Neue Programmiersprachen für .NET

Neue Blickwinkel

Schauen Sie ab und zu auf andere Programmiersprachen und wünschen Sie sich, dass ausgewählte Merkmale dieser Sprachen auch Bestandteil von C# oder auch Visual Basic wären? Nichts einfacher als das! Machen Sie direkten Gebrauch von diesen Sprachen und nutzen Sie deren Bibliotheken einfach im Verbund mit C# oder VB.

Auf einen Blick

Dipl.-Ing. **Andreas Maslo** leitet das Ingenieurbüro IngES, das sich mit dem Erstellen von EDV-Publikationen und mit Softwareentwicklung befasst. Er arbeitet außerdem als freier Journalist, EDV-Berater und Fachbuchautor. Sie erreichen ihn unter am@ing-es.de.

Inhalt

- Kurzvorstellung der .NET-Sprachen Phalanger, Nemerle, Vulcan.NET, P# und Prolog.NET
- Ein kurzes Beispiel zu jeder Sprache.

dnpCode

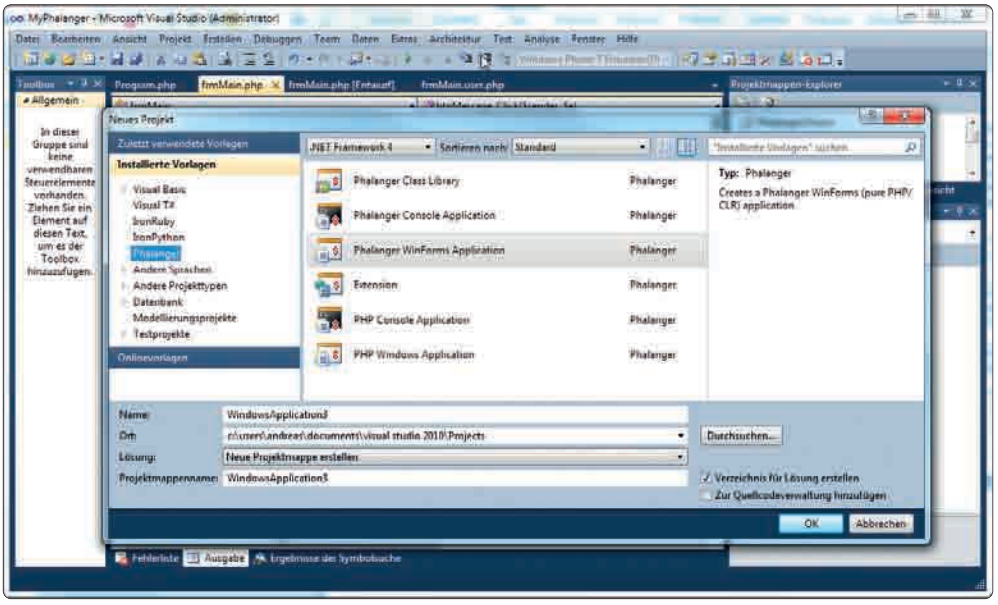
A1112AlternativeSprachen

Mit Visual Studio 2010 erhalten Sie die .NET-Sprachen Visual Basic, C# und F#, die sich beliebig miteinander kombinieren lassen. Neben den von Microsoft angebotenen Sprachen gibt es eine Vielzahl weiterer Sprachen für .NET, die – sofern es sich nicht um ausgewiesene Interpreter handelt – ebenfalls zur Anlage von Assemblies nutzbar sind. Die Sprachen Phalanger, Nemerle, Vulcan.NET, Prolog.NET und P# können Sie hier näher kennenlernen.

Phalanger 2.1 for .NET 4.0

Die Programmiersprache PHP (PHP Hypertext Preprocessor) ist eine der wichtigsten Programmiersprachen für die Entwicklung von Webanwendungen. Ihre Syntax orientiert sich an den Sprachen C und Perl. Sie wird in der Regel serverseitig unter Linux-Betriebssystemen verwendet und in HTML als Skriptsprache eingebettet. Mit PHP werden Webapplikationen und dynamische Webseiten erzeugt. Die Clientanfrage (Request) wird dabei serverseitig durch einen PHP-Interpreter verarbeitet und beantwortet (Response). Die Ergebnisse der Anfrage werden im Browser des Clients dargestellt. Aktuelle Versionen von PHP sind objekt- sowie speicheroptimiert und erlauben auch eine Datenbankanbindung. Auch wenn PHP vorrangig im Web zum Einsatz kommt, lassen sich damit auch Skripte für die Eingabeaufforderung entwickeln.

