3 Erste Schritte mit Visual Studio .NET

Visual Studio .NET ist – wie sollte man es von einem Microsoft-Produkt auch anders erwarten – in jeder Hinsicht überwältigend: Eine integrierte Entwicklungsumgebung, die Editor, Formulardesigner, Compiler, Online-Dokumentation (MSDN), Debugger, Profiler und Klassenbrowser miteinander vereint, für eine Vielzahl unterschiedlicher Projekte gut ist – und für eine Vielzahl unterschiedlicher Sprachen. Wie bei jeder neu eingeführten (und aller Abstraktion zum Trotz auf der Entwicklerseite reichlich systemnahen) Technik sind die Anforderungen an den Computer vergleichsweise hoch – nicht nur, was den freien Platz auf der Platte (2,4 GByte) für eine Komplettinstallation, die Leistungsfähigkeit des Prozessors und die Größe des Hauptspeichers, sondern auch, was den Aktualitätsgrad des Systems betrifft. Details dazu finden Sie in Anhang A, »Erfahrungen mit der Installation«, der die Installationsprozedur nebst den darin enthaltenen kleinen Fallstricken beschreibt.

In diesem Kapitel geht es weder um eine Liste der verfügbaren Compileroptionen und Sprachen noch um die Feinheiten der diversen Assistenten, die sich in diesem Entwicklungssystem tummeln. Vielmehr soll es ein erstes Gefühl dafür vermitteln, wie man einfachere Projekte mit Visual Studio .NET zu Platte bringt.

3.1 Windows-Anwendungen

Allen Erweiterungen zum Trotz unterscheidet sich Visual Studio .NET von seinen Vorgängern an der Oberfläche hauptsächlich in einem Punkt: Es wird dem »Visual« in seinem Namen gerecht, weil man hier – ähnlich wie bei VB und Delphi – visuelle Komponenten in Formulare einsetzen und über Methoden wie `OnClick()` sozusagen miteinander verdrahten kann. (Mit Microsofts C++ ist dasselbe nicht nur in den älteren Ausgaben von Visual Studio ein recht mühsamer Prozess, der seine Ursprünge im Ressourcen-Editor von Windows 3.0 auch 12 Jahre später nicht verleugnen kann.)

Im Prinzip wäre es denkbar, ein neues Projekt im Programmierstil der 70er Jahre anzufangen, also mit einer leeren Textdatei. VS.NET verfügt aber über eine Reihe von Assistenten, die von Ihnen einige grundlegende Dinge wissen wollen – beispielsweise, ob bei dem Projekt eine fensterbasierte Windows-Anwendung herauskommen soll, ein Web Service oder ein Konsolen-

programm –, und die dann ein entsprechendes Rahmengerüst anlegen. Bitte beachten Sie, dass diese Assistenten mehr oder weniger komplexe Makros sind und deshalb Einbahnstraßen darstellen: Wer ein als ASP angelegtes Projekt später als Windows-Fenster haben will, könnte sich theoretisch zwar an einen Umbau machen – praktisch ist dann aber ein neues Projekt entsprechenden Typs (und das Kopieren einzelner Programmabschnitte) wesentlich sinnvoller.

VS.NET zeigt beim Start auf der *Startseite* eine Liste der zuletzt bearbeiteten Projekte (die direkt nach der Installation natürlich leer ist). Ein Klick auf NEUES PROJEKT ruft den entsprechenden Assistenten auf den Plan. Wie in Abbildung 3.1 zu sehen, will VS.NET Projektdateien standardmäßig im Ordner »Eigene Dateien\Visual Studio-Projekte« unterbringen, was sich über DURCHSUCHEN quasi-permanent ändern lässt: Visual Studio verwendet das an dieser Stelle ausgewählte Basisverzeichnis im Weiteren so lange als Vorgabe, bis Sie erneut DURCHSUCHEN benutzen. Und wo es gerade um Änderungen geht: Auf der Startseite, über die Sie übrigens auch nach einem Wechseln des Basisverzeichnisses an die bereits existierenden Projekte herankommen, findet sich ein Wahlpunkt MEIN PROFIL. Mit ihm legen Sie fest, was Sie beim nächsten Start von VS.NET zu sehen bekommen wollen: die Projektliste, das zuletzt geladene Projekt, ein leeres Editorfenster usw. (Der Königsweg für das Einstellen dieser und weiterer Optionen führt über den Dialog OPTIONEN/UMGEBUNG, aufzurufen über das Menü EXTRAS.)

3.1.1 Programmgerüst und Dateistruktur

Fürs erste Ausprobieren reichen die Standardvorgaben. Wählen Sie also im linken Fenster *Visual C#-Projekte*, im rechten Fenster *Windows-Anwendung*, belassen Sie den Projektnamen auf *WindowsApplication1* und klicken Sie auf OK.

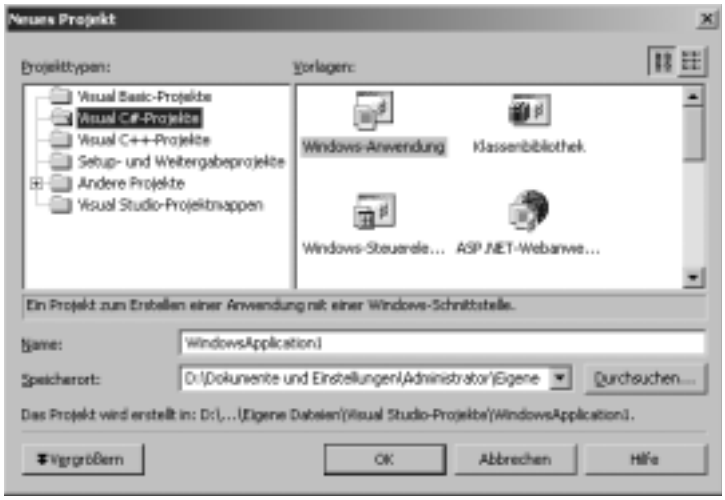


Abbildung 3.1: Der Assistent für neue Projekte nach der Auswahl von *Visual C#-Projekte*

VS.NET erzeugt daraufhin einen Ordner WINDOWSAPPLICATION1 (in *Eigene Dateien\Visual Studio-Projekte*), und hinterlegt darin einige Quelldateien mit dem Codegerüst des neuen Projekts sowie diverse weitere Unterordner. Abbildung 3.2 zeigt eine Exploreransicht der Dateistruktur. (Der Ordner VSMACROS enthält ebendies – Makros für Visual Studio – und ist nicht projektspezifisch; er wurde bei der Installation von Visual Studio .NET angelegt.)

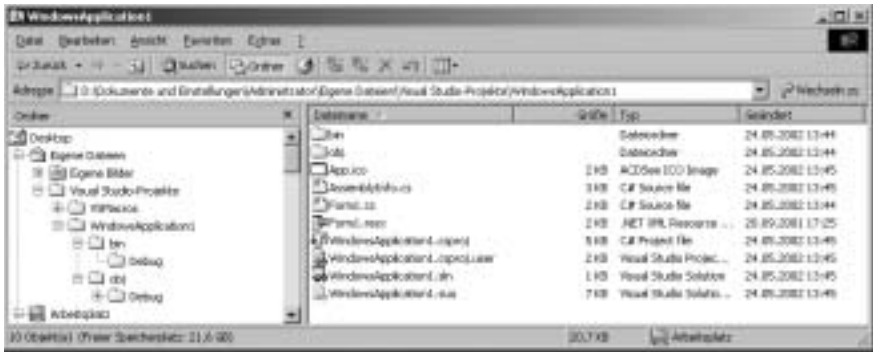


Abbildung 3.2: Die für eine Windows-Anwendung angelegte Verzeichnisstruktur

Im Folgenden ein kurzer Überblick über die einzelnen Dateien:

- *App.ico* – das Icon des Projekts, also das Symbol, mit dem die Anwendung später im Explorer erscheint. Diese Datei kann auch mit einem externen Programm beliebig verändert werden.
- *AssemblyInfo.cs* – das Grundgerüst für ein Assembly, mit dem sich unter anderem das Projekt digital signieren lässt. Die Datei lässt sich mit einem Texteditor (oder eben dem Editor von Visual Studio .NET) bearbeiten und enthält selbst recht ausführliche weitere Informationen.
- *Form1.cs* – das Grundgerüst der Windows-Anwendung (in C#). Den Inhalt dieser Datei bekommen Sie später im Editor zu sehen.
- *Form1.resx* – eine XML-Datei, die die Ressourcen des Formulars bzw. die ihm zugeordneten Komponenten, für Belange des .NET-Laufzeitumgebung beschreiben. Diese Datei wird vom Designer automatisch gepflegt.
- *WindowsApplication1.csproj* – die Konfiguration für das Projekt (Debug und Release, Referenzen auf die .NET-Umgebung). Einen Teil der in dieser Datei niedergelegten Information bekommen Sie später in den Eigenschaftsseiten des Projekts zu sehen.
- *WindowsApplication1.csproj.user* – weitere Konfigurationseinstellungen, hauptsächlich für den Debugger, ebenfalls über die Eigenschaftsseiten des Projekts zu erreichen.
- *WindowsApplication1.sln* – beschreibt, wie die Elemente der vorliegenden Projektmappe (Projekte, Projektelemente, Konfigurationen, Ver-

weise usw.) miteinander zu einer »Lösung« (Solution) verwoben sind. Diese Information ist für Ablauf des Link-Vorgangs wichtig, also den Zusammenbau der Assembly.

- *WindowsApplication1.suo* – beschreibt den aktuellen Zustand der Projektmappe, geöffnete Fenster und deren Arrangement, die Aufgabenliste sowie die sonstigen benutzerspezifische Einstellungen, die VS.NET benötigt, um eine geschlossene Projektmappe bei nächsten Öffnen wieder in den zuletzt vorhandenen Zustand zu versetzen.

Wenn Sie das Projekt später einmal als eigenständige Anwendung ausführen wollen, also nicht innerhalb von VS.NET, finden Sie die EXE-Datei im Unterordner *bin\Debug* bzw. bei einer Compilierung im Release-Modus im Unterordner *bin\Release*.

