Quellcode auf CD

Scripting .NET

Customizing Lösungen für .NET-Anwendungen mit JScript

von Karsten Strobel

„Was nicht passt, wird passend gemacht“ ist ein bekannter Spruch vom Bau. Dieses kreative Motto greift immer dann, wenn der Architekt mal nicht daran gedacht hat, in die Gästetoilette ein Fenster einzuzeichnen oder wenn er vielleicht die Garage gleich ein paar Zentimeter über die Grundstücksgrenze hinaus geplant hat. Kein Problem, denn für erfahrene Handwerker gibt es fast nichts, was sich nicht schnell noch vor Ort korrigieren ließe. Auch Softwareentwickler handeln gerne nach diesem Kredo. Schwierig wird es allerdings bei Standardsoftware, quasi den Fertighäusern der IT-Zunft. „Customizing“ heißt hier das Zaubermittel und Scripting ist das mächtigste Werkzeug dieser Handwerkskunst.

Kundenspezifische Anpassungen sind für viele Softwareentwickler die (professionelle) Existenzberechtigung Nummer eins. Sie schaffen für ihre Anwender Lösungen, die man so nicht von der Stange kaufen kann. Was einerseits ein Segen für das Geschäft ist, entpuppt sich mitunter andererseits als technische Sackgasse, wenn am laufenden Band individuelle Programme geschaffen werden, die sich mangels einheitlicher Codebasis nur schwer warten und weiterentwickeln lassen. *Das* universelle Produkt zu schaffen ist wiederum kaum möglich, lebt man doch davon, dass es diese gar nicht geben kann. Deshalb sind Lösungen, die eine weitgehende Konfigurierbarkeit einer Software erlauben, zum Beispiel in Form grafischer Druckvorlagen-Editoren oder der immer wieder anzutreffenden reservierten Datenbankfelder zur sonstigen Verwendung, so verbreitet.

Will man aber auch für das Unvorhersehbare vorsorgen, ohne von Fall zu Fall doch Sonderversionen anfertigen zu müssen, dann kann man die eigene Software mit Scriptingfunktionen ausstatten, um Anpassungen auch nach Fertigstellung eines Releases noch per Programmcode – und damit sehr flexibel – nachreichen zu können, ohne das eigentliche Produkt ändern zu müssen. Zunächst sollte man entscheiden, ob die Skriptprogrammierung nur vom Entwickler bzw. geschultem Personal oder von jedem Anwender ausgeführt werden soll. Ist letzteres der Fall, steigt der Aufwand für Dokumentation und Editier-Komfort natürlich erheblich und es sind höhere Maßstäbe in puncto Sicherheit zu setzen.

Mit Visual Studio for Applications (VSA) bietet Microsoft bzw. dessen Partnerfirma Summit Software [1] eine sehr professionelle IDE einschließlich Debugger für eine endkundentaugliche Scripting-Integration in .NET-Anwendungen an. Die ziemlich hohen Lizenzgebühren lassen diese Komfortvariante leider oft ausscheiden. Glücklicherweise sind die Runtime-Komponenten von VSA, also die Objekte zum Kompilieren und Ausführen von Skripten, bereits im .NET Framework enthalten, sodass man sie – ohne die glanzvolle Oberfläche – zum Nulltarif nutzen kann.

Man nehme...

Für eine Scripting-Lösung benötigt man zwei Zutaten: Eine Scripting Engine und den Scripting Host. Der Host ist nichts anderes als Ihr eigenes Programm. Von Hause aus stehen dem Host zwei Engines zur Verfügung, nämlich für Visual Basic .NET und für JScript.NET. Da diese Engines abgesehen von der jeweiligen Sprache noch ein paar andere Unterschiede aufweisen, sollten Sie sich für eine entscheiden. Ich verwende JScript.NET, weil die Sprache meinen durch C# geprägten Gewohnheiten näher kommt und weil die Engine in einigen Punkten einfacher zu handhaben ist.

Gegenüber der aus der COM-Welt bekannten Scripting Engine hat sich einiges geändert. Die Spracheigenschaften von VB und JScript wurden umgebaut und erweitert, um in das Korsett des .NET Frameworks zu passen. Man kann sogar sagen, dass der Begriff „Skript“ nicht mehr ganz zutreffend ist, denn die Programme werden nicht mehr interpretiert, sondern in native IL-Assemblies übersetzt und nicht anders ausgeführt als „echte“ .NET-Programme, was einen erheblichen Zugewinn an Performance und Stabilität bedeutet. Der größte Vorteil, der sich aus dem Architekturwechsel ergibt, ist die quasi nahtlose Integrierbarkeit der Scriptingebene in die Host-Anwendung.