Mit Phalanger ist eine PHP-Variante für .NET verfügbar [1]. Damit können Sie herkömmlichen PHP-Quelltext unter .NET 4.0 verwenden. Phalanger liegt als Compiler vor, der Intermediate-Language-Code (IL-Code) und Assemblies generiert, wie das auch C# und Visual Basic tun. Phalanger kann auf die gesamte Klassenbibliothek von .NET zugreifen und die Funktionalität von PHP mit .NET verknüpfen. .NET-Bibliotheken sind unter Phalanger ebenso nutzbar, wie Sie mit Phalanger angelegte Bibliotheken auch innerhalb anderer .NET-Sprachen verwenden können. Unter Phalanger importieren und verwenden Sie beliebige .NET-Namespaces, .NET-Generics (Erweiterung und Definition), .NET-Objekte, partielle Klassen, .NET-Eigenschaften, das .NET-Typensystem oder auch benutzerdefinierte Attribute. Phalanger kann sowohl unter .NET als auch unter Mono eingesetzt werden. Sie übernehmen vorhandenen PHP-Quelltext und erweitern diesen nach Bedarf um weitere .NET-Funktionsmerkmale. Bei Bedarf machen Sie Phalanger als Skriptsprache in eigenen .NET-Anwendungen verfügbar.



[Abb. 1] Phalanger integriert sich in Visual Studio 2010.

Obwohl Phalanger als Open Source angeboten wird, wird auch kommerzielle Unterstützung von den Entwicklern bereitgestellt. Phalanger integriert sich im Rahmen der Installation in Visual Studio 2010 und macht eigene Projektvorlagen für Klassenbibliotheken, Konsolenanwendungen, Windows-Forms-Applikationen und Erweiterungen (Extensions) verfügbar. Zusätzlich werden auch Vorlagen für PHP-Konsolen- und Windows-Anwendungen angeboten. In Visual Studio wird Syntax-Highlighting bereitgestellt, Autovervollständigen wird jedoch nicht angeboten. Phalanger-Anwendungen lassen sich mit dem Visual-Studio-Debugger testen.

Die Nutzung von .NET-Funktionalitäten in der sprachspezifischen Syntax soll hier an einer Windows-Forms-Anwendung gezeigt werden. Diese definiert ein Formular *frmMain*, das eine Schaltfläche *btnMessage* enthält. Wählen Sie diese an, wird die Meldung *Hallo Welt!* ausgegeben. Die Anwendung wird über die Routine *Main* ausgeführt, die für das Laden des Hauptformulars verantwortlich ist. Alle Programmbestandteile sind dem Namespace *PhalangerDemo* zugeordnet. Im Programm werden die Namespaces *System* und *System.Windows.Forms* importiert:

```
<?php
import namespace System;
import namespace
System:::Windows:::Forms;

import namespace PhalangerDemo;

class Program
{
 static function Main()
 {
  Application::EnableVisualStyles();
  Application::Run(new frmMain());
 }
}
?>
```

In Phalanger wird bei der Namespace-Angabe anstelle der Punkt- mit der dreifachen Doppelpunkt-Notation gearbeitet. Jede Anweisung endet wie bei C# mit einem Semikolon, Blockstrukturen werden durch geschweifte Klammern zusammengefasst. Objektbezogene Methoden und/oder Eigenschaften werden über die doppelte Doppelpunktnotation miteinander verkettet.

Listing 1 zeigt den Konstruktor *__construct* des Formulars *frmMain* und wie bezogen auf das aktuelle Formular (*$this*) über die Methode *InitializeComponent* das Formular und die darin enthaltene Schaltfläche initialisiert und via *Add* der Steuerelementauflistung *Controls* des Formulars angefügt wird. Mit der Ereignisprozedur *btnMessage_Click*, die per Eventhandler mit dem *Click*-Ereignis der Schaltfläche verbunden wird, wird der Meldungstext *Hallo Welt!* durch Aufruf der Methode *MessageBox* ausgegeben. Phalanger-Projektdateien erhalten das Standarddateikürzel *sln*, Quelldateien das Dateikürzel *php* zugewiesen.

