

Kapitel 9

.NET-Klassenbibliothek 4.0

In diesem Kapitel:

Einleitung	408
System	408
System.Collections	420
System.IO	421
System.Configuration	429
System.Diagnostics	435
System.Net	440
System.Text	446
Serialisierung	448
System.DirectoryServices	464
System.Management	473
System.Resources	475
Managed Extensibility Framework (MEF)	478
System.Security	492
System.Runtime.Caching	499

Einleitung

Die .NET-Klassenbibliothek Version 4.0 enthält mehr als 10.000 öffentliche Klassen, die Sie in Ihren Anwendungen nutzen können. Im Rahmen dieses Crashkurses ist es absolut unmöglich, alle .NET-Klassen auch nur zu erwähnen. Dieses Kapitel enthält eine Auswahl von Klassen aus unterschiedlichen Einsatzgebieten und mit unterschiedlichen Vorgehensweisen. Dabei erhielten im Zweifel solche Klassen den Vorzug, die in den letzten Versionen neu hinzugekommen sind. Eine komplette Online-Referenz der .NET-Klassen und ihrer Verfügbarkeit in den verschiedenen .NET-Versionen finden Sie unter [DOTNET03]. Ausgewählte Klassen sind auch beschrieben in dem Buch [HSFE01].

WICHTIG Auch Windows Forms, WPF, Webforms, Webservices, Remoting, WCF, WF, ADO.NET, LINQ, Entity Framework und XML sind Teile der .NET-Klassenbibliothek. Aufgrund ihrer zentralen Bedeutung sind den zugehörigen Klassen jedoch eigene Kapitel in diesem Buch zugeordnet.

HINWEIS Soweit nicht anders erwähnt, finden Sie den Quellcode zu den Beispielen in diesem Kapitel im Projekt *WWWings_VerschiedeneDemos*.

System

Der Namensraum System enthält einige wesentliche Klassen, die für das Funktionieren des .NET Framework unabdingbar sind:

- System.Object, die Mutter aller Klassen im .NET Framework
- System.ValueType, die Basisklasse für alle Wertetypen im .NET Framework
- Klassen für elementare Datentypen wie Byte, SByte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Char, String, Decimal und Boolean
- Klassen für zusammengesetzte Datentypen wie DateTime, TimeSpan, Array, Enum, Uri, Version und Guid
- System.DBNull zur Repräsentation von null-Werten in Datenbanken
- Klassen zur Umwandlung von Datentypen: Converter und BitConverter
- System.Random zur Generierung von Zufallszahlen
- System.Math für mathematische Operationen
- Die Klasse System.Console für die Interaktion mit dem Befehlsfenster
- Die Klasse System.Environment mit Informationen über das Betriebssystem und die aktuelle Anwendung
- Die Klasse System.GC zur Einflussnahme auf die automatische Speicherverwaltung (Garbage Collector)
- Vorlagen-Klassen zur Erzeugung typischer .NET-Konstrukte wie Ausnahmen (System.Exception), Annotationen (System.Attribute) und Ereignisparametern (System.EventArgs)
- Für das Remoting wichtige Klassen wie SerializableAttribute, AppDomain, Activator und MarshalByRefObject (siehe Kapitel zu .NET Remoting,¹ Webservices¹ und Windows Communication Foundation).

¹ dieses Zusatzkapitel können Sie als PDF auf dem Leser-Portal herunterladen

System.Object

Jede Klasse im .NET Framework erbt direkt oder indirekt von der Klasse `System.Object`. Jede Klasse, die nicht explizit von einer anderen Klasse erbt, erbt implizit von `System.Object`. Daher bildet `System.Object` die Wurzel der Vererbungshierarchie im .NET Framework. `System.Object` kann deshalb für das späte Binden eingesetzt werden. `System.Object` enthält sieben Methoden, die sich in drei Gruppen gliedern lassen:

- Mitglieder, die eine Standardfunktionalität für jede .NET-Klasse bereitstellen und die von erbbenden Klassen überschrieben werden können:
 - `ToString()` liefert eine Repräsentation des Objekts in Form einer Zeichenkette
 - `GetHashCode()` liefert eine Ganzzahlrepräsentation des Objekts
 - `Equals()` prüft, ob zwei Objektverweise auf dasselbe Objekt zeigen (Referenzidentität)
 - `GetType()` liefert Informationen über den Typ des Objekts in Form einer Instanz von `System.Type`
- Mitglieder, die nur innerhalb von erbbenden Klassen aufgerufen werden können:
 - `Finalize()` wird aufgerufen für Aufräumarbeiten im Rahmen der automatischen Speicherverwaltung
 - `MemberwiseClone()` erstellt eine flache Kopie eines Objekts
- Statische Mitglieder, die nur auf `System.Object` selbst aufgerufen werden dürfen:
 - `ReferenceEquals()` vergleicht zwei Objektverweise auf Referenzidentität

ToString()

Bei elementaren Datentypen liefert `ToString()` den Wert als Zeichenkette. Dabei kann der Entwickler durch optionale Formatierungskürzel die Darstellung bestimmen, z.B.: `x.ToString("C")` gibt die in der Variablen `x` enthaltene Zahl im Währungsformat aus.

Wichtige Kürzel sind:

- `C`: Währung
- `D`: Ganzzahl
- `E`: Zahl in Exponentialschreibweise
- `F`: Zahl in Festpunktformat
- `P`: Prozentzahl
- `X`: Hexadezimalzahl

TIPP Diese Kürzel werden auch von `Console.WriteLine` unterstützt:

```
Console.WriteLine("{0} kostet {1:C}!", Product, Preis);
```

Symbol	Beschreibung	Beispiel	Ergebnis
c	Währung	{0:c} für 10000000	1.000.000,00 €
d	Dezimalzahl	{0:d} für 10000000	1000000
e	Wissenschaftlich	{0:e} für 10000000	1,000000e+006
f	Festkommazahl	{0:f} für 10000000	1000000,00
g	Generisch	{0:g} für 10000000	1000000
n	Tausender Trennzeichen	{0:n} für 10000000	1.000.000,00
x	Hexadezimal	{0:x4} für 10000000	f4240
d	kurzes Datumsformat	{0:d} für "1.22.2010 00:44:15"	22.01.2010
D	langes Datumsformat	{0:D} für "1.22.2010 00:44:15"	Freitag, 22. Januar 2010
t	kurzes Zeitformat	{0:t} für "1.22.2010 00:44:15"	00:44
T	langes Zeitformat	{0:T} für "1.22.2010 00:44:15"	00:44:15
f	Datum und Uhrzeit komplett (kurz)	{0:f} für "1.22.2010 00:44:15"	Freitag, 22. Januar 2010 00:44
F	Datum und Uhrzeit komplett (lang)	{0:F} für "1.22.2010 00:44:15"	Freitag, 22. Januar 2010 00:44:15
g	Standard-Datum (kurz)	{0:g} für "1.22.2010 00:44:15"	22.01.2010 00:44
G	Standard-Datum (lang)	{0:G} für "1.22.2010 00:44:15"	22.01.2010 00:44:15
M	Tag des Monats	{0:M} für "1.22.2010 00:44:15"	22 Januar
Y	Monat und Jahr	{0:Y} für "1.22.2010 00:44:15"	Januar 2010
r	Datumsformat nach RFC1123	{0:r} für "1.22.2010 00:44:15"	Fri, 22 Jan 2010 00:44:15 GMT
s	sortierbares Datumsformat	{0:s} für "1.22.2010 00:44:15"	2010-01-22T00:44:15
u	universell sortierbares Datumsformat	{0:u} für "1.22.2010 00:44:15"	2010-01-22 00:44:15Z
U	universell sortierbares GMT-Datumsformat	{0:U} für "1.22.2010 00:44:15"	Donnerstag, 21. Januar 2010 23:44:15

Tabelle 9.1 Vordefinierte Formatierungen

Symbol	Beschreibung	Aufruf	Ergebnis
0	0-Platzhalter	{0:00.0000} für 10000000	1000000,0000
#	Zahl-Platzhalter	{0:({#}).##} für 10000000	(1000000)
.	Dezimalpunkt	{0:0.0} für 10000000	1000000,0
,	Tausender-Trennzeichen	{0:0,0} für 10000000	1.000.000
,.	Ganzzahliges Vielfaches von 1.000	{0:0,.} für 10000000	1000
%	Prozentwert	{0:0%} für 10000000	100000000%
e	Exponenten-Platzhalter	{0:00e+0} für 10000000	10e+5
dd	Tag	{0:dd} für "1.22.2010 00:44:15"	22 

Symbol	Beschreibung	Aufruf	Ergebnis
ddd	Tagname (kurz)	{0:ddd} für "1.22.2010 00:44:15"	Fr
dddd	Tagname (lang)	{0:dddd} für "1.22.2010 00:44:15"	Freitag
gg	Zeitalter	{0:gg} für "1.22.2010 00:44:15"	n.Chr.
hh	Stunde 2stellig	{0:hh} für "1.22.2010 00:44:15"	12
HH	Stunde 2stellig (24-Stunden)	{0:HH} für "1.22.2010 00:44:15"	00
mm	Minute	{0:mm} für "1.22.2010 00:44:15"	44
MM	Monat	{0:MM} für "1.22.2010 00:44:15"	01
MMM	Monatsname (Kürzel)	{0:MMM} für "1.22.2010 00:44:15"	Jan
MMMM	Monatsname (ausgeschrieben)	{0:MMMM} für "1.22.2010 00:44:15"	Januar
ss	Sekunde	{0:ss} für "1.22.2010 00:44:15"	15
yy	Jahr 2stellig	{0:yy} für "1.22.2010 00:44:15"	10
yyyy	Jahr 4stellig	{0:YY} für "1.22.2010 00:44:15"	2010
zz	Zeitzone (kurz)	{0:zz} für "1.22.2010 00:44:15"	+01
zzz	Zeitzone (lang)	{0:zzz} für "1.22.2010 00:44:15"	+01:00

Tabelle 9.2 Individualformatierungen

Objektidentität

Die Standardimplementierung von `ToString()` liefert den Namen der Klasse als Zeichenkette und `GetHashCode()` eine fortlaufende Nummer. Wenn Sie ein Objekt anhand einer Zeichenkette oder einer Zahl identifizieren wollen, sind beide Standardimplementierungen ungeeignet. Das folgende Beispiel zeigt eine Überschreibung der beiden Methoden mit einer Wertesemantik, d.h., `ToString()` und `GetHashCode()` liefern einen eindeutigen Wert für jede andere Kunden-ID (siehe im folgenden Beispiel: 0815 und 4711), aber gleiche Werte für gleiche Kunden-IDs (4711). Für jede Kunden-ID eine eindeutige Zeichenkette zu generieren, ist nicht schwer. Als eindeutige Zahl hätte die Kunden-ID verwendet werden können, wenn eine Eindeutigkeit innerhalb der Instanz dieser Klasse hinreichend ist. Für die klassenübergreifende Eindeutigkeit wird als Trick der Hashcode der in `ToString()` erzeugten Ausgabe, die den Klassennamen und die Kunden-ID enthält, verwendet. In der Klasse `System.String` hat Microsoft bereits eine Implementierung von `GetHashCode()` hinterlegt, die für zwei gleiche Zeichenketten den gleichen Hash-Wert liefert.

```
public class Kunde
{
    public int KundenNr;
    public string Name;
    public override string ToString()
    {
        return "Kunde.ID=" + KundenNr;
    }

    public override int GetHashCode()
```

```

{
    return this.ToString().GetHashCode();
}

public Kunde() { }
public Kunde(int KundenNr, string Name)
{
    this.KundenNr = KundenNr;
    this.Name = Name;
    Demo.Out("Kunde angelegt: Nr=" + KundenNr + " Name=" + Name);
}
}

```

Listing 9.1 Implementierung der Klasse *Kunde* [SystemObject.cs]

```

Kunde k1 = new Kunde(0815, "Meier");
Demo.Out("Dieser Kunde als Zeichenkette:" + k1.ToString());
Demo.Out("Dieser Kunde als Ganzzahl:" + k1.GetHashCode());

Kunde k2 = new Kunde(4711, "Müller");
Demo.Out("Dieser Kunde als Zeichenkette:" + k2.ToString());
Demo.Out("Dieser Kunde als Ganzzahl:" + k2.GetHashCode());

Kunde k3 = new Kunde(4711, "Müller");
Demo.Out("Dieser Kunde als Zeichenkette:" + k3.ToString());
Demo.Out("Dieser Kunde als Ganzzahl:" + k3.GetHashCode());

```

Listing 9.2 Testcode der Klasse *Kunde*

```

Kunde angelegt: Nr=815 Name=Meier
Dieser Kunde als Zeichenkette:Kunde.ID=815
Dieser Kunde als Ganzzahl:726408994
Kunde angelegt: Nr=4711 Name=Müller
Dieser Kunde als Zeichenkette:Kunde.ID=4711
Dieser Kunde als Ganzzahl:733982475
Kunde angelegt: Nr=4711 Name=Müller
Dieser Kunde als Zeichenkette:Kunde.ID=4711
Dieser Kunde als Ganzzahl:733982475

```

Listing 9.3 Ausgabe des Testcodes

System.Console

Diese zur Erstellung von Konsolenanwendungen wichtige Klasse wurde bereits in Kapitel 7 ausführlich vorgestellt.

System.Type

Die Frage nach ihrem Typ konnte eine Variable im *Component Object Model* (COM) aufgrund eines unzureichenden Bewusstseins ihrer selbst nur sehr pauschal mit ihrem Typnamen beantworten. Detail-

informationen über seine Fähigkeiten waren von einem COM-Objekt nicht direkt zu bekommen, sondern allenfalls indirekt über einen Zugriff auf die Typbibliothek, sofern eine solche überhaupt vorhanden war.

Im .NET Framework sind Metadaten Pflichtbestandteile einer jeden Komponente und Anwendung. Die Metadaten sind über den so genannten Reflection-Mechanismus innerhalb und außerhalb der Assembly auslesbar. Die Implementierung einer Methode `GetType()` in der Klasse `System.Object` sorgt dafür, dass jedes .NET-Objekt diese Methode besitzt und ein passendes Objekt vom Typ `System.Type` liefert, das Metainformationen über sich selbst enthält.

Wege zum Type-Objekt

Eine Instanz von `System.Type` erhält man auf verschiedenen Wegen, u.a. über die Methode `GetType()` eines jeden Objekts:

```
System.Data.DataSet d = new System.Data.DataSet();
System.Type t = d.GetType();
```

oder über die gleichnamige statische Methode in der Klasse `System.Type`:

```
t = System.Type.GetType("System.Data.DataSet");
```

Im zweiten Fall ist der Name des Typs einschließlich Namensraum in der korrekten Groß-/Kleinschreibung anzugeben. Kann der Typ nicht gefunden werden, liefert die Methode `Nothing` bzw. `null` zurück. Durch optionale Parameter kann gesteuert werden, ob stattdessen eine Ausnahme erzeugt und ob die Groß-/Kleinschreibung ignoriert werden soll.

`System.Type.GetType()` sucht nur in den geladenen Assemblys. Typen in anderen Assemblys findet man, indem man sich zunächst eine passende Instanz von `System.Reflection.Assembly` verschafft und dann `GetType()` auf dem `Assembly`-Objekt aufruft.

Auch für COM-Objekte kann man `Type`-Objekte erhalten, indem man `GetTypeFromCLSID()` oder `GetTypeFromProgID()` aufruft.

Informationen im Type-Objekt

Ein `Type`-Objekt liefert zahlreiche Informationen, beispielsweise die Namen und Namensbestandteile. Dabei erkennt man auch, dass jeder Typ einen GUID besitzt für den Fall, dass die Klasse über die COM-Interoperabilität genutzt werden soll.

```
Console.WriteLine(t.FullName);
Console.WriteLine(t.Name);
Console.WriteLine(t.Namespace);
Console.WriteLine(t.GUID);
```

Das Attribut `AssemblyQualifiedName` liefert den vollständigen Namen der Assembly, Assembly das zugehörige Assembly-Objekt. Den Namen der Basisklasse des Typs erhält man über `BaseType`. Zur Erinnerung: Jede Klasse im .NET Framework (außer `System.Object`) besitzt genau eine Oberklasse. Für Schnittstellen bleibt `BaseType` immer leer. `System.Type` verfügt über eine große Menge von `Boolean`-Attributen, die Auskunft über Eigenschaften des Typs geben. Bestimmbar sind zum Beispiel auf diese Weise die Art des Typs,

```
Console.WriteLine(t.IsClass);  
Console.WriteLine(t.IsInterface);  
Console.WriteLine(t.IsEnum);  
Console.WriteLine(t.IsPointer);  
Console.WriteLine(t.IsPrimitive);  
Console.WriteLine(t.IsValueType);  
Console.WriteLine(t.IsArray);
```

die Form der Übergabe,

```
Console.WriteLine(t.IsByRef);  
Console.WriteLine(t.IsMarshalByRef);
```

die Nutzbarkeit von COM aus,

```
Console.WriteLine(t.IsCOMObject);
```

die Sichtbarkeit

```
Console.WriteLine(t.IsPublic);  
Console.WriteLine(t.IsNotPublic);
```

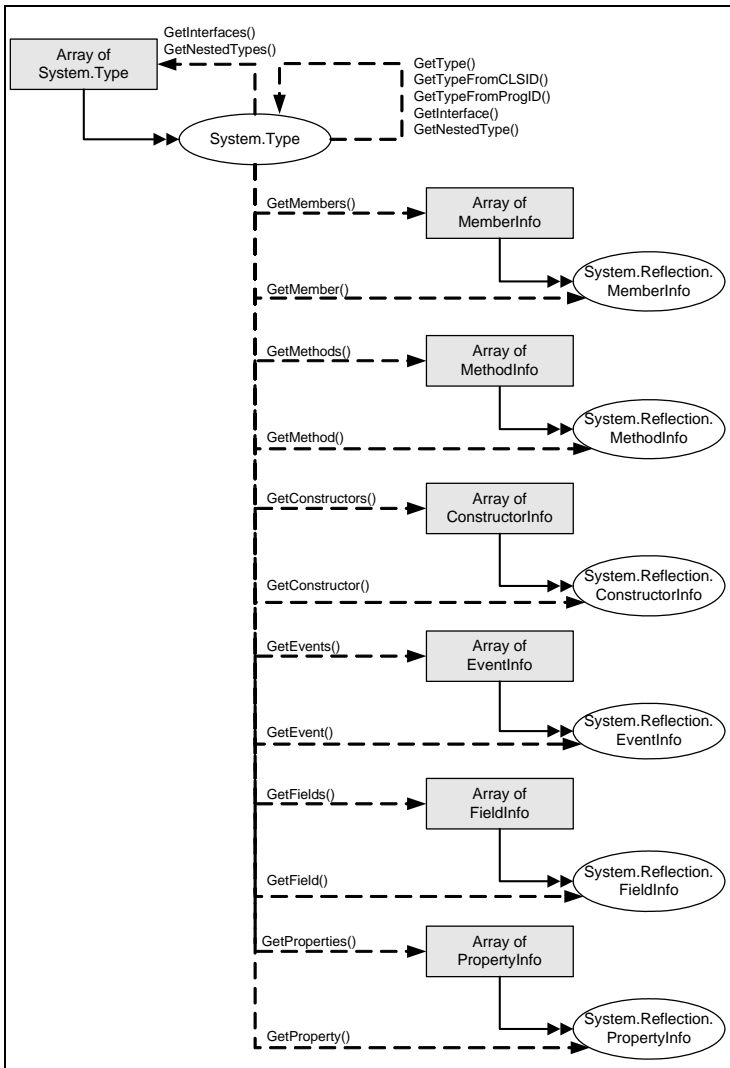
und die Serialisierbarkeit.

```
Console.WriteLine(t.IsSerializable);
```

Bei `IsAssignableFrom()` kann man einen zweiten Typ angeben und prüfen, ob sich eine Instanz des zweiten Typs an den ersten zuweisen lässt.

Mitglieder ermitteln

Die Definition des Typs kann man über das von `System.Type` ausgehende Objektmodell erforschen, das in `System.Reflection` implementiert ist. `Type` implementiert zahlreiche Methoden (`GetMethods()`, `GetProperties()`, `GetConstructors()` etc.), die Mengen mit den einzelnen Mitgliedern des Typs liefern. `GetMembers()` liefert eine übergreifende Liste aller Mitglieder.



Elementare Datentypen

Der Namensraum System enthält die Klassen für elementare Datentypen wie Byte, SByte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Char, String, Decimal und Boolean. Diese Klassen werden in den .NET-Sprachen oftmals durch eigene Schlüsselwörter repräsentiert, z. B. entspricht Int32 dem Schlüsselwort *int* in C# und Integer in Visual Basic.

Es gibt wenig Gemeinsamkeiten dieser Klasse: Die Zahlenklasse unterstützen alle `MinValue` und `MaxValue`, um den Wertebereich auszulesen. Einige Datentypen erlauben die Extraktion des jeweiligen Datentyps aus einer Zeichenkette mit `Parse()` bzw. `TryParse()`.

Typumwandlungen mit der TryParse-Methode

Die Parse()-Methode ermöglichte im .NET Framework 1.x die Umwandlung einer Zeichenkette in einen elementaren Datentyp, z. B. Int32, Double, String, Byte und DateTime. Einzig die Klasse System.Double besaß eine Methode TryParse(), mit der man vorab prüfen konnte, ob eine Umwandlung überhaupt möglich ist. Bei allen anderen Umwandlungen ergab die Umwandlung immer eine (zeitaufwendige) Ausnahme, wenn sie nicht möglich war. Seit .NET Version 2.0 besitzen alle elementaren Datentypklassen eine Methode TryParse(). Die Verwendung von TryParse() bringt in der Regel Geschwindigkeitsvorteile.

In dem folgenden Beispiel ist der Aufruf von Convert.ToInt32() oder Int32.Parse() risikoreich: Wenn der Eingabewert keine reine Zahl ist, kommt es zum Laufzeitfehler *Input string was not in a correct format*. TryParse() ist hier wesentlich besser.

```
string ZahlAsText = "1872";
Int32 zahl1;
Int32 zahl2;
Int32 zahl3;

// Laufzeitfehler, wenn keine reine Zahl!
zahl1 = System.Convert.ToInt32(ZahlAsText);
Console.WriteLine(zahl1 + 1);
zahl2 = System.Int32.Parse(ZahlAsText);
Console.WriteLine(zahl2 + 1);
// Gefahrlos:
if (Int32.TryParse(ZahlAsText, out zahl3))
{
    Console.WriteLine(zahl3 + 1);
}
```

Listing 9.4 TryParse() versus Parse() und Convert-Klasse

TIPP

Ab .NET 4.0 gibt es TryParse() auch für die Datentypen System.Enum, System.Guid und System.Version.

Datum und Uhrzeit

Seit dem .NET Framework 1.0 gibt es bereits die Klasse DateTime. DateTime repräsentiert ein Datum und eine Uhrzeit. Intern wird der Zeitpunkt als Anzahl der Einheiten in 100 Nanosekunden seit dem 1.1.0001 00:00 Uhr gespeichert. Die Zählung reicht bis zum 31.12.9999. Ein neues Jahrtausendproblem ist mit .NET-Anwendungen also vorerst nicht zu erwarten.

DateTime.Now liefert das aktuelle Datum mit der aktuellen Uhrzeit (gemäß der lokalen Systemuhr). DateTime besitzt zahlreiche Attribute, um Einzelangaben aus der Datenstruktur herauszuziehen: Date, Time, Day, Month, Year, Hour, Minute, Second, Millisecond, DayOfWeek, DayOfYear. Außerdem gibt es Rechenoperationen: AddYear(), AddMonth(), AddDays(), AddHours(), AddMinutes(), AddSeconds(). Bei allen Methoden kann man auch negative Werte angeben. Alternativ kann man bei Add() und Subtract() ein TimeSpan-Objekt übergeben, das einen Zeitraum definiert. Dies kann einfacher sein, z. B.:

```
// Welcher Wochentag ist in 5 Tagen, 4 Stunden, 3 Minuten, 2 Sekunden?  
Console.WriteLine(  
    DateTime.Now.AddDays(5).AddHours(4).AddMinutes(2).AddSeconds(1).DayOfWeek);  
// gleichbedeutend:  
Console.WriteLine(DateTime.Now.Add(new TimeSpan(5, 4, 3, 2)).DayOfWeek);
```

Mit dem .NET Framework 2.0 Service Pack 1 (daher auch verfügbar in .NET 3.5, da dort .NET 2.0 SP1 enthalten ist, vgl. Kapitel 1 »Einführung«) ist eine alternative Möglichkeit zur Darstellung von Datum und Zeit eingeführt worden: `DateTimeOffset` liefert zusätzlich Informationen über die Zeitzone in Form der Differenz (Offset) zur koordinierten Weltzeit (UTC). Nur mit `DateTimeOffset` kann man einen weltweit eindeutigen Zeitpunkt festlegen. `DateTimeOffset` bietet die gleichen Attribute und Methoden wie `DateTime` und zusätzlich das Attribut `Offset`.

Befehl	Ausgabe (Beispiel)
<code>DateTime.Now</code>	20.04.2008 11:07:10
<code>DateTimeOffset.Now</code>	20.04.2008 11:07:10 +02:00

Tabelle 9.3 `DateTime` vs. `DateTimeOffset`

Zeitzone

Mit dem .NET Framework 2.0 wurde die Klasse `TimeZone` eingeführt, mit der man die aktuelle Zeitzone ermitteln kann und Informationen über diese Zeitzone (z.B. Name, Differenz zur koordinierten Weltzeit (UTC), Dauer der Sommerzeit) ermitteln kann. Allerdings kann man mit `TimeZone` immer nur die eigene Zeitzone betrachten. Informationen über beliebige Zeitzone liefert die Klasse `TimeZoneInfo`, die aber erst ab dem .NET Framework 3.5 zur Verfügung steht.

```
Console.WriteLine("Es ist jetzt: " + DateTime.Now);  
System.TimeZone t = System.TimeZone.CurrentTimeZone;  
Console.WriteLine("Zeitzone: " + t.StandardName);  
Console.WriteLine("Sommerzeit? " + t.IsDaylightSavingTime(DateTime.Now));  
Console.WriteLine("Sommerzeit von " + t.GetDaylightChanges(DateTime.Now.Year).Start + " bis " +  
    t.GetDaylightChanges(DateTime.Now.Year).End);  
Console.WriteLine("Differenz zu koordinierter Weltzeit / Greenwich Mean Time): " +  
    t.GetUtcOffset(DateTime.Now));
```

Listing 9.5 Beispiel für den Einsatz von `TimeZone`

Verzögertes Instanzieren mit `System.Lazy`

Gastautor dieses Abschnitts: Manfred Steyer, entnommen aus »`.NET 4.0 Update`« [HS07]

Mit der generischen Klasse `System.Lazy` besteht die Möglichkeit, die Erzeugung eines Objekts bis zu dessen erster Verwendung hinauszuzögern. Zur Demonstration beinhaltet Listing 9.6 eine Klasse `Atomkraftwerk`. Da die Investitionskosten für ein AKW bekanntlich hoch sind, soll die Instanziierung eines solchen nur dann erfolgen, wenn es wirklich benötigt wird. Aus diesem Grund wird in Listing 9.7 die Klasse `LazyAtomkraftwerk` herangezogen. Zunächst wird eine Instanz von `LazyAtomkraftwerk` erzeugt. Mit der Eigenschaft `IsValueCreated` wird geprüft, ob das AKW bereits instanziiert wurde. Da es davor noch keine Verwendung fand, wird `false` geliefert. Anschließend wird über `Value` auf das AKW zugegriffen und mittels `Start` das Hochfahren des

AKWs veranlasst. Dieser erstmalige Zugriff auf `Value` veranlasst die Instanziierung. Ab diesem Zeitpunkt liefert `Value` diese Instanz. Ein nochmaliges Abrufen von `IsValueCreated` liefert somit `true`.

```
class Atomkraftwerk
{
    public Atomkraftwerk()
    {
        Console.WriteLine("Konstruktor: AKW");
    }

    public int Id { get; set; }
    public void Start()
    {
        Console.WriteLine("AKW #{0} nimmt Betrieb auf ...", Id);
    }

    public void Stop()
    {
        Console.WriteLine("AKW {0} stoppt Betrieb ...", Id);
    }
}
```

Listing 9.6 Atomkraftwerk mit hohen (gedachten) Investitionskosten

```
Lazy<Atomkraftwerk> lazyInvestment = new Lazy<Atomkraftwerk>();
Console.WriteLine("IsValueCreated: " + lazyInvestment.IsValueCreated);
lazyInvestment.Value.Start();
Console.WriteLine("IsValueCreated: " + lazyInvestment.IsValueCreated);
```

Listing 9.7 Verzögerte Instanziierung eines Atomkraftwerks

Für Fälle, in denen vor der ersten Verwendung weitere Vorbereitungsaufgaben durchzuführen sind, bietet `Lazy` auch das Hinterlegen einer `Factory`-Methode in Form eines `Lambda`-Ausdrucks an. In Listing 9.8 kommt beispielsweise eine `Factory`-Methode, welche auch die `Id` des Atomkraftwerks vergibt, zum Einsatz.

```
Lazy<Atomkraftwerk> lazyInvestment2 = new Lazy<Atomkraftwerk>(() => {
    Atomkraftwerk akw = new Atomkraftwerk();
    akw.Id = 4711;
    return akw;
});

Console.WriteLine("IsValueCreated: " + lazyInvestment2.IsValueCreated);
lazyInvestment2.Value.Start();
Console.WriteLine("IsValueCreated: " + lazyInvestment2.IsValueCreated);
```

Listing 9.8 Verzögerte Instanziierung eines Atomkraftwerks unter Verwendung einer `Factory`-Methode

TIPP Auch das `Threading`-Verhalten kann über den Konstruktor von `Lazy` beeinflusst werden. Eine Überladung des Konstruktors erwartet dazu beispielsweise ein boolsches Argument `isThreadSafe`. Der Wert `true` bewirkt hier, dass mittels Sperren sichergestellt wird, dass das gewünschte Objekt nur ein einziges Mal erzeugt wird, auch wenn mehrere Threads gleichzeitig die erste Ausführung von `Value` anstoßen.