3.1.2 Entwurf der ersten Windows-Anwendung

VS.NET zeigt nach all diesen Vorbereitungen ein leeres Formular in der Ansicht des Designers und befindet sich im visuell orientierten Entwurfsmodus. Die wesentlichen Aufgaben der drei umliegenden Fenster (vgl. Abbildung 3.3) sind:

- PROJEKTMAPPEN-EXPLORER – dient zur Navigation und zum Auffinden der einzelnen Elemente in der Projektmappe. Über das per Rechtsklick erreichbare Kontextmenü wechseln Sie beispielsweise bei Form1.cs zwischen dem visuellen Entwurfsmodus (DESIGNER ANZEIGEN) und dem Texteditor (CODE ANZEIGEN) mit dem C#-Quelltext des Formulars.
- KLASSENANSICHT – findet sich im gleichen Fenster wie der Projektmappen-Explorer gelegen und gibt den internen Zusammenhang aller in der Projektmappe befindlichen Projekte bis in die Ebene der einzelnen Klassenelemente wieder.
- EIGENSCHAFTEN – stellt die Eigenschaften (auch Ereignisse und Eigenschaftsseiten) des im Designer oder Projektmappen-Explorer ausgewählten Elements dar und ermöglicht es, sie zu manipulieren.
- AUFGABENLISTE – enthält die Fehlermeldungen und Warnungen des jeweils letzten Compilerlaufs sowie die im Quelltext verstreuten Tokenkommentare (wie `// TODO ...`) als Hyperlinks, die direkt (per Doppelklick) an die jeweilige Stelle im Quelltext führen.

Eine Schaltfläche in das Formular einfügen

Am linken Rand des Fensters findet sich eine Leiste, über die weitere Hilfsfenster erreichbar sind. Um in das leere Formular einige Windows-Steuerelemente einzusetzen, benötigen Sie das TOOLBOX-Fenster, das über das Menü ANSICHT/TOOLBOX, den Tastenbefehl Strg+Alt+X oder – falls vorhan-

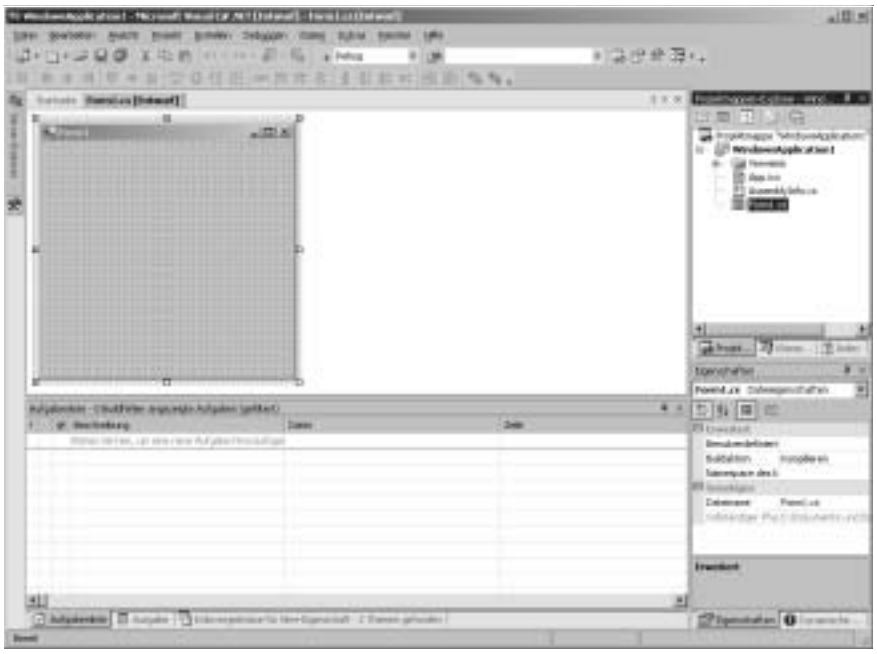


Abbildung 3.3: Formular für eine frisch generierte Windows-Anwendung

den – durch einen schlichten Klick auf den Reiter in der Leiste einzublenden ist. Die Pin-Schaltfläche in der Titelleiste fixiert das **Fenster und bzw. verbannt es wieder in die Leiste.** Zum Einfügen einer Schaltfläche klicken Sie in der **Toolbox auf** *Windows Forms*, dann auf *Button*, und schließlich im Formularentwurf auf die Stelle, an der die Schaltfläche im Fenster erscheinen soll (Abbildung 3.4).

Wenn die Schaltfläche lediglich als schwarzes Klötzchen (oder gleich gar nicht) erscheint, haben Sie die ***Maus beim Klicken bewegt, also gezogen –*** *und das interpretiert der Editor als* ***Größeneinstellung des aktuellen Elements****. Werfen Sie gegebenenfalls einen Blick ins Fenster* EIGENSCHAFTEN*: Wenn dort* `button1` *steht, können Sie die* ***Größe der Schaltfläche auch über*** *die Eigenschaft* `Size` *per numerischer Eingabe Ihren* ***Wünschen anpassen.***

Der Schaltfläche eine Aktion zuordnen

Um dieser Schaltfläche eine Aktion zuzuordnen, doppelklicken Sie darauf oder (allgemeiner) klicken im EIGENSCHAFTEN-Fenster auf das Blitzsymbol, suchen in der daraufhin erscheinenden Liste für Ereignisse den Eintrag `Click` und doppelklicken in der rechten (im Moment noch leeren) Spalte neben diesem Eintrag. **VS.NET fügt daraufhin das Codegerüst einer Methode** namens `button1_Click()` in den Quelltext ein, schaltet in den Texteditor um und setzt die Schreibmarke gleich an die richtige Stelle, damit Sie die Aktion sofort implementieren können (Abbildung 3.5).

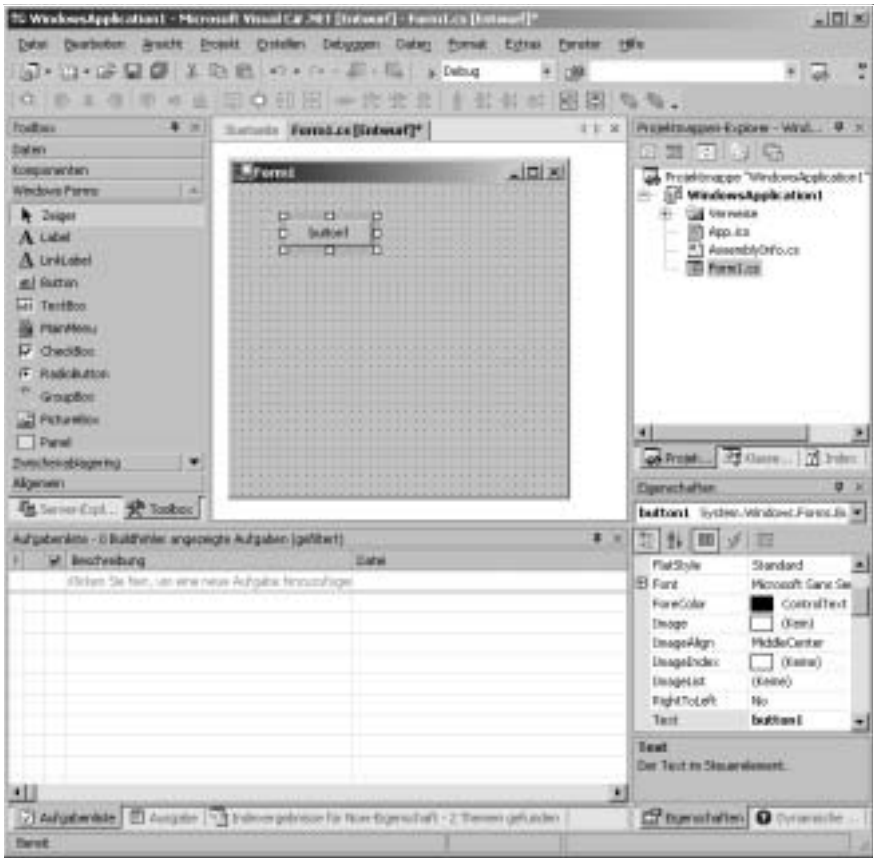


Abbildung 3.4: Das Formular mit einer eingefügten Schaltfläche, deren Eigenschaften über das gleichnamige Fenster veränderbar sind

Der Designer beschränkt sich (nicht nur bei derartigen Ereignissen) auf das Erzeugen des notwendigen Drumherums. Diese Methode ergänzen Sie nun um eine »echte« Aktion, nämlich die Ausgabe eines Meldungsfensters per `MessageBox`:

```
private void button1_Click(object sender, System.EventArgs e)
{
  MessageBox.Show ("Klick!", "Die erste Demo-Anwendung");
}
```

Anwendung übersetzen und starten

Der Menübefehl DEBUGGEN/STARTEN – ersatzweise F5 – weist VS.NET an, die Quelldateien dieses Jahrhundertwerks zu speichern, in Zwischencode zu übersetzen, mit den Laufzeitbibliotheken zu einer EXE-Datei zu binden und schließlich, wenn alles gut gegangen ist, auszuführen. (Wer von Microsoft Visual C/C++ her kommt, darf sich darüber freuen, dass die Rückfrage »Dateien x.c und y.cpp wurden verändert. Neu übersetzen?« endlich in den Bytehimmel gewandert ist.) Das Ergebnis zeigt Abbildung 3.6.

