JÜRGEN BAYER

Das C# 2008 Codebook

Auf der Interaktiv-CD

- Durchsuchbares HTML-Repository
- Umfangreiche Artikel zu API, WMI, ADSI, WPF, LINQ und LINQ to SQL
- Weblog mit Aktualisierungen
- eBook

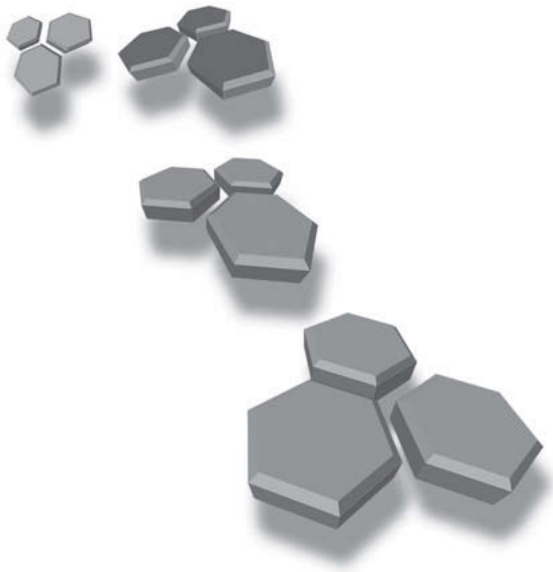
Auf der DVD

- Visual Studio Professional 2008 Edition (90-Tage-Testversion)

ADDISON-WESLEY

interaktiv CD DVD

Teil II Rezepte



Basics

Datum und Zeit

Anwendungen, Anwendungs-Konfiguration, Prozesse und Dienste

Dateisystem

Text-, binäre und Zip-Dateien

XML

System

Internet

Windows.Forms

WPF

LINQ und LINQ to SQL

Benutzer und Gruppen

Sicherheit

Multimedia

Bildbearbeitung

Zeichnen

COM-Interop mit Office

Reflection und Serialisierung

Threading und asynchroner Methodenaufruf

Datenbank-Programmierung

Anwendungen, Anwendungs-Konfiguration, Prozesse und Dienste

80 Den Dateinamen der Anwendung ermitteln

Den vollen Dateinamen der ausführenden Anwendung (inklusive Pfad) erhalten Sie in einer Windows-Anwendung über die Eigenschaft `ExecutablePath` des `Application`-Objekts:

```
string filename = System.Windows.Forms.Application.ExecutablePath;
```

Um die `Application`-Klasse verfügbar zu haben, benötigt die Anwendung einen Verweis auf die Assembly *System.Windows.Forms.dll*, was ohne Probleme auch in einer Konsolenanwendung oder Klassenbibliothek möglich ist.

Ohne diesen Verweis können Sie aber auch stattdessen die Eigenschaft `Location` eines `Assembly`-Objekts abfragen, das die »Eintritts-Assembly« (das ist die Assembly, über die das Programm gestartet wurde) repräsentiert. Dieses Objekt erhalten Sie über die Methode `GetEntryAssembly` der `Assembly`-Klasse, die Bestandteil des `System.Reflection`-Namensraums ist:

```
string filename = System.Reflection.Assembly.GetEntryAssembly().Location;
```

Hinweis

Als *Reflektion* (Reflection) werden Features bezeichnet, über die Sie in der Laufzeit eines Programms Informationen über dessen Assemblys, die darin enthaltenen Klassen und deren Schnittstellen, Methoden, Eigenschaften und Ereignisse etc. auslesen können. Ein weiterer Teil von Reflektion ist das dynamische Erzeugen von Klassen und Assemblys, also das Erzeugen von Programmcode in der Laufzeit eines Programms.

Häufig wird auch vorgeschlagen, die »ausführende Assembly« zu verwenden, die Sie über die Methode `GetExecutingAssembly` erreichen:

```
string filename = System.Reflection.Assembly.GetExecutingAssembly().Location;
```

Dabei müssen Sie allerdings aufpassen: Wird diese Methode in einer Klassenbibliothek aufgerufen, liefert sie ein `Assembly`-Objekt zurück, das die Assembly der *Klassenbibliothek* repräsentiert. Der ausgelesene Dateiname ist also in diesem Fall nicht der der Anwendung, sondern der der Klassenbibliothek.

In allen Fällen erhalten Sie den Dateinamen inklusive Pfad. Wenn Sie nur den Namen der Datei auslesen wollen, können Sie diesen über ein `FileInfo`-Objekt ermitteln. Ein solches Objekt dient dem Auslesen von Informationen zu einer Datei und liefert u. a. über die Eigenschaft `Name` den Dateinamen. Die folgende Funktion liest den Dateinamen der Anwendung aus und liefert diesen zurück:

```
public static string GetApplicationFilename()
{
   // FileInfo-Objekt für die Datei erzeugen, die die Eintritts-
   // Assembly speichert
   System.IO.FileInfo fi = new System.IO.FileInfo(
      System.Reflection.Assembly.GetEntryAssembly().Location);
   // Dateiname auslesen und zurückgeben
   return fi.Name;
}
```

Listing 130: Methode zum Ermitteln des Dateinamens der Anwendung

81 Das Verzeichnis der Anwendung ermitteln

In einer Anwendung mit Referenz auf die Assembly *System.Windows.Forms.dll* können Sie den Pfad der Anwendung über die Eigenschaft `StartupPath` des `Application`-Objekts ermitteln:

```
string applicationPath = System.Windows.Forms.Application.StartupPath;
```

Ohne Referenz auf die *System.Windows.Forms.dll* können Sie stattdessen auch den vollen Dateinamen der »Eintritts-Assembly« ermitteln und über ein `FileInfo`-Objekt den Pfad auslesen:

```
public static string GetApplicationFolderName()
{
   // FileInfo-Objekt für die Datei erzeugen, die die Eintritts-
   // Assembly speichert */
   System.IO.FileInfo fi = new System.IO.FileInfo(
      System.Reflection.Assembly.GetEntryAssembly().Location);

   // Den Pfad des Verzeichnisses der Datei zurückgeben
   return fi.DirectoryName;
}
```

Listing 131: Methode zur Ermittlung des Ordners, in dem die Anwendung gespeichert ist

Beide Varianten funktionieren auch ohne Probleme in einer Klassenbibliothek. Sie erhalten dann allerdings das Verzeichnis der Anwendung zurück, nicht das der Klassenbibliothek.

82 Befehlszeilenargumente auswerten

Viele Standardanwendungen können mit Befehlszeilenargumenten aufgerufen werden. Dem Windows-Explorer können Sie zum Beispiel beim Aufruf den Pfad zu einem Ordner übergeben, den dieser anzeigen soll:

```
explorer C:\Windows
```

Befehlszeilenargumente werden dabei durch Leerzeichen vom Programmdateinamen und von anderen Befehlszeilenargumenten getrennt. Beim folgenden Aufruf eines Programms:

```
Demo.exe Das ist ein Test
```

werden die Argumente "Das", "ist", "ein" und "Test" übergeben. Werden Argumente in Anführungszeichen eingeschlossen, resultiert ein einziges Argument:

```
Demo.exe "Das ist ein Test"
```

Dieses Beispiel resultiert in dem einzigen Argument "Das ist ein Test". Argumente, die selbst Leerzeichen enthalten, müssen beim Aufruf also in Anführungszeichen eingeschlossen werden:

```
explorer "C:\Dokumente und Einstellungen"
```

In Ihren (Windows-)Anwendungen können Sie Befehlszeilenargumente ebenfalls auswerten. In einer `Windows.Forms`- oder Konsolenanwendung können Sie dazu die `Main`-Methode, die normalerweise in der Klasse `Program` implementiert ist, um ein Argument vom Typ `string`-Array erweitern:

```
[STAThread]
static void Main(string[] arguments)
{
```

Listing 132: Erweitern der Main-Methode um ein String-Array-Argument

In diesem Array werden alle Argumente übergeben, die beim Aufruf des Programms angegeben wurden. Dieses Vorgehen ist jedoch nicht zu empfehlen, da Sie zum einen auf die Auswertung in der `Main`-Methode eingeschränkt sind. Zum anderen wird die `Main`-Methode in einer mit Visual Studio entwickelten WPF-Anwendung automatisch erzeugt und Sie haben keine Möglichkeit, darin zu programmieren.

Der flexiblere Weg ist der über die `GetCommandLineArgs`-Methode der `Environment`-Klasse aus dem Namensraum `System`. Diese Methode liefert ein `String`-Array zurück, das (anders als der Name vermuten lässt) den kompletten Aufruf der Anwendung enthält. Im ersten Element steht immer der Dateiname der Anwendung. In den folgenden Elementen werden alle Befehlszeilenargumente verwaltet. Damit ist es recht einfach, die einzelnen Argumente auszulesen.

Um keine Probleme bei der Auswertung zu haben, sollten Sie die Syntax der einzelnen Argumente so festlegen, dass diese prinzipiell keine Leerzeichen enthalten. In der Praxis bestehen viele Befehlszeilenargumente aus einem Binde- oder Schrägstrich, gefolgt von einem Namen, einem Doppelpunkt und einem Wert:

```
-Argumentname:Argumentwert
```

Beim Aufruf der Anwendung kann der Argumentwert in Anführungszeichen eingeschlossen werden, sofern dieser Leerzeichen enthält:

```
Imager.exe -imageFolder:"C:\Bilder für die Website"
```

Die Auswertung solcher Argumente ist dann relativ einfach. Dazu gehen Sie die Elemente des von `GetCommandLineArgs` zurückgegebenen `String`-Array ab dem Index 1 durch und vergleichen die einzelnen Argumente mit den erwarteten. Die Programmierung kann an beliebiger Stelle innerhalb der Anwendung erfolgen, wird aber meistens in der `Load`-Methode des Start-Formulars bzw. -Fensters oder in der `Main`-Methode untergebracht.

Das folgende Beispiel liest auf diese Weise die erwarteten Argumente *-debugmode* und *-imagefolder:Ordnerangabe* aus. Beim *imagefolder*-Argument wird die Angabe eines Ordners erwartet, der mit einem Doppelpunkt vom Argumentnamen getrennt wird. Um die Groß-/Kleinschreibung nicht zu berücksichtigen werden die Argumente über die `ToLower`-Methode in Kleinschreibung umgewandelt. Zur Sicherheit werden unbekannte Argumente in der `String`-Variablen `unknownArguments` gesammelt und falls vorhanden nach der Auswertung in einer `MessageBox` gemeldet.

Das Beispiel basiert auf einer WPF-Anwendung mit den üblichen Referenzen und `using`-Direktiven. Im Formular ist ein Label angelegt, das *lblArguments* heißt. In diesem Label werden die übergebenen Befehlszeilenargumente ausgegeben.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
   // Auswerten der Befehlszeilenargumente
   bool debugMode = false;
   string imageFolder = null;
   string unknownArguments = null;
   string[] args = Environment.GetCommandLineArgs();
   for (int i = 1; i < args.Length; i++)
   {
      string loweredArgument = args[i].ToLower();
      if (loweredArgument == "-debugmode")
      {
         debugMode = true;
      }
      else if (loweredArgument.StartsWith("-imagefolder:"))
      {
         // Den Argumentwert auslesen
         imageFolder = args[i].Substring(13, args[i].Length - 13);
      }
      else
      {
         // Unbekanntes Argument
         if (unknownArguments != null)
         {
            unknownArguments += ", ";
         }
         unknownArguments += args[i];
      }
   }

   // Unbekannte Argumente auswerten
   if (unknownArguments != null)
   {
      MessageBox.Show("Die folgenden Argumente sind ungültig: " +
```

Listing 133: Auswerten von Befehlszeilenargumenten

```
            unknownArguments, "Befehlszeilenargumente auswerten",
            MessageBoxButton.OK, MessageBoxImage.Exclamation);
    }

    // Die Argumente auswerten
    this.lblArguments.Content = "debugmode: " + debugMode +
        Environment.NewLine + "imageFolder: " + imageFolder;
}
```

Listing 133: Auswerten von Befehlszeilenargumenten (Forts.)

Sehr nett von Windows ist, dass in Anführungszeichen eingeschlossene Argumente automatisch so ausgewertet werden, dass die Anführungszeichen entfernt werden. Beim Aufruf mit den Argumenten

