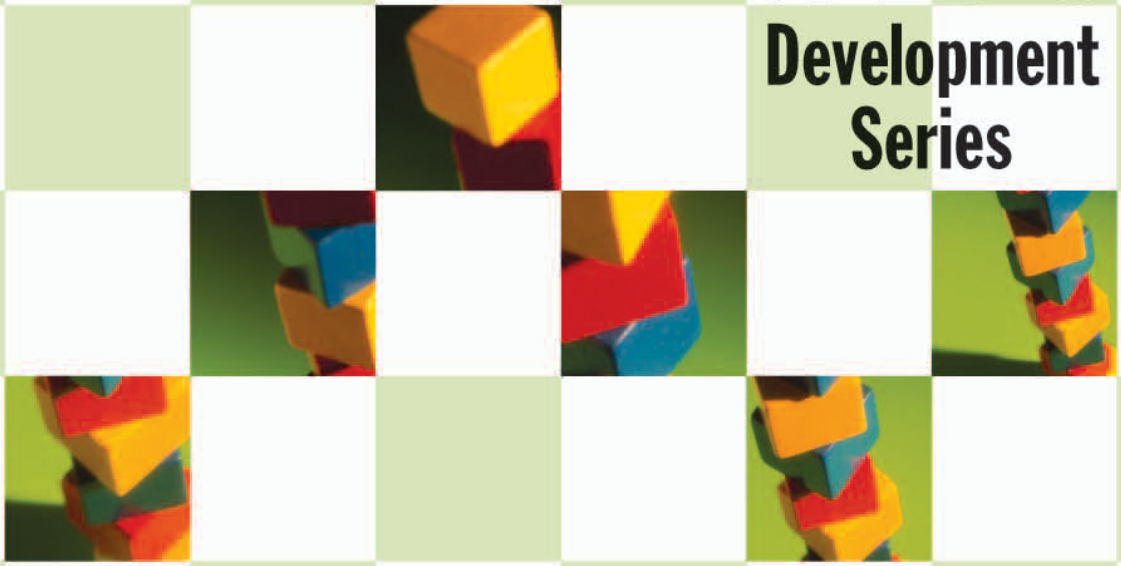
PEARSON Education

ADDISON-WESLEY

Das SQL Server 2005-Entwicklerbuch

Microsoft® .net™ Development Series



Bob Beauchemin
Dan Sullivan

SQLskills.com

ADDISON-WESLEY

3

Prozeduren und Funktionen in .NET-CLR-Sprachen

Neu in SQL Server 2005 ist die Unterstützung für die Common Language Runtime (CLR)-Sprachen. Damit hat man jetzt eine weitere Möglichkeit, um die Funktionalität von SQL Server zu erweitern. Bisher war hierfür jede Sprache geeignet, mit der sich eine DLL oder eine COM-Komponente erstellen lässt.

Erweiterten gespeicherten Prozeduren und COM-Komponenten fehlen jedoch die Sicherheit, die Zuverlässigkeit und in vielen Fällen die Leistungsfähigkeit, die Transact-SQL (T-SQL) zu bieten hat. In SQL Server 2005 kann man nun gespeicherte Prozeduren, benutzerdefinierte Funktionen und Trigger mit jeder .NET-CLR-Sprache erstellen und damit die Funktionalität von SQL Server hinsichtlich Sicherheit, Zuverlässigkeit und Leistung auf dem gleichen Niveau wie T-SQL erweitern.

3.1 SQL Server erweitern

Die Abfragefunktionalität von SQL Server kann mit T-SQL erweitert werden. Mit dieser Sprache lassen sich häufig vorkommende Aufgaben kapseln und man kann diese Aufgaben leichter verwalten und umsetzen. In T-SQL können Sie SQL-Stapel (Batches) schreiben, die allgemeine Aufgaben realisieren, und diese Stapel zur späteren Verwendung direkt in SQL Server speichern.

Ein großer Teil der Funktionalität von SQL Server stammt nicht von der Programmiersprache SQL, sondern von Erweiterungen, die Microsoft mithilfe seiner

proprietären Sprache T-SQL in SQL Server eingebunden hat. Mit T-SQL schreiben Sie imperativen Code.

Zum Beispiel gibt es eine von Microsoft geschriebene gespeicherte Prozedur namens `sp_adduser` für die Aufnahme eines neuen Benutzers in eine Datenbank. In SQL Server 2005 gilt diese gespeicherte Prozedur als veraltet, sie wird aber trotzdem noch unterstützt, damit vorhandene Skripts aus vorherigen Versionen von SQL Server weiterhin in SQL Server 2005 verwendet werden können. An die Stelle der gespeicherten Prozedur ist jetzt die Anweisung `CREATE USER` getreten. Beides funktioniert. Die gespeicherte Prozedur fügt die übergebenen Parameter – Anmeldename und Benutzername – in die betreffenden Datenbanktabellen ein. Wenn Sie einen Benutzer mit `sp_adduser` in die Datenbank einfügen, brauchen Sie die Einzelheiten nicht zu kennen, die innerhalb der Datenbank ablaufen. Praktisch könnte Microsoft die Art und Weise, wie SQL Server die Benutzer in einer Datenbank verwaltet, vollkommen ändern. Das hätte keinerlei Einfluss darauf, wie Sie einer Datenbank Benutzer hinzufügen.