Um ein Skriptprogramm auszuführen, muss der Host zunächst eine Instanz der Scripting Engine, auch Skriptmodul genannt, erzeugen. Dieses stellt dem Host eine *IVsaEngine*-Schnittstelle (siehe Listing 1) zur Verfügung. Der Host muss seinerseits eine *IVsaSite*-Schnittstelle implementieren:

```
public interface Microsoft.Vsa.IVsaSite
{
 void GetCompiledState(out Byte[] pe, out Byte[] debugInfo);
 object GetEventSourceInstance(string itemName, string
                                        eventSourceName);
 object GetGlobalInstance(string name);
 void Notify(string notify, object info);
 bool OnCompilerError(Microsoft.Vsa.IVsaError error);
}
```

und diese dem Skriptmodul über die Eigenschaft *IVsaEngine.Site* bekannt machen. Damit stehen Host und Skriptmodul in einer Wechselbeziehung, die, vereinfacht gesagt, so aussieht: Der Host versorgt das Skriptmodul mit dem Quellcode und meldet eigene Klassen als globale Elemente an, auf die das Skriptprogramm zugreifen darf. Der Host kann gezielt Funktionen des von ihm bereitgestellten Skripts aufrufen. Umgekehrt ruft das Skriptmodul Methoden des Hosts auf, um Fehler beim Kompilieren oder bei der Skript-Ausführung anzuzeigen, um Instanzen der vorher angemeldeten globalen Elemente zu erfragen oder um Statusinformationen abzurufen.

Vom Ziel aus wandern

Alles ziemlich theoretisch? Finde ich auch. Damit Sie erst einmal eine Vorstellung von dem Ziel erhalten, das ich verfolge, wenn ich gleich die einzelnen Schritte zur Implementierung einer Scripting-Anwendung darstelle, möchte ich mir ausnahmsweise selbst vorgreifen und Ihnen gleich zu Anfang die fertige Beispielanwendung vorstellen. Da sich aus Platzgründen nicht der ganze Quellcode abdrucken lässt, empfehle ich Ihnen, diesen auf der beiliegenden CD einzusehen und damit selbst zu experimentieren.

Abbildung 1 zeigt das Hauptfenster dieser Mini-Anwendung. Als Beispiel dient hier eine ganz einfache Preisberechnung. Der Anwender soll eine Menge, einen Stückpreis, Versandkosten und einen Rabattsatz eingeben. Heraus kommt ein daraus berechneter Gesamtpreis. Diese Aufgabe ist natürlich trivial, aber gerade bei solchen Preisberechnungen gibt es in der Praxis oft Anpassungsbedarf. So könnte man zum Beispiel auf die Idee kommen, dass der Rabatt zwar auf den Wert der Ware, aber nicht auf die Versandkosten gewährt werden soll. Nach dem Aktivieren des JScript-Programms, das bei diesem Beispiel über das Scripting-Fenster (Abb. 2) schon mitgeliefert wird und nur noch gestartet werden muss, ändern sich Funktion und Aussehen der Preisberechnung (Abb. 3) ein wenig.

Um Ihr Produkt mit solchen Möglichkeiten auszustatten, müssen Sie Ihre Anwendung zu einem Scripting Host machen, der die Scripting Engine erzeugt und mit dieser zusammenarbeitet. Es ist sinnvoll, hierfür eine eigene Klasse zu programmieren. Bei unserem Beispielprogramm geschieht dies mit der Klasse ScriptHost in der Datei *ScriptHost.cs* (siehe Listing 2, das sich im Begleitquellcode auf der beiliegenden CD findet). Im Konstruktor des *ScriptHost*-Objekts wird mit

```
meineEngine = new Microsoft.JScript.Vsa.VsaEngine();
```

eine Instanz der Scripting Engine für JScript erzeugt, die die Schnittstelle *IVsaEngine* unterstützt. Ein Zeiger auf dieses Interface, über das alle weiteren Zugriffe auf die Engine geschehen, landet in der privaten Variable *meineEngine*. Die weiteren Initialisierungsschnitte sind:

