Wohin mit dem Kleinkram? Anwendungsdaten unter .NET speichern und verwalten

Fast jede Anwendung speichert Daten zur Konfiguration und für diverse Einstellungen. In der Vor-.NET-Ära kamen hier meist die Registry oder INI-Dateien zum Einsatz. Mit XML-Konfigurationsdateien, dem Isolated-Storage-Konzept und der Möglichkeit der Datenspeicherung im Anwendungsdatenverzeichnis bietet .NET darüber hinausgehende Möglichkeiten, die eine größere Flexibilität und mehr Sicherheit gewährleisten. Der Artikel erleichtert die Auswahl einer Speicherungsmethode, indem er die jeweiligen Vor- und Nachteile erläutert.

___ Es gibt viele Arten von Anwendungsdaten, die in der einen oder anderen Form gespeichert werden müssen:

- GUI-Einstellungen, zum Beispiel Fenstergröße und Position,
- die zuletzt bearbeiteten Dokumente des Benutzers,
- Benutzerdaten und Berechtigungen,
- Konfigurationseinstellungen, zum Beispiel für Datenbankverbindungen.

Ein Hauptgrund für die gesonderte Speicherung von Einstellungen ist die bessere Administrierbarkeit der Anwendung. Anstatt beispielsweise den Namen des Servers, auf dem die Datenbank liegt, fest im Code zu hinterlegen, ermittelt man solche Informationen besser zur Laufzeit, um etwa bei einem Wechsel des Datenbank-Servers nicht die ganze Anwendung neu kompilieren zu müssen.

Bisher wurden solche Informationen meist in der Registry oder in einer INI-Datei gespeichert. Diese Medien haben jedoch einige Nachteile:

- INI-Dateien können keine hierarchischen Daten darstellen.
- INI-Dateien können nur einfache Name-Wert-Paare enthalten.
- Für das manuelle Ändern von Anwendungsdaten in der Registry muss der exakte Pfad bekannt sein.
- Das Übertragen von Registry-Einträgen auf einen anderen Rechner ist nicht ohne Weiteres möglich.

Für das Speichern von benutzerdefinierten Daten ist die Registry nach wie vor aktuell. Hier spielen vor allem Sicherheitsaspekte eine große Rolle.

Ein weiterer Punkt betrifft die Mobilität. Speichert man benutzerspezifische Daten in einer INI-Datei irgendwo auf dem Rechner, so ist es nicht ohne weiteres möglich, auf diese Einstellungen zuzugreifen, wenn man die Anwendung von einem anderen Rechner aus startet.

Das .NET Framework unterstützt die folgenden Konzepte, die sich im Wesentlichen durch die Aspekte Sicherheit, Administration, Deployment und Mobilität voneinander unterscheiden:

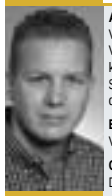
- XML-Konfigurationsdateien,
- Anwendungsdatenverzeichnisse,
- Registry,
- Isolated Storage.

dotnetpro beleuchtet diese Konzepte und ihre jeweiligen Vor- und Nachteile näher.

XML-Konfigurationsdateien

Für die Speicherung von Konfigurationseinstellungen der Anwendung sieht .NET XML-Konfigurationsdateien vor. Dies sind einfache XML-Dateien, die einer bestimmten Struktur folgen und den gleichen Dateinamen haben wie die Anwendung plus der Endung *.config*. Die Hauptaufgabe dieser Dateien ist die Konfiguration der Anwendung. Da nur lesend auf diese Dateien zugegriffen werden kann, eignen sie sich nicht zum Speichern von Anwendungsdaten. Die meisten Einstellungen, die sich zur Laufzeit ändern können und gespeichert werden müssen, gehören aber ohnehin nicht in eine Konfigurationsdatei, da sie oft benutzerbezogen beziehungsweise sicherheitsrelevant sind.