```
-debugMode -imageFolder:"C:\Bilder für die Website"
```

werden zum Beispiel "-debugMode" und "-imageFolder:C:\Bilder für die Website" übergeben.

Hinweis

Zum Testen von Befehlszeilenargumenten können Sie diese in Visual Studio in den Eigenschaften des Projekts im Register DEBUGGEN in das Feld BEFEHLSZEILENARGUMENTE eintragen.

83 User Account Control (UAC) berücksichtigen

Ihre Anwendungen sollten die Windows-Vista-Technologie *User Account Control* (UAC) berücksichtigen. Das ist schon deswegen notwendig, weil unter UAC per Voreinstellung alle Anwendungen grundsätzlich mit den eingeschränkten Rechten eines Standard-Benutzers ausgeführt werden, auch wenn der ausführende Benutzer ein Administrator ist. Ein Standard-Benutzer besitzt normalerweise nur eingeschränkte Rechte und kann bestimmte Aktionen, wie z. B. das Einstellen der Systemzeit, nicht ausführen.

Dieses Rezept klärt, wie Sie mit UAC umgehen sollten (ohne allerdings über den Sinn oder Unsinn von UAC zu diskutieren).

Exkurs

User Account Control ist eine Sicherheits-Technologie unter Windows-Vista, die dabei helfen soll, ein System abzusichern. UAC ist per Voreinstellung eingeschaltet, kann aber auch komplett abgeschaltet oder speziell konfiguriert werden. Das Prinzip von UAC basiert darauf, dass Anwendungen normalerweise mit den (eingeschränkten) Rechten ausgeführt werden, die ein Windows-Standardbenutzer besitzt. Das gilt auch dann, wenn der Windows-Benutzer ein Administrator ist. Dies wird dadurch erreicht, dass bereits der Windows-Explorer (der unter anderem auch den Desktop ausführt) mit einem Sicherheits-Token ausgeführt wird, das der Gruppe der Standardbenutzer zugeordnet ist. Jede Anwendung, die unter dem Explorer gestartet wird (was normalerweise jede gestartete Anwendung ist), erbt die Sicherheits-Einstellungen vom Explorer und wird damit auch unter dem eingeschränkten Sicherheits-Token ausgeführt.

Benötigt eine Anwendung nun zur Ausführung einer Aktion Administrationsrechte, fragt UAC in einer speziellen Sicherheits-Box ab, ob zur Ausführung der Aktion die Rechte der Anwendung angehoben werden sollen. Wie die Abfrage aussieht, hängt davon ab, ob der in Windows eingeloggte Benutzer Administrator ist. Ist der Benutzer Administrator, handelt es sich nur um eine einfache Abfrage. Ist der Benutzer kein Administrator, beinhaltet die Sicherheits-Abfrage einen Login, mit dem der Benutzer sich als Administrator authentifizieren kann.

Hinweis

Ausführlichere Informationen zu UAC finden Sie in dem Microsoft-Artikel »Windows Vista Application Development Requirements for User Account Control Compatibility«, den Sie an der Adresse *www.microsoft.com/downloads/details.aspx?FamilyID=BA73B169-A648-49AF-BC5E-A2EEBB74C16B* (sorry, aber die Adressen hat Microsoft so gewählt ...) downloaden können. Sie finden den Link zu dem Dokumenten-Download auch in der Datei *Links.html* auf der Buch-CD.

Einschränkungen

UAC führt zu einigen Einschränkungen für Anwendungen:

- Ohne Anhebung des Sicherheits-Levels (die, wie unten beschrieben, nicht immer ausgeführt wird) kann eine Anwendung keine administrativen Aufgaben ausführen. Außerdem sind die Rechte für bestimmte System-Ordner meist auf das Lesen eingeschränkt.
- Wird das Sicherheits-Level einer Anwendung von UAC angehoben, führt dies in der Regel zu Problemen bei der Kommunikation zwischen Fenstern, z. B. beim Drag&Drop. UAC erlaubt nämlich nicht, dass ein Fenster Nachrichten an ein Fenster mit einer höheren Priorität sendet. Als eines der wichtigsten Probleme dieses Sicherheits-Features sehe ich, dass unter UAC Drag&Drop zwischen einem Explorer-Fenster und einer höher gestuften Anwendung nicht mehr funktioniert.

Anhebung des Sicherheits-Levels

Das Anheben des Sicherheits-Level bei der Ausführung von Aktionen, die administrative Rechte erfordern, funktioniert bei einigen neueren Microsoft-Anwendungen relativ problemlos. Sie können das mit dem Windows-Explorer ausprobieren, indem Sie versuchen, im Windows-Ordner eine Datei oder einen Ordner anzulegen. Da ein Standardbenutzer (normalerweise) keine Schreibrechte im Windows-Ordner besitzt, fragt UAC ab, ob die Aktion mit den Rechten eines Administrators ausgeführt werden soll.

UAC überprüft auch, ob eine Anwendung eine Installations- oder Update-Anwendung ist, was an verschiedenen Indikatoren wie z. B. dem Text »install«, »setup« oder »update« im Dateinamen der Anwendung erkannt wird. Für diese Art von Anwendungen erfolgt dann auch eine Anhebung des Sicherheits-Levels.

Hinweis

Für andere Anwendungen, und dazu gehört nach meinen Tests auch z. B. Word 2007, erfolgt allerdings keine automatische Anhebung. Dies gilt zunächst einmal auch für normale .NET-Anwendungen: Für diese wird beim Versuch eine administrative Aufgabe auszuführen oder in einen Ordner zu schreiben, der für Standardbenutzer schreibgeschützt ist, eine `SecurityException` generiert.

Wenn Sie eine .NET-Anwendung entwickeln, die administrative Aufgaben ausführen soll, müssen Sie die gewünschte Anhebung des Sicherheits-Levels im Manifest der Anwendung deklarieren.

In Visual Studio fügen Sie dem Projekt dazu eine Anwendungs-Manifest-Datei (Application manifest file) hinzu. Wenn Sie die Rechte der Anwendung auf Administrator-Rechte anheben lassen wollen, sieht der Inhalt der Manifestdatei folgendermaßen aus:

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
  xmlns="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</asmv1:assembly>
```

Listing 134: Inhalt einer Manifestdatei zur Anhebung des Sicherheits-Levels der Anwendung auf das Niveau eines Administrators

Im `requestedExecutionLevel`-Element stellen Sie über das `level`-Attribut den angeforderten Sicherheits-Level ein. Möglich sind hier die folgenden Einstellungen:

- `asInvoker`: Die Anwendung wird grundsätzlich unter dem Sicherheits-Token des Prozesses ausgeführt, unter dem die Anwendung gestartet wurde.
- `highestAvailable`: Die Anwendung fordert an, dass sie unter dem höchsten Sicherheits-Token ausgeführt wird, das der aktuelle Benutzer anfordern kann.
- `requireAdministrator`: Die Anwendung fordert an, dass sie unter einem Administrator-Token ausgeführt wird.

Hinweis

Beachten Sie, dass das Anheben der Sicherheits-Levels beim Testen unter Visual Studio nicht funktioniert, da die Anwendung in diesem Fall unter dem Visual-Studio-Host-Prozess und Visual Studio nur unter dem Sicherheits-Token eines Standardbenutzers ausgeführt wird. Wenn Sie das Anheben testen wollen, müssen Sie das entsprechende Projekt unter Visual Studio mit STRG + F5 starten (was dazu führt, dass das Sicherheits-Level von Visual Studio angehoben wird) oder die erzeugte .exe-Datei direkt ausführen.

Hinweis

Beachten Sie, dass Sie die Anhebung des Sicherheits-Levels nur in Sonderfällen anfordern und in der Regel Anwendungen entwickeln sollten, die keine höheren Rechte als die eines Standardbenutzers erfordern.

Entwickeln von Anwendungen unter Rücksichtnahme auf UAC

Microsoft empfiehlt, Anwendungen grundsätzlich so zu entwickeln, dass diese keine höheren Rechte als die eines Standardbenutzers erfordern. Wenn Sie dies berücksichtigen, spielt UAC für Ihre Anwendungen keine Rolle.

Ihre Anwendungen sollten daher die folgenden Regeln einhalten:

- Sie sollten keine administrativen Aufgaben ausführen, wie zum Beispiel beim Start den Setup-Prozess vervollständigen.
- Sie sollten nicht direkt in das Windows-Verzeichnis, das Programm-Verzeichnis oder in deren Unterverzeichnisse schreiben.
- Die Anwendungen sollten Konfigurations- oder andere Daten, die geschrieben werden müssen, grundsätzlich in dem privaten Ordner des aktuellen Benutzers (unter Vista: C:\Users\<Benutzername>) oder im Daten-Ordner für alle Benutzer (unter Vista: C:\Users\All Users) verwalten.
- Ein automatisches (Online-)Update sollte eine Technik verwenden, die für Standardbenutzer verfügbar ist, wie z. B. das *Windows Installer 4.0 User Account Control Patching*.
- Die Anwendung sollte keine hart-codierten Pfade verwenden.

Hinweis

Zum Testen der Anforderung einer Anwendung auf administrative Anforderungen können Sie den von Microsoft bereitgestellten *Microsoft Standard User Analyzer* verwenden, den Sie im *Microsoft Application Compatibility Toolkit 5.0* finden. Laden Sie dieses an der Adresse *www.microsoft.com/downloads/details.aspx?FamilyID=24da89e9-b581-47b0-b45e-492dd6da2971* herunter (den Link finden Sie wie immer auch in der Datei *Links.html* auf der Buch-CD).

84 Ausnahmen global behandeln

Ausnahmen, die in der Anwendung nicht behandelt werden, werden von der CLR in einem einfachen Dialog gemeldet. Dieser Dialog (der bei `Windows.Forms`-Anwendungen per Voreinstellung ein anderer ist als bei Konsolen- und WPF-Anwendungen) ist mehr oder weniger aussagekräftig, wie Abbildung 55 und Abbildung 56 zeigen.

Abbildung 55: Meldung der CLR bei einer unbehandelten Ausnahme in einer Windows.Forms-Anwendung ohne Debug-Modus

Abbildung 56: Meldung der CLR bei einer unbehandelten Ausnahme in einer WPF- oder Konsolenanwendung oder in einer Windows.Forms-Anwendung, die zum Debugging eingestellt ist

Der DEBUGGEN-Schalter im neueren Dialog (Abbildung 56) ist nur dann vorhanden, wenn auf dem System ein Debugger installiert ist. Dies ist nur dann der Fall, wenn das .NET-Framework-SDK installiert ist. Über diesen Schalter kann die Anwendung debugt werden. Der Debugger berücksichtigt aber für den Fall, dass der Quellcode nicht im Ordner der Anwendung gespeichert ist (was ja in der Regel nicht der Fall ist), nur den CIL-Code. Damit können nur absolute System-Profis etwas anfangen.

Hinweis

Für `Windows.Forms`-Anwendungen können Sie in der Konfiguration einstellen, dass statt des Standard-Dialogs (Abbildung 55) der neuere Dialog (Abbildung 56) mit der Möglichkeit, zu debuggen, angezeigt wird. Dazu setzen Sie in der Maschinen- oder Anwendungskonfiguration im Element *system.windows.forms* das Attribut *jitDebugging* auf `true`. Zusätzlich dazu muss die Anwendung im Debug-Modus kompiliert worden sein.

Die meisten Benutzer werden wohl mit diesen Möglichkeiten überfordert sein. Außerdem kann es sein, dass beim Beenden der Anwendung Daten verloren gehen. Um dies zu verhindern können Sie selbstverständlich alle Ausnahmen im Programm explizit behandeln, was Sie dadurch erreichen, dass Sie zumindest in jeder Ereignismethode eine Ausnahmebehandlung implementieren. Eigentlich ist dies auch der bessere Weg, da Sie dem Benutzer dann genauere Informationen über den Kontext des Fehlers geben können.

Alternativ können Sie unbehandelte Ausnahmen aber auch global abfangen. Basis dieser Technik ist für WPF-Anwendungen das Ereignis `DispatcherUnhandledException` des Objekts, das die Anwendung repräsentiert. In einer `Windows.Forms`-Anwendung verwenden Sie das statische Ereignis `OnThreadException` der Klasse `Windows.Forms.Application`. Diese Ereignisse werden für alle unbehandelten Ausnahmen aufgerufen, die im UI-Thread der Anwendung eintreten.

Achtung

Diese Ereignisse werden allerdings nicht für unbehandelte Ausnahmen aufgerufen, die in einem anderen als dem UI-Thread auftreten. In eigenen Arbeits-Threads sollten Sie also trotz globaler Ausnahmebehandlung immer eine eigene Ausnahmebehandlung vorsehen.

In einer WPF-Anwendung erreichen Sie das `DispatcherUnhandledException`-Ereignis über die von `Application` abgeleitete Klasse (normalerweise die `App`-Klasse), die die WPF-Anwendung repräsentiert. Im Konstruktor können Sie das Ereignis zuweisen. Die Eigenschaft `Exception` des Ereignisargument-Objekts liefert die aufgetretene Ausnahme. Nach der Verarbeitung der Ausnahme müssen Sie die Eigenschaft `Handled` des übergebenen Ereignisargument-Objekts auf `true` setzen, damit die Ausnahme nicht an die CLR weitergegeben wird:

```
public partial class App : Application
{
   /* Konstruktor */
   public App()
   {
```

Listing 135: Globale Ausnahmebehandlung in einer WPF-Anwendung

```
      // Zuweisen der globalen Ausnahmebehandlung für den UI-Thread
      this.DispatcherUnhandledException +=  new System.Windows.Threading.
         DispatcherUnhandledExceptionEventHandler(
         this.App_DispatcherUnhandledException);
   }

   /* Behandelt alle unbehandelten Ausnahmen, die im UI-Thread auftreten.
      Behandelt keine unbehandelten Ausnahmen, die in anderen Threads
      auftreten! */
   private void App_DispatcherUnhandledException(object sender,
      System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e)
   {
      // Ausnahme ausgeben
      MessageBox.Show("Unerwarteter Fehler: " + e.Exception.Message,
         "Globale Ausnahmebehandlung in WPF", MessageBoxButton.OK,
         MessageBoxImage.Error);

      // Das Ereignis als 'Behandelt' kennzeichnen, damit die Ausnahme
      // nicht an die CLR weitergegeben wird
      e.Handled = true;

      // Hier können (und sollten) Sie Ausnahmen protokollieren.
      // Wenn Sie den Stack-Trace protokollieren, hilft diese enorm
      // bei der späteren Fehlersuche.
   }
}
```

Listing 135: Globale Ausnahmebehandlung in einer WPF-Anwendung (Forts.)

