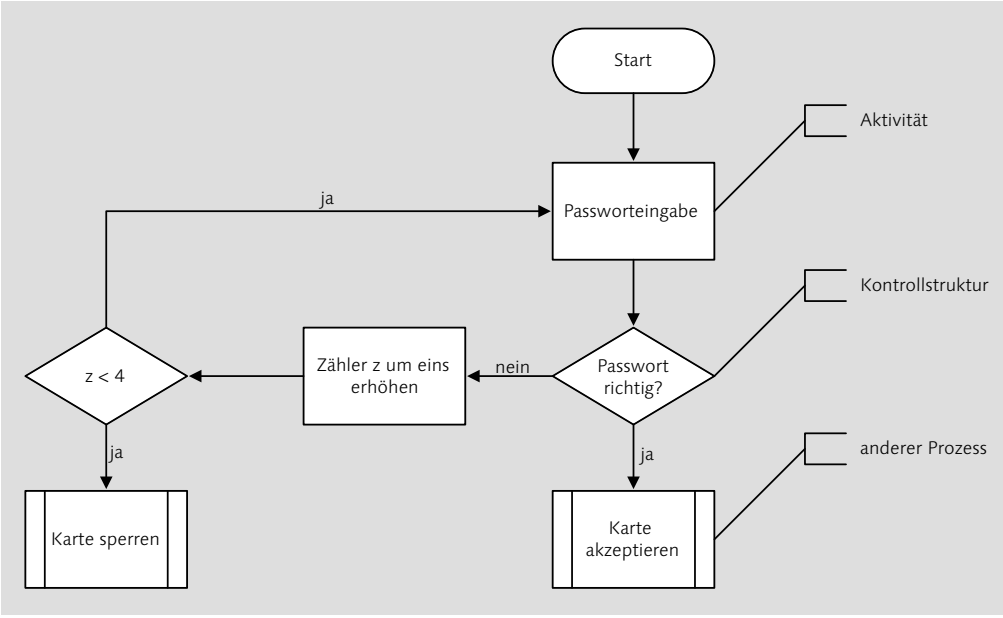


Abbildung 8.2 Ein einfacher Workflow

Was ist, wenn ein Client an den Prozess, in dem der Workflow läuft, überhaupt nicht herankommt? Das ist beispielsweise dann der Fall, wenn ein Workflow innerhalb eines Serverprozesses gehostet wird oder selbst einen WCF-Service bildet (der auf einem anderen Rechner laufen kann). Für solche Fälle kommuniziert der Workflow mittels Messaging-Aktivitäten mit der Außenwelt.

Variablen gibt es ebenfalls in WF. Im Gegensatz zu Argumenten werden sie intern verwendet, um den inneren Status eines Workflows zu speichern, beispielsweise eine getroffene Entscheidung in einer `Boolean`-Variablen. Variablen haben auch in der WF einen Gültigkeitsbereich, können also für jede Aktivität definiert werden und sind dann nur für diese Aktivität (und alle Kind-Aktivitäten) sichtbar.

Workflow-Arten

Bisher haben wir immer von *dem* Workflow gesprochen, näher betrachtet gibt es jedoch drei Arten von Workflows:

- *State Machine Workflows* gibt es seit WF 4.5 erstmals wieder, denn es gab sie vor der Version 4.0 schon einmal. State Machines sind eine besonders ausdrucksstarke Form, Prozesse abzubilden, elegant noch obendrein. Im Kern geht es darum, dass ein Workflow zu einer Zeit immer in einem gewissen Zustand ist und, je nach Ereignis, zwischen diesen Zuständen wechseln kann. Nehmen Sie einmal eine eingehende Bewerbung, die sich im Zustand »Eingegangen« befindet. Viel-

leicht wird der Bewerber eingeladen, vielleicht auf eine Warteliste gesetzt oder es wird ihm abgesagt. Alles Zustände, deren Wechsel Aktivitäten verursachen, beispielsweise eine E-Mail mit der Einladung, oder die Reservierung eines Termins im Kalender.

- *Sequenzielle* Workflows sind Workflows, die innerhalb einer *Sequence*-Aktivität ausgeführt werden. Ihre Aktivitäten laufen schrittweise ab, immer der Reihe nach, immer in dieselbe Richtung. Das macht sie berechenbar und einfach zu modellieren. Leider werden sie aber schnell komplex und unübersichtlich, weswegen sie sich eher für Aufgabenstellungen kleinerer und mittlerer Komplexität eignen. Außerdem liegt es nicht jedem Anwender, Prozesse so abzubilden, dass sie diese Restriktionen einhalten. Diese Art Workflows erinnert ein wenig an Struktogramme.
- *Flowchart*-Workflows bilden für die meisten Menschen Prozesse intuitiver ab, denn sie erlauben Rücksprünge innerhalb eines Workflows. Sie lassen sich aber auch mit sequenziellen Aktivitäten kombinieren und erinnern eher an Programmablaufpläne. Hier eine kleine Entscheidungshilfe, wann Sie welches der beiden Workflow-Modelle von WF 4.0 einsetzen sollten:

Sequence	Flowchart	State Machine
Dieser Typ eignet sich gut für einfache Prozesse, die linear ablaufen.	Damit können komplexere Workflows umgesetzt werden, die im Kern zwar auch linear sind, aber mit beliebigen Verzweigungen und Sprüngen innerhalb des Workflows.	Dieser Typ eignet sich gut für komplexere Prozesse, vor allem, wenn viele Bedingungen vorliegen, die den Status eines Workflows verändern.
Entwickler verstehen diesen Typ gut, da er ein wenig an Struktogramme erinnert.	Anwender verstehen diesen Typ gut, da ihnen Flowcharts auf Papier häufig begegnen.	State Machine Workflows erfordern ein wenig mehr Abstraktionsvermögen, sind aber vor allem häufig kürzer darzustellen.
Der Kontrollfluss ergibt sich aus den Kontrollstruktur-Aktivitäten (z. B. *If*), das ist weniger übersichtlich.	Der Kontrollfluss ergibt sich aus den Knoten und den Verzweigungen von dort aus, das ist (häufig) übersichtlicher.	Der Kontrollfluss ergibt sich aus den Ereignissen, die eine Änderung des Status bewirken, und aus den Aktivitäten entlang diesen Änderungen.

Tabelle 8.1 Auswahlhilfe für Workflow-Modelle

Sequence	Flowchart	State Machine
Ein Rücksprung ist nicht möglich.	Ein Rücksprung ist möglich.	Rücksprünge sind möglich, Dieselben Status können im Verarbeitungszyklus also mehrfach vorkommen.
Der Typ ist weniger gut geeignet, wenn menschliche Entscheidungen Teil des Workflows sind.	Er ist gut geeignet bei manuellen Prozessen.	Eignet sich, wenn man die Status gut benennen kann und klar ist, welche Ereignisse welche Auswirkungen auf den aktuellen Status haben.

Tabelle 8.1 Auswahlhilfe für Workflow-Modelle (Forts.)

8.1.3 Workflow Designer

Das augenfälligste Werkzeug in WF ist sicherlich der Workflow Designer in Visual Studio 2012, der für die neue Version abermals deutlich verbessert wurde und sich nun vor allem für größere Workflows deutlich besser eignet.

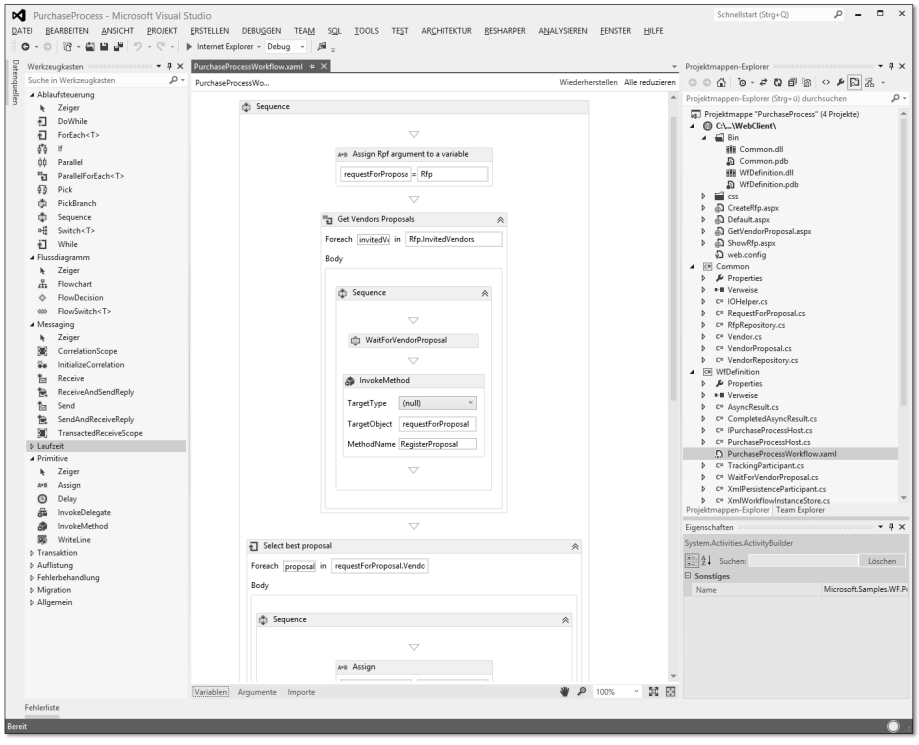


Abbildung 8.3 Workflow Designer

Der Designer kann aber nicht nur in Visual Studio verwendet werden, Sie können ihn auch in Ihre eigenen Anwendungen einbauen, was auch als *Designer Rehosting* bezeichnet wird. Damit können selbst Endanwender Workflows erstellen oder verändern, wobei Sie die Verwendung frei eingrenzen können. Dieses *Rehosting* des Designers war in WF 3.5 unnötig komplex, seit WF 4.0 sind dafür nur wenige Zeilen Code nötig. Allerdings stehen dann nicht alle Funktionen zur Verfügung.

So mancher Entwickler mag sich in der Version 4.0 der Workflow Foundation verwundert die Augen gerieben haben, dass Expressions dort nur in Visual-Basic-Notation möglich waren. Der oft gehörten Argumentation, »Power User« würden sich in VB heimisch fühlen, weil sie ja auch in Excel & Co. mit Visual Basic for Applications (VBA) umgehen würden, konnten sich viele nicht anschließen. Umso erfreulicher, dass in der Version 4.5 C#-Expressions möglich sind. Aber natürlich nicht nur.

Doch nun zu Werke: Lassen Sie uns eine kleine Tour de Force durch den Editor wagen.

Grafischer Designer

Workflows können recht schnell einen ansehnlichen Umfang erreichen. Gut, wenn der Editor dabei mehrere Werkzeuge für die Navigation bietet. Hier hat die aktuelle Version einiges dazugelernt.

Zunächst können Sie den gesamten Workflow bzw. alle Aktivitäten eines Workflows zusammenklappen.

Abbildung 8.4 Aktivitäten aus- bzw. einklappen

Die beiden Links erscheinen abwechselnd, Sie können also entweder alle Aktivitäten einklappen, alle Aktivitäten ausklappen, oder auf die vorherige Darstellung zurückspringen (Wiederherstellen).

Was für alle Aktivitäten gilt, geht natürlich auch für einzelne Aktivitäten, über den Pfeil rechts oben, der zwischen zu- und aufgeklappt hin- und herspringt:

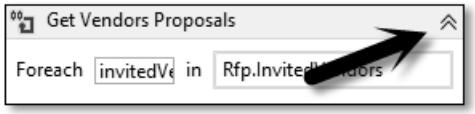


Abbildung 8.5 Eine einzelne Aktivität auf- oder zuklappen

Die Fußleiste beheimatet fünf Werkzeuge:

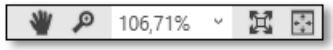


Abbildung 8.6 Die Werkzeuge der Fußleiste

- Das Handwerkzeug verschiebt den angezeigten Ausschnitt des Workflows. Sie deaktivieren diese Funktion, indem Sie erneut auf das Symbol klicken.
- Die Lupe ändert die Anzeige auf 100 %, was (zumeist) noch eine gute Lesbarkeit gewährleistet.
- Alternativ können Sie die Zoomstufe auch direkt eingeben, oder (über den Dropdown-Button des Eingabefelds) aus einigen voreingestellten Zoomstufen auswählen.
- Das nächste Symbol passt die Ansicht so an, dass der gesamte Workflow auf den Bildschirm passt. Das taugt aber eigentlich mehr zum Angeben als zum praktischen Arbeiten, es sei denn Ihre Workflows sind sehr klein, oder Ihr Bildschirm ist wandfüllend groß.
- Das letzte Symbol öffnet die Übersicht, die den dargestellten Ausschnitt gelb einrahmt. Sie können diesen Ausschnitt dort verschieben. Sie kommen so schnell von einem Ende des Workflows zum anderen.

Darüber hinaus funktionieren auch die Standards, wie das Browsen über das Mausrad und das Zoomen über die gedrückte Strg-Taste in Verbindung mit dem Mausrad.

Gliederungsansicht

Bei allem Komfort: Größeren Workflows lässt sich so nur mit Mühe beikommen. Das hat auch Microsoft erkannt und der WF 4.5 eine neue Gliederungsansicht spendiert, die wirklich Gold wert ist. Sie erreichen sie über Ansicht • Weitere Fenster • Dokumentengliederung oder, kürzer, über Strg+W, U. Wenn Sie noch nicht überzeugt sind, dann werfen Sie einmal einen Blick auf die Gegenüberstellung (siehe Abbildung 8.7).

Diese neue Ansicht eignet sich besonders, um die Hierarchie eines Workflows anzuzeigen, also beispielsweise um zu sehen, wie die Sequenzen ineinander verschachtelt sind. Außerdem erkennen Sie bereits an den Symbolen neben dem Text, um welche Art von Aktivität es sich handelt – dies ist sehr praktisch.

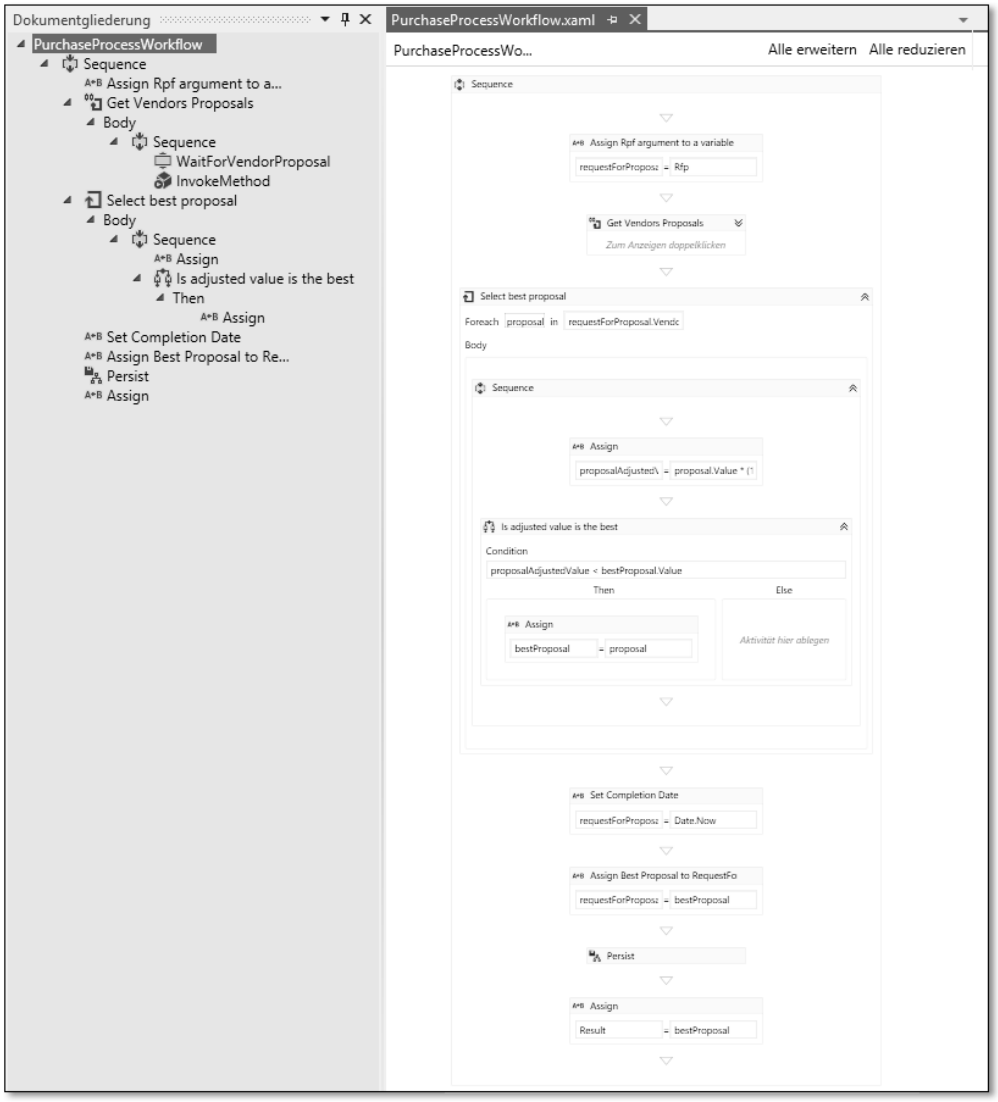


Abbildung 8.7 Vergleich Gliederungsansicht/Grafischer Designer

Suchen und Finden

... aber nicht ersetzen, könnte man ein weiteres neues Feature titulieren, das Ihnen im Designer-Rehosting jedoch nicht zur Verfügung steht. Mit Strg+F können Sie nun das altbekannte Suchfenster nutzen, um Workflow-Aktivitäten zu suchen. Dasselbe funktioniert für die Suche in Dateien, die Sie mit Strg+⇧+F aufrufen.