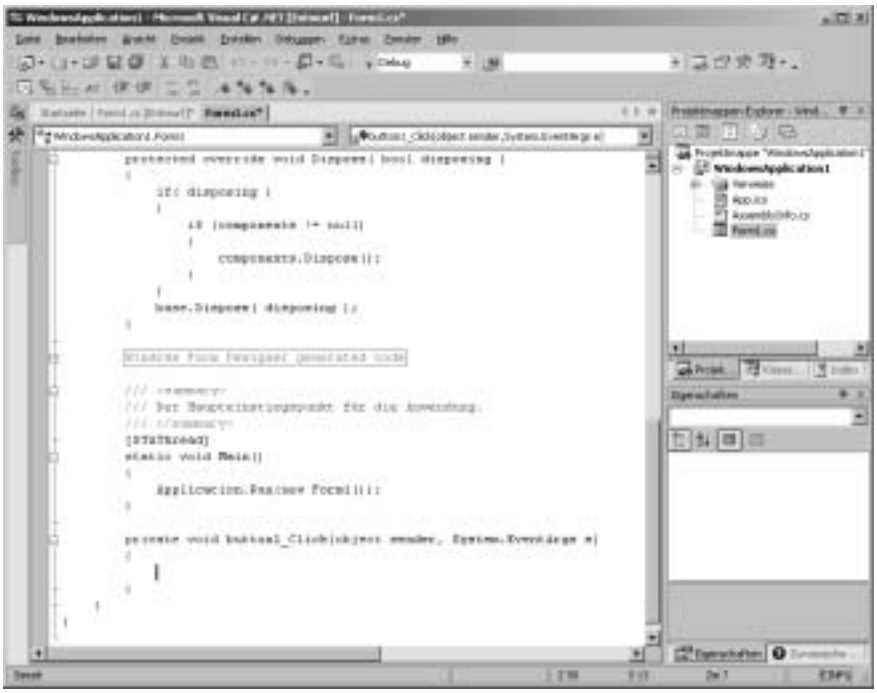


Abbildung 3.5: Eine vom Designer erzeugte Methode für Klicks auf *button1*

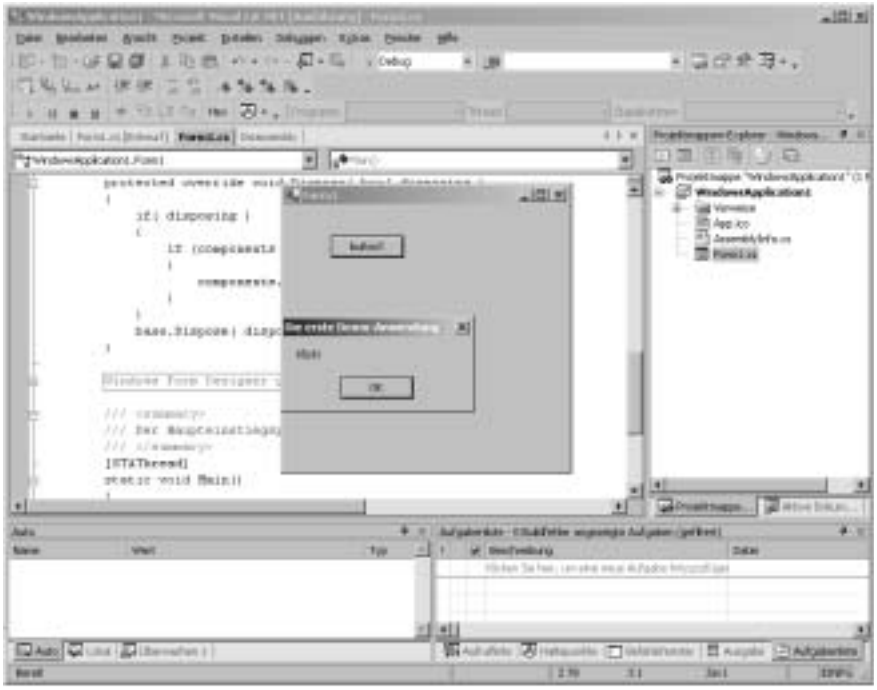


Abbildung 3.6: Das erste Programm, innerhalb von VS.NET ausgeführt

Das vom Windows Explorer aus ausführbare Programm, also die EXE-Datei *WindowsApplication1.exe*, finden Sie wie bereits erwähnt in *Eigene Dateien\Visual Studio-Projekte\WindowsApplication1\bin\Debug*.

Da .NET praktisch alle Elemente der grafischen Benutzeroberfläche von Windows als Komponenten verkapselt, wobei ihre Daten als Eigenschaften verfügbar sind, und Visual Studio .NET die Definition von Methoden zur Behandlung von Ereignissen per Klick möglich macht, geht das Erstellen von Windows-konformen Oberflächen hier genauso flüssig von der Hand wie bei VB 6 und Delphi. (Wer an Visual C/C++ gewöhnt ist, wird sich nach einer kurzen Einarbeitung höchstens fragen, wieso sein Hauscompiler dasselbe immer noch so entsetzlich umständlich macht.)

Kleiner Ausblick

Das folgende Fragment demonstriert die – zumindest an der Oberfläche – ausgesprochen simple »Verdrahtung« zweier Steuerelemente. Auf einen Klick in einem Listenfeld hin setzt es die Beschriftung einer Schaltfläche mit dem im Listenfeld ausgewählten Text:

```
private void listBox1_SelectedIndexChanged(object sender,
   System.EventArgs e)
{
    button1.Text = listBox1.Items[listBox1.SelectedIndex].ToString();
}
```

Wenn Sie eine Viertelstunde Zeit haben, dann können Sie das in diesem Abschnitt vorgestellte Projekt ja einmal entsprechend erweitern, oder – selbstverständlich nur um des Lerneffekts willen – dafür sorgen, dass `button1` auf Mausklicks hin seine Position im Formular so ändert, dass der Benutzer daneben getroffen hat. (Wer `button1_Click` entsprechend ausbauen will, sollte sich die Methoden `MousePosition` und `PointToClient` näher ansehen – deren Aufruf man sich nach der Zuordnung einer Methode `button1_MouseDown` allerdings sparen kann, weil bei dieser Methode die Mauszeigerposition als Parameter mitgeliefert wird. Die aktuelle Größe des Formulars lässt sich über `this.ClientSize` abfragen, der Rest sollte sich durch Studium der Methode `InitializeComponent` in der Datei *Form1.cs* erschließen – dort wird das Setzen der Position von `button1` vorexerziert.)

3.2 Konsolenanwendungen

Auch wenn sicher niemand mehr versuchen wird, eine Textverarbeitung für die Konsole (die in älteren Windows-Versionen noch weniger vornehm »DOS-Box« hieß) zu schreiben: In C# ist diese Art von Anwendungen nicht nur für das schnelle Ausprobieren syntaktischer Konstrukte und Sprachfeatures gut. Tatsächlich eignet sich eine Konsolenanwendung aber nicht zuletzt aufgrund der kürzeren Kompilierungszeit und des einfacheren Code-

gerüsts wunderbar als abschließende, praktische Antwort auf Syntax- oder Semantikfragen (und die Autoren geben gerne zu, dass fast alle Syntaxbeispiele in diesem Buch in einem Projekt *ConsoleApplicationxx* eingetippt worden sind).

Angelegt werden Konsolenprogramme in VS.NET mit denselben Schritten wie Windows-Anwendungen – nur dass Sie im Dialogfeld NEUES PROJEKT ein wenig weiter nach unten rollen müssen (siehe Abbildung 3.7). Die von VS.NET für Konsolenprogramme angelegte Ordnerstruktur ist dieselbe wie für Windows-Anwendungen, die ausführbare Datei findet sich hinterher im Unterordner *bin\debug* bzw. *bin\Release*.

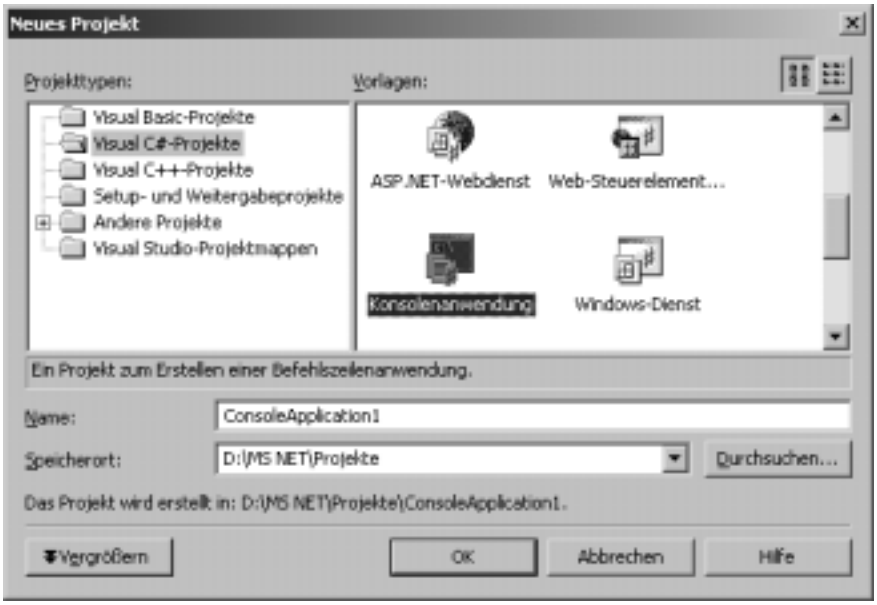


Abbildung 3.7: Anlegen eines Konsolenprogramms

Der vom Assistenten erzeugte Code ist allerdings um einiges kürzer und kann hier ohne weiteres komplett wiedergegeben werden.

```
using System;

namespace ConsoleApplication1
{
  /// <summary>
  /// Zusammendfassende Beschreibung für Class1.
  /// </summary>
  class Class1
  {
    /// <summary>
    /// Der Haupteinstiegspunkt für die Anwendung.
    /// </summary>
    [STAThread]
```

```
    static void Main(string[] args)
    {
      //
      // TODO: Fügen Sie hier Code hinzu, um die Anwendung zu starten
      //
    }
  }
}
```

Um aus diesem Gerüst das übliche »Hello, world«-Programm zu machen, ersetzen Sie den »**TODO**«-Kommentar im Rumpf der Methode `Main` durch:

```
Console.WriteLine("Hallo, Welt!");
Console.ReadLine();
```

Mit `WriteLine()` lässt sich noch einiges mehr anstellen als nur vorgefertigte Stringparameter auszugeben. Tatsächlich ist diese Methode für Alles und Jedes überladen – insgesamt 19 Mal. In ihrer für den praktischen Einsatz wichtigsten Variante ermöglicht die Methode die formatierte Ausgabe beliebig vieler Parameter beliebigen Datentyps, die durch einen im ersten Parameter zu übergebenden Formatstring beschrieben werden (C-Programmierer werden sich hier an `printf()` erinnert fühlen.) Die Syntax für den Formatstring erlaubt Platzhalter für den zweiten bis n-ten Parameter der Methode, wobei der zweite Parameter durch {0}, der dritte durch {1} und so weiter symbolisiert wird. Zusätzliche, durch Komma oder Doppelpunkt abzutrennende Formatsymbole in den geschweiften Klammern ermöglichen Formatierungen wie eine bestimmte Anzahl von Dezimalstellen nachdem Komma, das Auffüllen mit Leerzeichen bis zu einer bestimmten Breite etc. (Die Online-Hilfe bringt hier unter dem Stichwort »Formatieren von Zahlen« sowie zur Methode `String.Format()` eine Reihe aussagekräftiger Beispiele.)

Ersatzweise könnte man die Ausgabe des Programms also auch so zusammenschustern:

```
string Greeting = "Hallo";
Console.WriteLine("{0},{1,5}!", Greeting, "Welt");
```

Beachten Sie, dass das Leerzeichen in der Ausgabe hier über die Längenangabe *5* für den dritten Parameter der Methode zustandekommt.