In einer `Windows.Forms`-Anwendung weisen Sie dem `OnThreadException`-Ereignis in der `Main`-Methode (in *Program.cs*) eine passende Ereignisbehandlungsmethode zu. In `OnThreadException` existiert keine Handled-Eigenschaft, die Sie auf `true` setzen müssen (wie in WPF). Wenn in einer `Windows.Forms`-Anwendung Ausnahmen global abgefangen werden, werden diese implizit nicht an die CLR weitergegeben.

```
static class Program
{
   [STAThread]
   static void Main()
   {
      // Zuweisen der globalen Ausnahmebehandlung für den UI-Thread
      Application.ThreadException +=
         new System.Threading.ThreadExceptionEventHandler(
         Application_ThreadException);

      Application.EnableVisualStyles();
```

Listing 136: Globale Ausnahmebehandlung in einer Windows.Forms-Anwendung

```
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new MainForm());
   }

   /* Behandelt alle unbehandelten Ausnahmen, die im UI-Thread auftreten.
      Behandelt keine unbehandelten Ausnahmen, die in anderen Threads
      auftreten! */
   private static void Application_ThreadException(object sender,
      System.Threading.ThreadExceptionEventArgs e)
   {
      // Ausnahme ausgeben
      MessageBox.Show("Unerwarteter Fehler: " + e.Exception.Message,
         Application.ProductName, MessageBoxButtons.OK,
         MessageBoxIcon.Error);

      // Hier können (und sollten) Sie Ausnahmen protokollieren.
      // Wenn Sie den Stack-Trace protokollieren, hilft dies enorm
      // bei der späteren Fehlersuche.
   }
}
```

Listing 136: Globale Ausnahmebehandlung in einer Windows.Forms-Anwendung (Forts.)

Tipp

Falls Sie im Programm alle erwarteten Ausnahmen explizit abfragen und in der Methode für die globale Ereignisbehandlung eigentlich nur unerwartete Ausnahmen abgefangen werden, sollten Sie dem Anwender neben einer Information über den aufgetretenen Fehler die Möglichkeit geben, die Details (der Ausnahme-Meldung inkl. aller inneren Ausnahmen und den Stack-Trace) an eine Support-E-Mail-Adresse zu mailen (siehe Rezept 211). Um genauere Informationen über die Ausnahme zu erhalten sollten Sie dem Release der Anwendung die automatisch erstellte Debug-Informationsdatei (mit der Endung *.pdb*) mitliefern. So erhalten Sie im Stack-Trace zusätzliche Informationen über die Quelle des Fehlers (inklusive der Zeilennummer). Diese Informationen erleichtern das Debuggen von Fehlern, die lediglich beim Anwender auftreten.

85 Den Speicherverbrauch von .NET-Anwendungen verstehen und gegebenenfalls reduzieren

Schaut man sich die Speicherauslastung von .NET-Anwendungen im Windows-Taskmanager an, so scheint es, als würden diese Anwendungen relativ viel Speicher verbrauchen. Bereits eine minimale »Hello World«-Anwendung belegt in der Regel mehrere Megabyte. Größere Anwendungen kommen schnell auf ein Vielfaches von zehn oder sogar ein Vielfaches von einhundert Megabyte (besonders wenn diese fahrlässig programmiert wurden).

Die Speicherauslastung, die im Taskmanager angezeigt wird, zeigt jedoch nicht den tatsächlichen Speicherverbrauch der Anwendung. Vielmehr handelt es sich dabei um den für die Anwendung reservierten Speicher. Die CLR reserviert für eine laufende Anwendung relativ großzügig Speicher, damit bei der Ausführung der Anwendung auf das inperformante Neu-

Reservieren und Umschichten des Speichers verzichtet werden kann. Dabei wird natürlich der im Moment verfügbare Arbeitsspeicher berücksichtigt. Ist dieser relativ klein, erhält die Anwendung nur relativ wenig Speicher. Hinzu kommt, dass auch Windows aus Performancegründen relativ großzügig mit der Reservierung von Speicher umgeht. Im Prinzip brauchen Sie sich also um den Arbeitsspeicher keine Gedanken zu machen.

Wichtig ist allerdings, dass Sie in Ihren Programmen nach der Verwendung von Objekten deren `Dispose`-Methode aufrufen, sofern diese eine solche besitzen. `Dispose` sorgt dafür, dass externe Ressourcen freigegeben werden. Diese Methode wird zwar auch im Destruktor der Objekte, die eine solche besitzen, aufgerufen (sofern die entsprechenden Klassen sorgfältig entwickelt wurden). Die CLR stellt aber nicht unbedingt sicher, dass der Destruktor (der ja eigentlich nur eine Finalisierungsmethode ist) aufgerufen wird. Das gilt im Besonderen für kleine Objekte wie die, die beim Zeichnen mit GDI+ verwendet werden (siehe Rezept 338).

Um sich die Arbeit zu vereinfachen können Sie Objekte, die eine `Dispose`-Methode besitzen, auch in einer `using`-Direktive erzeugen:

```
string sourceFilename = Path.Combine(
   Application.StartupPath, "Les Crosets.jpg");
using (Bitmap sourceBitmap = new Bitmap(sourceFilename))
{
   // Bitmap bearbeiten
}
```

Listing 137: Verwenden der using-Direktive

Der Compiler setzt die `using`-Direktive so um, dass automatisch hinter allen Anweisungen, die in den Block-Klammern angegeben werden, auf eine sichere Weise `Dispose` aufgerufen wird. Der resultierende IL-Code würde dann zurückübersetzt in etwa folgendermaßen aussehen:

```
string sourceFilename = Path.Combine(
   Application.StartupPath, "Les Crosets.jpg");
Bitmap sourceBitmap = null;
try
{
   sourceBitmap = new Bitmap(sourceFilename))

   // Bitmap bearbeiten
}
finally
{
   if (sourceBitmap != null)
   {
      try
      {
         sourceBitmap.Dispose();
```

Listing 138: Das, was der Compiler aus einem using-Block macht

```
        }
        catch {}
    }
}
```

Listing 138: Das, was der Compiler aus einem using-Block macht (Forts.)

Über die sehr einfach anzuwendende `using`-Direktive stellen Sie sicher, dass Ihre Programme die teilweise sehr umfangreichen externen Ressourcen, die manche Objekte verwenden, explizit freigeben und damit möglichst wenig Speicher verbrauchen.

Falls Ihr Programm trotz des korrekten Aufrufs von `Dispose` zu viel Speicher »verbraucht«, können Sie zunächst versuchen die `Collect`-Methode des Garbage Collectors aufzurufen, den Sie über `System.GC` erreichen. `GC.Collect` »erzwingt« eine Garbage Collection. Diese Methode sorgt dafür, dass der Garbage Collector aufgefordert wird, seine Arbeit auszuführen und die Objekte, die nicht mehr referenziert werden, freizugeben. Dies führt jedoch nicht unbedingt dazu, dass die nicht mehr referenzierten Objekte auch wirklich freigegeben werden, denn der Garbage Collector führt seine Arbeit nur dann aus, wenn sich eine Garbage Collection lohnt und sich das Programm gerade in einem untätigen Zustand befindet.

Wenn Sie nichts übergeben, sorgt `Collect` für eine Müllsammlung in allen Generationen des GC. Sie können aber auch den Index der Generation übergeben, die behandelt werden soll. Eine Müllsammlung in der Generation 0 wird wesentlich schneller ausgeführt als eine Müllsammlung in allen Generationen:

```
GC.Collect(0);
```

In der Praxis würde ich aber immer alle Generationen behandeln, da der Sinn eines manuellen Aufrufs wohl sein sollte, möglichst viel Speicher freizugeben.

Wenn Sie `Collect` aufrufen, sollten Sie danach über die Methode `WaitForPendingFinalizers` dafür sorgen, dass der Finialisierer-Aufruf-Thread seine Arbeit ausgeführt hat. Danach sollten Sie `Collect` noch einmal aufrufen:

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

In absolut hartnäckigen Fällen können Sie auch die API-Funktion `SetProcessWorkingSetSize` aufrufen. Diese Funktion, die nur in Windows NT und ab Windows XP zur Verfügung steht, trimmt das von einer Anwendung reservierte *Working Set* (Set von physikalischen Speicherseiten). Am ersten Argument übergeben Sie den Handle des Prozesses Ihrer Anwendung, am zweiten Argument einen Byte-Minimalwert für die verbleibenden Speicherseiten und am dritten einen Maximalwert. Wenn Sie für den Minimal- und den Maximalwert -1 übergeben, reduziert `SetProcessWorkingSetSize` den reservierten physikalischen Arbeitsspeicher für den angegebenen Prozess kurzzeitig auf 0. Die CLR und Windows sorgen aber natürlich dafür, dass der reservierte Speicher sofort wieder auf einen für den aktuellen Zustand ausreichenden Betrag erhöht wird. Damit erreichen Sie, dass der reservierte Arbeitsspeicher minimiert wird. Im Prinzip entspricht das Ergebnis eines Aufrufs von `SetProcessWorkingSetSize` mit -1 an den beiden letzten Argumenten dem Minimieren und dem nachfolgenden Wiederherstellen des Hauptfensters einer Anwendung. Wenn Sie dies einmal ausprobieren, während Sie den Spei-

cherverbrauch der Anwendung im Taskmanager beobachten, werden Sie feststellen, dass dieser nach dem Minimieren und Wiederherstellen in der Regel enorm verkleinert wird.

`SetProcessWorkingSetSize` wird folgendermaßen deklariert:

```
[System.Runtime.InteropServices.DllImport("kernel32.dll")]
private static extern bool SetProcessWorkingSetSize(
   IntPtr procHandle, int min, int max);
```

Listing 139 zeigt eine Methode, die den reservierten Arbeitsspeicher für die aktuelle Anwendung auf ein Minimum reduziert. Diese Methode, die den Import der Namensräume `System` und `System.Diagnostics` erfordert, versucht zunächst die nicht mehr referenzierten Objekte freizugeben. Erst danach wird die `SetProcessWorkingSetSize`-Funktion aufgerufen um den reservierten Arbeitsspeicher zu reduzieren. Da diese Funktion nur in der NT-Linie der Microsoft-Betriebssysteme zur Verfügung steht, überprüft `ReduceMemoryUsage`, ob die aktuelle Plattform die Windows-32-Bit-NT-Plattform ist.

```
public static void ReduceMemoryUsage()
{
   // Garbage Collection erzwingen um freie Objekte zu zerstören
   GC.Collect();

   // Auf noch arbeitende Finalisierer warten
   GC.WaitForPendingFinalizers();

   // Den reservierten Arbeitsspeicher minimieren
   if (Environment.OSVersion.Platform == PlatformID.Win32NT)
   {
      SetProcessWorkingSetSize(
         Process.GetCurrentProcess().Handle, -1, -1);
   }
}
```

Listing 139: Minimieren des von der aktuellen Anwendung reservierten Arbeitsspeichers

Achtung

Sie sollten vorsichtig mit `ReduceMemoryUsage` umgehen. Zunächst kostet der Aufruf der `Collect`-Methode des Garbage Collectors unter Umständen einiges an Zeit. Auch das nachfolgende Minimieren des reservierten Speichers ist natürlich zeitaufwändig. Außerdem sollten Sie diese Funktion immer nur dann aufrufen, wenn Ihre Anwendung gerade nicht weiter beschäftigt ist. Ansonsten könnte der Aufruf zu unerwarteten Nebeneffekten führen. Achten Sie also darauf, dass nicht gerade ein Arbeits-Thread parallel läuft, während Sie `ReduceMemoryUsage` aufrufen. Gegebenenfalls rufen Sie diese Methode nicht automatisch auf, sondern überlassen dies dem Anwender, zum Beispiel über einen Menüpunkt. Wenn Ihre Anwendung eigene Threads einsetzt, sollten Sie diese in einer Auflistung verwalten, vor dem Aufruf von `ReduceMemoryUsage` durchgehen und über die `IsAlive`-Eigenschaft überprüfen, ob einer Ihrer Threads gerade noch ausgeführt wird. In diesem Fall sollten Sie `ReduceMemoryUsage` nicht aufrufen. In dem Beispiel zu diesem Rezept finden Sie eine entsprechende Programmierung.

86 Konfigurationsdaten in der .config-Datei verwalten

Wie Sie sicher wissen, kann eine Anwendung in einer XML-Datei, deren Name dem Dateinamen der Anwendung mit angehängter Endung *.config* entspricht und die im Ordner der Anwendung gespeichert ist, Konfigurationsdaten verwalten.