System.Tuple

Gastautor dieses Abschnitts: Manfred Steyer, entnommen aus ».NET 4.0 Update« [HS07]

Um zu verhindern, dass einzelne auf .NET basierende Sprachen das Konzept von Tupeln auf unterschiedliche Arten implementieren, wurde mit der Klasse `Tuple` (`System.Tuple`) eine einheitliche Tuple-Implementierung bereitgestellt. Ähnlich wie ein Array kann ein Tuple mehrere Werte aufnehmen. Anders als bei Arrays müssen diese jedoch nicht vom selben Typ sein. Die Typen dieser Werte werden als Typparameter angegeben. Da in Sprachen wie C# oder Visual Basic jedoch keine Möglichkeit für die Angabe einer variablen Anzahl von Typparametern vorgesehen ist, stehen u. a. sieben Implementierungen, welche mit ein bis sieben Typen parametrisiert werden können, zur Verfügung. Daneben existiert eine achte Implementierung, welche neben den soeben erwähnten sieben Werten einen weiteren Tuple, dessen Werte an den ursprünglichen angehängt werden, aufnimmt. Somit können auch längere Tuple abgebildet werden. Werden hingegen an die ersten sieben Elemente Tuple übergeben, so werden diese als Tuple im ursprünglichen Tuple behandelt und nicht an diesen angehängt. Auf die einzelnen Elemente wird über die Eigenschaften `Item1` bis `Item7` zugegriffen. Die Eigenschaft für den Zugriff auf das im achten Parameter angegebene Tuple nennt sich `Rest`.

Zur Erzeugung von Tuple wurde die statische Factory-Methode `Tuple.Create` eingerichtet. Werden zwei Tuple mit dem Vergleichsoperator (`==` in C#, `=` in Visual Basic) verglichen, so bezieht sich dieser Vergleich auf deren Speicheradressen. Ein Vergleich mittels `Equals` prüft hingegen, ob die beiden Tuple dieselben Werte in derselben Reihenfolge aufweisen. Dazu wird an die Implementierung von `Equals` der einzelnen Elemente delegiert.

Listing 9.9 demonstriert die Verwendung von Tuples. Dazu wird ein `Tuple<int,string,string,bool>` erzeugt und mit Werten belegt. Auf das zweite Element wird über die Eigenschaft `Item2` zugegriffen. Danach wird ein weiteres Tuple mit denselben Werten erzeugt. Der Vergleich mittels `Equals` liefert `true`, da beide Tuple dieselben Werte aufweisen; der Vergleich mit dem Vergleichsoperator liefert hingegen `false`, da es sich um zwei verschiedene Tuple handelt. Zur Demonstration der Simulation »längerer« Tuple wird anschließend ein Tuple mit 14 Einträgen angelegt, wobei sich die Einträge 8 bis 14 in einem verschachtelten Tuple, welcher an den achten Parameter von `Create` übergeben wird, befinden.

```
Tuple<int,string,string,bool> t1 = Tuple.Create(1, "Max", "Muster", true);
Console.WriteLine(t1.Item2);
var t2 = Tuple.Create(1, "Max", "Muster", true);
bool equals1 = t1.Equals(t2);
bool equals2 = (t1 == t2);

Console.WriteLine("equals1: " + equals1);
Console.WriteLine("equals2: " + equals2);

var nested = Tuple.Create(8, 9, 10, 11, 12, 13, 14);
var largeTuple = new Tuple<int,int,int,int,int,int,int,int,Tuple<int,int,int,int,int,int,int,int>(1, 2, 3, 4, 5, 6, 7, nested);
var ten = largeTuple.Rest.Item3;
```

Listing 9.9 Verwendung von Tuples

Ein weiteres Beispiel für die Verwendung von Tuples findet sich in Listing 9.10. Die Methode `Div` nimmt zwei Integer für eine Division entgegen. Das Ergebnis ist ein Tuple, welches sowohl das Ergebnis der Division als auch den Restwert beinhaltet.

```
static Tuple<int, int> Div(int a, int b)
{
    int r1 = a / b;
    int r2 = a % b;

    return Tuple.Create(r1, r2);
}
```

Listing 9.10 Rückgabe von mehreren Werten mittels Tuples

System.Collections

Fast jede Anwendung benötigt Datenstrukturen zur Speicherung von Objektmengen (alias Auflistungen oder Kollektionen). Das .NET Framework bietet hier zahlreiche Optionen an, die sich in zwei Gruppen unterteilen lassen:

- untypisierte Mengentypen
- typisierte Mengentypen (alias generische Mengentypen)

Generische Mengentypen sind neu seit .NET 2.0 und bieten gegenüber den untypisierten Mengentypen den Vorteil, dass eine generische Objektmenge bereits zur Entwicklungszeit auf einen bestimmten Inhaltstyp geprägt werden kann, sodass der Compiler schon feststellt, wenn der Menge Objekte falschen Typs hinzugefügt werden.

Das folgende Beispiel zeigt, dass der Compiler bei untypisierten Mengentypen nicht feststellt, wenn in eine Liste von Kunden versehentlich eine Instanz der Klasse *Lieferant* aufgenommen wird. Für den generischen Mengentyp akzeptiert der Compiler hingegen nur Instanzen der Klasse *Kunde* und von ihr abgeleitete Klassen (hier: *StammKunde*). Für die Nutzung generischer Klassen wurden die .NET-Sprachen ab .NET 2.0 um neue Konstrukte erweitert (vgl. Kapitel 6 »Sprachsyntax Visual Basic 2010 (VB.NET 10.0) und C# 2010 (C# 4.0)«).

```
// Untypisierter Mengentyp
System.Collections.Queue Kunden1 = new System.Collections.Queue();
Kunden1.Enqueue(new Kunde());
Kunden1.Enqueue(new StammKunde());
Kunden1.Enqueue(new Lieferant());

// Generischer Mengentyp
System.Collections.Generic.Queue<Kunde> Kunden2 = new System.Collections.Generic.Queue<Kunde>();
Kunden2.Enqueue(new Kunde());
Kunden2.Enqueue(new StammKunde());
// Compiler-Fehler: Kunden.Add(new Lieferant());
```

Listing 9.11 Verwendung von Objektmengen [System.Collections.cs]

Im obigen Beispiel wurde die Klasse *Queue* verwendet, die eine Warteschlange nach dem First-In-First-Out-Prinzip realisiert. Die nachfolgende Tabelle zeigt weitere Mengentypen, die seit .NET 2.0 bzw. .NET 3.5 zur Verfügung stehen. Bei generischen Mengentypen, die Attribut-Wert-Paare unterstützen, kann man auch für den Schlüssel einen eigenen Typ angeben. Die Namen der generischen Mengentypen sind leider nicht konsistent zu denen der untypisierten Mengentypen.

Mengentyp	Untypisiert (System.Collection)	Typisiert, generisch (System.Collection.Generic) Neu ab .NET 2.0
FIFO-Struktur (First-In-First-Out)	Queue	Queue<Typ>
LIFO-Struktur (Last-In-First-Out)	Stack	Stack<Typ>
Dynamische Menge für beliebige Objekte, Zugriff über Position, doppelte Elemente erlaubt	ArrayList	List<Typ>
Dynamische Menge für Bit-Werte	BitArray	–
Schlüssel-Wert-Paare (Zugriff nur per Schlüssel, keine doppelten Werte erlaubt)	HashTable	Dictionary<Schlüsseltyp,Werttyp>
Schlüssel-Wert-Paare (Zugriff per Schlüssel oder Index, keine doppelten Werte erlaubt)	SortedList	SortedList<Schlüsseltyp,Werttyp>
Doppelt verkettete Liste	–	LinkedList<Typ>
Schlüssel-Wert-Paare (Zugriff per Schlüssel oder Index, keine doppelten Werte erlaubt) mit speziellen Mengenoperationen (z. B. IntersectWith(), ExceptWith(), UnionWith() und IsSubsetOf())	–	HashSet<Typ> (Neu ab .NET 3.5)
Sortiertes HashSet	–	SortedSet<Typ> (Neu ab .NET 4.0)

Tabelle 9.4 Überblick über die Mengentypen in .NET 4.0

ACHTUNG Alle generischen Objektmengen sind nicht CLS-konform. Der Namensraum `System.Collections.ObjectModel` enthält Basisklassen zur Erstellung eigener generischer Objektmengen.

System.IO

Der Namensraum `System.IO` enthält Klassen zur Arbeit mit dem Dateisystem und mit Dateiinhalten.

Neuheiten seit .NET 2.0

Der Namensraum `System.IO` wurde in .NET 2.0 um folgende Funktionen erweitert:

- Zugriff auf Laufwerksinformationen: Klasse `DriveInfo`
- Vereinfachungen für das Lesen und Schreiben von Dateien durch Erweiterungen der Klasse `File` um Methoden wie `ReadAll()`, `ReadAllBytes()`, `ReadAllLines()`, `AppendText()`, `WriteLine()`, `WriteAllBytes()`, `WriteAllLines()` etc.
- Ermittlung der Pfade für besondere Verzeichnisse (z. B. *Desktop*, *Meine Dokumente*, *Meine Musik*): Klasse `SpecialDirectories`

- Auslesen und Verändern der Zugriffsrechtelisten (Access Control Lists (ACLs)) von Verzeichnissen und Dateien: Die Methoden `GetAccessControl()` und `SetAccessControl()` arbeiten mit Objekten aus dem neuen Namensraum `System.Security.AccessControl` zusammen.
- Komprimieren von Daten durch Klassen im Unternamensraum `System.IO.Compression`
- Der neue Unternamensraum `System.IO.Ports` ermöglicht den Zugriff auf die seriellen Anschlüsse des Computers. Im Zentrum des Namensraums steht die Klasse `SerialPort`. Ein Einsatzbeispiel dafür finden Sie in Form des Spiels *Space Invaders* in den Downloads zu diesem Buch (siehe auch Kapitel 7 »Konsolenanwendungen«).

Neuheiten seit .NET 3.0

Der Namensraum `System.IO` wurde in .NET 3.0 um folgende Funktionen erweitert:

- Der Unternamensraum `System.IO.Packaging` bietet Möglichkeiten zum Zusammenfassen von eigenständigen Dateien und Ressourcen zu einem Paket gemäß der Open Packaging Conventions (OPC). Die verschiedenen Teile eines Pakets können über so genannte *Package Relationships* miteinander verbunden sein. .NET bietet bisher als einziges physikalisches Format das ZIP-Format (Klasse `ZipPackage`). Grundsätzlich kann man aber eigene Formate (z. B. Datenbankspeicherung) auf Basis der abstrakten Basisklasse `Package` realisieren. Microsoft selbst bietet nun diese Paketierung in der XML Paper Specification (XPS), dem eigenen Konkurrenzformat zu PDF.

Neuheiten seit .NET 3.5

Der Namensraum `System.IO` wurde in .NET 3.5 um folgende Funktionen erweitert:

- Dreizehn Klassen im Unternamensraum `System.IO.Pipes` bieten die Möglichkeit zur Kommunikation über benannte und anonyme Pipes auch ohne den Einsatz von WCF

Neuheiten seit .NET 4.0

Der Namensraum `System.IO` wurde in .NET 4.0 um folgende Funktionen erweitert:

- `File.ReadLines()` liefert ein `IEnumerable<string>` zurück. Bisher gab es zum Lesen sämtlicher Zeilen nur `File.ReadAllLines()`, das ein `String-Array` lieferte.
- Ähnliches erlauben drei neue Methoden der Klasse `DirectoryInfo`: `EnumerateFiles()`, `EnumerateDirectories()` und `EnumerateFileSystemInfos()`. Diese Methoden liefern ein `IEnumerable<FileSystemInfo>` mit den Dateien und/oder Verzeichnissen, die sich innerhalb des jeweiligen Ordners befinden. `FileSystemInfo` ist die Basisklasse für `FileInfo` und `DirectoryInfo`.
- *Memory-Mapped Files*: Memory-Mapped Files bilden den Inhalt einer Datei auf den logischen Adressraum einer Applikation ab. Ein häufiges Anwendungsgebiet dafür stellt die Kommunikation zwischen Prozessen dar. Dies ist eine weitere Möglichkeit zur Interprozess-Kommunikation in .NET neben TCP/HTTP/FTP, Named Pipes, ASP.NET Webservices, .NET Remoting und WCF. Dieses Thema wird hier im Buch nicht behandelt. Dieses Buch behandelt nur Named Pipes, ASP.NET Webservices, .NET Remoting und WCF.

- `Path.Combine()` unterstützt nun auch *param-Arrays*, sodass beliebig viele Pfade zum Kombinieren als Parameter übergeben werden können
- Das Kopieren zwischen Streams ist jetzt einfacher mit der `Copy()`-Methode in allen Stream-Klassen.
- In der Vergangenheit konnte sich die Dateigröße nach einer Komprimierung mittels `DeflateStream` oder `GZipStream` vergrößern, wenn die zu komprimierenden Daten zuvor bereits komprimiert vorlagen. Darüber hinaus war es nicht möglich, Streams, welche größer als 4 GB waren, zu komprimieren. Diese beiden Eigenschaften treffen in .NET 4.0 nicht mehr zu.

Dateisystem

Im Mittelpunkt der Programmierung des Dateisystems stehen die Klassen `DriveInfo` (Laufwerk), `DirectoryInfo` (Verzeichnis) und `FileInfo` (Datei). Das Objektmodell dieser Klasse zeigt die nachstehende Abbildung. Die entsprechenden Objektmengen werden in Arrays des jeweiligen Objekttyps verwaltet; eigene Mengenklassen existieren im Gegensatz zu vielen anderen Anwendungsbereichen in der FCL nicht. Die Klasse `DriveInfo` ist neu seit .NET 2.0.

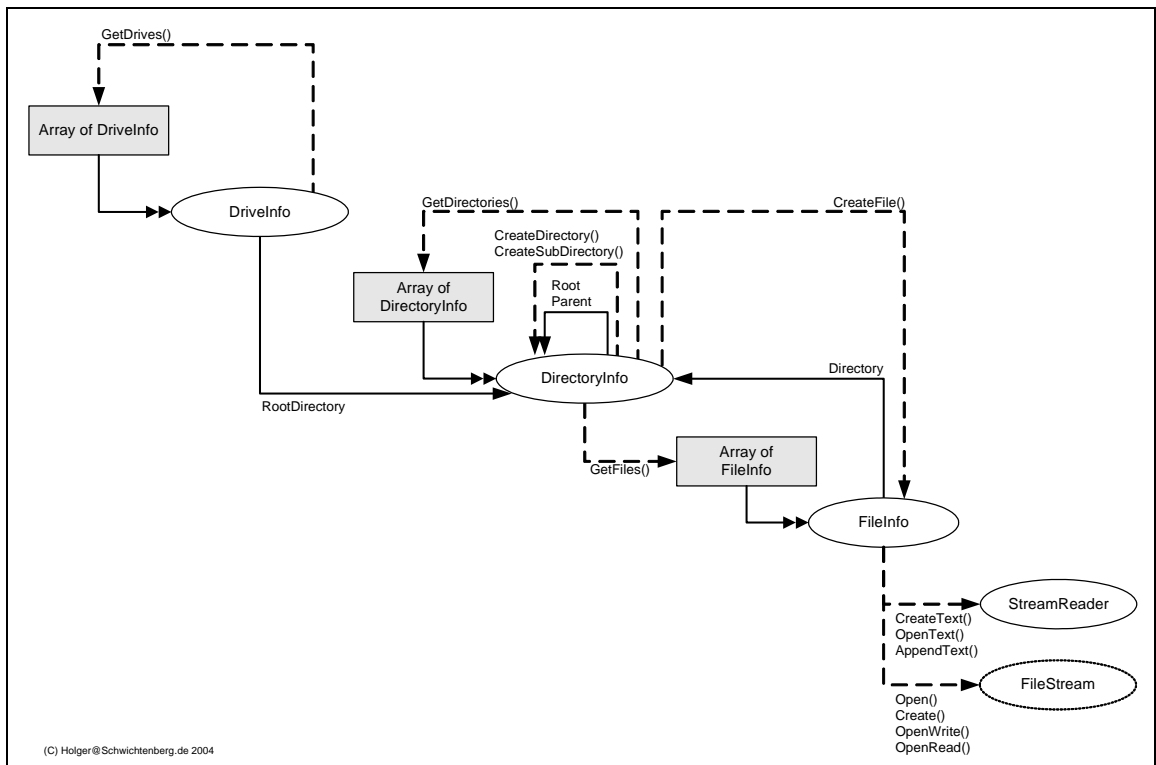


Abbildung 9.2 Objektmodell der Klassen `DriveInfo`, `DirectoryInfo` und `FileInfo`

Neben `DirectoryInfo` und `FileInfo` bietet die FCL alternativ für den Zugriff auf Verzeichnisse und Dateien auch die Klassen `File` und `Directory` an. Letztgenannte besitzen nur statische Mitglieder mit Dateisystemoperationen wie `Copy()`, `Delete()`, `Move()`, `Open()` und `GetCreationTime()`. Grundsätzlich bieten die beiden Klassenpaare äquivalente Funktionen in unterschiedlichen Darreichungsformen an. Für viele Anwendungen interessant ist `Directory.GetCurrentDirectory()`: Diese Methode liefert das aktuelle Arbeitsverzeichnis einer Anwendung.

Beispiel 1: Liste der Laufwerke

Das erste Beispiel zeigt, wie Sie alle Laufwerke in Ihrem Computersystem mit den zugehörigen Wurzelverzeichnissen auflisten können.

```
public void Laufwerke_Auflisten()
{
    foreach (DriveInfo di in DriveInfo.GetDrives())
    {
        Demo.Print("Laufwerk: " + di.Name);
        if (di.IsReady)
        {
            Demo.Print("  Bezeichnung: " + di.VolumeLabel);
            Demo.Print("  Typ: " + di.DriveType);
            Demo.Print("  Format: " + di.DriveFormat);
            Demo.Print("  Größe: " + di.TotalSize);
            Demo.Print("  Freier Platz: " + di.TotalFreeSpace);
            Demo.Print("  Wurzelordner: " + di.RootDirectory.FullName);
        }
        else
        {
            Demo.Print("  ist nicht bereit!");
        }
    }
}
```

Listing 9.12 Liste der Laufwerke ausgeben [System.IO.Demo.cs]

Beispiel 2: Liste der Dateien in einem Verzeichnis

Das zweite Listing nimmt Zugriff auf das Verzeichnis */_Daten/Dateisystem* unterhalb des aktuellen Arbeitsverzeichnisses. Zunächst gibt die Routine Informationen über das Verzeichnis aus, dann listet sie alle in dem Verzeichnis enthaltenen Textdateien auf, indem sie `GetFiles()` mit dem Muster **.txt* aufruft.

```
public void Datei_Liste() {
    // Liste der Dateien in einem bestimmten Ordner
    string verzeichnis = Directory.GetCurrentDirectory() + @"\_daten\dateisystem\";

    // Zugriff auf ein Dateiverzeichnis
    DirectoryInfo d = new DirectoryInfo(verzeichnis);

    // Prüfung auf Existenz des Verzeichnisses
    if ( !d.Exists ) {
        Demo.Print("Verzeichnis nicht vorhanden!");
        return;
    }
}
```

```
// Ausgabe von Informationen über den Ordner
Demo.Print("Erzeugt am: " + d.CreationTime);
Demo.Print("Zuletzt gelesen am : " + d.LastAccessTime);
Demo.Print("Zuletzt geändert am : " + d.LastWriteTime);
Demo.Print("Wurzelordner: " + d.Root.Name);
Demo.Print("Name des übergeordneten Ordners: " + d.Parent.Name);
Demo.Print("Pfad des übergeordneten Ordners: " + d.Parent.FullName);

// Liste aller Textdateien in diesem Verzeichnis
Demo.Print("Alle Textdateien in Ordner: " + d.FullName);
foreach (FileInfo f in d.GetFiles("*.txt"))
    Demo.Print(f.Name + ";" + f.Length + ";" + f.CreationTime);
}
```

Listing 9.13 Liste der Dateien in einem Verzeichnis ausgeben [System.IO.Demo.cs]

HINWEIS Leider fehlt in .NET 4.0 nach wie vor sowohl in der `Directory-` als auch in der `DirectoryInfo`-Klasse eine Methode zum Kopieren eines ganzen Verzeichnisses.

TIPP Die neue Möglichkeit `EnumerateFiles()` erlaubt mit LINQ (ein kleiner Vorgriff auf Kapitel 10!) einfache Abfragen, welche sich auf Dateien sowie Dateiinhalte beziehen. Die LINQ-Abfrage in Listing 9.14 ermittelt mit diesen Methoden beispielsweise jene Zeilen aus sämtlichen CSV-Dateien des aktuellen Ordners, bei welchen sich in der ersten Spalte der Wert *1* wiederfindet. Das Ergebnis dieser Abfrage – ein `IEnumerable<string>` – wird anschließend an die zuvor erwähnte Methode `WriteAllLines()` übergeben, um die gefundenen Zeilen in der Datei *result.txt* zu hinterlegen.

```
DirectoryInfo dir = new DirectoryInfo("d:\datenordner");

var info =
    from file in Directory.EnumerateFiles(".", "*.csv")
    from line in File.ReadLines(file)
    where line.Split(';')[0] == "1"
    select file + ": " + line;

File.WriteAllLines("result.txt", info);
```

Listing 9.14 LINQ-Abfrage über sämtliche CSV-Dateien eines Ordners

Dateiinhalte

Ähnlich wie Java arbeitet .NET mit Streams, um Inhalte von Dateien zu lesen und zu ändern. Ein Stream ist ganz allgemein eine Abfolge von Bytes. Eine Instanz der Klasse `FileStream` repräsentiert den Inhalt einer Text- oder Binärdatei. Andere Stream-Typen existieren z.B. für den Datenversand über ein Netzwerk (`System.Net.Sockets.NetworkStream`), für eine Byte-Folge im Hauptspeicher (`System.IO.MemoryStream`), eine verschlüsselte Byte-Folge (`System.Security.Cryptography.CryptoStream`) oder eine komprimierte Byte-Folge (`System.IO.Compression.GZipStream`). Basisklasse für alle Stream-Klassen ist `System.IO.Stream`.

System.IO.Stream definiert für jeden Stream einfache Operationen wie CanRead, CanWrite, Read(), Write() und Close(). Komfortablere Zugriffsmöglichkeiten (beispielsweise Peek(), ReadLine(), ReadToEnd(), WriteLine()) existieren in Form so genannter Reader- und Writer-Klassen.

- StreamReader und StreamWriter für Streams mit ASCII-Zeichen
- BinaryReader und BinaryWriter für Streams mit beliebigen Byte-Folgen

Beispiel 1: Textdatei schreiben

Das erste Beispiel zeigt das Schreiben einer Protokolldatei mithilfe der Klassen FileStream und StreamWriter. Bei einer Binärdatei würden Sie analog die Klasse BinaryWriter verwenden.

```
// Schreiben einer Protokolldatei
public void Textdatei_Schreiben()
{
    // Festlegung der Datei
    string dateiName = @"..\_daten\dateisystem\protokolldatei.txt";
    // Datei öffnen
    FileStream fs = new FileStream(dateiName, FileMode.OpenOrCreate, FileAccess.Write);
    // Stream öffnen
    StreamWriter w = new StreamWriter(fs);
    // Anfügen am Ende
    w.BaseStream.Seek(0, SeekOrigin.End);
    // Zeilen schreiben
    w.WriteLine("Start des Programms: " + DateTime.Now.ToString());
    // Zeichen schreiben ohne Umbruch
    w.Write("Datenblock: ");
    for (int i = 0; i < 26; i++)
        w.Write((char)(97 + i));
    // Zeilen schreiben
    w.WriteLine();
    w.WriteLine("Ende des Programms: " + DateTime.Now.ToString());
    // Writer und Stream schließen
    w.Close();

    fs.Close();
}
```

Listing 9.15 Schreiben einer Textdatei [System.IO.Demo.cs]

Beispiel 2: Textdatei lesen

Das zweite Beispiel zeigt das zeilenweise Lesen einer Protokolldatei mithilfe der Klassen FileStream und StreamReader. Bei einer Binärdatei würden Sie analog die Klasse BinaryReader verwenden.

```
// Lesen aus einer Protokolldatei
public void Textdatei_Lesen()
{
    // Festlegung der Datei
    string dateiName = @"..\_daten\dateisystem\protokolldatei.txt";
    // Datei öffnen
    FileStream fs = new FileStream(dateiName, FileMode.OpenOrCreate, FileAccess.ReadWrite);
```

```
// Stream öffnen
StreamReader r = new StreamReader(fs);
// Zeiger auf den Anfang
r.BaseStream.Seek(0, SeekOrigin.Begin);
// Alle Zeilen lesen und zeilenweise ausgeben
while (r.Peek() > -1)
    Demo.Print(r.ReadLine());
// Reader und Stream schließen
r.Close();

fs.Close();
}
```

Listing 9.16 Auslesen einer Textdatei [System.IO.Demo.cs]

Übertragen von Daten zwischen Streams

Jeder, der schon einmal mit Streams gearbeitet hat, kennt Routinen, welche blockweise aus einem Stream lesen und diese Blöcke in einen anderen Stream schreiben. Für solche Fälle besitzt die Klasse `Stream` ab .NET 4.0 eine Methode `CopyTo()`, welche den Stream, an welchen die Daten gesendet werden sollen, entgegennimmt. Eine Überladung dieser Methode nimmt zusätzlich eine Blockgröße in `Byte` entgegen. Diese gibt an, wie groß die zu übertragenden Blöcke sein sollen. Der Standardwert wurde hierfür auf 4096 festgelegt. Listing 9.17 zeigt ein Beispiel für den Einsatz dieser Methode. Dieses kopiert die aus dem Stream `input` gelesenen Daten in den Stream `output`.

```
public static void CopyStreamSample()
{
    FileStream input;
    FileStream output;

    using (input = new FileStream(@"c:\temp\a.txt", FileMode.Open))
    {
        using (output = new FileStream(@"c:\temp\b.txt", FileMode.CreateNew))
        {
            input.CopyTo(output);
        }
    }
}
```

Listing 9.17 Daten zwischen Streams übertragen

Kommunikation über Pipes (System.IO.Pipes)

Neu seit .NET 3.5 ist die Unterstützung für benannte und unbenannte Pipes. Pipes sind schon eine recht alte Kommunikationstechnik, die es in Windows schon seit der Einführung von Windows NT gibt. Pipes werden seit .NET 3.0 im Rahmen der Windows Communication Foundation (WCF) unterstützt; dort werden Methodenaufrufe mit Objekten als Parametern über Pipes verschickt zwischen zwei Prozessen in einem System. Ab .NET 3.5 kann man Pipes auch direkt ohne WCF nutzen. Dann versendet man aber auf einer niedrigeren Ebene Bytefolgen statt Objekte. Der Namensraum `System.IO.Pipes` unterstützt rechnerübergreifende benannte Pipes mit Vollduplexkommunikation sowie lokale unbenannte Pipes mit Einwegkommunikation.

Anonyme Pipes (Klassen `AnonymousPipeServerStream` und `AnonymousPipeClientStream`) sind nur zur Kommunikation zwischen Prozessen auf einem Computer möglich; die Nutzung über ein Netzwerk wird nicht unter-

stützt. Benannte Pipes (Klassen `NamedPipeServerStream` and `NamedPipeClientStream`) ermöglichen auch die rechnerübergreifende Kommunikation. Außerdem unterscheiden sich die beiden Pipe-Arten dadurch, dass anonyme Pipes nur eine 1:1-Zuordnung zwischen Client und Server sowie nur Einweg-Kommunikation unterstützen. Für 1:N-Kommunikation und Zweiwegverbindungen braucht man benannte Pipes.

Die Pipe-Unterstützung findet man im Namensraum `System.IO.Pipes` in der `System.Core.dll`.

Beispiel

Das folgende Beispiel zeigt einen einfachen Server und einen einfachen Client für eine benannte Pipe.

```
/// <summary>
/// Named Pipe Server
/// </summary>
public static void RunServer()
{
    Console.WriteLine("Starte Server...");
    using (var p = new NamedPipeServerStream("WWWingsPipe3"))
    {
        p.WaitForConnection();

        using (var r = new StreamReader(p))
        {
            while (true)
            {
                string Daten = r.ReadLine();
                if (Daten == null) break;
                Console.WriteLine("Daten empfangen: " + Daten);
            }
        }
        Console.WriteLine("Beende Server...");
    }
}

/// <summary>
/// Named Pipe Client
/// </summary>
public static void RunClient()
{
    Console.WriteLine("Starte Client...");
    NamedPipeClientStream p = new NamedPipeClientStream("WWWingsPipe3");
    p.Connect();
    StreamWriter w = new StreamWriter(p);
    //w.AutoFlush = true;
    Console.WriteLine("Sende Daten...");
    w.WriteLine("Daten...Daten...Daten...");
    w.WriteLine("Daten...Daten...Daten...");
    w.Close();
    p.Close();
    Console.WriteLine("Client beendet!");
}
```



Abbildung 9.3 Ausgabe des Servers aus dem obigen Beispiel

System.Configuration

Der Namensraum `System.Configuration` enthält Klassen für den Zugriff auf die XML-basierten .NET-Konfigurationsdateien (mit der Dateierdung `.config`). Eine allgemeine Einführung zu Konfigurationsdateien haben Sie bereits im Kapitel 4 »Grundkonzepte des .NET Framework 4.0« erhalten. Zur Straffung des Buchs wird an dieser Stelle daher auf eine Wiederholung verzichtet.

Neuheiten seit .NET 2.0

Für die Anwendungskonfigurationsdateien (`.config`) gibt es ab .NET 2.0 folgende Neuerungen:

- Speicherung benutzerspezifischer Daten in Konfigurationsdateien der Form `Benutzername.config`
- Ein neues typisiertes Modell für die Speicherung von Anwendungseinstellungen
- Seit .NET 2.0 können viele Elemente in einer Konfigurationsdatei durch XML Encryption verschlüsselt werden (Protected Configuration)
- Es existiert ein spezielles Konfigurationselement zur Ablage von Verbindungszeichenfolgen für ADO.NET
- Über die neue Klasse `ConfigurationManager` können die XML-Konfigurationsdateien nicht nur gelesen, sondern auch verändert werden

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <appSettings>
    <add key="Mailserver" value="Essen"/>
    <add key="SAPServer" value="Bochum"/>
  </appSettings>
  <connectionStrings>
    <add name="WWWDatenbank_MSSQL" connectionString="Server=marl; IntegratedSecurity=True; Database=WorldWideWings"
      providerName="System.Data.SqlClient" />
    <add name="WWWDatenbank_Access" connectionString="..."
      providerName="System.Data.OleDb" />
  </connectionStrings>
</configuration>
```

Abbildung 9.4 Beispiel für eine Anwendungskonfigurationsdatei

HINWEIS In .NET 3.0/3.5/4.0 gab es hier keine Neuerungen.

ConfigurationManager

Mit .NET 2.0 eingeführt wurde die Klasse `ConfigurationManager`, die alle Möglichkeiten der `.config`-Dateien bündelt. Die alte Klasse `System.Configuration.ConfigurationSettings` für den Zugang zu den Nutzerdaten in der `.config`-Datei (<appSettings>-Sektion) ist weiterhin vorhanden, sollte aber nicht mehr genutzt werden.

`ConfigurationManager` erlaubt den Zugriff auf die Konfigurationsdatei der laufenden Anwendung, die Konfigurationsdatei einer anderen Anwendung sowie auf die zentrale Konfigurationsdatei `machine.config`. Zur Nutzung der Klasse `ConfigurationManager` müssen Sie die `System.Configuration.dll` referenzieren.

Das folgende Listing zeigt verschiedene Anwendungsbeispiele.