- Zuweisen eines Strings auf die Eigenschaft *RootMoniker*. Dies sollte immer die erste Aktion nach dem Erzeugen der Engine sein. Diese Eigenschaft dient zur eindeutigen Identifikation der Instanz des Skriptmoduls. Der String muss dem Format *protokoll://pfad* entsprechen, also die Syntax eines URI aufweisen. Das etwas Schrullige daran ist, dass Sie sowohl Protokoll als auch Pfad frei erfinden können (und sollen), solange es nur eindeutig ist und nicht gegen die Syntaxregeln verstößt.
- Zuweisen der *IVsaSite*-Schnittstelle zu der *Site*-Eigenschaft der Engine. Diese Schnittstelle wird die Engine nutzen, um den Host aufzurufen, um zum Beispiel Kompilierfehler anzuzeigen. In unserem Beispiel wird *self* zugewiesen, weil das *ScriptHost*-Objekt die *IVsaSite*-Schnittstelle implementiert.
- Aufruf der Methode *InitNew()*. Dies soll immer nach der Zuweisung von *RootMoniker* und *Site* geschehen.
- Setzen von Optionen mit der Methode *SetOption()*. In meinem Beispiel werden die *Fast*- und die *autoRef*-Optionen auf *true* gesetzt. *Fast* weist das JScript-Modul an, auf einige Features der Sprache, wie zum Beispiel die Verwendung undeklarierter Variablen, zugunsten einer besseren Geschwindigkeit zu verzichten. Die Option *autoRef* erlaubt es der Engine, die im Skript mit der *import*-Direktive angegebenen Assemblies automatisch einzubinden. Nach dieser Option habe ich lange suchen müssen, weil sie sehr schlecht dokumentiert ist.
- Setzen der Eigenschaft *GenerateDebugInfo* auf *true*. Dies ist in der .NET Frame-

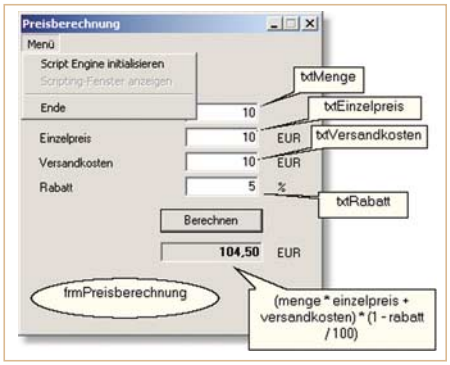

Abb. 1: Beispielanwendung vor Starten des Skripts

Listing 1

```
public interface Microsoft.Vsa.IVsaEngine
{
  Assembly Assembly { get; }
  Evidence Evidence { get; set; }
  bool GenerateDebugInfo { get; set; }
  bool IsCompiled { get; }
  bool IsDirty { get; }
  bool IsRunning { get; }
  IVsaItems Items { get; }
  string Language { get; }
  int LCID { get; set; }
  string Name { get; set; }
  string RootMoniker { get; set; }
  string RootNamespace { get; set; }
  IVsaSite Site { get; set; }
  string Version { get; }

  void Close();
  bool Compile();
  object GetOption(string name);
  void InitNew();
  bool IsValidIdentifier(string identifier);
  void LoadSourceState(Microsoft.Vsa.IVsaPersistSite
                                                site);
  void Reset();
  void RevokeCache();
  void Run();
  void SaveCompiledState(out Byte[] pe, out Byte[] pdb);
  void SaveSourceState(Microsoft.Vsa.IVsaPersistSite
                                                site);
  void SetOption(string name, object value);
}
```

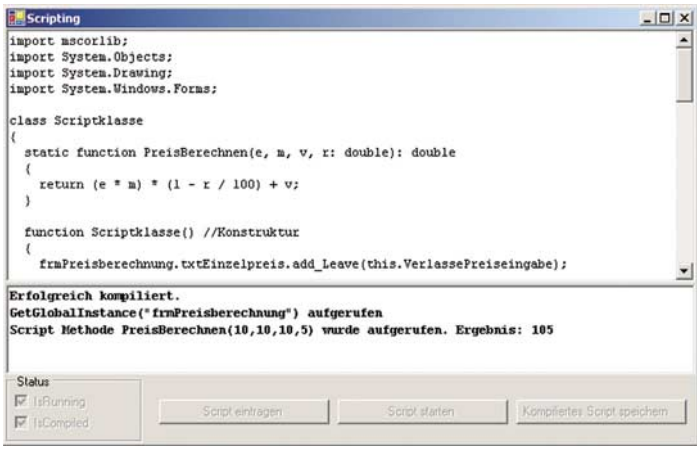

Abb. 2: Einfacher Skripteditor

work-Dokumentation so beschrieben, als bräuchte man diese Einstellung nur für Debug-Zwecke. Es hat sich aber herausgestellt, dass ohne die Debug-Informationen kein Aufruf von Methoden des Hosts durch das Skript möglich ist.

- Festlegen der *Name*- und der *Root-Namespace*-Eigenschaften. Hier müssen gültige Bezeichner eingetragen werden. *Name* kann beliebig gewählt werden und muss nur innerhalb der Hostanwendung eindeutig sein, falls mehrere Scripting Engines erzeugt werden (was selten vorkommen dürfte). *RootNamespace* legt die Bezeichnung des Stamm-Namensraums des Skripts fest. Auch diese Einstellung können sie willkürlich festlegen.

