

## Kapitel 13

# Datenzugriff mit System.Xml und LINQ to XML

### In diesem Kapitel:

XML-Programmierung mit .NET	720
Neuerungen in .NET 2.0	720
Neuerungen in .NET 3.0	721
Neuerungen in .NET 3.5	721
Neuerungen in .NET 4.0	721
Überblick über die XML-Zugriffsmodelle	721
XML-DOM (XmlDocument)	722
XML-Leser (XmlReader)	726
XML-Schreiber (XmlWriter)	728
XPathNavigator (XPath Data Model)	730
LINQ to XML	732
Vergleich der Zugriffsformen	737
Ableiten eines Schemas aus XML-Dokumenten	738
XML Style Sheet Transformations (XSLT)	739

# XML-Programmierung mit .NET

Neben der Möglichkeit, ein DataSet in XML zu exportieren bzw. ein XML-Dokument in ein DataSet zu importieren, bietet das .NET Framework auch verschiedene Möglichkeiten, XML-Dokumente und XML-Fragmente direkt zu bearbeiten.

Im Namensraum `System.Xml` werden folgende Standards des World Wide Web Consortiums (siehe [W3C]) unterstützt:

- XML 1.0 – einschließlich Document Type Definitions (DTD)
- XML Namespaces 1.0
- XML Schema 1.0 (XSD)
- XPath 1.0
- XSLT 1.0
- DOM Level 1 und 2
- XML Encryption

---

**HINWEIS** Neben den hier erwähnten XML-Standards findet man in dem Namensraum `System.Xml` weitere Möglichkeiten zum Zugriff auf XML-Dokumente, die nicht beim W3C standardisiert sind, z. B. `XmlReader`, `XmlWriter` und LINQ to XML.

---

## Neuerungen in .NET 2.0

Im Bereich der XML-Unterstützung gibt es folgende Neuheiten seit .NET 2.0 gegenüber .NET 1.1:

- Die schnellere Klasse `XmlCompiledTransform` ersetzt die Klasse `XmlTransform`
- Erzeugen eines XML-Schemas (XSD) aus einem XML-Dokument mit der Klasse `XmlSchemaInference`
- Schema-Validierung direkt aus der Klasse `XmlDocument` mit der Methode `Validate()`
- Die Klasse `XPathNavigator` kann nun auch Dokumente ändern, wenn als Basis ein `XmlDocument` verwendet wird
- `XmlReader` und `XmlWriter` lösen `XmlTextReader`, `XmlValidationReader` und `XmlTextWriter` ab und besitzen mehr Optionen (z. B. Behandlung von Leerräumen)
- Konvertierungsunterstützung zwischen CLR- und XML-Datentypen in den Klassen `XmlWriter`, `XmlReader` und `XPathNavigator`
- Verschlüsselung von XML-Dokumenten bzw. Dokumentfragmenten auf Basis von W3C XML Encryption

## Neuerungen in .NET 3.0

Der Namensraum *System.Xml* in .NET 3.0 enthält gegenüber .NET 2.0 einige wenige Ergänzungen (z. B. *XmlDictionary* sowie einige abstrakte Basisklassen). Diese Erweiterungen, die in *System.Runtime.Serialization.dll* implementiert sind, werden von WCF (siehe dazu eigenes Kapitel) benötigt. Ob eine Verwendung unabhängig von WCF sinnvoll ist, ist von Microsoft derzeit nicht dokumentiert.

## Neuerungen in .NET 3.5

Neu seit .NET 3.5 ist LINQ to XML, das eine weitere Programmierschnittstelle für das Lesen und Schreiben von XML-Dokumenten bereitstellt. Für das Abfragen von XML-Dokumenten kann hier die LINQ-Syntax anstelle von XPath eingesetzt werden. Auch das Verändern von XML-Dokumenten ist durch eine neue Klasse vereinfacht. Intern basiert LINQ to XML auf der bereits in .NET 1.0 eingeführten Klasse *XmlDocument*.

## Neuerungen in .NET 4.0

Es gibt in .NET 4.0 keine nennenswerten Neuerungen für die Arbeit mit XML im .NET Framework. Visual Studio 2010 enthält einen neuen Designer für XML-Schema-Dateien (XSD).

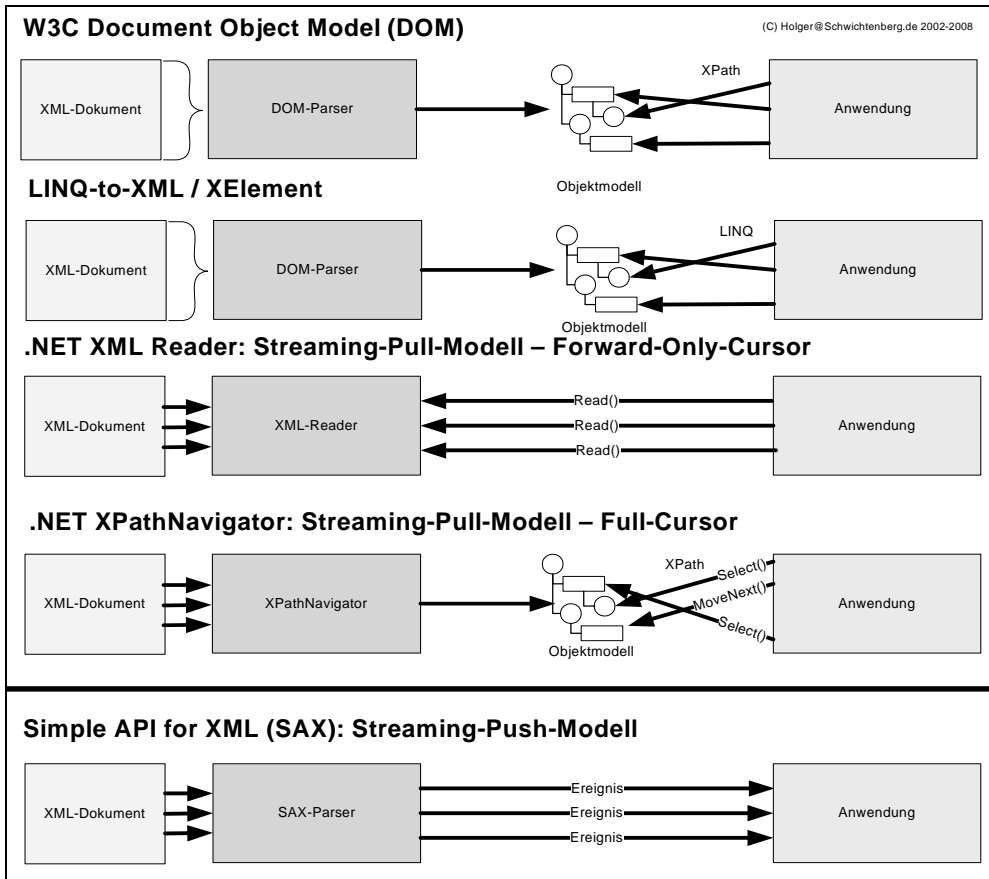
# Überblick über die XML-Zugriffsmodelle

Das .NET Framework 4.0 enthält vier Zugriffsmodelle für XML:

- XML-DOM mit den Klassen *XmlDocument*, *XmlElement*, *XmlAttribute* etc.
- LINQ to XML mit den Klassen *XDocument*, *XElement*, *XAttribute* etc.
- Klasse *XmlReader*
- XPath Data Model mit der Klasse *XPathNavigator*

Das XML-DOM ist ein sehr komfortables und in Hinblick auf den erzeugten XML-Code zuverlässiges Modell, jedoch nicht sehr effizient, wenn es nur darum geht, Daten aus einer XML-Datei zu lesen, da zu Beginn immer alle Daten in den Hauptspeicher geladen werden müssen, um dort ein Objektmodell aufzubauen (siehe Abbildung). Neu seit .NET 3.5 ist LINQ to XML, das intern auf dem XML-DOM basiert, aber dem Entwickler eine komfortablere Programmierschnittstelle mit Unterstützung der LINQ-Syntax zur Verfügung stellt.