```
public void Config_Demo()
{
    string MailServer = ConfigurationManager.AppSettings["Mailserver"];
    string WWWSQLDatenbank =
    ConfigurationManager.ConnectionStrings["WWWDatenbank_MSSQL"].ConnectionString;
    string WWWAccessDatenbank =
    ConfigurationManager.ConnectionStrings["WWWDatenbank_Access"].ConnectionString;
    Console.WriteLine("Mailserver: " + MailServer);
    Console.WriteLine("Verbindungszeichenfolge 1: " + WWWSQLDatenbank);
    Console.WriteLine("Verbindungszeichenfolge 2: " + WWWAccessDatenbank);
    // --- Liste aller Verbindungszeichenfolgen
    foreach (ConnectionStringSettings cset in ConfigurationManager.ConnectionStrings)
    {
        Console.WriteLine(cset.Name + " = " + cset.ConnectionString);
    }
    // --- Ändern einer Einstellung
    Configuration c = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
    ConfigurationSection aset = c.GetSection("appSettings");
    ConfigurationManager.AppSettings["Mailserver"] = "Bochum";
    c.Save(ConfigurationSaveMode.Full);
    Console.WriteLine("Mailserver nun: " + ConfigurationManager.AppSettings["Mailserver"]);

    // --- Zugriff auf machine.config
    Configuration mc = ConfigurationManager.OpenMachineConfiguration();
    Console.WriteLine("machine.config liegt hier: " + mc.FilePath);
    // --- Liste der Sektionen in der machine.config
    foreach (ConfigurationSection s in mc.Sections)
    {
        Console.WriteLine("Sektion: " + s.SectionInformation.Name);
        Console.WriteLine("  Schreibschutz?: " + s.SectionInformation.IsLocked);
        Console.WriteLine("  Verschlüsselt?: " + s.SectionInformation.IsProtected);
    }
}
```

Listing 9.18 Anwendungsbeispiele der Klasse ConfigurationManager [Configuration.cs]

Verschlüsselte Sektionen

.NET erlaubt seit Version 2.0, einzelne Sektionen der *.config*-Datei zu verschlüsseln (*Protected Configuration*). Für die Verschlüsselung sind zwei verschiedene Provider verfügbar: RSA und das Windows Data Protection API (DAPI). Da DAPI-Schlüssel rechner-spezifisch sind, ist die Verwendung von RSA vorteilhaft, wenn die Anwendung auf einen anderen Rechner migriert werden soll. Ein Werkzeug liefert Microsoft derzeit nur für *web.config*-Dateien. Die Verschlüsselung für die übrigen Anwendungskonfigurationsdateien muss in eigenem Programmcode erfolgen.

ACHTUNG Nicht alle Sektionen können verschlüsselt werden. Ausnahmen sind beispielsweise `<mssql>`, `<system.runtime.remoting>` und `<protectedData>`.

Verschlüsselung von web.config-Dateien mit aspnet_regiis.exe

Das im .NET Framework Redistributable enthaltene Werkzeug *aspnet_regiis.exe* erlaubt es, die notwendigen Schritte auszuführen.

Erforderlich ist zuerst, dass das Benutzerkonto, unter dem die Webanwendung läuft, Zugriff auf den zu verwendenden Schlüssel hat. Wenn kein expliziter Schlüsselcontainer angegeben wird, wird der eingebaute `NetFrameworkConfigurationKey` verwendet. Das Standardbenutzerkonto für Webanwendungen erhält mit folgendem Befehl Zugriff auf den Schlüssel:

```
aspnet_regiis -pa "NetFrameworkConfigurationKey" "Server04\ASPNET"
```

TIPP Falls Sie nicht wissen, unter welcher Identität Ihre Webanwendung läuft, können Sie diese mit `Response.Write(System.Security.Principal.WindowsIdentity.GetCurrent().Name);` ermitteln. Weitere Informationen über die Anwendungsidentität in Webanwendungen erhalten Sie im Zusatzkapitel zu ASP.NET, welches Sie als PDF auf dem Leser-Portal herunterladen können.

Mit dem folgenden Befehl wird dann die Sektion `<connectionStrings>` in der *web.config*-Datei in der Wurzel der Webanwendung *WWWings_Web*, die als erste Website im IIS eingetragen ist, mit dem RSA-Provider asymmetrisch verschlüsselt:

```
aspnet_regiis -pe "connectionStrings" -prov "RSAProtectedConfigurationProvider" -app "W3SVC/1/WWWings_Web"
```

Das Ergebnis ist in den nachstehenden Listings dokumentiert.

```
<protectedData>
  <protectedDataSections>
    <add name="connectionStrings" provider="RSAProtectedConfigurationProvider"
      inheritedByChildren="false" />
  </protectedDataSections>
</protectedData>
```

Listing 9.19 Festlegung, welche Elemente verschlüsselt werden sollen

```
<EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
  xmlns="http://www.w3.org/2001/04/xmlenc#">
  <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
  <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
    <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
      <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
      <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <KeyName>Rsa Key</KeyName>
      </KeyInfo>
    <CipherData>
      <CipherValue>UVwRwnqcICxUQRJ0cxFU0dc4+mf/v/yOLaXlQhEa/ecpvEemloBa4J6o04Cg0BKC5J0erxmDqX55B08aVRLvb6iudH
yt/hMLXSDwx7Q1SisttS9AD/L55jfbY6cUo83VL9yG0ivJgv9MJocANI/hqIMd9XUe16NjRxnXOVbiJ8=</CipherValue>
    </CipherData>
  </EncryptedKey>
```

```

    </KeyInfo>
    <CipherData>
    <CipherValue>YEkARSQzYdCOU+JrVuIB0RczV1z9nt+JcIA9/pJh4n9s8XCuGFBQC+AqA+T6T/hc75dNEWuHPbfZ99S1guXXoZhrzn
smRfKMUR10tWxdfHfuiE9dcESuVriggovtspstUhnQvoB1JSJu174zmbDzh11mkIM7EBI26W0w16fqXPH/...
/J0JaUM7hLUc65lHGJtMU2vfo15muzpaP+wCbGYeMr/mK52wZT2An3NvojXGadqmahm6vfIsVWTBaxhMC2uMVZ04Sjbafa4QqItVHPW
rSQZ9Yt0cDGXMPJPd77cBUS/vwZickGwL9bQ6pHqHYaqLs8eD19bN</CipherValue>
    </CipherData>
  </EncryptedData>
</connectionStrings>

```

Listing 9.20 Verschlüsselte Verbindungszeichenfolgen

Sie können die Verschlüsselung wieder aufheben, indem Sie aufrufen:

```
aspnet_regiis -pd "connectionStrings" -app "/"
```

Verschlüsselung per Programmcode

Für Anwendungskonfigurationsdateien von Windows- und Konsolenanwendungen bleibt derzeit nur die Möglichkeit, die Verschlüsselung per Programmcode selbst zu schreiben. Dies ist zum Glück kein großer Aufwand: Die Klasse `ConfigurationSection` bietet dafür zwei einfache Methoden, `ProtectSection()` und `UnprotectSection()`. Kurioserweise werden diese Methoden in der MSDN-Dokumentation als »not intended to be used directly from your code« bezeichnet.

Das folgende Beispiel kehrt den Verschlüsselungsstatus der Sektion `<connectionStrings>` in der Anwendungs-konfigurationsdatei um. Verwendet wird hier der DAPI-Provider.

```

public void Config_Verschluesseln()
{
    Console.WriteLine("Zugriff auf Sektion...");
    Configuration c = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
    ConfigurationSection s = c.GetSection("connectionStrings");
    Console.WriteLine("  Name: " + s.SectionInformation.Name);
    Console.WriteLine("  Verschlüsselt?: " + s.SectionInformation.IsProtected);
    if (s.SectionInformation.IsProtected)
    {
        Console.WriteLine("Verschlüssele...");
        s.SectionInformation.ProtectSection(ProtectedConfiguration.DataProtectionProviderName);
    }
    else
    {
        Console.WriteLine("Entschlüssele...");
        s.SectionInformation.UnprotectSection();
    }
    c.Save();
    Console.WriteLine("Kontrolle...");
    Configuration c2 = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
    ConfigurationSection s2 = c2.GetSection("connectionStrings");
    Console.WriteLine("  Name: " + s2.SectionInformation.Name);
    Console.WriteLine("  Verschlüsselt?: " + s2.SectionInformation.IsProtected);
    Console.WriteLine("Nutzen...");
}

```

```
string WWWAccessDatenbank =  
    ConfigurationManager.ConnectionStrings["WWWDatenbank_MSSQL"].ConnectionString;  
Console.WriteLine("Verbindungszeichenfolge: " + WWWAccessDatenbank);  
Console.WriteLine("Fertig! :-)");  
}
```

Listing 9.21 Verschlüsseln und Entschlüsseln der Verbindungszeichenfolgen [Configuration.cs]

Anwendungseinstellungen (Application Settings)

Neben den Sektionen <appSettings> und <connectionStrings> bietet .NET seit Version 2.0 noch ein weiteres Verfahren zur Speicherung von Einstellungen an. Anwendungseinstellungen (engl.: *Application Settings*) können auf Anwendungs- oder Benutzerebene abgelegt werden.

Einstellungen auf Anwendungsebene werden in der Sektion <applicationSettings> in der Anwendungs-konfigurationsdatei (*name.exe.config*) gespeichert. Benutzereinstellungen liegen in XML-Dateien der Form *user.config* im Profil des jeweiligen Benutzers, wobei der Name des Herstellers, der Name der Anwendung und die Versionsnummer der Anwendung im Pfad erscheinen (z.B. *C:\Documents and Settings\HS\Local Settings\Application Data\www.IT-Visions.de\ConsoleUI_CS\1.0.0.0*). Wenn für den Benutzer wandernde Profile (Roaming Profiles) eingerichtet sind, wird die Datei im *Roaming Profile* abgelegt. Wenn Einstellungen im lokalen Profil und im *Roaming Profile* vorhanden sind, hat das *Roaming Profile* Vorrang.

Den Pfad, in dem die Benutzereinstellungsdatei gespeichert wird, ermitteln Sie mit `System.Windows.Forms.Application.LocalUserAppDataPath`. Die Versionsnummer der Anwendung ist im Pfad enthalten. Eine Anwendung kann mit der Methode `GetPreviousVersion()` in der Klasse `LocalFileSettingsProvider` Einstellungen aus der vorherigen Version ermitteln bzw. mit `Upgrade()` die Einstellungen auf die aktuelle Version migrieren.

Vorgabewerte für die benutzerspezifischen Einstellungen können in der Anwendungs-konfigurationsdatei in der <userSettings>-Sektion definiert werden. Auch Windows Forms-Steuerelemente können diese Verfahren nutzen, um benutzerspezifische Anpassungen abzulegen. Mit diesem Mechanismus wird die Windows-Registrierungsdatenbank ein wenig näher an ihr Grab getragen.

HINWEIS

Laut der Dokumentation sind die Einstellungen auf Anwendungsebene »aus Sicherheitsgründen für die meisten Anwendungen« schreibgeschützt; sie können also durch die Anwendung selbst nicht verändert werden, sondern nur durch einen manuellen Eingriff in die *.config*-Datei mit einem XML-Editor.

Providermodell

Grundsätzlich ist eine andere Form der Speicherung möglich, da die Speicherung auf einem Providermodell basiert. Die seit .NET 2.0 mitgelieferte Implementierung für Windows- und Konsolenanwendungen ist der `LocalFileSettingsProvider`. Die in ASP.NET (ab Version 2.0) verwendete Klasse `SqlProfileProvider` zur Speicherung von Profildaten im Microsoft SQL Server ist ebenso wie der `LocalFileSettingsProvider` von `System.Configuration.SettingsProvider` abgeleitet.

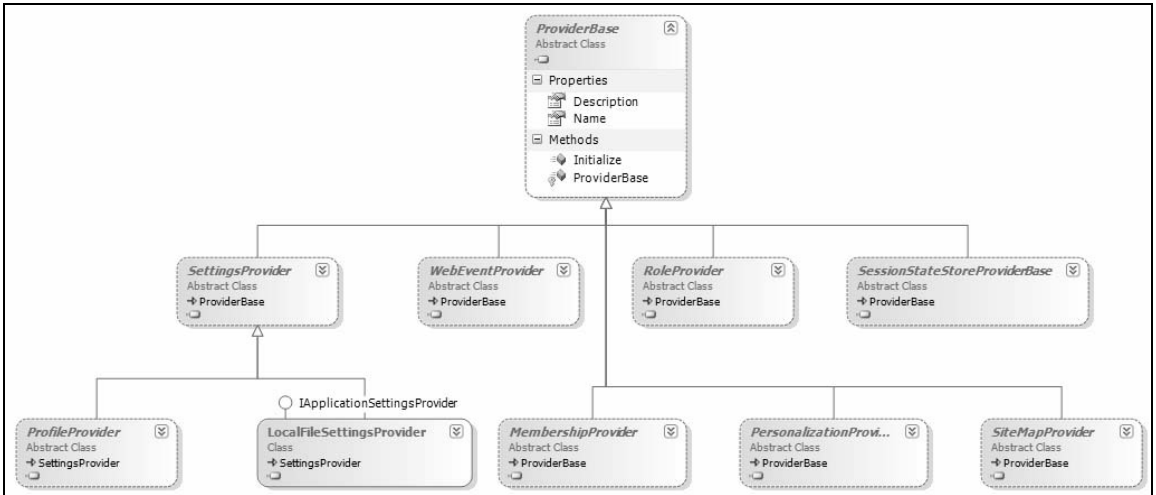


Abbildung 9.5 Provider im .NET-Providermodell

Nutzung der Anwendungseinstellung

Microsoft hat vorgesehen, dass die Anwendungskonfigurationseinstellungen komfortabel über frühes Binden genutzt werden können. Visual Studio bietet dafür folgende Unterstützung:

- Definieren von Anwendungseinstellungsvariablen über die Registerkarte *Einstellungen* (Settings) in den Projekteigenschaften (Visual Studio speichert diese Einstellungen in einer *.settings*-Datei im Projekt ab)
- Automatische Generierung einer von `System.Configuration.ApplicationSettingsBase` abgeleiteten Verpakungsklasse mit Namen `Settings`, die alle Einstellungen als typisierte Datenmitglieder anbietet

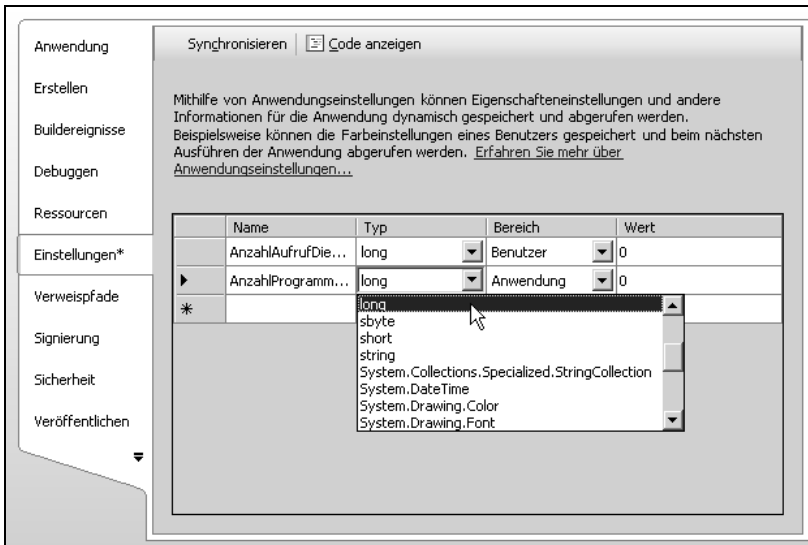


Abbildung 9.6 Definition von Anwendungseinstellungsvariablen in den Projekteigenschaften von Visual Studio (ab 2005)

Das folgende Beispiel zeigt die Nutzung der Settings-Klasse in C#. In Visual Basic werden die Einstellungen über `My.Settings` bereitgestellt.

```
public void SettingsDemos()
{
    ConsoleUI_CS.Properties.Settings s = new ConsoleUI_CS.Properties.Settings();
    s.AnzahlAufrufDiesesNutzers++;
    s.AnzahlProgrammAufrufe++;
    s.Save();
    Console.WriteLine("Anzahl Programmaufrufe aller Nutzer: " + s.AnzahlProgrammAufrufe.ToString());
    Console.WriteLine("Anzahl Programmaufrufe dieses Nutzer: " + s.AnzahlAufrufDiesesNutzers.ToString());
}
```

Listing 9.22 Nutzung der Einstellungen über die generierte Verpackungsklasse [Configuration.cs]

HINWEIS Da die Klasse `ApplicationSettingsBase` ein Providermodell nutzt, um die Einstellungen zu laden und zu speichern, besteht grundsätzlich die Möglichkeit, einen anderen Datenspeicher zu verwenden, indem Sie eine von `SettingsProvider` abgeleitete Klasse erstellen und der Settings-Klasse als Provider (Attribut `Providers`) anbieten. Der Standardprovider ist der `LocalFileSettingsProvider`.

System.Diagnostics

Der Namensraum `System.Diagnostics` enthält Klassen für die Verwaltung von Prozessen, das Tracing von Anwendungen und die Nutzung der Windows-Ereignisprotokolle (inkl. *Event Tracing for Windows* (ETW) in `System.Diagnostics.Eventing`) sowie der Windows-Leistungsindikatoren (`System.Diagnostics.PerformanceData`) und die Steuerung der statischen Codeanalyse (`System.Diagnostics.CodeAnalysis`).

HINWEIS In .NET 4.0 sind neu hingekommen Klassen für Prä- und Postbedingungen und Invarianten in Methoden (*Code Contracts*) im Sub-Namensraum `System.Diagnostics.Contracts`. Code Contracts können hier aus Platzgründen nicht besprochen werden. Hier sei verwiesen auf das Geschwisterbuch »*.NET 4.0 Update*« [HS07].

Im Folgenden behandelt werden können nur Prozess- und Ereignisprotokollverwaltung.

Prozesse

Das .NET Framework verwaltet die Windows-Prozesse durch Instanzen der Klasse `System.Diagnostics.Process`. Eine Liste aller Prozesse erhalten Sie über `GetProcess()` oder `GetProcessByName()`. Ein `Process`-Objekt enthält Informationen über seinen Zustand, die den Prozess realisierenden Module und die von dem Prozess gestarteten Threads (siehe Abbildung des Objektmodells).

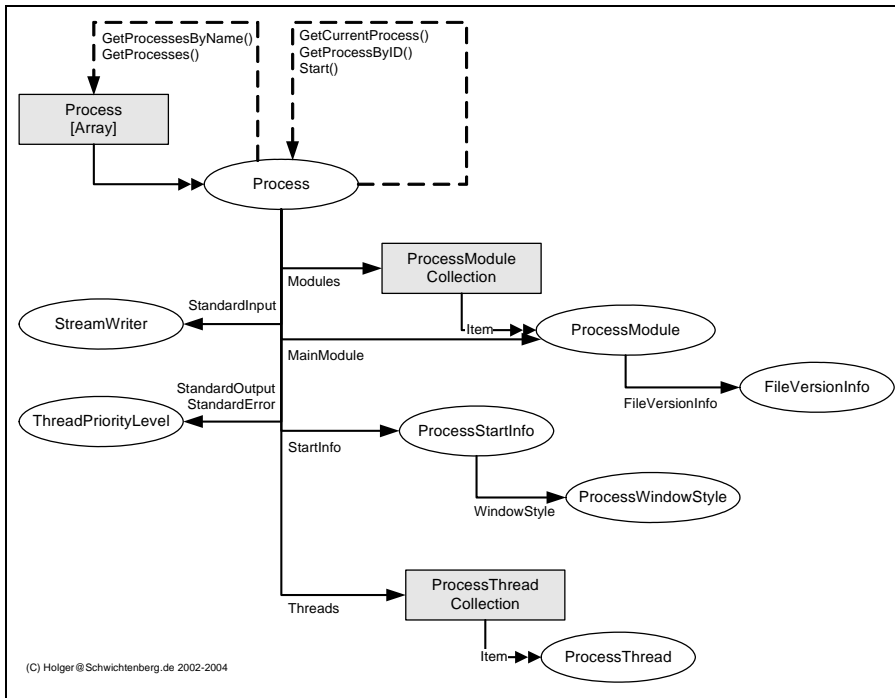


Abbildung 9.7 Objektmodell für die Prozessverwaltung

Beispiel 1: Liste der Prozesse

Das erste Beispiel listet alle laufenden Prozesse auf.

```
// Liste aller Instanzen des Internet Explorer
public void Prozesse_AuflistenSelektiv()
{
    // Liste ausgewählter Prozesse
    string processName = "iexplore";
    // Liste der Prozesse holen
    Process[] pp = Process.GetProcessesByName(processName);
    // Ausgaben
    Demo.Print("Prozesse mit Namen: " + processName);
    Demo.Print("Anzahl von Prozessen: " + pp.Length);
    // Schleife über alle Instanzen
    foreach (Process p in pp)
        Demo.Print(p.Id + ":" + p.StartTime);
}
```

Listing 9.23 Liste aller laufenden Prozesse [Prozesse.cs]

Beispiel 2: Prozess starten

Das zweite Beispiel startet den Internet Explorer. In weiteren Parametern könnten Sie optional eine andere Benutzeridentität für den neuen Prozess angeben.

```
// Start eines Prozesses
public void Prozess_Starten()
{
    Process.Start("IExplore.exe", "http://www.dotnetframework.de");
}
```

Listing 9.24 Den Internet Explorer als Prozess starten [Prozesse.cs]

HINWEIS Ein Beispiel für den Start eines Prozesses unter einem anderen Benutzerkonto finden Sie im Kapitel 7 »Konsolenanwendungen«.

Beispiel 3: Prozess beenden

Im dritten Beispiel werden alle Instanzen von Microsoft Word beendet. Die Methode `CloseMainWindow()` entspricht dem Aufruf von *Datei/Beenden* im Hauptfenster einer Windows-Anwendung durch den Benutzer. Wenn Änderungen an einem Dokument vorgenommen wurden, fragt die Anwendung den Benutzer zunächst, ob diese gespeichert werden sollen. Der Benutzer kann speichern oder den Vorgang abbrechen. Die Methode liefert `True` zurück, wenn der Beendigungsvorgang angestoßen werden konnte. `True` als Rückgabewert sagt aber nichts darüber aus, ob die Anwendung auch tatsächlich geschlossen wurde. `False` bedeutet, die Anwendung kann derzeit nicht normal beendet werden, z.B. weil ein Dialogfeld geöffnet ist. Härter ist die Methode `Kill()`: Hier hat der Benutzer keinen Einfluss auf das weitere Geschehen, denn der Prozess wird sofort ohne Nachfrage beendet. Änderungen an offenen Dokumenten werden nicht gespeichert.

```
// Alle Word-Prozesse beenden
public void Prozess_Stoppen()
{
    string processName = "winword";
    // Prozesse dieses Namens ermitteln
    Process[] pp = Process.GetProcessesByName(processName);
    // Schleife über diese Prozesse
    Demo.Print("Beenden aller Prozesse mit Namen: " + processName);
    Demo.Print("Anzahl Prozesse: " + pp.Length);
    foreach (Process p in pp)
    {
        Demo.Print("Prozess: " + p.Id + " wird beendet...");
        // Beenden-Anfrage stellen...
        if (p.CloseMainWindow())
        {
            // Normales Ende
            Demo.Print(" Prozess wurde normal beendet!");
        }
        else
        {
            // Keine Reaktion -> gewaltsames Ende
            p.Kill();
            Demo.Print(" Prozess wurde gewaltsam beendet!");
        }
    }
}
```

Listing 9.25 Prozess beenden [Prozesse.cs]

Ereignisprotokolle

Der Zugriff mit dem .NET Framework auf die Windows-Ereignisprotokolle, die Sie mit der Windows-Ereignisanzeige einsehen können, ist auf den ersten Blick fast selbsterklärend, die Tücken stecken aber im Detail. Sie benötigen drei Klassen:

- `EventLog` repräsentiert ein Ereignisprotokoll (z. B. *Anwendung*, *System*). Das Anlegen eigener Protokolle ist über diese Klasse möglich.
- `EventLogEntryCollection` ist die Auflistung der Einträge in einem Ereignisprotokoll.
- `EventLogEntry` ist eine Klasse für einzelne Einträge.

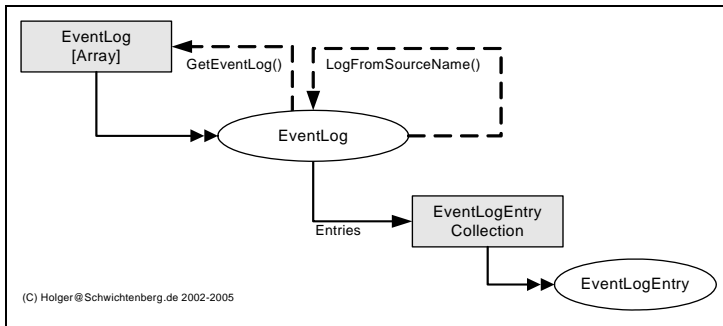


Abbildung 9.8 Objektmodell

Beispiel 1: Lesen vorhandener Einträge

Die folgende Unteroutine listet die letzten zwanzig Einträge aus dem Windows-Anwendungsprotokoll auf.

```
// Auslesen von Einträgen aus einem Ereignisprotokoll
public void EreignisprotokollEintraege_Lesen()
{
    // Name des Ereignisprotokolls
    string logname = "Application";
    // Anzahl der auszugebenden Einträge
    long anzahl = 20;
    // Zähler
    long count = 0;
    // -- Zugriff auf das Ereignisprotokoll
    EventLog log = new EventLog(logname);
    Demo.Print("Letzte " + anzahl.ToString() + " Einträge von " + log.Entries.Count +
        " Einträgen aus dem Protokoll " + log.Log + " auf dem Computer " + log.MachineName);
    // Schleife über alle Einträge
    foreach (EventLogEntry eintrag in log.Entries)
    {
        count += 1;
        if (count > log.Entries.Count - anzahl)
    
```



```
{
    Demo.Print(eintrag.EntryType + ":" +
        eintrag.InstanceId + ":" +
        eintrag.Category + ":" +
        eintrag.Message + ":" +
        eintrag.Source + ":" +
        eintrag.TimeGenerated + ":" +
        eintrag.TimeWritten + ":" +
        eintrag.UserName + ":");
}
}
```

Listing 9.26 Auslesen des Windows-Anwendungsprotokolls [Eventlog.cs]

Sie können mit den .NET-Klassen nicht in den Ereignisprotokollen suchen. Wenn Sie diese Funktionalität benötigen, müssen Sie auf die WMI-Klasse Win32_EventLogEntry ausweichen.

Beispiel 2: Anlegen eines Ereignisprotokolleintrags

Nach dem vorherigen Beispiel könnte man vermuten, dass die Methode WriteEntry() in der Klasse EventLog dazu dient, einen neuen Eintrag in ein Ereignisprotokoll zu schreiben, das zuvor instanziiert wurde. Allerdings ist WriteEntry() eine statische Methode, die zudem keinen Parameter für den Ereignisprotokollnamen bietet, sondern nur die Angabe einer Ereignisquelle erlaubt. Daher muss man zunächst eine Ereignisquelle mit CreateEventSource() erzeugen. Hier allerdings ist das .NET Framework sehr komfortabel: Man kann einen Ereignisquellennamen mit einem Ereignisprotokoll auf einem bestimmten Computer assoziieren, und das .NET Framework legt das Protokoll neu an, falls es nicht existiert. Beachten muss man nur, dass man für jedes Protokoll, das man beschreiben möchte, eine eigene Ereignisquelle benötigt.