Nachdem die Scripting Engine auf diese Weise initialisiert ist, muss sie gefüttert werden, damit sie etwas Nützliches tun kann. Das Futter kommt in Form von Einträgen in die *Items*-Aufzählungseigenschaft, die von der *IVsaEngine*-Schnittstelle angeboten wird. Über solche Items werden Quellcode-Elemente und globale Objekte an die Engine übergeben. Erzeugt werden die Einträge mit der Methode *CreateItem*, die vom *Items*-Objekt angeboten wird und folgende Signatur hat:

```
IVsaItem CreateItem(string name, VsaItemType itemType,
                    VsaItemFlag itemFlag);
```

Als Funktionsergebnis erhält der Aufrufer also eine Referenz auf eine *IVsaItem*-Schnittstelle. Dies ist aber nur ein Basistyp, von dem drei verschiedene speziellere Schnittstellen abgeleitet sind, sodass das Ergebnis entsprechend gecastet werden kann. Der Typ des Items und damit auch der tatsächliche Typ der zurückgelieferten Schnittstellen hängt von dem Parameter *itemType* ab, dem einer der drei Aufzählungswerte *VsaItemType.AppGlobal*, *VsaItemType.Code* oder *VsaItemType.Reference* übergeben werden kann. Ein Aufruf mit der Typangabe *VsaItemType.AppGlobal* erzeugt ein Objekt, das die Schnittstelle *IVsaGlobalItem* unterstützt. Bei *VsaItemType.Code* wird eine *IVsaCodeItem*-Schnittstelle geliefert. In unserem Beispiel kommen diese beiden Item-Typen vor, aber kein *VsaItemType.Reference* (liefert *IVsaReferenceItem*), mit dem es möglich ist, Verweise auf Assemblies, die das Skriptmodul laden soll, zu spezifizieren. Auf diese recht umständliche Art der Referenzierung können wir dank der oben erwähnten Option *autoRef* verzichten, solange sich die vom Skript verwendeten Assemblies im GAC befinden. Der *itemFlag*-Parameter von *CreateItem()* hat für die JScript-Engine keine Bedeutung und kann immer mit *VsaItemFlag.None* angegeben werden.

In meinem Beispiel wird ein globales Item von dem *ScriptHost*-Objekt bei Aufruf der Methode *LerneGlobalesObjekt()*, ein Code Item von der Methode *ScriptEintragen()* erzeugt (siehe nochmals Listing 2 auf der CD). In beiden Fällen wird *meineEngine.Items.CreateItem()* aufgerufen, aber es wird jeweils ein anderer *itemType* übergeben. Das Ergebnis wird entsprechend im ersten Fall als *IVsaGlobalItem*, im zweiten Fall als *IVsaCodeItem* interpretiert. Über die zurückgegebene Schnittstelle werden dann die Eigenschaften des Items belegt.

Was ein Code Item sein soll, ist nicht schwer zu erraten, zumal Sie in der *ScriptEintragen()*-Methode sehen, dass der *string*-Parameter *strQuellcode* der *SourceText*-Eigenschaft des *IVsaCodeItem* zugewiesen wird. Ein globales Item ist eine benannte Referenz auf einen Objekttyp, den das Skript verwenden kann. Wenn das Skript davon Gebrauch macht, erkundigt sich die Scripting Engine durch einen Aufruf der Schnittstellenmethode

```
object IVsaSite.GetGlobalInstance(string name)
```

unter Angabe des Namens nach der zu verwendenden Instanz. Was heißt das alles? Werfen Sie einen Blick auf die Codezeilen, mit denen die Anwendung das *ScriptHost*-Objekt erzeugt. Dies geschieht, wenn Sie im Preiseingabeformular den Menüeintrag SCRIPT ENGINE INITIALISIEREN anklicken:

```
private void miScriptEngineInit_Click(object sender, System.
                                      EventArgs e)
{
 myScriptHost = new ScriptHost();
 myScriptHost.LerneGlobalesObjekt(this, this.Name);
 /*...*/
}
```

Sie sehen hier, dass – nach dem Erzeugen des Script Hosts – die Anwendung durch Aufruf von *LerneGlobalesObjekt(this, this.Name)* das *Form*-Objekt (*this*) unter dem in der *Name*-Eigenschaft des Formulars angegebenen Namen (nämlich *frmPreisberechnung*) als globales Objekt anmeldet. Der Name wird innerhalb von *LerneGlobalesObejekt()* als *name*-Parameter an *CreateItem()* übergeben und der Typ des *Form*-Objekts wird der *TypeString*-Eigenschaft des dabei erzeugten *IVsaGlobalItem* zugewiesen. Damit kennt die Engine den Typ, nicht aber die Objektinstanz. Das *ScriptHost*-Objekt merkt sich daher die Verbindung zwischen Namen und Objektinstanz in der Sorted List *globalObjList* und sucht diese anhand des Namens wieder heraus, wenn die Engine die *IVsaSite.GetGlobalInstance()*-Methode aufruft. Durch diese Vermittlung wird es dem Skript ermöglicht, auf bestimmte Objekte der Host-Anwendung zuzugreifen.