In beiden Fällen erreichen Sie so nicht nur Aktivitäten, sondern auch andere Elemente, wie Variablen oder Argumente. Das zählt zu den Features, bei denen man sich fragt: Warum hat das so lange gedauert?

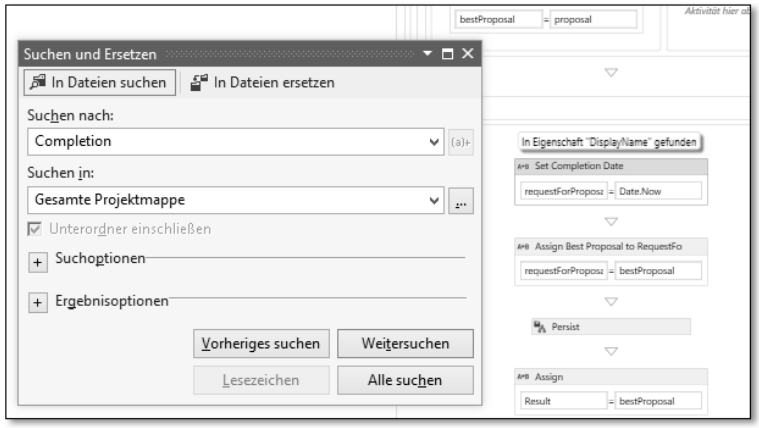


Abbildung 8.8 Schnellsuche nach Aktivitäten

Selektieren

Das Selektieren geht so vonstatten, wie Sie es wohl erwarten würden, oder sagen wir nahezu. Denn mit gedrückter Strg-Taste können Sie zwar mehrere Aktivitäten markieren, allerdings nur, wenn Sie richtig klicken:

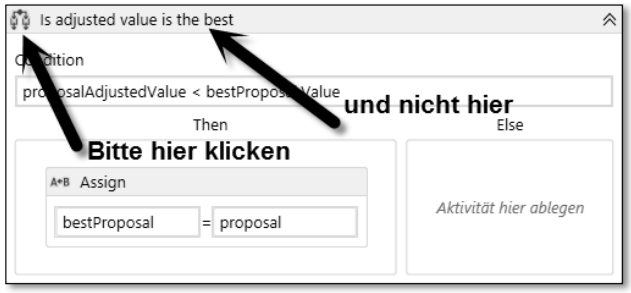


Abbildung 8.9 Die richtige Mehrfachselektion von Aktivitäten

Alternativ, auch das ist neu, können Sie mehrere Aktivitäten auch selektieren, indem Sie einen Rahmen um sie ziehen.

Leider ist die Dokumentengliederung nicht in der Lage, mehr als ein Element (Aktivität) auszuwählen, obwohl auch hier Mehrfachselektionen angezeigt werden. Es ist also noch Raum für Verbesserung vorhanden.

Fokussieren

Manchmal hilft alles nichts: Man muss sich auf einen Teilbereich eines Workflows konzentrieren. Nichts leichter als das, denn wenn Sie auf die Titelzeile einer Aktivität doppelt klicken, machen Sie diese Aktivität (und alle Aktivitäten, die sie vielleicht enthält) zur einzigen Aktivität auf dem Bildschirm, Sie zoomen also in sie hinein.

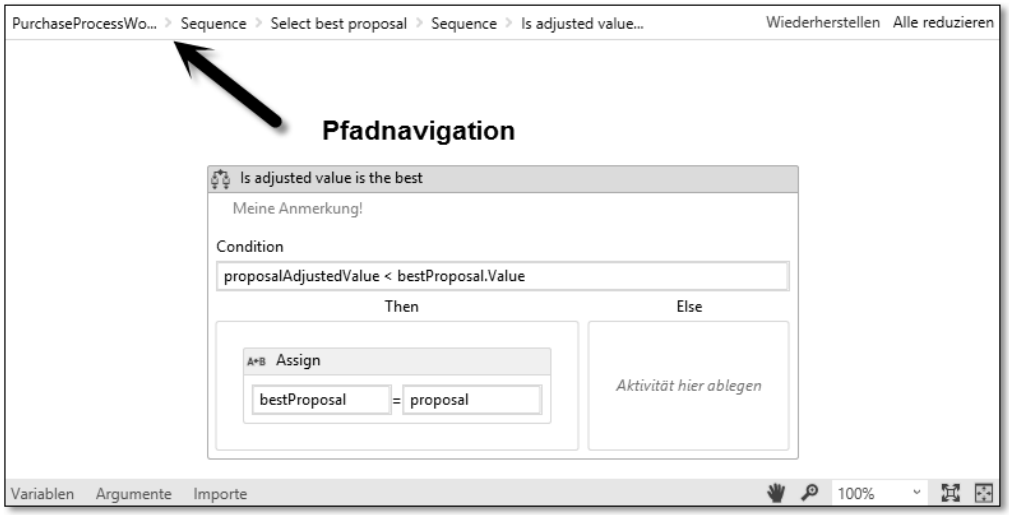


Abbildung 8.10 Hineinzoomen in eine Aktivität

Eine Pfadnavigation, im Englischen besonders liebevoll »Brotkrumennavigation« genannt, ermöglicht es Ihnen, durch einfachen Klick wieder auf eine höhere Ebene zu gelangen, bis hin zum gesamten Workflow.

Anmerkungen

Ein weiteres neues Feature ist eher klein und unscheinbar, aber dennoch äußerst nützlich: die Fähigkeit, einzelne Elemente eines Workflows zu kommentieren. Kommentiert werden können Aktivitäten, aber auch Variablen, Argumente oder Flowchart-Knoten, Konstrukte, auf die ich später noch eingehe.

An dieser Stelle begnüge ich mich mit dem Hinweis, wie es geht, nämlich über das Kontextmenü des zu kommentierenden Elements, z. B. einer Aktivität, genauer: unter ANMERKUNGEN • ANMERKUNG HINZUFÜGEN. So kommentierte Elemente werden mit einem kleinen Symbol gekennzeichnet, das die Anmerkung sichtbar werden lässt; streichen Sie mit der Maus darüber:

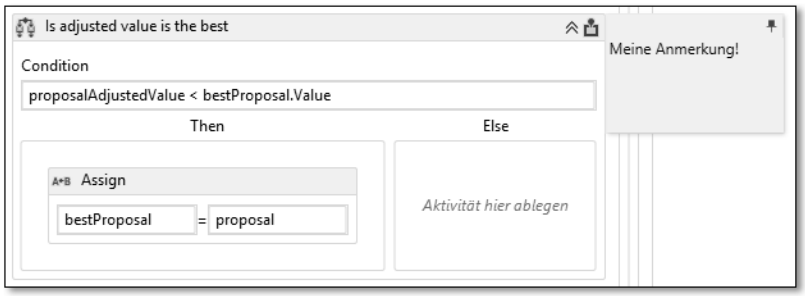


Abbildung 8.11 Anmerkungen für eine Aktivität

Im Kontextmenü finden Sie noch die weitere Option Anmerkungen • Alle Anmerkungen anzeigen, die sämtliche Anmerkungen ständig sichtbar einblendet, sowie Optionen zum Löschen oder Bearbeiten bestehender Anmerkungen.

8.1.4 Windows Workflow Foundation im Überblick

In Abschnitt 8.1.2, »Der Workflow«, haben wir gesehen, in welchen Einsatzszenarien Workflows ihre Stärke ausspielen können und wo sie fehl am Platz sind. Mit der Workflow Foundation steht eine Technologie zur Verfügung, die die Erstellung von Workflows einerseits einfach macht, sich aber andererseits auch für fortgeschrittene Szenarien eignet.

Natürlich könnten Sie Workflows auch auf herkömmliche Weise programmieren, wie unzählige Anwendungen beweisen. Aber wie so häufig gilt gerade bei Workflows: Auch aus einfachen Anforderungen ergeben sich schnell komplexe Programme. Mit der WF stellt uns Microsoft so einiges bereit, um diese Komplexität reduzieren. Sehen Sie selbst.

Base Activity Library und Custom Activities

Der Designer beinhaltet die bereits angesprochene Base Activity Library, eine Sammlung grundlegender Aktivitäten. In der Praxis werden Sie häufig um die Erstellung eigener Aktivitäten nicht herumkommen, die Sie dann in der Toolbox sehen und von dort aus verwenden können.

Eigene Aktivitäten sind in WF 4.5 recht einfach zu erstellen. Für viele Zwecke genügt es, einfach eine Klasse abzuleiten, eine einfache abstrakte Methode zu implementieren und das Projekt zu erstellen, wie wir später noch sehen werden.

Laufzeitumgebung für Workflows

Für die Ausführung von Workflows kommt die *Workflow Runtime* zum Einsatz, die wiederum in einem Host-Prozess ausgeführt wird, den Sie bereitstellen. Als Host kommen so unterschiedliche Technologien wie ASP.NET, NT-Services oder der IIS infrage, aber natürlich können Sie Workflows auch in »ganz gewöhnliche« Anwendungen einbinden und damit selbst hosten.

Die WF-Runtime beginnt bei der äußersten Aktivität eines Workflows und führt dann die Aktivitäten der Reihe nach aus. Die Kontrollstrukturen geben dabei den Pfad durch den Workflow vor. Die Laufzeitumgebung stellt außerdem eine Reihe von Infrastruktur-Komponenten bereit.

Unterstützung für asynchrone Kommunikation und Persistenz

Die WF-Runtime kann einen Workflow selbst persistieren und entladen, oder Sie können diese Aufgabe übernehmen, indem Sie die `Persist`-Aktivität aus der BAL ver-

wenden. Analog dazu kann sie auch gespeicherte Workflows wieder reaktivieren und die Abarbeitung wieder aufnehmen. Natürlich muss die Runtime dafür selbst aktiv sein, weswegen in der Praxis häufiger eine Kombination aus IIS, WAS, WCF und WF eingesetzt wird (siehe Abbildung 8.12).

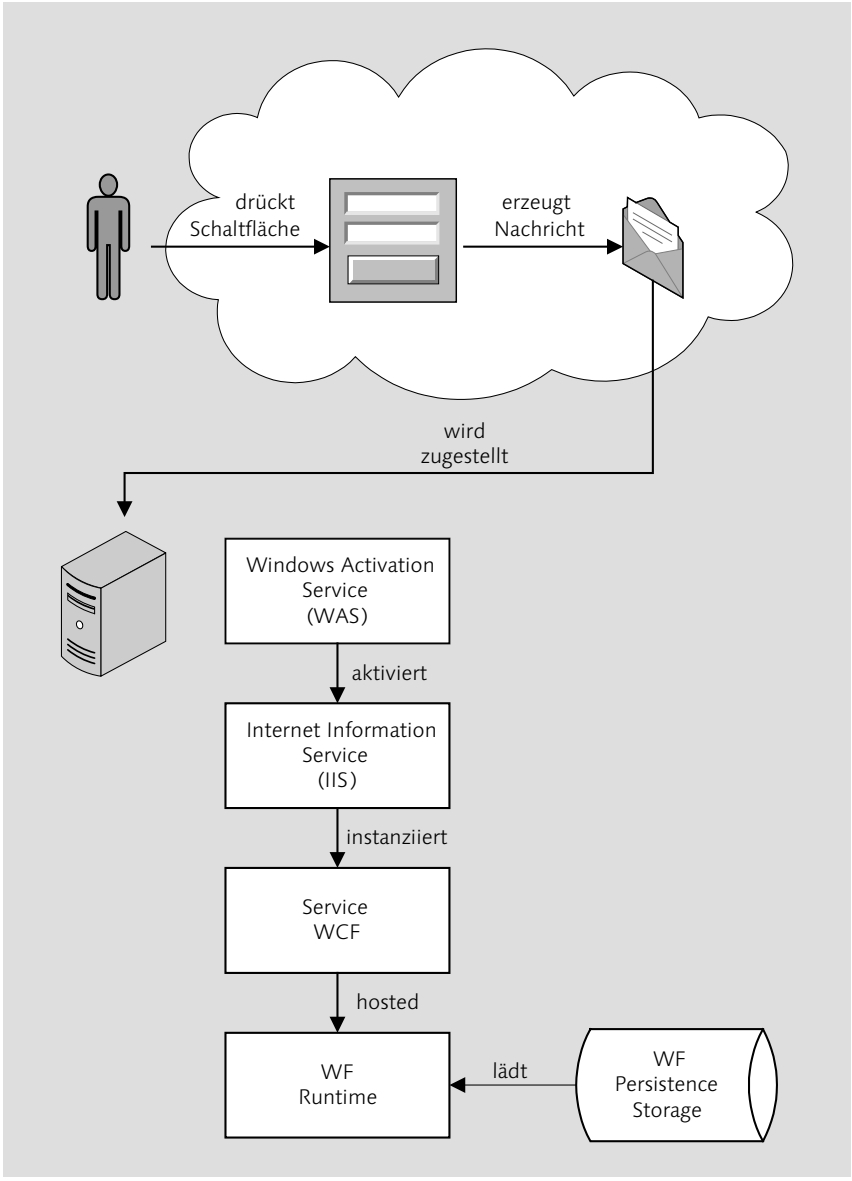


Abbildung 8.12 Reaktivieren eines Workflows aufgrund einer eingehenden Nachricht

Nehmen wir einmal an, ein Vertriebsmitarbeiter hätte einen Urlaubsantrag an den Vertriebsleiter in der Zentrale der Kalimba Sunfood GmbH gestellt. Der Workflow

wurde persistiert und entladen und wartet nun auf die Antwort des Personalleiters, bevorzugt dessen Genehmigung.

Der Vertriebsleiter startet seine Webanwendung und drückt die Schaltfläche GENEHMIGUNG. Der dahinter stehende WCF-Service versendet eine Nachricht an das IT-System der Niederlassung. Dort wird die Nachricht vom *Windows Activation Service* (*WAS*) entgegengenommen, einem Bestandteil des IIS. Der Workflow Service wird nun instanziiert, der für die Urlaubsanträge zuständig ist. Die WF-Runtime lädt nun den Workflow aus der Datenbank und führt ihn dort fort, wo er zuvor beendet wurde. Der Urlaub ist genehmigt.

Integration mit WCF

Wir haben es gerade gesehen: WCF und WF sind ein Duo, das sich gut ergänzt. Die Integration beider Technologien wurde in WF 4.0 deutlich verbessert und vereinfacht. Dabei sind mehrere Szenarien denkbar; die wichtigsten sind:

- Ein Workflow ruft Methoden eines WCF- oder (z. B. Java-)Webservices auf, tritt also als Client auf.
- Ein Workflow stellt selbst die eigene Funktionalität als Service zur Verfügung, tritt also als Server auf und kann somit Anfragen beantworten. Solche Services nennt die WF *Workflow Services*. Sie lassen sich nur per Konfiguration erstellen und beispielsweise auf dem IIS hosten. Wie immer gilt aber: Wer mehr Kontrolle über den Prozess haben möchte, kann die Konfiguration beeinflussen oder im Code Einfluss auf die Laufzeitumgebung nehmen.

Für die Kopplung von WCF und WF gibt es in WF 4.5 einen eigenen Projekttyp und in Abschnitt 8.11.3, »Fallbeispiel – Teil 1: Der Laborservice«, ein ausführliches Beispiel.

XAML

Workflows speichert WF in *Language* XAML-Dateien. *XAML* (*Extensible Application Markup*) ist eine von Microsoft entwickelte allgemeingültige Beschreibungssprache für Oberflächenelemente. Sie wird Ihnen schon begegnet sein, wenn Sie bereits mit WPF oder Silverlight entwickelt haben, da sie dort die Benutzeroberfläche und das Verhalten der Anwendung beschreibt.

Wenn Sie möchten, dann ist diese XML-Sprache so etwas wie der gemeinsame Nenner zwischen Entwickler und Designer, denn beide können zunächst unabhängig voneinander arbeiten. Für den Designer steht mit *Expressions Blend* sogar ein eigenes Produkt zur Verfügung, das in seiner Bedienung eher an ein Grafikprogramm erinnert als an eine Programmierumgebung. Später werden Code und XAML dann zusammengeführt und bilden damit, zur Laufzeit, das fertige Programm.

Workflows können Sie also auch von Hand bearbeiten, direkt im XAML-Code. Oder Sie verwenden den Designer, ganz wie Sie wünschen. Sie können den Code zur Laufzeit aus einer Datei oder einer anderen Quelle laden. Der Code, aus dem Aktivitäten gemacht sind, bleibt aber weiterhin Teil einer Assembly, die Sie mit ausliefern möchten, wenn Sie sich nicht auf die Standardaktivitäten beschränken wollen. Dieses Verfahren ist eigentlich viel konsequenter, als beispielsweise Code-Behind-Dateien einzusetzen, denn es erlaubt eine eindeutige und leicht nachvollziehbare Trennung zwischen Konfiguration und Programmierung.

8.2 Fallbeispiel

Gerade für die Workflow Foundation gilt: Das Ganze ist mehr als die Summe seiner Teile. Ihre Stärke zeigt sich daher vor allem im Zusammenspiel der einzelnen Komponenten. Dieses Buch möchte Ihnen keine isolierten Funktionen erklären, sondern Sie in die Lage versetzen, die Workflow Foundation in eigenen Projekten einzusetzen, in Projekten jenseits von *Hello Word*. Andererseits ist der Platz in diesem Buch beschränkt, da hilft auch alles Betteln der Autoren nichts.