SUMMARY



Auf einen Blick
Von den Anforderungen an Sicherheit, Administration und Mobilität hängt das Verfahren ab, wie anwendungsbezogene Daten gespeichert werden. Der Artikel nennt Auswahlkriterien für die Verwendung fünf verschiedener Speicherungsmethoden: XML-Konfigurationsdateien, Registry, Anwendungsdatenverzeichnis, Isolated Storage und Datenbank.

Eingesetzte Anwendungen
Visual Studio .NET

CD-Code
DrillDown01

Autor
Jörg Neumann ist Programmierer bei der Firma KEEP IT SIMPLE GmbH in Hamburg. Für Fragen und Anregungen erreichen Sie ihn unter Joerg.Neumann@KEEPITSIMPLE.de.

Windows sieht für die Verwaltung von Anwendungsdaten zwei Konzepte vor: Zum einen das Sichern in der Registry und zum anderen das Speichern von Dateien im Anwendungsdatenverzeichnis. Für beide Konzepte bietet das .NET Framework Unterstützung, wie die folgenden Abschnitte zeigen.

Speichern von Einstellungen im Anwendungsdatenverzeichnis

Das Anwendungsdatenverzeichnis ist direkt mit dem Profil des angemeldeten Benutzers verknüpft und befindet sich bei Windows 2000 und XP beispielsweise unter

```
C:\Dokumente und Einstellungen\Benutzername\Anwendungsdaten
```

Benutzername steht hier stellvertretend für den Namen des angemeldeten Benutzers. Unterhalb von *Anwendungsdaten* werden die Einstellungen in Unterverzeichnissen nach diesem Schema gespeichert:

```
Firmenname\Anwendungsname\Version
```

Der exakte Pfad für die Anwendung kann zur Laufzeit über die Eigenschaft *UserAppDataPath* des *Application*-Objekts ermittelt werden. Dieser Pfad wird mithilfe der Assembly-Attribute *AssemblyCompany*, *AssemblyProduct* und *AssemblyVersion* sowie dem Namen des angemeldeten Benutzer generiert.

Die Attribute können in der Datei *AssemblyInfo.cs* editiert werden, die automatisch von Visual Studio beim Anlegen eines neuen Projekts erstellt wird. Verfügt man nicht über diese Datei, kann man die Attributdeklarationen auch direkt im Code vornehmen.

```
[assembly: AssemblyCompany("MyCompany")]
[assembly: AssemblyProduct("MyProduct")]
[assembly: AssemblyVersion("1.0.*")]
```

Weist man den Attributen keinen Wert zu, wird für *AssemblyCompany* und *AssemblyProduct* jeweils der Name der Assembly verwendet. Auf Einstellungen, die im genannten Pfad gespeichert werden, kann ein Benutzer auch von einem anderen Computer aus zugreifen, wenn er sich im Netz mit seinem Namen anmeldet. Das ist möglich, weil alle Daten serverseitig im Profil des Anwenders gespeichert werden. Das als Roaming bezeichnete Verfahren muss für jedes Profil auf dem Server explizit eingestellt werden.

Dies ist zwar eine recht praktische Angelegenheit, kann aber zu Performance-Einbußen und verstärktem Netzwerk-Traffic führen, wenn größere Binärdateien im Verzeichnis abgelegt werden. Möchte man das Roaming der Einstellungen vermeiden, sollte man die lokale Variante des Anwendungsdatenverzeichnisses verwenden, welche unter Windows 2000 und Windows XP unter folgendem Pfad zu finden ist:

```
C:\Dokumente und Einstellungen\Benutzername\
Lokale Einstellungen\Anwendungsdaten
```

Zur Laufzeit kann man den Pfad über die Eigenschaft *LocalUserAppDataPath* des *Application*-Objekts ermitteln. Die Bildung erfolgt analog zu den oben genannten Attributen.

Auch für das Speichern von benutzerübergreifenden Daten gibt es ein eigenes Verzeichnis. Unter Windows 2000/XP ist das:

```
C:\Dokumente und Einstellungen\All Users\Anwendungsdaten
```

Wie bei den oben genannten Varianten werden auch hier wieder die entsprechenden Assembly-Attribute an den Pfad angehängt. Zur Laufzeit kann das Verzeichnis über die *CommonAppDataPath*-Eigenschaft des *Application*-Objekts ermittelt werden.