Als echte Alternative zum DOM existiert für das reine Lesen von XML-Dokumenten das so genannte Streaming-Modell, bei dem nur jeweils ein aktueller Knoten im Hauptspeicher gehalten wird statt des ganzen Dokumentbaums. Streaming-Ansätze existieren in drei Ausprägungen: Push-Modell (alias Simple API for XML, SAX), Pull-Modell (*XmlReader*) und Cursor-Modell (*XPathNavigator*). Der Unterschied zwischen den drei Ansätzen liegt darin, dass SAX ein ereignisgetriebenes Modell (Callbacks) verwendet, während *XmlReader* und *XPathNavigator* sich die Daten holen (Pull-Modell). Der Unterschied zwischen den beiden Letztgenannten ist, dass der *XPathNavigator* auch Rücksprünge erlaubt, während der *XmlReader* ein reiner Vorwärts-Cursor ist.



**Abbildung 13.1** Architekturvergleich zwischen den XML-Zugriffsmodellen

#### HINWEIS

Das in der Java-Welt beliebte SAX wird auch in .NET 4.0 nicht von Microsoft unterstützt. Microsoft setzt auf die Pull-Modelle. SAX für .NET existiert aber als Open-Source-Projekt [SF01].

## XML-DOM (XmlDocument)

Das XML-Document Object Model (DOM) ermöglicht die komplette Repräsentation eines XML-Dokuments im Hauptspeicher durch ein Objektmodell. XML-DOM ist ein Standard des World Wide Web Consortium (W3C) [W3C03], der im .NET Framework mit Erweiterungen implementiert ist.

Die typische Arbeit mit dem XML-DOM besteht aus folgenden Schritten:

- Laden eines Dokuments in den Hauptspeicher, wobei das Dokument immer komplett geladen werden muss: Die Klasse `XmlDocument` stellt dafür die Methoden `Load()` und `LoadXml()` bereit.
- Gezielter Zugriff auf einzelne Knoten mit `SelectSingleNode()` oder auf Knotenmengen mit `SelectNodes()` unter Verwendung von XPath-Ausdrücken
- Iteration über die Knoten mit `XmlNode` und `XmlNodeList` (siehe Darstellung des Objektmodells)
- Speichern des Dokuments mit `Save()`

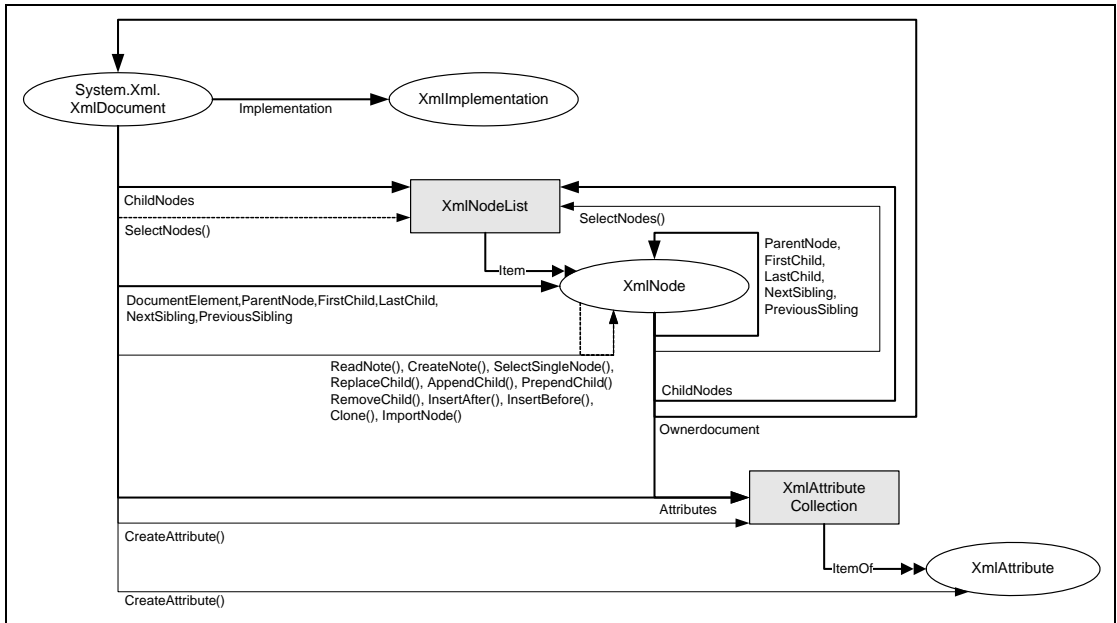


Abbildung 13.2 Objektmodell des XML-DOM in .NET

## Beispiel

Das Beispiel nutzt das nachstehend abgebildete Dokument und zeigt die Erhöhung der Platzanzahl im Flug Nummer 200 um zehn Sitzplätze. Das Dokument finden Sie in den Codebeispielen zu diesem Buch. Wie Sie das Dokument per Programmcode aus der mitgelieferten Datenbank *WorldWideWings* erstellen können, erfahren Sie später im Abschnitt zur `XmlWriter`-Klasse.

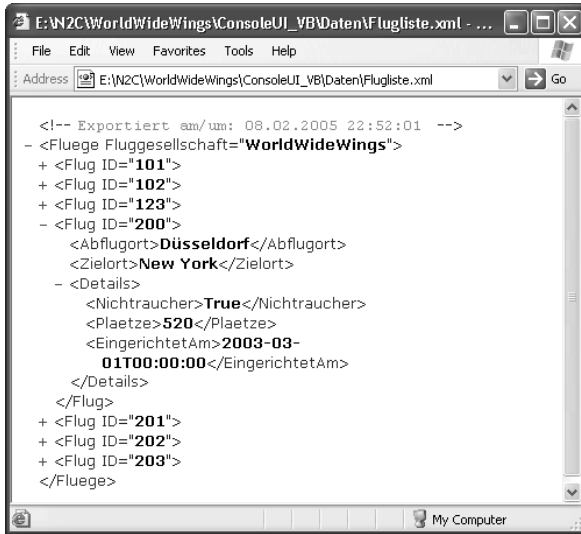


Abbildung 13.3 XML-Beispieldokument

```
Sub PlatzAnzahlErhoehen()
    Dim d As New XmlDocument()
    Dim e As XmlElement
    Const FlugNr As String = "200"
    ' --- Dokument laden
    d.Load(DATEI)
    ' --- Einzelnes Element auswählen
    e = d.SelectSingleNode("//*[@ID='" & FlugNr & "']/Details/Plaetze")
    Demo.Print("Alter Wert:" & e.FirstChild.Value)
    ' --- Counter erhöhen!
    e.FirstChild.Value += 10
    Demo.Print("Neuer Wert:" & e.FirstChild.Value)
    ' --- Dokument speichern
    d.Save(DATEI)
End Sub
```

Listing 13.1 Veränderung eines XML-Dokuments mit dem XML-DOM (in VB)

```
public void PlatzAnzahlErhoehen()
{
    XmlDocument d = new XmlDocument();
    XmlElement e = null;
    const string FlugNr = "200";
    // --- Dokument laden
    d.Load(DATEI);
    // --- Einzelnes Element auswählen
    e = d.SelectSingleNode("//*[@ID='" + FlugNr + "']/Details/Plaetze") as XmlElement;
    Demo.Print("Alter Wert:" + e.FirstChild.Value);
    // --- Counter erhöhen!
    e.FirstChild.Value = (Convert.ToInt64(e.FirstChild.Value) + 10).ToString();
    Demo.Print("Neuer Wert:" + e.FirstChild.Value);
    // --- Dokument speichern
    d.Save(DATEI);
}
```

Listing 13.2 Veränderung eines XML-Dokuments mit dem XML-DOM (in C#).