Ich habe mich daher entschieden, einen großen Teil der Funktionalität der WF in ein einziges Fallbeispiel zu integrieren, das wir gemeinsam Schritt für Schritt aufbauen werden. Sie benötigen dafür außer einem SQL Server (Express genügt) keine weiteren Voraussetzungen, das Fallbeispiel ist eigenständig lauffähig und daher recht umfangreich. Aber keine Sorge: Sie können an verschiedenen Stellen einsteigen, indem Sie einfach die Projektmappe des jeweiligen Abschnittes laden.

Sie werden belohnt durch eine mehrschichtige, asynchrone Anwendung, die einen großen Teil der Möglichkeiten abdeckt. Entgegen dem allgemeinen Trend werden wir weder Taschenrechner noch Auftragsbearbeitungen programmieren. Stattdessen werden wir Wareneingang und Labor mit neuer Software ausstatten und einen Prozess zwischen den beiden Abteilungen automatisieren. Die genauen Anforderungen beschreibe ich im jeweiligen Abschnitt, hier erläutere ich einige grundlegende Informationen.

Die Warenannahme des Zentrallagers der Kalimba Sunfood GmbH erhält Produkte aus der ganzen Welt, um sie zwischenzulagern und an den Großhandel zu vertreiben. Wenn eine Ware eintrifft, dann durchläuft sie einen ganz bestimmten Workflow.

1. Für jeden Wareneingang wird geprüft, ob es auch eine zugehörige Bestellung gibt. Ohne Bestellung wird die Ware nicht angenommen und landet auf einem speziellen Abstellplatz. Der Lagerleiter entscheidet dann in Zusammenarbeit mit dem Einkaufsleiter über das weitere Schicksal der Ware.

2. Danach wird die Ware mengenmäßig kontrolliert, meistens durch eine automatische Wiegeapparatur. Stimmt die Menge, ist alles in Ordnung, wenn nicht, dann entscheidet das Maß der Abweichung über die weitere Vorgehensweise.
3. Jede Lieferung wird auf Sicht und im Labor kontrolliert. Dabei gelten verschiedene Vorschriften für die verschiedenen Warenkategorien, danach richtet sich vor allem die Größe der zu ziehenden Stichprobe. Es hängt vom Laborergebnis ab, ob die Ware weiterverkauft werden darf oder nicht.
4. Danach wird die Ware eingelagert, wobei für jede Palette ein Code generiert wird, der gut sichtbar angebracht wird.
5. Zum Schluss wird der Lagerleiter über die Einlagerung informiert und der Lagerbestand erhöht.

Unsere Aufgabe wird es nun sein, eine Anwendung zu entwickeln, die den Wareneingang und das Labor unterstützt, natürlich auf Basis der Workflow Foundation.

8.3 Der erste sequenzielle Workflow

Grau ist alle Theorie, doch Workflows sind bunt! Beginnen wir also mit unserem ersten Workflow. Bevor es losgeht, noch zwei Hinweise:

Gewöhnen Sie sich am besten gleich an, alle Aktivitäten sinnvoll zu benennen. Einerseits werden Ihre Workflows dadurch viel leichter lesbar, andererseits werden die spätere Fehlersuche und die Ablaufverfolgung dadurch intuitiver. Die Benennung erfolgt durch die Eigenschaft `DisplayName`.

Auch kleine Workflows werden schnell unübersichtlich. Wenn Sie also nicht gerade in den neuesten WQUXGA-Monitor mit 50"-Bildschirmdiagonale investieren möchten, beachten Sie bitte die Hinweise im Abschnitt 8.1.3, »Workflow Designer«, und üben Sie ein wenig.

8.3.1 Das Projekt einrichten

Legen Sie zunächst unter Datei • Neu • Projekt ein neues Workflow-Projekt an. Visual Studio 2012 bietet Ihnen hierfür vier Projektvorlagen zur Auswahl an.

Wählen Sie Konsolenanwendung für Workflows aus und benennen Sie das Projekt *Wareneingang*. Dabei erledigt Visual Studio die folgenden Dinge für uns:

- Es fügt die entsprechenden Verweise hinzu, allen voran `System.Activities`.
- Es erstellt eine *App.config*-Datei, in der als Laufzeitvoraussetzung das .NET Framework 4.5 angegeben ist.
- Es erstellt einen leeren Workflow, *Workflow1* genannt.

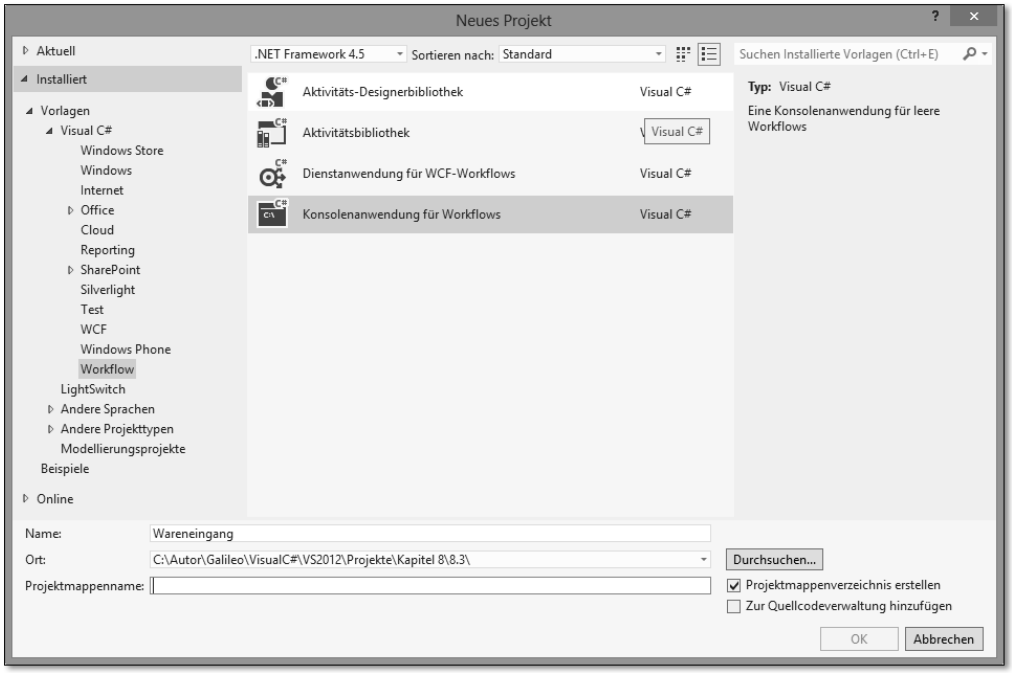


Abbildung 8.13 Ein neues Workflow-Projekt anlegen

Zudem erstellt es den Startpunkt der Anwendung, nämlich die Klasse `Program` für uns. Der Inhalt ist wenig geheimnisvoll.

```
using System;
using System.Linq;
using System.Activities;
using System.Activities.Statements;

namespace Wareneingang
{

    class Program
    {
        static void Main(string[] args)
        {
            Activity workflow1 = new Workflow1();
            WorkflowInvoker.Invoke(workflow1);
        }
    }
}
```

Listing 8.1 Der Inhalt der Datei program.cs

In ihr wird lediglich eine neue Instanz des (noch leeren) Workflows `Workflow1` erzeugt und der Workflow Runtime übergeben. Damit wir später Ausgaben auf der Konsole noch lesen können, bevor die Anwendung geschlossen wird, fügen Sie bitte unmittelbar danach die folgende Zeile hinzu:

```
Console.ReadLine();
```

Wie ich schon erwähnt habe, werden Workflows in XML-Dateien mit der Endung *.xaml* gespeichert, so auch unser Workflow.

Sie können den Code auch ansehen, indem Sie im Kontextmenü der Datei *Workflow1.xaml* den Menüpunkt CODE ANZEIGEN auswählen. Wenn Sie schon einmal dabei sind, benennen Sie den Workflow doch gleich in `WorkflowWareneingang` um:

```
<Activity x:Class="Wareneingang.WorkflowWareneingang"
```

Benennen Sie nun noch den Dateinamen in *WorkflowWareneingang.xaml* um. Die XAML-Datei ist zum jetzigen Zeitpunkt noch gänzlich uninteressant und enthält lediglich einige Namespace-Definitionen.

8.3.2 Den Workflow gestalten

Sie können jetzt den Workflow im Designer öffnen, entweder durch einen Doppelklick auf die XAML-Datei oder durch Drücken von ⇧+F7. Der Workflow ist leer, aber das wird sich nun ändern. Ziehen Sie aus der Toolbox (also dem »Werkzeugkasten«) eine *Sequence* (im Karteireiter ABLAUFSTEUERUNG) auf die Aufforderung AKTIVITÄT HIER ABLEGEN. Innerhalb einer Sequence werden alle Aktivitäten sequenziell, also nacheinander, ausgeführt. Benennen Sie die Sequence in *Wareneingangsprüfung* um, indem Sie die `DisplayName`-Eigenschaft setzen oder, das geht schneller, in den Kopfbereich der Aktivität klicken und zu tippen beginnen.

Wie Sie in Abschnitt 8.2, »Fallbeispiel«, erfahren haben, besteht die Wareneingangsprüfung bei der Kalimba Sunfood GmbH aus den folgenden Phasen, die nacheinander durchlaufen werden:

1. Bestellprüfung
2. Mengenprüfung
3. Laborprüfung
4. Einlagerung

Fügen Sie nun für jeden der Bereiche eine eigene Sequence ein und beachten Sie dabei die Reihenfolge. Benennen Sie sie jeweils nach den Phasen. Visual Studio zeigt Ihnen mithilfe von Pfeilen an, wo Sie eine Aktivität ablegen können (siehe Abbildung 8.14).

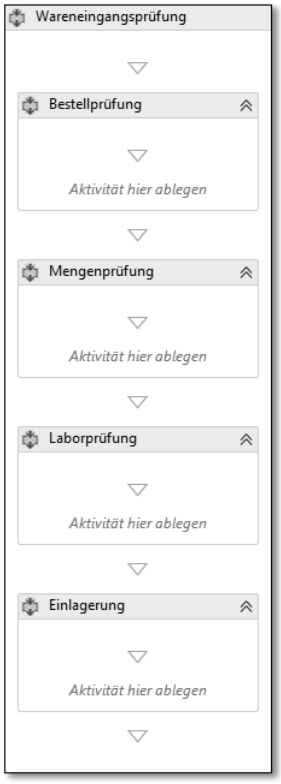


Abbildung 8.14 Der Workflow bisher

Damit haben wir das Grundgerüst geschaffen, das wir im Laufe dieses Kapitels ausfüllen werden. Für unseren ersten Workflow modellieren wir die Sequence *Laborprüfung*.

Ein Workflow wäre recht sinnlos, wenn er nicht Parameter entgegennehmen, verarbeiten und zurückgeben könnte. Für unsere Laborprüfung benötigen wir einige Angaben von außen und zwar

- die Menge, die angeliefert wurde,
- die Art der Ware, denn danach richtet sich die Stichprobe und der weitere Verlauf des Workflows,
- eine Referenznummer, damit wir den Wareneingang später mit der Bestellung abgleichen können.

Am unteren Rand des Workflow Designers gibt es dafür die Schaltfläche ARGUMENTE. Dort können wir diese Angaben nun definieren.

1. Klicken Sie auf ARGUMENTE.
2. Erstellen Sie die folgenden Argumente:

Name	Richtung	Argumenttyp
Menge	*Ein*	Int32
Bestellkennzeichen	*Ein*	String

3. Für die Art der Ware bietet sich eine Enumeration (enum) an, die wir zuvor im Code anlegen müssen. Erstellen Sie dafür ein eigenes Projekt *WareneingangLibrary*, das wir später noch für andere Zwecke benötigen werden, und wählen Sie als Projekttyp KLASSENBIBLIOTHEK. Wir werden dieses Projekt später noch um weitere Klassen erweitern. Fügen Sie dieses neu erstellte Projekt unserem Workflow-Projekt als Projektreferenz hinzu, sodass Sie auf den Typ innerhalb des Workflows zugreifen können.
4. Erstellen Sie nun eine Enumeration innerhalb des Projekts *WareneingangLibrary*.

```
public enum Warenart
{
   Obst,
   Fertigprodukt,
   Grundstoff
}
```

Listing 8.2 Warenart.cs

5. Sie sollten nun das Projekt erstellen, damit der Workflow Designer den Typ kennt. Überhaupt sollten Sie sich gleich zu Beginn angewöhnen, das Projekt sofort zu erstellen, wenn Ihnen etwas merkwürdig vorkommt. Viele Probleme lösen sich dadurch. Dies ist ein Zugeständnis an die Flexibilität, einerseits Typen in Konfiguration zu speichern und sie andererseits wie bei einem Workflow im Code ansprechen zu können.
6. Fügen Sie dem Workflow nun zwei weitere Argumente hinzu. Für das erste Argument wählen Sie bitte als Argumenttyp NACH TYPEN SUCHEN... und anschließend den Typ Warenart aus dem Namespace WareneingangLibrary. Es genügt, wenn Sie die ersten Buchstaben des Typs eingeben, der Designer zeigt Ihnen dann alle Typen an, die mit dem Suchbegriff beginnen.

Name	Richtung	Argumenttyp
AngeliefertesProdukt	*Ein*	Warenart
PruefungErfolgreich	*Aus*	Boolean

Ihre Argumente sollten nun so aussehen wie in Abbildung 8.15 dargestellt.

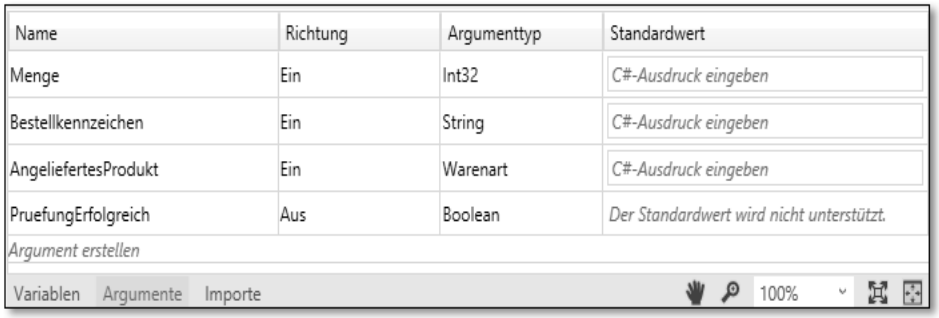


Abbildung 8.15 Die Argumente für unseren Workflow

Die Liste der Typen ist sicherlich gewöhnungsbedürftig und erscheint willkürlich. Sofern das entsprechende Projekt referenziert ist, können Sie aber über den gezeigten Weg auf nahezu alle Typen in .NET und in Ihrer Anwendung zugreifen. Mit der beschriebenen Suchfunktion kann man sich rasch daran gewöhnen.

Fügen wir nun die Logik der Aktivität *Laborprüfung* hinzu. Klicken Sie dafür doppelt auf diese Aktivität (also auf das Icon), um sie exklusiv im Designer zu öffnen.

1. Ziehen Sie aus dem Register PRIMITIVE die Aktivität *WriteLine* in die Aktivität *Laborprüfung*. Damit lassen sich einerseits Meldungen auf der Konsole ausgeben und andererseits Texte in Dateien schreiben.
2. Benennen Sie die Aktivität in *Start Laborprüfung* um und setzen Sie die Eigenschaft *Text* auf »Start der Laborprüfung«. Vergessen Sie nicht die Anführungszeichen, denn es soll kein Text eingegeben werden, sondern ein C#-Ausdruck. Im Falle einer Zeichenkette schließt das die Anführungszeichen mit ein. Welchen Typ eine Expression erwartet, sehen Sie, wenn Sie den zugehörigen Dialog öffnen (über den ...-Button). Sie müssen in Ihrer Expression dann ein Objekt dieses Typs zurückgeben. Im Falle der *WriteLine*-Aktivität ist das ein einfacher String. Haben Sie bemerkt, dass Visual Studio Ihre Eingabe schon während des Tippens auf syntaktische Richtigkeit hin überprüft hat?
3. Fügen Sie anschließend unterhalb der gerade erstellten *WriteLine*-Aktivität eine Sequenz ein, die Sie *Stichprobenauswahl* nennen.
4. Um die Größe der Stichprobe zu speichern, benötigen wir eine Variable. Variablen fügen Sie auf dieselbe Art und Weise hinzu wie Argumente, aber unter dem Register VARIABLEN. Benennen Sie die Variable `Stichprobe` und geben Sie als Typ `Int32` an. Variablen haben auch in WF einen Gültigkeitsbereich. Die Variable `Stichprobe` ist nur innerhalb der Aktivität *Laborprüfung* interessant, wählen Sie daher diesen Eintrag aus der Liste aus. Die Variablen sind auch in untergeordneten Aktivitäten sichtbar, ganz so wie in den C#-Scopes.