```
// Erzeugung eines neuen Eintrags in einem Ereignisprotokoll
public void EreignisprotokollEintrag_Schreiben()
{
    string logname = "N2C-Buch";
    string source = "N2C-StartStop";
    string computer = "Essen";
    // Quelle und ggf. Ereignisprotokoll anlegen
    if (!EventLog.SourceExists(source, computer))
    {
        EventSourceCreationData escd = new EventSourceCreationData(source, logname);
        escd.MachineName = computer;
        EventLog.CreateEventSource(escd);
        Demo.Print("Quelle angelegt!");
    }
    // Eintrag schreiben
    EventLog.WriteEntry(source, "Anwendung gestartet", EventLogEntryType.Information, 1234);
    // Bildschirmausgabe
    Demo.Print("Eintrag geschrieben!");
}
```

Listing 9.27 Schreiben in ein anwendungsspezifisches Ereignisprotokoll [Eventlog.cs]

System.Net

Der Namensraum System.Net enthält die Unterstützung für verschiedene Protokolle aus der TCP/IP-Protokollfamilie. Insbesondere werden Internet Control Message Protocol (ICMP), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) und Domain Name Service (DNS) unterstützt. Auch die direkte Socket-Programmierung ist möglich. Außerdem können Sie statistische Informationen aus dem TCP/IP-Protokoll erhalten.

Neuheiten seit .NET 2.0

In diesem Namensraum gibt es zahlreiche Neuerungen seit .NET 2.0, insbesondere

- FTP-Unterstützung (System.Net.FtpWebRequest und System.Net.FtpWebResponse)
- Statistische Informationen aus dem TCP/IP-Protokoll (System.Net.NetworkInformation.*)
- SSL-Unterstützung (System.Net.Security.SslStream)
- Zugriff auf Netzwerk-Konfigurationsinformationen aus den Anwendungskonfigurationsdateien (System.Net.Configuration.*)
- HTTP-Zwischenspeicherung (System.Net.Cache.*)
- Klasse HttpListener zur Erzeugung eigener HTTP-Server
- Automatische Dekomprimierung von komprimierten HTTP-Nachrichten

Neuheiten seit .NET 3.5

In diesem Namensraum gibt es ab .NET 3.5 folgende Neuerungen:

- Unterstützung für Peer-To-Peer-Netzwerkkommunikation (System.Net.PeerToPeer)

Neuheiten seit .NET 4.0

In diesem Namensraum gibt es ab .NET 4.0 folgende Neuerungen:

- Verbesserung für Windows-Authentifizierung in HttpWebRequest, HttpListener, SmtplibClient, SslStream und NegotiateStream zur Abwehr von Relay-Angriffen (nur für Windows7/Windows Server 2008R2) siehe [MSDN32].
- Verbesserungen für NAT Traversal mit IPv6 und Teredo
- Leistungsindikatoren für HttpWebRequest: *HttpWebRequests Created/Sec*, *WebRequests Average Lifetime*, *HttpWebRequests Queued/Sec*, *HttpWebRequests Average Queue Time*, *HttpWebRequests Aborted/Sec*, *HttpWebRequests Failed/Sec*
- HttpWebRequest: AddRange, Host
- SmtplibClient: SSL-Unterstützung
- MailMessage: Mail-Headers
- NetworkCredential: Akzeptiert auch die seit .NET 2.0 eingeführte Klasse SecureString zur Kennwortübergabe

HTTP- und FTP-Unterstützung

Die bereits in .NET-Version 1.x vorhandene abstrakte Klasse `System.Net.WebRequest`, die bisher nur die Unterklassen `FileWebRequest` und `HttpWebRequest` besaß, hat schon in .NET 2.0 eine neue Abteilung für das FTP-Protokoll erhalten: `FtpWebRequest`. Zwar konnte man mit einigen Codezeilen auch selbst einen FTP-Zugriff implementieren oder auf kommerzielle Komponenten zurückgreifen, mit dem neuen Klassenpärchen `FtpWebRequest` und `FtpWebResponse` ist es jedoch einfacher bzw. kostengünstiger.

Ein `FtpWebRequest`-Objekt kann über die statische Methode `Create()` wahlweise auf der Klasse `FtpWebRequest` oder der Basisklasse `WebRequest` erzeugt werden. In letzterem Fall entscheidet das verwendete URI-Schema darüber, welche der Unterklassen die Methode liefert. Wichtig ist, dass der Entwickler die auszuführende Aktion in der Eigenschaft `Method` hinterlegt, z. B. `ftp.Method = WebRequestMethods.Ftp.UploadFile`.

Beim Herunterladen einer Datei erhält man von dem `FtpWebRequest`-Objekt ein `FtpWebResponse`-Objekt, das wiederum über `GetResponseStream()` die heruntergeladenen Dateiinhalte in einem Stream bereitstellt.

```
FtpWebResponse response = ftp.GetResponse();  
Stream responseStream = response.GetResponseStream();
```

Beim Heraufladen liefert das `FtpWebRequest`-Objekt über `GetRequestStream()` einen Datenstrom, den der Entwickler beschreiben kann.

```
Stream requestStream = ftp.GetRequestStream();
```

Beispiele

Die beiden folgenden Beispiele zeigen den Abruf einer Webseite per HTTP und das Hochladen einer Datei per FTP.

```
// Absenden einer HTTP-Anfrage  
public void HTTP_Anfrage()  
{  
    const string URL = "http://www.it-visions.de";  
    // Anfrage definieren  
    HttpWebRequest frage = (HttpWebRequest)HttpWebRequest.Create(URL);  
    // Antwort holen  
    HttpWebResponse antwort = (HttpWebResponse)frage.GetResponse();  
    // Metadaten  
    Demo.Print("Antwortlänge: " + antwort.ContentLength)  
    Demo.Print("Status: " + antwort.StatusCode);  
    Demo.Print("Letzte Änderung: " + antwort.LastModified)  
    Demo.Print("Inhaltstyp: " + antwort.ContentType);  
    // Inhalt ausgeben  
    StreamReader sr = new StreamReader(antwort.GetResponseStream());  
    Demo.Print(sr.ReadToEnd());  
}
```

Listing 9.28 Abruf einer Webseite per http [Netzwerk.cs]

```
// Inhalt eines FTP-Verzeichnisses auflisten
public void FTPInhaltAuflisten()
{
    const string URL = @"ftp://Server02/";
    StreamReader reader = null;
    FtpWebRequest ftp = (FtpWebRequest)WebRequest.Create(URL);
    ftp.Credentials = new NetworkCredential("hs", "geheim");
    ftp.Method = WebRequestMethods.Ftp.ListDirectoryDetails;
    FtpWebResponse Response = (FtpWebResponse)ftp.GetResponse();
    reader = new StreamReader(Response.GetResponseStream());
    Demo.Print(reader.ReadToEnd());
    Demo.Print("Fertig!");
    reader.Close();
}
```

Listing 9.29 Auflisten eines FTP-Verzeichnisses [Netzwerk.cs]

```
// Herunterladen einer Datei per FTP
public void FTPDownload()
{
    const string URL = @"ftp://Server02/Flugdaten.xml";
    const string LOKALERPFAD = @"c:\temp\Flugdaten_download.xml";
    FtpWebRequest ftp = (FtpWebRequest)FtpWebRequest.Create(URL);
    ftp.Method = WebRequestMethods.Ftp.DownloadFile;
    ftp.Credentials = new NetworkCredential("hs", "geheim");
    FtpWebResponse response = (FtpWebResponse) ftp.GetResponse();
    Stream responseStream = response.GetResponseStream();
    Demo.Print("Lade Datei...");
    FileStream fileStream = null;
    // Öffnen der Zieldatei
    fileStream = File.Create(LOKALERPFAD);
    byte[] buffer = new byte[1024];
    int bytesRead;
    // Einlesen und in Datei kopieren
    while (true)
    {
        bytesRead = responseStream.Read(buffer, 0, buffer.Length);
        if (bytesRead == 0)
            break;
        fileStream.Write(buffer, 0, bytesRead);
        Console.Write(".");
    }
    // Alles schließen!
    responseStream.Close();
    fileStream.Close();
    Demo.Print("Fertig!");
}
```

Listing 9.30 Herunterladen einer Datei per FTP [Netzwerk.cs]

```
// Heraufladen einer Datei per FTP
public void FTPUpload()
{
    const string URL = @"ftp://Server02/Flugdaten.xml";
```

```

const string LOKALERPFAD = @"c:\temp\Flugdaten.xml";
// --- Anfrage erstellen
FtpWebRequest ftp = (FtpWebRequest)WebRequest.Create(URL);
ftp.Method = WebRequestMethods.Ftp.UploadFile;
ftp.Credentials = new NetworkCredential("hs", "geheim");
ftp.Timeout = System.Threading.Timeout.Infinite;
Stream requestStream = ftp.GetRequestStream();
Demo.Print("Sende Datei...");
// --- Kopieren des Inhalts aus der Datei
const int bufferLength = 2048;
byte[] buffer = new byte[bufferLength];
int count = 0;
int readBytes = 0;
// Öffnen der Quelldatei
FileStream fileStream = File.OpenRead(LOKALERPFAD);
// Einlesen der Quelldatei
do
{
    readBytes = fileStream.Read(buffer, 0, bufferLength);
    requestStream.Write(buffer, 0, bufferLength);
    count += readBytes;
    Console.Write(".");
}
while (readBytes != 0);
// --- Antworten holen
requestStream.Close();
FtpWebResponse response = (FtpWebResponse)ftp.GetResponse();
Demo.Print("Fertig!");
response.Close();
fileStream.Close();
}

```

Listing 9.31 Heraufladen einer Datei per FTP [Netzwerk.cs]

System.Net.NetworkInformation

Der seit .NET 2.0 neu eingeführte Unterraum `NetworkInformation` bietet statistische Informationen aus dem TCP/IP-Protokoll sowie die in .NET 1.x nicht (bzw. nur durch WMI) vorhandene Unterstützung für das Pingen.

Beispiel 1: Ausführung eines Pings

Das Beispiel zeigt die asynchrone Ausführung eines Pings über das Internet Control Message Protocol (ICMP).

```

// Ping ausführen
public void Run_Ping()
{
    const string COMPUTER = "www.IT-Visions.de";
    // Ping synchron ausführen
    Ping p = new Ping();
    PingReply pr = p.Send(COMPUTER);
    Console.WriteLine(pr.Status + ";" + pr.RoundtripTime);
}

```

```
// Ping asynchron ausführen
p.PingCompleted += new PingCompletedEventHandler(p_PingCompleted);
p.SendAsync("Duisburg", null);
}
// Callback für Ping
static void p_PingCompleted(object sender, PingCompletedEventArgs e)
{ Console.WriteLine(e.Reply.Status + ";" + e.Reply.RoundtripTime + "ms");
}
```

Listing 9.32 Ausführung eines Pings [Netzwerk.cs]

Beispiel 2: Netzwerkdatenverkehrstatistik

Die folgende Routine liefert statistische Daten aus dem TCP/IP-Protokoll-Stack.

```
// Statistische Daten aus dem TCP/IP-Protokoll-Stack
public void NetStatistik()
{
    IPGlobalProperties ipgp = IPGlobalProperties.GetIPGlobalProperties();
    TcpStatistics t = ipgp.GetTcpIPv4Statistics();
    Demo.Print(t.CurrentConnections);
    Demo.Print(t.ErrorsReceived);
    Demo.Print(t.MaximumConnections);
    Demo.Print(t.SegmentsReceived);
    Demo.Print(t.SegmentsSent);
}
```

Listing 9.33 Statistik aus dem TCP/IP-Stack [Netzwerk.cs]

Netzwerkstatus

`NetworkInterface.GetAllNetworkInterfaces()` liefert ein Array von `NetworkInterface`-Objekten, die Auskunft über den Status des Netzwerks geben.

```
public void NetStatus()
{
    NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
    foreach (NetworkInterface n in adapters)
    {
        Console.WriteLine("Netzwerkstatus: {0} = {1}", n.Name, n.OperationalStatus);
    }
}
```

Listing 9.34 Ausgabe des Netzwerkstatus [Netzwerk.cs]

Die Klasse `NetworkChange` stellt die Ereignisse `NetworkAvailabilityChanged()` und `NetworkAddressChanged()` bereit, mit denen Veränderungen der Netzwerkverfügbarkeit überwacht werden können.

```

public void NetStatusWarten()
{
    NetStatus();
    NetworkChange.NetworkAvailabilityChanged +=
        new NetworkAvailabilityChangedEventHandler(NetworkChange_NetworkAvailabilityChanged);
    Console.WriteLine("Überwachung...");
    Console.ReadLine();
}
void NetworkChange_NetworkAvailabilityChanged(object sender, NetworkAvailabilityEventArgs e)
{
    NetStatus();
}

```

Listing 9.35 Warten auf eine Änderung im Netzwerkstatus [Netzwerk.cs]

System.Net.Mail und System.Net.Mime

Im .NET Framework 1.x existierte bereits die Unterstützung für das Senden von SMTP-E-Mail-Nachrichten. Allerdings war diese Funktion versteckt im Namensraum `System.Web.Mail` – einige Entwickler von Desktop-Anwendungen trauten sich nicht, die implementierende *System.Web.dll* in ihren Anwendungen zu referenzieren.

Seit .NET 2.0 hat Microsoft nun in `System.Net.Mail` und `System.Net.Mime` eine alternative Möglichkeit zum Mail-Versand in einem neutralen Namensraum untergebracht, die außerdem mehr Optionen bietet. Insbesondere besteht die Möglichkeit, eine Nachricht auch direkt an das */Drop*-Verzeichnis eines SMTP-Dienstes der Internet Information Services (IIS) zu übergeben (*DeliveryMethod*). Authentifizierung ist möglich über die Eigenschaft *Credentials* (siehe Listing 9.36).

Um die Abwärtskompatibilität sicherzustellen, bleibt der Namensraum `System.Web.Mail` erhalten. Hier findet man also ein gutes Beispiel dafür, wie strategische Fehlentscheidungen bei der Implementierung einer Klassenbibliothek in der Zukunft zu Verwirrung bei den Entwicklern führen können, wenn diese sich nun fragen werden, warum es denn zwei ähnliche Klassen zum E-Mail-Versand gibt.

```

public void MailSenden_Demo()
{
    Console.WriteLine("Senden einer E-Mail...");
    const string server = "Server121.IT-Visions.net";
    // --- Nachricht erzeugen
    MailMessage message = new MailMessage(
        "hs@Server121.IT-Visions.net", "hs@Server121.IT-Visions.net",
        "Betreff", "Body");
    // Anhang hinzufügen
    string file = @"d:\temp\flugdaten.xml";
    Attachment attach = new Attachment(file);
    ContentDisposition disposition = attach.ContentDisposition;
    disposition.CreationDate = System.IO.File.GetCreationTime(file);
    disposition.ModificationDate = System.IO.File.GetLastWriteTime(file);
    disposition.ReadDate = System.IO.File.GetLastAccessTime(file);
    message.Attachments.Add(attach);
    // Nachricht senden
    SmtpClient client = new SmtpClient(server);
    client.DeliveryMethod = SmtpDeliveryMethod.Network;
}

```

```
//client.Credentials = new NetworkCredential("benutzer", "kennwort");
client.Send(message);
// Ende
Console.WriteLine("Nachricht gesendet!");
}
```

Listing 9.36 Senden einer E-Mail [System.Net.Mail.cs]

System.Text

Der Namensraum System.Text realisiert zwei Funktionen im Zusammenhang mit Texten:

- Umwandlung von Texten in Zeichencodes und umgekehrt
- Suchen und Ersetzen in Texten mit regulären Ausdrücken

Textcodierung

System.Text enthält die Implementierung verschiedener Zeichencodes.

- ASCIIEncoding (Unicode Code Page 20127)
- UTF7Encoding (Unicode Code Page 65000)
- UTF8Encoding (Unicode Code Page 65001)
- UnicodeEncoding (Little-endian Unicode Code Page 1200 + Big-endian Unicode Code Page 1201)
- UTF32Encoding (Little-endian Unicode Code Page 65005 + Unicode Code Page 65006)

Das folgende Beispiel zeigt die Umwandlung einer Zeichenkette in die entsprechende ASCII-Byte-Folge mit der Klasse ASCIIEncoding.

```
public void Encoding_Test()
{
    byte[] NameAlsBytes;
    string Name = "Dr. Holger Schwichtenberg";
    Demo.Print("--- Umwandlung zwischen Bytes und Zeichenkette");
    ASCIIEncoding Encoder = new ASCIIEncoding();
    NameAlsBytes = Encoder.GetBytes(Name);
    Demo.Print("Dieser Name als Byte-Folge:");
    foreach (byte b in NameAlsBytes)
    {
        Console.Write(b + " ");
    }
    Demo.Print("");
    Name = Encoder.GetString(NameAlsBytes);
    Demo.Print("Ursprünglicher Name:" + Name)
}
```

Listing 9.37 Textcodierung [Text.cs]

Reguläre Ausdrücke

Der Namensraum `System.Text.RegularExpressions` bietet Unterstützung für reguläre Ausdrücke zum Mustervergleich und Ersetzen in Zeichenketten. Die Klasse `Regex` stellt mit `IsMatch()` und `Replace()` zwei einfache Methoden bereit. Wenn ein regulärer Ausdruck mehrfach verwendet werden soll, ist es aus Leistungsgründen sinnvoll, ein `Regex`-Objekt mit dem regulären Ausdruck zu instanziiieren und dann die verschiedenen Eingaben mit den Methoden `Match()` und `Replace()` anzuwenden.

Beispiel 1: Mustervergleich

In dem folgenden Beispiel prüft ein regulärer Ausdruck, ob die Eingabezeichenkette ein Global Unique Identifier (GUID) oder eine E-Mail-Adresse ist.

```
public void Mustervergleich()
{
    const string RA_GUID = @"\[0-9a-fA-F\]{8}-\[0-9a-fA-F\]{4}-\[0-9a-fA-F\]{4}-\[0-9a-fA-F\]{4}-\[0-9a-fA-F\]{12}\\";
    const string RA_IPAdresse = @"\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b";
    const string RA_Email = @"^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$";
    const string EINGABE1 = @"{00000615-0000-0010-8000-00AA006D2EA4}";
    const string EINGABE2 = @"192.168.123.355"; // Fehler!
    const string EINGABE3 = @"hs@IT-Visions.de";
    Demo.Print("GUID korrekt? " + Regex.IsMatch(EINGABE1, RA_GUID));
    Demo.Print("IP-Adresse korrekt? " + Regex.IsMatch(EINGABE2, RA_IPAdresse));
    Demo.Print("E-Mail-Adresse korrekt? " + Regex.IsMatch(EINGABE3, RA_Email));
}
```

Listing 9.38 Mustervergleich [Text.cs]

Beispiel 2: Musterersatz

Im zweiten Beispiel wird eine Datumszeichenkette der Form 08/01/1972 umgewandelt in 01-08-1972.

```
public void Musterersatz()
{
    const String EINGABE = @"08/01/1972";
    const string MUSTER = @"\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b";
    const string ERSATZ = @"${day}-${month}-${year}";
    Demo.Print("Alt: " + EINGABE);
    Demo.Print("Neu: " + Regex.Replace(EINGABE, MUSTER, ERSATZ));
}
```

Listing 9.39 Musterersatz [Text.cs]

Serialisierung

Der Begriff *Serialisierung* bezeichnet die Umwandlung des Zustands eines Objekts in eine Folge von Bytes. *Deserialisierung* ist der umgekehrte Vorgang, bei dem aus einer Byte-Folge wieder ein programmierbares Objekt erzeugt wird. Dabei wird der ursprüngliche Zustand des Objekts wiederhergestellt. Serialisierung und Deserialisierung sind als Synonym für die beim Remote Procedure Call (RPC) verwendeten Begriffe *Marshaling* und *Unmarshaling* zu sehen.

Wann immer Objekte zwischen zwei Umgebungen ausgetauscht werden müssen, die keinen gemeinsamen Speicher besitzen, ist die Serialisierung und spätere Deserialisierung des Objekts notwendig. Im .NET Framework besteht eine solche Barriere nicht nur zwischen Computern und Prozessen, sondern auch zwischen Application Domains. Daneben wird die Serialisierung/Deserialisierung auch benötigt, wenn Objektpersistenz erreicht werden soll, also Objekte (dauerhaft) auf einem Medium gespeichert werden sollen.

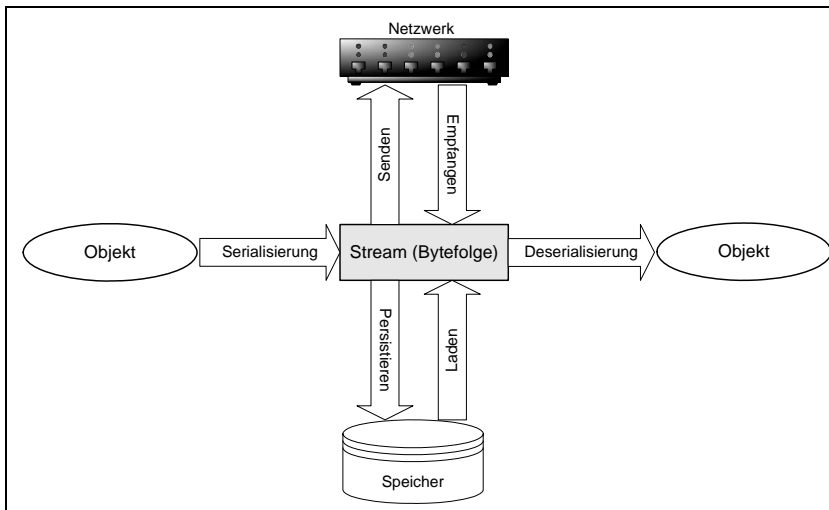


Abbildung 9.9 Serialisierung, Persistenz und Datenübertragung

HINWEIS

Dieses Kapitel ist nicht mit einem Namensraum der .NET-Klassenbibliothek überschrieben, weil hier Klassen aus verschiedenen Namensräumen behandelt werden.

Serialisierer

Eine Klasse, die der Serialisierung von Objekten dient, heißt *Serialisierer*. Die .NET Framework Class Library Version bietet fünf verschiedene Serialisierer mit unterschiedlichen Eigenschaften an:

- `XmlSerializer`
- `BinaryFormatter`
- `SoapFormatter`
- `DataContractSerializer`
- `NetDataContractSerializer`

HINWEIS Die ersten drei gibt es seit .NET 1.0, die letzten beiden seit .NET 3.0. Sie wurden im Zuge der Einführung von WCF hinzugefügt. `NetDataContractSerializer` und `DataContractSerializer` liegen im Namensraum `System.Runtime.Serialization`, sind aber nicht kompatibel mit den bisher dort geführten und in .NET Remoting verwendeten Klassen `BinaryFormatter` und `SoapFormatter`. Die neuen Serialisierer basieren nicht auf der `IFormatter`-Schnittstelle.

BinaryFormatter

Der `BinaryFormatter` ist einer der beiden Serialisierer für .NET Remoting. Das Format ist proprietär, d.h. nicht standardisiert. Eine Wiedergabe des Formats ist hier nicht möglich.

Der `BinaryFormatter` wird auch außerhalb von .NET Remoting gerne zur Serialisierung eingesetzt, weil das Format kompakt ist. Beispielsweise nutzt auch Windows Workflow Foundation (WF) den Formatter, um Workflows in einer Datenbank zu persistieren.

SoapFormatter

Der `SoapFormatter` ist der zweite Serialisierer für das .NET Remoting. Er erzeugt immer *RPC/Encoded SOAP*, also nur eine der vier im Standard vorgesehenen Varianten. Der `SoapFormatter` hat heute kaum noch Bedeutung. Daher wird darauf nicht näher eingegangen.

```
- <SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
  ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
  ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
  ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
- <SOAP-ENV:Body>
- <a1:Flug id="ref-1"
  xmlns:a1="http://schemas.microsoft.com/clr/nsassem/de.WWWings/WWWings.GL.CS%
  2C%20Version%3D0.5.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%
  3Df7cce9d321b21deb">
  <Status>neu</Status>
  <flugNr>101</flugNr>
  <abflugOrt id="ref-3">Berlin</abflugOrt>
  <zielOrt id="ref-4">Frankfurt</zielOrt>
  <plaetze>250</plaetze>
  <freiePlaetze>111</freiePlaetze>
  <nichtraucherflug>false</nichtraucherflug>
  <_Datum>2008-07-04T03:10:25.0000000+02:00</_Datum>
  <MitarbeiterNr>0</MitarbeiterNr>
  <MarshalByRefObject_x002B___identity xsi:type="xsd:anyType" xsi:null="1" />
  </a1:Flug>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Abbildung 9.10 Serialisierung eines Flugobjekts mit dem `SoapFormatter`

XmlSerializer

Der `XmlSerializer` ist der bereits in .NET 1.0 eingeführte und von ASP.NET-basierten XML-Webservices verwendete Serialisierer. Der `XmlSerializer` serialisiert alle öffentlichen Field-Attribute und öffentlichen Property-Attribute einer Klasse, ignoriert jedoch alle privaten Mitglieder. Ausgenommen werden nur diejenigen Felder, die explizit mit `[XmlIgnoreAttribute]` annotiert sind.

```

<?xml version="1.0" ?>
- <Flug xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <FlugNr>101</FlugNr>
  <AbflugOrt>Berlin</AbflugOrt>
  <ZielOrt>Stuttgart</ZielOrt>
  <Plaetze>250</Plaetze>
  <FreiePlaetze>0</FreiePlaetze>
  <Nichtraucherflug>false</Nichtraucherflug>
  <Datum>2006-01-14T12:46:02</Datum>
</Flug>

```

Abbildung 9.11 Serialisierung eines Flugobjekts mit dem XmlSerializer

DataContractSerializer

Microsoft hat sich entschlossen, in WCF einen neuen Serialisierer zu entwickeln, der ein expliziteres Modell hinsichtlich der zu veröffentlichenden Datenelemente sowie die Serialisierung von Objektbäumen mit zirkulären Referenzen unterstützt. Der DataContractSerializer ist der Standard-Serialisierer in WCF.

```

- <Flug xmlns="http://schemas.datacontract.org/2004/07/de.WWWings"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <__identity i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System" />
  <MitarbeiterNr>0</MitarbeiterNr>
  <Status>neu</Status>
  <_Datum>2006-01-14T12:46:02</_Datum>
  <abflugOrt>Berlin</abflugOrt>
  <flugNr>101</flugNr>
  <freiePlaetze>0</freiePlaetze>
  <nichtraucherflug>false</nichtraucherflug>
  <pilot i:nil="true"
    xmlns:a="http://schemas.datacontract.org/2004/07/de.WWWings.MitarbeiterSystem" />
  <plaetze>250</plaetze>
  <zielOrt>Stuttgart</zielOrt>
</Flug>

```

Abbildung 9.12 Serialisierung eines Flug-Objekts mit DataContractSerializer

NetDataContractSerializer

NetDataContractSerializer und DataContractSerializer sind fast identisch. Der NetDataContractSerializer unterscheidet sich in drei Punkten vom DataContractSerializer:

- Der NetDataContractSerializer nimmt Typinformationen über die der Serialisierung zugrunde liegenden CLR-Klassen mit in den Serialisierungsstrom auf, was erfordert, dass die Gegenseite diese CLR-Klassen kennt. Dies widerspricht zwar dem Prinzip der Entkopplung, sorgt aber in reinen .NET-Umgebungen dafür, dass Client und Server mit den gleichen Datentypen arbeiten können.
- Beim Instanzieren der Serialisiererklasse müssen die CLR-Typen nicht angegeben werden. Diese ermittelt der Serialisierer zur Laufzeit selbstständig.
- Objektbäume mit zirkulären Referenzen können serialisiert werden ohne eine zusätzliche Einstellung

```

- <Flug z:Id="1" z:Type="de.WWWings.Flug" z:Assembly="WWWings.GL.CS,
  Version=1.0.2570.22303, Culture=neutral, PublicKeyToken=f7cce9d321b21deb"
  xmlns="http://schemas.datacontract.org/2004/07/de.WWWings"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <__identity i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System" />
  <MitarbeiterNr>0</MitarbeiterNr>
  <Status>neu</Status>
  <_Datum>2006-01-14T12:46:02</_Datum>
  <abflugOrt z:Id="2">Berlin</abflugOrt>
  <flugNr>101</flugNr>
  <freiePlaetze>0</freiePlaetze>
  <nichtraucherflug>false</nichtraucherflug>
  <pilot i:nil="true"
    xmlns:a="http://schemas.datacontract.org/2004/07/de.WWWings.MitarbeiterSystem" />
  <plaetze>250</plaetze>
  <zielOrt z:Id="3">Stuttgart</zielOrt>
</Flug>

```

Abbildung 9.13 Serialisierung eines Flug-Objekts mit NetDataContractSerializer

Einsatz der Serialisierer

Die fünf Serialisierer besitzen keine gemeinsame Oberklasse (lediglich die beiden neueren Serialisierer haben eine gemeinsame Oberklasse). Dies bedeutet, dass die Serialisierer unterschiedlich angesprochen werden. Allerdings lassen sich diese Unterschiede leicht kapseln, wie die folgende Klasse `UniversalSerialisierer` mit den beiden Methoden `Save()` und `Load()` zeigt. `Save()` speichert ein Objekt bzw. eine Objekthierarchie in einer Datei ab. `Load()` lädt sie wieder aus der Datei.

```

public enum SerialTyp : int
{
    SOAP,
    XML,
    BINARY,
    DATACONTRACT,
    NETDATACONTRACT
}

public class UniversalSerialisierer
{
    // ### Serialisieren in Datei
    public static void Save(SerialTyp Typ, object obj, string datei)
    {
        // Datei öffnen
        FileStream stream = null;
        stream = new FileStream(datei, FileMode.Create, FileAccess.Write, FileShare.None);
        // Fallunterscheidung
        object Serializer = null;
        switch (Typ)
        {
            case SerialTyp.BINARY:
                Serializer = new BinaryFormatter();
                ((BinaryFormatter)Serializer).Serialize(stream, obj);
                break;

```

```

case SerialTyp.SOAP:
    Serializer = new SoapFormatter();
    ((SoapFormatter)Serializer).Serialize(stream, obj);
    break;
case SerialTyp.XML:
    Serializer = new XmlSerializer(obj.GetType());
    ((XmlSerializer)Serializer).Serialize(stream, obj);
    break;
case SerialTyp.DATACONTRACT:
    DataContractSerializer dcs = new DataContractSerializer(obj.GetType());
    dcs.WriteObject(stream, obj);
    stream.Close();
    break;
case SerialTyp.NETDATACONTRACT:
    XmlDictionaryWriter writer2 = XmlDictionaryWriter.CreateTextWriter(stream);
    NetDataContractSerializer ser2 = new NetDataContractSerializer();
    ser2.WriteObject(writer2, obj);
    writer2.Close();
    break;
default:
    MessageBox.Show("Nicht unterstütztes Serialisierungsformat!");
    System.Environment.Exit(1);
    break;
}
// Datei schließen
stream.Close();
// Ausgabe
Demo.Print("Objekt wurde gespeichert in " + datei);
}

// ### Deserialisieren aus Datei
public static object Load(SerialTyp typ, string Datei)
{
    return load(typ, Datei, null);
}

// ### Deserialisieren aus Datei
public static object load(SerialTyp typ, string Datei, Type Objekttyp)
{
    // Datei öffnen
    FileStream stream = null;
    stream = new FileStream(Datei, FileMode.Open);
    // Fallunterscheidung
    object Serializer = null;
    // Ergebnis
    object o = null;

    switch (typ)
    {
        case SerialTyp.BINARY:
            Serializer = new BinaryFormatter();
            o = ((BinaryFormatter)Serializer).Deserialize(stream);
            break;
        case SerialTyp.SOAP:
            Serializer = new SoapFormatter();

```

```

    o = ((SoapFormatter)Serializer).Deserialize(stream);
    break;
case SerialTyp.XML:
    if (Objekttyp == null)
    {
        MessageBox.Show("Fehler: Für den XML-Serialisierer muss der zu " +
            "deserialisierende Objekttyp bekannt sein!");
        System.Environment.Exit(1);
    }
    Serializer = new XmlSerializer(Objekttyp);
    o = ((XmlSerializer)Serializer).Deserialize(stream);
    break;
case SerialTyp.DATACONTRACT:
    XmlDictionaryReader reader =
        XmlDictionaryReader.CreateBinaryReader(stream, new XmlDictionaryReaderQuotas());
    DataContractSerializer ser2 = new DataContractSerializer(Objekttyp);
    o = ser2.ReadObject(reader, true);
    break;
case SerialTyp.NETDATACONTRACT:
    XmlDictionaryReader reader2 =
        XmlDictionaryReader.CreateTextReader(stream, new XmlDictionaryReaderQuotas());
    NetDataContractSerializer ser = new NetDataContractSerializer();
    o = ser.ReadObject(reader2, true);
    break;
default:
    MessageBox.Show("Nicht unterstütztes Serialisierungsformat!");
    System.Environment.Exit(1);
    break;
}

// Datei schließen
stream.Close();
// Ausgabe
Demo.Print("Objekt wurde geladen aus " + Datei);
// Objekt zurückliefern
return o;
}
}

```

Listing 9.40 Die Klasse UniversalSerialisierer kapselt die Handhabung aller fünf Serialisierungen in Bezug auf das Speichern und Laden (für die Persistenz in Dateien im Dateisystem)

Serialisierbarkeit

Je nach Serialisierer und Anwendungsgebiet können alle oder nur bestimmte Klassen serialisiert werden. Die Serialisierbarkeit eines Objekts ist nur möglich, wenn alle zu übertragenden Datenmitglieder auch serialisierbar sind.