Startklar machen

Der Quellcode wird, wie gesagt, von *ScriptHost.ScriptEintragen()* als Code Item angemeldet. Sie sehen hier, dass beim Aufruf von *CreateItem()* der vorher als *RootNamespace* festgelegte String als Name

für das Item übergeben wird. Das ist eine völlig willkürliche Entscheidung meinerseits, denn Sie dürfen einen beliebigen Namen verwenden, solange dieser für alle Code Items eindeutig ist. Da mein Scripting Host nur ein einziges Code Item unterstützt (obwohl mehrere möglich wären), brauche ich mir über die Eindeutigkeit natürlich keine Sorgen zu machen. Meine Beispielanwendung trägt den Skriptcode ein, sobald Sie in dem Scripting-Fenster (Abb. 2) die entsprechende Schaltfläche anklicken. Das kann mehrmals hintereinander geschehen, denn in *ScriptHost.ScriptEintragen()* wird das vorherige Code Item vor dem Eintragen des neuen gegebenenfalls verworfen. Anschließend wird *meineEngine.Compile()* aufgerufen und die Scripting Engine wandelt den Quellcode in IL-Code um. Falls es beim Kompilieren zu Fehlern kommt, ruft die Engine die *IVsaSite.OnCompilerError()*-Methode auf, um dem Host Gelegenheit zu geben, den Fehler anzuzeigen. Das geschieht in meinem Beispiel indirekt, indem der *LogCompileError*-Delegate aufgerufen wird, der im Scripting-Fenster die Fehlermeldung in die *lbLog*-List Box einträgt. Bei fehlerfreier Kompilierung liefert *meineEngine.Compile()* *true* zurück und die *IsCompiled*-Eigenschaft zeigt ebenfalls *true* an. Die Ampel steht also auf Grün. Wir müssen nur noch losfahren, was durch den Aufruf der Methode *meineEngine.Run()* geschieht, sobald die Schaltfläche SCRIPT STARTEN betätigt wird.

Haben Sie auch schon einen Blick auf den Quelltext des Skripts in Listing 3 (ebenfalls im Begleitcode auf der CD) geworfen? Für diejenigen, die mit JScript.NET nicht vertraut sind, empfehle ich die Lektüre der Einführung in diese Sprache in der SDK-Dokumentation [2]. Das Skript tritt erst dann in Aktion, wenn der Host eine Funktion darin aufruft. Meines Wissens ist dies nur durch Aufruf einer Objektmethode möglich. Da der Host keine Instanzen der im Skript programmierten Klassen besitzt, bietet sich eine statische Objektmethode als Einsprungspunkt für den Host an, da für so einen Aufruf keine Instanz erforderlich ist.

Wie Sie sehen, definiert das Beispielskript eine Klasse namens *Scriptklasse*, deren erste Funktion die statische Methode *PreisBerechnen* ist. Beim Betätigen der Schaltfläche BERECHNEN im Hauptformular kommt folgender Code zur Ausführung:

```
//...
if ((myScriptHost != null) && myScriptHost.IsRunning)
{
 object[] par = {menge, einzelpreis, versandkosten, rabatt};
 script_result = myScriptHost.ScriptMethodeAufrufen
                                        ("PreisBerechnen", par);
 if (script_result is double) preis = (double)script_result;
                                        else script_result = null;
}
if (script_result == null)
 preis = (menge * einzelpreis + versandkosten) *
                                        (1 - rabatt / 100);
```

Wenn also *myScriptHost* initialisiert und das Skriptprogramm mit *Run()* gestartet worden ist, wird mit diesen Codezeilen die Methode *myScriptHost.ScriptMethodeAufrufen()* aufgerufen, wobei der Methodenname und ein Array mit den Aufrufparametern (die auf die Skriptimplementierung abgestimmt sein müssen) mit angegeben werden. Sehen Sie sich die Implementierung von *ScriptMethodeAufrufen()* an. Die Methode verwendet die Eigenschaft *Assembly* der Engine-Schnittstelle, die eine Instanz des im Namespace *System.Reflection* definierten *Assembly*-Objekts liefert. Diese Assembly ist das kompilierte Skript – ein weiterer Hinweis darauf, wie tief das VSA-Scripting in die .NET-Fundamente eingelassen ist. Der Abruf von Typinformationen für die „Scriptklasse" (die natürlich im Skriptquellcode genau so heißen muss) mit *Assembly.GetType()*, der *MethodInfo* mit *GetMethod()* und schließlich der Aufruf der Methode mit *Invoke()* unterscheiden sich nicht von der Technik, die Sie verwenden könnten, um eine mit C# oder VB.NET programmierte Assembly zu verwenden.