5. Die Größe der Stichprobe hängt von der Warenart ab, es werden 5 % der angelieferten Obstmenge kontrolliert, aber nur 2 % der angelieferten Fertigprodukte. Dafür können Sie der Sequence *Stichprobenauswahl* eine *Switch-<T>*-Aktivität hinzufügen. Als Typ wählen Sie bitte `WareneingangLibrary.Warenart`, den Dialog dafür kennen Sie ja inzwischen bereits. Als Ausdruck geben Sie bitte das Argument *AngeliefertesProdukt* an, denn abhängig vom Inhalt dieses Arguments soll sich die Anweisung verzweigen.
6. Klicken Sie nun auf NEUEN FALL HINZUFÜGEN, und zwar dreimal, einmal für *Obst*, einmal für *Fertigprodukt* und einmal für *Grundstoff*. Die drei Werte können Sie bequem aus der Dropdown-Liste auswählen. Visual Studio erstellt automatisch einen Default-Zweig für die *Switch*-Anweisung, den wir aber nicht benötigen, da wir alle drei möglichen Fälle explizit behandeln.
7. Die Aktivität *Assign* ist dafür zuständig, der Variablen `Stichprobe` einen Wert zuzuweisen. Ziehen Sie eine solche Aktivität daher nun in den Zweig *Obst,* den Sie dafür vorher durch Anklicken öffnen müssen. Geben Sie im linken Eingabefeld `Stichprobe` ein, und schreiben Sie in das rechte Feld den Ausdruck `Convert.ToInt32(Menge * 0.05)`. Verfahren Sie für den Zweig *Fertigprodukt* analog, dort ist der *Multiplikator* `0.02` (2 %) sowie für *Grundstoff* `0.01` (1 %).
8. Legen Sie nun eine weitere *WriteLine*-Aktivität unterhalb der *Switch*-Aktivität ab, die uns die so ermittelte Stichprobengröße ausgibt. Als Ausdruck geben Sie bitte ein: `"Stichprobengröße: "+Stichprobe.ToString()`.

Damit haben wir schon eine Menge erreicht. Wir haben das Grundgerüst erstellt und die Stichprobengröße in Abhängigkeit von der Warenart ermittelt. Bereits jetzt ist die Aktivität *Laborprüfung* so groß, dass sie kaum mehr auf eine Seite passt, wie Abbildung 8.16 zeigt.

Bevor Sie das Projekt starten können, müssen wir noch die Datei *Program.cs* anpassen, denn wir hatten ja zuvor den Namen des Workflows geändert:

```
Activity workflow1 = new WorkflowWareneingang();
WorkflowInvoker.Invoke(workflow1);
```

Starten Sie nun das Projekt. Als Größe der Stichprobe wird `0` angezeigt. Das liegt natürlich daran, dass wir zwar Argumente definiert, aber keine Werte zugewiesen haben. Wir müssen also noch die Workflow-Instanz von außen mit Werten füttern.

1. Öffnen Sie hierfür abermals die Datei *Program.cs*, in der die Workflow-Instanz erzeugt und der Workflow gestartet werden.
2. Importieren Sie anschließend den Namespace `System.Collections.Generic`, sowie `WareneingangLibrary` (für die Warenart).

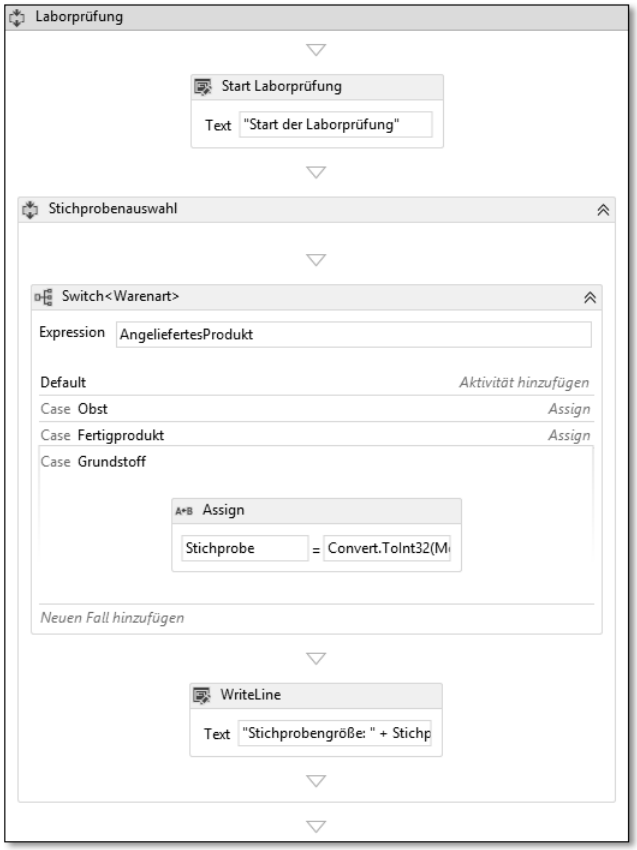


Abbildung 8.16 Die Ermittlung der Stichprobengröße für die Laborprüfung

3. Argumente werden mit einer Variablen vom Typ `Dictionary<string, object>` übergeben. Dort bricht WF mit der Typsicherheit, wie Sie an `object` erkennen. Sie müssen also selbst darauf achten, die richtigen Typen den richtigen Argumenten zuzuweisen. Modifizieren Sie den Code wie folgt:

```
Dictionary<string, object> Arguments = new Dictionary<string, object>();
Arguments.Add("Menge", 400);
Arguments.Add("Bestellkennzeichen", "B1456");
Arguments.Add("AngeliefertesProdukt", Warenart.Obst);
Activity workflow = new WorkflowWareneingang();
WorkflowInvoker.Invoke(workflow, Arguments);
Console.ReadLine();
```

Listing 8.3 Die modifizierte Program.cs-Datei mit der Übergabe von Argumenten an die Workflow-Instanz

Das soll für unsere Zwecke genügen. In der Praxis wollen Sie vielleicht eine Ableitung der `Dictionary`-Klasse verwenden, in der Sie für jedes Argument ein Property anlegen, um die Typsicherheit wiederherzustellen. Starten Sie nun Ihren Workflow.

Abbildung 8.17 Ausgabe der nun fertiggestellten Stichprobenauswahl

Wenn Sie nun `Warenart.Fertigprodukt` als `AngeliefertesProdukt` übergeben, dann errechnet der Workflow eine Stichprobengröße von 8, ganz so, wie es sein soll. Damit ist unsere Aufgabe aber noch nicht ganz erfüllt, wir müssen die Stichprobe nun noch an das Labor schicken und das Ergebnis auswerten. Wechseln Sie hierzu in die Aktivität *Laborprüfung*.

1. Fügen Sie nach *Stichprobenauswahl* eine weitere Sequence hinzu und benennen sie *Stichprobenauswertung*.
2. Dorthinein platzieren Sie eine *ForEach<T>*-Aktivität und geben als Typ `Int32` an, was schon vorausgewählt ist.
3. Damit wir mittels `foreach` über eine Menge von Integer-Werten iterieren können, hilft uns `Enumerable`. Geben Sie als Schleifenvariable `probe` an und als Ausdruck `Enumerable.Range(1, Stichprobe)`. Der Schleifenkörper wird nun so oft durchlaufen, wie Stichproben-Elemente vorhanden sind.
4. Nun wäre der richtige Zeitpunkt gekommen, um eine höchst raffinierte Laborapparatur an den Workflow anzubinden, die Stichprobe dieser Apparatur zu übergeben und das Ergebnis automatisiert auszuwerten. Das klingt kompliziert, wäre in Wirklichkeit aber einfach, denn wir könnten ein synchrones Pattern für den Zugriff verwenden. Die Prüfung bei der Kalimba Sunfood GmbH erfolgt aber höchst manuell, durch Herbert, unseren langjährigen Mitarbeiter im Lebensmittellabor. Hierbei handelt es sich um einen asynchronen Vorgang, denn wir wissen nicht, wann der Mitarbeiter die Prüfungen abgeschlossen hat; wir müssen also darauf warten, bevor der Wareneingangs-Workflow weitergeführt werden kann. Dieser Aufgabe werden wir uns später annehmen. Für den Augenblick definieren wir eine weitere Variable `FehlerhafteProben` im Gültigkeitsbereich `Laborprüfung` und geben als Typ `Int32` an. Diese Variable initialisiert WF mit 0, was für unsere Zwecke sinnvoll ist. Später werden wir die Variable für jede fehlgeschlagene Prüfung um eins erhöhen. Für jetzt lassen wir den Schleifenkörper unserer *ForEach*-Aktivität einfach leer.
5. Wir können dennoch weitermachen, denn am Schluss müssen wir nun auswerten, ob die Laborprüfung erfolgreich verlaufen ist oder nicht. Fügen Sie nach der so-

eben erstellten Aktivität (also der foreach-Aktivität) zur Einreichung der Laborproben daher eine *If*-Aktivität ein. Als fehlerhaft soll gelten, wenn mehr als 10 % der geprüften Proben die Prüfung nicht bestanden hat. In diesem Fall wird die Warenlieferung nicht akzeptiert und an den Lieferanten zurückgesendet. Als *Condition* schreiben Sie daher: `FehlerhafteProben > Convert.ToInt32(Stichprobe * 0.1)`

6. Der *Then*-Zweig wird durchlaufen, wenn die Bedingung zutrifft, wenn also mehr als 10 % der Proben fehlerhaft waren. Ziehen Sie dorthinein eine *Assign*-Aktivität, und weisen Sie der Variablen `PruefungErfolgreich` den Wert `false` zu. Im *Else*-Zweig weisen Sie den Wert `true` zu.
7. Dem Argument `PruefungErfolgreich` hatten wir die Richtung *Aus* zugewiesen. Wir können den Wert nun abgreifen, nachdem der Workflow beendet wurde. Die Ausgabe-Parameter stellt uns die `Invoke`-Methode zur Verfügung, wir müssen, wieder in der Datei *Program.cs,* nur den Aufruf ein wenig ändern:

```
...
IDictionary<string, object> output =
  WorkflowInvoker.Invoke(workflow, Arguments);
Console.WriteLine("Prüfung erfolgreich? "+output["PruefungErfolgreich"]
.ToString());
```

8. Starten Sie nun den Workflow erneut und die Ausgabe sieht so aus wie in Abbildung 8.18 dargestellt.

Abbildung 8.18 Ausgabe des Prüfungsergebnisses

8.3.3 Der weitere Ausbau

Damit haben Sie nun Ihren ersten Workflow erstellt. Herzlichen Glückwunsch! Und dieser Workflow kann schon eine ganze Menge. Vor allem aber ist er erweiterbar und konfigurierbar. Wir könnten ohne großen Aufwand neue Warenarten hinzufügen, die Stichprobengrößen verändern oder die Bedingungen für eine erfolgreiche Prüfung neu festlegen. Es bleiben aber noch viele Wünsche übrig, um nur einige zu nennen:

- Das Labor ist noch nicht angebunden, die Anzahl der fehlerhaften Proben wird also niemals erhöht, die Prüfung ist daher immer erfolgreich.
- Es wäre wünschenswert, wenn wir die Abarbeitung des Workflows überwachen könnten.
- Die meisten Aktivitäten sind noch leer, zum Beispiel die Bestellprüfung.

- Wir wollen nicht nur Werte verändern oder Kontrollstrukturen einsetzen, sondern auch Code ausführen. Dafür benötigen wir eigene Aktivitäten.
- Unser Workflow wird innerhalb des Prozesses der Anwendung ausgeführt und ist davon abhängig. Stürzt die Anwendung ab, bricht auch der Workflow ab. Besser wäre es, wenn der Workflow in einer Datenbank persistiert wäre, um ihn bei Bedarf wieder aufzunehmen und weiter auszuführen.

Wir werden das Beispiel also im Laufe des Kapitels weiter ausbauen. Doch zunächst erwarten Sie einige wichtige Grundlagen zu Flowcharts. Abbildung 8.19 zeigt zum Abschluss noch den gesamten Workflow, wie wir ihn bisher erstellt haben.

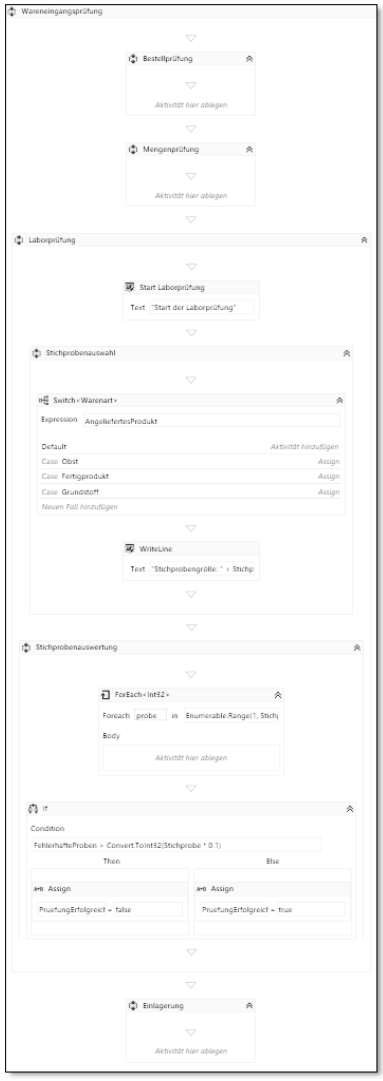


Abbildung 8.19 Der erste Workflow

8.4 Der erste Flowchart-Workflow

Wie ich schon erwähnt habe, sind Flowchart-Workflows für die meisten Anwender besser zu verstehen und entsprechen am ehesten dem, was man landläufig unter einem Geschäftsprozess versteht.

Während sequenzielle Workflows die Aktivitäten der Reihe nach ausführen, werden die Aktivitäten in einem Flowchart-Workflow entlang der Pfeile ausgeführt. Damit sind auch Rücksprünge möglich. Um zu entscheiden, in welcher Richtung der Workflow weiter ausgeführt wird, gibt es zwei »Entscheidungsaktivitäten«, die in WF als Knoten bezeichnet werden: *FlowDecision* und *FlowSwitch<T>*. Diese beiden Knoten arbeiten wie die `if`- und `switch`-Anweisung im Code. Genauer genommen gibt es noch eine weitere Aktivität: die *FlowStep*-Aktivität. Wir können sie in der Toolbox allerdings nicht finden, denn es handelt sich dabei um die Pfeile, die auch eigene Eigenschaften besitzen.

Erinnern Sie sich? Es gibt in WF nur Aktivitäten, denn auch ein Workflow ist nur eine Aktivität auf oberster Ebene. Es ist daher ohne Weiteres möglich, beide Workflow-Typen in einem Workflow zu mischen und jeweils das Beste aus beiden Welten zu verwenden. Das machen wir uns zunutze und bauen unser Beispiel um, ohne die bereits erstellten Aktivitäten verwerfen zu müssen.

8.4.1 Wareneingang reloaded

Abbildung 8.20 zeigt unseren Workflow auf der obersten Ebene. Überlegen wir uns, was wohl passiert, wenn die Bestellprüfung fehlschlägt, wenn also für den Wareneingang gar keine Bestellung gefunden wurde. In einem solchen Fall wären alle anderen Aktivitäten überflüssig.

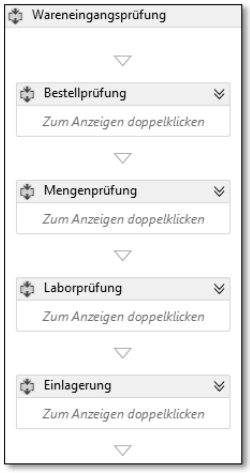


Abbildung 8.20 Die Wareneingangsprüfung im Überblick

Wollten wir diesen Sachverhalt mit einem sequenziellen Workflow abbilden, so müssten wir zu Beginn einer jeden Aktivität überprüfen, ob nicht vielleicht eine vorherige Aktivität das `PruefungErfolgreich`-Argument bereits auf `true` gesetzt hat. Das trägt nicht gerade zur Übersichtlichkeit bei und ist, nebenbei bemerkt, auch fehleranfällig.

Andererseits gibt es sehr wohl Aktivitäten innerhalb des Workflows, in denen eine einfache Sequenz vollkommen ausreicht oder gar die einzige Möglichkeit darstellt, beispielsweise die Laborprüfung. Häufig wird daher die äußere Struktur eines Workflows mit einem Flowchart-Workflow erstellt, während die einzelnen Aktivitäten oft sequenziell sind. Und genau das wollen wir jetzt auch machen. Seit WF 4.5 gibt es noch den dritten Typ Workflow, die sogenannten State Machines, die wir uns aber noch für später aufheben.

8.4.2 Den Wareneingangs-Workflow umbauen

Vorbereitung

1. Laden Sie bitte das Projekt *WorkflowWareneingang* aus dem letzten Abschnitt oder von der Buch-CD.
2. Klappen Sie alle Aktivitäten mittels ALLE REDUZIEREN für mehr Übersicht zu.
3. Wählen Sie im Kontextmenü des Projekts *Wareneingang* HINZUFÜGEN • NEUES ELEMENT aus, wählen Sie dort aus den installierten Vorlagen WORKFLOW aus, und legen Sie eine neue Aktivität an. Benennen Sie diese *WareneingangFlowchart*. Mit diesem Workflow werden wir für den Rest des Kapitels weiterarbeiten.

Sie haben nun einen neuen Workflow erzeugt, obwohl Sie als Typ *Aktivität* ausgewählt haben. Das ist ein guter Beweis dafür, dass *Workflow* und *Aktivität* synonyme Begriffe sind.

Flowchart

Noch ist der Workflow leer, und der Visual Studio Workflow Designer fordert uns auf: AKTIVITÄT HIER ABLEGEN. Die Einzahl sagt es uns bereits: Wir können nur eine einzige Aktivität dort ablegen. Viele Aktivitäten nehmen nur eine Aktivität auf. Das ist in der Praxis kein Problem, denn wir könnten ja eine *Sequence*-Aktivität hinzufügen, in der wir dann wiederum beliebig viele Aktivitäten oder natürlich auch einen weiteren Flowchart-Workflow ablegen könnten. Wir werden später übrigens eine eigene Aktivität entwickeln, die von Haus aus mehrere Kind-Aktivitäten aufnehmen kann.