Möchte man die Einstellungen in einer anderen Struktur als der oben beschriebenen speichern, so kann man das entsprechende Anwendungsdatenverzeichnis für *Roaming-Benutzer*, *Nicht-Roaming-Benutzer* und *Alle Benutzer* über die *GetSpecialFolder*-Methode der *Environment*-Klasse abfragen.

```
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)
Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData)
Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData)
```

Diese liefern den Root-Pfad des Anwendungsdatenverzeichnisses (ohne *Firmenname*/*Produktname*/*Version*) zurück.

Speichern von Einstellungen in der Registry

Das .NET Framework bietet neben seinen allgemeinen Registry-Klassen spezielle Klassen für die Verwaltung von Anwendungseinstellungen, die über die *Application*-Klasse zugänglich sind.

- *UserAppDataRegistry* bietet Zugriff auf einen benutzerspezifischen Registry-Schlüssel.
- *CommonAppDataRegistry* bietet Zugriff auf einen benutzerübergreifenden Registry-Schlüssel.

Die Bildungsregeln für die Registry-Pfade, in denen die Einstellungen gespeichert werden, sind analog zu denen des Anwendungsdatenverzeichnisses, also *Firmenname\Anwendungsname\Version*. Ein Beispiel für einen benutzerdefinierten Schlüssel:

```
HKEY_CURRENT_USER\Software\MyCompany\MyProduct\1.0.917.27777
```

Ein Beispiel für einen benutzerübergreifenden Schlüssel:

```
HKEY_LOCAL_MACHINE\SOFTWARE\MyCompany\MyProduct\1.0.917.27777
```

Die oben genannten Methoden liefern ein *RegistryKey*-Objekt zurück, mit dem man auf Unterschlüssel und Werte zugreifen kann. Das folgende Beispiel legt die Unterschlüssel *MySubKey1* und *MySubKey2* an und fügt eine Zeichenfolge mit dem Namen *MySetting* hinzu:

```
Application.UserAppDataRegistry.CreateSubKey(@"MySubKey1\MySubKey2")
.SetValue("MySetting","MyValue");
```

Folgender Code liest den Wert wieder aus:

```
string mySetting = Application.UserAppDataRegistry.OpenSubKey(@
"MySubKey1\MySubKey2").GetValue("MySetting").ToString();
```

Möchte man nicht die vorgegebene Struktur aus *Firmenname/Anwendungsname/Version* verwenden, weil man zum Beispiel die Einstellungen nicht auf Versionsebene verwalten will, so greift man direkt über die Registry-Klassen zu:

```
Microsoft.Win32.Registry.CurrentUser.CreateSubKey(@"Software\
MyCompany\MyProduct\MySubKey1\MySubKey2").SetValue("MySetting",
"MyValue");string mySetting = Microsoft.Win32.Registry.CurrentUser
.OpenSubKey(@"Software\MyCompany\MyProduct\MySubKey1\MySubKey2")
.GetValue("MySetting").ToString();
```

Eine Beschreibung der Klassen finden Sie unter [1].

Speichern von Einstellungen im Isolated Storage

Den Isolated Storage kann man sich als eine Art virtuelles Dateisystem vorstellen, das auf eine Kombination aus Assembly, Benutzer und Domäne beschränkt ist und keinen direkten Zugriff auf Daten anderer Assemblies zulässt – eine Art *Sandbox* also. Es ermöglicht die Speicherung von Einstellungen und Temporärdaten für Code-Komponenten, die nur eingeschränkt vertrauenswürdig sind (wie etwa Code aus dem Internet) und keine Rechte für den Zugriff auf Dateisystem oder Registry haben.