**HINWEIS** Bitte beachten Sie folgende Hinweise zur Arbeit mit dem XML-DOM-Objektmodell:

- Das Attribut `Value` liefert immer eine Zeichenkette, unabhängig davon, was im XML-Schema als Datentyp definiert ist. Wenn man C# oder in Visual Basic die Option `Strict` verwendet, ist im obigen Beispiel eine Typkonvertierung notwendig.
- Textinhalte sind im XML-DOM auch Knoten. Ein Element, das nicht leer ist, enthält den Wert nicht direkt, sondern einen Knoten vom Typ `Text` mit Namen `#Text`. Daher kann man in dem obigen Beispiel auf den Inhalt von `<Platze>` nicht direkt mit `.Value` zugreifen, sondern muss auf den Inhalt des ersten Unterknotens zugreifen mit `.FirstChild.Value`.
- Das Attribut `InnerText` liefert den Text eines Elements und aller Unterelemente. `InnerXml` liefert den Inhalt eines Elements als XML-Fragment.
- Das XML-DOM sorgt selbst dafür, dass beim Speichern von Sonderzeichen (z. B. `<`, `>`, `&`) diese entsprechend der XML-Notation codiert werden (z. B. `&lt;`, `&gt;`, `&amp;`).

## Unterstützung für XML-Namensräume

Das XML-DOM-Objektmodell bietet Unterstützung für XML-Namensräume durch die Klasse `XmlNamespaceManager`, mit der man die `NameTable` eines `XmlDocument`-Objekts befüllen kann. Anschließend kann man in XPath-Ausdrücken auf die im `XmlNamespaceManager` hinterlegten Kürzel zugreifen.

```
...
// --- Namensräume definieren
XmlNamespaceManager namespaces = new XmlNamespaceManager(doc.NameTable);
namespaces.AddNamespace("www", "http://www.IT-Visions.de/WWWings");
...
// XPath-Selektion mit Namensraum
XmlElement e = doc.SelectSingleNode("//www:Flug[1]/www:Details", namespaces) as XmlElement;
```

**Listing 13.3** Handhabung von Namensräumen (Fragmente eines Beispiels aus den begleitenden Codebeispielen in C#)

## Validierung von XML-Dokumenten

Zur Validierung eines XML-Dokuments gegen ein oder mehrere XSD-Schemata (Gültigkeitsprüfung) bietet die Klasse `XmlDocument` die Methode `Validate()`. Zuvor muss man die Schemas-Liste der Klasse `XmlDocument` befüllen. Die Methode `Validate()` signalisiert Validierungsfehler in Form des Ereignisses `ValidationEventHandler()`.

```
// === Validieren gegen XSD mit DOM
public static void xml_DOM_Validate()
{
    XmlDocument doc = new XmlDocument();
    doc.Load("Flugliste_mit_Namespace.xml");
    doc.Schemas.Add(null, @"Flugliste.xsd");
    System.Xml.Schema.ValidationEventHandler h = new System.Xml.Schema.ValidationEventHandler(Error);
    doc.Validate(h);
}

static void Error(object sender, System.Xml.Schema.ValidationEventArgs e)
{
    Demo.Print(e.Message);
}
```

**Listing 13.4** Validierung gegen ein XSD-Schema (Beispiel in C#)

## XML-Leser (XmlReader)

Die Klasse `System.Xml.XmlReader` zeichnet sich durch folgende Eigenschaften aus:

- Pull-Modell mit reiner Vorwärtsbewegung (forward-only)
- schneller Lesezugriff auf ein XML-Dokument
- kein Schreibzugriff
- keine Zwischenspeicherung der Inhalte
- linearer Datenstrom (Ausnahme: *Attribute*)
- Cursor steht immer auf einem Element des Gesamtdokuments
- Bewegung durch das Dokument mit `Read()`
- Attribute müssen extra berücksichtigt werden (`HasAttributes()`, `MoveNextAttribute()`)
- Elemente können übersprungen werden (z. B. `MoveToContent()`, `Skip()`)
- Unterstützt XML-Namensräume
- optionale Validierung gegen XSD oder DTD.

### HINWEIS

Während in .NET 1.0 die Klasse `XmlReader` nicht direkt, sondern nur über die abgeleiteten Klassen `XmlTextReader` und `XmlValidatingReader` genutzt werden konnte, empfiehlt Microsoft nun, die Klasse `XmlReader` durch Verwendung der neuen statischen Methode `Create()` zu instanziiieren. `Create()` bietet zahlreiche Optionen, die auch die Validierung gegen eine DTD oder ein XSD-Schema umfassen.

### Beispiel

Das besondere Verhalten der `XmlReader`-Klasse soll an einem Beispiel veranschaulicht werden, bei dem alle Elemente und alle Attribute durchlaufen und ausgegeben werden. Ausgegeben werden Name, Wert und Knotentyp. `Depth` gibt Aufschluss über die Ebene.

```
Sub LeseFlugdaten()
    Dim rs As XmlReaderSettings = New XmlReaderSettings
    rs.ConformanceLevel = ConformanceLevel.Fragment
    rs.IgnoreWhitespace = True
    rs.IgnoreComments = True
    Dim r As XmlReader = XmlReader.Create("../daten/Flugliste.xml", rs)
    ' --- Schleife zum Lesen
    Do While r.Read
        Demo.Print(StrDup(r.Depth, " ") & r.Name & ":" & r.Value & " (" & r.NodeType.ToString & ")")
        If r.HasAttributes Then
            Dim i As Integer
            For i = 0 To r.AttributeCount - 1
                r.MoveToAttribute(i)
                Demo.Print(StrDup(r.Depth, " ") & "- " & r.Name & "=" & r.Value & " (" & r.NodeType.ToString & ")")
            Next i
        End If
    Loop
    r.Close()
End Sub
```

**Listing 13.5** Beispiel zum Auslesen eines XML-Dokuments mit dem `XmlReader` (VB)



Das Beispiel liefert die nachfolgend abgedruckte Ausgabe. In dem Ausschnitt erkennt man bereits, dass das Element an der jeweils aktuellen Cursorposition entweder einen Namen oder einen Wert besitzt. Eine Ausnahme gilt nur für XML-Attribute.

```
Flug: (Element)
  - ID=101 (Attribute)
  Abflugort: (Element)
    :Düsseldorf (Text)
  Abflugort: (EndElement)
  Zielort: (Element)
    :München (Text)
  Zielort: (EndElement)
  Details: (Element)
    Nichtraucher: (Element)
      :True (Text)
    Nichtraucher: (EndElement)
  Plaetze: (Element)
    :200 (Text)
  Plaetze: (EndElement)
  EingerichtetAm: (Element)
    :2002-01-01T00:00:00 (Text)
  EingerichtetAm: (EndElement)
  Details: (EndElement)
```

**Listing 13.6** Ausschnitt aus der Ausgabe des obigen Beispiels

Die Klasse `XmlReader` bietet u. a. folgende weitere Möglichkeiten:

- Sie können das nächste Element gezielt mit einem bestimmten Namen anspringen:

```
reader.ReadStartElement("Details")
```

- Möchte man die Ausgabe der *Ende*-Tags verhindern, kann man prüfen:

```
If (Not r.NodeType = XmlNodeType.EndElement) Then ...
```