1. Ziehen Sie nun eine *Flowchart*-Aktivität aus dem Reiter FLUSSDIAGRAMM in den Designer, und benennen Sie diese Wareneingangsprüfung, so wie im sequenziellen Workflow.

2. Markieren Sie alle vier Aktivitäten aus dem sequenziellen Workflow, und kopieren Sie diese in den neu erstellten Flowchart. Sie können mehrere Aktivitäten mit Hilfe der Strg-Taste auswählen, und auch Strg+C, Strg+V und Strg+X funktionieren wie gewohnt innerhalb eines Workflows und über Workflows hinweg, oder aber Sie ziehen einen Markierungsrahmen um die zu kopierenden Aktivitäten. Es fällt sofort auf, dass die Anordnung der Aktivitäten frei ist. Sie können – und sollten – die Aktivitäten also so anordnen, dass der Workflow an Übersicht gewinnt. Natürlich steht es Ihnen auch frei, die zugrundeliegenden XAML-Dateien von Hand zu editieren, wenn Sie das bevorzugen. Für die Platzierung können Sie auch die Pfeiltasten verwenden, was manchmal einfacher ist, weil Sie die Aktivitäten dann in gröberen Schritten verschieben können.
3. Platzieren Sie die Aktivität *Bestellprüfung* ganz nach oben, unterhalb des Start-Icons. In einem Flowchart-Workflow müssen Sie die Reihenfolge der Ausführung selbst angeben. Es darf also keine unverknüpften Aktivitäten geben, da diese sonst nie ausgeführt würden. Wenn Sie nun mit der Maus über START gehen, dann erscheinen Konnektoren, kleine graue Rechtecke, an den Kanten (siehe Abbildung 8.21).

Abbildung 8.21 Konnektoren

4. Streichen Sie mit der Maus über den unteren Konnektor, halten Sie die linke Maustaste gedrückt, und ziehen Sie den nun erscheinenden Pfeil auf die *Bestellprüfung*-Aktivität, für die jetzt ebenfalls Konnektoren erscheinen. Sie haben nun diese Aktivität als erste Aktivität festgelegt.
5. Ziehen Sie jetzt eine *FlowDecision*-Aktivität auf die Fläche unterhalb der eben verbundenen Aktivität, und verbinden Sie beide Aktivitäten auf dieselbe Weise. Sie werden dabei bemerken, dass die *FlowDecision*-Aktivität gleich drei Eingabekonnektoren aufweist, aus denen Sie sich einen aussuchen können. Außerdem gibt es noch zwei Eingabekonnektoren, einen für `true` und einen für `false`, ja nachdem, was in der Aktivität entschieden wird. Neu ist die Fähigkeit, auch dieser Aktivität einen Namen zu geben, über die bereits bekannte `DisplayName`-Eigenschaft.
6. Dafür benötigen wir einen Ausdruck als Entscheidungskriterium. Dafür dient das Argument `PruefungErfolgreich`. Nur wenn dieses Argument `true` ist, die Prüfung also bisher erfolgreich war, soll der Workflow weiter ausgeführt werden. Legen Sie vorher bitte noch alle vier Argumente erneut so an wie im sequenziellen Workflow. Leider können Sie die Argumente mit Designerunterstützung nicht kopieren, Sie müssen sie manuell neu anlegen – oder in der XAML-Datei direkt kopieren. Die Variablen wurden übrigens beim Kopieren der Aktivitäten über-

nommen, denn sie sind Bestandteil der jeweiligen Aktivität und nicht des gesamten Workflows.

7. Geben Sie nun für die *Decision*-Aktivität als *Condition* das Argument `PruefungErfolgreich` an. Wenn das Argument den Wert `true` besitzt, dann wird der `true`-Zweig durchlaufen, ansonsten der `false`-Zweig. Die Konnektoren für diese beiden Zweige sehen Sie, wenn Sie mit der Maus über die Decision fahren.
8. Verbinden Sie nun den `true`-Konnektor der *Decision*-Aktivität mit der nächsten sequenziellen Aktivität, der Mengenprüfung.
9. Verfahren Sie so für alle weiteren Aktivitäten, ie in Abbildung 8.22 zu sehen. Ein klein wenig schneller geht das übrigens, wenn Sie eine Aktivität gleich auf einen Konnektor fallenlassen, weil der Designer dann die Verbindung gleich für Sie zieht; Sie sparen sich also einen Arbeitsschritt.

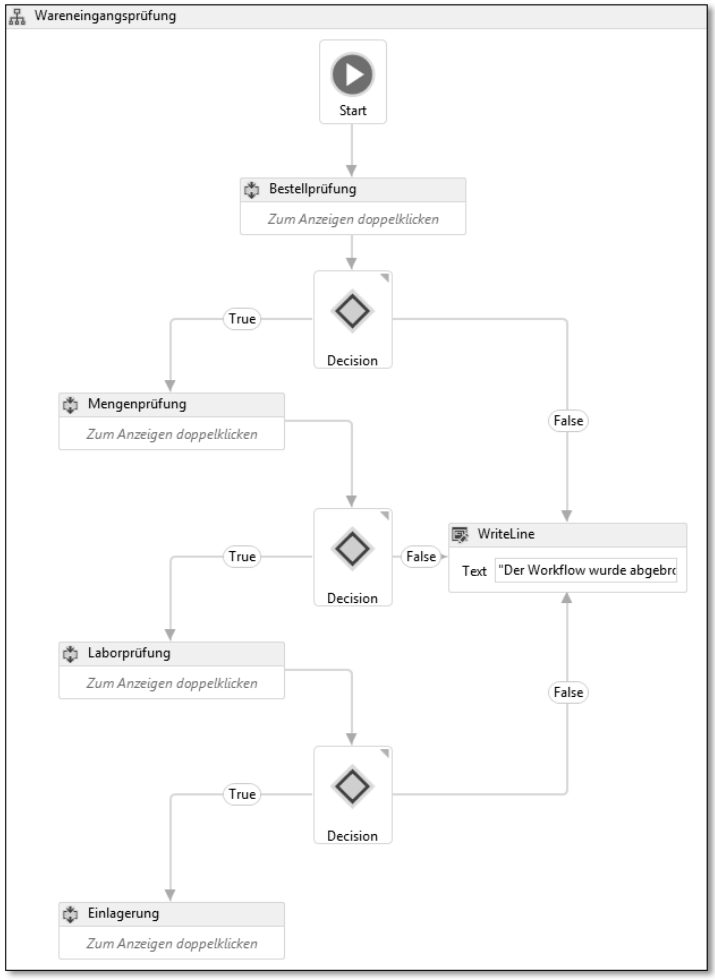


Abbildung 8.22 Der vorläufig fertige Flowchart-Workflow

10. Erstellen Sie eine neue *WriteLine*-Aktivität, benennen Sie diese mit *Abbruch*, und ändern Sie die Text-Eigenschaft in »Der Workflow wurde abgebrochen, weil eine Prüfung fehlgeschlagen ist«.
11. Verbinden Sie nun die jeweiligen `false`-Zweige der *Decision*-Aktivitäten mit dieser Aktivität.
12. Modifizieren Sie die *Program.cs*, und laden Sie anstelle des Workflows *Workflow-Wareneingang* den gerade neu erstellten Workflow *WareneingangFlowchart*.

Der Workflow sollte nun in etwa so aussehen wie Abbildung 8.23 dargestellt. Sie können ihn nun ausführen. Das In-Argument `PruefungErfolgreich` wird jedoch mit `false` initialisiert, die Meldung wird also ausgegeben, weil die Prüfung bereits bei der ersten *FlowDecision* den `false`-Zweig durchläuft, also abbricht. Wenn Sie möchten, dann platzieren Sie anschließend eine *Assign*-Aktivität in die *Bestellprüfung*, in der Sie dem Argument den Wert `true` zuweisen, und führen Sie den Workflow erneut aus; es werden alle Sequenzen durchlaufen. Praxisnah ist das freilich nicht, denn ob eine Prüfung erfolgreich war oder nicht, soll sich ja in den entsprechenden Prüfungs-Aktivitäten entscheiden – entfernen Sie die Zuweisung also später wieder. Und vergessen Sie nicht, dass Sie auch Anmerkungen hinzufügen können – für Aktivitäten.

Das ist nun ein Workflow, auf dem sich aufbauen lässt. Wir hätten übrigens den `false`-Zweig der *Decision*-Aktivitäten nicht unbedingt benötigt. Die Workflow Runtime hätte den Workflow auch so beendet, da sie auf die logisch letzte Aktivität gestoßen wäre, von der aus eine weitere Ausführung nicht mehr möglich ist. Dennoch ist es immer eine gute Idee, beide Zweige auszuführen, schon deshalb, weil explizite Aktionen (wie unsere Ausgabe durch *WriteLine*) einem Außenstehenden mehr Informationen geben als das implizite Verhalten der Runtime.

Debugging

Ein besonders praktisches Feature ist der Visual Studio Debugger für Workflows, denn er erlaubt einen grafischen Einblick in den Workflow während der Ausführung. Wählen Sie hierfür aus dem Kontextmenü einer Aktivität HALTEPUNKT • HALTEPUNKT EINFÜGEN. Der Haltepunkt wird mit einem roten Punkt angezeigt, ganz so wie im Source-Code-Editor von Visual Studio (siehe Abbildung 8.23).

Abbildung 8.23 Eine Aktivität mit einem zugewiesenen Haltepunkt

Starten Sie nun Ihre Anwendung, und der Debugger wird an der Aktivität mit dem Haltepunkt eine Pause einlegen (siehe Abbildung 8.24).

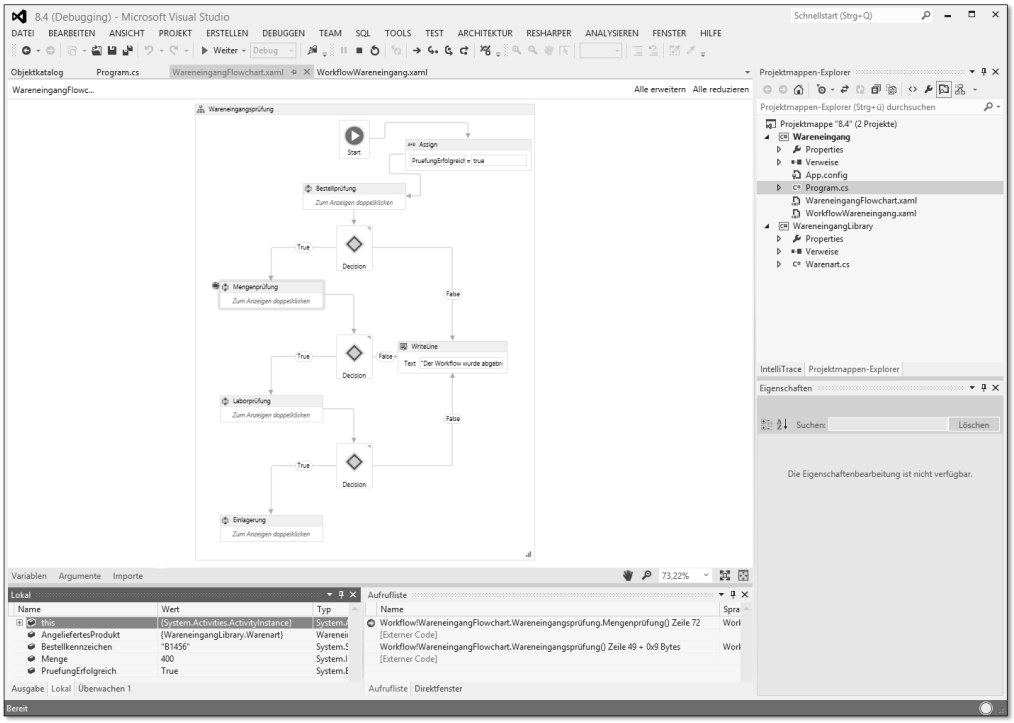


Abbildung 8.24 Der Visual Studio Debugger für Workflows

Die Möglichkeiten unterscheiden sich nicht sehr von denen des klassischen C#-Debuggings. So können Sie die lokalen Variablen inspizieren (als auch die Argumente) und den Workflow schrittweise durchlaufen. Mit *Einzelschritt* springen Sie in die einzelnen Aktivitäten hinein, während Sie mit *Prozedurschritt* jeweils zur nächsten Aktivität derselben Ebene springen.

Auf *FlowDecision*- und *FlowSwitch*-Aktivitäten können Sie keine Haltepunkte setzen, was schade ist, da Sie so den Moment der Entscheidung nicht mitverfolgen können. Aber natürlich steht es Ihnen frei, die nachfolgenden Aktivitäten mit Haltepunkten zu versehen, was praktisch auf dasselbe hinausläuft, da Sie so die Auswirkungen der Entscheidung überprüfen können.

8.5 Workflows laden und ausführen

Wir haben die Runtime bereits verwendet, um Workflows zu starten, dem Workflow Werte (= Argumente) zu übergeben und die *Out*-Argumente aus den Workflows heraus zu verarbeiten. Aber sie kann noch mehr. Grund genug, ihr einen eigenen Abschnitt zu widmen. In den folgenden Abschnitten werden wir den Workflow dann mit den eingebauten und selbst entwickelten Aktivitäten weiter ausbauen.

8.5.1 Workflows in XAML ausführen

Im einfachsten Fall besteht die Ausführung eines Workflows aus einer einzigen Zeile Code:

```
WorkflowInvoker.Invoke(new SomeWorkflow());
```

Wir haben im Beispiel bereits *In*-Argumente übergeben und *Out*-Argumente nach Beendigung des Workflows empfangen. WF verwendet hier durch den Einsatz einer `IDictionary<string, object>`-Collection eine lose Kopplung, Sie müssen also selbst auf die richtige Parameter-Bezeichnung und die richtigen Parameter-Typen achten.

8.5.2 Workflows in Code ausführen

Der Einsatz von XAML ist nicht zwingend, Sie können Ihre Workflows auch vollständig im Code definieren. Ich gehe im diesem Kapitel nicht weiter darauf ein, weil ich finde, dass gerade der deklarative Ansatz – die Modellierung des Workflows statt seiner Programmierung – einen wesentlichen Vorteil der WF ausmacht. Für die besonders Neugierigen hier aber noch ein kleines Beispiel:

```
static void WriteSomething()
{
   Sequence actSequence = new Sequence
   {
      Variables =
      {
         new Variable<string>{Name="eineVariable", Default="Standardtext"}
      },
      Activities =
      {
         new WriteLine{Text=new
         VisualBasicValue<string>("eineVariable")}
      }
   };
   WorkflowInvoker.Invoke(actSequence);
}
```

Listing 8.4 Ein einfacher Workflow im Code

Ein wenig verwirrend ist auch hier der Bezug zu Visual Basic. Ansonsten erklärt sich der Workflow weitgehend von selbst. Auch der Aufruf mittels `WorkflowInvoker` ist derselbe.

8.5.3 WorkflowApplication

Eine weitere Methode, Workflows zu starten, besteht darin, die Klasse `WorkflowApplication` zu verwenden. Die `Invoke`-Methode der Klasse `WorkflowInvoker` übernimmt die Kontrolle des Workflows im aufrufenden Thread. Die Ausführung wird also so lange blockiert, bis die Workflow Runtime die Kontrolle wieder an den Aufrufer übergibt.

Das ist nicht immer sinnvoll, denken Sie zum Beispiel an langlaufende Workflows oder das Starten von mehreren Workflows aus dem Code heraus. Die Klasse `WorkflowApplication` können Sie instanziieren. Sie besitzt eine `Run`-Methode zur Ausführung des Workflows und startet zu diesem Zweck einen eigenen Thread. Der ausführende Code erhält die Kontrolle also wieder zurück.

```
WorkflowApplication wa = new WorkflowApplication(new WareneingangFlowchart(),
  Arguments);
wa.Run();
```

Erwartungsgemäß liefert die `Run`-Methode keine Werte zurück, der Aufruf ist ja asynchron. Dafür bietet die `WorkflowApplication` aber einige Ereignisse an, für die Sie eigene Delegates schreiben können. Das Ereignis `Completed` zeigt die Beendigung eines Workflows an:

```
wa.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
   string pruefungErfolgreich =
   e.Outputs["Pruefung erfolgreich"].ToString();
   Console.WriteLine("PrüfungErfolgreich? {0}",
   pruefungErfolgreich);
};
```

Die Klasse `WorkflowApplicationCompletedEventArgs` übermittelt wichtige Informationen an den Delegaten:

- `CompletionState` gibt einen Wert aus der Aufzählung `ActivityInstanceState` zurück. Eine Aktivität (respektive ein Workflow) wird üblicherweise abgeschlossen sein, `CompletionState` enthält dann den Wert `Closed`. Weitere Werte sind: `Canceled` = abgebrochen, `Executing` = Aktivität wird noch ausgeführt, und `Faulted` = Aktivität befindet sich in einem Fehlerzustand.
- `Outputs` enthält die *Out*-Argumente, wie gewohnt als `Dictionary<string, object>`-Objekt.
- `InstanceId` enthält eine GUID, welche die Workflow-Instanz eindeutig identifiziert. Damit könnten Sie denselben Handler für mehrere Workflow-Instanzen verwenden.