Diese Datei ist grundlegend folgendermaßen aufgebaut:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

</configuration>
```

Listing 140: Eine Minimal-Konfigurationsdatei

Im `configuration`-Element können verschiedene weitere Konfigurations-Elemente angelegt werden. Dabei können Sie die in der Datei *machine.config* vordefinierte Konfiguration für die Anwendung (in der Regel teilweise) überschreiben bzw. neu definieren. Dies soll hier allerdings kein Thema sein.

Zusätzlich dazu können Sie in der *.config*-Datei Ihre eigenen Konfigurationsdaten verwalten. Visual Studio 2008 macht diese Aufgabe leicht.

In Visual Studio 2008 wählen Sie dazu in den Eigenschaften des Projekts das Register EINSTELLUNGEN und stellen dort die Konfigurationswerte ein, die Sie benötigen. Dabei können Sie neben dem Namen auch den Datentyp einstellen und festlegen, welchen Geltungsbereich (Scope) die Einstellung besitzt.

Der Geltungsbereich »Application« bedeutet, dass diese Einstellung ausschließlich in der *.config*-Datei verwaltet wird, die im Anwendungsverzeichnis gespeichert ist. Solche Einstellungen können nur gelesen (also nicht in der Anwendung mit neuen Werten beschrieben) werden. Der Geltungsbereich »User« bewirkt, dass der Wert der Einstellung im Programm auch geändert und zurückgeschrieben werden kann. Das Zurückschreiben erfolgt per Voreinstellung allerdings nicht in die *.config*-Datei im Anwendungsverzeichnis, sondern in eine spezielle Konfigurationsdatei im Verzeichnis für die Anwendungsdaten des aktuellen Benutzers[1]. Lediglich die Default-Werte der Einstellung, die gelesen werden, wenn noch keine benutzerspezifische Konfigurationsdatei vorhanden ist, können in der *.config*-Datei im Anwendungsverzeichnis verwaltet werden.

1. Die Regeln zum Erhalt des »Designed for Windows XP«-Logo für eine Anwendung besagen u. a., dass Anwendungen Konfigurationsdaten nicht im Anwendungsverzeichnis speichern dürfen. Solche Daten müssen im Anwendungdaten-Verzeichnis des lokalen aktuellen Benutzers (normalerweise ist das das Verzeichnis *C:\Dokumente und Einstellungen\<Benutzername>\Lokale Einstellungen\Anwendungsdaten*), im äquivalenten Verzeichnis für »Roaming«-Benutzer (»wandernde« Benutzer) oder im Verzeichnis für alle Benutzer gespeichert werden.

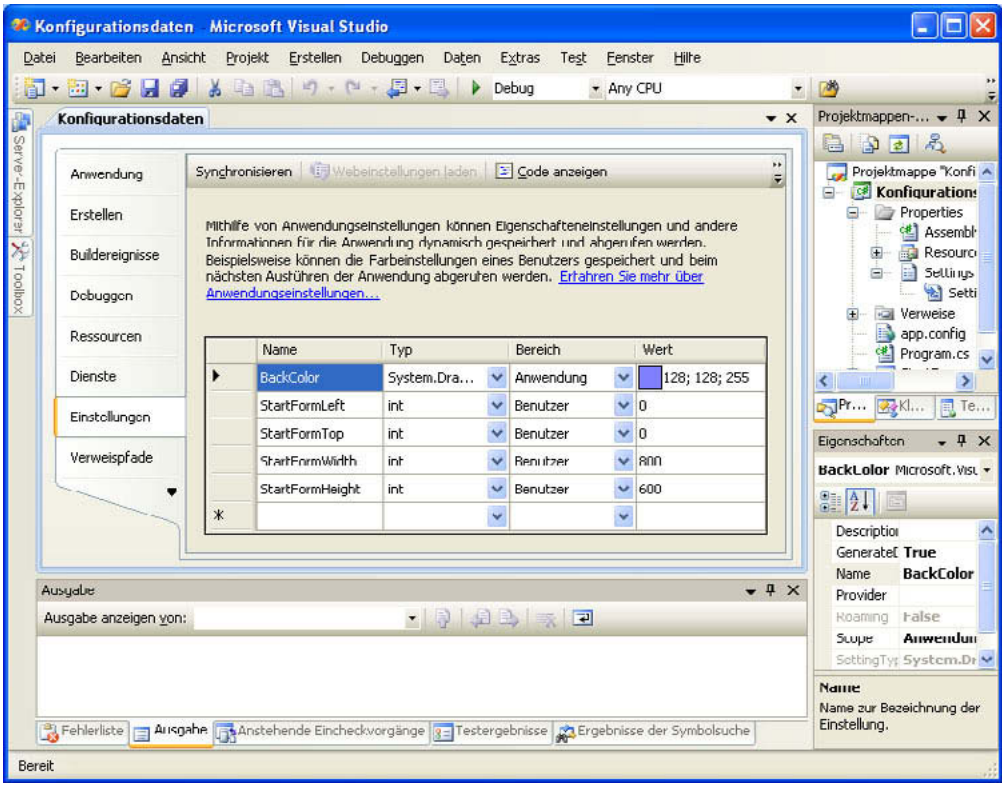


Abbildung 57: Bearbeiten von Konfigurationswerten in Visual Studio

Wenn Sie die geänderte Konfiguration speichern, passt Visual Studio die Konfigurationsdatei an und erzeugt eine Klasse `Settings` im Ordner *Properties*. Über eine Instanz dieser Klasse können Sie die Konfigurationsdaten typsicher lesen. Die Default-Instanz erreichen Sie über die `Default`-Eigenschaft der `Settings`-Klasse:

```
// Hintergrundfarbe des Formulars einlesen
this.BackColor = Properties.Settings.Default.BackColor;

// Position und Größe des Formulars auslesen
this.Left = Properties.Settings.Default.StartFormLeft;
this.Top = Properties.Settings.Default.StartFormTop;
this.Width = Properties.Settings.Default.StartFormWidth;
this.Height = Properties.Settings.Default.StartFormHeight;
```

Listing 141: Lesen von Konfigurationsdaten, die über eine Settings-Klasse verwaltet werden

Diejenigen Einstellungen, die den Geltungsbereich »User« besitzen, können Sie mit neuen Werten versehen und die geänderte Konfiguration über die `Save`-Methode speichern:

```
// Position und Größe des Formulars in der Konfiguration ablegen
Properties.Settings.Default.StartFormLeft = this.Left;
Properties.Settings.Default.StartFormTop = this.Top;
Properties.Settings.Default.StartFormWidth = this.Width;
Properties.Settings.Default.StartFormHeight = this.Height;

// Konfiguration speichern
try
{
   Properties.Settings.Default.Save();
}
catch (Exception ex)
{
   MessageBox.Show("Fehler beim Speichern der Konfiguration: " + ex.Message,
      Application.ProductName, MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Listing 142: Speichern von Konfigurationsdaten, die über eine Settings-Klasse verwaltet werden

Hinweis

Die `Settings`-Klasse verwendet Provider für das Lesen und Speichern der Daten. Per Default wird eine Instanz der Klasse `LocalFileSettingsProvider` eingesetzt. Diese verwaltet Konfigurationsdaten mit dem Geltungsbereich »User« im Ordner für lokale Benutzer (normalerweise ist das unter XP der Ordner *C:\Dokumente und Einstellungen\<Benutzername>\Lokale Einstellungen\Anwendungsdaten* und unter Vista der Ordner *C:\Users\<Benutzername>\AppData\Local*). In diesem Ordner wird mit einer gewöhnungsbedürftigen Logik ein Unterordner erzeugt. Dabei wird zunächst ein Ordner angelegt, der dem Schema *Firmenname\Dateiname der Assembly* entspricht. Ist das Assembly-Attribut `AssemblyCompany` nicht (in der Datei *Properties\AssemblyInfo.cs*) angegeben, entfällt der Ordner *Firmenname*. In diesem Ordner wird ein Ordner angelegt, dessen Name etwas eigenartig erscheint. Er beginnt mit den Dateinamen der Assembly (wobei die Endung nur mit zwei Stellen ausgegeben wird). Dann folgt »_Url_« und danach ein verschlüsselt angegebener URL. In diesem Ordner ist schließlich ein Ordner angelegt, der die Versionsnummer der Anwendung trägt, und in diesem eine Datei *user.config* gespeichert, die die Konfigurationsdaten mit dem Geltungsbereich »User« verwaltet. Etwas verwirrend, aber so können Konfigurationsdaten eindeutig einer bestimmten Anwendung bzw. einer bestimmten Version einer Anwendung zugeordnet werden. Der Debugger von Visual Studio 2008 ruft übrigens beim Start einer Windowsanwendung eine spezielle *.exe*-Datei auf, die den Namen der Anwendungsassembly mit *.vshost.exe* am Ende trägt. Die Konfigurationsdaten für eine so gestartete Anwendung werden in einem separaten Ordner verwaltet, in dessen Namen *.vshos_Url_* statt *.exe_Url_* angegeben ist. So ist sichergestellt, dass Sie die Konfiguration einer bereits produktiv laufenden Anwendung beim Testen nicht überschreiben.

Achtung

Die Einstellungen werden also zum einen Assembly-spezifisch und zum anderen ortsspezifisch gespeichert. Für eine normale Anwendung (keine ClickOnce-Anwendung), bedeutet dies:

- dass die Einstellungen verloren gehen, wenn die Anwendung in einen anderen Ordner kopiert wird,
- dass die Einstellungen verloren gehen, wenn die Anwendungsdatei umbenannt wird oder in den Assembly-Attributen der Firmenname geändert wird
- und dass die Einstellungen bei jeder Aktualisierung der Anwendung (mit einer neuen Versionsnummer) verloren gehen.

Besonders der letzte Punkt ist sehr ärgerlich (aber es gibt eine Lösung dafür). Sie können von einem Anwender nicht erwarten, dass er bei jeder Aktualisierung der Anwendung alle persönlichen Einstellungen erneut vornimmt.

Das Aktualisierungs-Problem gilt aber nur für normale Anwendungen. Für eine Anwendung, die per ClickOnce installiert wird, gehen die Einstellungen bei einem automatischen Update nicht verloren. Der Grund dafür ist, dass bei der (automatischen) Aktualisierung einer ClickOnce-Anwendung implizit die Methode `Upgrade` der Default-`Settings`-Instanz aufgerufen wird. Diese Methode liest die Einstellungen der vorherigen Version der Anwendung, die am besten zu der aktuellen Version passt, und schreibt diese in den Ordner der aktuellen Version.

Das ist auch der (relativ unbekannte und schlecht dokumentierte) Trick um zu erreichen, dass nach einer Aktualisierung einer Anwendung die Einstellungen nicht verloren gehen: Rufen Sie die `Upgrade`-Methode der `Settings`-Instanz auf. Die Frage ist nur, wann Sie das machen sollten.

Tipp

Der Trick dazu ist, in den Einstellungen eine spezielle boolesche Einstellung mit dem Bereich »Benutzer«, vielleicht mit dem Namen *CallUpgrade*, zu verwalten. Die Voreinstellung setzen Sie auf `true`. Beim Start der Anwendung überprüfen Sie, ob *CallUpgrade* `true` ist, und rufen in diesem Fall `Upgrade` auf. *CallUpgrade* setzen Sie dann auf `false` und speichern die Einstellungen:

```
if (Settings.Default.CallUpgrade)
{
   Settings.Default.Upgrade();
   Settings.Default.CallUpgrade = false;
   Settings.Default.Save();
}
```

Listing 143: Der ultimative Trick zum Lesen der Benutzer-Einstellungen nach dem Aktualisieren einer (normalen) Anwendung

Hinweis

Etwas problematisch an dieser Speicherung von Konfigurationsdaten ist zudem, dass die Daten automatisch nur beim Start der Anwendung ausgelesen werden. Werden die Daten in der Konfigurationsdatei geändert, während die Anwendung läuft, sind die Änderungen zunächst nicht sichtbar, auch wenn die Anwendung die Daten erneut abfragt. Dies kann in Anwendungen, die permanent im Hintergrund laufen (z. B. Windows-Dienste) zu einem Problem werden. Sie können jedoch in den Fällen, in denen geänderte Einstellungen in der Anwendung sichtbar sein müssen, vor dem Lesen einfach die `Reload`-Methode der `Settings`-Instanz aufrufen. Diese Methode führt dazu, dass die Einstellungen neu eingelesen werden.

Hinweis

Sie können auch eigene Provider implementieren, indem Sie eine Klasse entwickeln, die von der Klasse `SettingsProvider` abgeleitet ist und die die `IApplicationSettingsProvider`-Schnittstelle implementiert. So könnten Sie z. B. einen Provider entwickeln, der die Daten in einer »richtigen« XML-Datei verwaltet. Der neue Provider wird dann über das `SettingsProvider`-Attribut in der Settings-(Designer-)Datei angegeben, was aber zumindest für den Visual-Studio-2008-Designer etwas unklar ist, denn dieser überschreibt die Einstellungen in dieser Datei gnadenlos bei allen Änderungen (dagegen hilft ein weiterer Klassen-Teil der partiellen Klasse, der die Klassen entsprechend attributiert). Leider hat Microsoft dieses wichtige Thema nur sehr spärlich dokumentiert und liefert keine Beispiele. Bisher schlugen meine Versuche, einen eigenen Provider zu implementieren, leider immer fehl. In Internet war zu dieser Zeit auch nichts darüber zu finden. Falls Sie einen funktionierenden eigenen Provider entwickeln oder finden, wäre ich über eine E-Mail dankbar und würde das Ganze dann im Erratum veröffentlichen.