Man unterscheidet zwischen der expliziten Festlegung der Serialisierbarkeit durch Annotation und der automatischen Serialisierbarkeit (alias POCO-Serialisierung).

Explizite Festlegung der Serialisierung

Es gibt folgende Möglichkeiten der expliziten Festlegung der Serialisierbarkeit einer Klasse:

- Die Klasse ist mit `[System.Serializable]` annotiert oder
- die Klasse ist mit `[System.Runtime.Serialization.DataContract]` annotiert oder
- die Klasse ist mit `[System.Runtime.Serialization.CollectionDataContractAttribute]` annotiert oder
- die Klasse implementiert die Schnittstelle `ISerializable`

Die Steuerung der zu serialisierenden Mitglieder erfolgt

- bei der Verwendung von `[Serializable]` über `[NonSerializedAttribute]`, d.h., man muss alle Mitglieder zusätzlich annotieren, die *nicht serialisiert* werden sollen;
- bei der Verwendung von `[DataContract]` über `[DataMember]`, d.h., man muss alle Mitglieder zusätzlich annotieren, die *serialisiert* werden sollen.

ACHTUNG Nicht alle Serialisierer unterstützen alle Formen (siehe Vergleichstabelle in einem späteren Abschnitt).

WICHTIG Falls ein Typ sowohl mit `[Serializable]` als auch mit `[DataContract]` annotiert ist, dann hat `[DataContract]` Vorrang. Mit `[DataMember]` annotierte Properties müssen immer sowohl einen Getter als auch einen Setter besitzen. Sonst kommt es zu einem Fehler des Typs `InvalidDataContractException`.

Die elementaren Datentypen, die im .NET Framework enthalten sind (z.B. `System.String`, `System.Int32`, `System.DateTime`), sind bereits als serialisierbar gekennzeichnet.

HINWEIS Sobald Polymorphismus im Spiel ist (z.B. weil eine Klasse von einer anderen Klasse als `System.Object` erbt oder eine Klasse ein Datenmitglied besitzt, hinter dem sich mehrere verschiedene Klassen verbergen können), ist zu beachten, dass dem Serialisierer ein Hinweis auf alle Klassen gegeben werden muss, die dort erwartet werden können. Dies geschieht im Fall des XML-Serialisierers mit der Annotation `[XmlInclude(GetType(Klasse))]` und im Fall des `DataContractSerializer` mit `[KnownType(GetType(Klasse))]`. Die anderen drei Serialisierer können dieses Szenario automatisch auflösen.

Häufig tritt das Problem bei Objektmengen auf, die nicht streng typisiert sind. Hier ist es ratsam, typisierte Objektmengen (siehe Abschnitt »System.Collections«) zu verwenden, bei denen das Problem gar nicht auftreten kann.

POCO-Serialisierung

Ein Plain Old CLR Object (POCO) ist ein ganz normales .NET-Objekt, das keine durch die Infrastruktur bedingte Basisklasse, Annotationen oder eine Erweiterung auf Bytecode-Ebene (MSIL/CIL) erfordert. POCO-Serialisierung heißt also, dass eine Klasse nicht explizit als *serialisierbar* gekennzeichnet werden muss.

POCO-Serialisierung unterstützen XML-Serialisierer (`XmlSerializer`) und seit .NET 3.5 Service Pack 1 auch die Klassen `DataContractSerializer` und `NetDataContractSerializer`.

Bei der POCO-Serialisierung sind vier Dinge zu beachten:

- In diesem Fall werden nur öffentliche *Field*- und *Property*-Attribute serialisiert
- Die Attribute müssen lesbar (einen Getter besitzen) und beschreibbar (einen Setter besitzen) sein

- Die Klasse braucht einen parameterlosen Konstruktor
- Die POCO-Serialisierung ist deaktiviert im `DataContractSerializer`, sobald eine Klasse eine der o.g. Annotationen besitzt. Der `XmlSerializer` ignoriert diese Annotationen aber und macht immer POCO-Serialisierung

HINWEIS Der Vorteil der POCO-Serialisierung ist, dass man beliebige Typen serialisieren kann, auch wenn man diese nicht selbst geschrieben hat und keinen Einfluss auf die Ausstattung mit Annotationen hat. Der Nachteil ist, dass man bei der POCO-Serialisierung keinen Einfluss auf die Struktur des Serialisierungsformats hat.

Beispiel für ein serialisierbares Objektmodell

Das folgende Listing implementiert ein Objektmodell (siehe Abbildung), das für alle Serialisierungsfälle gerüstet ist, weil es sowohl `[Serializable]` als auch `[DataContract]` als Annotation bietet.

HINWEIS Es wäre wesentlich besser, in dem Objektmodell anstelle von `ArrayList` typisierte Listen (`List<Typ>`) zu verwenden. Allerdings wurde dies hier aus didaktischen Gründen nicht gemacht, denn dadurch sieht man, wie dem Serialisierer Zusatzinformationen in Form der Annotation `[XmlInclude]` und `[KnownType]` gegeben werden müssen.

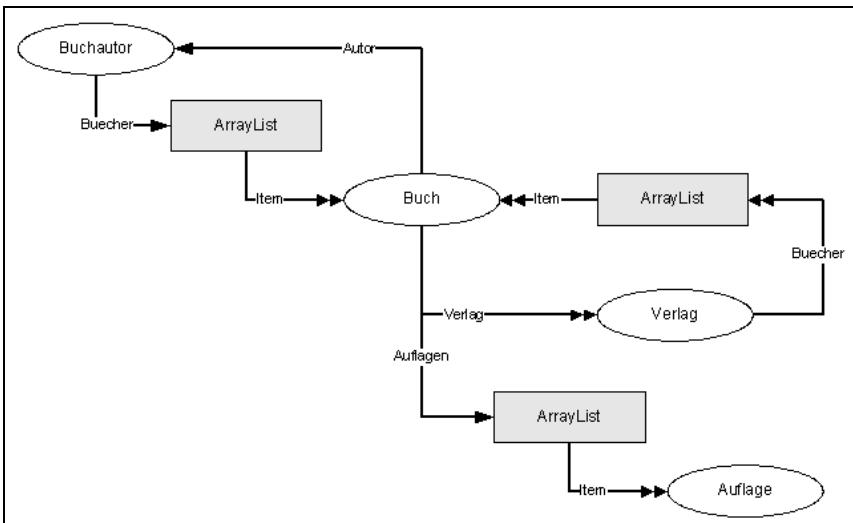


Abbildung 9.14 Das Objektmodell, das im folgenden Listing implementiert wird

```
// =====
[Serializable]
public class Autor
{
    public string Name;
    public char Geschlecht;
    private System.DateTime _Geb;
    public string[] Schwerpunkte;
}
```

```

public string Wohnort { get; set; }

private int _PLZ;
public int PLZ
{
    get
    {
        return _PLZ;
    }
    set
    {
        _PLZ = value;
    }
}

public int Alter
{
    get
    {
        return (DateTime.Now.Year - _Geb.Year);
    }
}

public DateTime Geb
{
    set
    {
        Demo.Print("Geb setzen...");
        _Geb = value;
    }
    get
    {
        Demo.Print("Geb lesen...");
        return _Geb;
    }
}

public Autor()
{
    Demo.Print("Parameterloser Kontruktor der Klasse 'Autor' aufgerufen...");
}

public Autor(string Name)
{
    Name = Name;
    Demo.Print("Kontruktor der Klasse 'Autor' aufgerufen...");
}
}

// =====

[Serializable(), XmlInclude(typeof(Auflage))]
[KnownType(typeof(Auflage))]
public class Buch

```

```

{
    public string Titel;
    public Autor Autor;
    public ArrayList Auflagen = new ArrayList();
    [NonSerialized()]
    public System.DateTime Deserialisiert;
    public Verlag Verlag;

    public Buch()
    {

    }
    public Buch(string Titel)
    {
        this.Titel = Titel;
    }
}

// =====

[Serializable()]
[XmlInclude(typeof(Buch))]
[KnownType(typeof(Buch))]
public class Buchautor : Autor
{
    public ArrayList buecher = new ArrayList();

    public Buchautor(string Name)
    {
        this.Name = Name;
        Demo.Print("Kontruktor der Klasse 'Buchautor' aufgerufen...");
    }

    public Buchautor()
    {
        Demo.Print("Parameterloser Kontruktor der Klasse 'Buchautor' aufgerufen...");
    }
}

// =====

[Serializable(), XmlInclude(typeof(Buch))]
[KnownType(typeof(Buch))]
public class Verlag
{
    public string Name;
    public string Ort;
    public ArrayList Buecher = new ArrayList();

    public Verlag()
    {
    }
}

```

```
public Verlag(string Verlagsname)
{
    Name = Verlagsname;
}

// =====

[Serializable()]
public class Auflage
{
    public byte Nr;
    public int Jahr;
    public string ISBn;

    public Auflage()
    {
    }

    public Auflage(byte Nr, int Jahr, string ISBN)
    {
        this.Nr = Nr;
        this.Jahr = Jahr;
        this.ISBn = ISBN;
    }
}
```

Listing 9.41 Beispiel für ein zur Serialisierung annotiertes Objektmodell

Vergleich der Serialisierer

Die folgende Tabelle zeigt alle fünf Serialisierer im Vergleich. Wichtige Kriterien sind dabei:

- Was wird serialisiert? (*Field*- und/oder *Property*-Attribute, nur öffentliche oder auch private Mitglieder)
- Alle Serialisierer können nicht nur einzelne Objekte, sondern auch Objektbäume serialisieren. Allerdings können nicht alle Serialisierer Objektbäume mit zirkulären Referenzen serialisieren (Nur der `DataContractSerializer` und der `NetDataContractSerializer` können auch Objektbäume mit zirkulären Referenzen serialisieren.)
- Führen Änderungen in der Klassendefinition (z. B. neue Mitglieder) zu einer Inkompatibilität existierender serialisierter Objekte mit der neuen Klasse? In .NET 1.x gab es diese Einschränkung bei `BinaryFormatter` und `SoapFormatter`; dies konnte nur durch die manuelle Abbildung im Rahmen der Schnittstelle `ISerializable` gelöst werden. .NET 2.0 führte als neue Annotation `[OptionalField]` ein, mit der neu hinzugekommene Mitglieder bei der Deserialisierung außer Acht gelassen werden können.

	XML	DataContract (alias Shared Contract)	NetDataContract (alias Shared Type)	Binär	SOAP
Klasse	XmlSerializer	DataContractSerializer	NetDataContractSerializer	BinaryFormatter	SoapFormatter
Einsatzgebiete	XML-Webservices, WCF, Objektpersistenz (eingeschränkt)	WCF, Objektpersistenz	WCF, Objektpersistenz	.NET Remoting, Objektpersistenz	.NET Remoting, XML-Webservices (eingeschränkt), Objektpersistenz
Namensraum	System.Xml.Serialization	System.Runtime.Serialization	System.Runtime.Serialization	System.Runtime.Serialization.Formatters.Binary	System.Runtime.Serialization.Formatters.Soap
Assembly	system.xml.dll	system.runtime.serialization.dll	system.runtime.serialization.dll	mscorlib.dll	system.runtime.serialization.formatters.soap.dll
Oberklasse	System.Object	System.Runtime.Serialization.XmlObjectSerializer	System.Runtime.Serialization.XmlObjectSerializer	System.Object	System.Object
Anweisung zur Verwendung in WCF	[System.ServiceModel.XmlSerializerFormatAttribute]	Keine (Standard)	Relativ aufwändig über ein selbst zu definierendes WCF-Verhalten	Nicht möglich	Nicht möglich
Implementierte Schnittstellen	Keine	Keine	Keine	IRemotingFormatter, IFormatter	IRemotingFormatter, IFormatter
Ausgabeform	Echter Baum von XML-Elementen	Echter Baum von XML-Elementen oder »Flaches« XML mit Verweisen; wahlweise in Textform, MTOM oder kompakter Binärform	»Flaches« XML mit Verweisen; wahlweise in Textform, MTOM oder kompakter Binärform	Von Microsoft entwickeltes binäres Format	rpc/encoded SOAP mit eindeutigen Objektreferenzen
Typ der zu serialisierenden Klasse muss explizit angegeben werden	Ja	Nein	Nein	Nein	Nein
POCO-Serialisierung	Ja, außer bei Vererbung	Optional ab .NET 3.5 Service Pack 1	Nein	Nein	Nein
Voraussetzungen für den zu serialisierenden Typ bei Nicht-POCO-Serialisierung	Keine	[Serializable], [DataContract], [CollectionDataContractAttribute], ISerializable	[Serializable], [DataContract], [CollectionDataContractAttribute], ISerializable	[Serializable]	[Serializable]
Typen der zu serialisierenden Klasse müssen explizit angegeben werden	Ja, im Konstruktor des Serialisierers	Ja, im Konstruktor des Serialisierers	Nein	Nein	Nein

	XML	DataContract (alias Shared Contract)	NetDataContract (alias Shared Type)	Binär	SOAP
Typen von Oberklassen oder abhängigen Klassen müssen explizit angegeben werden	Ja, XmlInclude- (typeof (Klasse))	Ja, im Konstruktor des Serialisierers durch eine Liste <i>KnowTypes</i> oder [ServiceKnownType]-Annotationen	Nein	Nein	Nein
Typ der CLR-Klasse wird mit serialisiert und muss der Gegenstelle bekannt sein	Nein	Nein	Ja	Ja	Ja
Serialisierung öffentlicher Fields	Ja	Ja	Ja	Ja	Ja
Serialisierung privater Fields	Nein	Ja (Nein bei POCO-Serialisierung)	Ja	Ja	Ja
Serialisierung öffentlicher Properties	Ja	Nein (Ja bei POCO-Serialisierung)	Nein	Nein	Nein
Serialisierung privater Properties	Nein	Nein	Nein	Nein	Nein
Standard-Anordnung der Elemente	Wie in Klasse	Alphabetisch	Alphabetisch	Ja	Ja
Aufruf der Property-Routinen	Ja	Nein	Nein	Nein	Nein
Objekthierarchien mit zirkulären Referenzen	Nein	Ja (Optional)	Ja (Standard-einstellung)	Ja	Ja
Generische Klassen	Ja	Ja	Ja	Ja	Nein
Aufruf des parameterlosen Konstruktors bei Deserialisierung	Ja	Nein (Ja, bei POCO-Serialisierung)	Nein	Nein	Nein
Unterstützung für Initialisierung	Konstruktor	Beliebige mit [OnSerializingAttribute], [OnSerializedAttribute], [OnDeserializingAttribute] bzw. [OnDeserializedAttribute] annotierte Methoden	Beliebige mit [OnSerializingAttribute], [OnSerializedAttribute], [OnDeserializingAttribute] bzw. [OnDeserializedAttribute] annotierte Methoden	ISerializable	ISerializable
Steuerung der zu serialisierenden Elemente	[XmlIgnoreAttribute], XmlSerializable	[NonSerialized], [DataMember], [EnumMember], XmlSerializable, ISerializable	[NonSerialized], [DataMember], [EnumMember], XmlSerializable, ISerializable	Über [NonSerialized] oder ISerializable	Über [NonSerialized] oder ISerializable ▶

	XML	DataContract (alias Shared Contract)	NetDataContract (alias Shared Type)	Binär	SOAP
Steuerung der Serialisierungsdetails (z. B. Namen und Reihenfolge)	[XmlElement], [XmlAttribute], IXmlSerializable	[NonSerialized], [DataMember], [EnumMember], IXmlSerializable, ISerializable	[NonSerialized], [DataMember], [EnumMember], IXmlSerializable, ISerializable	Über ISerializable	Über ISerializable
Steuerung der Deserialisierung	IXmlSerializable	IXmlSerializable, ISerializable	IXmlSerializable, ISerializable	Über IDeserialization Callback	Über IDeserializa- tionCallback
Explizite Versionierungsunterstützung	Nein, nur IXmlSerializable	IExtensibleData- Object / Exten- sionData / Exten- sionDataObject	IExtensibleData- Object / Exten- sionData / Exten- sionDataObject	Nein (nur ISerializable)	Nein (nur ISerializable)
Serialisierung in Streams	Ja	Ja	Ja	Ja	Ja
Serialisierung in TextWriter und XmlWriter	Ja	Ja	Ja	Nein	Nein
Serialisierung in XmlDictionaryWriter	Nein	Ja	Ja	Nein	Nein
Werkzeuge	Xsd.exe	SvcUtil.exe	SvcUtil.exe	Keine	Keine

Tabelle 9.5 Vergleich zwischen den .NET-Serialisierern

Welche Mitglieder eines Objekts serialisiert werden, hängt ab von der Verwendung einer Serialisierungsannotation und dem Serialisierer. Die folgenden Abbildungen zeigen dies am Beispiel der nachfolgend wiedergegebenen Klasse TestKlasse. Die Klasse wird wahlweise gar nicht annotiert (*POCO*), mit [Serializable] annotiert oder mit [DataContract] und [DataMember] annotiert. Nur der letzte Fall ist hier abgedruckt.

```
[DataContract]
public class TestKlasse
{
    [DataMember]
    public string Art = "[DataContract]";
    [DataMember]
    public string PublicField = "x";
    [DataMember]
    public readonly string PublicReadOnlyField = "x";
    [DataMember]
    private string PrivateField = "x";
    [DataMember]
    private readonly string PrivateReadOnlyField = "x";
    [DataMember]
    public string PublicAutomaticProperty { get; set; }
    [DataMember]
    private string PrivateAutomaticProperty { get; set; }
    [DataMember]
    public string PublicProperty
    {
        get
```

```

{
    return PublicField;
}
set
{
    PublicField = value;
}
}
// [DataMember] geht nicht!
public string PublicGetterProperty
{
    get
    {
        return PublicField;
    }
}
}

```

Listing 9.42 Die Datenklasse für den Test der Serialisierer

XML

```

<?xml version="1.0" ?>
- <TestKlasse xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Art>POCO</Art>
  <PublicField>x</PublicField>
  <PublicProperty>x</PublicProperty>
</TestKlasse>

```

DCS

```

<TestKlasse z:Id="1"
  xmlns="http://schemas.datacontract.org/2004/07/VerschiedeneDemos_CS.FCL3"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <Art z:Id="2">POCO</Art>
  <PublicAutomaticProperty i:nil="true" />
  <PublicField z:Id="3">x</PublicField>
  <PublicProperty z:Ref="3" i:nil="true" />
</TestKlasse>

```

NDCS

```

- <TestKlasse z:Id="1" z:Type="VerschiedeneDemos_CS.FCL3.TestKlasse"
  z:Assembly="WWings.VerschiedeneDemos, Version=0.5.0.0, Culture=neutral,
  PublicKeyToken=null"
  xmlns="http://schemas.datacontract.org/2004/07/VerschiedeneDemos_CS.FCL3"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <Art z:Id="2">POCO</Art>
  <PublicAutomaticProperty i:nil="true" />
  <PublicField z:Id="3">x</PublicField>
  <PublicProperty z:Ref="3" i:nil="true" />
</TestKlasse>

```

Abbildung 9.15 Serialisierung der Datenklasse ohne Annotation (POCO-Fall)


```

<?xml version="1.0" ?>
- <TestKlasse xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Art>[Serializable]</Art>
  <PublicField>x</PublicField>
  <PublicProperty>x</PublicProperty>
</TestKlasse>

```

XML

```

- <TestKlasse z:Id="1"
  xmlns="http://schemas.datacontract.org/2004/07/VerschiedeneDemos_CS.FCL3"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <Art z:Id="2">[Serializable]</Art>
  <PrivateField z:Id="3">x</PrivateField>
  <PrivateReadOnlyField z:Ref="3" i:nil="true" />
  <PublicField z:Ref="3" i:nil="true" />
  <PublicReadOnlyField z:Ref="3" i:nil="true" />
  <x003C_PrivateAutomaticProperty_x003E_k__BackingField i:nil="true" />
  <x003C_PublicAutomaticProperty_x003E_k__BackingField i:nil="true" />
</TestKlasse>

```

DCS

```

- <TestKlasse z:Id="1" z:Type="VerschiedeneDemos_CS.FCL3.TestKlasse"
  z:Assembly="WWings.VerschiedeneDemos, Version=0.5.0.0, Culture=neutral,
  PublicKeyToken=null"
  xmlns="http://schemas.datacontract.org/2004/07/VerschiedeneDemos_CS.FCL3"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <Art z:Id="2">[Serializable]</Art>
  <PrivateField z:Id="3">x</PrivateField>
  <PrivateReadOnlyField z:Ref="3" i:nil="true" />
  <PublicField z:Ref="3" i:nil="true" />
  <PublicReadOnlyField z:Ref="3" i:nil="true" />
  <x003C_PrivateAutomaticProperty_x003E_k__BackingField i:nil="true" />
  <x003C_PublicAutomaticProperty_x003E_k__BackingField i:nil="true" />
</TestKlasse>

```

NDCS

Abbildung 9.16 Serialisierung der Datenklasse bei Annotation mit [Serializable]

```

<?xml version="1.0" ?>
- <TestKlasse xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Art>[DataContract]</Art>
  <PublicField>x</PublicField>
  <PublicProperty>x</PublicProperty>
</TestKlasse>

```

XML

```

- <TestKlasse z:Id="1"
  xmlns="http://schemas.datacontract.org/2004/07/VerschiedeneDemos_CS.FCL3"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <Art z:Id="2">[DataContract]</Art>
  <PrivateAutomaticProperty i:nil="true" />
  <PrivateField z:Id="3">x</PrivateField>
  <PrivateReadOnlyField z:Ref="3" i:nil="true" />
  <PublicAutomaticProperty i:nil="true" />
  <PublicField z:Ref="3" i:nil="true" />
  <PublicProperty z:Ref="3" i:nil="true" />
  <PublicReadOnlyField z:Ref="3" i:nil="true" />
</TestKlasse>

```

DCS

```

- <TestKlasse z:Id="1" z:Type="VerschiedeneDemos_CS.FCL3.TestKlasse"
  z:Assembly="WWings.VerschiedeneDemos, Version=0.5.0.0, Culture=neutral,
  PublicKeyToken=null"
  xmlns="http://schemas.datacontract.org/2004/07/VerschiedeneDemos_CS.FCL3"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <Art z:Id="2">[DataContract]</Art>
  <PrivateAutomaticProperty i:nil="true" />
  <PrivateField z:Id="3">x</PrivateField>
  <PrivateReadOnlyField z:Ref="3" i:nil="true" />
  <PublicAutomaticProperty i:nil="true" />
  <PublicField z:Ref="3" i:nil="true" />
  <PublicProperty z:Ref="3" i:nil="true" />
  <PublicReadOnlyField z:Ref="3" i:nil="true" />
</TestKlasse>

```

NDCS

Abbildung 9.17 Serialisierung der Datenklasse bei Annotation mit [DataContract] und [DataMember]

System.DirectoryServices

System.DirectoryServices enthält Klassen für den Zugriff auf Verzeichnisdienste. Angesprochen werden können insbesondere LDAP-basierte Verzeichnisdienste (z.B. Active Directory, Lotus Notes, Exchange Server, Open LDAP), aber auch einige nicht LDAP-basierte Verzeichnisdienste wie die Windows-Benutzerdatenbank SAM, die Metabase der Internet Information Services (IIS) und Novell Netware.

Die FCL-Komponente System.DirectoryServices ist eine Verpackung für die COM-Komponente *Active Directory Service Interface (ADSI)*. Über das Attribut NativeObject hat man jederzeit die Möglichkeit, direkt mit den zu Grunde liegenden COM-Schnittstellen zu arbeiten; dies ist in einigen Fällen empfehlenswert und in anderen Fällen sogar notwendig, weil dort spezielle Attribute und Methoden zur Verfügung stehen.

HINWEIS

Neu seit .NET 2.0 sind die Unternehmensräume System.DirectoryServices.ActiveDirectory (siehe unten) und System.DirectoryServices.Protocols (siehe unten). Neu seit .NET 3.5 ist der Unternehmensraum System.DirectoryService.AccountManagement (siehe unten).

Allgemeines Objektmodell

Die Klasse DirectoryEntry repräsentiert eine beliebige Art von Eintrag in einem beliebigen Verzeichnisdienst. DirectoryEntries sind Container (also Einträge, die andere Einträge enthalten können) in einem Verzeichnisdienst. Da Verzeichnisdienste mehrwertige Attribute besitzen können, enthält die PropertyCollection eines DirectoryEntry-Objekts nicht Werte, sondern eine Objektmenge von Werten.

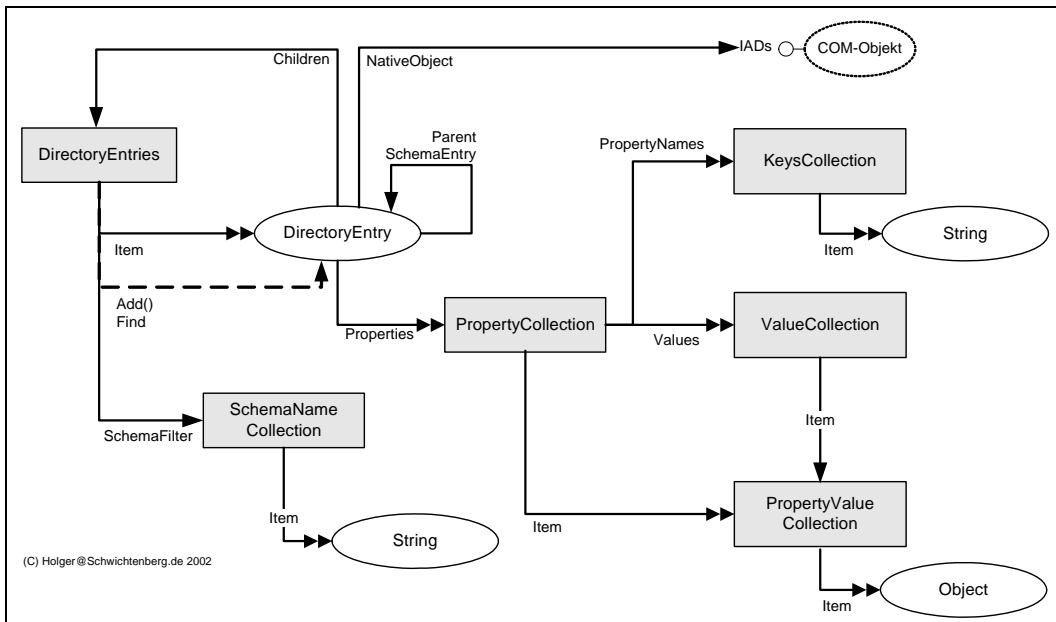


Abbildung 9.18 Objektmodell für das Durchlaufen eines Verzeichnisdienstes

TIPP

Während System.DirectoryService nur sehr allgemeine Klassen zur Behandlung jeglicher Einträge in einem Verzeichnisdienst bietet, existiert ab .NET 3.5 der Namensraum System.DirectoryService.AccountManagement mit speziellen Klassen zur Verwaltung von Benutzerkonten, Computerkonten und Gruppen.

Beispiel: Anlegen eines Benutzers in Active Directory

Beim Anlegen eines Benutzerkontos ist zunächst unter Verwendung des LDAP-Pfads ein DirectoryEntry-Objekt zu instanzieren für den Container, der den neuen Benutzer beheimaten soll. Mit dem Attribut Children ist dann das zugehörige DirectoryEntries-Objekt zu beschaffen, das die Methode Add() zur Verfügung stellt. Nach dem Füllen der Properties-Objektmenge sind die Änderungen mit CommitChanges() an den Verzeichnisdienst zu übergeben. Für das Aktivieren des Benutzerkontos, das Setzen des Kennworts und das Hinzufügen zu einer Gruppe ist der Rückgriff auf die COM-Schnittstelle IADsUser notwendig, da in System.DirectoryService entsprechende Möglichkeiten leider immer noch fehlen. Einfacher geht es mit der Bibliothek DirectoryService.AccountManagement (siehe unten).