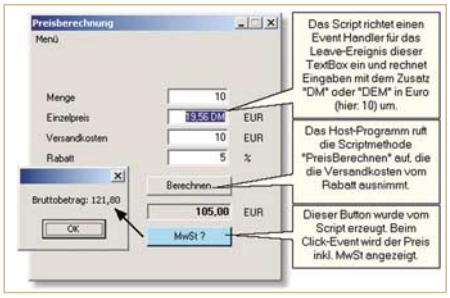

Abb. 3: Nach dem Starten des Skripts

Die Anwendung versucht also, eine Scripting-Methode für die Preisberechnung zu aktivieren. Wenn dies gelingt, wird das Berechnungsergebnis des Skripts verwendet, anderenfalls wird die fest programmierte Formel angewendet, die eventuell einen etwas niedrigeren Preis liefert. Bei dieser (klassischen) Form der Skriptanwendung liegt die Initiative beim Hostprogramm. Darin müssen die Möglichkeit des Eingriffs mittels Scripting und die Randbedingungen (welche Parameter von welchem Typ gibt es usw.) schon zum Zeitpunkt der Anwendungsprogrammierung genau geplant worden sein, damit – wie in unserem Beispiel bei Klicken auf den BERECHNEN-Button – das Skript in Aktion treten kann. Etwas eleganter, aber im Ansatz nicht wesentlich anders wäre es gewesen, für die Preisberechnung ein

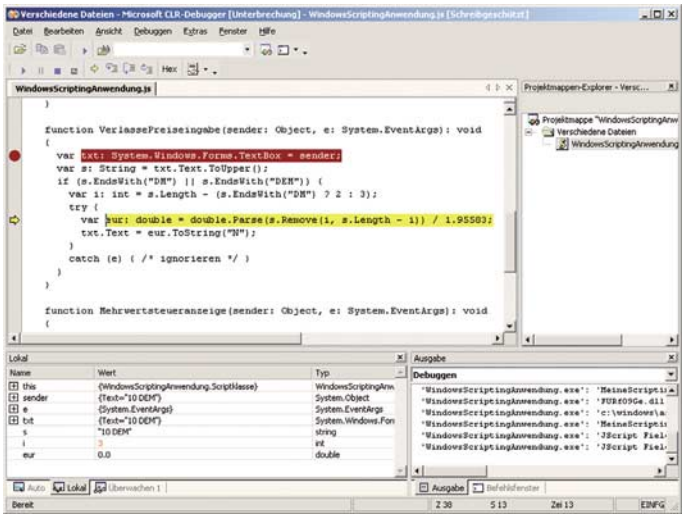

Abb. 4: Skript-Debugging

Event zu definieren und dem Skript die Gelegenheit zu geben, auf dieses Event zu reagieren. Die von mir gerade angedeutete Fähigkeit des Skripts, Ereignisse des Hostprogramms zu verarbeiten, eröffnet aber in anderer Hinsicht sehr interessante Möglichkeiten.

Ganz am Ende des JScripts sehen Sie folgenden Einzeiler:

```
new Scriptklasse();
```

Diese Programmzeile ist nicht in eine Funktion oder eine Methode eingefasst, sondern steht völlig für sich allein. Solche Anweisungen werden von der JScript-Engine unmittelbar beim Aufruf von *Run()* ausgeführt. Zweck der Übung ist, eine Instanz der *Scriptklasse* zu erstellen, wobei der Konstruktor *function Scriptklasse()* ausgeführt wird. Die erste Zeile des Konstruktors lautet:

```
frmPreisberechnung.txtEinzelpreis.add_Leave(this.
                                    VerlassePreiseingabe);
```