Der `ReadLine()`-Aufruf wartet auf eine Eingabe beliebiger Länge, die mit einem ↵ abgeschlossen sein muss, verwirft die eingelesenen Zeichen sofort wieder und hat hier nur einen einzigen Zweck: Das mittels F5 im Debugger gestartete Programm so lange aufzuhalten, bis der Benutzer die vorangehenden Ausgaben bewundert hat. Ärgerlicherweise öffnet VS.NET nämlich für Konsolenprogramme ein eigenes Konsolenfenster, lässt das Programm ablaufen und schließt dieses Konsolenfenster sofort nach Ende des Programms wieder. Erfolgt der Start der Anwendung hingegen mit dem

Menübefehl DEBUGGEN/STARTEN OHNE DEBUGGER (bzw. Strg+F5), ist `ReadLine()` nicht erforderlich, weil die Laufzeitumgebung dann von sich aus eine entsprechende Abfrage einsetzt, wie in Abbildung 3.8 zu sehen.

Abbildung 3.8: Ausgabe des ersten Konsolenprogramms

3.2.1 Kommandozeilenparameter

Konsolenanwendungen werden üblicherweise mit Kommandozeilen-Parametern gesteuert, die dann ihrerseits als Array `args` der Methode `Main` verfügbar sind. Das Festlegen solcher Kommandozeilen-Parameter für Testläufe innerhalb der Entwicklungsumgebung ist in VS.NET leider derartig gut versteckt, dass wir nicht darum herum kommen, dieses einführende Kapitel mit einer detaillierten Anleitung zu versehen:

1. Wählen Sie im PROJEKTMAPPEN-EXPLORER den Projektnamen der Anwendung (hier: *ConsoleApplication1*) aus.
2. Ein Rechtsklick auf dem Projektnamen bringt ein Kontextmenü zum Vorschein, in dem Sie den Befehl EIGENSCHAFTEN wählen. Alternative Wege sind ANSICHT/EIGENSCHAFTSSEITEN, der Tastenbefehl ⇧+F4 und ein Klick auf das rechte der vier Symbole im Fenster EIGENSCHAFTEN.
3. Wählen Sie im daraufhin erscheinenden Dialogfeld CONSOLEAPPLICATION1-EIGENSCHAFTSSEITEN im linken Fenster den Ordner *Konfigurationseigenschaften* und dort den Unterpunkt *Debuggen*.
4. Wenn Sie nun das rechte Fenster dieses Dialogfeldes bis zum Punkt *Startoptionen* rollen, kommt schlussendlich ein Feld *Befehlszeilenargumente* in Sicht (siehe Abbildung 3.9).

Es wäre sicher nett, wenn sich diese Einstellung auch direkt über das Fenster EIGENSCHAFTEN erreichen ließe. Wie Abbildung 3.9 sozusagen nebenbei zeigt, ist dieses Fenster in der aktuellen Version von VS.NET sowieso nur recht spärlich besetzt.

Der folgende Codeauszug zeigt der Vollständigkeit halber den programmgesteuerten Zugriff auf solche Parameter:

```
for (int x = 0; x < args.Length; x++)
  Console.WriteLine("Parameter {0}: '{1}'", x, args[x]);
```

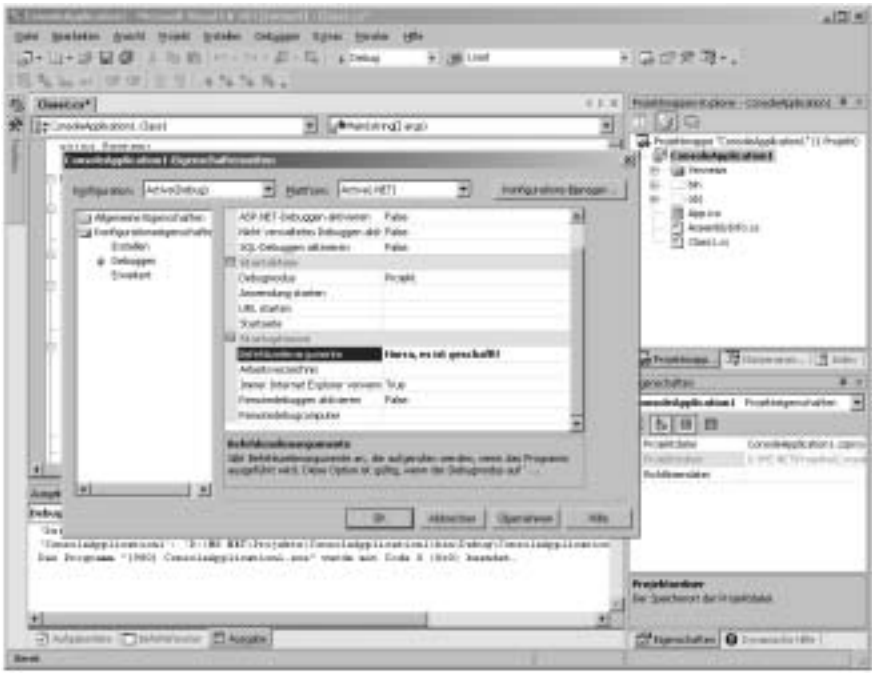


Abbildung 3.9: Fast schon ein easter egg – Definition von Kommandozeilenparametern in VS.NET

In Abbildung 3.10 finden Sie schließlich das Ergebnis. Wie dort zu sehen, liefert die .NET-Umgebung im Element mit dem Index 0 schlicht den ersten Kommandozeilenparameter – und tanzt damit aus der Reihe der meisten anderen Programmiersprachen, die ursprünglich einmal DOS zur Grundlage hatten: Dort ist das Element 0 von *args* (bei Delphi: *ParamStr*) für den Namen der ausführbaren Datei reserviert und deshalb immer besetzt. In einer C#-Anwendung kann `args` dagegen ohne weiteres 0 Elemente enthalten.

Abbildung 3.10: Ausgewertete Kommandozeilenparameter

Windows-Anwendungen bekommen ebenfalls Kommandozeilenparameter zu sehen, soweit sie beim Aufruf angegeben werden – etwa über eine Verknüpfung auf dem Desktop. Wer diese Möglichkeit in einer Windows-Anwendung vorsehen will, muss in VS.NET exakt dieselben Einstellungen vornehmen wie für ein Konsolenprogramm. Des Weiteren ist ein Nachtrag der Parameterdeklaration `string[] args` *für die Methode* `Main()` *erforderlich: Der Assistent vereinbart diese Methode leider in der parameterlosen Variante.*

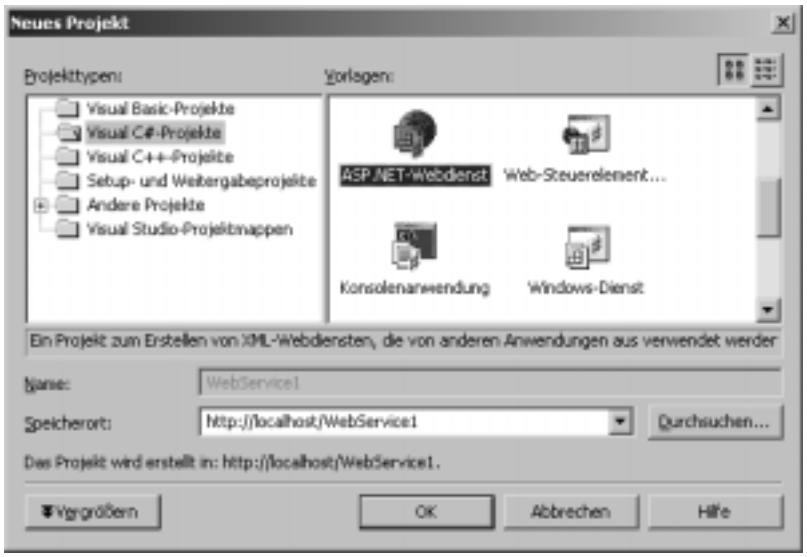
3.3 Webdienste

Die zusammen mit .NET eingeführten Webdienste stellen einen höchst eleganten Weg zur losen Koppelung von Systemen dar, die – im Gegensatz zu Microsofts DCOM – auch übers Internet und damit über die Grenzen von Firewalls hinweg funktioniert, weil der Datentransfer auf bekannt »harmlosen« Protokollen wie HTML, XML und SOAP aufbaut. Was man damit anfangen kann, lässt sich am einfachsten an einem Beispiel verdeutlichen: Stellen Sie sich eine Webseite vor, die für eine Suche nach Artikeln die ersten zehn Fundstellen in einer Liste darstellt. Mit herkömmlichen Techniken wird diese Seite serverseitig per CGI oder ASP generiert, und enthält eine Schaltfläche, deren Funktion man nur so umschreiben kann: »Anweisung an den Server: Komplette HTML-Seite erneut per CGI oder ASP generieren, Suchlauf in der Datenbank erneut ausführen, diesmal die ersten 10 Ergebnisse verwerfen und das zweite Zehnerpack an Fundstellen einbauen. Danach alles erneut übertragen.« Auch wenn diese Beschreibung unterschlägt, dass sich Tausende von Entwicklern den Kopf darüber zerbrochen haben, wie man diese katastrophale Verschwendung an Rechenzeit und Datenaufkommen einigermaßen in den Griff bekommt, und dabei natürlich nicht ganz ohne Erfolg geblieben sind: Die »Interaktivität« von Webseiten basiert nun einmal weitgehend darauf, dass der Server auf Eingaben und Mausklicks hin eine neue HTML-Seite generiert.

Wie praktisch wäre es doch, in einer HTML-Seite eine visuelle Komponente unterzubringen, die die Abfrage des nächsten darzustellenden Elements über den schlichten Aufruf einer serverseitigen Funktion erledigt, wobei Parameter und die zu erwartenden Ergebnisse in der Syntax von C# formuliert werden – mithin in derselben Weise wie bei einem Methodenaufruf. Nun, genau so sieht ein Webdienst für den Programmierer aus – und zwar sowohl auf der Client- als auch auf der Serverseite. Mehr noch: Webdienste lassen sich sowohl von (im Internet Explorer ablaufenden ASP.NET-Anwendungen) als auch von schlichten (.Net-konformen) Windows-Anwendungen aus gleichermaßen nutzen.

Angelegt werden Projekte für Webdienste in VS.NET mit denselben Schritten wie Windows-Anwendungen und Konsolenprogramme, wobei hier allerdings bereits im Dialogfeld NEUES PROJEKT der erste Unterschied auffallen sollte (vgl. Abbildung 3.11): Die Standardvorgabe für den Speicherort ist ein Unterverzeichnis von `http://localhost` – und wie nunmehr unschwer zu erraten, setzen Projekte dieser Art einen laufenden Internet Information Server auf dem Entwicklungssystem voraus. (Dessen Standardkonfiguration sorgt wiederum dafür, dass dieser URL zu `c:\inetpub\wwwroot` aufgelöst wird.)