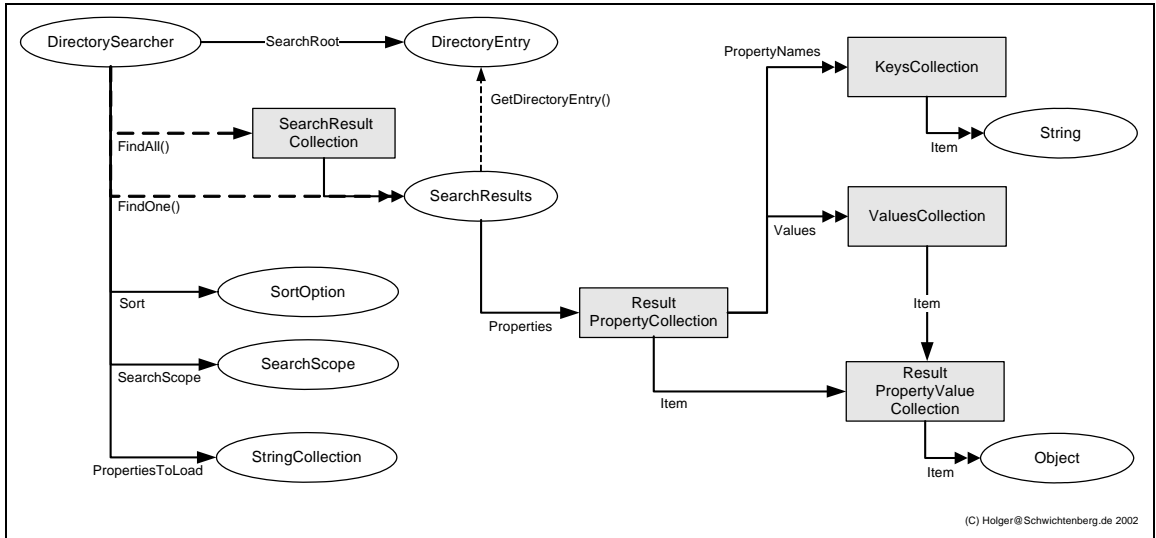
```
// Anlegen eines Benutzers in Active Directory
public void ADS_Benutzer_Anlegen()
{
    // AD-spezifische Parameter, bitte anpassen!
    const string LDAP_OU = "LDAP://Server121/OU=essen,DC=IT-Visions,DC=net";
    const string NAME = "hs";
    const string PASSWORD = "geheim";
    const string LDAP_Gruppe = "LDAP://Server121/CN=Domänen-Admins,CN=Users,DC=IT-Visions,DC=net";

    // Anlegen eines User-Objekts in Active Directory
    Demo.Print("# Anlegen des Benutzerkontos: " + NAME);
    // Zugriff auf IADS
    DirectoryEntry ou = new DirectoryEntry(LDAP_OU);
    // Zugriff auf IADSContainer
    DirectoryEntries c = ou.Children;
    // Neues Objekt erzeugen
    DirectoryEntry u = c.Add("cn=" + NAME, "user");
    // Verzeichnisattribute festlegen
    u.Properties["sAMAccountName"].Add(NAME);
    u.Properties["l"].Add("Essen-Byfang");
    u.Properties["telephoneNumber"].Add("++49 201 7490700");
    u.Properties["mail"].Add("hs@IT-Visions.de");
    // Änderungen speichern
    u.CommitChanges();
    Demo.Print("Benutzer angelegt: " + u.Path.ToString());
    // Kennwort setzen
    u.Invoke("SetPassword", PASSWORD);
    // Konto aktivieren
    ((ActiveDs.IADsUser)(u.NativeObject)).AccountDisabled = false;
    u.CommitChanges();
    Demo.Print("Benutzer aktiviert!");
    // AD-Benutzer einer AD-Gruppe hinzufügen
    // Zugriff auf Eintrag
    DirectoryEntry g = new DirectoryEntry(LDAP_Gruppe);
    // IADsGroup::Add() aufrufen
    g.Invoke("Add", u.Path.ToString());
    // Bestätigung
    Demo.Print("Benutzer zu Gruppe hinzugefügt!");
}
```

Listing 9.43 Anlegen eines Benutzers in Active Directory [ActiveDirectory.cs]

Objektmodell für die Suche

System.DirectoryService unterstützt die Suche nur in LDAP-basierten Verzeichnisdiensten und stellt dafür ein eigenes Objektmodell zur Verfügung (siehe Abbildung 9.19).



(C) Holger@Schwichtenberg.de 2002

Abbildung 9.19 Objektmodell für die Suchfunktion in System.DirectoryServices

Beispiel: Suchen in Active Directory

Die nachfolgende Unterroutine sucht nach allen Benutzern in Active Directory, deren Verzeichnisname mit »h« beginnt. Die Bedingung muss sich dabei zur Steigerung der Suchgeschwindigkeit sowohl auf die objectclass als auch auf die objectcategory beziehen. Durch PropertiesToLoad legt der Entwickler fest, welche Verzeichnisattribute in der Ergebnismenge enthalten sein sollen.

TIPP

Bitte beachten Sie, dass Sie einen Laufzeitfehler erhalten, wenn Sie versuchen, Verzeichnisattribute abzurufen, die nicht in allen Verzeichnisobjekten mit einem Wert belegt sind.

```

// Ausführen einer LDAP-Suche im AD
public void ADS_Suche()
{
    const string LDAP_Wurzel = "LDAP://SERVER01/DC=IT-Visions,DC=net";
    const string BEDINGUNG = "(&(objectclass=user)(objectcategory=person)(cn=h*))";
    Demo.Print("Suchanfrage im ADS: " + BEDINGUNG);
    // Instanziierung der Suchklasse
    DirectorySearcher suche = new DirectorySearcher();
    // Festlegung des Ausgangspunkts
    suche.SearchRoot = new DirectoryEntry(LDAP_Wurzel);
    // Festlegung der LDAP-Query
    suche.Filter = BEDINGUNG;
    // Suchtiefe festlegen
    suche.SearchScope = SearchScope.Subtree;
}
  
```

```

// Ergebnisattribute festlegen
suche.PropertiesToLoad.Add("displayName");
suche.PropertiesToLoad.Add("1");
suche.PropertiesToLoad.Add("description");
// Suche starten
SearchResultCollection ergebnisliste = suche.FindAll();
// Ergebnismenge ausgeben
Demo.Print("Anzahl Ergebnisobjekte: " + ergebnisliste.Count);
foreach (SearchResult ergebnis in ergebnisliste)
{
    foreach (System.Collections.DictionaryEntry de in ergebnis.Properties)
    {
        Demo.Print(de.Key.ToString() + "=" + ((ResultPropertyValueCollection) de.Value)[0].ToString());
    }
}
// Ergebnis lesen
Demo.Print(ergebnis.Properties["displayName"].Count.ToString());
Demo.Print(ergebnis.Properties["displayName"][0].ToString());
Demo.Print(" wohnt in " + ergebnis.Properties["1"][0].ToString());
}
}

```

Listing 9.44 Suchen in Active Directory [ActiveDirectory.cs]

System.DirectoryServices.ActiveDirectory

Neu seit .NET 2.0 ist der Unternamensraum `System.DirectoryServices.ActiveDirectory` (alias Active Directory Management Objects, ADMO). Dieser Namensraum implementiert einige Active Directory-spezifische Funktionen, die nicht auf andere Verzeichnisdienste anwendbar sind.

Insbesondere bietet dieser Namensraum Klassen zur Verwaltung der Gesamtstruktur von Active Directory, beispielsweise `Forest`, `Domain`, `ActiveDirectoryPartition`, `DomainController`, `GlobalCatalog` und `ActiveDirectorySubnet`. Auch einige spezielle Klassen für den »Active Directory Lightweight Directory Services (AD LDS)« (früher: Active Directory Application Mode (ADAM)), eine funktionsreduzierte Version von Active Directory zum Einsatz als Datenspeicher für eigene Anwendungen, werden mit Klassen wie `ADAMInstanceCollection` und `ADAMInstance` unterstützt.

Beispiel 1: Informationen über die Domäne und den Forest

Das Beispiel liefert Informationen über die Domäne, zu der der aktuelle Computer gehört, und über den Verzeichnisswald (Forest), zu dem diese Domäne gehört.

```

public void DomaenenInfos()
{
    // Aktuelle Domäne ermitteln
    Domain d = System.DirectoryServices.ActiveDirectory.Domain.GetCurrentDomain();
    // Informationen über aktuelle Domäne
    Console.WriteLine("Name: " + d.Name);
    Console.WriteLine("Domain Mode: " + d.DomainMode);
    Console.WriteLine("Inhaber der InfrastructureRole: " + d.InfrastructureRoleOwner.Name);
    Console.WriteLine("Inhaber der PdcRole: " + d.PdcRoleOwner.Name);
}

```

```
// Informationen über Forest der aktuellen Domäne
Forest f = d.Forest;
Console.WriteLine("Name des Forest: " + f.Name);
Console.WriteLine("Modus des Forest: " + f.ForestMode);
}
```

Listing 9.45 Informationen über eine Active Directory-Domäne [ActiveDirectory.cs]

Beispiel 2: Liste der Domänen-Controller und ihrer Rollen

Im zweiten Beispiel werden alle Domänen-Controller (und deren Rollen) aus einer speziellen Domäne aufgelistet.

```
// Alle DC ausgeben
public void DomaenenController()
{
    // Aktuelle Domäne ermitteln
    DirectoryContext dc = DirectoryContext.Open("LDAP://dc=IT-Visions,dc=net");
    Domain d = System.DirectoryServices.ActiveDirectory.Domain.GetObject(dc);
    DomainControllerCollection DCs = d.DomainControllers;
    // Schleife über alle Domänen-Controller
    foreach (DomainController DC in DCs)
    {
        Console.WriteLine("Name: " + DC.Name);
        Console.WriteLine("IP: " + DC.IPAddress.ToString());
        Console.WriteLine("Zeit: " + DC.CurrentTime.ToString());
        Console.WriteLine("Rollen:");
        // Schleife über alle Rollen des Domänen-Controllers
        foreach (ActiveDirectoryRole R in DC.Roles)
        {
            Console.WriteLine("- " + R.ToString());
        }
    }
}
```

Listing 9.46 Liste der Domänen-Controller und ihrer Rollen [ActiveDirectory.cs]

System.DirectoryServices.Protocol

Eine weitere Neuheit im .NET Framework 2.0 war die Unterstützung für die Directory Services Markup Language (DSML). DSML ist ein Standard der OASIS zum Zugriff auf Verzeichnisdienste via XML und SOAP. Dieser Namensraum kann hier aus Platzgründen leider nicht thematisiert werden.

System.DirectoryService.AccountManagement

Während System.DirectoryService nur sehr allgemeine Klassen zur Behandlung jeglicher Einträge in einem Verzeichnisdienst bietet, existiert seit .NET 3.5 der Namensraum System.DirectoryService.AccountManagement mit speziellen Klassen zur Verwaltung von Benutzerkonten (Klasse UserPrincipal), Computerkonten (Klasse ComputerPrincipal) und Gruppen (Klasse GroupPrincipal). Die neuen Klassen sind enthalten in der Assembly *System.DirectoryServices.AccountManagement.dll*.

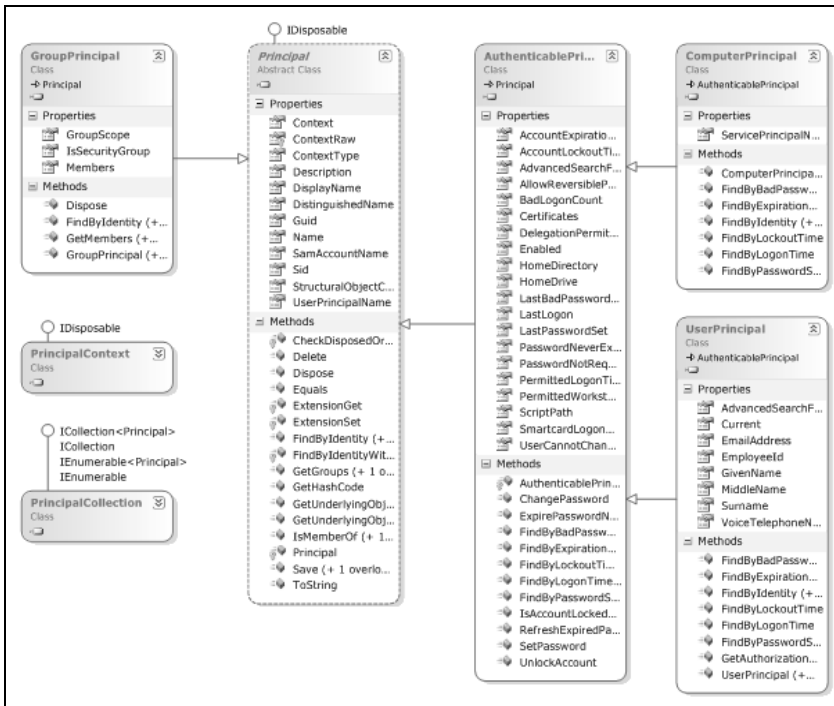


Abbildung 9.20 Klassendiagramm der wichtigsten Klassen aus System.DirectoryService.AccountManagement

Klasse PrincipalContext

PrincipalContext dient dazu, einen Verzeichnisbaum auszuwählen. Es gibt drei Arten von Verzeichnisbäumen, die in ContextType unterstützt werden:

- **Domain** Active Directory (AD)
- **ApplicationDirectory** Active Directory Lightweight Directory Services (AD LDS), früher Active Directory Application Mode (ADAM)
- **Machine** Lokale Benutzerkonten in der Windows Security Account Manager (SAM)-Datenbank

In dem nachfolgenden Beispiel wird eine bestimmte Organisationseinheit in Active Directory über einen bestimmten Domänencontroller angesprochen:

```
PrincipalContext ctx = new PrincipalContext(ContextType.Domain, "Server02",
                                           "OU=Training,DC=IT-Visions,DC=local");
```

TIPP Eine hilfreiche Methode ist `ValidateCredentials()` mit der man eine Kombination aus Benutzernamen und Kennwort prüfen kann.

```
PrincipalContext ctx =
    new PrincipalContext(ContextType.Domain, null, "DC=IT-Visions,DC=local");
bool b = ctx.ValidateCredentials("HS", "Sehr+Geheim");
```

Klasse UserPrincipal

Die Klasse `UserPrincipal` repräsentiert einen Benutzer und eine Reihe von dessen Eigenschaften. Außerdem stellt diese Klasse statische Methoden bereit, um Benutzer und Benutzerlisten zu erhalten:

- `UserPrincipal.Current` ist der angemeldete Benutzer
- `UserPrincipal.FindByIdentity()` dient dazu, einen bestimmten Benutzer zu finden. Dabei gibt es verschiedene Möglichkeiten: z.B. anhand von Verzeichnisnamen (`IdentityType.DistinguishedName`), Anmeldenamen (`IdentityType.SamAccountName`), Security Identifier (`IdentityType.Sid`), Verzeichnis-GUID (`IdentityType.Guid`).
- Mit den Methoden `FindByBadPasswordAttempt()`, `FindByPasswordSetTime()` und `FindByLogonTime()` findet man alle Benutzer, auf die das sich aus dem Methodennamen ergebende Kriterium zutrifft (siehe auch Beispiel unten)

Beispiel: Benutzerdaten lesen

Das Beispiel zeigt das Auslesen von Benutzerdaten.

```
public static void BenutzerSuchen()
{
    // Stamm für alle weiteren Operationen festlegen
    PrincipalContext ctx = new PrincipalContext(ContextType.Domain, "Server02", "DC=IT-Visions,DC=local");

    Console.WriteLine("---- Aktueller Benutzer: ");

    Console.WriteLine("Namen:");
    Console.WriteLine(UserPrincipal.Current.Name);
    Console.WriteLine(UserPrincipal.Current.SamAccountName);
    Console.WriteLine(UserPrincipal.Current.DistinguishedName);
    Console.WriteLine(UserPrincipal.Current.DisplayName);
    Console.WriteLine(UserPrincipal.Current.EmployeeId);
    Console.WriteLine(UserPrincipal.Current.GivenName);
    Console.WriteLine(UserPrincipal.Current.Surname);
    Console.WriteLine(UserPrincipal.Current.Description);
    Console.WriteLine(UserPrincipal.Current.Sid);

    Console.WriteLine("Adressen:");
    Console.WriteLine(UserPrincipal.Current.EmailAddress);
    Console.WriteLine(UserPrincipal.Current.VoiceTelephoneNumber);
    Console.WriteLine("Konfiguration:");
    Console.WriteLine(UserPrincipal.Current.HomeDirectory);
    Console.WriteLine(UserPrincipal.Current.HomeDrive);
    Console.WriteLine(UserPrincipal.Current.Enabled);

    Console.WriteLine("Anmeldungen:");
    Console.WriteLine(UserPrincipal.Current.LastLogon);
    Console.WriteLine(UserPrincipal.Current.BadLogonCount);
    Console.WriteLine(UserPrincipal.Current.LastPasswordSet);
    Console.WriteLine(UserPrincipal.Current.PasswordNeverExpires);
    Console.WriteLine(UserPrincipal.Current.UserCannotChangePassword);
}
```



```

Console.WriteLine("---- Bestimmter Benutzer: ");
UserPrincipal u = UserPrincipal.FindByIdentity(ctx, IdentityType.SamAccountName, "hs");
Console.WriteLine(u.Name);

Console.WriteLine(
    "Alle Benutzer, die in den letzten 30 Tagen ihr Kennwort falsch eingegeben haben:");
foreach (var p in UserPrincipal.FindByBadPasswordAttempt(ctx,
    DateTime.Now.Subtract(TimeSpan.FromDays(30)), MatchType.GreaterThanOrEquals))
{
    Console.WriteLine(p.Name + ": " + p.LastBadPasswordAttempt);
    // p.ExpirePasswordNow();
}

Console.WriteLine("Alle Benutzer, die in den letzten 30 Tagen ihr Kennwort geändert haben:");
foreach (var p in UserPrincipal.FindByPasswordSetTime(ctx,
    DateTime.Now.Subtract(TimeSpan.FromDays(30)), MatchType.GreaterThanOrEquals))
{
    Console.WriteLine(p.Name + ": " + p.LastPasswordSet);
}

Console.WriteLine("Alle Benutzer, die sich in den letzten 24 Stunden angemeldet haben:");
foreach (var p in UserPrincipal.FindByLogonTime(ctx, DateTime.Now.Subtract(TimeSpan.FromHours(24)),
    MatchType.GreaterThanOrEquals))
{
    Console.WriteLine(p.Name + ": " + p.LastLogon);
}
}

```

Listing 9.47 Benutzersuche und Ausgabe von Informationen über Benutzer

HINWEIS Anders als bei der Arbeit mit der Klasse `DirectoryEntry` werden hier leere Verzeichnisattribute als leere Zeichenketten oder *null*-Werte (bei Datums-/Zeitangaben) signalisiert.

Beispiel: Benutzerdaten verändern

Viele Attribute der Klasse `UserPrincipal` sind beschreibbar. Änderungen können durch Aufruf von `Save()` gespeichert werden.

```

public static void BenutzerAendern()
{
    // Stamm für alle weiteren Operationen festlegen
    PrincipalContext ctx = new PrincipalContext(ContextType.Domain, "Server02", "DC=IT-Visions,DC=local");

    Console.WriteLine("--- Benutzer holen: ");
    UserPrincipal u = UserPrincipal.FindByIdentity(ctx, IdentityType.SamAccountName, "hs");
    Console.WriteLine(u.Name);
    Console.WriteLine(u.EmailAddress);

    Console.WriteLine("--- Benutzer verändern: ");
    u.EmailAddress = "hs@IT-Visions.de";

    Console.WriteLine("--- Änderungen speichern: ");
    u.Save();
}

```

Listing 9.48 Änderungen von Benutzerdaten

TIPP

Zum Neuvergeben des Kennworts kann man `SetPassword()` aufrufen. Zum Ändern eines Kennworts unter Angabe des alten Kennworts kann man `ChangePassword()` aufrufen. Zum Entsperren eines Kontos kann man `UnlockAccount()` aufrufen. Zum Löschen eines Benutzers kann man `Delete()` aufrufen. Über die Methode `GetUnderlyingObject()` kann man ein `DirectoryEntry`-Objekt erhalten, um weitere oder benutzerdefinierte Verzeichnisattribute auf »klassische« Weise (d. h. im Stil von .NET 1.x/2.0) abzurufen.

Beispiel: Benutzer anlegen

Auch das Anlegen eines Benutzerkontos ist einfacher als bei der Verwendung der Klasse `DirectoryEntry`.

```
public static void BenutzerAnlegen()
{
    // Stamm für alle weiteren Operationen festlegen
    PrincipalContext ctx = new PrincipalContext(ContextType.Domain, "Server02", "cn=Users,DC=IT-
Visions,DC=local");
    // Benutzerobjekt erzeugen
    UserPrincipal b = new UserPrincipal(ctx);
    // Daten setzen
    b.Surname = "Schwichtenberg";
    b.GivenName = "Felix";
    b.DisplayName = "Felix Schwichtenberg";
    b.SamAccountName = "FS";
    b.SetPassword("Sehr+Geheim");
    b.Enabled = true;
    // Speichern
    b.Save();
}
```

Listing 9.49 Anlegen eines neuen Benutzers in Active Directory

Klasse GroupPrincipal

Die Klasse `GroupPrincipal` repräsentiert eine Benutzergruppe. Einzige statische Methode ist `FindByIdentity()` zum Auffinden einer Gruppe. Über das Instanzmitglied `Members` kann man Benutzer bzw. andere Gruppen hinzufügen (`Add()`) oder entfernen (`Remove()`).

```
public static void BenutzerZuGruppe()
{
    PrincipalContext ctx = new PrincipalContext(ContextType.Domain, "Server02", "DC=IT-Visions,DC=local");
    Console.WriteLine("--- Benutzer holen");
    UserPrincipal b = UserPrincipal.FindByIdentity(ctx, IdentityType.SamAccountName, "FS");
    Console.WriteLine("--- Gruppe holen");
    GroupPrincipal group = GroupPrincipal.FindByIdentity(ctx, "Domain Admins");
    Console.WriteLine("--- Benutzer zuordnen");
    group.Members.Add(b);
    Console.WriteLine("--- Gruppe speichern");
    group.Save();
}
```

Listing 9.50 Hinzufügen eines Benutzers zu einer Gruppe in Active Directory

System.Management

Der Namensraum System.Management bietet Zugriff auf die Windows Management Instrumentation (WMI). WMI ist ein objektorientiertes Rahmenwerk für das System- und Netzwerkmanagement; die aktuellen Windows-Versionen bieten jeweils mehrere tausend Klassen. System.Management realisiert mit wenigen Klassen ein Meta-Objektmodell für beliebige WMI-Klassen. System.Management.ManagementObject repräsentiert eine Instanz eines WMI-Objekts und System.Management.ManagementObjectCollection eine Menge von WMI-Objekten.

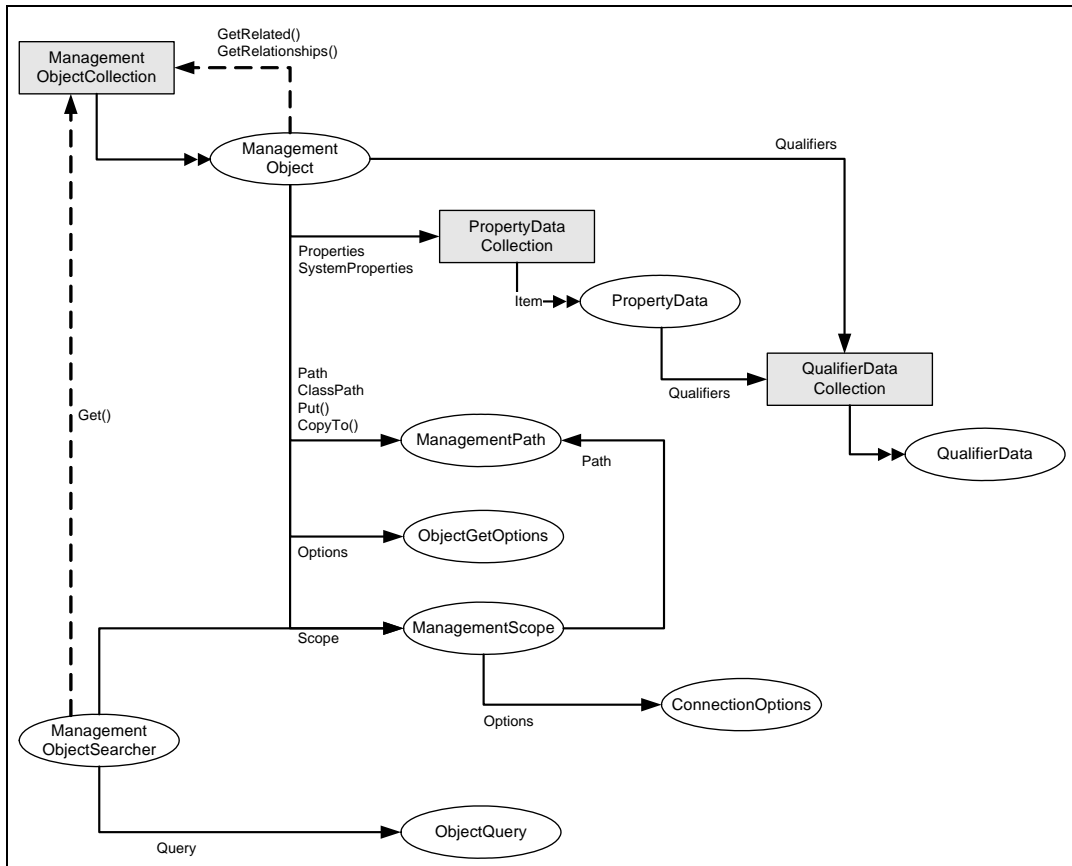


Abbildung 9.21 Objektmodell in System.Management

Beispiel 1: Daten über ein einzelnes WMI-Objekt

Das erste Beispiel demonstriert den Zugriff auf die Instanz *C* der WMI-Klasse *Win32_LogicalDisk*.

```
// Auslesen von Daten über Laufwerk C
public void WMI_Laufwerk_Auslesen()
{
    // WMI-Pfad
    const string pfad = @"\\root\\cimv2:Win32_LogicalDisk.DeviceID='C:'";
}
```

```
// Zugriff auf Managed Object
ManagementObject mo = new ManagementObject(pfad);
// Attribute des Meta-Objekts
Demo.PrintSubHeader("Metainformation:");
Demo.Print("kompletter WMI-Pfad: " + mo.Path.Path);
Demo.Print("Relativer Pfad: " + mo.Path.RelativePath);
Demo.Print("Computer: " + mo.Path.Server);
Demo.Print("Namensraum: " + mo.Path.NamespacePath);
Demo.Print("Standardpfad: " + ManagementPath.DefaultPath.Path);
Demo.Print("Klassenname: " + mo.Path.ClassName);

// Attribute des Managed Object
Demo.PrintSubHeader("Zugriff auf einzelne Attribute");
Demo.Print("Name: " + mo["Caption"]);
Demo.Print("Dateisystem: " + mo["FileSystem"]);
Demo.Print("Freie Bytes: " + Convert.ToInt64(mo["FreeSpace"]).ToString());
}
```

Listing 9.51 Daten über ein Laufwerk auslesen [WMI.cs]

Beispiel 2: Ausführen einer WMI-Methode

Im zweiten Beispiel erfolgt der Aufruf der Methode `Reboot()` in einer Instanz der Klasse `Win32_OperatingSystem`. Im WMI-Repository existiert nur eine einzige Instanz dieser Klasse; da deren Schlüsselattributwert aber nicht bekannt ist, kann die Instanz nicht direkt aufgerufen werden. Die Klasse `System.Management.ManagementClass` repräsentiert eine beliebige WMI-Klasse und stellt mit `GetInstances()` eine Möglichkeit bereit, alle Instanzen der WMI-Klasse zu erhalten.

```
// Informationen über ein Betriebssystem und Neustart
public void WMI_Computer_Reboot()
{
    const string COMPUTER = "Server02";
    // Zugriff auf Klasse
    ManagementClass mc = new ManagementClass(@"\\" + COMPUTER + @"\root\CIMV2:Win32_OperatingSystem");
    // Instanzen holen
    ManagementObjectCollection menge = mc.GetInstances();
    // Es gibt zwar nur ein Objekt, aber man kann nicht direkt darauf zugreifen
    foreach (ManagementObject mo in menge)
    {
        Demo.Print("Informationen über das Betriebssystem auf: " + mo.Path.Server);
        Demo.Print("Name: " + mo["Name"].ToString());
        Demo.Print("Hersteller: " + mo["Manufacturer"].ToString());
        Demo.Print("Typ: " + mo["OSType"].ToString());
        Demo.Print("Sprache: " + mo["OSLanguage"].ToString());
        Demo.Print("Version: " + mo["Version"].ToString());
        Demo.Print("Systemverzeichnis: " + mo["SystemDirectory"].ToString());
        Demo.Print("Registrierter Benutzer: " + mo["RegisteredUser"].ToString());
        Demo.PrintSubHeader("Reboot wird initiiert...");
        mo.InvokeMethod("reboot", null);
        Demo.PrintSubHeader("Reboot eingeleitet!");
    }
}
```

Listing 9.52 Neustart eines Rechners [WMI.cs]

Beispiel 3: Ausführen einer WMI-Abfrage

.NET unterstützt auch Abfragen in der WMI Query Language (WQL), die SQL sehr ähnlich ist. Die folgende Unterroutine führt eine WQL-Abfrage aus, die alle laufenden Systemdienste ermittelt. Ein Systemdienst wird in WMI durch die Klasse `Win32_Service` repräsentiert.

```
// WQL-Datenabfrage: Liste aller laufenden Dienste
public void WMI_Dienste_Auflisten()
{
    // Abfragebefehl (WQL)
    const string ABFRAGE = "select name,state from Win32_Service where state='running'";
    const string COMPUTER = "Server02";
    Demo.PrintSubHeader("WQL-Datenabfrage: Liste aller laufenden Dienste");
    ManagementScope scope = new ManagementScope("\\\\" + COMPUTER);
    // Abfrage erzeugen
    SelectQuery sq = new SelectQuery(ABFRAGE);
    ManagementObjectSearcher suche = new ManagementObjectSearcher(scope, sq);
    // Abfrage ausführen
    ManagementObjectCollection menge = suche.Get();
    // Ergebnisse ausgeben
    foreach (ManagementObject mo in menge)
        Demo.Print("Dienst: " + mo["Name"].ToString() + " Zustand: " + mo["state"].ToString());
}
```

Listing 9.53 Liste aller laufenden Dienste [WMI.cs]

System.Resources

Texte, Grafiken, Videos und andere Elemente, die leicht austauschbar sein müssen (z.B. für die Lokalisierung von Anwendungen auf andere Sprachen) sollten nicht in eine Anwendung hineinkompiliert werden, sondern in getrennten Ressourcendateien vorliegen. Das .NET Framework besitzt eine Infrastruktur für die Verwaltung von Ressourcendateien in verschiedenen Sprachen. Ressourcen können in *.resource*-Dateien oder kompilierten Assemblys (so genannten *Satelliten-Assemblys*) vorliegen. In beiden Fällen wird pro Sprache eine Datei verwendet, die jeweils den Namen der zugehörigen Assembly trägt mit dem Zusatz *resources*. Die Trennung der verschiedenen Sprachressourcen erfolgt entweder durch einen Sprachzusatz im Dateinamen oder durch entsprechende Unterverzeichnisse. Dabei kommen die Sprachkürzel nach RFC 1766 zum Einsatz, beispielsweise *de-DE* (für Deutsch in Deutschland), *de-AT* (für österreichisches Deutsch), *en-GB* (für britisches Englisch) und *en-US* (für amerikanisches Englisch). Eine allgemeine (invariante) Sprachressource ohne Länderkürzel kann definiert werden für alle nicht explizit behandelten Sprachen.

```
E:\N2C\WORLDWIDEWINGS\CONSOLEUI_CS\BIN\DEBUG
ConsoleUI_CS.exe
ConsoleUI_CS.exe.config
ITV_DemoViewer.dll

--de-AT
  ConsoleUI_CS.resources.dll

--de-DE
  ConsoleUI_CS.resources.dll

--en
  ConsoleUI_CS.resources.dll

--en-US
  ConsoleUI_CS.resources.dll
```

Abbildung 9.22 Beispiele für eine Ordnerstruktur

ACHTUNG Es ist möglich, Grafiken und andere Dateien direkt als Ressourcen in eine Assembly einzubetten oder mit der Assembly zu verlinken. Diese direkt eingebetteten Ressourcen unterstützen aber nicht die Lokalisierung. Für die Lokalisierung müssen Sie die Grafiken etc. in eine *.resx*-Datei integrieren.