Listing 1

Windows-Forms-Anwendung in Phalanger.

```
<?
 import namespace PhalangerDemo;

 namespace PhalangerDemo {

 [Export]
 partial class frmMain extends System:::Windows:::Forms:::Form {
  private $btnMessage;
  private $components = NULL;

  public function __construct()
   : parent() {
   $this->InitializeComponent();
  }

  public function InitializeComponent() {
   $this->btnMessage = new System:::Windows:::Forms:::Button();
   $this->SuspendLayout();
   // btnMessage
   $this->btnMessage->Location = new System:::Drawing:::Point(43, 28);
   $this->btnMessage->Name = "btnMessage";
   $this->btnMessage->Size = new System:::Drawing:::Size(75, 23);
   $this->btnMessage->TabIndex = 0;
   $this->btnMessage->Text = "Klick mich!";
   $this->btnMessage->UseVisualStyleBackColor = true;
   $this->btnMessage->Click->Add(new System:::EventHandler(
     array($this, "btnMessage_Click")));
   // frmMain
   $this->AutoScaleDimensions = new System:::Drawing:::SizeF(6, 13);
   $this->AutoScaleMode = System:::Windows:::Forms:::AutoScaleMode::Font;
   $this->ClientSize = new System:::Drawing:::Size(292, 266);
   $this->Controls->Add($this->btnMessage);
   $this->Name = "frmMain";
   $this->Text = "Phalanger-Demo";
   $this->ResumeLayout(false);
  }

  private function btnMessage_Click(System:::Object $sender, System:::EventArgs $e) {
   System:::Windows:::Forms:::MessageBox::Show("Hallo Welt!");
  }
 }
}
?>
```

Vulcan.NET

Vulcan.NET ist eine kommerzielle Programmiersprache, die auf der objektorientierten xBase-Datenbanksprache Visual Objects der Version 2.8 basiert [2]. Sie legt besonderen Wert auf Rückwärtskompatibilität und kann dementsprechend auch die bereits vorhandenen Quelltexte von Visual Objects übernehmen. Vulcan.NET macht somit Visual Objects für .NET verfügbar und bietet gleichermaßen alle Klassen und Objekte sowie Methoden, Eigenschaften und Ereignisse von .NET an. Die Datenbanksprache richtet sich nach der Common Language Infrastruktur (CLI) und ist objektorientiert, bietet das Überladen von Methoden und Operatoren, streng typisierte Datenfelder, Referenz-, Enumerations- und Wertetypen aber auch Low-Level-Zeigeroperationen an. Die Sprache integriert sich im Rahmen der Installation wahlweise in Visual Studio 2005, 2008 oder 2010 und bindet optional auch eine umfassende Hilfe in die IDE ein. Projektvorlagen stehen sowohl für Visual-Objects-kompati-

ble MDI/SDI-Oberflächen als auch für die folgenden .NET-Projekttypen bereit:

- Windows-Forms-Anwendung,
- Windows-Forms-MDI-Anwendung,
- Klassenbibliothek,
- Konsolenanwendung,
- WPF-Anwendung,
- WPF-Steuerelementbibliothek,
- leeres Projekt.

Für Windows-Forms- und WPF-Anwendungen werden in Visual Studio die entsprechenden Designer angebunden. Im Quelltexteditor wird mit Syntaxfarbgebung und Autovervollständigen gearbeitet (Abbildung 2). Alternativ zu Visual Studio wird mit VulcanIDE auch eine spracheigene Entwicklungsumgebung angeboten.

Mit dem Quelltexteditor lassen sich Programme schritt- oder prozedurweise ausführen sowie Fehler überwachen. Die für Phalanger codierte Windows-Forms-Beispielanwendung lässt sich auch in Vulcan.NET umsetzen. Die Programmausführung startet in der Funktion *Start*, die das Formular *frmMain* lädt und ausführt und nach der Beendigung einen Exitcode zurückgibt. Die für die Ausführung benötigten Namespaces werden per *#using*-Anweisungen angebunden:

```
#using System
#using System.Windows.Forms

[STAThread] ;
FUNCTION Start() AS INT
 LOCAL exitCode AS INT
 Application.EnableVisualStyles()
 Application.
   SetCompatibleTextRenderingDefault(
    false)
 Application.Run( frmMain{} )
 RETURN exitCode
```

Listing 2

Windows-Forms-Anwendung in Vulcan.NET.