- Um einen Knoten komplett zu überspringen, verwendet man `Skip()`:

```
If r.Name = "Details" Then r.Skip() ' Knoten überspringen
```

- Ob ein Element überhaupt einen Wert hat, prüfen Sie mit `HasValue()`:

```
If Not r.HasValue Then ...
```

- Neu seit .NET 2.0 ist die Möglichkeit, beim Einlesen des Elementinhalts automatisch eine Konvertierung von einem XML-Typ in einen skalaren CLR-Typ durchzuführen:

```
r.ReadContentAsDateTime(), r.ReadContentAsInt(), r.ReadContentAsDouble() etc.
```

## Validierung

Die `XmlReader`-Klasse kann auch eine Validierung eines Dokuments gegen ein XML-Schema oder eine DTD vornehmen. Diese Funktion wurde in .NET 1.x durch die Klasse `XmlValidationReader` wahrgenommen und ist nunmehr eine Option in der `XmlReaderSettings`-Klasse. Die Klasse `XmlReader` erzeugt ein `ValidationEvent`, wenn ein Fehler gefunden wird. Die Validierung wird nach dem Aufruf der Ereignisbehandlungsroutine fortgeführt.

```

Sub LeseFlugdaten3()
    Dim rs As XmlReaderSettings = New XmlReaderSettings
    rs.ConformanceLevel = ConformanceLevel.Fragment
    rs.IgnoreWhitespace = True
    rs.IgnoreComments = True
    rs.ValidationType = ValidationType.Schema
    rs.Schemas.Add(Nothing, "..\..\daten\Flugliste1.xsd")
    Dim r As XmlReader = XmlReader.Create("..\..\daten\Flugliste.xml", rs)
    ' --- Ereignisbehandlung festlegen
    AddHandler rs.ValidationEventHandler, AddressOf ValidationsFehler
    Demo.Print("Validierung beginnt...")
    ' --- Daten einlesen und dabei validieren!
    While r.Read() : End While
    If Fehler Then
        Demo.Print("Validierung fehlgeschlagen!")
    Else
        Demo.Print("Validierung erfolgreich beendet!")
    End If
    r.Close()
End Sub

' === Validierungsfehler anzeigen
Public Sub ValidationsFehler(ByVal sender As Object, ByVal args As ValidationEventArgs)
    Fehler = True
    Demo.Print(("Fehler: " & args.Message))
End Sub

```

**Listing 13.7** Validierung eines XML-Dokuments gegen ein XML-Schema (VB)

**HINWEIS** Eine Validierung durch das alte XML-Data Reduced (XDR) wird von der Klasse `XmlReader` nicht mehr unterstützt. Hierzu müssen Sie die alte (weiterhin vorhandene) Klasse `XmlValidationReader` einsetzen.

## XML-Schreiber (`XmlWriter`)

Die Klasse `XmlWriter` ist das schreibende Pendant zum `XmlReader`. Der `XmlWriter` erlaubt nur das Vorwärts-Schreiben von Dokumenten und abstrahiert weit weniger von den XML-Syntaxregeln als das XML-DOM. Der Entwickler muss zwar keine spitzen Klammern schreiben, aber selbst dafür sorgen, dass es zu jedem geöffneten Element ein passendes schließendes Element gibt: Für jeden Aufruf von `WriteStartElement()` muss es an der richtigen Stelle ein `WriteEndElement()` geben.

Zeichenketten können einfach mit `WriteElementString()` geschrieben werden, das sowohl Start- als auch Endelement automatisch mit einfügt. Verfügbar seit .NET Framework 2.0 ist die Methode `WriteValue()`, die einen CLR-Typ automatisch in den passenden XML-Typ konvertiert. Dies ist z.B. für Datums- oder Zeitangaben wichtig. Manuell können Typen über die Klasse `System.Xml.XmlConvert` umgewandelt werden.

**HINWEIS** In .NET 1.x konnten keine Instanzen der Klasse `XmlWriter` erzeugt werden; dort war die Nutzung eines Streaming-Writers nur möglich über die von `XmlWriter` abgeleitete Klasse `XmlTextWriter`. In .NET 2.0/3.0/3.5/4.0 existiert `XmlTextWriter` weiterhin; Microsoft empfiehlt jedoch nun die Nutzung der Oberklasse `XmlWriter` über ihre neue statische Methode `Create()`, da hier mehr Einstellungsmöglichkeiten bestehen als bei der Klasse `XmlTextWriter`. `Create()` erlaubt die Angaben einer Instanz der neuen Klasse `XmlWriterSettings`. Hier kann beispielsweise festgelegt werden,

- ob Elemente eingerückt werden sollen (`Indent`),
- welche Einrückung verwendet werden soll (`IndentChars`),
- ob die XML-Startdeklaration automatisch eingefügt werden soll (`OmitXmlDeclaration`),

- ob der XmlWriter prüfen soll, ob ein wohlgeformtes Dokument entsteht (ConformanceLevel),
- ob der XmlWriter prüfen soll, dass nur erlaubte Zeichen eingefügt werden (CheckCharacters).

Die Ausgabe des XML-Schreibers kann in eine Datei, ein Stream-Objekt, ein TextWriter-Objekt oder ein StringBuilder-Objekt erfolgen.

```
Sub ExportiereFluege()
    Const CONNSTRING As String = "Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=WorldWideWings;Data Source=Server01\sqlexpress"
    Const SQL As String = "Select * from FL Fluege"
    Demo.PrintHeader("Export der Flüge (XmlWriter-Demo)")
    Dim xw As XmlWriter
    Demo.Print("Vorbereitungen...")
    ' --- Optionen für Dokument setzen
    Dim ws As XmlWriterSettings = New XmlWriterSettings()
    ws.Indent = True
    ws.IndentChars = (ControlChars.Tab)
    ws.OmitXmlDeclaration = True
    ' --- XMLWriter erzeugen
    xw = XmlWriter.Create("Flugliste.xml", ws)
    ' --- Dokument beginnen
    xw.WriteStartDocument()
    ' --- Kommentar einfügen
    xw.WriteComment("Exportiert am/um: " & DateTime.Now)
    ' --- Wurzelement
    xw.WriteStartElement("Fluege")
    xw.WriteAttributeString("Fluggesellschaft", "WorldWideWings")
    ' --- Datenbank öffnen
    Dim sqlConn As SqlConnection = New SqlConnection(CONNSTRING)
    sqlConn.Open()
    Dim sqlCommand As SqlCommand = sqlConn.CreateCommand
    sqlCommand.CommandText = SQL
    Dim dr As SqlDataReader = sqlCommand.ExecuteReader
    Demo.Print("Exportieren der Flüge...")
    While dr.Read
        xw.WriteStartElement("Flug")
        xw.WriteAttributeString("ID", dr("Fl_FlugNr").ToString())
        xw.WriteElementString("Abflugort", dr("Fl_Abflugort").ToString())
        xw.WriteElementString("Zielort", dr("Fl_Zielort").ToString())
        xw.WriteStartElement("Details")
        xw.WriteElementString("Nichtraucher", dr("Fl_Nichtraucherflug").ToString())
        xw.WriteElementString("Plaetze", dr("Fl_Plaetze").ToString())
        xw.WriteStartElement("EingerichtetAm")
        xw.WriteValue(dr("Fl_EingerichtetAm"))
        xw.WriteEndElement()
        xw.WriteEndElement()
        xw.WriteEndElement()
    End While
    ' --- Elemente schließen
    xw.WriteEndElement()
    xw.WriteEndDocument()
    ' --- Datei schließen
    xw.Close()
    ' --- Datenbank schließen
    dr.Close()
    sqlConn.Close()
    Demo.Print("Exportieren der Flüge beendet!")
End Sub
```