- `TerminationException` gibt die Exception zurück, die im Zusammenhang mit dem Abbruch des Workflows steht.

Das Event `OnUnhandledException` wird aufgerufen, sobald es im Workflow zu einem Fehler gekommen ist, der nicht von Ihnen behandelt wurde:

```
wa.OnUnhandledException =
 delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
{
   Console.WriteLine("Fehlerquelle: "+
      e.ExceptionSource.DisplayName);
   return UnhandledExceptionAction.Terminate;
};
```

Listing 8.5 Auf unbehandelte Fehler reagieren

Die Argumente der zugehörigen Klasse `WorkflowApplicationUnhandledExceptionEventArgs`:

- `ExceptionSource` liefert die Aktivität, die die Quelle für den Fehler ist, `ExceptionSourceInstanceId` deren GUID.
- `UnhandledException` enthält die nicht behandelte Exception.
- Sie müssen zudem angeben, was nun geschehen soll. Dafür verwenden Sie einen Wert aus der Enumeration `UnhandledExceptionAction`.

Das Ereignis `Aborted` tritt auf, sobald ein Workflow abgebrochen wurde. Dies kann ein Entwickler selbst auslösen, indem er die `Abort`-Methode der Klasse `WorkflowApplication` verwendet, oder die Runtime entscheidet sich dafür.

```
wa.Aborted = delegate(WorkflowApplicationAbortedEventArgs e)
{
   Console.WriteLine(e.Reason);
};
```

Listing 8.6 Ein Workflow wird abgebrochen.

Die Informationen, die Ihnen `WorkflowApplicationAbortedEventArgs` liefert, sind spärlich:

- `Reason` ist die Exception, die zum Abbruch des Workflows geführt hat. Wurde er explizit abgebrochen, wie weiter oben gezeigt, so erhalten Sie hier eine `WorkflowApplicationAbortedException`.
- `InstanceId` enthält auch hier wieder die eindeutige Identifikation der Workflow-Instanz.

Index

- (private) ... 144
(protected) ... 144
+ (public) ... 144
~ (paket) ... 144

A

Abbrecheranalyse ... 1055
Abkürzung ... 371
Ablaufplan ... 1042
Abnahmetest ... 993, 1060, 1085
Abonnenten ... 844
Abort ... 252
Aborted ... 252
AbortRequested ... 251
AboveNormal ... 255
Abstrakte Klasse ... 153, 398
Abstrakte Methode ... 153
Abstraktion ... 138
Accumulate ... 642
Achsenmethoden ... 723
ACID ... 328
ACID-Prinzip ... 109
Action-Delegat ... 320
Activity ... 802
Activity<TResult> ... 805
ActivityDesigner ... 820
ActivityInstanceState ... 770, 805
ActivityScheduledQuery ... 844
ActivityScheduledRecord ... 846
ActivityStateQuery ... 844, 845
ActivityStateRecord ... 846
Add (Interlocked) ... 285
AddAfterSelf ... 728
AddBeforeSelf ... 728
AddFirst ... 728
AddImplementationVariable ... 816
AddOrUpdate ... 290
AddValidationError ... 813
ADO.NET ... 71
ADO.NET Data Services ... 705
ADO.NET Entity Framework ... 72, 678
Adresse ... 459, 867
  Service ... 451
Advanced Services ... 658
AggregateException ... 263, 265, 270, 409
Aggregatfunktionen
  benutzerdefinierte ... 642
Aggregation ... 157
Aktivität ... 733
  Assign- ... 767, 776
  Delay- ... 788
  DoWhile- ... 789
  eigene entwickeln ... 800
  eingebaute ... 772
  Flowchart- ... 764
  FlowDecision- ... 765
  ForEach<T>- ... 775
  InvokeMethod- ... 780
  Parallel- ... 780
  ParallelForEach<T>- ... 777
  Persist- ... 841
  RemoveFromCollection<T>- ... 776
  Switch-<T>- ... 758
  TransactionScope- ... 829
  While- ... 789
Aktivitätsdiagramm ... 124
Algorithmische Zeitschätzung ... 1122
ALTER FULLTEXT INDEX ... 661
Alternative ... 310
Ambient-Transaktion ... 329
Analyse, stilistische ... 994
Ancestors ... 724
Ancestors<T> ... 724
AncestorsAndSelf ... 724
Anforderung ... 44, 1087
  Aufbau ... 48
  funktionale ... 44, 1089
  Merkmale ... 49
  nicht-funktionale ... 44, 1089
  Umfang ... 49
Annahmeschluss ... 1083
Anonyme Methode ... 318, 323
Anwendung
  mehrschichtige ... 115
  monolithische ... 114
Anwendungsbeschreibung ... 1089
Anwendungsfalldiagramm ... 123
Anwendungstyp ... 41
API ... 175
App Bar ... 920, 954
app.config ... 203
App_Code ... 486

AppFabric ... 84, 495
Application Architect ... 43
Application Block ... 224
Application Pool ... 485
Application Server Extensions ... 495
Appx Manifest ... 930
Äquivalenzklasse ... 1036
Architekt → Softwarearchitekt
Architektur → Softwarearchitektur
Architekturfehler ... 1015
Argumente ... 755, 812
ASP.NET MVC 2 ... 65
ASP.NET Web-API ... 705
Assert ... 1071
Assign-Aktivität ... 761, 767, 776
Assoziation ... 154
  reflexive ... 155
Assoziationsklasse ... 156
async ... 272
async/await ... 272
AsyncCodeActivity ... 805
Asynchrone Ausführung ... 216
Asynchrone Schnittstelle ... 173
Asynchrone Verarbeitung ... 242
atomic ... 328
AtomPub ... 706
AttachedToParent ... 260, 266
Attribute ... 143, 148
  benennen ... 376
  Sichtbarkeit ... 144
Attributes ... 724
auditLogLocation ... 595
Aufwand/Dauer (Unterscheidung) ... 1104
Ausdruck, regulärer ... 307
Ausfallsicherheit ... 83
Ausführung, asynchrone ... 216
Ausnahmen ... 112
AuthenticationType ... 593
Authentifizierung ... 91, 93
  Verfahren ... 588
  WCF ... 582
Auto-commit ... 329
Automatische Speicherverwaltung ... 348
Autorisierung ... 91, 93
  WCF ... 583, 595
await ... 272

B

Background ... 252
BAL ... 772
Bandwurmcode ... 382
Base Activity Library ... 739, 748, 772
Baseline Provisioning ... 1046
basicHttpBinding ... 462, 591
Basisadresse ... 467
Batch ... 716
Bedingungen, kontextsensitive ... 313
Begleitete Einführung ... 1086
BehaviorElementExtension ... 603
BelowNormal ... 255
Benannte Methode ... 318
Benennung
  Attribute ... 376
  Code-Elemente ... 368
  Klassen ... 375
  Namespaces ... 374
  Schnittstellen ... 375
  Verzeichnisse ... 377
Benutzerdefinierte Aggregatfunktion ... 642
Benutzerdefinierte Datentypen ... 628
Benutzerdefinierte Funktion ... 638
Benutzeroberfläche ... 180
  Fallstudie ... 192
  gute ... 183
Berechtigungssatz ... 635
Best Practices ... 28
Beta-Version ... 1083
Betaversion ... 1084
Bewertete Function Points ... 1120
Beziehung ... 53
  Dependency Injection ... 54
  enge Kopplung ... 53
  hierarchische ... 53
  lose Kopplung ... 54
  zwischen Klassen ... 151
Bidirektionale Schnittstelle ... 173
BinaryFormatter ... 341, 343
Binding ... 451, 460, 867
  vordefiniertes ... 461
BindingConfiguration ... 467
BizTalk ... 100, 737
Blackbox ... 1034
BlockingCollection ... 288
Bookmark ... 825
BookmarkResumptionQuery ... 844
BookmarkResumptionRecord ... 847
Bottom-up-Ansatz ... 53
Boxing ... 393
BPML ... 56, 738
BPMN ... 738
Break ... 271

Bug ... 1012
Bugzilla ... 1095
Business Component ... 67
gestalten ... 69
Business Entity ... 67
Business Layer ... 67
Empfehlungen ... 69
Business Process Modeling Language ... 56, 738
Business Process Modeling Notation ... 738
Businesslogik
Datenbank ... 645
Byte ... 1039

C

Cache ... 88
CacheMetadata ... 812
Callback ... 397, 541
CamelCase ... 373
Canceled ... 806
CancellationHandler ... 833
CancellationScope ... 837
CancellationScope-Aktivität ... 837
CancellationToken ... 262, 271
CancellationTokenSource ... 262
CancelRequestedQuery ... 844
CancelRequestedRecord ... 847
CanCreateInstance ... 890
catch ... 293
Change Request ... 1010, 1079, 1087
Change Tracking ... 108
ChannelFactory ... 514, 613
ChannelFactory<T> ... 515
Charms ... 919
CI ... 1044
Client
Identität ... 592
WCF ... 446, 468, 499
ClientBase<T> ... 504
clientCredentialType ... 590
Client-Server
Architektur ... 115
System ... 115
Closed ... 806
CloseTimeout ... 477
Closure ... 325
clr enabled ... 627
Code
formatieren ... 376
guter ... 365
kommentieren ... 386
Code (Forts.)
Lesbarkeit ... 365
schlechter ... 362
Struktur ... 376
Wartbarkeit ... 365
Codeabdeckung ... 1071
CodeActivity ... 804, 806, 811
CodeActivity<TResult> ... 805
CodeActivityContext ... 804
Code-Analyse ... 994
Codeanalyse ... 420
Codeausschnitt-Manager ... 427
Code-Elemente
benennen ... 368
Code-Metrik ... 994
Codereview ... 1061
CodeRush ... 415
Code-Smell ... 412
Code-Snippets ... 425
Code-Styleguide ... 367
Codezeilen-Länge ... 382
Collect ... 353
Collections
threadsichere ... 288
ColumnCount ... 641
Commit ... 329
Committable Transaction ... 332
CommunicationException ... 549, 553
Community Technology Preview ... 1084
CompareExchange ... 285
CompensableActivity-Aktivität ... 832
Compensation ... 833
CompensationHandler ... 832
CompensationToken ... 833
Complete ... 330
CompleteAdding ... 289
Completed ... 770
CompletionCondition ... 838
CompletionState ... 770
Component ... 434
Composite Activities ... 736
Conceptual Model ... 679
Conceptual Schema Definition Language ... 679
Concurrency ... 106
ConcurrencyMode ... 283
ConcurrencyMode.Reentrant ... 545
ConcurrentBag ... 288
ConcurrentDictionary ... 290
ConcurrentStack ... 288
ConfigurationManager ... 204
Confirmation ... 833

consistent ... 328
contains ... 663
CONTAINSTABLE ... 664
Content-based correlation ... 862, 890
ContentHandle ... 890
Context connection ... 638
Context correlation ... 862
ContinueOnError ... 716
ContinueWith ... 266
Continuous Delivery ... 997, 1043
Continuous Deployment ... 997
Continuous Integration ... 997, 1044
Contract ... 82, 116, 452, 463, 867
Contract (WinRT) ... 969
Contract First Development ... 503
Contracts (WinRT) ... 921
Controller ... 65
CorrelatesOn ... 891
Correlation ... 862, 888
CorrelationHandle ... 862, 890
CorrelationInitializers ... 890
CorrelationScope-Aktivität ... 862
CR ... 1084, 1087
CreateBehavior ... 603
Critical ... 1009
CSDL ... 679
CSV ... 175
CTP ... 1084
CurrentThread ... 250
Custom Activity ... 748, 801
Custom Bindings ... 463
Custom TrackingParticipant ... 849
CustomBindings ... 461
Customizing ... 135
CustomTrackingQuery ... 844
CustomTrackingRecord ... 847, 852
Cycle Time ... 1045

D

Data Architect ... 43
Data Contracts ... 464, 523, 862, 868
  Kompatibilität ... 534
  Versionierung ... 534
Data Layer ... 71, 75
  Merkmale ... 75
Data Member ... 868
Data Member (Attribut) ... 523
Data Transfer Objects ... 70, 523
DataContract ... 441
DataContract (Attribut) ... 523, 868
DataContractSerializer ... 346, 524
DataService ... 706
DataServiceCollection ... 712
DataServiceContext ... 711
Daten
  abfragen ... 651, 662
  modifizieren ... 654
  sinnlose ... 1039
Datenbank ... 625
  Businesslogik ... 645
  Konfiguration ... 210
Datenkonvertierung ... 1030
Datenmodell ... 124
Datensatzsperre ... 107
Datentypen
  benutzerdefinierte ... 628
Datenzugriff ... 625
DateTime ... 1038
DbContext ... 695
DBMS ... 625
DbSet ... 692
DbUpdateConcurrencyException ... 701
Deadlock ... 282
Debugging
  Workflows ... 767
Decrement ... 285
Default (GCCollectionMode) ... 354
Defekt ... 1012
Definitionsbereich ... 1037
Delay-Aktivität ... 788
Delegat ... 317
  Action- ... 320
  Func- ... 320
delegate (Schlüsselwort) ... 317
Delphi-Methode ... 1114
Demilitarisierte Zone ... 92
Denial of Service ... 92, 245
Dependency Injection ... 54, 166, 411
Deployment ... 103, 1085
Deployment Notes ... 1086
DescendantNodes<T> ... 724
DescendantNodesAndSelf ... 724
Deserialisierung ... 339
Deserialize ... 343
Design ... 131
Design Patterns ... 28
Designentscheidungen ... 160
Designer einbinden ... 905
Designer Rehosting ... 904
Designfehler ... 1015
Designmerkmale ... 81

DevOps ... 102
Diagrammtypen ... 123
Aktivitätsdiagramm ... 124
Anwendungsfalldiagramm ... 123
Entity-Relationship-Diagramm ... 125
Klassendiagramm ... 143
Kommunikationsdiagramm ... 124
Komponentendiagramm ... 123
Kompositionsstrukturdiagramm ... 123
Paketdiagramm ... 123
Sequenzdiagramm ... 124
Use-Case-diagramm ... 123
Verhaltensdiagramm ... 123
Zustandsdiagramm ... 124
Dialog ... 215
gestalten ... 219
Dienstverweis ... 448
aktualisieren ... 501
hinzufügen ... 499
konfigurieren ... 501
Dirty Read ... 110
Disconnected Data ... 71
DiscoveryClient ... 458
DisplayName ... 752
Dispose ... 356
disposing ... 358
Distributed Transaction Coordinator ... 328, 329, 557, 559
DistributedIdentifier ... 330, 563
DMZ ... 92
Dokumentation ... 121, 164
der Konfiguration ... 198
Inhalte ... 126
DOM ... 721
DoS ... 92, 245
DoWhile-Aktivität ... 789, 795
DTC ... 557, 559
DTO ... 70, 523
Dublin ... 495
dueTime ... 258
Duplex-Kommunikation ... 538
durable ... 328
Dynamischer Test ... 994

E

Eager Loading ... 695
EAI ... 56, 100, 172
EF ... 678
Effektive Priorität ... 255
Effizienz ... 163
Eigene Exception-Klassen ... 304
Eigenschaften ... 406
oder Funktionen ... 400
Einrückung ... 380, 383
Einschränkung ... 1090
Einzahl ... 372
Einzelschätzung ... 1112
Eleganz ... 365
Elements<T> ... 724
Eltern-Thread ... 266
Embedded-Systeme ... 41
Empirische Zeitschätzung ... 1122
Encoding ... 461
EncryptAndSign ... 589
Endpoint ... 451
EndpointAddress ... 458
Endpunkte ... 451
eigene ... 453
konfigurationslose ... 452
Standard- ... 456
Enge Kopplung ... 53, 82
Enlistment ... 330
Enter (Monitor) ... 287
Enterprise Application Integration 56, 100, 172
Enterprise Library ... 131, 224, 553
Enterprise Service Bus ... 101
Entität ... 698
Entity Framework ... 72, 678
Entity SQL ... 681, 691, 694
Entity-Relationship-Diagramm ... 125
EntityState ... 699
Entscheidungstabellen ... 1041
Entwurf ... 121
Entwurfsmuster ... 28
Enum ... 405
enum
WCF ... 529
EnumMember ... 529
EPK ... 56
ER-Diagramm ... 125
Ereignisanzeige ... 854
Ereignisgesteuerte Prozesskette ... 56
Ereignisse ... 843
Ereignissteuerung ... 793
Erscheinungsdatum ... 1083
Ersetzen ... 308
Erweiterbarkeit ... 134, 166
Erweiterungsmethoden ... 335, 403, 723
ESB ... 101
Escape ... 310
ETW ... 843

EtwTrackingParticipant ... 848
Event ... 397
Event Tracing for Windows ... 843
EventData ... 641
Evolvierbarkeit ... 134
Exception ... 292, 303, 407
fangen ... 294
Hierarchie ... 167, 299, 550
Logging ... 168
unbehandelte ... 301
Exception Handling ... 112, 167, 291
WCF ... 509, 546
Exception Handling Application Block ... 233
Exception-Klassen, eigene ... 304
ExceptionPolicy ... 235
ExceptionSource ... 771
ExceptionSourceInstanceId ... 771
Exchange (Interlocked) ... 285
Execute ... 805
ExecuteAndSend ... 638
Executing ... 806
ExecutionProperties ... 829
expansion ... 667
Exploratives Testen ... 1001, 1035
Expression Tree ... 322, 324
Extensibility ... 134
Extensible Application Markup Language ... 750
ExtensionData ... 538
Extreme Programming ... 997
Eye Tracking ... 1055