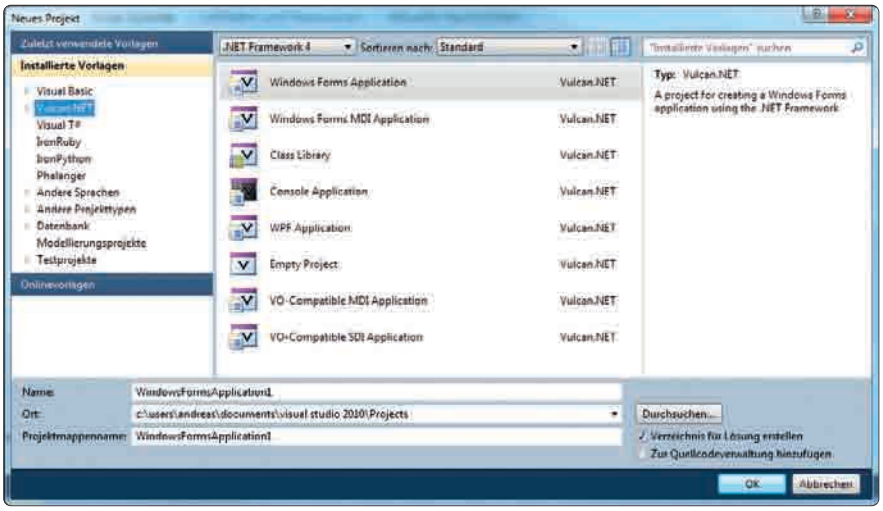
```
#using System
#using System.Windows.Forms

CLASS MyForm INHERIT System.Windows.Forms.Form
 PRIVATE btnMessage AS System.Windows.Forms.Button
 PRIVATE components AS System.ComponentModel.IContainer
 CONSTRUCTOR()
  SUPER()
  SELF:InitializeComponent()
  RETURN

  PROTECTED METHOD Dispose( disposing AS LOGIC ) AS VOID
   IF disposing && components != NULL
    components:Dispose()
   ENDIF
   SUPER:Dispose( disposing )
   RETURN

  PRIVATE METHOD InitializeComponent() AS System.Void
   SELF:btnMessage := System.Windows.Forms.Button{}
   SELF:SuspendLayout()
   // btnMessage
   SELF:btnMessage:Location := System.Drawing.Point{25, 24}
   SELF:btnMessage:Name := "btnMessage"
   SELF:btnMessage:Size := System.Drawing.Size{145, 23}
   SELF:btnMessage:TabIndex := 0
   SELF:btnMessage:Text := "Klick mich!"
   SELF:btnMessage:UseVisualStyleBackColor := TRUE
   SELF:btnMessage:Click += System.EventHandler{ SELF, @btnMessage_Click() }
   // MyForm
   SELF:AutoScaleDimensions := System.Drawing.SizeF{((Single) 6), ((Single) 13)}
   SELF:AutoScaleMode := System.Windows.Forms.AutoScaleMode.Font
   SELF:ClientSize := System.Drawing.Size{284, 71}
   SELF:Controls:Add(SELF:btnMessage)
   SELF:MaximizeBox := FALSE
   SELF:MinimizeBox := FALSE
   SELF:Name := "MyForm"
   SELF:StartPosition := System.Windows.Forms.FormStartPosition.CenterScreen
   SELF:Text := "Vulcan Demo"
   SELF:ResumeLayout(FALSE)

   PRIVATE METHOD btnMessage_Click( sender AS System.Object, e AS
     System.EventArgs ) AS System.Void
     MessageBox.Show("Hallo Welt!")
     RETURN
END CLASS
```

Listing 2 zeigt das Beispielprogramm für Vulcan.NET. Auch hier werden zunächst die benötigten Namensräume eingebunden. Die Formularklasse trägt den Namen *MyForm*, die im Konstruktor die Methode *InitializeComponent* aufruft. In dieser Methode werden erneut das Formular (diesmal unter dem Bezeichner *SELF* angesprochen), sowie die Schaltfläche *btnMessage* initialisiert. Die Schaltfläche wird auch hier per Event-Handler mit dem *Click*-Ereignis von *btnMessage* und der Ereignisprozedur *btnMessage_Click* verbunden. Die Ereignisprozedur gibt via *MessageBox* wieder eine einfache Meldung aus.

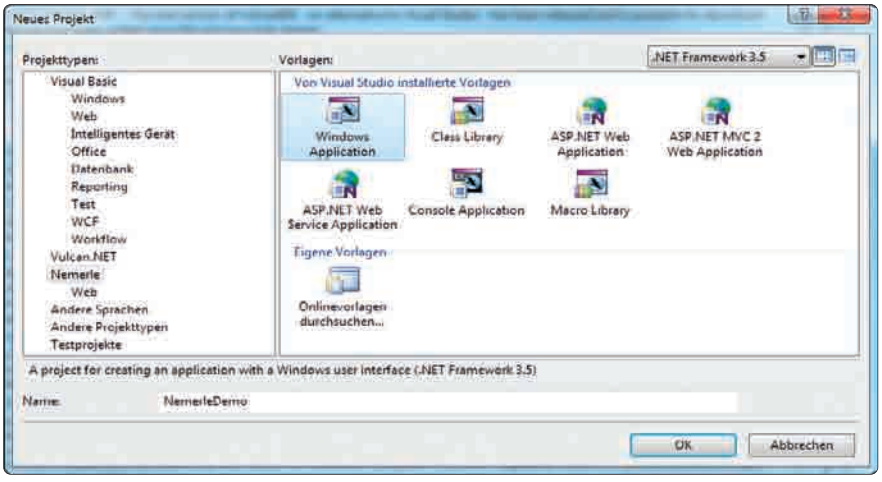
Vulcan.NET-Projektdateien erhalten das Standarddateikürzel *sln* und Quelldateien das Dateikürzel *prg* zugewiesen. Schlüsselwörter werden in Quelltexten üblicherweise in Großschrift eingegeben und verwaltet.