Darin liegen gleich drei interessante Erkenntnisse verborgen: Erstens ist dies die erste Stelle, an der auf das globale Objekt *frmPreisberechnung* zugegriffen wird, und zwar über genau den Bezeichner, der beim Erzeugen des entsprechenden globalen Items als Name angegeben worden ist. Zweitens ist hier zu sehen, dass das Skript ohne weiteres (durch späte Bindung) auch auf Objekte wie das *txtEinzelpreis-Feld* (vom Typ *TextBox*) zugreifen kann. Voraussetzung dafür ist allerdings, dass die Eigenschaft *Modifiers* für dieses Objekt auf *public* (und nicht auf das defaultmäßige *private*) gesetzt ist, damit das Feld von außen auch sichtbar ist. Und drittens ist hier die JScript-spezifische Syntax zu sehen, mit der das Skript sich für ein Event, nämlich das *Leave*-Event der Text Box, eintragen kann. Es reicht nämlich aus, dem Event-Namen die Einleitung *add_* voranzustellen und in Klammen die Methode des Skriptobjekts anzugeben, die das Event behandeln soll. In diesem Fall ist dies die Methode *VerlassePreiseingabe*, deren Signatur (also Parameter und Rückgabewert) natürlich mit der Definition des Events übereinstimmen muss.

Schmuggelware

Das *Leave*-Event wird jedes Mal ausgelöst, wenn der Focus die Text Box verlässt. Der *sender*-Parameter von *VerlassePreiseingabe* wird einfach als Referenz auf eine Text Box aufgefasst und der Inhalt der Eingabe wird aus der *Text*-Eigenschaft ausgelesen. Wenn der Benutzer den Zusatz *DM* oder *DEM* eingetippt hat, wird eine Umrechnung in Euro vorgenommen und der Euro-Wert wird wieder in die *Text*-Eigenschaft geschrieben. Auf diese Weise schmuggelt das Skript eine neue Funktion in die Anwendung. Und im Gegensatz zu dem vorherigen Beispiel (Aufruf von *PreisBerechnen*) musste dies nicht bereits bei der Programmierung der Anwendung vorbereitet werden.

Noch augenfälliger ist dies bei dem nächsten Eingriff, den der Konstruktor der *ScriptKlasse* vornimmt. Beim Starten des Skripts wird nämlich auch ein neuer Button erzeugt und der *Controls*-Auflistung des Formulars hinzugefügt. Die Positionierung und Größe des Buttons orientiert sich dabei an Größe und Position des schon vorhandenen Buttons *btnBerechnen*, um das Layout des Formulars nicht zu stören. Die ungewöhnliche Farbe habe ich diesem Button nur deshalb zugewiesen, damit er in den Abbildungen gleich auffällt. Das *Click*-Event des neuen Buttons wird ebenfalls von einer Skript-Methode behandelt. Übrigens können statische Methoden nicht als Event Handler dienen.

Geister, die ich rief

Nachdem nun alles so schön funktioniert, sollten wir auch darüber nachdenken, wie man ein Skript wieder anhält und eventuell gegen eine neue Version austauscht, weil man zum Beispiel den Quelltext geändert hat und die neue Version ausprobieren möchte. Leider ist das nicht ganz so einfach, wie man vermuten sollte. Die *IVsaEngine*-Schnittstelle bietet zwar eine Methode namens *Reset()* an, nach deren Aufruf (durch Zuweisung von *false* an die *IsRunning*-Eigenschaft des *ScriptHost*-Objekts) das Code Item ersetzt und erneut kompiliert und gestartet werden kann, aber die Sache hat leider einen oder sogar zwei Haken.

Erstens führt das fortgesetzte Kompilieren, Starten, Stoppen, erneut Kompilieren usw. jedes Mal zum Erzeugen einer neuen Assembly, die auch dann im Speicher geladen bleibt, wenn sie nicht mehr gebraucht wird. Dies ist nicht anders, als wenn Sie in Ihrem Anwendungsprogramm selbst *Assembly.Load()* aufrufen, denn auch da haben Sie es schwer, dieses Code-Objekt wieder loszuwerden. Es gibt nämlich keine Unload-Technik für Assemblies. Gleiches gilt übrigens für Bibliotheken bei Win32 (sprich DLLs). Die einzige Möglichkeit, eine Assembly zuverlässig wieder loszuwerden, ist das Entladen der AppDomain, in deren Kontext sie geladen wurde [3]. In unserem Beispiel gibt es für die ganze Anwendung aber nur eine AppDomain und es wäre auch ziemlich kompliziert, es anders zu machen. Daher müssen wir leider in Kauf nehmen, dass mit jeder neuen Skript-Assembly der Speicher zusätzlich belastet wird, was natürlich nicht endlos möglich ist.