F

Facade ... 63
FailFast ... 409
Fallbeispiel ... 29
Fassade ... 63
Fault Contracts ... 234, 464, 551
Faulted ... 548, 806
FaultException ... 548
FaultPropagationQuery ... 845
FaultPropagationRecord ... 847
Faults ... 547
Feature ... 1013
Fehler ... 1087
Fehlerbehandlung ... 112, 291, 407
WCF ... 509, 546
Fehlermeldung ... 168
Fehlerpriorität ... 1009
Change Request ... 1010
Critical ... 1009
Fehlerpriorität ... 1009 (Forts.)
Low ... 1010
Medium ... 1009
Showstopper ... 1009
Fehlerstatus ... 1011
Feldtest ... 1058
Filestream ... 667
Volltextsuche ... 676
Filled ... 982
Finalizer ... 355
finally ... 302, 410
FindCriteria ... 458
FindResponse ... 458
Fire and Forget ... 174, 538, 735
Flatten ... 266
Float ... 1039
Flowchart-Aktivität ... 764, 790
Flowchart-Workflow ... 741, 763
FlowDecision-Aktivität ... 763, 765
FlowStep-Aktivität ... 763
FlowSwitch-Aktivität ... 763, 790
FOR DELETE ... 640
FOR INSERT ... 640
FOR UPDATE ... 640
Forced ... 354
ForEach-Aktivität ... 760, 776
Forecast ... 1107
Format.Native ... 630
Formatierung ... 380
Fortschrittsbalken ... 212, 215
Marquee-Mode ... 216
Frame-Klasse ... 945
Freetext ... 663
FREETEXTTABLE ... 664
FromCurrentSynchronizationContext ... 271
FULL POPULATION ... 661
FullScreenLandscape ... 980
FullScreenPortrait ... 979
FULLTEXT INDEX ... 661
fulltext_load_thesaurus_file ... 667
Func-Delegat ... 320
Function Points ... 1116
bewertete ... 1120
unbewertete ... 1119
Funktionale Anforderung ... 44, 1089
Funktionales Modell ... 123
Funktionen ... 378, 398, 1087
benutzerdefinierte ... 638
Größe ... 399
Zuständigkeit ... 399

Funktionsobjekt ... 324
FxCop ... 420

G

Garbage Collection ... 349
Garbage Collector ... 348, 354
Gated-Check-In ... 368
GC ... 348
GCCollectionMode ... 354
Gemini ... 1095
Generalisierung/Spezialisierung ... 151
Generationen ... 350
Generics (WCF) ... 533
German Testing Board ... 1002
Geschachtelte TransactionScopes ... 331
Geschäftsprozess ... 55
Modellierung ... 56
Geschäftsprozessmodell ... 1043
Geschäftsregel ... 1089
GetBaseException ... 304
GetConsumingEnumerable ... 290
GetGeneration ... 354
Gleichzeitige Verarbeitung ... 106
goes to ... 319
Grenzwert ... 1037
GridView ... 944
Groups ... 316
Gruppe (Regex) ... 315
Gruppenschätzung ... 1114
Gruppierung ... 310
GUI ... 180
Konzept ... 193
Prototyp ... 195

H

HandleException ... 235
HelpLink ... 303
Hierarchie ... 139
Hierarchische Beziehung ... 53
Highest ... 255
Hintergrund-Thread ... 255
Horizontale Skalierung ... 90
Hosting ... 446, 471
HTML5 ... 181
httpGetEnabled ... 456

I

IClientMessageInspector ... 604
ICommunicationObject ... 515
Identität (WCF) ... 592
IDisposable ... 357
IEndpointBehavior ... 603
IExtensibleDataObject ... 507, 537
If-Aktivität ... 761
IFilter ... 676
IFormatter ... 343
IgnoreCase ... 315
IIS ... 483
IIS-Express ... 473
IMetadataExchange ... 454
immersive ... 922
Immutable ... 283
impersonateCallerForAllOperations ... 594
Impersonation ... 584, 594
ImpersonationOption ... 594
inactivityTimeout ... 471
InArgument ... 807
IncludeExceptionDetailInFaults ... 464, 510, 553
Increment ... 284
INCREMENTAL POPULATION ... 661
Indexed Views ... 661
Indizes anlegen ... 651
Informationen in Dialogen ... 219
Infrastructure Architect ... 43
Init ... 642
InnerException ... 265, 304, 409
InnerExceptions ... 270
INotifyCollectionChanged ... 712
INotifyPropertyChanged ... 507
InOutArgument ... 807
Installation ... 1085
installutil ... 480
InstanceCompletionAction ... 839
InstanceContext ... 544
InstanceContextMode ... 571
InstanceEncodingOption ... 839
InstanceId ... 770, 771
InstanceStore ... 839
Instanzen (WCF) ... 569
Instanziierung
Per Call ... 570
Per Session ... 570
Singleton ... 570
Instrumentation ... 103
Int32 ... 1039
Int64 ... 1039

Integration ... 98, 169, 172
Integration Architect ... 43
Integrationstest ... 993
Interlocked ... 284
Internet Information Server ... 483
Interoperabilität ... 98
InvalidOperationException ... 833
Inversion of Control ... 166
Invoke ... 761, 769
InvokeMethod-Aktivität ... 780
IObjectContextAdapter ... 695
IoC ... 166
IPC ... 462
IProducerConsumerCollection ... 288
IRemotingFormatter ... 343
IRequestHandler ... 706
Is-a-Beziehung ... 152
IsAnonymous ... 593
IsBackground ... 255
IsByteOrdered ... 630
IsCancellationRequested ... 263
IsCompleted ... 270
IsExceptional ... 271
IsFixedLength ... 630
IsGuest ... 593
IsHeld ... 286
IsHeldByCurrentThread ... 286
isolated ... 328
Isolation Level ... 109, 331, 565
IsOneWay ... 540
isRequired ... 536
IsServerGC ... 355
IsStopped ... 271
Issue ... 1012
Ist-eine-Beziehung ... 152
ISTQB Certified Tester ... 1002
IsUpdatedColumn ... 641

J

Java Script Object Notation ... 706
Join ... 251
JSON ... 706

K

Kalimba Sunfood GmbH ... 29
Kernel Transaction Manager ... 559
Kernel-Mode-Objekte ... 283
Key User ... 1079
Kind-Task ... 266
Klassen ... 140
abstrakte ... 153, 398
benennen ... 375
Beziehungen ... 151
Größe ... 394
gutes und schlechtes Design ... 159
Hierarchie ... 391
oder Schnittstellen ... 391
oder struct ... 393
partielle ... 377
statische ... 398
Klassenattribut ... 144
Klassendiagramm ... 143
Klassenoperation ... 145
Knoten
hinzufügen ... 654
Known Issues ... 1005
KnownType ... 533
Kommentare
// vs. / */* ... 388
/// ... 389
Platzierung ... 391
sinnvolle ... 386
Visual Studio ... 389
Kommunikationsdiagramm ... 124
Kommunikationsweg der Schnittstellen ... 175
Komparative Zeitschätzung ... 1122
Kompatibilität von Data Contracts ... 534
Kompensation ... 831
Kompensationen ... 109, 558
Komponente → Softwarekomponente
Komponentendiagramm ... 123
Komponententest ... 993, 1053
Komposition ... 157
Kompositionsstrukturdiagramm ... 123
Konfiguration ... 105, 197
Backup ... 198
Dokumentation ... 198
in der Datenbank ... 210
Versionierung ... 198
WCF ... 464
Konfigurationslose Endpunkte ... 452
Konfigurationsmanagement
Probleme ... 1019
Konstruktoren ... 399
Kontextsensitive Bedingungen ... 313
Kontextwechsel ... 248
Kontrollstrukturen ... 739
Kopplung ... 82
enge ... 82
logische ... 170

Kopplung ... 82 (Forts.)
lose ... 82
KTM ... 559

L

Lambda Expression ... 324
Lambda Statements ... 324
Lambda-Ausdruck ... 317, 319, 324
Lastbegrenzung ... 579
Lasttest ... 1065
Laufzeitfehler ... 1016
Laufzeitumgebung ... 1028
Workflows ... 748
Layer ... 82
Business ... 67
Data ... 71
Nachteile ... 61
Presentation ... 64
Softwarearchitektur ... 59
UI ... 64
Vorteile ... 60
LayoutAwarePage ... 977
Lazy Loading ... 695
Lead Architect ... 43
Leerwörter ... 373
Leerzeichen ... 386
Leichtgewichtige Transaktion ... 329
Leistungsüberwachung ... 359
Lesbarkeit von Code ... 365
Lightweight Transaction Manager ... 559
like ... 656
Lines of Code ... 994
LINQ ... 381, 712
to Entities ... 691
to XML ... 719
LINQ to EF ... 681
ListView ... 945
LLBLGen ... 74, 678
Load ... 842
Load Balancer ... 85
Load Balancing ... 734
LoC ... 994
LocalIdentifier ... 330
Locking ... 281
log4net ... 553
LogCategory ... 237
Logger ... 226
Logging ... 104, 169
Empfehlungen ... 232
von Exceptions ... 168
Logging Application Block ... 225
Logische Kopplung ... 170
Logische Sperre ... 107
Logischer Fehler ... 1016
Logisches Modell ... 679
Lokale Transaktion ... 328
Lokalisierung ... 97
Lookahead ... 313
Lookbehind ... 313
Lose Kopplung ... 54, 82
Lost Updates ... 110
Low ... 1010
Lowest ... 255
LowestBreakIteration ... 270, 271
LTM ... 559

M

machine.config ... 210
Managed Heap ... 348
Managed Services ... 478
Managed Thread ... 248
ManagedThreadId ... 251
Mantis ... 989
Marquee-Mode ... 216
Match ... 307
MatchCollection ... 308
MaxConcurrentCalls ... 580
MaxConcurrentInstances ... 580
MaxConcurrentSessions ... 580
MaxDegreeOfParallelism ... 271
Medium ... 1009
Mehrschichtige Anwendung ... 115
Mehrzahl ... 372
Meilensteine ... 1078
Merge ... 643
Message Contracts ... 464
Message Queuing ... 101, 607
MessageAuthenticationAuditLevel ... 596
MessageInspector ... 604
Metadata Exchange ... 454
MethodBase ... 304
Methoden
abstrahieren ... 418
abstrakte ... 153
anonyme ... 318, 323
benannte ... 318
Erweiterungs- ... 335, 403, 723
extrahieren ... 419
überladen (WCF) ... 521
zusammenlegen ... 419

MethodImpl ... 288
MEX-Endpunkte ... 453
mexHttpBinding ... 454
mexHttpsBinding ... 454
MexNamedPipeBinding ... 454
mexTcpBinding ... 454
Microsoft App Store ... 922
Microsoft BizTalk ... 100, 737
Microsoft Fakes ... 989
Microsoft Message Queuing ... 101, 329
Middlewaresysteme ... 329
Milestones ... 1078
Mobile Anwendung ... 41
Mocking Framework ... 989
Mock-Objekte ... 1051
Model ... 64
Model View Controller ... 64
Modell ... 138
  Daten ... 124
  funktionales ... 123
  Sicherheit ... 124
  Verteilung ... 124
Modultest ... 1050
Monitor ... 286
Monitoring ... 103, 358
Monolithische Anwendung ... 114
Moq ... 989
Moving Targets ... 1110
MSDTC ... 328, 329
MSL ... 680
MSMQ ... 101, 607
  installieren ... 609
MTOM ... 462
MulticastDelegate ... 318
Multiplizität ... 145, 155
Multithreading ... 242
  In .NET ... 249
Multi-Tier-Anwendung ... 115
MVC ... 64, 705
  Vorteile ... 66

N

Namenskonventionen ... 374
Namespace
  benennen ... 374
NativeActivity ... 805
NativeActivity<TResult> ... 805
NativeActivityContext ... 805
Natural User Interface ... 917
NEAR ... 665
Nebenläufigkeit ... 242
netMsmqBinding ... 462, 591
netNamedPipeBinding ... 462, 588
netTcpBinding ... 462, 588
Never change a running system ... 412
nHibernate ... 74, 678
Nicht-funktionale Anforderung ... 44, 1089
NMock ... 989
Nomenklatur ... 371
Non-Repeatable Read ... 110
NonSerialized ... 343
Normal ... 255
NotifyRethrow ... 236
NotOnCanceled ... 267
NotOnFaulted ... 267
NotOnRunToCompletion ... 267
NotSupportedException ... 258
NServiceBus ... 545
nServiceBus ... 102
NT-Services ... 478
NUI ... 917
Null ... 1038
NUnit ... 988

O

ObjectContext ... 695
ObjectDisposedException ... 358
Objekte ... 140
  identifizieren ... 141
Objektorientierte Analyse und Design ... 135
Objektorientierte Programmierung ... 136
Objektrelationale Mapper ... 72
ObservableCollection ... 712
Occasionally Trouble Forecast ... 1109
ODP ... 705
OleTransactions ... 560
OnDeserialized ... 344
OnDeserializing ... 344
OnlyOnCanceled ... 267
OnlyOnFaulted ... 267
OnlyOnRanToCompletion ... 267
OnSerialized ... 344
OnSerializing ... 344
OnStart ... 479
OnStop ... 479
OnUnhandledException ... 771
OOA/OOD ... 135
OOP ... 136
Open Data Protocol ... 705
OpenTimeout ... 477

OperationCanceledException ... 263
OperationContext ... 543
OperationContract ... 443, 463, 862
Operationen ... 150
Optimistisches Locking ... 701
Optimized ... 354
OptionalField ... 345
Orchestration Layer ... 68, 858
Orchestration Services ... 858
order by ... 653
OrderDataService ... 707
ORM ... 72, 678
OR-Mapper ... 70, 72, 678
OutArgument ... 807
Outputs ... 770
Oversubscription ... 249

P

Paketdiagramm ... 123
PAP ... 1042
Parallel.For ... 269
Parallel.ForEach ... 269
Parallel.Invoke ... 272
Parallel-Aktivität ... 780
Parallele Schleifen ... 269
Parallele Verarbeitung ... 241, 778
ParallelForEach<T>-Aktivität ... 777, 782
ParallelLookResult ... 270
ParallelLoopState ... 271
Parameter ... 402
abstrahieren ... 418
entfernen ... 418
hinzufügen ... 417
ParameterizedThreadStart ... 251
Partielle Klassen ... 377
PascalCase ... 373
Penetrationstest ... 1064
Per Call ... 570
Per Session ... 570, 573
Performance ... 86
by Design ... 729
Performancetest ... 1065
Persist ... 842
Persist-Aktivität ... 841
Persistenz ... 838, 888
Phantom ... 110
Phasenmodell ... 997
Pick-Aktivität ... 795
PickBranch-Aktivität ... 795
Plain Old CLR Object ... 698
Plugin ... 97
POCO ... 698
Polymorphie ... 151
PostHandlingAction ... 235
Präfix ... 370
Predicate ... 326
Presentation Layer ... 64
Principal ... 593
PrincipalPermissionMode ... 596
PrincipalPermisson ... 596
Priorisierung ... 1082
der Testfälle ... 1026
Priorität ... 255, 1092
Priority ... 255
private ... 144
ProcessRequestForMessage ... 706
Programmablaufplan ... 1042
Programmzustände ... 1040
Promotable Single Phase Enlistment ... 329
protected ... 144
protectionLevel ... 589
Prototyp GUI ... 195
Proxy ... 440
erzeugen ... 499
verwenden ... 507
Prozess ... 55
Prozesskette
ereignisgesteuerte ... 56
Prozess-Kontextwechsel ... 248
Prozessorientierung ... 732
PSPE ... 329
public ... 144
Pulse ... 288
PulseAll ... 288

Q

QS-Team ... 1058
Quantifizierer ... 312
query ... 652
Query correlation initializers ... 890
Query Expressions ... 381
Queues anlegen ... 609
QueueUserWorkItem ... 257

R

Race Condition ... 282
RANK ... 664
RC ... 1084
Read (Interlocked) ... 285

Reason ... 771
Receive-Aktivität ... 881
ReceiveAndSendReply-Aktivität ... 882
Redundanz ... 84
Refactoring ... 52, 411
Gründe ... 411
Muster ... 415
Prozess ... 413
Reference.cs ... 506
Reference.svcmap ... 501
Reflexive Assoziation ... 155
Regex ... 307
Gruppe ... 315
Regex.Split ... 308
RegexOptions ... 315
Regressionsfehler ... 1014
Regressionstest ... 1018
Reguläre Ausdrücke ... 307
RegularExpressions ... 307
Rehosting ... 743, 904
Release ... 1078, 1085
Planung ... 1082
Prozess ... 1079
Vorankündigung ... 1083
Zyklus ... 1079
Release Candidate ... 1084
Release Management ... 1078
Release Notes ... 1085
Release-Management ... 1018
reliableSession ... 470
Remove ... 724
Remove<T> ... 724
RemoveFromCollection<T>-Aktivität ... 776
replacement ... 667
ReplaceOnUpdate ... 716
Representational State Transfer ... 706
Reproduzierbarkeit von Fehlern ... 1013
Request/Reply ... 538, 735
RequiredArgument ... 812
Requirements ... 1088
Requirements Management ... 50
RequiresNew ... 331
Reservierte Wörter ... 373
ResetAbort ... 253
ReSharper ... 415
Ressourcen ... 168
transaktionale ... 329
transaktionelle ... 556
Ressourcenkonflikte ... 568
REST ... 706
Resume ... 253
Review ... 993
RIA ... 42
Rich Client ... 42
Rich Internet Application ... 42
Roadmap ... 1078, 1083
Robustheit ... 133
Rollback ... 329
Rollenbasierte Sicherheit ... 596
rowversion ... 107
Running ... 251