Nemerle 1.0

Nemerle ist eine hybride Computersprache, die in der ersten Version seit Mai 2011 verfügbar ist und mit der kombiniert imperativ, objektorientiert und funktional programmiert werden kann. Die Sprache bietet somit gegenüber C# und Visual Basic erweiterte Merkmale, wobei ein besonderer Schwerpunkt auf dem Bereich Metaprogrammierung liegt. Darüber werden Anwendungen erzeugt, die ihrerseits Programme erzeugen oder vorhandene manipulieren können. Um die Funktionalität verfügbar zu machen, stellt Nemerle Makros zur Verfügung – so wie die Programmiersprache Lisp. Mit Nemerle erhalten Sie bereits eine Makrobibliothek, über die spezielle Funktionen verfügbar gemacht werden, wie beispielsweise eine LINQ-Unterstützung, erweiterte Operatoren oder die generelle Unterstützung für AOP (Aspect Oriented Programmierung). Das Programmierpradigma AOP erlaubt es, generische Funktionalitäten über mehrere



[Abb. 2] Vulcan.NET in Visual Studio 2010.



[Abb. 3] Nemerle-Projektvorlagen unter Visual Studio 2008.

Klassen hinweg einzusetzen. Auch Language Oriented Programming (LOP) wird von Nemerle unterstützt. Mithilfe von LOP lassen sich eigene domänenspezifische Sprachen (DSL – Domain Specific Languages) definieren und einsetzen.

Über *Nemerle.Peg* erhalten Sie eine Makrobibliothek für einen Parser-Generator, der auf der PEG-Notation basiert (Parsing Expression Grammar). XML-Literale lassen sich in Programmen über die Makrobibliothek *Nemerle.xml* verarbeiten. Nemerle ist schnell, leistungsfähig und komplex. Sie erhalten die installierbare Programmfassung unter [3]. Eine umfassende Sprachreferenz ist unter [4] erhältlich.

In der aktuellen Version ist Nemerle ausschließlich in Visual Studio 2008 integrierbar. Hier gab es im Rahmen des Tests allerdings Probleme beim Speichern von Projekten. An der Integrationskomponente für Visual Studio 2010 wird noch gearbeitet. Wann diese verfügbar sein wird, ist derzeit noch nicht abzusehen.

Unter Visual Studio 2008 wird eine Syntaxfarbgebung und eingeschränktes Autovervollständigen geboten. Projektvorlagen stehen für Windows-Forms-Anwendungen, Klassenbibliotheken, ASP.NET-Webanwendungen, ASP.NET-Webdienste, Konsolenanwendungen und Makrobibliotheken zur Verfügung (Abbildung 3).

Setzen Sie das obige Beispielprogramm mit Nemerle um, ergeben sich nur wenige Unterschiede gegenüber den Phalanger- und Vulcan.NET-Varianten. Auch hier wird die Programmausführung per Startprozedur vorgenommen, die den Namen *Main* trägt. Verwendete .NET-Namensräume werden über eine *using*-Anweisung angebunden. Auch unter Nemerle wird in der Beispielanwendung die Startprozedur zum Laden und Anzeigen des Hauptformulars verwendet.