Abbildung 3.11: Anlegen eines Projekts für einen Webdienst



INFO

Die Version des Internet Information Server, die sich auf der Installations-CD von Windows 2000 findet, hat eine Reihe von Sicherheitslücken, die von Viren wie Nimda ausgenutzt werden. Eine entsprechende Aktualisierung per Windows-Update ist deshalb auch dann ein unbedingtes Muss, wenn Sie eine Firewall vorgeschaltet haben. Wer auf Nummer Sicher gehen will, führt dieses Windows-Update direkt nach der Installation aus, und schaltet den IIS vor dem Herstellen der Internetverbindung erst einmal über START/PROGRAMME/VERWALTUNG/DIENSTE *ab. (Ärgerlicherweise ist dieser Tipp nicht nur etwas für ausgesprochene Paranoiker: Im April 2002 war eine Unzahl von Systemen weltweit von diesem Virus befallen, und einen der Autoren hat es wenige Minuten nach einer Neuinstallation wg. Nimda ein zweites Mal erwischt, nämlich während des allfälligen Windows-Updates.)*

INFO

Im Rahmen der Installation legt VS.NET ein lokales Benutzerkonto mit dem Namen VS Developers an, das Schreibrechte für wwwroot hat. Über dieses Konto (und das http-Protokoll) werden sämtliche Operationen mit Webdienst-Projektdateien abgewickelt.

Für das Anlegen von Projekten für einen neuen Webdienst nimmt sich VS.NET etwas mehr Zeit. Ein Grund dafür dürfte sein, dass die Dateizugriffe ausschließlich per http geschehen, ein zweiter, dass dabei eine komplette Baumstruktur entsteht (deren Ordner allerdings überwiegend leer sind). Das Ergebnis einer erfolgreichen Kompilierung ist eine (ISAPI-)DLL im Ordner *\localhost\bin*; separateVerzeichnisse für Debug- und Release-Versionen gibt es hier nicht.

3.3.1 Implementation eines Dienstes

Für einen neu angelegten Webdienst zeigt VS.NET erst einmal (wie bei einer Windows-Anwendung) den Designer, über den Sie nun aus der Toolbox oder dem Server-Explorer Komponenten in das Projekt einsetzen könnten. Visuelle Komponenten wie Schaltflächen sind zwar zulässig, haben aber nicht viel Sinn, da der Webdienst auf dem Server ausgeführt wird. Tatsächlich geht es hier in erster Linie um Komponenten für den Zugriff auf Datenbanken und damit um ein Thema, das den Rahmen dieses Einführungskapitels sprengen würde.

Nach dem Umschalten auf die Codeansicht zeigt VS.NET ein (im Moment noch auskommentiertes) Beispiel für den einfachsten aller Dienste: eine parameterlose Methode, die einen fixen String zurückgibt (vgl. Abbildung 3.12).

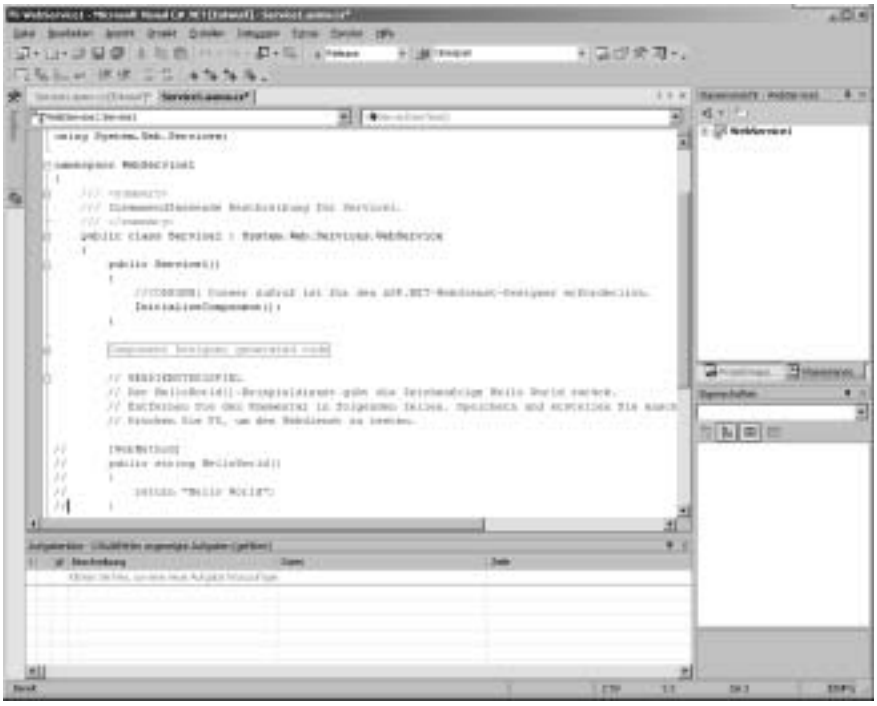


Abbildung 3.12: Der Assistent gibt einen Webdienst als Beispiel in auskommentierter Form vor.

Zur Demonstration entfernen Sie die Kommentarzeichen (alle fünf – also auch das vor `[WebMethod]`) und ergänzen den Quelltext um zwei weitere Methoden:

```
[WebMethod]
public string HelloWorld()
{
  return "Hello World";
}
```

```
[WebMethod]
public string ServerDateTime()
{
  return "Aktuelle Serverzeit ist: " + System.DateTime.Now.ToString();
}

[WebMethod]
public bool ServerPassword(string Username, string PW)
{
  return (Username == "Fritz") && (PW == "geheim");
}
```

Tatsächlich ist das bereits alles, was Sie tun müssen – und zwar nicht nur zur Definition dreier Webdienste, sondern auch für die notwendigen Tests: Bei der Compilierung (am einfachsten: per F5) erzeugt VS.NET selbstständig einen Client in Form einfacher HTML-Seiten, der den Aufruf dieser drei Funktionen ausführt. Dass `ServerPassword` zwei Strings als Parameter erwartet, bekommt VS.NET ebenfalls mit: Wie in Abbildung 3.13 zu sehen, enthält die HTML-Seite entsprechende Eingabefelder.

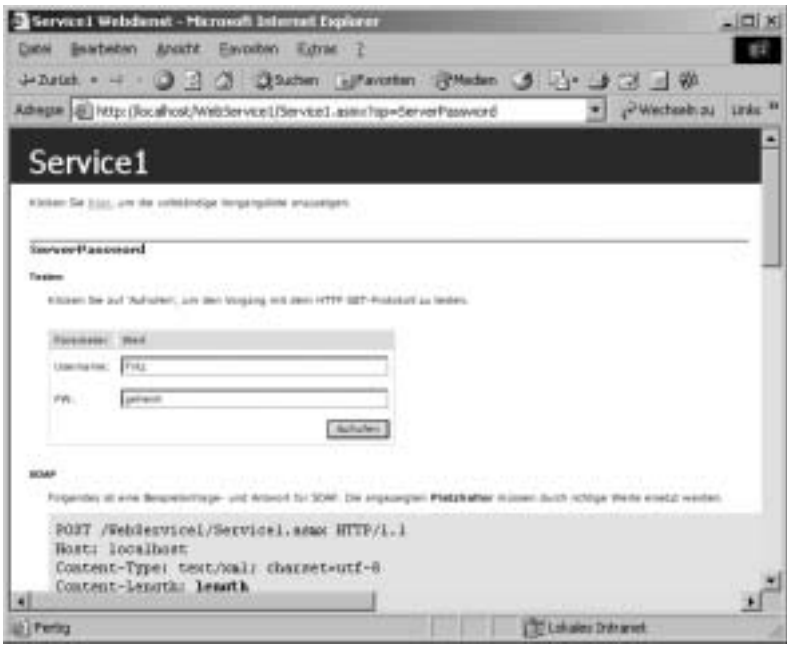


Abbildung 3.13: Von VS.NET für einen Webdienste generierte HTML-Seiten mit Eingabefeldern für die Parameter

Abbildung 3.14 zeigt das Ergebnis des Funktionsaufrufs in XML. Die automatisch generierte Testumgebung beschränkt sich auf die Darstellung dieses Ergebnisses in einem eigenen Fenster des Internet Explorers und ein schlichtes `http GET`. Die für den Aufruf des jeweiligen Diensts erzeugte Seite enthält ein weitgehend ausgearbeitetes Quelltext-Beispiel sowohl für http als auch für SOAP.

Abbildung 3.14: Das Ergebnis der Abfrage

3.3.2 Ein Webdienst-Client

Der in diesem Abschnitt vorgestellte ASP.NET-Client basiert auf WebForms und ist zwar vom Frontend einer e-Commerce-Website ähnlich weit entfernt wie die zuvor besprochenen Webdienste, demonstriert aber das Prinzip recht anschaulich: Er erlaubt die Abfrage der Serverzeit – also den Aufruf des Webdienstes `ServerDateTime` – erst nach einer korrekten Anmeldung. (Wer jetzt stutzt, weil `ServerDateTime` wie zuvor demonstriert voraussetzungslos aufgerufen werden kann, mithin die »Sicherheit« hier clientseitig implementiert ist, hat nicht nur den vorangehenden Abschnitt gut verstanden, sondern auch recht – aber schließlich geht es um eine Demonstration.)

Zum Anlegen eines ASP.NET-Clients rufen Sie den Assistenten mit DATEI/NEU/PROJEKT auf den Plan und wählen im rechten Fenster ASP.NET-Webanwendung. Wie in Abbildung 3.15 zu sehen, schlägt VS.NET auch hier `http://localhost` als Speicherort vor. Belassen Sie es bei den vorgeschlagenen Namen.