Der zweite Haken ist noch tückischer. Die *Reset()*-Methode der Engine löst nicht die manuell (z.B. mit *add_Leave*) hinzugefügten Event Handler und sie gibt natürlich auch nicht den vom Skript erzeugten But-

ton wieder frei. Tatsächlich werden die Event Handler weiterhin aufgerufen, auch dann, wenn bereits ein neues Skript geladen ist und läuft. Um mein Beispielprogramm möglichst einfach zu halten, habe ich hier ganz auf die Möglichkeit, mehrere Versionen des Skripts während einer Programmsitzung zu starten, verzichtet. Wenn Sie diese Option aber doch schaffen wollen, dann können Sie vor dem *IVsaEngine.Reset()*-Aufruf eine Aufräummethode in Ihrem Skript aktivieren (oder vielleicht ein *OnBeforeReset*-Event anbieten), um dem Skript vor dem Ableben die Gelegenheit zu geben, seinen Nachlass zu ordnen. Um Event Handler zu lösen, verwenden Sie in JScript einfach das Präfix *remove_* (statt *add_*) vor dem Eventnamen.

Entwanzen

Nach den schlechten Nachrichten über die Schwierigkeiten mit dem Entladen eines Skripts, nun noch eine gute: Skriptprogramme lassen sich hervorragen debuggen. Alles was Sie tun müssen, ist beim Erzeugen der Engine mit *IVsaEngine.SetOption()* die Option *DebugDirectory* zu setzen und einen Pfad anzugeben, in dem die Skript-Quellcodes für das Debugging zwischengespeichert werden können. Im Konstruktor des *ScriptHost*-Objekts habe ich eine entsprechende Anweisung schon als Kommentar vorgesehen. Natürlich könnte auch das Anwendungsprogramm zur Laufzeit aufgrund einer Einstellung entscheiden, ob das Verzeichnis gesetzt werden soll. Wird hier ein gültiges Verzeichnis eingestellt, so erzeugt die Engine dort beim Kompilieren eine Quellcodedatei.

Sie können sich mit einem geeigneten Debugger (Visual Studio .NET oder DbgCLR.exe aus dem Framework SDK) mit dem Prozess verbinden bzw. die Anwendung unter der Kontrolle des Debuggers starten. Beim Kompilieren des Skripts wird dann in dem festgelegten Verzeichnis eine Datei mit der Endung *.js* (für JScript) erzeugt. Diese können Sie mit dem Debugger öffnen und beliebige Haltepunkte darin setzen, was sogar auch dann funktioniert, wenn das Anwendungsprogramm ohne Debug-Informationen umgewandelt worden ist. Abbildung 4 zeigt DbgCLR im Einsatz.

Es steht Ihnen übrigens frei, nach jedem Programmstart den Quellcode des Skripts zu laden, kompilieren und auszuführen. Alternativ können Sie den kompilierten Zustand des Skripts speichern und später wieder erneut laden. Um Ihnen diese Möglichkeit zu illustrieren, bietet das Scripting-Fenster meines Beispiel-programms eine Schaltfläche an, deren *Click*-Handler die *ScriptHost*-Methode *KompiliertesScriptSpeichern()* aufruft. Darin werden mit *IVsaEngine.SaveCompiledState()* der Binärcode und die Debug-Daten des bereits kompilierten (aber noch nicht gestarteten) Skripts in Byte-Arrays übertragen und dann in eine Datei mit der willkürlich gewählten Endung *.bin* gespeichert. Die Methode *StartAusBinFileVersuchen()*, die nach dem Initialisieren des ScriptHost aufgerufen werden kann, überprüft, ob die *.bin*-Datei vorhanden ist und versucht gegebenenfalls (ohne vorheriges Laden eines Code Items) die Engine mit *IVsaEngine.Run()* zu startet. Ein Starten ohne geladenes Programm nimmt die Engine zum Anlass, dem Host durch Aufruf von *IVsaSite.GetCompilesState()* um die Binärdaten zu bitten, die eben dort wieder aus der Datei gelesen und an die Engine übergeben werden. Es ist also möglich, ein kompiliertes Skript laufen zu lassen, ohne dass der Quellcode vorhanden ist. Natürlich bleibt es letztlich Ihnen überlassen, ob Sie Quellcode oder Binärdaten speichern und ob Sie dies in einer Datei oder vielleicht in einer Datenbank tun.

Fazit

Die Einbettung von VSA in die CLR und das .NET Framework ermöglichen ein barrierefreies Zusammenwirken von Hostprogramm und Skript, woraus sich sehr anspruchsvolle Möglichkeiten für ein Customizing von Anwendungssoftware ergeben.

Beim Schreiben dieses Artikels hatte ich das Szenario vor Augen, bei dem der Entwickler selbst auch die Programmierung des Skripts übernimmt. Überlegungen in Richtung Sicherheit, etwa zur Vermeidung von böswilligen Eingriffen in den Programmablauf, spielten hier praktisch keine Rolle.

Links & Literatur

[1] vsa.summitsoft.com/

[2] .NET Framework SDK Dokumentation, „JScript .NET“

[3] Jochen Reinelt, „Hin und weg“, *dot.net magazin* 4.2003