Die *#pragma*-Anweisung ist eine Compiler-Direktive, die Warnmeldungen unterdrückt. Ansonsten wird auch hier die Ähnlichkeit in der Syntax zu C# deutlich.

```
#pragma warning disable 10005

using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace NemerleDemo
{
 static class Program
 {
  [STAThread]
  static Main() : void
  {
   Application.EnableVisualStyles();
   Application.
    SetCompatibleTextRenderingDefault(
      false);
   Application.Run(MainForm());
  }
 }
}
```

Listing 3 zeigt den Quelltext zum Beispielformular *MainForm*. Auch hier wird über einen Konstruktor (*this*) der partiellen Klasse die Initialisierung des Formulars über die Methode *InitializeComponent* angestoßen. Die Ereignisprozedur *btnMessage_Click*, die bei der Schaltflächenanwahl des Formulars ausgeführt wird, erhält zwei Parameter übergeben. Diese werden hier mit einem Tiefstrich versehen, damit die Nichtverarbeitung dieser Parameter nicht zur Warnmeldung führt. Die Meldungsausgabe wird auch hier mit der Methode *Show*, der Anweisung *MessageBox*

Listing 3

Windows-Forms-Anwendung in Nemerle.

```
#pragma warning disable 10005

using Nemerle.Collections;
using Nemerle.Text;
using Nemerle.Utility;
...
namespace NemerleDemo
{
 public partial class MainForm : Form
 {
  public this()
  {
   InitializeComponent();
  }

  private btnMessage_Click (
    _sender : object,  _e :
    System.EventArgs) : void
  {
   MessageBox.Show("Hallo Welt!");
  }
 }
}
```

durchgeführt. Von Nemerle werden Sie darauf hingewiesen, dass der Rückgabewert dieser Funktion nicht verwertet wird. Um auch diese Warnmeldung zu unterbinden, wird dem Quelltext die im Listing angeführte *#pragma*-Anweisung vorangestellt.

Der vom Designer angelegte Quelltext wird partiell über eine gesonderte Klasse verwaltet (Listing 4). Hier finden Sie erneut die Methode *InitializeComponent*, über die die Schaltfläche und das Formular definiert werden. Dabei wird erneut die Schaltfläche der *Controls*-Auflistung des Formulars angefügt und die Ereignisprozedur zur Schaltflächenanwahl mit dem entsprechenden Schaltflächenereignis verknüpft. Im Quellverzeichnis zu den Beispielprogrammen finden Sie die hier angeführten Quelltextpassagen aufgrund der fehlerhaften Funktion zum Speichern des Projektes lediglich im Textformat.

Listing 4

Vom Designer angelegter Quelltext (Nemerle).

```
#pragma warning disable 10001

namespace NemerleDemo
{
  public partial class MainForm
  {
   private  mutable  components : System.ComponentModel.IContainer = null;

   protected override Dispose(disposing : bool) :  void
   {
    when (disposing && components != null)
    components.Dispose();
    base.Dispose(disposing);
   }

   private InitializeComponent() :  void
   {
    this.btnMessage = System.Windows.Forms.Button();
    this.SuspendLayout();
    // btnMessage
    this.btnMessage.Location = System.Drawing.Point(53,  22);
    this.btnMessage.Name = "btnMessage";
    this.btnMessage.Size = System.Drawing.Size(182,  23);
    this.btnMessage.TabIndex = 0;
    this.btnMessage.Text = "Klick mich!";
    this.btnMessage.UseVisualStyleBackColor = true;
    this.btnMessage.Click += System.EventHandler(this.btnMessage_Click);
    // frmMain
    this.AutoScaleDimensions = System.Drawing.SizeF(6f,  13f);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = System.Drawing.Size(284,  68);
    this.Controls.Add(this.btnMessage);
    this.MaximizeBox = false;
    this.MinimizeBox = false;
    this.Name = "frmMain";
    this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;
    this.Text = "Nemerle Demo";
    this.ResumeLayout(false);
   }

   private mutable btnMessage : System.Windows.Forms.Button;
  }
}
```