S

SandCastle ... 390
SaveChanges ... 701, 714
SaveChangesOptions ... 716
ScheduleActivity ... 818, 838
Scheduler ... 248
ScheduleSample-Aktivität ... 875
Schema Language ... 680
Schicht → Layer
Schleifen, parallele ... 269
Schnittstellen ... 52, 158, 169
abstrahieren ... 420
Arten ... 175
asynchrone ... 173
benennen ... 375
bidirektionale ... 173
Empfehlungen ... 404
Kommunikationsweg ... 175
Mindeststandards ... 176
synchrone ... 173
unidirektionale ... 173
Schnittstellenanforderung ... 1089
Scrum ... 1083
SDM ... 1081
Sealed ... 395
Sections ... 205
Security Auditing ... 595
security mode ... 586
SecurityContext ... 593
SecurityException ... 598
Seiteneffekte ... 1018
Selfhosting ... 474
Semantischer Fehler ... 1016
SendingRequest ... 713
SendReply-Aktivität ... 885
SendReplyToReceive-Aktivität ... 883
Separation of Concerns ... 395
Sequence-Aktivität ... 754, 776
Sequenzdiagramm ... 124

Sequenzielle Workflows ... 741
Serialisierer ... 342
Serialisierung ... 339
Attribute ... 344
Serializable ... 332, 340, 531
SerializationException ... 343
Serialize ... 341, 343
Server-Modus ... 354
Service ... 42, 82, 176
Adresse ... 451
Binding ... 451
Contract ... 452
Identität ... 593
Überblick ... 434
WCF ... 437
Service Configuration Editor ... 465, 469
Service Contracts ... 463, 516
Hierarchien ... 522
Service Discovery ... 457
Service Level Agreement ... 1078
ServiceAuthorizationAuditLevel ... 596
ServiceBase ... 479
ServiceBehavior ... 545
ServiceContract ... 441, 452, 463, 862
serviceDebug ... 446, 549
serviceDiscovery ... 457
ServiceHost ... 453, 476, 870
serviceMetadata ... 448, 456
Serviceorientierte Architektur ... 115, 433, 857
serviceSecurityAudit ... 595
set ... 311
Settings ... 201
Settings-Klasse ... 202
SetValue ... 728, 810
ShouldExitCurrentIteration ... 271
Showstopper ... 1009
Sicherheit ... 91
In Schnittstellen ... 179
rollenbasierte ... 596
WCF ... 581
Sicherheitsanalyse ... 994
Sicherheitsgrenze ... 92
Sicherheitsmodell ... 124
Sicherheitsstufe ... 635
Sicherheitstest ... 1064
Sichtbarkeit
verändern ... 419
von Attributen ... 144
WCF-Methoden ... 520
Sichten
hinzufügen ... 661
Sign ... 589
Single Responsibility Principle ... 405
Single Sign On ... 93
Singleton ... 578
Instanziierung ... 570
Skalierbarkeit ... 90, 733
Skalierung
horizontale ... 90
vertikale ... 90
SLA ... 1078
Sleep ... 253
SnagIt ... 989
Snapped ... 980
Snippets ... 425
SOA ... 115, 857
SOAP Faults ... 550
SoC ... 395
Software
verteilte ... 76
Verteilung ... 1085
Weiterentwicklung ... 1077
Software Demand Manager ... 1017, 1081
Softwarearchitekt ... 43
Anforderungen ... 44
Merkmale ... 43
Rollen ... 43
Unterscheidung ... 43
Softwarearchitektur ... 41
Anforderungen ... 44
Anwendungstypen ... 41
Aufgaben ... 40
Die wichtigsten Fragen ... 80
Entscheidungen ... 39
gute und schlechte ... 39
Prozess ... 119
serviceorientierte ... 115, 433, 857
Softwarearchitekturgrenze ... 52
Softwarearchitekturmodell ... 114
Softwarearchitekturziele definieren ... 120
Softwaredesign ... 131
Ziele ... 132
Softwarekomponente ... 51
Arten ... 51
Beziehungen ... 53
identifizieren ... 51
UI ... 66
Vorgehensweise zur Bildung ... 52
Software-Metrik ... 994
Softwarepflege ... 1077
Softwareverteilung ... 103
Sortierung ... 653

Source ... 303
Speicherleaks ... 349
Speicherverwaltung, automatische ... 348
Sperre, logische ... 107
Spezialisierung/Generalisierung ... 151
Spezifikation ... 989
Spezifikationsfehler ... 1015
Spezifikationslücke ... 1012
Spezifikationstest ... 1048
SpinLock ... 285
Spinning ... 284, 285
Sporadischer Fehler ... 1014
SQL Server ... 627
  einrichten ... 839
SQL Server Data Tools ... 626
SQL-CLR
  vs. T-SQL ... 646
SqlContext ... 638
SqlFunction ... 639
SqlProcedure ... 636
SqlString ... 644
SqlTrigger ... 640
SqlUserDefinedAggregate ... 643
SqlWorkflowInstanceStore ... 839
SqlWorkflowInstanceStoreLogic ... 840
SqlWorkflowInstanceStoreSchema ... 840
SSDL ... 679
SSDT ... 626
SSO ... 93
StackTrace ... 300, 303
Staging-System ... 1086
Stammelement ... 350
Standard-Bindings ... 452
Standard-Endpunkte ... 452, 456
Standardkonstruktoren ... 399
START UPDATE POPULATION ... 661
StartNew ... 260
State Machine Workflows ... 740
State-Aktivität ... 895
State-Machine-Workflows ... 893
Statische Codeanalyse ... 420
Statische Klasse ... 398
Statischer Test ... 993
Statusinformationen ... 98
Stilistische Analyse ... 994
Stop ... 271
Stopped ... 251
Stoppwörter ... 665
StopRequested ... 252
Storage Model ... 679
Storage Schema Definition Language ... 679
Stored Procedure ... 636
Stresstest ... 1065
String.Replace ... 308
String.Split ... 308
Struktogramm ... 1042
Styleguide ... 186
Suchen ... 308
Suchoptionen ... 315
Suffix ... 370
Suppress ... 331
suppressAuditFailure ... 595
SuppressFinalize ... 358
Surface ... 913
Suspend ... 253
Suspended ... 252
SuspendRequested ... 252
SuspensionManager ... 963
Svcutil.exe ... 502
Switch-<T>-Aktivität ... 758
Synchrone Schnittstelle ... 173
Synchronization Pattern ... 778
Syntactic Sugar ... 275
Syntaxfehler ... 1016
System.Configuration ... 201
System.Exception ... 303
Systemanforderung ... 1089
Systemtest ... 993, 1057

T

Tabellen
  hinzufügen ... 661
Tabletsimulator ... 977
TargetSite ... 304
Task
  abbrechen ... 262
  erzeugen ... 259
  Hierarchien ... 266
  Kind- ... 266
  starten ... 259
Task Parallel Library ... 259
Task.Factory ... 260
Task.Run ... 261
TaskContinuationOptions ... 267
TaskCreationOptions ... 260
TaskFactory ... 262
TaskPool ... 270
TaskScheduler ... 268, 271
TDD ... 998, 1052
Technologieentscheidung ... 121
Teile-Ganzes-Beziehung ... 157

Terminate ... 643
TerminateWorkflow-Aktivität ... 799, 800
TerminationException ... 771
Test
Ablauf ... 1003
Abnahme- ... 1060
Arten ... 1048
Designer ... 1022
Durchführung ... 1007
dynamischer ... 994
Feld- ... 1058
Freigabe ... 1004
Komponenten- ... 1053
Leiter ... 1022
Organisation ... 1003, 1017
Planung ... 1004, 1017
Protokollierung ... 1007
Rechner ... 1032
sinnlose Daten ... 1039
Spezifikations- ... 1048
statischer ... 993
System- ... 1057
Umgebung ... 1028
Umgebung einrichten ... 1006
Unit- ... 1066
Usability- ... 1054
Verfahren ... 1034
Werkzeuge ... 1034
Zeitpunkt ... 996
Ziele ... 989
Test Driven Design ... 1052
Test Driven Development ... 998
Testbarkeit ... 366
TestClass ... 1069
Testdaten ... 1031
Testen, exploratives ... 1035
Tester ... 999, 1022
Schulung ... 1005
Testfall ... 1024
Priorisierung ... 1026
Testfreigabe ... 1084
Testlink ... 988
TestMethod ... 1069
Testreife ... 1004
Testteam ... 1019
Organisationsformen ... 1020
Rollen ... 1021
test-to-fail ... 1001, 1035
test-to-pass ... 1000, 1035
Thesaurus ... 666
Thread ... 247, 250
abbrechen ... 252
Eltern- ... 266
starten ... 250
Status ... 251
Synchronisierung ... 281
unterbrechen ... 253
warten auf ... 254
Thread Pool ... 257
ThreadAbortException ... 252
Threadsichere Collections ... 288
ThreadStart ... 251
ThreadState ... 251
ThreadStatic ... 243
Throttling ... 245, 579
throw ... 300
Throw-Aktivität ... 785
ThrowIfCancellationRequested ... 263
ThrowNewException ... 236
Tier ... 63, 76
Abgrenzung ... 63
Timeout ... 331
Timeout.Infinite ... 258
TimeoutException ... 553
Timer ... 258
TimerCallback ... 258
Top-down-Ansatz ... 52
TPL ... 259
Tracing ... 334, 843, 854
Tracing-Level ... 844
Tracking ... 843
Objekte ... 846
Profile ... 844
TrackingParticipant ... 848
TrackingQuery ... 845
TrackingRecord ... 844
Transaction.Current ... 330
TransactionAutoComplete ... 564
transactionFlow ... 560, 561
TransactionFlowOption ... 561
TransactionScope ... 329, 330, 331
geschachtelter ... 331
TransactionScope-Aktivität ... 829
TransactionScopeRequired ... 562
Transaktion ... 106, 108, 327, 554, 827
Ambient- ... 329
Konzept ... 555
leichtgewichtige ... 329
übermitteln ... 560
verteilte ... 555
WCF ... 560

Transaktionelle Ressourcen ... 329, 556
Transaktionsmanager ... 328, 558
Transaktionsprotokolle ... 560
Transaktionsreplikation ... 611
Transitions (Workflow) ... 896
Transparenz ... 736
TransportCredentialOnly ... 591
Transportprotokoll ... 460
Transportsitzung ... 470
TransportWithMessageCredential ... 591
Tray Notification Area ... 215
Trigger ... 639
TriggerAction ... 641
try ... 293, 409
TryAdd ... 288, 290
TryCatch-Aktivität ... 784
TryEnter ... 286, 287
Try-Methoden ... 410
TryTake ... 288
T-SQL ... 646
vs. SQL-CLR ... 646
T-SQL-Prozedur ... 636
TypeScript ... 137

U

udpDiscoveryEndpoint ... 457
UI-Komponente ... 66
UI-Layer ... 64
Umfeld analysieren ... 120
UML ... 123, 143, 1040
Unbehandelte Exceptions ... 301
Unbewertete Function Points ... 1119
Unboxing ... 393
Unescape ... 310
UnhandledException ... 771
UnhandledExceptionAction ... 771
Unidirektionale Schnittstelle ... 173
Uniform Resource Identifier ... 459
Units of Work ... 1100
Unit-Test ... 1050
Unit-Tests ... 411
Workshop ... 1066
Unity ... 166
Unload ... 842
Unsicherheit in der Zeitschätzung ... 1102
Unsicherheitskorridor ... 1099
Unstarted ... 251
UpdateObject ... 715
URI ... 459
Usability ... 195
Usability-Test ... 196, 1054
Use-Case-Diagramm ... 123
user.config ... 203
User-Mode-Objekte ... 283
using ... 303

V

Validierung ... 94, 312
Variablen ... 379
Custom Activity ... 816
Velocity ... 84, 495
Verarbeitung
asynchrone ... 242
parallele ... 241, 778
Verbesserungsvorschlag ... 1013
Vererbungshierarchie ... 152
Verhaltensanalyse ... 1055
Verhaltensdiagramme ... 123
Verschlüsselung ... 91
WCF ... 585
Version Tolerant Serialization ... 345
Versionierung ... 107, 345, 534
der Konfiguration ... 198
gleichzeitige Bearbeitung ... 107
unzureichende ... 1018
Versionshistorie ... 1078
Versionsnummer ... 1083
Verteilte Software ... 76
Ebenen ... 79
Empfehlungen ... 78
Verteilte Transaktion ... 328, 555
Verteilungsmodell ... 124
Vertikale Skalierung ... 90
Verzeichnis
benennen und anlegen ... 377
View ... 65
Virtual ... 396
Visual Studio
Kommentare ... 389
UML-Diagramme ... 146
VisualStateManager ... 983
Volltext-Filter-Dämon ... 659
Volltextkatalog anlegen ... 659
Volltext-Stopplisten ... 666
Volltextsuche ... 656
Vordergrund-Thread ... 254
Vorgängerversion ... 1033
VTS ... 345

W

Wait (Monitor) ... 287
WaitCallback ... 257
WaitForPendingFinalizer ... 354
WaitSleepJoin ... 252, 253
Walkthrough ... 1063
Wartbarkeit ... 133, 162, 365, 733
Wartezeit ... 214
WAS ... 490
Wasserfallmodell ... 996
WCF ... 433
  Authentifizierung ... 582
  Autorisierung ... 583, 595
  Clients ... 446, 468, 499
  erweitern ... 598
  Exception Handling ... 509
  Fehlerbehandlung ... 546
  Identität ... 592
  Instanzen ... 569
  Konfiguration ... 464
  Methoden überladen ... 521
  mit WF ... 857
  Services ... 437
  Sicherheit ... 581
  Transaktionen ... 560
  Transportsitzung ... 470
  Verschlüsselung ... 585
WCF Data Services ... 704
  erstellen ... 706
  testen ... 708
WCF Diensthost ... 473
WCF-Methoden, Sichtbarkeit ... 520
WcfSvcHost ... 473, 508
WcfSvcHost.exe ... 473
WCF-Testclient ... 447, 473
WcfTestClient.exe ... 473
Weasle-Words ... 373
Web Services Description Language ... 116
web.config ... 445
Webanwendung ... 42
Web-API ... 705
Weboberfläche ... 183
Weiterentwicklung von Software ... 1077
WF ... 731
  Fallbeispiel ... 751
  Fehlerbehandlung ... 783
  Integration mit WCF ... 750, 857
While-Aktivität ... 789
Whitebox ... 1034
Wiederverwendbarkeit ... 736
Windows 8 ... 913
Windows Activation Services ... 490
Windows Application Model ... 923
Windows Communication Foundation → WCF
Windows Presentation Foundation ... 181
Windows Runtime →WinRT
Windows-8-GUI ... 917
Windows-Ereignisanzeige ... 854
WindowsIdentity ... 593
WindowsImpersonationContext ... 594
WinForms ... 182
WinRT ... 915
Worker-Prozess ... 488
Workflow ... 67, 732, 738
  Arten ... 740
  ausführen ... 768
  Debugging ... 767
  Designer ... 742
  Flowchart- ... 741, 763
  gestalten ... 754
  Instanz ... 738
  laden ... 768
  Laufzeitumgebung ... 748
  Runtime ... 754
  sequenzielle ... 741
  State Machine ... 740
Workflow Foundation → WF
Workflow Service ... 857
WorkflowApplication ... 770, 842
WorkflowApplicationAbortedEventArgs ... 771
WorkflowApplicationUnhandled-
  ExceptionEventArgs ... 771
WorkflowInstance ... 846
WorkflowInstanceAbortedRecord ... 846
WorkflowInstanceQuery ... 845
WorkflowInstanceRecord ... 846
WorkflowInstanceSuspendedRecord ... 846
WorkflowInstanceTerminatedRecord ... 846
WorkflowInstanceUnhandledException-
  Record ... 846
WorkflowInvoker ... 769
Workstation-Modus ... 354
Wortgrenzen ... 314
WPF ... 181, 182
WriteLine ... 757
WS-AT ... 560
WsAtomicTransaction ... 560
WS-BasicProfile ... 461
WSDL ... 116
wsDualHttpBinding ... 462
wsHttpBinding ... 462, 591

X

XAML ... 750
XAttribute ... 720, 721
XCData ... 721
XComment ... 720, 721
XDeclaration ... 720, 721
XDocument ... 720, 721
XElement ... 720, 721
XML
Dateien laden ... 724
Daten verändern ... 727
in der Datenbank ... 647
XmlAttribute ... 347
XmlDocument ... 719
XmlElement ... 347, 719
XmlEnum ... 347
XmlIgnore ... 347
XmlRoot ... 347
XmlSerializer ... 342, 345
XML-Server ... 175
XmlWriterSettings ... 346
XName ... 721, 724
XNamespace ... 721
XNode ... 721
XProcessingInstruction ... 721
XQuery ... 652
XStreamingElement ... 721

Z

Zeichenauswahl ... 311
Zeitkorridor ... 1102
Zeitschätzung ... 1096
algorithmische ... 1122
empirische ... 1122
komparative ... 1122
Methoden ... 1112
Unsicherheit ... 1102
Zeitüberschreitungen ... 1108
Zeitwahrnehmung ... 212
Zero Trouble Forecast ... 1109
Zugriffsverletzungen ... 349
Zustandsdiagramm ... 124
Zustandsübergang (Workflow) ... 899
Zuverlässigkeit ... 133
Zuverlässigkeitstest ... 1065