Der Isolated Storage wird vom System verwaltet und kann nur über die entsprechenden Klassen im .NET Framework angesprochen werden. Dies hat unter anderem den Vorteil, dass beim Zugriff nicht explizit ein Dateipfad angeben werden muss. Der Isolated Storage unterstüzt auch das Roaming, was dem Benutzer ermöglicht, auf seine Einstellungen von anderen Computern aus im Netz zuzugreifen.

Die Nutzung des Isolated Storage ist vor allem in folgenden Szenarien sinnvoll:

- Zum Speichern von Einstellungen von Code-Komponenten, die aus dem Internet heruntergeladen werden und keine Schreibrechte für das Dateisystem besitzen.
- Zum Speichern von Daten in Webanwendungen.
- Zum Speichern von Daten in Komponenten, die von mehreren Anwendungen genutzt werden.

Die einzige Voraussetzung, die eine Code-Komponente erfüllen muss, um den Isolated Storage nutzen zu können, ist das Recht *IsolatedStorageFilePermission*. Dieses Recht ist jedoch standardmäßig den meisten Codegruppen zugewiesen. Auch die maximale Datenmenge, die von einer Assembly im Isolated Storage abgelegt werden kann, hängt von den Rechten der entsprechenden Codegruppe ab. Die Daten werden im Isolated Storage in so genannten Stores verwaltet, die entweder auf Assembly/Anwender oder auf Assembly/Anwender/Domain beschränkt sind.

Zur Verwaltung des Isolated Storage stellt Microsoft ein Tool namens *StoreAdm.exe* bereit, welches im .NET SDK enthalten ist. Nähere Informationen finden Sie unter [2].

Die Zugriffsklassen für den Isolated Storage befinden sich im Namespace *System.IO.IsolatedStorage*. Im folgenden Beispiel wird eine Datei in einem Store erstellt, der in Bezug auf Assembly und Benutzer isoliert ist.

```
IsolatedStorageFile isoStore =  IsolatedStorageFile.GetStore(
IsolatedStorageScope.User | IsolatedStorageScope.Assembly, null, null);
```

Anschließend wird ein *StreamWriter* erstellt, der im Konstruktor einen neuen *IsolatedStorageFileStream* mitbekommt. Diesem übergibt man Dateiname, File Mode und das soeben erstellte *IsolatedStorageFile*-Objekt.

```
StreamWriter writer = new StreamWriter(new
IsolatedStorageFileStream("MySettings.txt",
FileMode.CreateNew,isoStore));
```

Jetzt können die Daten in den Stream geschrieben werden.

```
writer.WriteLine("Mein Inhalt");
writer.Close();
```

Bevor man eine neue Datei im Store erstellt, sollte man sicherheitshalber prüfen, ob eine Datei mit gleichem Namen bereits existiert. Diese Prüfung übernimmt die Funktion *FileExistsInStore*:

```
private bool FileExistsInStore(IsolatedStorageFile isoStore, string
fileName)
{
   bool bReturnValue = false;

   string[] fileNames = isoStore.GetFileNames(fileName);
   foreach (string file in fileNames)
   {
      if(file.ToLower() == fileName.ToLower())
      {
         bReturnValue = true;
      }
   }

   return bReturnValue;
}
```

Das Auslesen einer Datei funktioniert ähnlich wie das Erstellen. Zunächst öffnet man folgenden Store:

```
IsolatedStorageFile isoStore =  IsolatedStorageFile.GetStore(
IsolatedStorageScope.User | IsolatedStorageScope.Assembly, null, null);
```

Dann erstellt man einen *StreamReader* und weist ihm einen neuen *IsolatedStorageFileStream* zu.

```
StreamReader reader = new StreamReader(new
IsolatedStorageFileStream("MySettings.txt", FileMode.Open,isoStore));
```

Damit kann man die Datei auslesen.

```
String sb = reader.ReadLine();
Console.WriteLine(sb.ToString());
reader.Close();
```

Das Löschen einer Datei wird über die *Remove*-Methode des *IsolatedStorageFile*-Objekts bewerkstelligt.

```
isoStore.Remove();
```

Um alle Dateien und Verzeichnisse im aktuellen Gültigkeitsbereich zu löschen, ruft man die *Remove*-Methode des *IsolatedStorageFile*-Objekts auf:

```
IsolatedStorageFile.Remove();
```

Die Beispielanwendung *IsolatedStorageDemo* (siehe Abbildung 1) demonstriert diese Verfahren. Der Isolated Storage bietet

Abbildung 1 | Zugriff auf den Isolated Storage.

jedoch noch wesentlich mehr Funktionalität. Einen Überblick finden Sie unter [3].