**Listing 13.8** Nutzung des XmlWriter (VB)

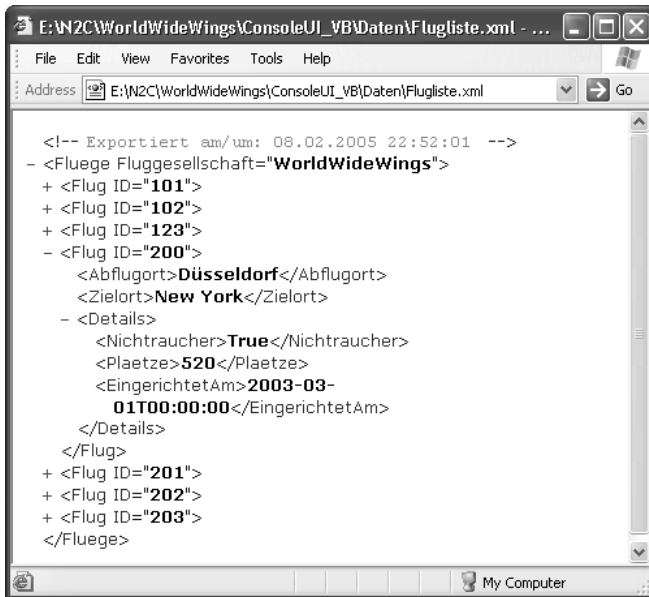


Abbildung 13.4 Ausgabe des XmlWriter-Beispiels

## XPathNavigator (XPath Data Model)

Die Klasse `XPathNavigator` stellt einen Lesezugriff auf XML-Dokumente bereit und bietet dennoch Selektion via XPath und beliebige Navigation ähnlich wie das XML DOM. Der `XPathNavigator` ist jedoch wesentlich schneller als das XML DOM.

Ein `XPathNavigator` kann wahlweise auf Basis einer Instanz der Klassen `XPathDocument` oder `XmlDocument` jeweils mit der Methode `CreateNavigator()` erzeugt werden. In beiden Fällen können Daten gelesen werden. Neu seit .NET 2.0 ist, dass im Fall der Verwendung des `XmlDocument` als Basis die XML-Daten auch verändert werden können.

Die Klasse `XPathNavigator` bietet folgenden Funktionsumfang:

- Cursor-Modell: `MoveToNext()`, `MoveToPrevious()`
- Enumeration: `For/Each`-Unterstützung (neu seit .NET 2.0)
- Ab- und Aufsteigen im XML-Baum: `MoveToFirstAttribute()`, `MoveToFirstChild()`, `MoveToParent()`, `MoveToRoot()`.
- Unterstützung für Selektion mit XPath: z. B. `Select("*/Flug")`. `XPathNodeIterator` repräsentiert die Menge gewählter Knoten (Ergebnis von `Select()`); nur Vorwärts-Bewegung ist möglich
- Methoden `InsertAfter()`, `InsertBefore()`, `SetValue()`, `DeleteSelf()`, `MoveToChild()` etc. zur Veränderung von XML-Dokumenten, sofern der Navigator auf einem `XmlDocument`-Objekt beruht

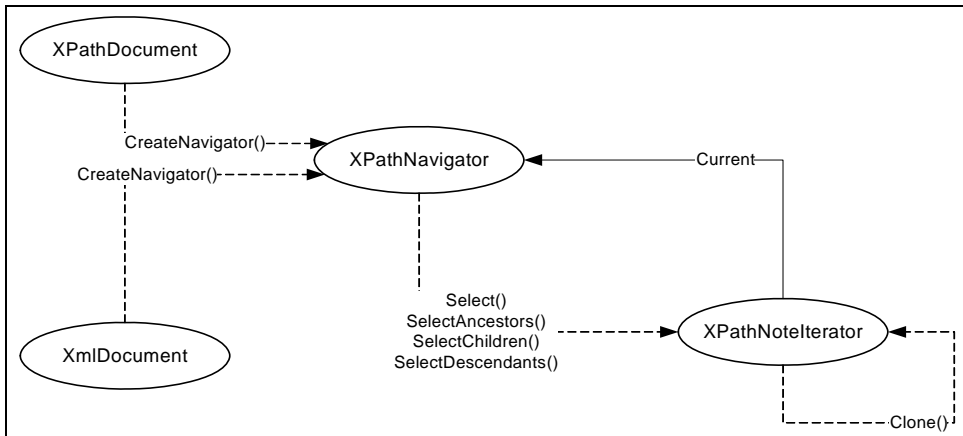


Abbildung 13.5 Objektmodell des XPathNavigator

## Beispiel

Das folgende VB.NET-Listing zeigt den Umgang mit dem XPathNavigator: Aus dem XPathDocument wird mithilfe der Methode CreateNavigator() das Navigationsobjekt gewonnen. Auf diesem wird die XPath-Anweisung `*//Flug` angewendet, die alle `<Flug>`-Knoten selektiert. Über den zurückgelieferten XPathNodeIterator kann mit `for each` iteriert werden. Für die Weiterverarbeitung ist zunächst wieder ein XPathNavigator-Objekt über die `Current`-Eigenschaft des XPathNodeIterator zu gewinnen. Etwas umständlich ist, dass `Select()` immer einen XPathNodeIterator liefert; wenn nur ein Element gefunden wurde, ist daher dennoch immer `MoveNext()` vor dem Zugriff auf `Current` notwendig. Ein XPathNavigator unterstützt auch Bewegungen wie `MoveToFirstAttribute()`, `MoveToFirstChild()`, `MoveToParent()` und `MoveToRoot()`.

**HINWEIS** In dem folgenden Beispiel wird außerdem davon ausgegangen, dass das Eingabedokument XML-Namensräume verwendet. In diesem Fall muss ein `XmlNamespaceManager` bei der Ausführung von `Select()` verwendet werden, weil sonst keine Elemente gefunden werden können.

```

Sub Abflughafen()
    Dim nr As String, wert As String
    Dim doc As XPathDocument = New XPathDocument(DATE1)
    Dim nav As XPathNavigator = doc.CreateNavigator()

    ' --- Pfad
    Const XPATH As String = "*/www:Flug"

    ' --- Namensräume definieren
    Dim namespaces As XmlNamespaceManager = New XmlNamespaceManager(nav.NameTable)
    namespaces.AddNamespace("www", "http://www.IT-Visions.de/WWWings")

    ' --- Selektion der URL-Elemente
    Dim iterator As XPathNodeIterator = nav.Select(XPATH, namespaces)
    out("Anzahl gefundener Knoten:" & iterator.Count)

    For Each n As XPathNavigator In iterator
        ' --- Auslesen des Attributs "ID" des Flugs
        nr = n.GetAttribute("ID", "")
        ' --- Navigation zum Unterelement "Abflugort"
    
```

```

Dim it2 As XPathNodeIterator = n.Select("www:Abflugort", namespaces)
it2.MoveNext()
' --- Auslesen des Inhalts von Count
wert = it2.Current.Value
' it2.Current.SetValue("bonn")
' it2.Current.Value = 0 nicht erlaubt!
out("Flug Nr. " & nr & " startet in " & wert)
Next
End Sub

```

**Listing 13.9** Nutzung der Klasse *XPathNavigator* (VB)

## LINQ to XML

LINQ to XML ist eine Programmierschnittstelle, die den Zugriff auf das XML-DOM (Klasse *XmlDocument*, *XmlElement*, *XmlAttribute*, ...) stark vereinfacht. LINQ to XML wurde mit dem .NET Framework 3.5 eingeführt und umfasst insbesondere folgende Funktionen:

- Abfrage von XML-Dokumenten/-Fragmenten mit der LINQ-Abfragesyntax (zu Language Integrated Query (LINQ) siehe gleichnamiges Kapitel in diesem Buch)
- Erstellen und Verändern von XML-Dokumenten/-Fragmenten mit einer neuen Klassenbibliothek im Namensraum *System.Xml.Linq* (Klassen *XDocument*, *XmlElement*, *XmlAttribute*, *XProcessingInstruction*, *XComment*, *XDeclaration*, usw.)