87 Konfigurationsdaten aus dem appSettings-Element der .config-Datei lesen

Neben der Möglichkeit, eine `Settings`-Instanz zu verwenden (siehe Rezept 86), können Sie Konfigurationsdaten aus der Konfigurationsdatei der Anwendung auch auf die alte (.NET-1.1-) Art verwalten. Dies besitzt in meinen Augen immer noch zwei kleine Vorteile: Zum einen sieht die Konfigurationsdatei wesentlich aufgeräumter aus als bei der Verwendung der `Settings`-Klasse und kann deswegen einfacher von Benutzern oder Administratoren verändert werden. Zum anderen können Sie ohne Probleme Konfigurationsdaten an einer zentralen Stelle verwalten, unabhängig davon, welcher Benutzer die Anwendung verwendet.

Die zu lesenden Daten müssen im Element `appSettings` unterhalb des `configuration`-Elements gespeichert sein:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appSettings>
      <add key="server" value="Zaphod" />
      <add key="userId" value="Trillian" />
```

Listing 144: Eine Beispiel-Konfigurationsdatei

```
      <add key="password" value="42" />
    </appSettings>
</configuration>
```

Listing 144: Eine Beispiel-Konfigurationsdatei (Forts.)

Die `add`-Elemente im `appSettings`-Element speichern anwendungsspezifische Daten. Diese Daten können Sie über die `AppSettings`-Eigenschaft der Klasse `ConfigurationManager` aus dem Namensraum `System.Configuration` auslesen:

```
string server = ConfigurationManager.AppSettings["Server"];
string userId = ConfigurationManager.AppSettings["UserId"];
string password = ConfigurationManager.AppSettings["password"];
```

Listing 145: Auslesen von Konfigurationsdaten aus dem appSettings-Element der .config-Datei

Das Programm benötigt dazu eine Referenz auf die Assembly *System.Configuration.dll*.

Hinweis

Wie ich es bereits in Rezept 86 beschrieben habe, leidet auch diese Art der Speicherung von Konfigurationsdateien an dem Problem, dass die Daten automatisch und nur beim Start der Anwendung ausgelesen werden. Werden die Daten in der Konfigurationsdatei geändert, während die Anwendung läuft, sind die Änderungen nicht sichtbar, auch wenn die Anwendung die Daten erneut abfragt. Als Lösung dieses Problems können Sie eine eigene XML-Datei für die Konfiguration verwenden, wie ich es im Rezept 90 zeige.

88 Externe Konfigurationsdateien lesen und schreiben

In einigen Fällen ist es notwendig, Konfigurationsdateien anderer Anwendungen zu lesen oder zu schreiben. Das ist z. B. der Fall, wenn Sie ein Setup für eine Anwendung erstellen, in dem der Anwender grundlegende Einstellungen vornehmen soll (die er z. B. in einem Windows-Installer-Setup über eine benutzerdefinierte Aktion eingibt). Das Setup-Programm muss dann die Konfigurationsdatei der installierten Anwendung mit den vom Anwender eingegebenen Daten überschreiben.

Dazu verwenden Sie die `ConfigurationManager`-Klasse (aus dem Namensraum `System.Configuration`), die es ermöglicht, alle Arten von .NET-Konfigurationsdateien (u. a. auch die *machine.config*) zu bearbeiten. Leider ist die Anwendung der `ConfigurationManager` für das Schreiben anderer als der eigenen Konfigurationsdateien zum einen nicht besonders gut dokumentiert und besitzt zum anderen zwei Fallstricke.

Dieses Rezept zeigt aber, wie Sie die Anwendungskonfiguration einer anderen Anwendung überschreiben können. Es arbeitet mit der folgenden Konfiguration, die einen Verbindungsstring, die anwendungsspezifische Einstellung »InstallationDate« und den Defaultwert der benutzerdefinierten Einstellung »DataFolder« verwaltet:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
```

```
        <sectionGroup name="applicationSettings" ... >
            <section name="Demo.Properties.Settings" ... />
        </sectionGroup>
        <sectionGroup name="userSettings" ... >
            <section name="Demo.Properties.Settings" ... />
        </sectionGroup>
    </configSections>


   <connectionStrings>
      <add name="DataConnection" connectionString=""/>
   </connectionStrings>
    <applicationSettings>
        <Demo.Properties.Settings>
            <setting name="InstallationDate" serializeAs="String">
                <value/>
            </setting>
        </Demo.Properties.Settings>
    </applicationSettings>
    <userSettings>
        <Demo.Properties.Settings>
            <setting name="DataFolder" serializeAs="String">
                <value/>
            </setting>
        </Demo.Properties.Settings>
    </userSettings>
</configuration>
```

Hinweis

Das Rezept zeigt lediglich, wie Sie die Konfiguration ändern, die im Anwendungsordner der anderen Anwendung verwaltet wird. Das Ändern einer benutzerspezifischen Konfigurationsdatei ist wesentlich komplexer, weil diese in einem speziellen Unterordner im Ordner der Anwendungsdaten für wandernde Benutzer (Roaming user) verwaltet wird. Ich denke aber nicht, dass das Schreiben in eine benutzerspezifische Konfigurationsdatei einer anderen Anwendung notwendig sein sollte, da Sie die Defaulteinstellungen der benutzerspezifischen Werte in der Konfiguration der Anwendung ändern können (wie dies in diesem Rezept auch realisiert ist).

Um die Anwendungs-Konfigurationsdatei einer anderen Anwendung zu bearbeiten, rufen Sie die statische Methode `OpenExeConfiguration` auf, der Sie den Pfad zu der *.exe*-Datei der anderen Anwendung übergeben. Diese Methode gibt ein `Configuration`-Objekt zurückgibt, über das Sie Zugriff auf die Konfiguration erhalten. Dabei arbeiten Sie immer mit Sektionen. Die Sektion für die Verbindungsstrings erreichen Sie z. B. über die `ConnectionStrings`-Eigenschaft, die ein `ConnectionStringsSection`-Objekt referenziert. Über dessen Eigenschaft `ConnectionStrings` können Sie die Verbindungsstrings lesen und schreiben.

Achtung

Hier wartet der erste Fallstrick: Wenn Sie lediglich einen vorhandenen Verbindungsstring neu definieren, indem Sie dessen Einstellungen überschreiben, führt dies nicht unbedingt [2] dazu, dass Ihre Änderungen später auch in die Konfigurationsdatei geschrieben werden. Das Problem liegt darin, dass der Konfigurations-Manager beim Ändern einer vorhandenen Einstellung grundsätzlich nicht erkennt, dass diese geändert wurde [3]. Sie sollten immer ein neues `ConnectionStringSettings`-Objekt erzeugen und dieses der `ConnectionStrings`-Auflistung zuweisen. Im einfachsten Fall löschen Sie vorher alle vorhandenen und fügen das neue einfach an.

```
// Konfiguration für die Anwendungsdaten der anderen Anwendung öffnen
string appPath = Path.GetDirectoryName(
   Assembly.GetEntryAssembly().Location);
string exeFileName = "C:\\Demo\\Demo.exe";
Configuration configuration = ConfigurationManager.OpenExeConfiguration(
   exeFileName);

// Verbindungs-Einstellungen lesen
Console.WriteLine("Alte Verbindungsstring-Einstellung:");
ConnectionStringsSection csSection = configuration.ConnectionStrings;
if (csSection.ConnectionStrings["DataConnection"] != null)
{
   Console.WriteLine("Verbindung: " +
      csSection.ConnectionStrings["DataConnection"].ConnectionString);
}
else
{
   Console.WriteLine("Die Verbindungsstring-Einstellung " +
      "'DataConnection' existiert nicht");
}

// Verbindungs-Einstellung neu definieren
configuration.ConnectionStrings.ConnectionStrings.Clear();
configuration.ConnectionStrings.ConnectionStrings.Add(
   new ConnectionStringSettings("DataConnection",
   "Server=.\\SQLEXPRESS;Database=Demo;User=sa;Password=Galaxy"));
```

Listing 146: Lesen und Schreiben von Verbindungsstring-Einstellungen der Konfigurationsdatei einer anderen Anwendung

2. Dies hängt davon ab, mit welchem *saveMode* Sie die `Save`-Methode aufrufen.
3. Das liegt daran, dass eine Einstellung ein komplexes Objekt ist. Wenn eine Eigenschaft dieses separaten Objekts geändert wird, erhält der Konfigurationsmanager, der lediglich eine Referenz auf das Einstellungsobjekt hält, darüber keine Information.

Um die anwendungsspezifischen Einstellungen zu schreiben, benötigen Sie zunächst eine Referenz auf die Sektions-Gruppe, die diese Einstellungen verwaltet. Über diese ermitteln Sie dann eine Referenz auf die Sektion:

```
// Die Sektion einlesen, die die anwendungsspezifische
// Konfiguration enthält
ConfigurationSectionGroup applicationSettings =
   configuration.SectionGroups["applicationSettings"];
ClientSettingsSection settingSection =
   applicationSettings.Sections["Demo.Properties.Settings"]
   as ClientSettingsSection;
```

Listing 147: Ermitteln der Sektion für die anwendungsspezifischen Einstellungen

Leider können Sie dann nicht direkt auf die einzelnen Einstellungen zugreifen, die Sie über die `Settings`-Eigenschaft erreichen, da diese keinen Indexer bietet, dem Sie den Namen der Einstellung übergeben können. Sie müssen die einzelnen Einstellungen durchgehen und nach derjenigen suchen, die Sie verändern wollen.

Achtung

Und hier ist der zweite Fallstrick enthalten (der aber mit dem ersten zusammenhängt). Den Wert der Einstellung erhalten Sie über die Eigenschaft `Value`. `Value` referenziert ein `SettingValueElement`-Objekt. Dessen Eigenschaft `ValueXml` referenziert ein `XmlNode`-Objekt, das schließlich den Wert der Einstellung verwaltet. Leider können Sie nicht einfach nur die Eigenschaft `InnerText` dieses Objekts beschreiben, um den Wert zu ändern. In diesem Fall bekommt der Konfigurations-Manager die Änderung nicht mit und schreibt den geänderten Wert nicht in die Konfiguration.

Um dieses Problem zu lösen, müssen Sie der `ValueXml`-Eigenschaft ein neues `XmlNode`-Objekt zuweisen, das den geänderten Wert enthält. Da `XmlNode` keinen Konstruktor und auch keine Methode zur Erzeugung zur Verfügung stellt, müssen Sie zudem ein Dummy-`XmlDocument`-Objekt erzeugen, über das Sie die `XmlNode`-Instanz erzeugen können.

```
// XmlDocument-Dummy-Dokument zur Erstellung der XmlNode-Instanzen erzeugen
XmlDocument dummyDoc = new XmlDocument();

// Alle SettingElement-Instanzen durchgehen, um die Einstellungen
// zu finden, die geändert werden sollen
foreach (SettingElement settingElement in settingSection.Settings)
{
   if (settingElement.Name == "InstallationDate")
   {
      // Die Einstellung ändern, indem ein neues XML-Element
      // geschrieben wird. Dies ist leider notwendig, da der
```

Listing 148: Schreiben von anwendungsspezifischen Einstellungen

```
      // Konfigurationsmanager ansonsten nicht erkennt, dass
      // diese Daten geändert wurden, und die Daten einfach
      // nicht in die Konfiguration schreibt.
      settingElement.Value.ValueXml = dummyDoc.CreateElement("value");
      settingElement.Value.ValueXml.InnerText = DateTime.Now.ToString("u");
   }
}
```

Listing 148: Schreiben von anwendungsspezifischen Einstellungen (Forts.)

Für die benutzerspezifischen Einstellungen (bei denen Sie die Defaultwerte ändern können) gehen Sie ähnlich vor:

```
// Die Sektion einlesen, die die benutzerspezifische Konfiguration enthält
ConfigurationSectionGroup userSettings =
   configuration.SectionGroups["userSettings"];
settingSection =
   userSettings.Sections["Demo.Properties.Settings"]
   as ClientSettingsSection;

// Alle SettingElement-Instanzen durchgehen, um die Einstellungen
// zu finden, die geändert werden sollen
foreach (SettingElement settingElement in settingSection.Settings)
{
   if (settingElement.Name == "DataFolder")
   {
      settingElement.Value.ValueXml = dummyDoc.CreateElement("value");
      settingElement.Value.ValueXml.InnerText = "C:\\Data";
   }
}
```

Listing 149: Schreiben der Defaultwerte von benutzerspezifischen Einstellungen

Schließlich speichern Sie Ihre Änderungen über die `Save`-Methode. Dieser Methode können Sie einen Wert der `ConfigurationSaveMode`-Aufzählung übergeben:

- `Minimal`: Bewirkt, dass nur Einstellungen geschrieben werden, die sich von den geerbten Werten unterscheiden
- `Full`: Bewirkt, dass alle Einstellungen (auch geerbte) in die Konfiguration geschrieben werden
- `Modified`: Bewirkt, dass alle geänderten Einstellungen in die Konfiguration geschrieben werden, auch wenn deren Wert mit dem Wert einer geerbten Einstellung identisch ist

Für unseren Fall ist `Modified` am besten geeignet. Dabei gehen eventuell vorhandene Einstellungen, die Sie nicht ändern, nicht verloren und die Konfigurationsdatei bleibt prinzipiell mit der vorhergehenden identisch. Da `Modified` die Voreinstellung ist (was allerdings nicht dokumentiert ist), können Sie `Save` auch ohne Argument aufrufen:

```
// Konfiguration speichern
configuration.Save();
```

Listing 150: Speichern der geänderten Konfiguration

89 Konfigurationsdaten in eigenen Abschnitten speichern

Die Konfigurationsdaten einer Anwendung können Sie zwar relativ einfach in den Anwendungs-Einstellungen (`Settings`) speichern, beim Entwickeln von Komponenten ist es aber häufig notwendig, diese Daten dem Anwender oder Programmierer über die normale Anwendungskonfigurationsdatei verfügbar zu machen. In diesem Fall macht es Sinn, die Konfigurationsdaten in einem eigenen Abschnitt der *.config*-Datei zu speichern, der wesentlich einfacher zu pflegen ist als das mittlerweile veraltete `appSettings`-Element.