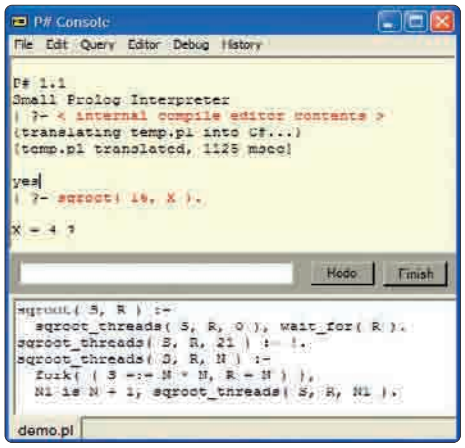
P#

Neben den imperativen, funktionalen und objektorientierten Sprachen gibt es logische beziehungsweise prädikative Programmiersprachen. Logische Programme setzen sich nicht aus einer Folge von Anweisungen zusammen, sondern aus einer Menge von Axiomen. Dabei handelt es sich um eine Datenbasis, die aus Fakten und Regeln besteht, an die bestimmte Anfragen gesendet werden. Der bekannteste Vertreter dieser Sprachen ist Prolog. Prolog wurde bereits in den 70er Jahren entwickelt. Seit 1995 ist die Sprache per ISO-Norm spezifiziert. Sie wird bevorzugt in den Bereichen Expertensysteme, Computerlinguistik oder auch Künstliche Intelligenz (KI) eingesetzt. P# ist eine Prolog-Implementierung für .NET. Allerdings handelt es sich bei P# nicht um einen eigenständigen Compiler, sondern einen Cross-Compiler, der aus Prolog-Quelltext C#-Quelltext generiert, der dann unter C# zusammen mit einer Loader-Klasse kompiliert und in Assemblies eingebunden werden kann. Die Loader-Klasse trägt den Namen *Loader.cs* und liegt P# im Quelltext bei. Die Verarbeitung der Prolog-Bestandteile erfolgt über eine spezielle Laufzeitbibliothek mit dem Namen *PSharp.dll*, die neben dem P#-Laufzeitsystem auch die P#-Funktionsbibliothek beinhaltet und die mit dem generierten und übersetzten C#-Quelltext per Reflection interagiert, wobei die Interaktion durch die Loader-Klasse angestoßen wird. P#-Initialisierungscode wird über diese DLL zuerst geladen. Dieser ermittelt die in die C#-Assemblies eingebundenen Übersetzungen der Prolog-Prädikate. P#-Anwendungen sind eine Mischung aus P# und C# und entsprechend sind spezielle Methoden für den wechselweisen Zugriff und die Interaktion der beiden Sprachen dokumentiert. Sie erhalten P# zusammen mit einer kompakten Benutzerumgebung und einer ausführlichen Dokumentation unter [5].

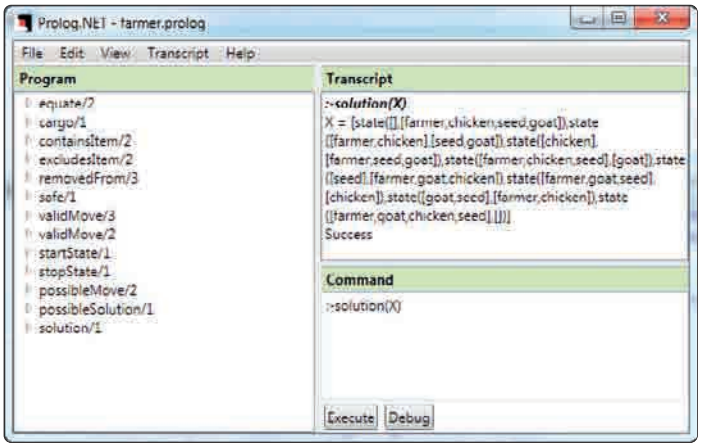
Eine Integration in Visual Studio wird aufgrund der beschriebenen Architektur nicht verfügbar gemacht. Sie erhalten P# als Ziparchiv, das die Konsolenanwendung sowie die Laufzeitbibliothek beinhaltet. Letztere müssen Sie mit allen Anwendungen verteilen, die die Funktionalität von P# nutzen sollen. Anwendungen lassen sich dementsprechend nicht alleine mit P# generieren. Eine C#-Anwendung wird in jedem Fall als Frontend benötigt. Aus diesem Grunde soll hier keine Beispielanwendung, sondern lediglich das grundlegende Definieren und Abfragen des Datenbestandes unter Prolog gezeigt werden. Um Vornamen als Prädikate zu definieren, können Sie die folgenden Prädikate eingeben:

```
vorname(Peter)
vorname(Klaus)
vorname(Michael)
```

Im Anschluss daran können Sie zu dieser Datenbasis Abfragen stellen: Die folgenden



[Abb. 4] Die Eingabe-Konsole von P#.



[Abb. 5] Die Workbench zu Prolog.NET.

interaktiven Abfragen ermitteln beispielsweise, ob zwei der angegebenen Namen Bestandteil der Datenbasis sind, wobei das Ergebnis per Aussage (*yes/no*) zurückgeliefert wird. Als Abfrageoperator kommt hier der Operator *?-* zum Einsatz, der je nach Prolog-System auch variieren kann (vergleiche Prolog.NET unten):

```
?- vorname(Peter).
yes.
?- vorname (Tom).
no.
```

Eine variablenbezogene Anfrage liefert alle passenden Werte zurück:

```
?-Vorname(x).
x=Peter
x=Klaus
x=Michael
no.
```

Entsprechend lassen sich auch komplexere Datenbestände definieren und in Bezug zueinander setzen. Beispiele dafür sind die Beziehungen zwischen Personen (zum Beispiel Verwandtschaft, Freunde, Arbeitskollegen). Entsprechende Beispiele sind im Internet verfügbar und sollen hier nicht detailliert behandelt werden. Über die P#-Eingabekonsole können Sie selbst Prädikatdefinitionen vornehmen und verarbeiten (Abbildung 4).