Tabelle 1 | Vor- und Nachteile der einzelnen Speicherorte.

Typ	Schreibender Zugriff	Leichte Administration	Roaming	Geschwindigkeit	Bindung über IDE	Sicherheit
Konfigurationsdatei		X		+	X	-
Registry	X	X	X	+		+
Anwendungsdatenverzeichnis	X		X	-[2]		+
Isolated Storage	X		X	-[2]		+
Datenbank	X		X[1]	-[2]		+

1 Wenn Datenbank auf einem Server liegt.
2 Wenn Roaming verwendet wird.

Entscheidungshilfen

Nachdem nun alle wichtigen Speicherorte für Anwendungseinstellungen vorgestellt wurden, stellt sich die Frage, welcher am besten für die jeweilige Anwendung geeignet ist. Dies hängt vor allem von den Anforderungen an Geschwindigkeit, Sicherheit und Mobilität ab.

Grundsätzlich gehören alle Einstellungen, die sich auf die Konfiguration der Anwendung beziehen und nicht zur Laufzeit gespeichert werden müssen, in die Konfigurationsdatei der Anwendung. Benutzerbezogene und sicherheitsrelevante Daten sollte man dort jedoch nicht speichern. Für diese Fälle bietet sich eher die Registry an. Sollen die Daten jedoch leicht auf ein anderes System übertragbar sein, speichert man die Daten am sinnvollsten im Anwendungsdatenverzeichnis. Möchte man alle Anwendungsdaten an einer zentralen Stelle speichern, ist hingegen eine Datenbank als Speichermedium empfehlenswert.

Bei einer zentralen Speicherung sollte man jedoch immer die Abhängigkeit vom Netzwerk im Hinterkopf behalten. Dies spielt nicht nur bei einem Netzausfall eine Rolle, sondern macht die Anwendung auch für den Offline-Betrieb – zum Beispiel auf einem Notebook – ungeeignet.

Komponenten und Anwendungen, die von einer potenziell unsicheren Quelle stammen, sollten ihre Einstellungen immer im Isolated Storage speichern. Zur Erleichterung der Entscheidungsfindung dient Tabelle 1. Dort sind alle Speicherorte mit ihren Vor- und Nachteilen aufgeführt.

Zugriff auf Einstellungen aus der Anwendung

Für das Bereithalten der Einstellungen definiert man am besten eine Klasse, die über Eigenschaften die Daten zugänglich macht. Der Vorteil einer Klasse – im Gegensatz zu Variablen oder Strukturen – ist vor allem die Möglichkeit, in den Eigenschaftsdeklarationen Validierungen durchführen zu können. Das Laden und Speichern der Daten sollte man in einer separaten Klasse erledigen. Durch diese Kapselung wird eine spätere Änderung des Speichermediums erheblich vereinfacht.

Die Beispielklasse *AppSettingsManager* übernimmt diese Aufgabe. Sie bietet eine *LoadSettings*- und eine *SaveSettings*-Methode für das Laden und Speichern der Einstellungen. Außerdem hostet sie die Einstellungsklasse (*AppSettings*) in Form einer statischen Variablen, die über die Eigenschaft *Settings* zugänglich ist.