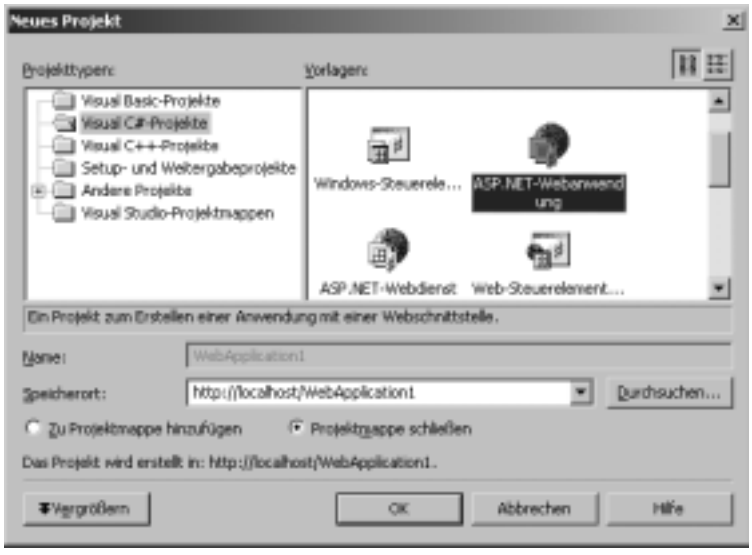
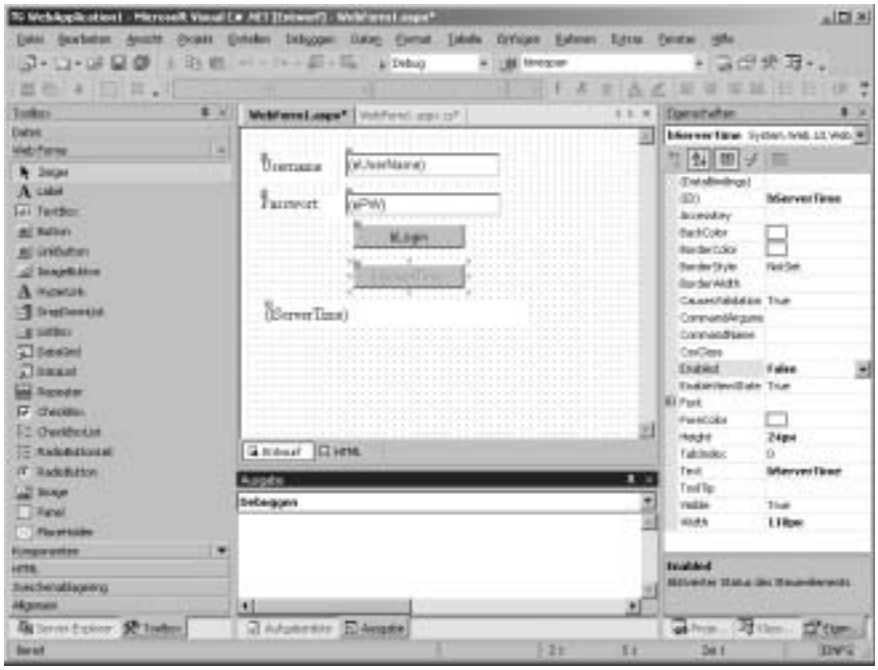


Abbildung 3.15: ASP.NET-Clients werden ebenfalls in http://localhost gespeichert.

Nachdem der Assistent seine Arbeit getan hat (was auch hier wieder einen Moment dauern kann), zeigt VS.NET ein leeres Webformular. Aktivieren Sie die Toolbox (Strg+Alt+X), wählen Sie dort WEBFORMS und schmücken Sie das Formular mit insgesamt fünf Komponenten: Zwei Labels (IDs: lUsername, lPW), zwei Texteingabefeldern (eUsername, ePW), einem dritten Label (lServerTime) und schließlich zwei Schaltflächen (bLogin, bServerTime). In Abbildung 3.16 haben wir für die Eingabefelder und die Schaltfläche die Komponentennamen zusätzlich als Eigenschaft *Text* eingetragen, um die Zusammenhänge deutlich zu machen. Wie dort zu sehen, ist die Eigenschaft `bServerTime.Enabled` auf `false` gesetzt: diese Schaltfläche soll erst verwendbar sein, wenn sich der Benutzer angemeldet hat.

Wer an Visual Basic oder Delphi gewöhnt ist, wird mit dem Einsetzen einer Handvoll Standardkomponenten in ein Formular wohl kaum Schwierigkeiten haben. Für leidgeplagte C/C++-Programmier gilt: Wer die Windows-Beispielanwendung überblättert hat, möge einige Seiten zurückgehen. Das Einsetzen visueller Komponenten und die Zuordnung von Methoden zur Ereignisbehandlung ist dort detailliert beschrieben.

Abbildung 3.16: Entwurf des Client-Formulars



Beim Design von Windows-Anwendungen gibt VS.NET Standardgrößen für Labels, Schaltflächen usw. vor, weshalb man dort mit einem einfachen Klick zum Einsetzen solcher Elemente auskommt. Bei WebForms ist dagegen für all diese Elemente das Aufziehen eines Rahmens verlangt. Mit einer Ausrichtung am Raster (fünftes Symbol in der zweiten Zeile) tut man sich leichter bei der Größenbestimmung.

3.3.3 Aufruf des Webdienstes ServerTime

Welche **Aktion der Schaltfläche** `bServerTime` zuzuordnen ist, dürfte sich bereits aus ihrem Namen ergeben: Der Aufruf des **Webdienstes und das Einsetzen** des Ergebnisses in die Eigenschaft *Text* des Labels `lServerTime`. Wobei natürlich die Frage zu klären bleibt, wie man an den Webdienst herankommt.

Die erste Teilantwort darauf ist der Menübefehl PROJEKT/WEBVERWEIS HINZUFÜGEN, der das gleichnamige Dialogfeld auf den Plan ruft, in dem **Sie nun** – eine aktive Internetverbindung vorausgesetzt – weltweit nach aktiven Webdiensten suchen können. Da es im Moment aber um einen Dienst geht, der **ausschließlich lokal auf Ihrem System verfügbar ist, müssen Sie den URL** selbst einsetzen: `http://localhost/webservice1/service1.asmx`. Wie in Abbildung 3.17 zu sehen, zeigt das Dialogfeld danach die Auswahlseite für die im vorangehenden Abschnitt erstellten Webdienste an. **Schließen Sie das Dialog**feld mit einem Klick auf VERWEIS HINZUFÜGEN.

Abbildung 3.17: Über die direkte Eingabe der lokalen URL sind die selbst erstellten Webdienste erreichbar und lassen sich dem Projekt als Verweis hinzufügen.

In der Projektmappe sollten Sie nun einen zusätzlichen **Teilbaum** *Webverweise/localhost* sehen, der zwei Beschreibungsdateien (service1.disco und service1.wsdl) enthält. Abbildung 3.18 zeigt die **Veränderungen im Klassen**system der Anwendung: Der Assistent hat eine Klasse `Service1` erzeugt, die eine Hüllfunktion **übernimmt: Ihre Methoden, zu denen auch** `ServerDateTime` und `ServerPassword` gehören, erzeugen Serveraufrufe im SOAP-**Format, lesen** die per XML zurückgegebenen Ergebnisse und geben ihrerseits **Daten im** gewohnten Format von C# zurück.

Abbildung 3.18: Die von VS.NET generierte Hüllklasse Service1 mit ihren Methoden

Klicks auf bServerTime

Was nach diesen Vorbereitungen noch zu tun bleibt, ist größtenteils Verdrahtungsarbeit. Beim Klick auf `bServerTime` soll der Webdienst `ServerTime` aufgerufen und der zurückgegebene String über das Label `lServerTime` dargestellt werden.

Wählen Sie also `bServerTime` im Formular aus und klicken Sie im Fenster EIGENSCHAFTEN auf das Blitzsymbol. Ein Doppelklick in der rechten Spalte neben dem Ereignis *Click* erzeugt die Methode `bServerTime_Click`, deren Rumpf Sie folgendermaßen ausfüllen:

```
private void bServerTime_Click(object sender, System.EventArgs e)
{
  // Instanz der Hüllklasse anlegen
  localhost.Service1 ws = new localhost.Service1();

  // Aufruf eines Webdiensts
  lServerTime.Text = ws.ServerDateTime;
}
```

Wer will, kann `bServerTime.Enabled` einmal kurz auf `true` zurücksetzen und das Programm danach per F5 ausprobieren.

3.3.4 Aufruf des Webdienstes ServerPassword

Der Aufruf dieses Dienstes läuft syntaktisch nach demselben Strickmuster wie der Aufruf von `ServerTime`. Die Eigenschaft `Enabled` der beiden Schaltflächen wird abhängig vom Funktionsergebnis gesetzt. Verpassen Sie `bLogin` also nach demselben Muster wie `bServerTime` eine Methode `Click`, deren Rumpf dann folgendermaßen ausgefüllt wird:

```
private void bLogin_Click(object sender, System.EventArgs e)
{ bool LoggedIn;

  // Instanz der Hüllklasse anlegen
  localhost.Service1 ws = new localhost.Service1();

  // Aufruf von ServerLogin, mit Übergabe der
  // Werte von eUserName und ePW als Parameter
  LoggedIn = ws.ServerPassword(eUserName.Text, ePW.Text);

  bLogin.Enabled = !LoggedIn;
  bServerTime.Enabled = LoggedIn;
}
```

Wie zu sehen, stellt sich auch der Webdienst `Serverpassword` dem Programm als normale Klasse mit Methoden dar. Abbildung 3.19 zeigt schließlich das Ergebnis.

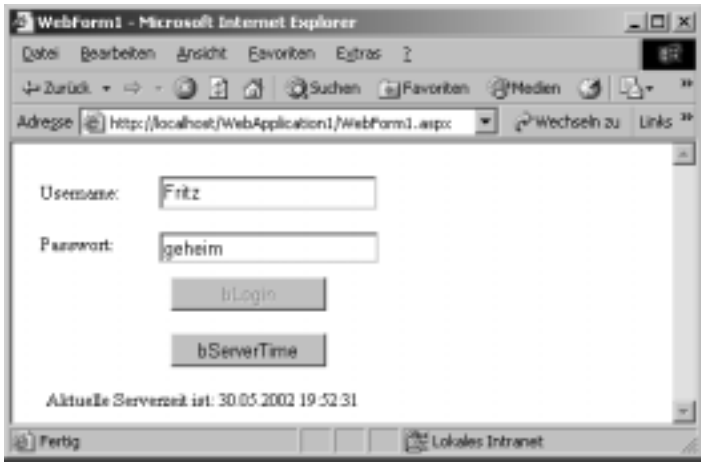


Abbildung 3.19: Das fertige Webformular nach einem gelungenen Login und der Abfrage der Serverzeit

Natürlich ließe sich hier noch einiges tun: Das Passwort sollte sinnvollerweise nur über Sternchen dargestellt werden, die Eingabefelder könnten über eine (gemeinsame Methode) `TextChanged` dafür sorgen, dass `bLogin` nur dann wählbar ist, wenn beide auch wirklich Text enthalten und so weiter.

So einfach dieses Beispiel aber auch ist: Wenn Sie die letzten knapp 10 Seiten praktisch am Computer nachvollzogen haben, sind Sie nun stolzer Besitzer eines Server/Client-Systems, das nicht nur theoretisch selbst dann funktioniert, wenn zwischen den beteiligten Computern ein ganzer Kontinent liegt. Den dafür notwendigen Programmieraufwand kennen Sie inzwischen – und genau jetzt sollte der Moment gekommen sein, wo sich DCOM- und CORBA-Programmierer einen Hochachtungsschluck auf die neuen Herrlichkeiten genehmigen (oder verzweifelt überlegen, wie lange man sie noch zwingen wird, mit ihren alten Techniken zu arbeiten).