Prolog.NET

Eine weitere Prolog-Variante liegt mit dem Open-Source-Projekt Prolog.NET vor [6]. Dieses macht einen eigenständigen Compiler, einen Laufzeit-Interpreter samt Benutzerumgebung sowie eine Laufzeitbibliothek für Prolog-Anwendungen verfügbar. Neben dem Prolog-Compiler (*prologc.exe*) bietet Prolog.NET Spracherweiterungen und einen Codegenerator an. Der Compiler kann ausführbare Programme (*exe*), Klassenbibliotheken (*dll*) oder abstrakten Maschinencode im XML-Format generieren. Das letztgenannte Format kann über den mitgelieferten Runtime-Interpreter verarbeitet werden und liegt im WAM-Format vor (Warren Abstract Machine). Der Compiler wird an der Eingabeaufforderung über Kommandozeilenschalter gesteuert. Der Schalter *target* gibt an, welches Ausgabeformat erzeugt werden soll. Der Prolog-Code wird durch den Compiler in wiederverwendbare .NET-Objekte kompiliert. Die Speicherverwaltung übernimmt später der Garbage Collector des .NET Frameworks. Prolog.NET ist für .NET und Mono verfügbar und einsetzbar unter Windows/Linux und Mac OS X. Während der Installation werden die Laufzeitbibliotheken im Global Assembly Cache (GAC) eingerichtet. Mit dem Dienstprogramm ILMerge von Microsoft Research kann die Laufzeitbibliothek für eine vereinfachte Verteilung später auch direkt in ausführbare Programme oder DLLs integriert werden. Prolog.NET bietet Schnittstellen, über die Sie auf .NET-Klassen oder -Objekte zugreifen, Methoden ausführen oder Eigenschaften setzen und abfragen. Über die Laufzeitbibliothek werden Prolog-Standardprädikate für Vergleiche, Ein- und Ausgaben verfügbar gemacht. Alternativ können auch Clientanwendungen direkt auf Prolog.NET zugreifen und Programme mit Prädikaten definieren, Abfragen absenden und Programme ausführen, wobei der Abfrageoperator *:-* zum Einsatz kommt (Listing 5).

Listing 5

Clientseitige Nutzung von Prolog.NET.

```
/* (Quelle: Prolog .NET Reference Guide) */

CodeSentence codeSentence;
codeSentence =
  Parser.Parse("hello(world)");
Program program = new Program();
program.Add(codeSentence);
codeSentence = Parser.Parse(":-hello(X)");
Query query = new Query(codeSentence);
PrologMachine machine =
  PrologMachine.Create(program, query);
ExecutionResults results = machine.Run();
```

Neben dem Compiler und dem Interpreter stehen für Prolog.NET eine Benutzeroberfläche zum Eingeben, Ausführen und Debuggen von Prolog.NET-Programmen (PrologWorkbench) (Abbildung 5) sowie eine Konsolenanwendung bereit, die für Testzwecke genutzt wird.

Damit sind die interessantesten, neuen oder in der Funktionalität von C# und Visual Basic abweichenden Sprachen behandelt. Einige andere, wie beispielsweise IronScheme, sind derzeit nur eingeschränkt verwendbar, da sie keine brauchbaren Benutzerumgebungen verfügbar machen und teilweise ausschließlich im Quelltextformat vorliegen. IronScheme ist ein Lisp-Dialekt, der das ältere IronLisp-Projekt abgelöst hat. Obwohl ursprünglich geplant, nutzt IronScheme aufgrund der Sprachanforderungen derzeit die Dynamic Language Runtime noch nicht. Die dotnetpro wird die Entwicklung anderer .NET-Sprachen weiter verfolgen und diese, sofern sinnvoll, zu einem späteren Zeitpunkt vorstellen. **[bl]**

[1] Phalanger, http://phalanger.codeplex.com
[2] Vulcan.NET, http://www.govulcan.net/portal
[3] Nemerle, http://nemerle.org
[4] Nemerle Sprachreferenz, http://nemerle.org/reference.html
[5] P#, http://homepages.inf.ed.ac.uk/stg/research/Psharp
[6] Prolog.NET, http://prolog.hodroj.net/