Die Anwendung braucht nun lediglich eine Instanzvariable von *AppSettingsManager* zu erzeugen. Im *Load*-Ereignis des Startfensters ruft sie die *LoadSettings*-Methode auf, um die Daten zu laden. Nun kann sie auf die Einstellungen über die *Settings*-Eigenschaft zugreifen. Im *Closing*-Ereignis des Formulars wird der aktuelle Stand durch den Aufruf der *SaveSettings*-Methode gespeichert. Wie die eigentliche Speicherung erfolgt, hängt vom gewählten Speichermedium ab. Im Folgenden wird die Speicherung in einer XML-Datei beschrieben. Eine Anpassung an ein anderes Medium ist jedoch leicht möglich. In der *SaveSettings*-Methode werden mittels Reflection alle Eigenschaften der *AppSettings*-Klasse ermittelt und deren Werte serialisiert. Wichtig ist in diesem Zusammenhang, dass der Typ der jeweiligen Eigenschaft die Serialisierung unterstützen muss. Die meisten Typen des Frameworks bringen diese Funktionalität von Haus aus mit. Bei eigenen Typen erreicht man dies, indem man sie mit dem *Serializable*-Attribut versieht oder die *ISerializable*-Schnittstelle implementiert.

Die Implementierung der *SaveSettings*-Methode zeigt Listing 1. Zunächst wird mit dem *XmlTextWriter* eine neue XML-Datei erstellt. Nun werden die Eigenschaften der *AppSettings*-Klasse in einer Schleife durchlaufen und je ein Element mit dem Namen der Eigenschaft wird erstellt. Mittels *propInfo.GetValue* wird der Inhalt ausgelesen und mit dem *BinaryFormatter* in einen *MemoryStream* serialisiert. Dieser Stream wird schließlich mit der *WriteBase64*-Methode des *XmlTextWriter*s in die XML-Datei geschrieben.

In der *LoadSettings*-Methode findet der umgekehrte Weg statt (siehe Listing 2). Zunächst wird mit dem *XmlTextReader* die Datei geöffnet und durchlaufen. Mit der *ReadBase64*-Methode werden die Binärdaten gelesen und in einem *MemoryStream* gespeichert. Dieser wird im Anschluss mit dem *BinaryFormatter* deserialisiert und der entsprechenden Eigenschaft per *SetValue* zugewiesen.

Enthält die *AppSettings*-Klasse Einstellungen, die in unterschiedlichen Medien gespeichert werden, so bietet es sich an, die entsprechenden Eigenschaften mit eigenen Attributen zu versehen. Diese Attribute könnten dann per Reflection ermittelt und die Einstellungen an dem entsprechenden Speicherort abgelegt werden.

Das folgende Beispiel erstellt ein Attribut für *Roaming*-Einstellungen:

```
[AttributeUsage(AttributeTargets.Property)]
public class RoamingAttribute:System.Attribute
{
   public RoamingAttribute()
   {
   }
}
```

Dieses Attribut ließe sich etwa einer Einstellung in *AppSettings* zuweisen:

```
[Roaming]
public string Vorlagenverzeichnis
{
   get{return m_vorlagenverzeichnis;}
   set{m_vorlagenverzeichnis = value;}
}
```

In einer Schleife würde man alle Eigenschaften durchlaufen (wie dies bereits in Listing 1 gezeigt wurde) und prüfen, ob das *Roaming*-Attribut gesetzt ist.

```
if (null != (RoamingAttribute)Attribute.GetCustomAttribute(propInfo,
typeof(RoamingAttribute)))
```

Ist dies der Fall, wird die Einstellung in einer Datei im *Roaming*-Verzeichnis gespeichert.

Das gleiche Verfahren könnte man auch für alle anderen Speichermedien verwenden. Dieses Vorgehen hat vor allem den Vorteil, dass man nicht für jedes Speichermedium eine eigene Einstellungsklasse definieren muss und die Speicherorte der Einstellungen leicht änderbar sind.

Zugriff auf Einstellungen über die Benutzeroberfläche

Um Einstellungen über die Anwendung zugänglich zu machen, erstellte man in der Vergangenheit meist einen *Optionen*-Dialog, über den der Benutzer Zugriff hatte. Dies hat jedoch den Nachteil, dass beim Hinzufügen und Löschen von Einstellungen das GUI überarbeitet werden muss.