Zur Verwaltung von Konfigurationsdaten in einer eigenen Sektion müssen Sie einen Handler entwickeln. Ein solcher implementiert die Schnittstelle `IConfigurationSectionHandler`, deren Methode `Create` für alle registrierten, benutzerdefinierten Konfigurationsabschnitte aufgerufen wird. Über das Argument `section`, das ein `XmlNode`-Objekt referenziert, erhalten Sie Zugriff auf das XML-Element, das die Daten verwaltet.

In der Anwendung müssen (eigentlich erst später) die neuen Abschnitte in der Konfigurationsdatei registriert werden. Das folgende Listing registriert die Abschnitte *system* und *database* im Element *codebook*:

```
<?xml version="1.0" encoding="utf-8" ?>

<configuration>

   <!-- Bekanntmachung der eigenen Konfigurations-Sektionen -->
   <configSections>
      <sectionGroup name="codebook">
         <section
           name="database"
           type="Addison_Wesley.Codebook.Configuration.
           ConfigSectionHandler, Config-Handler"/>
         <section
           name="system"
           type="Addison_Wesley.Codebook.Configuration.
           ConfigSectionHandler, Config-Handler"/>
      </sectionGroup>
   </configSections>
```

Listing 151: Registration eigener Konfigurations-Abschnitte

Das `sectionGroup`-Element legt über das Attribut `name` den Namen des XML-Elements fest, das die Konfigurationsdaten verwaltet. Im Element `section` werden die einzelnen Abschnitte definiert. Das Attribut `name` bestimmt dabei den Namen des XML-Elements. Das Attribut `type` legt fest, welche Klasse für das Handling dieser Elemente zuständig ist. Dabei geben Sie zuerst den

vollen Namen der Klasse (inklusive Namensraum) und durch ein Komma getrennt die Assembly an, in der diese Klasse gespeichert ist (im Beispiel *Config-Handler*). Das `sectionGroup`-Element kann (natürlich) auch komplett in der Haupt-Konfigurationsdatei *machine.config* gespeichert werden, was u. U. Sinn macht, wenn es sich um Konfigurationsdaten handelt, die Sie öfter verwenden. Dann müssen Sie aber auch dafür sorgen, dass diese Eintragung in der Maschinenkonfiguration auf den Computern eingetragen wird, auf dem Ihr Programm ausgeführt werden soll.

In der Konfigurationsdatei folgen dann Ihre eigenen Konfigurationssektionen:

```
    <codebook>
      <database>
        <server>Zaphod</server>
        <userId>Trillian</userId>
        <password>42</password>
      </database>

      <system>
        <lastAccess>01.01.2003 10:00:00</lastAccess>
      </system>
    </codebook>

</configuration>
```

Listing 152: Eigene Konfigurations-Abschnitte

Zur Verwaltung der später eingelesenen Daten benötigen Sie zunächst für jeden benutzerdefinierten Konfigurationsabschnitt eine Klasse.

```
/* Klasse zur Speicherung der Datenbank-Konfiguration */
public class DatabaseSection
{
   public string Server;
   public string UserId;
   public string Password;

   internal DatabaseSection(string server, string userId, string password)
   {
      this.Server = server;
      this.UserId = userId;
      this.Password = password;
   }
}

/* Klasse zur Speicherung der System-Konfiguration */
public class SystemSection
```

Listing 153: Klassen zur Speicherung der eingelesenen Daten

```
{
   public string LastAccess;

   internal SystemSection(string lastAccess)
   {
      this.LastAccess = lastAccess;
   }
}
```

Listing 153: Klassen zur Speicherung der eingelesenen Daten (Forts.)

Zum Lesen der eigenen Konfigurationsabschnitte müssen Sie eine Klasse entwickeln, die die Schnittstelle `IConfigurationSectionHandler` implementiert. Der `Create`-Methode dieser Klasse werden später Informationen über das übergeordnete Element, ein Konfigurationskontext (nur für ASP.NET-Anwendungen, wo dieser den virtuellen Pfad der Konfigurationsdatei enthält) und ein `XmlNode`-Objekt übergeben, das das XML-Element mit den Konfigurationsdaten enthält. Über dieses Objekt können Sie die Daten lesen.

Die Methode `Create` in Listing 154 fragt zunächst über die `Name`-Eigenschaft ab, um welchen Abschnitt es sich handelt. Dann werden die entsprechenden Daten eingelesen und damit eine Instanz der Klasse erzeugt, die für diese Daten vorgesehen ist. Diese Instanz gibt `Create` dann zurück.

Etwas problematisch ist, dass Konfigurationsdaten auch fehlen können. `Create` fragt deshalb vor dem Lesen ab, ob die einzelnen Unterelemente vorhanden sind. Für den Fall, dass ein unbekannter Konfigurationsabschnitt registriert wurde, generiert die Methode eine Ausnahme:

```
/* Klasse, die den Konfigurations-Sektions-Handler implementiert */
public class ConfigSectionHandler: IConfigurationSectionHandler
{
   /* Implementierung der Create-Methode */
   public object Create(object parent, object configContext, XmlNode section)
   {
      if (section.Name == "database")
      {
         // Einlesen der Unterelemente der Sektion
         XmlNode subNode;
         string server = null;
         subNode = section.SelectSingleNode("server");
         if (subNode != null)
         {
            server = subNode.InnerText;
         }

         string userId = null;
         subNode = section.SelectSingleNode("userId");
```

Listing 154: Handler für eigene Konfigurationsabschnitte

```
            if (subNode != null)
            {
               userId = subNode.InnerText;
            }

            string password = null;
            subNode = section.SelectSingleNode("password");
            if (subNode != null)
            {
               password = subNode.InnerText;
            }

            // Neue DatabaseConfig-Instanz zurückgeben
            return new DatabaseSection(server, userId, password);

         }
         else if (section.Name == "system")
         {
            // Einlesen der Unterelemente der Sektion
            XmlNode subNode;
            string lastAccess = null;
            subNode = section.SelectSingleNode("lastAccess");
            if (subNode != null)
            {
               lastAccess = subNode.InnerText;
            }

            // Neue SystemConfig-Instanz zurückgeben
            return new SystemSection(lastAccess);
         }
         else
         {
            // Unbekannte Sektion: Ausnahme werfen
            throw new ConfigurationErrorsException("Unbekannte Konfigurations-" +
               "Sektion '" + section.Name + "'");
         }
      }
   }
```

Listing 154: Handler für eigene Konfigurationsabschnitte (Forts.)

Nun müssen Sie die Konfigurationsdaten in der von Ihnen entwickelten Komponente lediglich noch über die Methode `ConfigurationSettings.GetConfig` einlesen, indem Sie den relativen Pfad zum Konfigurationsabschnitt angeben. `GetConfig` gibt das in `Create` erzeugte Objekt zurück, wenn der Abschnitt gefunden wird. Im anderen Fall wird `null` zurückgegeben. So können Sie beim Einlesen überprüfen, ob der Abschnitt vorhanden ist. Das Beispiel-Programm in

Listing 155 implementiert dies in Form einer einfachen Konsolenanwendung. Sie müssen die Namensräume `System` und `System.Configuration` importieren, um dieses Programm kompilieren zu können.

```
DatabaseSection databaseSection = null;
SystemSection systemSection = null;

try
{
   // Einlesen der Konfiguration
   systemSection = (SystemSection)ConfigurationManager.GetSection(
      "codebook/system");
   databaseSection = (DatabaseSection)ConfigurationManager.GetSection(
      "codebook/database");

   // Überprüfen, ob der System-Konfigurationsabschnitt eingelesen wurde
   if (systemSection != null)
   {
      // Ausgeben der Konfigurationsdaten
      Console.WriteLine("LastAccess: {0}", systemSection.LastAccess);
   }
   else
   {
      Console.WriteLine("Die Systemkonfiguration konnte nicht " +
         "eingelesen werden");
   }

   // Überprüfen, ob der Datenbank-Konfigurationsabschnitt eingelesen wurde
   if (databaseSection != null)
   {
      // Ausgeben der Konfigurationsdaten
      Console.WriteLine("Server: {0}", databaseSection.Server);
      Console.WriteLine("UserId: {0}", databaseSection.UserId);
      Console.WriteLine("Passwort: {0}", databaseSection.Password);
   }
   else
   {
      Console.WriteLine("Die Datenbankkonfiguration konnte nicht " +
         "eingelesen werden");
   }
}
catch (Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

Listing 155: Einlesen eigener Konfigurationsabschnitte in einer Konsolenanwendung

90 Anwendungskonfiguration über eine eigene XML-Datei

Die Anwendungskonfiguration über die Standard-Konfigurationsdatei einer Anwendung (siehe die vorhergehenden Rezepte) ist zwar recht einfach, besitzt aber leider auch einige Nachteile. Der wesentliche Nachteil ist, dass die Verwaltung der Daten in der Konfigurationsdatei für einen unbedarften Anwender u. U. etwas undurchsichtig ist.

Die im Folgenden beschriebene Klasse `Config` ermöglicht hingegen die Verwaltung von Konfigurationsdaten in einer einfachen XML-Datei, die beispielsweise folgendermaßen aussieht:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<config>
  <Database>
    <Server>Zaphod</Server>
    <UserId>Trillian</UserId>
    <Password>42</Password>
  </Database>
  <System>
    <StartTime>10:00</StartTime>
    <LastAccess>12.04.2003 14:03:36</LastAccess>
  </System>
</config>
```

Listing 156: Eine eigene Konfigurationsdatei

Die einzelnen Einstellungen sind in normalen XML-Elementen gespeichert und werden über übergeordnete Elemente in Sektionen eingeteilt. Eine Instanz der Klasse `Config` ermöglicht über eine Auflistung den Zugriff auf die einzelnen Sektionen. Innerhalb jeder Sektion steht eine weitere Auflistung für die Einstellungen zur Verfügung. Für jede zu speichernde Einstellung verwaltet die Klasse den Namen, den Wert, einen Defaultwert und eine Information darüber, ob die Einstellung beim Lesen der Konfigurationsdatei in dieser gefunden wurde.

Zur Realisation dieser Idee ist zunächst eine Klasse `Setting` zur Speicherung der einzelnen Einstellungen notwendig.

```
using System;
using System.IO;
using System.Xml;
using System.Collections.Generic;

...

public class Setting
{
   /* Der Name der Einstellung */
   public string Name;
```

Listing 157: Klasse zur Speicherung einer Einstellung

```
   /* Der Wert der Einstellung */
   public string Value;

   /* Der Defaultwert für das Lesen */
   public string DefaultValue;

   /* Gibt an, ob die Einstellung in der Datei gefunden wurde */
   public bool WasInFile;

   /* Konstruktor */
   public Setting(string name, string defaultValue)
   {
      this.Name = name;
      this.DefaultValue = defaultValue;
   }
}
```

Listing 157: Klasse zur Speicherung einer Einstellung (Forts.)

Die Klasse `Settings` implementiert eine Auflistung vom Typ `Dictionary` und ist spezialisiert auf Instanzen von `Setting`. `Settings` wird von der generischen `Dictionary`-Klasse, die als Grundlage für die Erzeugung von typsicheren assoziativen Auflistungen dient, abgeleitet. Die Methode `Add` dient dem Hinzufügen einer neuen Einstellung.

```
public class Settings : Dictionary<string, Setting>
{
   /* Methode zum Hinzufügen einer neuen Einstellung */
   public void Add(string settingName, string defaultValue)
   {
      this.Add(settingName,
         new Setting(settingName, defaultValue));
   }
}
```

Listing 158: Klasse zur Verwaltung eine Auflistung von Einstellungen

Die Klasse `Section` verwaltet die Daten einer Sektion. Neben dem Namen derselben enthält diese Klasse eine Eigenschaft `Settings` vom Typ der Einstellungs-Auflistung. In dieser Auflistung sollen die Einstellungen verwaltet werden, die der Sektion zugeordnet sind. Damit die Auflistung automatisch zur Verfügung steht, wird die Instanz der Klasse `Settings` im Konstruktor erzeugt:

```
public class Section
{
   /* Der Name der Sektion */
   public string Name;

   /* Die Einstellungen der Sektion */
   public Settings Settings;

   /* Konstruktor */
   public Section(string sectionName)
   {
      this.Name = sectionName;
      this.Settings = new Settings();
   }
}
```

Listing 159: Klasse zur Speicherung einer Einstellungs-Sektion

Ähnlich der Auflistung zur Verwaltung von Einstellungen implementiert die Klasse `Sections` eine `Dictionary`-Auflistung zur Verwaltung mehrerer `Section`-Instanzen:

```
public class Sections: Dictionary<string, Section>
{
   /* Fügt der Auflistung ein neues Section-Objekt hinzu */
   public void Add(string name)
   {
      this.Add(name, new Section(name));
   }
}
```

Listing 160: Auflistungs-Klasse zur Speicherung von Einstellungs-Sektionen

Die eigentliche Haupt-Klasse `Config` verwaltet in einer privaten Eigenschaft den Dateinamen der Konfigurationsdatei und in einer öffentlichen Eigenschaft eine Instanz der Klasse `Sections`. Im Konstruktor wird der Dateiname übergeben und die `Sections`-Instanz erzeugt:

```
public class Config
{
   /* Speichert den Dateinamen der XML-Datei */
   private string fileName;

   /* Verwaltet die Sektionen */
   public Sections Sections;
```

Listing 161: Eigenschaften und Konstruktor der Klasse Config

```
/* Konstruktor */
public Config(string fileName)
{
   this.fileName = fileName;
   this.Sections = new Sections();
}
```

Listing 161: Eigenschaften und Konstruktor der Klasse Config (Forts.)