Erstellung von Ressourcendateien

Visual Studio unterstützt die Erstellung von Ressourcendateien mit Name-Wert-Paaren im XML-Format (*.resx*). Der Editor für *.resx*-Dateien unterstützt seit Visual Studio 2005 auch die Ablage von Grafikdateien innerhalb des XML-Dokuments. Bei der Kompilierung werden *.resx*-Dateien automatisch in *.resource*-Dateien umgewandelt, die der Sprach-Compiler verarbeiten können.

Bei der Verwendung der Kommandozeilen-Compiler müssen Ressourcen entweder direkt in *.resource*-Dateien vorliegen oder aber zunächst mit dem Werkzeug *resgen.exe* (aus dem .NET SDK) umgewandelt werden. *Resgen.exe* erlaubt als Eingabeformate *.resx* und *.txt*.

```
<data name="ConfirmationBody">
  <value xml:space="preserve">We like to confirm the following flights:</value>
</data>
<data name="ConfirmationSubject">
  <value xml:space="preserve">Booking Confirmation</value>
  <comment xml:space="preserve">Betreffzeile</comment>
</data>
<data name="Goodbye">
  <value xml:space="preserve">Kind regards</value>
</data>
<data name="Hello">
  <value xml:space="preserve">Dear</value>
</data>
<data name="ThankYou">
  <value xml:space="preserve">Thank you for flying with us.</value>
</data>
```

Abbildung 9.23 Beispiel für eine *.resx*-Datei (Texte für den Versand von Buchungsbestätigungen)

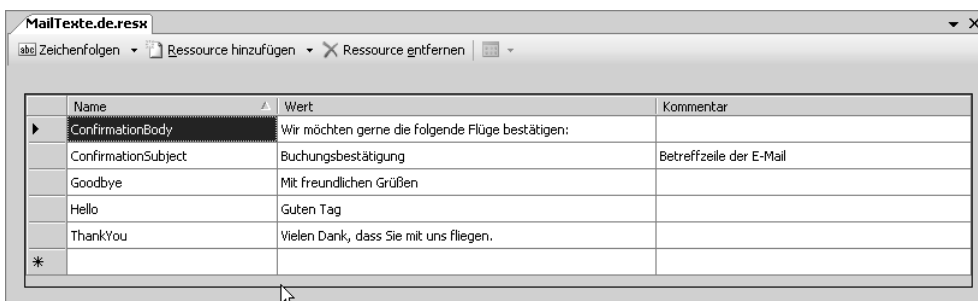


Abbildung 9.24 Ressourcen-Editor in Visual Studio

Zugriff auf Ressourcendateien

Der Zugriff auf Ressourcen ist sehr einfach über die Klasse `System.Resources.ResourceManager`: Diese benötigt beim Instanzieren den Namen der Ressource und ein `Assembly`-Objekt. Das `Assembly`-Objekt für die Assembly, in der sich der ablaufende Code befindet, erhält man über `System.Reflection.Assembly.GetExecutingAssembly()`. Der

Name der Ressource, den Visual Studio vergibt, besteht aus dem Namen der Assembly und dem Namen der .resx-Datei ohne Länderkürzel und Dateinamenserweiterung. Wenn die .resx-Datei in einem Unterverzeichnis des Projekts liegt, fügt Visual Studio dem Ressourcennamen den Unterverzeichnisnamen hinzu. Die Ressourcendateien *MailTexte.resx* im Unterverzeichnis *Ressourcen* des Projekts *WWWings_ConsoleUI_CS* haben also den Namen *WWWings_ConsoleUI_CS.Ressourcen.MailTexte*.

Die *ResourceManager*-Klasse stellt zur Beschaffung von Ressourcenelementen die Methoden *GetString()*, *GetObject()* und *GetStream()* bereit. Anzugeben ist der Name des Ressourcenelements unter Beachtung der Groß-/Kleinschreibung. Die Relevanz der Groß-/Kleinschreibung kann aber deaktiviert werden.

```
ResourceManager rm = new ResourceManager("WWWings.ConsoleUI_CS.Ressourcen.MailTexte",
System.Reflection.Assembly.GetExecutingAssembly());
rm.IgnoreCase = true;
MailBody =
rm.GetString("Hello") + "\n" +
rm.GetString("ConfirmationBody") + "\n" +
FlightList + "\n" +
rm.GetString("ThankYou") + "\n" +
rm.GetString("Goodbye") + "\n";
rm.ReleaseAllResources();
```

Listing 9.54 Auslesen von Ressourcen mit der Klasse *ResourceManager* [ResourceDemo.cs]

Das *ResourceManager*-Objekt verwendet als Sprache die Spracheinstellungen des aktuellen Threads (*System.Threading.Thread.CurrentThread.CurrentUICulture*). Zunächst sucht das *ResourceManager*-Objekt eine passende Ressourcendatei (z.B. *MailTexte.de-DE*). Wenn diese nicht gefunden wird, wird in einer Rückfallstrategie nach der übergeordneten Sprache (hier: *MailTexte.de*) gesucht. Wird auch diese nicht gefunden, verwendet der *ResourceManager* die invariante Ressourcendatei. Die Rückfallstrategie wird bei jedem einzelnen Ressourcenelement angewendet. So müssen in den Ressourcendateien *MailTexte.de-AT* und *MailTexte.de-DE* nur solche Ressourcenelemente definiert werden, bei denen die beiden Sprachen abweichen. Alle gleichen Elemente können in *MailTexte.de* definiert werden.

TIPP Die Spracheinstellung des aktuellen Threads richtet sich beim Programmstart nach den Spracheinstellungen von Windows. Sie können die Sprache während des Programmlaufs beliebig oft wechseln mit:

```
Thread.CurrentThread.CurrentUICulture = new System.Globalization.CultureInfo("en-GB");
```

Die aktuelle Sprache ermitteln Sie mit

```
System.Threading.Thread.CurrentThread.CurrentUICulture.Name;
```

oder

```
System.Threading.Thread.CurrentThread.CurrentUICulture.NativeName;
```

Name liefert immer den englischen Sprachnamen, *NativeName* den Namen der Sprache in der aktuellen Sprache.

Streng typisierte Ressourcen

Die Nutzung des `ResourceManager`-Objekts ist zwar recht einfach, aber fehleranfällig, weil die Namen der Ressourcenelemente als Zeichenketten angegeben werden. Jeder Tippfehler führt somit dazu, dass das Ressourcenelement nicht gefunden wird. Verfügbar seit .NET Framework 2.0 ist die Unterstützung für streng typisierte Ressourcen (*Strongly-Typed Resources*). Das Werkzeug *resgen.exe* bietet dem Entwickler seit .NET 2.0 die neue Option, eine Verpackungsklasse für eine beliebige Ressourcendatei zu generieren, sodass er mit frühem Binden auf die Ressourcennamen zugreifen und somit Laufzeitfehler wegen falscher Ressourcennamen vermeiden kann.

Die Verpackungsklasse kapselt den Zugriff auf das `ResourceManager`-Objekt und bietet ein statisches, nur lesbares Property-Mitglied für jedes Ressourcenelement. Der Code für den Zugriff auf die Ressource *Mail-Texte* wird somit kürzer und weniger fehleranfällig:

```
// streng typisiert (verwendet die generierte MailTexte.cs)
MailBody =
    WWWIngs_ConsoleUI_CS.MailTexte.Hello + "\n" +
    WWWIngs_ConsoleUI_CS.MailTexte.ConfirmationBoby + "\n" +
    FlightList + "\n" +
    WWWIngs_ConsoleUI_CS.MailTexte.ThankYou + "\n" +
    WWWIngs_ConsoleUI_CS.MailTexte.Goodbye + "\n";
```

Listing 9.55 Auslesen von Ressourcen mit der Klasse `ResourceManager` [RessourceDemo.cs]

Die Erstellung der Verpackungsklasse erfolgt mit *resgen.exe* über den Kommandozeilenparameter `/s`, bei dem die Zielsprache und der Namensraum für die generierte Klasse anzugeben sind.

```
Resgen /str:C#,de.itvisions
```

Alternativ dazu kann die Generierung auch über die Klasse `System.Resources.Tools.StronglyTypedResourceBuilder` angestoßen werden.

Visual Studio stellt für die in einigen Projekttypen automatisch angelegte *resources.resx* automatisch Verpackungsklassen bereit. In Visual Basic-Projekten erfolgt der Zugriff über `My.Resources`, in C# über `Anwendungsname.Properties.Resources`.

Managed Extensibility Framework (MEF)

Gastautor dieses Abschnitts: Manfred Steyer, entnommen aus »NET 4.0 Update« [HS07]

Der Ruf nach Standardapplikationen wird in der Regel von der Forderung nach der Möglichkeit, diese an die eigenen Bedürfnisse anzupassen, begleitet. Das Managed Extensibility Framework (MEF) geht auf diese Forderung ein, indem es die Implementierung von Plug-In-Mechanismen erleichtert. Dieser Abschnitt beschreibt die Anwendung dieses Frameworks, welches seit Version 4.0 im Namensraum `System.ComponentModel.Composition` Teil des .NET Framework ist.

HINWEIS Es gab schon frühere Ansätze für Erweiterbarkeit in .NET:

- Visual Studio for Applications (VSA) alias Script for .NET (Namensraum `Microsoft.Vsa`) – seit 1.1 und in .NET 2.0 schon wieder als *obsolete* gekennzeichnet
- Managed Add-In Framework (MAF) (Namensraum `System.AddIn`) – seit .NET 3.5. Diese Bibliothek ist weiterhin vorhanden, das in .NET 4.0 eingeführte MEF ist aber leichtgewichtiger und einfacher.

MAF bietet gegenüber MEF allerdings den Vorteil, dass man in MAF die Erweiterungen besser von der Hauptanwendung isolieren kann. MAF-Erweiterungen können dafür in eigenen Application Domains laufen.

Überblick

Bevor die Möglichkeiten von MEF im Detail besprochen werden, gibt dieser Abschnitt einen Überblick über die Architektur und bietet zur Veranschaulichung dieser ein einfaches einführendes Beispiel.

Architektur

Das Entwickeln von Applikationen, welche durch Plug-Ins erweitert werden können, ist zwar seit den ersten Tagen von .NET möglich, allerdings musste die dazu nötige Infrastruktur selbst erstellt werden. Das Managed Extensibility Framework (MEF), welches Teil von .NET 4.0 ist, entlastet von dieser Tätigkeit, indem es eine Infrastruktur für das Bereitstellen, Auffinden und Konsumieren von Plugins bietet.

Die Architektur von MEF sieht dazu die Möglichkeit vor, so genannte Parts zu definieren. Dabei handelt es sich um Programmteile, welche bestimmte Funktionalitäten anderen Parts bereitstellen oder bestimmte Funktionalitäten von anderen Parts benötigen. Im ersten Fall ist von *Exports* die Rede; im zweiten von *Imports*. Dabei ist es nicht notwendig, dass die einzelnen Parts einander kennen, denn das Verdrahten der einzelnen Imports mit entsprechenden Exports wird von einem *Composition Container* übernommen. Dieser verwendet einen Katalog, um sich über die zur Verfügung stehenden Parts zu informieren. Der Katalog kann dazu beispielsweise eine genannte Assembly oder alle Assemblys eines Verzeichnisses nach Parts durchsuchen. Alternativ dazu können dem Katalog einzelne Parts auch direkt bekannt gemacht werden. Um sicherzustellen, dass Imports nur mit geeigneten Exports verdrahtet werden, wird für beide ein Vertrag definiert, welcher aus der Angabe eines Typs und/oder eines Strings besteht. Nur wenn beide Seiten denselben Vertrag aufweisen, findet eine Verdrahtung statt.

Erste Schritte mit MEF

Zur Veranschaulichung der Möglichkeiten von MEF verwendet dieses Kapitel eine Implementierung eines erweiterbaren Logging-Framework. Dieses stellt eine Klasse *Logger* zur Verfügung, welche sich zum Formatieren der zu protokollierenden Nachrichten auf eine Implementierung der Schnittstelle *IFormatter* abstützt. Listing 9.56 zeigt diese Schnittstelle inklusive des von ihm verwendeten Enums *LogLevel*. Listing 9.57 zeigt eine Implementierung von *IFormatter*. Durch die Verwendung des Attributs *Export* wird diese Implementierung zum Part erhoben und da in diesem Fall *Export* auf Klassenebene angewandt wurde, wird die gesamte Klasse als Export zur Verfügung gestellt. Der mit einem Export einhergehende Vertrag wird durch die Eigenschaften dieses Attributs festgelegt. Im betrachteten Fall besteht der Vertrag aus dem Typ *IFormatter*.

Listing 9.58 zeigt die Implementierung des Loggers. Zum Zuweisen des benötigten Formatters wurde eine Eigenschaft vom Typ `IFormatter` eingerichtet. Diese wird im Zuge der Protokollierung innerhalb der Methode `Log` zum Formatieren der Nachricht verwendet, bevor diese auf der Konsole ausgegeben wird. Durch Verwendung des Attributes `Import` auf der Ebene dieser Eigenschaft wird festgelegt, dass ihr eine exportierte Instanz von `IFormatter` zugewiesen werden soll. Für die Definition des Vertrages wird auch hier, wie beim Export in Listing 9.57, der Typ `IFormatter` herangezogen. Diese explizite Typangabe wäre in diesem Fall jedoch nicht nötig, denn standardmäßig wird der Typ des annotierten Konstrukts herangezogen und dieser lautet im betrachteten Fall auch auf `IFormatter`. Beim Export in Listing 9.57 ist diese Typangabe allerdings notwendig, denn der Name der exportierten Klasse lautet hier auf `SimpleFormatter` und nicht auf `IFormatter`. Die Tatsache, dass es sich beim Ersteren um einen Subtyp des Letzteren handelt, ist an dieser Stelle nicht von Bedeutung.

Für das Befriedigen der als Imports gekennzeichneten Abhängigkeiten ist die Methode `Compose`, welche vom Konstruktor aufgerufen wird, verantwortlich. Sie erstellt zunächst einen *AssemblyCatalog*, welcher die sich gerade in Ausführung befindliche Assembly nach Parts durchsucht. Anschließend wird auf Basis dieses Kataloges ein *CompositionContainer* instanziiert. Der Aufruf von `ComposeParts`, im Zuge dessen das aktuelle Objekt (*this*) übergeben wird, bewirkt, dass die Importe dieses Objekts mit entsprechenden vom *Catalog* gelieferten Exporten verdrahtet werden, sofern auf beiden Seiten ein identischer Vertrag vorliegt und der Export mit dem Import typkompatibel ist. Falls nötig, bewirkt dieser Methodenaufruf auch ein Auflösen von Abhängigkeiten zwischen den einzelnen Parts.

```
public interface IFormatter
{
    string Format(string className, DateTime date, string text, LogLevel level);
}
public enum LogLevel
{
    DEBUG = 0,
    INFO = 1,
    WARNING = 2,
    ERROR = 3,
    FATAL = 4,
    WARP_CORE_BREACH = 5,
    SUPER_NOVA = 6
}
```

Listing 9.56 Schnittstelle `IFormatter` und der verwendete Enum `LogLevel`

```
[Export(typeof(IFormatter))]
public class SimpleFormatter : IFormatter
{
    public SimpleFormatter()
    {
        Console.WriteLine("SimpleFormatter - Konstruktor");
    }

    public string Format(string className, DateTime date, string text, LogLevel level)
    {
        string strLevel = level.ToString().ToUpper();
        string strDate = date.ToString();

        return strDate + " " + strLevel + " " + className + ":\n" + text + "\n\n";
    }
}
```

Listing 9.57 Implementierung eines einfachen Formatters

```
public class Logger
{
    string className;
    [Import(typeof(IFormatter))]
    public IFormatter formatter { get; set; }

    public Logger(string className)
    {
        this.className = className;
        Compose();
    }

    public void Log(string text, LogLevel level)
    {
        string formatted;
        formatted = formatter.Format(className, DateTime.Now, text, level);
        Console.WriteLine(formatted);
    }

    private void Compose()
    {
        var catalog = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
        var container = new CompositionContainer(catalog);
        container.ComposeParts(this);
    }
}
```

Listing 9.58 Implementierung des durch MEF veränderbaren Loggers

Imports

Neben dem Importieren von Instanzen exportierter Klassen in typkompatible Felder bietet MEF noch weitere Möglichkeiten, um Abhängigkeiten aufzulösen. Dieser Abschnitt geht auf diese nun näher ein.

Auflistungen importieren

Der Import in Listing 9.58 sieht vor, dass es exakt einen passenden *Export* gibt. Gibt es keinen oder mehr, wird vom Container im Zuge des Verdrahtens eine Ausnahme ausgelöst. Als Alternative dazu bietet das Attribut *ImportMany* die Möglichkeit, alle passenden Exporte des verwendeten Katalogs über eine durch *IEnumerable* repräsentierte Auflistung oder über ein Array bereitzustellen. Listing 9.59 bis Listing 9.61 veranschaulichen dies. Listing 9.59 zeigt dazu eine Schnittstelle *IDestination*. Diese Schnittstelle sieht eine Methode *Write* vor, an welche zu protokollierende Nachrichten übergeben werden können. Listing 9.60 zeigt zwei Implementierungen von *IDestination*: Die *ConsoleDestination* schreibt den übergebenen Text auf die Konsole; die *FileDestination* in eine Datei namens *log.txt*. Beide werden mittels *Export* zur Verfügung gestellt, wobei für den Vertrag jeweils der Typ *IDestination* herangezogen wird. Eine erweiterte Variante des Loggers findet sich in Listing 9.61. Diese weist eine Eigenschaft *Destinations* vom Typ *IEnumerable* auf. Annotiert wurde sie mittels *ImportMany*; für den Vertrag wurde ebenfalls der Typ *IDestination* verwendet. Dies veranlasst den Container, in diese Eigenschaft sämtliche passenden Exports zu importieren. Durch das

Setzen der Eigenschaft *AllowRecomposition* wird darüber hinaus festgelegt, dass Änderungen an der Menge der im Container zur Verfügung stehenden passenden Exports an diese Eigenschaft weitergegeben werden sollen. Standardmäßig ist diese Option nicht aktiviert.

```
public interface IDestination
{
    void Write(string text);
}
```

Listing 9.59 Schnittstelle IDestination

```
[Export(typeof(IDestination))]
public class ConsoleDestination: IDestination
{
    public void Write(string text)
    {
        Console.WriteLine(text);
    }
}

[Export(typeof(IDestination))]
public class FileDestination : IDestination
{
    public void Write(string text)
    {
        File.AppendAllText("log.txt", text + "\n");
    }
}
```

Listing 9.60 Implementierungen von IDestination

```
public class Logger
{
    string className;
    [Import(typeof(IFormatter))]
    public IFormatter Formatter { get; set; };

    [ImportMany(typeof(IDestination), AllowRecomposition=true)]
    public IEnumerable<IDestination> Destinations { get; set; }

    public Logger(string className)
    {
        this.className = className;
        Compose();
    }

    public void Log(string text, LogLevel level)
    {
        string formatted;
        formatted = Formatter.Format(className, DateTime.Now, text, level);
        foreach(IDestination d in Destinations) {
            d.Write(formatted);
        }
    }
}
```

```

    }

    }

    private void Compose()
    {
        var catalog = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
        var container = new CompositionContainer(catalog);
        container.ComposeParts(this);
    }
}

```

Listing 9.61 Durch MEF erweiterbarer Logger

Import in Felder und Konstruktorargumente

Bis jetzt wurde lediglich beschrieben, wie Abhängigkeiten in Eigenschaften importiert werden können. Als Ziel eines *Imports* können jedoch auch Felder und Konstruktorargumente herangezogen werden. Felder sind dazu auf dieselbe Art und Weise wie Eigenschaften mittels *Import* oder *ImportMany* zu annotieren. Für Konstruktoren, in deren Argumente Abhängigkeiten importiert werden sollen, steht – wie in Listing 9.62 gezeigt – das Attribut *ImportingConstructor* zur Verfügung. In diesem Beispiel wird eine Implementierung von *IDestination* gezeigt, welche die übergebene Nachricht in eine Datei schreibt, wobei im Gegensatz zum weiter oben betrachteten Beispiel der Name dieser Datei konfigurierbar ist. Dazu erwartet der Konstruktor als erstes Argument eine Instanz von *FileDestinationConfig*. Diese weist eine Eigenschaft *FileName* auf, welche den gewünschten Namen der Datei enthält. Um anzuzeigen, dass vom Container die benötigte Instanz von *FileDestinationConfig* in das Konstruktorargument zu importieren ist, wird der Konstruktor mit dem Attribut *ImportingConstructor* versehen.

```

[Export(typeof(IDestination))]
public class ConfigurableFileDestination : IDestination
{
    FileDestinationConfig config;
    [ImportingConstructor]
    public ConfigurableFileDestination(FileDestinationConfig config)
    {
        this.config = config;
    }

    public void Write(string text)
    {
        File.AppendAllText(config.FileName, text + "\n");
    }
}

```

Listing 9.62 Import in Konstruktor

Verzögerter Import

Der Import von Parts kann auch verzögert gestaltet werden, sodass die einzelnen Abhängigkeiten erst beim ersten Zugriff aufgelöst werden. Dazu wird die generische Klasse `Lazy<T>` herangezogen. Die Verwendung dieses Generics wird mit Listing 9.63 demonstriert, indem für das zu importierende Feld der Typ `Lazy<IFormatter>` festgelegt wird. In der Methode `Log` wird nun auf den Formatter über Eigenschaft `Value` von `Lazy<T>` zugegriffen. Diese veranlasst beim ersten Zugriff den Import.

```
public class Logger
{
    [...]

    [Import(typeof(IFormatter))]
    private Lazy<IFormatter> formatter = null;

    [...]

    public void Log(string text, LogLevel level)
    {
        string formatted;
        formatted = formatter.Value.Format(className, DateTime.Now, text, level);
        foreach (IDestination d in destinations)
        {
            d.Write(formatted);
        }

    }

    [...]
}
```

Listing 9.63 Verzögerter Import durch `Lazy<...>`

Exports

Zu den verschiedenen Spielarten zur Verwendung von Imports stehen auch entsprechende Export-Konstrukte zur Verfügung. Dieser Abschnitt geht auf diese ein.

Export von Eigenschaften

Neben dem Export ganzer Klassen besteht auch die Möglichkeit, die sich hinter Eigenschaften verbergenden Werte zu exportieren. Dazu sind die einzelnen Eigenschaften analog zu den zu exportierenden Klassen mittels `Export` zu annotieren. Listing 9.64 demonstriert dies mit der Eigenschaft `Config`, welche die in Listing 9.62 importierte Instanz von `FileDestinationConfig` zur Verfügung stellt.

```
public class FileDestinationConfigManager
{
    [Export(typeof(FileDestinationConfig))]
    public FileDestinationConfig Config
    {
        get
```

```
{
return new FileDestinationConfig() { FileName = "anotherLog.txt" };
}
}
```

Listing 9.64 Export einer Eigenschaft

Neben der Tatsache, dass durch diesen Mechanismus die Erzeugung von zu exportierenden Instanzen kontrolliert werden kann, bietet er auch die Möglichkeit, Instanzen von Klassen, deren Quellcode nicht zur Verfügung steht, zu exportieren.

Export von Methoden als Delegaten

In manchen Situationen bietet sich zur Realisierung von Einsprungpunkten der Einsatz von Delegaten als Ersatz für Schnittstellen an. Für diese Fälle bietet MEF die Möglichkeit, Methoden als Delegaten zu exportieren. Dazu ist die gewünschte Methode lediglich mit dem Attribut `Export` zu versehen. Listing 9.65 demonstriert dies. Hier wird die Methode `Route`, welche einen `className` sowie ein `LogLevel` entgegennimmt und sich daraufhin für die zu verwendenden Destinations entscheidet, mittels `Export` als Delegate exportiert. Zu Demonstrationszwecken findet an dieser Stelle ein String und kein Typ zur Definition des Vertrages Verwendung. Um Konflikte mit anderen Frameworks zu vermeiden, wurde dieser String mit einem Präfix, der mit dem Namen des Projekts korreliert, versehen.

```
[Export("MefSample::Router")]
public static IEnumerable<IDestination> Route(string className, LogLevel level)
{
    List<IDestination> result = new List<IDestination>();
    result.Add(new FileDestination());
    if (level > LogLevel.INFO) result.Add(new ConsoleDestination());

    return result;
}
```

Listing 9.65 Export eines Delegates

Listing 9.66 demonstriert den Import von Delegates, indem ein Feld vom Typ des in Listing 9.65 exportierten Delegates implementiert wird. Damit die Verträge übereinstimmen, wurde für diesen Import derselbe String wie in Listing 9.65 herangezogen. Alternativ dazu können auch sämtliche andere hier besprochenen Möglichkeiten für den Import von Abhängigkeiten mit Delegates verwendet werden.

```
public class Logger
{
    [Import("MefSample::Router")]
    private Func<string, LogLevel, IEnumerable<IDestination>> routeMessage = null;

    [...]
}
```

Listing 9.66 Import eines Delegates

Metadaten

In manchen Situationen kann es notwendig sein, Informationen über Exports zu erhalten. Aus diesem Grunde besteht die Möglichkeit, für diese durch Verwendung des Attributs `ExportMetadata` Metadaten festzulegen.

Nicht typisierte Metadaten

Die einfachste Möglichkeit zum Exportieren von Metadaten besteht in der Verwendung von nicht typisierten Metadaten. In Listing 9.67 wird diese Möglichkeit benutzt, um für die beiden weiter oben betrachteten Implementierungen von *IDestination* eine Eigenschaft *type* auf Metaebene zu definieren. Im Falle der *ConsoleDestination* wird diese auf *console* gesetzt; im Fall der *FileDestination* auf *disk*.

```
[Export(typeof(IDestination))]
[ExportMetadata("type", "console")]
public class ConsoleDestination: IDestination
{
    public void Write(string text)
    {
        Console.WriteLine(text);
    }
}

[Export(typeof(IDestination))]
[ExportMetadata("type", "disk")]
public class FileDestination : IDestination
{
    public void Write(string text)
    {
        File.AppendAllText("log.txt", text + "\n");
    }
}
```

Listing 9.67 Export von Metadaten

Diese Metadaten werden in Listing 9.68 neben den exportierten Destinations konsumiert. Dazu wird für die zu importierende Eigenschaft *Destinations* ein Array des Typs `Lazy<T, TMetadata>` herangezogen. *T* bezeichnet dabei den Typ des Imports; *TMetadata* einen Typ, welcher die dazugehörigen Metadaten beinhaltet. Ersterer wird auf *IDestination* festgelegt; Letzterer auf ein `IDictionary<string, object>`. Die Methode `Log` iteriert sämtliche Destinations. Dabei wird über die Eigenschaft *Metadata* auf das Dictionary mit den Metadaten zugegriffen sowie mittels *Value* auf den eigentlichen Import.

```
public class Logger
{
    string className;
    [Import(typeof(IFormatter))]
    private IFormatter formatter = null;

    [ImportMany(typeof(IDestination))]
    public Lazy<IDestination, IDictionary<string, object>>[] Destinations { get; set; }
```



```

public Logger(string className)
{
    this.className = className;
    Compose();
}

public void Log(string text, LogLevel level)
{
    string formatted;
    formatted = formatter.Format(className, DateTime.Now, text, level);
    foreach (var d in Destinations)
    {
        Console.WriteLine("type: " + d.Metadata["type"].ToString());
        d.Value.Write(formatted);
    }
}

private void Compose()
{
    var catalog = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
    var container = new CompositionContainer(catalog);
    container.ComposeParts(this);
}

```

Listing 9.68 Importieren von Metadaten

Typisierte Metadaten

Neben der soeben vorgestellten generischen Möglichkeit zur Bereitstellung und Konsumierung von Metadaten ist es auch möglich, mit Metadaten in typisierter Manier zu arbeiten. Dazu können benutzerdefinierte Attribute, welche von `ExportAttribute` erben und die Metadaten über Eigenschaften anbieten, implementiert werden. Listing 9.69 demonstriert dies. Zunächst wird eine Schnittstelle definiert, welche die benötigten Eigenschaften, die nur gelesen werden können, aufweist. Im betrachteten Fall beschränkt sich diese Menge auf die Eigenschaft `type`.