Für dieses Problem stellt das Framework das *PropertyGrid*-Steuerelement zu Verfügung, das auch in der Visual-Studio-IDE verwendet wird, um die Eigenschaften eines Objekts anzuzeigen. Es bietet nicht nur ein intelligentes GUI, sondern ermöglicht auch eine einfache Bindung an jede Art von Klasse. Es liest die Eigenschaften des angegebenen Objekts aus und stellt sie entweder alphabetisch oder nach Kategorien gruppiert dar (siehe Abbildung 2). Änderungen, die der Benutzer durchführt, werden automatisch

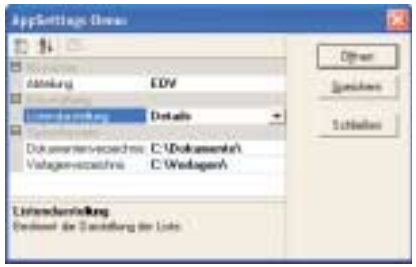


Abbildung 2 | Darstellung von Einstellungen mit dem PropertyGrid.

an das zugewiesene Objekt weitergegeben. Das Einfügen des *PropertyGrid* in ein Formular gestaltet sich etwas komplizierter als bei anderen Steuerelementen, da es standardmässig nicht in der Toolbox angezeigt wird.

Um es über die Toolbox zugänglich zu machen, müssen Sie zunächst folgende Schritte durchführen:

- Öffnen Sie das Kontextmenü der Toolbox und wählen Sie den Punkt *Toolbox anpassen* aus.
- Wählen Sie den Reiter *.NET-Framework Komponenten* aus.
- Markieren Sie den Eintrag *PropertyGrid*.

Um nun die Einstellungsklasse an das *PropertyGrid* zu binden, genügt folgende Codezeile:

Listing 1 Die SaveSettings-Methode schreibt die Eigenschaften der AppSettings-Klasse in binärer Form in eine XML-Datei.

```
public void SaveSettings()
{
   try
   {
      if (null!=m_appSettings)
      {
         int byteCount = 0;
         Byte[] byteBuffer = new Byte[50];
         MemoryStream memStream;
         BinaryFormatter binFormatter = new BinaryFormatter();

         // Datei erstellen
         System.Xml.XmlTextWriter xmlWriter = new XmlTextWriter(
         m_fileName, System.Text.Encoding.Default);
         xmlWriter.Formatting = Formatting.Indented;
         xmlWriter.WriteStartDocument();
         xmlWriter.WriteStartElement("ApplicationSettings");

         // Eigenschaften ermitteln
         Type typeObj = m_appSettings.GetType();
         PropertyInfo[] propInfos = typeObj.GetProperties(
            BindingFlags.Public |
            BindingFlags.Instance |
            BindingFlags.SetField);

         // Alle Eigenschaften durchlaufen
         foreach (PropertyInfo propInfo in propInfos)
         {
            // Element mit Eigenschaftsnamen erstellen
            xmlWriter.WriteStartElement(propInfo.Name);

            // Inhalt der Eigenschaft ermitteln
            object propValue = propInfo.GetValue(m_appSettings, null);
            if (propValue != null)
            {
               // Inhalt serialisieren
               memStream = new MemoryStream();
               binFormatter.Serialize(memStream, propValue);

               // Inhalt als Base64-String einfügen
               memStream.Position = 0;
               byteCount = 0;
               do
               {
                  byteCount = memStream.Read(byteBuffer, 0,
                  byteBuffer.Length);
                  xmlWriter.WriteBase64(byteBuffer, 0, byteCount);
               }
               while (byteCount == byteBuffer.Length);
            }
            xmlWriter.WriteEndElement();
         }
         xmlWriter.WriteEndDocument();
         xmlWriter.Flush();
         xmlWriter.Close();
      }
      else
         throw new ArgumentException("Es wurde ein AppSetting-Objekt
         angegeben!");
   }
   catch(Exception e)
      {throw e;}
}
```