Die Methode `Load` ermöglicht das Lesen der Einstellungen aus der Konfigurationsdatei. Diese Methode erzeugt dazu zunächst eine Instanz der Klasse `XmlDocument` und lädt die über das Feld `fileName` angegebene XML-Datei. Für den Fall, dass beim Laden ein Fehler auftritt, fängt die Methode die möglichen Ausnahmen ab und erzeugt daraus direkt eine neue Ausnahme, allerdings mit einer genaueren Fehlermeldung. Danach geht `Load` alle Sektionen und in diesen alle Einstellungen durch und ermittelt über einen XPath-Ausdruck das XML-Element, das diese Einstellung speichert. Wird das Element gefunden, liest die Methode dessen Wert aus und speichert ihn im aktuellen `Setting`-Objekt. Um später auswerten zu können, ob Einstellungen erfolgreich gelesen wurden, wird das Feld `WasInFile` des `Setting`-Objekts entsprechend mit `true` oder `false` beschrieben. `Load` gibt `true` zurück, wenn alle Einstellungen gelesen werden konnten. Konnte auch noch eine Einstellung nicht gelesen werden, gibt diese Methode `false` zurück.

```
public bool Load()
{
   // Variable für den Rückgabewert
   bool returnValue = true;

   // XmlDocument-Objekt für die Einstellungs-Datei erzeugen
   XmlDocument xmlDoc = new XmlDocument();

   // Datei laden
   try
   {
      xmlDoc.Load(this.fileName);
   }
   catch (IOException ex)
   {
      throw new IOException("Fehler beim Laden der Konfigurationsdatei '" +
         this.fileName + "': " + ex.Message);

   }
   catch (XmlException ex)
   {
      throw new XmlException("Fehler beim Laden der Konfigurationsdatei '" +
         this.fileName + "': " + ex.Message, ex);
   }
```

Listing 162: Methode zum Lesen der Konfigurationsdatei

```
      // Alle Sektionen durchgehen und die Einstellungen einlesen
      foreach (Section section in this.Sections.Values)
      {
         // Alle Einstellungen der Sektion durchlaufen
         foreach (Setting setting in section.Settings.Values)
         {
            // Einstellung im XML-Dokument lokalisieren
            XmlNode settingNode = xmlDoc.SelectSingleNode(
               "/config/" + section.Name + "/" + setting.Name);
            if (settingNode != null)
            {
               // Einstellung gefunden
               setting.Value = settingNode.InnerText;
               setting.WasInFile = true;
            }
            else
            {
               // Einstellung nicht gefunden
               setting.Value = setting.DefaultValue;
               setting.WasInFile = false;
               returnValue = false;
            }
         }
      }

      // Ergebnis zurückmelden
      return returnValue;
   }
```

Listing 162: Methode zum Lesen der Konfigurationsdatei (Forts.)

Die Methode `Save` speichert die Konfigurationsdaten in die XML-Datei. Sie erzeugt dazu eine neue Instanz der Klasse `XmlDocument` und initialisiert diese mit dem Skelett des XML-Dokuments (wobei als kleiner Trick die `LoadXml`-Methode mit einem String aufgerufen wird, der genau dieses Skelett beinhaltet).

Dann geht `Save` die einzelnen Sektionen der Einstellungen durch, um innerhalb des XML-Dokuments für jede Sektion und für jede Einstellung ein entsprechendes Element zu erzeugen. Schließlich wird die Datei noch gespeichert, wobei natürlich eventuelle Ausnahmen abgefangen und mit einer genaueren Fehlermeldung weitergegeben werden.

```
   public void Save()
   {
      // XmlDocument-Objekt für die Einstellungs-Datei erzeugen
      XmlDocument xmlDoc = new XmlDocument();
```

Listing 163: Methode zum Speichern der Konfigurationsdatei

```
      // Skelett der XML-Datei erzeugen
      xmlDoc.LoadXml("<?xml version=\"1.0\" encoding=\"utf-8\" " +
         "standalone=\"yes\"?><config></config>");

      // Alle Sektionen durchgehen und die Einstellungen schreiben
      foreach (Section section in this.Sections.Values)
      {
         // Element für die Sektion erzeugen und anfügen
         XmlElement sectionElement = xmlDoc.CreateElement(section.Name);
         xmlDoc.DocumentElement.AppendChild(sectionElement);

         // Alle Einstellungen der Sektion durchlaufen
         foreach (Setting setting in section.Settings.Values)
         {
            // Einstellungs-Element erzeugen und anfügen
            XmlElement settingElement =
               xmlDoc.CreateElement(setting.Name);
            settingElement.InnerText = setting.Value;
            sectionElement.AppendChild(settingElement);
         }
      }

      // Datei speichern
      try
      {
         xmlDoc.Save(this.fileName);
      }
      catch (IOException ex)
      {
         throw new IOException("Fehler beim Speichern der " +
            "Konfigurationsdatei '" + this.fileName + "': " + ex.Message);

      }
      catch (XmlException ex)
      {
         throw new XmlException("Fehler beim Speichern der " +
            " Konfigurationsdatei '" + this.fileName + "': " +
            ex.Message, ex);
      }
   }
}
```

Listing 163: Methode zum Speichern der Konfigurationsdatei (Forts.)

Die beispielhafte Anwendung dieser Klasse zeigt das folgende Listing:

```
// Config-Instanz erzeugen
Config config = new Config(Path.Combine(Application.StartupPath,
   "Config.xml"));

// Konfigurations-Sektionen und Einstellungen definieren
config.Sections.Add("System");
config.Sections.Add("Database");
config.Sections["System"].Settings.Add("LastAccess", "00:00");
config.Sections["Database"].Settings.Add("Server", "(local)");

// Datei einlesen
config.Load();

// Ein Datum lesen und ausgeben
Setting setting = config.Sections["System"].Settings["LastAccess"];
if (setting.WasInFile == false)
{
   Console.WriteLine("Einstellung System/LastAccess nicht gefunden");
}
Console.WriteLine("LastAccess: {0}", setting.Value);

// Den Wert ändern
setting.Value = DateTime.Now.ToString();

// Datei speichern
config.Save();
```

Listing 164: Anwendung der Klasse Config zur Verwaltung von Anwendungs-Konfigurationsdate

Hinweis

Die in diesem Rezept vorgestellten Klassen könnten noch wesentlich erweitert werden. So könnte die `Setting`-Klasse zusätzlich den Datentyp der Einstellung und bei numerischen Daten einen Minimal- und Maximalwert verwalten. Beim Einlesen bzw. Schreiben der Daten könnten diese dann auf den korrekten Datentyp und die Einhaltung der Wertgrenzen überprüft werden. Eine weitere sinnvolle Idee ist die automatische Erzeugung eines Formulars über eine weitere Methode, in dem der Anwender die Einstellungen bearbeiten kann. Das Formular könnte die einzelnen Sektionen und deren Einstellungen in einem `TreeView`-Steuerelement darstellen. Wählt der Anwender eine Einstellung aus, wird diese in einem separaten Bereich des Formulars zur Bearbeitung in einer `TextBox` dargestellt.

91 Daten aus der Registry lesen und in die Registry schreiben

Über die Klasse `Registry` aus dem Namensraum `Microsoft.Win32` können Sie Daten aus der Windows-Registry lesen, Daten manipulieren, neue Daten in vorhandenen Registry-Schlüsseln erzeugen und neue Schlüssel erzeugen.

Basis für die Arbeit mit der Registry ist die Klasse `RegistryKey`, die einen (Unter-) Schlüssel der Registry repräsentiert. Über verschiedene Methoden können Sie den Schlüssel bearbeiten. Die Methode `GetValue` liest z. B. den Wert eines Eintrags in einem Schlüssel, über die Methode `SetValue` können Sie einen Wert schreiben. Die Methode `CreateSubKey` ermöglicht das Erzeugen eines Unterschlüssels, über `DeleteSubKey` können Sie einen Unterschlüssel löschen. Die `OpenSubKey`-Methode öffnet einen Schlüssel, der einem anderen Schlüssel untergeordnet ist.

Die Wurzel-Schlüssel der Registry erreichen Sie über die statischen Eigenschaften `ClassesRoot (HKEY_CLASSES_ROOT)`, `CurrentConfig (HKEY_CURRENT_CONFIG)`, `CurrentUser (HKEY_CURRENT_USER)`, `DynData (HKEY_DYN_DATA)`, `LocalMachine (HKEY_LOCAL_MACHINE)`, `PerformanceData (HKEY_PERFORMANCE_DATA)` und `Users` (`HKEY_USERS`). Da diese Eigenschaften ein `RegistryKey`-Objekt referenzieren, können Sie über die `OpenSubKey`-Methode zu einem Unterschlüssel navigieren. Die einzelnen Schlüsselnamen des (relativen) Pfads geben Sie durch Backslashes getrennt an.

Das folgende Beispiel liest den Wert *DOC-PATH* des Schlüssels *HKEY_CURRENT_USER\Software\Microsoft\Office\11.0\Word\Options*. Das Programm benötigt den Import der Namensräume `System` und `Microsoft.Win32`.

```
RegistryKey regKey = Registry.CurrentUser.OpenSubKey(
   @"Software\Microsoft\Office\11.0\Word\Options");

string wordDocPath = null;
if (regKey != null)
{
    object regValue = regKey.GetValue("DOC-PATH");
    if (regValue != null)
    {
        wordDocPath = regValue.ToString();
        Console.WriteLine(wordDocPath);
    }
    else
    {
        Console.WriteLine("Wert nicht gefunden");
    }
}
else
{
    Console.WriteLine("Schlüssel nicht gefunden");
}
```

Listing 165: Lesen eines Registry-Eintrags

Beim Schreiben sollten Sie für jeden Unterschlüssel überprüfen, ob dieser existiert, und den Schlüssel gegebenenfalls anlegen, bis Sie am letzten Schlüssel angelangt sind und dort den Wert schreiben können. Da das Ganze für die Praxis ein wenig kompliziert ist, habe ich eine Klasse `RegistryUtils` entwickelt, die den Registry-Zugriff erleichtert.

Den Methoden dieser Klasse wird der Registry-Pfad in der unter Windows üblichen Form übergeben, wobei der Wurzel-Schlüssel allerdings separat übergeben wird. Ein solcher Pfad sieht dann z. B. so aus:

Software\Microsoft\Office\10.0\Word\Options\DOC-PATH

Um den Wurzel-Schlüssel angeben zu können, habe ich zunächst eine Aufzählung implementiert:

```
public enum RegistryRootKeys
{
    HKEY_CLASSES_ROOT,
    HKEY_CURRENT_CONFIG,
    HKEY_CURRENT_USER,
    HKEY_DYN_DATA,
    HKEY_LOCAL_MACHINE,
    HKEY_PERFORMANCE_DATA,
    HKEY_USERS
}
```

Listing 166: Aufzählung zur Angabe der Registry-Wurzel-Schlüssel

Die Methoden der Klasse `RegistryUtils` fangen mögliche Ausnahmen (z. B. die Ausnahme vom Typ `UnauthorizedAccessException` bei einem unautorisierten Zugriff) nicht ab, sodass diese an den Aufrufer weitergereicht werden.

Die erste (private) Methode `GetRegistryRootKey` gibt eine Referenz auf das zu einem Wert der `RegistryRootKeys`-Aufzählung passende `RegistryKey`-Objekt zurück. Diese Methode wird in den anderen Methoden der Klasse aufgerufen.

```
public class RegistryUtils
{
   private static RegistryKey GetRegistryRootKey(RegistryRootKeys rootKey)
   {
      // Ermitteln des Registry-Wurzel-Schlüssels
      RegistryKey regKey = null;
      switch (rootKey)
      {
         case RegistryRootKeys.HKEY_CLASSES_ROOT:
            regKey = Registry.ClassesRoot;
            break;
```

Listing 167: Methode zum Ermitteln einer Referenz auf ein Wurzel-RegistryKey-Objekt

```
            case RegistryRootKeys.HKEY_CURRENT_CONFIG:
               regKey = Registry.CurrentConfig;
               break;

            case RegistryRootKeys.HKEY_CURRENT_USER:
               regKey = Registry.CurrentUser;
               break;

            case RegistryRootKeys.HKEY_DYN_DATA:
               regKey = Registry.DynData;
               break;

            case RegistryRootKeys.HKEY_LOCAL_MACHINE:
               regKey = Registry.LocalMachine;
               break;

            case RegistryRootKeys.HKEY_PERFORMANCE_DATA:
               regKey = Registry.PerformanceData;
               break;

            case RegistryRootKeys.HKEY_USERS:
               regKey = Registry.Users;
               break;
         }

         return regKey;
      }
```

Listing 167: Methode zum Ermitteln einer Referenz auf ein Wurzel-RegistryKey-Objekt (Forts.)