3.3.5 Die ausführbare Datei

... hört in diesem Fall auf den Namen *webform1.aspx* und befindet sich – solange Sie es bei den Standardvorgaben belassen haben – im Ordner *wwwroot\WebApplication1*. Wenn Ihr Entwicklungssystem über ein LAN mit anderen Systemen verbunden ist, sollten Sie die Seite einmal von einem der anderen Systeme aus über den Internet Explorer abrufen. Als erster Teil des URL lässt sich gegebenenfalls auch die IP-Adresse des Entwicklungssystems verwenden – beispielsweise:

```
http://192.168.0.3/webapplication1/webform1.aspx
```

3.4 Fehlersuche

Was sich heutzutage integrierte Entwicklungsumgebung nennen darf, muss nicht nur Editor, Compiler, Linker und Online-Dokumentation (vulgo: Hilfestellung) nahtlos integrieren, sondern auch einen Debugger, der idealerweise sowohl auf Quelltext- als auch auf Maschinenebene funktioniert. VS.NET genügt diesen Forderungen in vorbildlicher Weise, wobei die Ebene des Maschinencodes hier *nicht* den Symbolen und Befehlen der Intermediate Language entspricht, wie es eigentlich zu erwarten wäre, sondern echten, vom JIT-Compiler erzeugten Prozessorbefehlen.

3.4.1 Haltepunkte

Das Setzen von Haltepunkten ist auch in einem laufenden Programm möglich, sofern dieses Programm aus VS.NET heraus und mit dem Befehl DEBUGGEN/STARTEN – ersatzweise F5 – gestartet worden ist.

Um einen Haltepunkt an beliebiger Stelle eines Quelltexts einzufügen, setzen Sie entweder den Cursor auf die jeweilige Zeile und verwenden die Taste F9 oder den Befehl HALTEPUNKT EINFÜGEN aus dem per *Rechtsklick* aufzurufenden Kontextmenü. Alternativ tut es auch ein *Linksklick* in den linken Fensterrahmen auf der Höhe der jeweiligen Zeile (zu klicken ist in diesem Fall exakt dort, wo in Abbildung 3.20 der Kreis erscheint). Mit einem Linksklick in diesen Kreis bzw. der Taste F9 wird man den Haltepunkt übrigens auch wieder los.

Linksklicks vor einer Quelltextzeile sind an sich der schnellste Weg zum Setzen (und Entfernen) von Haltepunkten. Unerfreulicherweise bietet die aktuelle Version von VS.NET nur über Umwege die Möglichkeit zur Festlegung von Eigenschaften eines auf diese Weise gesetzten Haltepunkts. Wenn Sie die Unterbrechung des Programmlaufs von Bedingungen abhängig machen wollen, müssen Sie den Befehl NEUER HALTEPUNKT... des Kontextmenüs oder Strg+B benutzen.

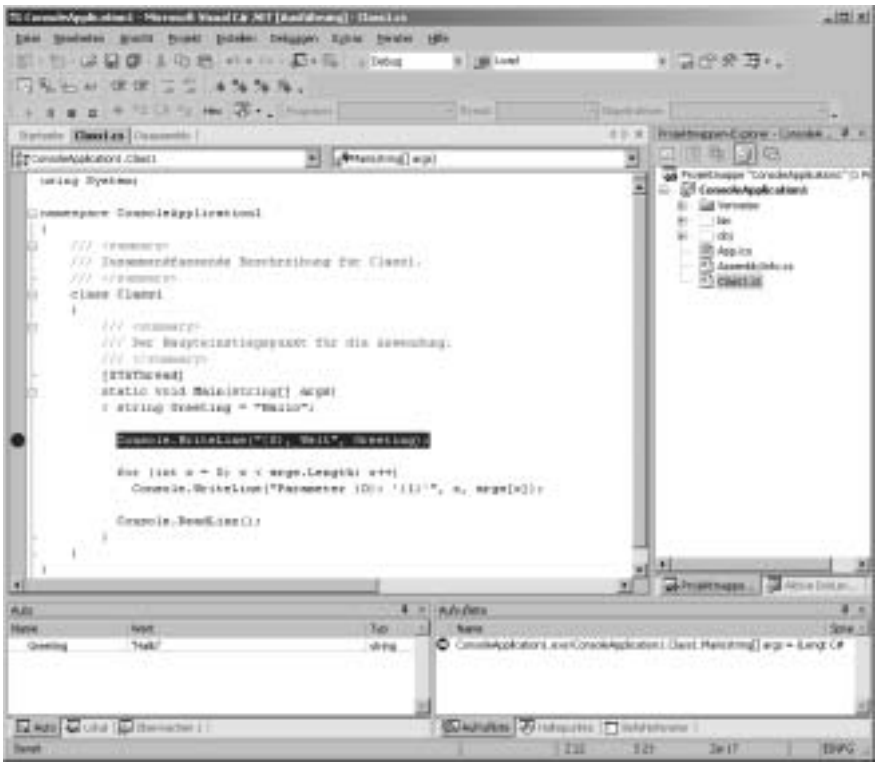


Abbildung 3.20: VS.NET beim Erreichen eines zuvor gesetzten Haltepunkts

Nachdem VS.NET das Programm an einem Haltepunkt unterbrochen hat, bekommen Sie in einem Fenster namens AUFRUFLISTE die Hierarchie der Methodenaufrufe zu sehen, die zu diesem Haltepunkt geführt haben. (Im gegebenen Beispiel enthält diese Hierarchie gerade einmal ein Element, nämlich die Methode `Main` – vgl. Abbildung 3.20.)

Das – offensichtlich erst nach einer Unterbrechung zu erreichende – Fenster HALTEPUNKTE listet die einzelnen Haltepunkte auf und erlaubt per Kontextmenü die Veränderung ihrer Eigenschaften, beispielsweise durch eine (in C# formulierte) Abbruchbedingung oder einen Wiederholungszähler.

Untersuchen von Variablen

Im Fenster AUTO zeigt VS.NET automatisch (daher der Name) die Werte der Variablen an, die in der aktuellen Anweisung erscheinen. Wenn Sie sämtliche lokalen Variablen der aktuellen Methode im Blick haben wollen, klicken Sie auf den Reiter LOKAL. Strukturierte Variablen werden in beiden Fenstern mit einem vorangestellten Pluszeichen angezeigt; ein Klick darauf stellt die einzelnen Elemente (oder Felder, je nach Variablentyp) dar – vgl. Abbildung 3.21.

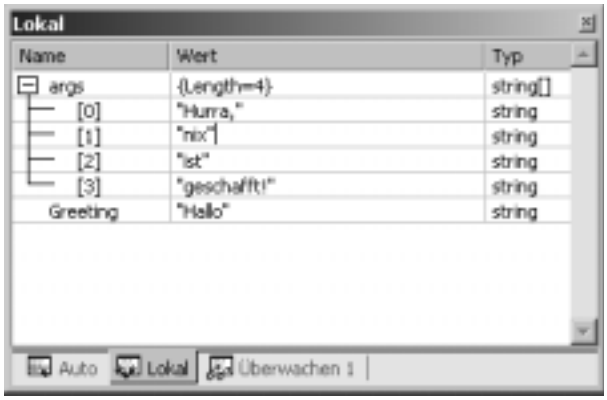
Abbildung 3.21: Lokale Variablen nach Erreichen eines Haltepunktes

Wenn Sie eine Variable permanent im Blick behalten wollen, verwenden Sie das Fenster ÜBERWACHEN. *Dessen Bedienung lässt zwar erst einmal stutzen, weil man im Kontextmenü (wie auch in den beiden anderen Fenstern) einen Befehl* NEU *vergeblich sucht. Des Rätsels Lösung: Klicken Sie einfach ins linke Feld der ersten Zeile, und tragen Sie den gewünschten Variablennamen ein.*

Ändern von Variablen

Das willkürliche Ändern eines Variablenwerts zu Testzwecken funktioniert in allen drei Fenstern (AUTO, LOKAL und ÜBERWACHEN) auf dieselbe, nach kurzer Eingewöhnungszeit doch recht intuitive Weise: Klicken Sie in der Spalte *Wert* an die gewünschte Stelle, und tragen Sie den neuen Wert ein. Bei Strings ist in diesem Fall darauf zu achten, dass die Anführungszeichen erhalten bleiben – vgl. Abbildung 3.22.

Abbildung 3.22: Manuelle Änderung des Wertes einer Variablen



Um auf die Schnelle einen einzelnen, im Fenster AUTO nicht angezeigten Variablenwert zu untersuchen oder zu verändern, kann man schließlich die SCHNELLÜBERWACHUNG benutzen, die über das Menü DEBUGGEN bzw. den Tastenbefehl Alt+Strg+Q erreichbar ist.

Fortsetzen des Programms

Zur Fortsetzung eines per Haltepunkt unterbrochenen Programms bietet VS.NET mehrere Funktionen, die ihrerseits wiederum auf zwei oder gar drei Wegen erreichbar sind:

- *Einzelschritt* – führt die jeweils nächste Anweisung aus. Wenn es dabei um den Aufruf einer Methode geht, deren Quelltext verfügbar ist, wird lediglich die erste Anweisung dieser Methode ausgeführt. Erreichbar über DEBUGGEN/EINZELSCHRITT, das Symbol »Einzelschritt« in der Menüleiste und den Tastenbefehl F11.
- *Prozedurschritt* – unterscheidet sich von Einzelschritt nur in einem Punkt: Methodenaufrufe, die in der jeweils nächsten Anweisung enthalten sind, werden en bloc ausgeführt. Erreichbar über DEBUGGEN/PROZEDURSCHRITT, das Symbol »Prozedurschritt« in der Menüleiste und den Tastenbefehl F10.
- *Ausführen bis Rücksprung* – führt die aktuelle Methode bis zu einem beliebigen `return` en bloc aus und hält das Programm dann an. Erreichbar über DEBUGGEN/AUSFÜHREN BIS RÜCKSPRUNG, das Symbol »Ausführen bis Rücksprung« in der Menüleiste und den Tastenbefehl ⇧+F11. (Delphi-Programmierer, die gerne ein halbes Dutzend *Exit*-Anweisungen verstreuen, vermissen ein Äquivalent dieses Befehls immer noch. Nach unserer Meinung ja zu Recht, weil Methoden mit einer Vielzahl von »Ausgängen« unübersichtlich sind – aber hier kann C# seine Verwandtschaft zu C/C++ eben doch nicht verleugnen.)
- *Weiter* – setzt die Ausführung bis zum nächsten Haltepunkt oder dem regulären Ende des Programms fort. Erreichbar über DEBUGGEN/WEITER, das Starttastensymbol in der Menüleiste und den Tastenbefehl F5 (der während einer Debuggersitzung also nicht die Bedeutung »Programm abbrechen und neu starten« hat, wie man vermuten könnte).
- *Debuggen beenden* – bricht die Ausführung ab und wirft das Programm aus dem Hauptspeicher (was mit der Freispeicherverwaltung von .NET auch spurlos klappt – bei unverwaltetem Code, der Systemressourcen belegt, muss man sich bei diesem Befehl dagegen auf die Verwaltung von Windows verlassen). Erreichbar über DEBUGGEN/DEBUGGEN BEENDEN, das Stoptastensymbol in der Menüleiste und den Tastenbefehl ⇧+F5.