Listing 2 Die LoadSettings-Methode deserialisiert die gespeicherten Einstellungen aus der XML-Datei.

```
public bool LoadSettings()
{
   try
   {
      int byteCount = 0;
      string elementName = "";
      Byte[] byteBuffer = new Byte[50];
      MemoryStream memStream;
      BinaryFormatter binFormatter = new BinaryFormatter();

      // Datei einlesen
      System.Xml.XmlTextReader xmlReader = new XmlTextReader(m_fileName);
      while (xmlReader.Read())
      {
         if (xmlReader.NodeType == XmlNodeType.Element)
         {
            // Einstellungsname auslesen
            elementName = xmlReader.Name;

            // Binärdaten auslesen
            if (elementName != "ApplicationSettings")
            {
               memStream = new MemoryStream();
               byteCount = 0;
               do
               {
                  byteCount = xmlReader.ReadBase64(byteBuffer, 0,
                  byteBuffer.Length);
                  memStream.Write(byteBuffer, 0, byteCount);
               }
               while (byteCount == byteBuffer.Length);

               // Eigenschaft deserialisieren
               if (memStream.Length!=0)
               {
                  memStream.Position = 0;
                  object propValue = binFormatter.Deserialize(memStream);
                  Type typeObj = m_appSettings.GetType();
                  PropertyInfo propInfo = typeObj.GetProperty(elementName);
                  propInfo.SetValue(m_appSettings, propValue, null);
               }

            }
         }
      }
      xmlReader.Close();

      return true;
   }
   catch(Exception e)
   {
      throw e;}
}
```

```
propertyGrid1.SelectedObject = settingsManager.Settings;
```

Möchte man die Einstellungen in verschiedene Gruppen einteilen, so versieht man die entsprechenden Eigenschaften in der *AppSettings*-Klasse mit dem Attribut *Category*, über das man den Titel der Gruppe angibt. Zusätzlich besteht die Möglichkeit, einen kurzen Beschreibungstext für die Einstellung zu hinterlegen, der im Fußbereich des *PropertyGrid*s angezeigt wird. Dies geschieht über das *Description*-Attribut. Soll eine Einstellung nicht im Grid angezeigt werden, so versieht man sie mit dem Attribut *[Browsable(false)]*. Um eine Einstellung schreibgeschützt darzustellen, setzt man das *ReadOnly*-Attribut auf *true*. Um diese Attribute verwenden zu können, muss der Namespace *System.ComponentModel* in der entsprechenden Assembly eingebunden werden.

Das folgende Beispiel weist der Gruppe *Darstellung* die Einstellung *Listendarstellung* zu und versieht sie mit einem Beschreibungstext:

```
[Category("Darstellung"),
Description("Bestimmt die Darstellung der Liste.")]
public string Listendarstellung
{
   get{return m_listendarstellung;}
   set{m_listendarstellung = value;}
}
```

Weiterführende Informationen über das *PropertyGrid* finden Sie unter [4].

Fazit

Wie dotnetpro zeigt, ist das Verwalten von Anwendungsdaten komplexer, als es auf den ersten Blick scheint. Die Entscheidung, wo man am besten die Einstellungen speichert, hängt im Wesentlichen von den Anforderungen an Sicherheit, Administration und Mobilität ab. Jedoch bietet das .NET Framework ein reichhaltiges Angebot an Klassen und Infrastruktur, um diesen Anforderungen gerecht zu werden. |||||||

[1] Zugreifen auf die Registrierung mit Visual Basic .NET
www.microsoft.com/germany/ms/msdnbiblio/show_all.asp?siteid=545236

[2] Isolated Storage Tool (Storeadm.exe)
http://msdn.microsoft.com/library/en-us/cptools/html/cpgrfisolatedstorageutilitystoreadmexe.asp

[3] Introduction to Isolated Storage http://msdn.microsoft.com/library/en-us/cpguide/html/cpconisolatedstorage.asp

[4] Getting the Most Out of the .NET Framework PropertyGrid Control
http://msdn.microsoft.com/library/?url=/library/en-us/dndotnet/html/usingpropgrid.asp