Die Methode zum Lesen eines Registry-Eintrags (`ReadValue`) ruft zunächst `GetRegistryRootKey` auf um das zum übergebenen Wurzel-Schlüssel-Wert passende `RegistryKey`-Objekt zu referenzieren. Über diese Referenz ruft `ReadValue` dann die `OpenSubKey`-Methode unter der Übergabe des Registry-Pfades auf. Wird der Schlüssel gefunden, liest die Methode den Wert des übergebenen Wertnamens. Im Erfolgsfall liefert `ReadValue` den gelesenen Wert zurück. Wird der Wert nicht gefunden, gibt diese Methode den übergebenen Defaultwert zurück.

```
public static object ReadValue(RegistryRootKeys rootKey, string keyPath,
   string valueName, object defaultValue)
{
   // Schlüssel ermitteln
   RegistryKey regKey = GetRegistryRootKey(rootKey).OpenSubKey(keyPath);
   // Wert auslesen, wenn der Unterschlüssel gefunden wurde
   object regValue = defaultValue;
   if (regKey != null)
```

Listing 168: Methode zum Lesen eines Registry-Eintrags

```
    {
        regValue = regKey.GetValue(valueName);
    }

    return regValue;
}
```

Listing 168: Methode zum Lesen eines Registry-Eintrags (Forts.)

`ReadValue` gibt (entsprechend der `GetValue`-Methode des `RegistryKey`-Objekts) einen String zurück, wenn der Registry-Eintrag vom Typ `REG_SZ` oder `REG_EXPAND_SZ` ist. Bei einem mehrteiligen String (`REG_MULTI_SZ`) wird ein String-Array mit den einzelnen Teilen zurückgegeben. Ein numerischer Wert (`REG_DWORD`) wird als `int` zurückgegeben, binäre Daten (`REG_BINARY`) als Byte-Array.

Ich habe bewusst bei dieser Methode auf die Erzeugung einer Ausnahme für den Fall, dass der Eintrag nicht gefunden wurde, verzichtet. Durch die Übergabe eines geeigneten Defaultwerts können Sie recht flexibel auf das Nichtvorhandensein eines Eintrags reagieren. Übergeben Sie beim Aufruf dieser Methode als Defaultwert `null`, können Sie problemlos entscheiden, ob der Eintrag gefunden wurde. Gibt `ReadValue` in diesem Fall `null` zurück, wurde der Eintrag nicht gefunden (für leere, aber vorhandene Einträge gibt `ReadValue` je nach Datentyp des Eintrags einen Leerstring, 0 oder ein leeres Array zurück).

Die Methode zum Schreiben von Einträgen (`WriteValue`) arbeitet ähnlich. Übergeben wird wie bei `ReadValue` ein Wert der Aufzählung `RegistryRootKeys`, der den Wurzel-Schlüssel bezeichnet, der Pfad des Schlüssels und der Name des Eintrags. Dann folgen der Wert des Eintrags und eine Information darüber, ob `WriteValue` Unterschlüssel anlegen soll, falls diese nicht gefunden wurden.

Abhängig vom Typ des übergebenen zu schreibenden Wertes erzeugt die in `WriteValue` verwendete `SetValue`-Methode einen Eintrag vom Typ `REG_SZ`, `REG_DWORD` oder `REG_BINARY`. Ein `REG_BINARY`-Eintrag wird erzeugt, wenn Sie ein Byte-Array übergeben. Ein `REG_DWORD`-Wert wird erzeugt, wenn Sie einen `unit`- oder `int`-Wert übergeben. In allen anderen Fällen wird ein `REG_SZ`-Wert erzeugt.

`WriteValue` ermittelt zuerst eine Referenz auf das Wurzel-`RegistryKey`-Element und zerlegt den Pfad dann in seine Teile. Beim Durchgehen des Pfads wird ein nicht existierender Unterschlüssel über die `CreateSubKey`-Methode angelegt, falls das Argument `createIfNotExist` `true` ist. Der jeweilige Schlüssel muss dazu durch die Übergabe von `true` am zweiten Argument der `OpenSubKey`-Methode so geöffnet werden, dass auch ein Schreiben möglich ist. Schließlich wird noch der Wert geschrieben, wenn der letzte Unterschlüssel ermittelt werden konnte. Für den Fall, dass dieser nicht gefunden wird, erzeugt `WriteValue` eine Ausnahme.

```
public static void WriteValue(RegistryRootKeys rootKey, string keyPath,
    string valueName, object value, bool createIfNotExist)
{
```

Listing 169: Methode zum Schreiben eines Registry-Eintrags

```
      // Wurzel-Schlüssel ermitteln
      RegistryKey regKey = GetRegistryRootKey(rootKey);

      // Den Pfad in seine Einzelteile zerlegen
      string[] pathToken = keyPath.Split('\\');

      // Pfad durchgehen und den Schlüssel referenzieren
      RegistryKey subKey = null;
      for (int i = 0; i < pathToken.Length; i++)
      {
         if (regKey != null)
         {
            // Unterschlüssel zum Lesen und Schreiben öffnen
            subKey = regKey.OpenSubKey(pathToken[i], true);

            if (subKey == null && createIfNotExist)
            {
               // Unterschlüssel nicht gefunden: Schlüssel erzeugen
               // falls dies gewünscht ist
               subKey = regKey.CreateSubKey(pathToken[i]);
            }
            regKey = subKey;
         }
      }

      // Wert schreiben, wenn der Unterschlüssel gefunden wurde
      if (regKey != null)
      {
         regKey.SetValue(valueName, value);
      }
      else
      {
         // Ausnahme werfen
         throw new Exception("Schlüssel " + rootKey.ToString() + "\\" +
            keyPath + " nicht gefunden");
      }
   }
```

Listing 169: Methode zum Schreiben eines Registry-Eintrags (Forts.)

Nun fehlt noch eine Methode zum Löschen eines Eintrags. `DeleteValue` referenziert dazu zunächst den Schlüssel, der dem übergebenen Pfad entspricht. Wird der Schlüssel gefunden, ruft `DeleteValue` dessen `Delete`-Methode. Im negativen Fall erzeugt die Methode eine Ausnahme:

```
public static void DeleteValue(RegistryRootKeys rootKey, string keyPath,
   string valueName)
{
   // Unterschlüssel zum Schreiben öffnen
   RegistryKey regKey = GetRegistryRootKey(rootKey).OpenSubKey(
      keyPath, true);
   // Wert über den übergeordneten Schlüssel löschen
   // wenn dieser gefunden wurde
   if (regKey != null)
   {
      regKey.DeleteValue(valueName, false);
   }
   else
   {
      // Schlüssel nicht gefunden: Ausnahme werfen
      throw new Exception("Schlüssel " + rootKey.ToString() + "\\" +
         keyPath + " nicht gefunden");
   }
}
```

Listing 170: Methode zum Löschen eines Werts in der Registry

Die Methode `DeleteKey` löscht schließlich einen ganzen Schlüssel. Da ein Schlüssel über die `DeleteSubKey`-Methode des übergeordneten Schlüssels gelöscht werden muss, zerlegt `DeleteKey` den übergebenen Pfad zunächst in den Pfad zum übergeordneten Schlüssel und den Namen des zu löschenden Schlüssels. Mit dem so ermittelten Pfad referenziert die Methode den übergeordneten Schlüssel und ruft dessen `DeleteSubKey`-Methode mit dem Namen des Unterschlüssels auf. Für den Fall, dass der übergeordnete Schlüssel nicht gefunden wurde, erzeugt `DeleteKey` eine Ausnahme:

```
public static void DeleteKey(RegistryRootKeys rootKey, string keyPath)
{
   // Den Pfad zum übergeordneten Schlüssel und den Namen des zu
   // löschenden Schlüssels ermitteln
   int i = keyPath.LastIndexOf("\\");
   string parentKeyPath = keyPath.Substring(0, i);
   string keyName = keyPath.Substring(i + 1, keyPath.Length - i - 1);

   // Den dem zu löschenden Schlüssel übergeordneten Schlüssel zum
   // Schreiben öffnen
   RegistryKey regKey = GetRegistryRootKey(rootKey).OpenSubKey(
      parentKeyPath, true);

   if (regKey != null)
   {
```

Listing 171: Methode zum Löschen eines Schlüssels in der Registry

```
            // Schlüssel über den übergeordneten Schlüssel löschen
            regKey.DeleteSubKey(keyName);
         }
         else
         {
            // Schlüssel nicht gefunden: Ausnahme werfen
            throw new Exception("Schlüssel " + rootKey.ToString() + "\\" +
               keyPath + " nicht gefunden");
         }
      }
   }
```

Listing 171: Methode zum Löschen eines Schlüssels in der Registry (Forts.)

Damit ist die Klasse fertig. Listing 172 zeigt eine beispielhafte Anwendung dieser Klasse.

```
// Wert des Eintrags HKEY_CURRENT_USER\Software\Microsoft\Office\
// 11.0\Word\Options\DOC-PATH lesen
object wordDocPath = RegistryUtils.ReadValue(
   RegistryRootKeys.HKEY_CURRENT_USER,
   @"Software\Microsoft\Office\11.0\Word\Options", "DOC-PATH", null);
if (wordDocPath != null)
{
   Console.WriteLine(wordDocPath);
}
else
{
   Console.WriteLine("Eintrag nicht gefunden");
}

// Eintrag
// HKEY_CURRENT_USER\Software\Addison-Wesley\Codebook\Version
// so schreiben, dass der Eintrag inklusive allen Schlüsseln
// automatisch angelegt wird, falls er noch nicht existiert
RegistryUtils.WriteValue(RegistryRootKeys.HKEY_CURRENT_USER,
   @"Software\Addison-Wesley\Codebook", "Version", "1.0", true);

// Eintrag
// HKEY_CURRENT_USER\Software\Addison-Wesley\Codebook\Samples schreiben
RegistryUtils.WriteValue(RegistryRootKeys.HKEY_CURRENT_USER,
   @"Software\Addison-Wesley\Codebook\Samples", "Registry",
   "c:\\Samples\\Registry", true);
```

Listing 172: Anwendung der Klasse RegistryUtils

```
// Schlüssel
// HKEY_CURRENT_USER\Software\Addison-Wesley\Codebook\Samples löschen
RegistryUtils.DeleteKey(RegistryRootKeys.HKEY_CURRENT_USER,
   @"Software\Addison-Wesley\Codebook\Samples");

// DWord-Wert schreiben
RegistryUtils.WriteValue(RegistryRootKeys.HKEY_CURRENT_USER,
   @"Software\Addison-Wesley\Codebook", "DWord-Test-Value", 123, true);

// Binary-Wert schreiben
byte[] data = new byte[255];
for (byte i = 0; i < 255; i++)
{
   data[i] = i;
}
RegistryUtils.WriteValue(RegistryRootKeys.HKEY_CURRENT_USER,
   @"Software\Addison-Wesley\Codebook", "Binary-Test-Value", data, true);
```

Listing 172: Anwendung der Klasse RegistryUtils (Forts.)

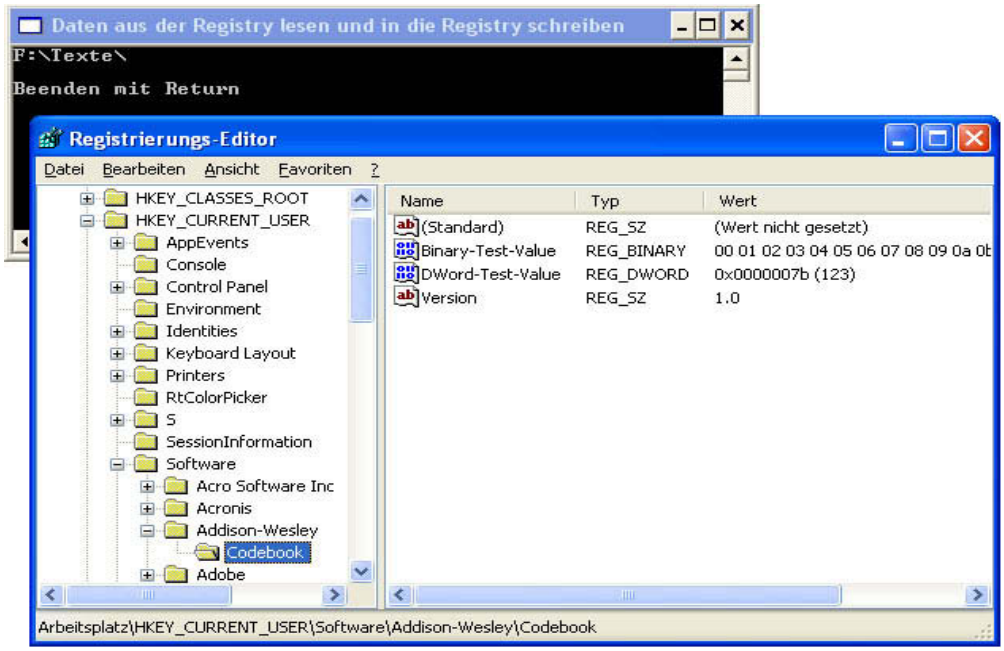


Abbildung 58: Das Beispielprogramm zum Lesen und Schreiben der Registry und der Registrierungs-Editor mit dem erzeugten Registry-Schlüssel