### HINWEIS

Voraussetzung für die Nutzung von LINQ to XML ist die Referenzierung der Assembly *System.Xml.Linq.dll* und des Namensraums *System.Xml.Linq*.

## Laden von XML

Die Klasse *XmlElement* ermöglicht das Laden von XML aus Dateien oder einem Reader (*System.Xml.XmlReader* oder *System.IO.TextReader*) mit der Methode *Load()* und die Aufnahme von XML aus einer Zeichenkette mit der Methode *Parse()*.

```

XmlElement x = System.Xml.Linq.XmlElement.Parse(
@"
<Flug ID='347'>
  <Abflugort>Madrid</Abflugort>
  <Zielort>Paris</Zielort>
  <FreiePlaetze>1</FreiePlaetze>
  <Details>
    <Nichtraucher>true</Nichtraucher>
    <Plaetze>250</Plaetze><EingerichtetAm/>
  </Details>
  <Passagiere>
    <Passagier>Müller</Passagier>
    <Passagier>Meier</Passagier>
    <Passagier>Schulze</Passagier>
  </Passagiere>
</Flug>");

```

**Listing 13.10** Zuweisen eines XML-Fragments in C# (seit Version 2008)

In Visual Basic (seit Version 2008) kann man die Zuweisung an XElement auch ohne Verpacken des XML in eine Zeichenkette erledigen, da Visual Basic (seit Version 2008) XML-Literale unterstützt (vgl. Kapitel zur Sprachsyntax).

```
Dim x As XElement = _
    <Flug ID="347"> _
        <Abflugort>Madrid</Abflugort>
        <Zielort>Paris</Zielort>
        <FreiePlaetze>1</FreiePlaetze>
        <Details>
            <Nichtraucher>True</Nichtraucher>
            <Plaetze>250</Plaetze>
            <EingerichtetAm/>
        </Details>
        <Passagiere>
            <Passagier>Müller</Passagier>
            <Passagier>Meier</Passagier>
            <Passagier>Schulze</Passagier>
        </Passagiere>
    </Flug>
```

**Listing 13.11** Zuweisen eines XML-Fragments in VB (seit Version 2008)

## Zugriff auf Elemente

Die Klasse XElement bietet zum direkten Zugriff auf bestimmte Knoten die Attribute Element, Elements, Attribute und Attributes. Die Namen im Singular verwendet man, wenn es nur einen Knoten gibt oder man den ersten Knoten ansprechen will. Diese Attribute liefern wieder eine Instanz von XElement. Die Attribute mit Namen im Plural liefern hingegen eine Liste von XElement-Objekten (IEnumerable<XElement>).

```
// Zugriff auf ein Attribut
Console.WriteLine(x.Attribute("ID").Value);
// Zugriff auf Elemente
Console.WriteLine(x.Element("Abflugort").Value); // Madrid
Console.WriteLine(x.Element("Zielort").Value); // Paris
Console.WriteLine(x.Element("Details").Element("Plaetze").Value); // 250
Console.WriteLine(x.Element("Passagiere").Element("Passagier").Value); // Müller
Console.WriteLine(x.Element("Passagiere").Elements("Passagier").ElementAt(1).Value); // Meier

// Schleife über Elemente
foreach (XElement p in x.Element("Passagiere").Elements("Passagier"))
    Console.WriteLine(p.Value);
```

**Listing 13.12** Direktzugriff auf Knoten in C#

---

**TIPP** Beim Aufruf der Methode ToString() auf einer Instanz von XElement liefert die Klasse das XML-Fragment, das in XElement gespeichert ist, als Zeichenkette.

---

In Visual Basic (seit Version 2008) geht auch dies wieder einfacher durch die XML-Literale.

---

**HINWEIS** Natürlich kann man in VB weiterhin alternativ auch die längere Syntax verwenden, z. B.

---

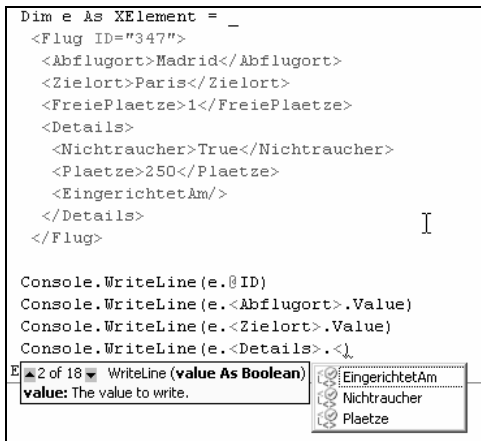
```
Console.WriteLine(x.Element("Details").Element("Plaetze").Value) ' 250
```

---

```
' Zugriff auf ein Attribut
Console.WriteLine(x.@ID)
' Zugriff auf Elemente
Console.WriteLine(x.<Abflugort>.Value) ' Madrid
Console.WriteLine(x.<Zielort>.Value) ' Paris
Console.WriteLine(x.<Details>.<Plaetze>.Value) ' 250
Console.WriteLine(x.<Passagiere>.<Passagier>(0).Value) ' = Müller!
Console.WriteLine(x.<Passagiere>.<Passagier>(1).Value) ' = Meier!
' Schleife über Elemente
For Each p As XElement In x.<Passagiere>.<Passagier>
    Console.WriteLine(p.Value)
Next
```

### Listing 13.13 Direktzugriff auf Knoten in VB

**TIPP** Wenn man in Visual Studio (seit Version 2008) ein Schema mit in das Projekt aufnimmt, erhält man auch IntelliSense-Eingabeunterstützung für XML-Literale.



**Abbildung 13.6** IntelliSense-Eingabeunterstützung für XML-Literale in Visual Basic (ab Version 2008)

## Abfrage von XML

Anstelle von XPath-Ausdrücken kann man zum Suchen und Filtern in LINQ to XML die LINQ-Syntax verwenden. Man kann alle Knotenmengen (Typ `IEnumerable<XElement>`) abfragen. Das Ergebnis ist im Normalfall wieder ein Objekt, das `IEnumerable<XElement>` implementiert. Wenn ein Operator verwendet wird, der die Menge auf ein Element beschränkt (z. B. `FirstOrDefault()`, `SingleOrDefault()`), dann ist das Ergebnis eine Instanz von `XElement`. Projektionen und Aggregationen sind möglich (siehe folgendes Listing).

```
// Natürlich kann man Dokumente laden...
XElement doc = XElement.Load("Daten/flugliste.xml");
// LINQ mit mehreren Ergebnissen
IEnumerable<XElement> query = from f in doc.Elements("Flug")
    where Convert.ToInt32(f.Element("FreiePlaetze").Value) > 100
    select f;
foreach (XElement result in query)
    Console.WriteLine(result.Attribute("ID").Value + "=" + result.Element("FreiePlaetze").Value);
```



```
// LINQ mit einem Ergebnisknoten
XElement ErsterFlugVonRom = (from f in doc.Elements("Flug")
                             where f.Element("Abflugort").Value == "Rom"
                             select f).FirstOrDefault();
Console.WriteLine(ErsterFlugVonRom.Attribute("ID").Value);
// Projektion
var query2 = from f in doc.Elements("Flug")
              where Convert.ToInt32(f.Element("FreiePlaetze").Value) > 100
              select new { ID = f.Attribute("ID").Value, Plaetze = f.Element("FreiePlaetze").Value };
foreach (var result in query2)
    Console.WriteLine(result.ID + "=" + result.Plaetze);
// Summe
long SummeFreiePlaetze = (from f in doc.Elements("Flug") select f).Sum(x =>
(long)x.Element("FreiePlaetze"));
Console.WriteLine(SummeFreiePlaetze);
```