Mit dem Pausetastensymbol in der Menüleiste bzw. Strg+Pause *lässt sich der Lauf eines (unter Kontrolle des Debuggers gestarteten) Programms sozusagen ungezielt unterbrechen – beispielsweise, wenn das Programm in eine Endlosschleife geraten ist.*

Dank des verwalteten Codes haben Sie bei C# eine hohe Chance, nach einer derartigen Unterbrechung tatsächlich die fehlerhafte Stelle im Quelltext angezeigt zu bekommen. (Bei Delphi und Microsoft C/C++ landet man mit solchen Stopversuchen leider meist in irgendeiner Routine des Systemkerns.)

3.5 Konfigurationen

Wie wohl auch nicht anders zu erwarten, verfügt VS.NET über eine Vielzahl an Einstellmöglichkeiten, bleibt aber bei allen Sprachen – der neu entwickelten, schlanken Laufzeitumgebung .NET sei Dank – zumindest in der aktuellen Version noch weit bescheidener als Microsoft C/C++ (von dem ja nicht nur Spötter behaupten, es verfüge garantiert auch über einen Kommandozeilenschalter, mit dem man den Prozessor rückwärts laufen lassen kann). Die folgenden Abschnitte verstehen sich dennoch weiß Gott nicht als Referenz, sondern beschränken sich bewusst auf die Dinge, die man bei der Einarbeitung in VS.NET über kurz oder lang sucht.

3.5.1 Debug und Release

Die Voreinstellung für neue Projekte ist sinnvollerweise der Debug-Modus: Der Compiler erzeugt zusätzliche Informationen für den Debugger, die ausführbare Datei wird im Unterordner *bin\Debug* abgelegt. Im Release-Modus fehlen die zusätzlichen Informationen, die ausführbare Datei landet im Unterordner *bin\Release* des Projekts.

Das Umschalten zwischen den beiden Modi geschieht *ausschließlich* über den Menübefehl ERSTELLEN/KONFIGURATIONS-MANAGER (siehe Abbildung 3.23), mit dem sich bei Bedarf auch weitere, beliebig benannte Konfigurationen wie beispielsweise Spezialversionen eines Programms für den firmeninternen Bedarf anlegen lassen.

Für die Festlegung, was im einen und was im anderen Modus geschieht, ist nicht der Konfigurations-Manager zuständig – das läuft über die Eigenschaften des Projekts, die über das Kontextmenü des Projektnamens im Projektmappen-Explorer erreichbar sind. Dort festgelegte *Allgemeine Eigenschaften* gelten für jede Konfiguration (weshalb das Listenfenster für die Konfiguration auch deaktiviert ist, solange Sie sich in diesen Eigenschaften bewegen). Nach Auswahl der Unterabteilung *Konfigurationseigenschaften* ist das Listenfenster zur Auswahl der Konfiguration aktiviert (siehe Abbildung 3.24).

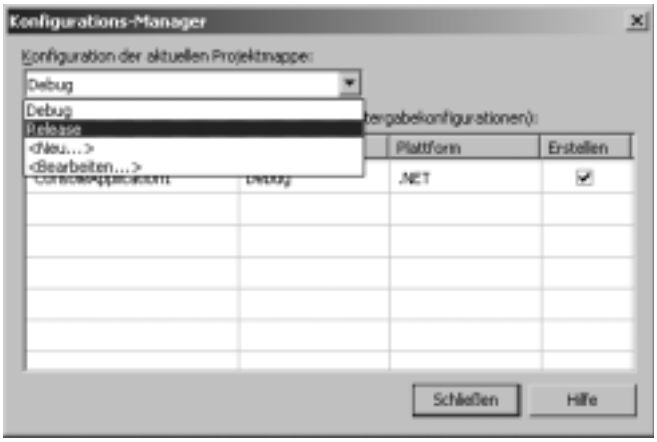


Abbildung 3.23: Umschalten zwischen Debug und Release über den Konfigurations-Manager

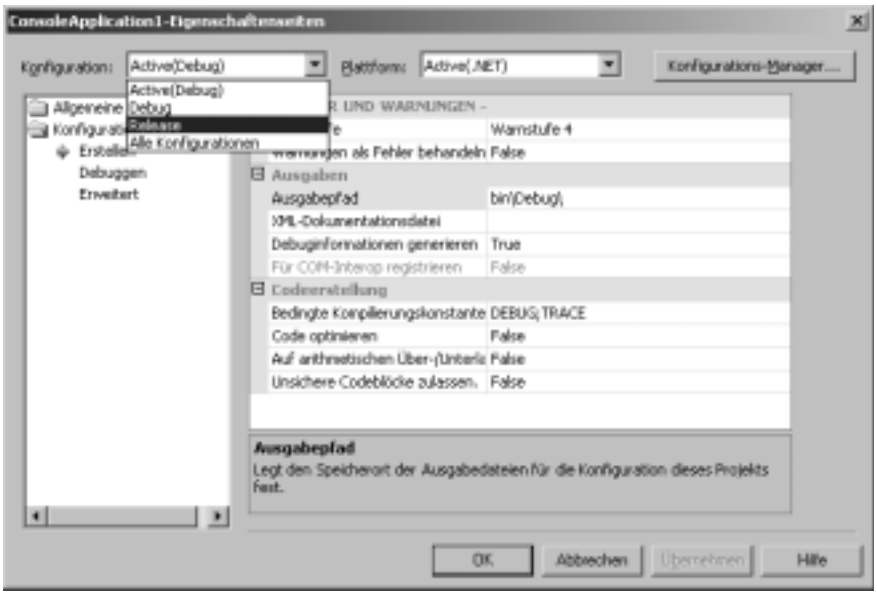


Abbildung 3.24: Festlegen der Einstellungen für die Konfigurationen über die Projekt-eigenschaften

Bei der ersten Compilierung nach der Umschaltung in den Release-Modus legt VS.NET zwei weitere Unterordner für das Projekt an: *bin\Release* und *obj\Release*. In *bin\Release* wird schließlich die ausführbare EXE-Datei gespeichert.

Wie durch den Zusatz »Active« im Listenfenster auch angedeutet, geht es hier nicht um das Umschalten zwischen Konfiguration, sondern um das Festlegen ihrer Eigenschaften. Für die Auswahl der aktiven Konfiguration ist wie gesagt ausschließlich der Konfigurations-Manager zuständig, der konsequent in den Eigenschaftsseiten auch noch einmal eine eigene Schaltfläche spendiert bekommen hat.

:-)
TIPP

Wenn Sie Eigenschaften festlegen wollen, die für beide Compilierungsmodi gleichermaßen gelten, wählen Sie vorher in den Projekteigenschaften Alle Konfigurationen (vgl. Abbildung 3.24). Ansonsten behandelt VS.NET sämtliche Angaben – wie beispielsweise Kommandozeilenparameter – für die Debug- und Release-Konfiguration als individuell.

3.5.2 Aktionen beim Start von VS.NET

Der weitaus größte Teil der Einstellmöglichkeiten für VS.NET selbst verbirgt sich in einem Dialogfeld, das über den Menübefehl EXTRAS/OPTIONEN zu erreichen ist (siehe Abbildung 3.25).

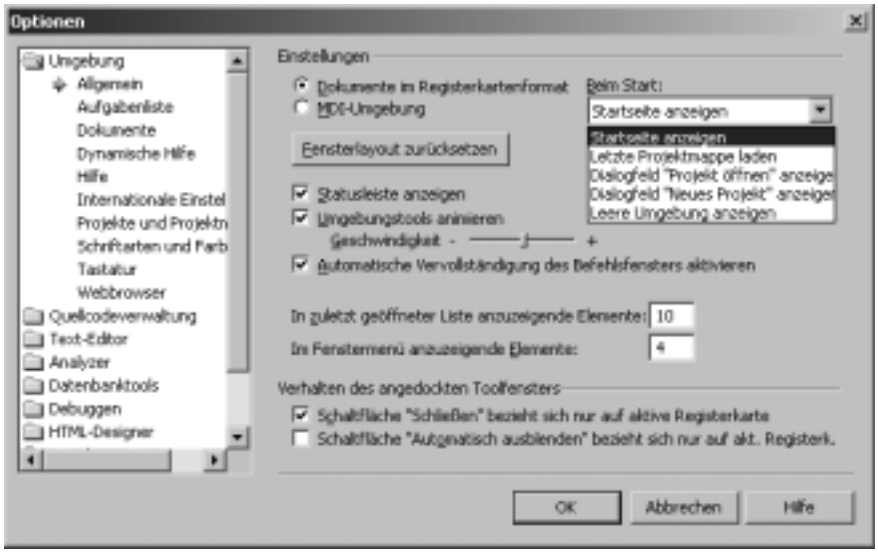


Abbildung 3.25: Die Startoptionen im Dialogfeld *Extras/Optionen*

Für das Studium dieses Dialogfelds sollte man sich auf jeden Fall einen Moment Zeit nehmen: Über die Abteilung *Text-Editor* lässt sich beispielsweise festlegen, ob, wie heftig, und mit was (nämlich Leerzeichen oder Tabs) die automatische Einrückung zuschlagen soll.

Eine Option zur automatischen Speicherung aller bearbeiteten Dateien vor dem Start eines neu erstellten Programms aus VS.NET heraus werden Sie übrigens vergeblich suchen – einfach deshalb, weil die Entwicklungsumgebung grundsätzlich alles Veränderte in Sicherheit bringt, bevor sie irgendein anderes Programm startet. (In die Kategorie »anderes Programm« fallen hier auch die über das Menü EXTRAS zu erreichenden Hilfsprogramme wie Spy++ oder ILDASM.)

Und schließlich gibt es auch noch einen Wermutstropfen im Zusammenhang mit Dateien: Sicherungskopien von Quelltexten (á la *Form1.cs.bak* oder *Form1.~cs*) haben die Entwickler von Visual Studio leider schon immer für überflüssig gehalten.