Diese Schnittstelle wird von der Klasse `DestinationAttribute`, welche zusätzlich von `ExportAttribute` erbt, implementiert. Der Konstruktor dieser Klasse delegiert an den Konstruktor der Basisklasse weiter und gibt dabei den Vertrag an, welcher von den später annotierten Parts angenommen werden soll. Darüber hinaus wird die über die Schnittstelle vorgegebene Eigenschaft implementiert und ihr zusätzlich auch ein Setter spendiert. Damit diese Klasse als Attribut verwendet werden kann, wird sie mittels `MetadataAttribute` sowie `AttributeUsage` annotiert. Bei der Verwendung von `AttributeUsage` wird `AttributeTargets.Class` sowie `AllowMultiple=false` angegeben. Ersteres legt fest, dass das implementierte Attribut lediglich auf Klassenebene verwendet werden darf; Letzteres schließt eine mehrfache Verwendung pro Klasse aus.

```

public interface IDestinationMetaData {
    String type { get; }
}

[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class DestinationAttribute : ExportAttribute, IDestinationMetaData

```

```
{
public DestinationAttribute() : base(typeof(IDestination)) { }
public String type { get; set; }
}
```

Listing 9.69 Benutzerdefiniertes Attribut zum Bereitstellen von Metadaten

Ein Beispiel für den Einsatz dieses benutzerdefinierten Attributs findet sich in Listing 9.70. Eine Angabe von Export ist bei Verwendung solcher Attribute nicht mehr nötig und der Vertrag des Exports wird über das Attribut festgelegt.

```
[Destination(type="console")]
public class ConsoleDestination: IDestination
{
public void Write(string text)
{
Console.WriteLine(text);
}
}

[Destination(type="disk")]
public class FileDestination : IDestination
{
public void Write(string text)
{
File.AppendAllText("log.txt", text + "\n");
}
}
```

Listing 9.70 Verwendung eines benutzerdefinierten Attributs zum Exportieren von Metadaten

Wie auf solche Metadaten zugegriffen werden kann, zeigt der Logger in Listing 9.71. Für die zu importierenden Destinations wird nun ein Array des Typs `Lazy<IDestination, IDestinationMetaData>` angelegt, wobei es sich bei `IDestinationMetaData` um die in Listing 9.69 gezeigte Schnittstelle handelt. In der Methode `Log` werden, wie gewohnt, sämtliche Destinations durchlaufen und dabei mittels `MetaData` auf die Metadaten sowie mittels `Value` auf den eigentlichen Import zugegriffen. Der Unterschied zur bereits betrachteten Implementierung ist jedoch, dass es sich nun bei `Metadata` um eine Implementierung von `IDestinationMetaData`, welche typisierte Eigenschaften für sämtliche Metadaten anbietet, handelt. Diese Implementierung wird von MEF dynamisch erzeugt. Aus diesem Grund ist darauf zu achten, dass diese Schnittstelle lediglich nur-lesbare Eigenschaften aufweist. Streng genommen muss die Attribut-Implementierung diese Schnittstelle auch gar nicht implementieren: Es muss lediglich die Eigenschaften aus der Schnittstelle beinhalten.

```
public class Logger
{
string className;
[Import(typeof(IFormatter))]
private IFormatter formatter = null;

[ImportMany(typeof(IDestination))]
```

```
public Lazy<IDestination, IDestinationMetadata>[] destinations { get; set; }

public Logger(string className)
{
    this.className = className;
    Compose();
}

public void Log(string text, LogLevel level)
{
    string formatted;
    formatted = formatter.Format(className, DateTime.Now, text, level);
    foreach (var d in destinations)
    {
        Console.WriteLine("sending message to destination of type " + d.Metadata.type);
        d.Value.Write(formatted);
    }
}

private void Compose()
{
    var catalog = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
    var container = new CompositionContainer(catalog);
    container.ComposeParts(this);
}
```

Listing 9.71 Import der durch ein benutzerdefiniertes Attribut exportierten Parts und Metadaten

Kataloge

Kataloge werden verwendet, um Informationen über Parts bereitzustellen. Nachdem dieser Abschnitt die unterschiedlichen Arten von Katalogen vorgestellt hat, wird gezeigt, wie diese gefiltert werden können.

Arten von Katalogen

In den vorangegangenen Beispielen wurden lediglich Kataloge der Klasse `AssemblyCatalog`, welche eine angegebene Assembly nach Parts durchsucht, verwendet. Daneben stehen noch weitere Arten von Katalogen zur Verfügung: Ein `DirectoryCatalog` durchsucht beispielsweise sämtliche Assemblys eines angegebenen Verzeichnisses nach Parts. Ein `TypeCatalog` bietet hingegen nur jene Parts an, deren Typen ihm manuell bekannt gemacht worden sind. Der `AggregationCatalog` bietet die Möglichkeit, unterschiedliche Kataloge zu kombinieren, indem er beliebig viele Kataloge aufnimmt und an diese weiterdelegiert.

Listing 9.72 demonstriert die Verwendung dieser Katalog-Implementierungen. Zunächst wird ein `AssemblyCatalog` für die aktuelle Assembly erstellt; danach ein `DirectoryCatalog`, welcher das Verzeichnis *Plugins* durchsucht, sowie ein `TypeCatalog`, dem der Part `AdvancedSimonSaysDestination` bekannt gemacht wird. Anschließend findet die Instanziierung eines `AggregationCatalog` statt. Dieser nimmt die anderen drei Kataloge auf und wird in weiterer Folge vom Container zum Auflösen der Abhängigkeiten in der aktuellen Klasse verwendet.

```
private void Compose()
{
    var catalog1 = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
    var catalog2 = new DirectoryCatalog("Plugins");
    var catalog3 = new TypeCatalog(typeof(AdvancedPlugin.AdvancedSimonSaysDestination));
    var catalog = new AggregateCatalog(catalog1, catalog2, catalog3);

    var container = new CompositionContainer(catalog);
    container.ComposeParts(this);
}
```

Listing 9.72 Demonstration der Verwendung von Katalogen

Falls diese im Lieferumfang von MEF enthaltenen Kataloge nicht ausreichen sollten, können auch benutzer-definierte Kataloge implementiert werden, indem von *ComposablePartCatalog* geerbt wird.

Kataloge filtern

Nicht immer sollen alle über einen Katalog zur Verfügung gestellten Parts Verwendung finden. Deswegen kann ein Dekorator implementiert werden, welcher die Parts eines ummantelten Katalogs filtert. Listing 9.73 zeigt eine Implementierung eines solchen Katalogs, welcher teilweise aus der Dokumentation zu MEF entnommen wurde. Der Konstruktor nimmt zum einen den zu dekorierenden Katalog sowie zum anderen einen Lambdaausdruck entgegen. Mit diesem Ausdruck werden die Parts des dekorierten Katalogs gefiltert, indem er an dessen Methode *Where* übergeben wird. Das Ergebnis dieses Aufrufs wird im privaten Feld *_partsQuery* abgelegt, wobei sich die Eigenschaft *Parts* auf dieses Feld abstützt.

```
public class FilteredCatalog : ComposablePartCatalog
{
    private readonly IQueryable<ComposablePartDefinition> _partsQuery;
    public FilteredCatalog(ComposablePartCatalog inner,
        Expression<Func<ComposablePartDefinition, bool>> expression)
    {
        _partsQuery = inner.Parts.Where(expression);
    }

    public override IQueryable<ComposablePartDefinition> Parts
    {
        get
        {
            return _partsQuery;
        }
    }
}
```

Listing 9.73 Filterbarer Katalog

Ein Beispiel für die Anwendung dieses Dekorators findet sich in Listing 9.74. Hier wird ein *AssemblyCatalog* zusammen mit einem Lambdaausdruck an den Konstruktor von *FilteredCatalog* übergeben. Der Ausdruck definiert einen Filter. Dieser legt fest, dass Exporte mit dem Vertrag *typeof(IDestination)* den Wert *console* in der Metadaten-Eigenschaft *type* aufweisen müssen. Zur Vereinfachung wird dabei jeweils nur der erste

Export der einzelnen Parts, welcher über die Methode `First` bezogen wird, berücksichtigt. Anschließend wird dieser Katalog wie gehabt verwendet, indem er an einen `CompositionContainer` übergeben wird, welcher die Abhängigkeiten der aktuellen Klassen auflöst.

```
private void Compose()
{
    var catalog = new AssemblyCatalog(System.Reflection.Assembly.GetExecutingAssembly());
    var filteredCatalog = new FilteredCatalog(catalog,
    p =>
    p.ExportDefinitions.First().ContractName == typeof(IDestination).ToString()
    && p.ExportDefinitions.First().Metadata.ContainsKey("type")
    && p.ExportDefinitions.First().Metadata["type"].ToString() == "console"
    || p.ExportDefinitions.First().ContractName != typeof(IDestination).ToString()
    );

    var container = new CompositionContainer(filteredCatalog);
    container.ComposeParts(this);
}
```

Listing 9.74 Filtern eines Katalogs

Fazit zu MEF

Die Möglichkeit zur Implementierung von Plugin-Mechanismen besteht seit den ersten Tagen von .NET. Mit MEF wurde jedoch nun von Microsoft ein Framework bereitgestellt, welches solch ein Vorhaben erheblich erleichtert.

Dieses Framework bietet einige Möglichkeiten zur Auflösung von Abhängigkeiten, welche bereits von Inversion of Control (IoC)-Frameworks, wie *Spring.NET* [SPRING01] oder *Castle-Project* [CASTLE01], bekannt sind. Somit wird durch den Einsatz von MEF auch die Testbarkeit der entwickelten Applikation erhöht, da im Zuge von automatisierten Tests Attrappen (Mock-Objekte) anstatt der eigentlichen Abhängigkeiten verwendet werden können.

Allerdings gibt es auch einige Unterschiede zu IoC-Containern, deren primäres Ziel das Auflösen von Abhängigkeiten *innerhalb* einer Applikation ist, denn MEF wurde primär für die Bereitstellung von Plugin-Mechanismen, welche Abhängigkeiten mittels *externer* Komponenten (Plugins) auflösen, entworfen. So sieht MEF zum Beispiel keine zentrale Konfiguration vor, da die einzelnen Parts im Zuge des Durchsuchens der gefundenen Plugins ermittelt werden. Auch AOP²-spezifische Features, wie Transaktionshandling, sind für MEF aus demselben Grund nicht vorgesehen. Somit ist MEF weniger als Alternative zu IoC-Containern, sondern mehr als Ergänzung solcher zu sehen.

HINWEIS Interessierte finden im Anhang eine Einführung in den IoC-Container *Spring.NET*. Diese zeigt, wie *Spring.NET* zusammen mit *ASP.NET MVC* verwendet werden kann, um die Wartbarkeit und vor allem Testbarkeit einer Webapplikation zu erhöhen. Die gezeigten Techniken sind allerdings auch auf andere Formen von Applikationen übertragbar.

² Aspektorientierte Programmierung

System.Security

Der Namensraum System.Security enthält zahlreiche Klassen für die in nachfolgender Tabelle genannten Zwecke.

Zweck	Namensraum	Status in .NET 2.0	Status in .NET 3.0	Status in .NET 3.5
Code Access Security (CAS)	System.Security und System.Security.Permissions		Leicht erweitert	
Verschlüsselte Zeichenketten	System.Security, Klasse SecureString	Neu!		
Public Key Cryptography Standard (PKCS)	System.Security.Cryptography.Pkcs	Neu!		
X.509-Zertifikate	System.Security.Cryptography.X509Certificates	Erweitert (insbes. für den Zugriff auf Zertifikatsspeicher)		Leicht erweitert
Signierung von XML-Dokumenten gemäß XML-Signature Syntax and Processing [W3C01]	System.Security.Cryptography.Xml	Neu!		
Symmetrische Verschlüsselung (z. B. AES, DES, RC2, Rijndael, TripleDES) und asymmetrische Verschlüsselung (z. B. DSA, RSA, ECDSA, ECDH)	System.Security.Cryptography			Erweitert (siehe unten)
Hash-Verfahren (z. B. MD5, PIPEMD160, SHA1, SHA256, SHA284, SHA512)	System.Security.Cryptography			
SSL-Authentifizierung	System.Security.Authentication	Neu!		
Berechtigung von Windows-Systembausteinen (Zugriffsrechte-listen)	System.Security.AccessControl	Neu!		

Tabelle 9.6 Namensraum System.Security

Die meisten der o.g. Themen können aufgrund ihrer Komplexität nicht in diesem Buch behandelt werden. Zwei der neu hinzugekommenen Funktionen sollen hier jedoch kurz vorgestellt werden: verschlüsselte Zeichenketten (SecureString) und der Zugriff auf Berechtigungen von Windows-Systembausteinen (Unter-namensraum System.Security.AccessControl.*).

HINWEIS Das .NET Framework enthält einige neue Sicherheitsklassen. Dies sind zum Teil neue Verfahren, z. B.

- Advanced Encryption Standard (AES) mit Schlüsselgrößen von 128 und 256 Bit für die Verschlüsselung (bisher schon vorhanden, aber nicht zertifiziert gemäß Federal Information Processing Standard (FIPS))
- Secure Hash Algorithm (SHA-256 und SHA-384) für das Hashing
- Elliptic Curve Digital Signature Algorithm (ECDSA) mit 256-Bit und 384-Bit-Kurven Prime-Moduli zur Signatur
- Elliptic Curve Diffie-Hellman (ECDH) mit 256-Bit und 384-Bit-Kurven Prime-Moduli für Schlüsselaustausch/Geheimvereinbarung

Das .NET Framework unterstützt alle Sicherheitsverfahren der National Security Agency (NSA) Suite B.

Außerdem gibt es einige neue Implementierungen für bestehende Verfahren, z. B. Klasse MD5Cng statt MD5 und SHA256Cng statt SHA256. *Cng* steht dabei jeweils für *Cryptography API Next Generation*, ein Projekt bei Microsoft, mit dem das bestehende Cryptography API langfristig durch eine neue, flexiblere Schnittstelle ersetzt werden soll.

System.Security.SecureString

Die Klasse `SecureString` legt eine Zeichenkette in verschlüsselter Form im Hauptspeicher ab, sodass ein Schutz für sensible Daten (wie Kennwörter) besteht. Damit wird seit .NET 2.0 das Sicherheitsproblem eliminiert, welches dadurch entsteht, dass der Entwickler keinen Einfluss darauf hat, wann erzeugte Zeichenketten der automatischen Speicherbereinigung unterworfen und im Speicher überschrieben werden. Angreifer könnten so durch ein Speicherabbild sensible Informationen erhalten.

Neben der verschlüsselten Ablage unterscheidet sich `SecureString` von `System.String` dadurch, dass ein einmal gesetzter Wert noch verändert werden kann (`AppendChar()`, `RemoveAt()`, `SetAt()`), bis die Zeichenkette explizit als nicht mehr veränderbar (`MakeReadOnly()`) gekennzeichnet wird. Außerdem hat die Anwendung Einfluss auf die Vernichtung des Werts aus dem Hauptspeicher (`Clear()`).

Zweck eines `SecureString` ist die Übergabe an eine andere Klasse, z. B. das Attribut `Password` in der Klasse `ProcessStartInfo` zum Start eines Prozesses unter einem anderen Benutzerkonto. Ein entsprechendes Beispiel finden Sie in dem Kapitel 7 »Konsolenanwendungen«.

Die Rückumwandlung eines `SecureString`-Objekts in eine normale Zeichenkette ist leider nicht trivial. Man benötigt dazu die Klasse `System.Runtime.InteropServices.Marshal` (siehe Listing).

```
public void SecureStringRueckumwandlung()
{
    SecureString s = new SecureString();
    s.AppendChar('g');
    s.AppendChar('e');
    s.AppendChar('h');
    s.AppendChar('e');
    s.AppendChar('i');
    s.AppendChar('m');
    s.MakeReadOnly();
    IntPtr p = System.Runtime.InteropServices.Marshal.SecureStringToBSTR(s);
    string KlartextKennwort = System.Runtime.InteropServices.Marshal.PtrToStringUni(p);
    Demo.Print(KlartextKennwort);
}
```

Listing 9.75 Ausgabe eines `SecureString`

System.Security.AccessControl.*

Der Namensraum enthält zahlreiche Klassen zur Verwaltung von Zugriffsrechtelisten (Access Control Lists, ACLs). Dieser Namensraum wird insbesondere von den Klassen `System.IO.File`, `System.IO.Directory`, `Microsoft.Win32.RegistryKey` und `System.Threading.Semaphore` verwendet. Für jede Art von Ressource, deren ACLs verwaltet werden können, bietet der Namensraum `AccessControl` eine Klasse an, die von `System.Security.AccessControl.ObjectSecurity` abgeleitet ist. Beispielsweise dient `System.Security.AccessControl.FileSecurity` dazu, die ACLs einer Datei im Dateisystem zu lesen und zu verarbeiten.

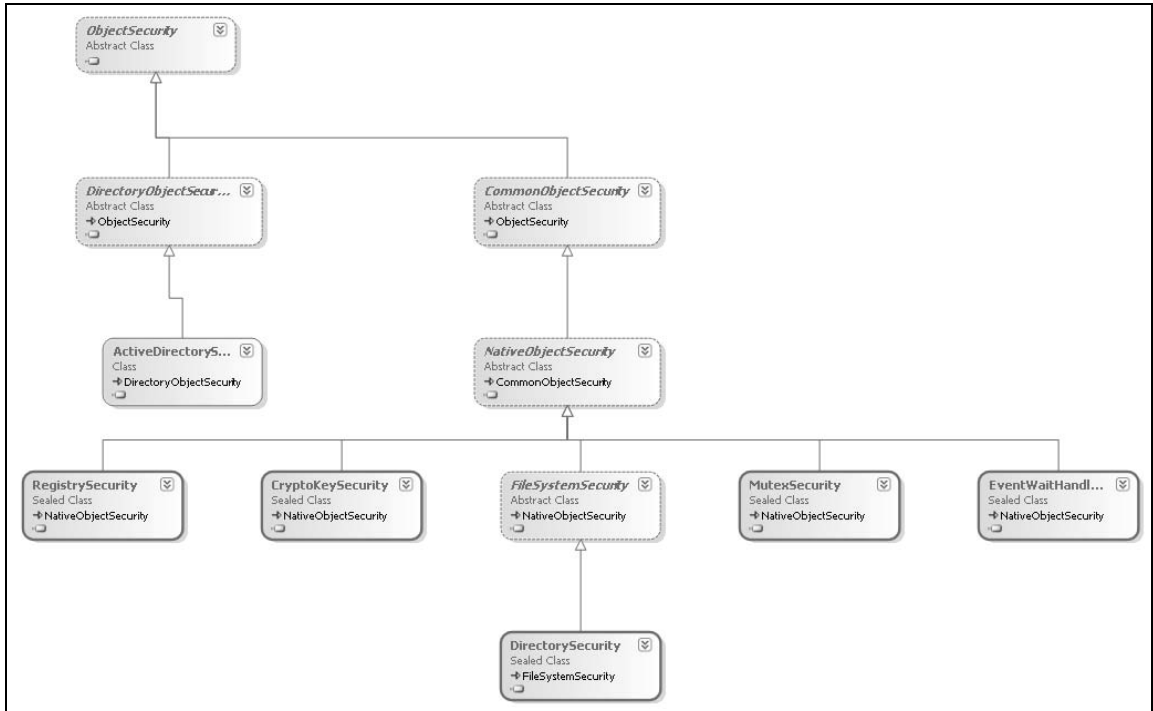


Abbildung 9.25 Vererbungshierarchie der Klassen zur ACL-Speicherung

Über die gesamte .NET-Klassenbibliothek verteilt findet man Klassen, die eine Methode `GetAccessControl()` besitzen, welche ein von der Klasse `ObjectSecurity` abgeleitetes Objekt liefert. Beispiele für solche Klassen sind:

- `System.IO.File`
- `System.IO.Directory`
- `System.IO.FileInfo`
- `System.IO.DirectoryInfo`
- `Microsoft.Win32.RegistryKey`
- `System.Threading.Semaphore`

Kontenname und Security Identifier (SID)

Der Namensraum `System.Security.AccessControl` verwendet Klassen aus `System.Security.Principal` zur Darstellung der Berechtigungsträger (Benutzer und Gruppen). `System.Security.Principal` unterstützt die beiden in Windows bekannten Bezeichner für Berechtigungsträger:

- Prinzipalname (z. B. `ITVisions\hs`) durch die Klasse `System.Security.Principal.NTAccount`
- Security-Identifier (z. B. `S-1-5-21-565061207-3232948068-1095265983-500`) durch die Klasse `System.Security.Principal.SecurityIdentifier`.

Jeder Benutzer und jede Benutzergruppe besitzt einen so genannten Security Identifier (kurz: SID), der den Benutzer bzw. die Gruppe eindeutig identifiziert. Ein SID ist ein Zahlen-Array variabler Länge. In Textform wird der SID mit einem beginnenden »S« dargestellt. Die Umwandlung eines Prinzipalnamens in einen SID und umgekehrt erfolgt mithilfe der Methode `Translate()` in der Klasse `IdentityReference`, welche die Basis-Klasse für `NTAccount` und `SecurityIdentifier` ist.

In Windows eingebaute Benutzer und Gruppen besitzen einen so genannten *Well-Known-Security-Identifier*. .NET stellt seit Version 2.0 eine Auflistung `System.Security.Principal.WellKnownSidType` bereit, die man zur Instanziierung der Klasse `SecurityIdentifier` einsetzen kann. Man umgeht damit die sprachspezifischen Unterschiede des Betriebssystems (*Guests / Gäste*).

```
Demo.Print("Administratoren: " + new
System.Security.Principal.SecurityIdentifier(System.Security.Principal.WellKnownSidType.
BuiltInAdministratorsSid, null).Value);
```

Einige eingebaute Benutzer und Gruppen beinhalten den SID der Domäne in ihrem eigenen SID. In diesem Fall muss bei der Instanziierung der Klasse `SecurityIdentifier` der Domänen-SID mit angegeben werden. Leider schweigt sich die Dokumentation darüber aus, woher man den Domain-SID mit .NET-Methoden bekommt. Auch im WWW findet man noch kein Beispiel dafür.

Eine andere Möglichkeit zum Zugriff auf eingebaute Benutzer und Gruppen besteht in der Verwendung der in der Security Descriptor Definition Language (SDDL) definierten Abkürzungen für die eingebauten Benutzer und Gruppen (siehe Tabelle).

```
Demo.Print("Administratoren: " + new System.Security.Principal.SecurityIdentifier("BA").Value);
```

SDDL-Abkürzung	Bedeutung
»AO«	Account operators
»AN«	Anonymous logon
»AU«	Authenticated users
»BA«	Built-in administrators
»BG«	Built-in guests
»BO«	Backup operators
»BU«	Built-in users
»CA«	Certificate server administrators ▶

SDDL-Abkürzung	Bedeutung
»CG«	Creator group
»CO«	Creator owner
»DA«	Domain administrators
»DC«	Domain computers
»DD«	Domain controllers
»DG«	Domain guests
»DU«	Domain users
»EA«	Enterprise administrators
»ED«	Enterprise domain controllers
»WD«	Everyone
»PA«	Group Policy administrators
»IU«	Interactively logged-on user
»LA«	Local administrator
»LG«	Local guest
»LS«	Local service account
»SY«	Local system
»NU«	Network logon user
»NO«	Network configuration operators
»NS«	Network service account
»PO«	Printer operators
»PS«	Personal self
»PU«	Power users
»RS«	RAS servers group
»RD«	Terminal server users
»RE«	Replicator
»RC«	Restricted code
»SA«	Schema administrators
»SO«	Server operators
»SU«	Service logon user

Tabelle 9.7 SDDL-Abkürzungen für eingebaute Benutzer und Gruppen

DACL auslesen

Das erste Listing zeigt die Ausgabe der einzelnen Access Control Entries (ACEs) der Discretionary ACL (DACL) einer Datei im Dateisystem. Zunächst wird die Methode `GetAccessControl()` in der Klasse `System.IO.File` genutzt, um den kompletten Sicherheitsdeskriptor in Form eines `FileSecurity`-Objekts zu erhalten.

`GetOwner()` liefert den Besitzer der Datei. Als Parameter ist anzugeben, in welcher Form (`SecurityIdentifier` oder `NTAccount`) man die Information erhalten möchte.

```
Demo.Print("Besitzer SID: " +  
objFS.GetOwner(typeof(System.Security.Principal.SecurityIdentifier)).Value);  
Demo.Print("Besitzer Name: " + objFS.GetOwner(typeof(System.Security.Principal.NTAccount)).Value);
```

Das `FileSecurity`-Objekt liefert über `GetAccessRules()` die einzelnen ACEs der DACL in Form von `FileSystemAccessRule`-Objekten. `GetAuditRules()` würde die ACEs der System-ACL (SACL) liefern, die die Überwachungseinstellungen enthält. Auch hierbei ist wieder die gewünschte Form für den Berechtigungsträger anzugeben.

Bei `GetAccessRules()` kann bestimmt werden, ob nur die expliziten (erster Boolean-Parameter) und/oder die vererbten ACEs in der Ergebnismenge enthalten sein sollen. Die expliziten ACEs erscheinen immer zuerst in der Liste.

```
// Auslesen einer Datei-ACL  
public void SDAuslesen()  
{  
    const string DATEI = @"c:\HolgerSchwichtenberg.doc";  
    // Hole ACL  
    FileSecurity objFS = File.GetAccessControl(DATEI);  
    // Besitzer  
    Demo.Print("Besitzer SID: " +  
        objFS.GetOwner(typeof(System.Security.Principal.SecurityIdentifier)).Value);  
    Demo.Print("Besitzer Name: " +  
        objFS.GetOwner(typeof(System.Security.Principal.NTAccount)).Value);  
    // Hole einzelne ACEs aus ACL  
    AuthorizationRuleCollection ACEs = objFS.GetAccessRules(true, true,  
        typeof(System.Security.Principal.NTAccount));  
    // Schleife über alle ACEs  
    foreach (FileSystemAccessRule ACE in ACEs)  
    {  
        Demo.Print("Benutzer/Gruppe {0}: {1} {2} ({3})",  
            ACE.IdentityReference.ToString(),  
            ACE.FileSystemRights,  
            ACE.AccessControlType == AccessControlType.Allow ? "zugelassen" : "verweigert",  
            ACE.IsInherited ? "vererbt" : "explizit"  
        );  
    }  
}
```

Listing 9.76 Auslesen einer Datei-ACL [Security.cs]

```

Besitzer SID: S-1-5-32-544
Besitzer Name: BUILTIN\Administrators
Benutzer/Gruppe BUILTIN\Administrators: FullControl zugelassen (explizit)
Benutzer/Gruppe ESSEN\hs: FullControl zugelassen (explizit)
Benutzer/Gruppe ESSEN\hp: Read, Synchronize zugelassen (explizit)
Benutzer/Gruppe BUILTIN\Administrators: FullControl zugelassen (vererbt)
Benutzer/Gruppe NT AUTHORITY\SYSTEM: FullControl zugelassen (vererbt)
Benutzer/Gruppe BUILTIN\Users: ReadAndExecute, Synchronize zugelassen (vererbt)

```

Listing 9.77 Eine mögliche Ausgabe des obigen Beispiels

DACL setzen

Das folgende Listing zeigt das Ergänzen einer ACE zu einer ACL einer Datei. Wieder erfolgt der Zugriff über die statische Methode `GetAccessControl()` der `File`-Klasse. Neue `FileSystemAccessRule`-Objekte können mithilfe von `NTAccount`-Objekten oder `SecurityIdentifier`-Objekten erzeugt werden. Als weitere Parameter werden die zu vergebenden Rechte, z.B.

- `FileSystemRights.Read`
- `FileSystemRights.Write`
- `FileSystemRights.Modify` `FileSystemRights.FullControl`

sowie der Berechtigungstyp (`AccessControlType.Allow` oder `AccessControlType.Deny`) bei der Instanziierung genannt.

```

// Hinzufügen von ACEs
public void SDErgaenzen()
{
    const string DATEI = @"t:\HolgerSchwichtenberg.doc";
    // Hole ACL
    FileSecurity objFS = File.GetAccessControl(DATEI);
    // ACE erzeugen
    FileSystemAccessRule rule = new FileSystemAccessRule(
        new System.Security.Principal.NTAccount(@"Essen\hs"), FileSystemRights.Modify,
        AccessControlType.Deny);
    FileSystemAccessRule rule2 = new FileSystemAccessRule(
        new System.Security.Principal.SecurityIdentifier(
            System.Security.Principal.WellKnownSidType.BuiltinAdministratorsSid, null),
        FileSystemRights.FullControl,
        AccessControlType.Allow);
    // Regel hinzufügen
    objFS.AddAccessRule(rule);
    objFS.AddAccessRule(rule2);
    // Speichern
    File.SetAccessControl(DATEI, objFS);
}
}

```

Listing 9.78 Hinzufügen eines ACE [Security.cs]

System.Runtime.Caching

Gastautor dieses Abschnitts: Manfred Steyer, entnommen aus »NET 4.0 Update« [HS07]

Das .NET Framework enthält seit der ersten Version eine Cache-Implementierung – allerdings lediglich für Web-Applikationen. Um auch andere Applikationen von Caching-Mechanismen profitieren zu lassen, werden entsprechende Konstrukte ab Version 4 für sämtliche Applikationen über den Namespace `System.Runtime.Caching` in der gleichnamigen Assembly angeboten. Die damit bereitgestellte Implementierung ähnelt jener von ASP.NET und sieht zurzeit lediglich einen In-Memory-Cache (`MemoryCache`) vor. Weitere Cache-Implementierungen können jedoch durch Ableiten von der Basisklasse `ObjectCache` entwickelt werden.

Ein Beispiel für die Verwendung von `MemoryCache` findet sich in Listing 9.79. Zunächst wird hier über die statische Eigenschaft `MemoryCache.Default` eine Referenz auf die global bereitgestellte Standard-Instanz von `MemoryCache` bezogen und versucht, über den Schlüssel `someText` einen gecachten String abzurufen. Konnte dieser nicht ermittelt werden, wird der Eintrag aus einer Datei gelesen sowie unter dem mit ihm assoziierten Schlüssel `someText` im Cache abgelegt. Im Zuge dessen wird auch eine `CacheItemPolicy` übergeben. Diese legt fest, wann der Cache-Eintrag wieder aus dem Cache entfernt werden soll. Dazu wird mit der Eigenschaft `SlidingExpiration` festgelegt, nach welchem Zeitraum, in welchem nicht auf das Element zugegriffen wird, dieses zu entfernen ist. Alternativ dazu kann mit der Eigenschaft `AbsoluteExpiration` eine absolute Zeitspanne, nach welcher der Eintrag entfernt werden soll, angeführt werden. Daneben bietet `Priority` vom Enum-Typ `CacheItemPriority` die Möglichkeit zu definieren, dass ein Eintrag gar nicht aus dem Cache entfernt werden soll. Dies wird mit dem Wert `NotRemovable` angezeigt. Zusätzlich zur `SlidingExpiration` wird auch ein `ChangeMonitor` vom Typ `HostFileChangeMonitor` registriert. Dieser überwacht die an den Konstruktor übergebenen Dateien und / oder Ordner auf Änderungen. Im Fall einer Änderung werden die mit der `Policy` assoziierten Einträge aus dem Cache entfernt. Neben dem `HostFileChangeMonitor` existiert auch ein `SqlChangeMonitor`, welcher eine Instanz der seit .NET 2.0 verfügbaren Klasse `SqlDependency` kapselt und diese zur Überwachung von Änderungen in einer SQL Server-Datenbank (ab Version 2005) verwendet. Weitere derartige Mechanismen können durch Ableiten von der Basisklasse `ChangeMonitor` oder einer deren Subklassen bereitgestellt werden.

```
const string source = @"c:\temp\someTextFile.txt";
ObjectCache cache = MemoryCache.Default;
string fileContents = cache["someText"] as string;

if (fileContents == null)
{
    Console.WriteLine("Element war nicht (mehr) im Cache!");
    CacheItemPolicy policy = new CacheItemPolicy();
    policy.SlidingExpiration = TimeSpan.FromSeconds(50);

    List<string> filePaths = new List<string>();
    filePaths.Add(source);
    policy.ChangeMonitors.Add(new HostFileChangeMonitor(filePaths));

    fileContents = File.ReadAllText(source);
    cache.Set("someText", fileContents, policy);
}
Console.WriteLine(fileContents);
```

Listing 9.79 Verwendung von `MemoryCache`