**Listing 13.14** LINQ-Abfragen über XML-Dokumente (C#)

Auch hier ist Visual Basic wieder prägnanter mit XML-Literalen. Aus Platzgründen wird hier aber nur ein Beispiel abgedruckt.

```
' Natürlich kann man Dokumente laden...
Dim doc As XElement = XElement.Load("flugliste.xml")
' und LINQ einsetzen zum Abfragen
Dim query As XElement =
    From i In doc.<Fluege>.<Flug>
    Where i.<FreiePlaetze>(0).Value > 100
    Select i
For Each result As XElement In query.<Flug>
    Console.WriteLine(result.@ID.ToString() & "=" &
        result.<FreiePlaetze>(0).Value)
```

**Listing 13.15** LINQ-Abfrage über XML-Dokumente (VB)

## Verändern von XML-Inhalten

Die in die Klasse `XElement` geladenen Inhalte sind veränderbar und die Veränderungen können mit der Methode `Save()` in eine Datei gespeichert werden.

---

**ACHTUNG** Zu beachten ist, dass `Save()` immer nur den Inhalt des XML-Elements speichert, auf dem `Save()` ausgeführt wurde. Dies ermöglicht auf einfache Weise, Fragmente aus einem Dokument zu extrahieren. Wenn man aber – wie in dem folgenden Beispiel – Teile eines Dokuments auswählt und verändert, dann muss man am Ende darauf achten, dass man den Speichervorgang auf dem Gesamtdokument ausführt und nicht auf dem ausgewählten Teil. Sonst wird das Gesamtdokument durch den ausgewählten Teil ersetzt.

---

```
public static void XML_Fluege_Change()
{
    // Bestehende Flugliste laden
    XElement fluege = XElement.Load("_Daten/flugliste.xml");
    // LINQ mit einem Ergebnisknoten
    XElement flug = (from f in fluege.Elements("Flug")
                     where Convert.ToInt64(f.Attribute("ID").Value) == 101
                     select f).FirstOrDefault();
    if (flug == null) return; // Kein Flug gefunden
```

```
// Freie Plätze reduzieren
Console.WriteLine("Freie Plätze vorher: " + flug.Element("FreiePlaetze").Value);
flug.Element("FreiePlaetze").Value = (Convert.ToInt64(flug.Element("FreiePlaetze").Value) -
                                     1).ToString();
Console.WriteLine("Freie Plätze nachher: " + flug.Element("FreiePlaetze").Value);
// Liste komplett (!) speichern
Fluege.Save("_Daten/flugliste.xml");
Console.WriteLine("Gespeichert: " + Fluege.ToString());
}
}
```

**Listing 13.16** Veränderung eines XML-Dokuments mit dem LINQ to XML (in C#)

Visual Basic (seit Version 2008) ist hier wieder wesentlich prägnanter, u.a. deshalb, weil die Typkonvertierungen nicht notwendig sind. Dies gilt allerdings nur im Standardmodus. Wenn man den Visual Basic-Compiler mit Option Strict On in den strengeren Modus schaltet, dann sind auch hier bei dem Vergleich der Flugnummer und beim Rechnen mit der Platzanzahl Konvertierungen von String nach Int32 (alias long) notwendig.

```
Public Shared Sub XML Fluege Change()
    ' Bestehende Flugliste laden
    Dim Fluege As XElement = XElement.Load("_Daten/flugliste.xml")
    ' LINQ mit einem Ergebnisknoten
    Dim flug As XElement = (From f In Fluege.<Flug> _
                           Where f.①ID = 101 _
                           Select f).FirstOrDefault()
    If flug Is Nothing Then ' Kein Flug gefunden
        Return
    End If
    ' Freie Plätze reduzieren
    Console.WriteLine("Freie Plätze vorher: " & flug.<FreiePlaetze>.Value)
    flug.<FreiePlaetze>.Value -= 1
    Console.WriteLine("Freie Plätze nachher: " & flug.<FreiePlaetze>.Value)
    ' Liste komplett (!) speichern
    Fluege.Save("_Daten/flugliste.xml")
    Console.WriteLine("Gespeichert: " & Fluege.ToString())
End Sub
```

**Listing 13.17** Veränderung eines XML-Dokuments mit dem LINQ to XML (in VB)

## Verändern der XML-Struktur

Die Klasse XElement erlaubt auch das Verändern der Struktur. Add() legt ein neues Element an und fügt es der Elementmenge am Ende an. Unterelemente muss man nicht mit Add() hinzufügen, denn der Konstruktor der Klasse XElement bietet als zweiten Parameter einen Parameterarray, d.h. eine variable Menge von Knoten, die als Unterknoten zu dem zu erzeugenden Element angelegt werden sollen. Hier kann man weitere Elemente erzeugen, die wieder Unterelemente direkt angeben dürfen.

Das folgende Listing zeigt das Hinzufügen eines Elements mit Unterelementen.

**HINWEIS** Diese Syntax, die Microsoft *funktionale Konstruktion* nennt, ist sehr prägnant. Sie kann aber auch sehr schnell sehr unübersichtlich werden.

```
// Bestehende Flugliste laden
XElement fluege = XElement.Load("_Daten/flugliste.xml");
// Hinzufügen eines Flugelements
fluege.Add(new XElement("Flug",
    new XAttribute("ID",123),
    new XElement("Abflugort", "Berlin"),
    new XElement("Zielort", "Moskau"),
    new XElement("FreiePlaetze", 250),
    new XElement("Details",
        new XElement("Nichtraucher",true),
        new XElement("Plaetze",250)),
    new XElement("Passagiere")
));
// Liste speichern
fluege.Save("_Daten/flugliste.xml");
```

**Listing 13.18** Hinzufügen eines Elementknotens zu einem bestehenden XML-Dokument

**HINWEIS** Alternativ zum Anfügen am Ende mit `Add()` bietet `XElement` auch eine gezielte Einfügemethode wie `AddAfterSelf()`, `AddBeforeSelf()` und `AddFirst()`. Für das Entfernen gibt es `Remove()`, `RemoveAll()` und `RemoveNodes()`.

## Vergleich der Zugriffsformen

Die beiden folgenden Tabellen liefern einen abschließenden Vergleich zwischen den vier in .NET verfügbaren Zugriffsformen auf XML-Dokumente.

	XmlDocument	LINQ to XML	XPathNavigator	XmlReader	XmlWriter
Abstraktionsniveau	Knoten	Knoten	Knoten	Tags	Tags
W3C-Standard	Ja (W3C DOM)	Nein	Nein	Nein	Nein
Selektion	XPath	LINQ	XPath	Nicht verfügbar	Nicht verfügbar
Daten lesen	Ja	Ja	Ja	Ja	Nein
Daten und Struktur ändern	Ja	Ja	Ja, wenn auf Basis von DOM erstellt (seit .NET 2.0)	Nein	Nur komplettes Neuschreiben des Dokuments
Beliebige Navigation	Ja	Ja	Ja	Nein (Nur Vorwärts)	Nein
Namensräume	Ja	Ja	Ja	Ja	Ja
Validierung	Möglich	Möglich	Nein (nur indirekt über XmlDocument)	Optional	Nein
Geschwindigkeit beim Laden	Sehr Langsam	Sehr Langsam	Langsam	Nicht verfügbar	Entfällt
Geschwindigkeit beim Durchlauf	Langsam	Langsam	Schnell	Mittel	Entfällt
Geschwindigkeit beim Schreiben	Langsam	Langsam	Langsam	Entfällt	Schnell

**Tabelle 13.1** Vergleich der verschiedenen Optionen zum Lesen von XML-Dokumenten, insbesondere in Hinblick auf die Geschwindigkeit

	Vorbereitungszeit	Dokument durchlaufen
XmlDocument (XML-DOM)	9699 ms	6268 ms
XmlReader	0 ms	2953 ms
XPathNavigator auf Basis eines XPathDocument-Objekts	5503 ms	284 ms
XPathNavigator auf Basis eines XmlDocument-Objekts	9010 ms	807 ms

**Tabelle 13.2** Leistungsvergleich an einem Beispiel (Dokumentendatei 6,7 MB)

## Ableiten eines Schemas aus XML-Dokumenten

Visual Studio bietet bereits seit der 2002er-Version die Möglichkeit, ein XML-Schema für ein gegebenes XML-Dokument zu erzeugen (Menü *XML/Create Schema*). Die gleiche Möglichkeit besteht auch mithilfe des Werkzeugs *xsd.exe* aus dem .NET SDK.

Diese Funktion existiert ebenfalls in der .NET-Klassenbibliothek durch die Klasse `System.Xml.Schema.XmlSchemaInference`. Eine Instanz von `XmlSchemaInference` ist dabei noch flexibler als die Funktion von Visual Studio und *xsd.exe*, weil mehrere XML-Dokumente als Eingabe berücksichtigt werden können. Je mehr verschiedene Eingabedokumente verwendet werden, desto besser ist die Ableitung des Schemas. Die Dokumente sind in Form von `XmlReader`-Objekten an die Methode `InferSchema()` zu übergeben. Die Ausgabe kann aus mehreren Schemata bestehen.

```
' === Ableiten eines Schemas für Flugliste.xml
Sub SchemaFuerFluegeXMLErzeugen()
    Dim reader1 As XmlReader = XmlReader.Create("../..\daten\Flugliste.xml")
    Dim reader2 As XmlReader = XmlReader.Create("../..\daten\Flugliste2.xml")
    Dim schemaSet As XmlSchemaSet = New XmlSchemaSet()
    Dim inference As XmlSchemaInference = New XmlSchemaInference()
    schemaSet = inference.InferSchema(reader1)
    schemaSet = inference.InferSchema(reader2)
    Demo.Print("Ausgabe der generierten Schemata...")
    Dim i As Integer
    For Each schema As XmlSchema In schemaSet.Schemas
        i += 1
        Dim Dateiname As String = "../..\daten\Flugliste" & i & ".xsd"
        Dim fs As New System.IO.FileStream(Dateiname, IO.FileMode.Create)
        Console.WriteLine("Erzeuge " & Dateiname & "...")
        schema.Write(fs)
        fs.Close()
    Next
End Sub
```

**Listing 13.19** Beispiel für das Ableiten eines Schemas aus einer XML-Datei (VB)

# XML Style Sheet Transformations (XSLT)

In .NET 2.0 hatte Microsoft den in der Klassenbibliothek enthaltenen XSLT-Prozessor hinsichtlich der Performanz stark überarbeitet. Die neue Klasse wird unter dem Namen `System.Xml.Xsl.XslCompiledTransform` angeboten. Die alte Klasse `System.Xml.Xsl.XslTransform` ist aus Kompatibilitätsgründen weiterhin vorhanden. Unterstützt wird der XSLT-Standard Version 1.0.

**WICHTIG** Die Klasse `XslCompiledTransform` ist nicht in der `System.Xml.dll`, sondern in der `System.Data.SqlXml.dll` implementiert.

Die Klasse `XslCompiledTransform` bietet eine `Load()`-Methode zum Laden eines XSLT-Stylesheets und eine Methode `Transform()` zur Umwandlung. `Transform()` unterstützt verschiedene Eingabearten (z.B. Pfad zu einem Dokument, `XmlReader`, `XPathDocument`, `XPathNavigator`) und verschiedene Ausgabearten (Pfad, `XmlWriter`, `Stream`, `TextWriter`).

## Beispiel

Das folgende Listing zeigt die Transformation einer XML-Datei in eine HTML-Datei mithilfe einer XSLT-Datei.

```
Sub Flugliste_in_HTML_umwandeln()  
    Const EINGABEDATEI = "Flugliste.xml"  
    Const XSLTDATEI = "Flugliste.xsl"  
    Const AUSGABEDATEI = "Flugliste.htm"  
    Demo.Print("Transformation der Datei : " & EINGABEDATEI)  
    ' --- Klassen instanziiieren  
    Dim xslt As New System.Xml.Xsl.XslCompiledTransform()  
    ' --- Stylesheet laden  
    xslt.Load(XSLTDATEI)  
    ' --- Dokument laden, transformieren und speichern  
    xslt.Transform(EINGABEDATEI, AUSGABEDATEI)  
    Demo.Print("Transformation in Datei erfolgreich: " & AUSGABEDATEI)  
End Sub
```

**Listing 13.20** Umwandeln einer XML-Datei in eine HTML-Datei mithilfe von XSLT (VB)

Im zweiten Beispiel erfolgt eine Umwandlung von einem XML-Dokument in ein anderes. Dabei erwartet das XSLT-Stylesheet einen Parameter *Datum*, der das aktuelle Datum (und optional die Uhrzeit) enthält. Diese Information gibt das Stylesheet in einen Kommentar in das Ausgabedokument. Der Parameter ist in einer Objektmenge vom Typ `XsltArgumentList` zu übergeben. Diese Objektmenge ist als Parameter bei `Transform()` zu verwenden. Wenn dieser Parameter verwendet wird, kann man als Ein- und Ausgabeoption nicht mehr den Dateipfad verwenden. Daher wird in dem folgenden Listing die Eingabedatei durch ein `XPathDocument` eingelesen und die Ausgabedatei durch einen `XmlWriter` erzeugt.

```

' === XSLT Transformation mit Parameterübergabe (von Navigator zu
Public Sub Flugliste_in_XML_umwandeln()
    Const EINGABEDATEI As String = "Flugliste.xml"
    Const XSLTDATEI As String = "Flugtransformation.xslt"
    Const AUSGABEDATEI As String = "Flugliste2.xml"
    Demo.Print ("Transformation der Datei... " & EINGABEDATEI & " mit " & XSLTDATEI)
    ' --- Eingabedatei
    Dim doc As New XPathDocument(EINGABEDATEI)
    ' --- Parameter
    Dim argList As XsltArgumentList = New XsltArgumentList()
    argList.AddParam("Datum", "", DateTime.Now.ToString())
    ' --- Writer für Ausgabe
    Dim settings As New XmlWriterSettings
    settings.Indent = True
    Dim writer As System.Xml.XmlWriter = System.Xml.XmlWriter.Create(AUSGABEDATEI, settings)
    ' --- XSLT-Prozessor instanziiieren
    Dim xslt As New System.Xml.Xsl.XslCompiledTransform()
    ' --- Stylesheet laden
    xslt.Load(XSLTDATEI)
    ' --- Dokument laden, transformieren und speichern
    xslt.Transform(doc, argList, writer)
    ' --- Writer schließen
    writer.Close()
    Demo.Print ("Transformation in Datei erfolgreich: " & AUSGABEDATEI)
End Sub

```

**Listing 13.21** Umwandeln einer XML-Datei in eine andere XML-Datei mithilfe von XSLT (VB)