Vor der Version 2005 konnte man SQL Server ausschließlich mit T-SQL erweitern oder eine erweiterte gespeicherte Prozedur bzw. eine COM-Komponente schreiben. Praktisch war es unumgänglich, dass Sie die Sprache T-SQL kennen. Viele Programmierer mussten also eine weitere Programmiersprache lernen, mit der sie aber viel weniger als in ihrer primären Sprache gearbeitet haben. Ein Visual Basic 2005-Programmierer versucht sich vielleicht mit »Dim id, ach nein, es heißt ja Declare id, ups, nein, es war Declare @id int.« an einer Variablendeklaration. Ähnliche Irrfahrten durch ein Meer möglicher Syntaxfehler erwarten Programmierer von anderen Sprachen, wenn sie versuchen, eine T-SQL-basierte gespeicherte Prozedur zu schreiben.

Erweiterte gespeicherte Prozeduren verlangen, dass eine ziemlich umständlich zu erzeugende DLL erstellt wird. C++-Programmierern steht allerdings in Visual Studio ein Assistent zur Seite, um diese DLL zu erzeugen; sie müssen dann nur noch die gewünschte Funktionalität ausfüllen. In ähnlicher Weise können Visual Basic 6-Programmierer eine COM-Komponente erzeugen und sie in SQL Server über die gespeicherte Prozedur `sp_OACreate` verwenden. Damit ist es C++ - und Visual Basic 6-Programmierern möglich, SQL Server zu erweitern und dabei in ihrer vertrauten Programmierumgebung zu bleiben. Erweiterte gespeicherte Prozeduren und COM-Komponenten besitzen Eigenschaften, die T-SQL nicht aufweist, weil sie auf Systemdienste außerhalb von SQL Server zugreifen können. Zum Beispiel handelt es sich bei der Erweiterung zu SQL Server, mit der das Verschicken von E-Mails möglich ist, um eine erweiterte gespeicherte Prozedur. In T-SQL hätte man das gar nicht schreiben können.

Erweiterte gespeicherte Prozeduren haben ihre eigenen Probleme. Obwohl man durchaus erweiterte gespeicherte Prozeduren schreiben kann, die sicher und zuverlässig sind, machen es uns die hierfür verwendeten Sprachen nicht gerade leicht. Im Allgemeinen muss eine erweiterte gespeicherte Prozedur oder eine COM-Komponente wesentlich tiefgründiger untersucht werden, als eine auf T-SQL basierende gespeicherte Prozedur, und in manchen Fällen kommt sie nicht an die Leistung von T-SQL heran.

SQL Server 2005 ändert dies nun alles. Man kann SQL Server in jeder CLR-Sprache erweitern. Die CLR ist Teil des .NET Framework. Erweiterungen, die in der CLR laufen, können genauso sicher und zuverlässig sein wie T-SQL und genauso flexibel wie eine erweiterte gespeicherte Prozedur bzw. eine COM-Komponente. Das heißt, dass Nicht-T-SQL-Entwickler eine vertraute Entwicklungsumgebung nutzen können, um die Funktionalität von SQL Server zu erweitern.

Außerdem gibt es einige Aufgaben, für die die CLR einfach besser geeignet ist. Das gilt normalerweise für Operationen, die numerische Berechnungen oder Zeichenfolgenmanipulationen umfassen.

Falls Sie einmal die Aussage hören: »Da SQL Server 2005 jetzt die CLR verwendet, ist jeder Visual Basic- und C#-Programmierer auch ein Datenbankprogrammierer!«, sollten Sie das ganz schnell wieder vergessen. Für Mengenoperationen ist die CLR keinesfalls besser geeignet – hier ist SQL der klare Gewinner. SQL-Ausdrücke kann die CLR allerdings genau wie T-SQL ausführen und zudem mit etwa der gleichen Effizienz. Auch wenn man Code in einer CLR-Sprache schreiben kann, muss man trotzdem in der Lage sein, eine Anweisung wie `SELECT DISTINCT A.au_fname, A.au_lname FROM authors A JOIN titleauthors T ON A.au_id = T.au_id` zu formulieren, wenn man alle Autoren finden möchte, die etwas veröffentlicht haben.

Kapitel 2 hat sich bereits damit beschäftigt, wie SQL Server 2005 die CLR hostet und Assemblys lädt. Dieses Kapitel konzentriert sich nun darauf, wie man die Methoden in diesen Assemblys als gespeicherte Prozeduren, Funktionen und Trigger für Operationen, die nicht auf SQL Server zugreifen, verwendet. In Kapitel 4 erfahren Sie dann, wie diese Methoden direkt auf SQL Server zugreifen können, was sich positiv auf die Leistung auswirkt.

3.2 Grundlagen der CLR-Erweiterung

Eine öffentliche statische Methode einer öffentlichen Klasse von einer CLR-basierten Sprache kann – mit einigen Einschränkungen – als gespeicherte Prozedur, benutzerdefinierte Funktion oder Trigger in SQL Server verwendet werden. Wir gehen später auf die konkreten Einschränkungen ein. Allgemein gesagt

beschränken sie die Methodenparameter auf diejenigen, die innerhalb von SQL Server sinnvoll sind und sich direkt auf einen SQL Server-Typ abbilden lassen.

Die Verwendung von CLR-Methoden in SQL Server veranschaulichen wir am Beispiel einer Datenbank für eine Firma, die Riemenantriebe herstellt. Ein Riemenantrieb – oder eine Transmission – besteht aus zwei runden Seilrollen, die über einen Riemen miteinander verbunden sind. Eingesetzt werden Riemenantriebe in den unterschiedlichsten Geräten, angefangen bei Rasenmähern bis hin zu Bohrmaschinen. Ein Beispiel ist in Abbildung 3.1 zu sehen.

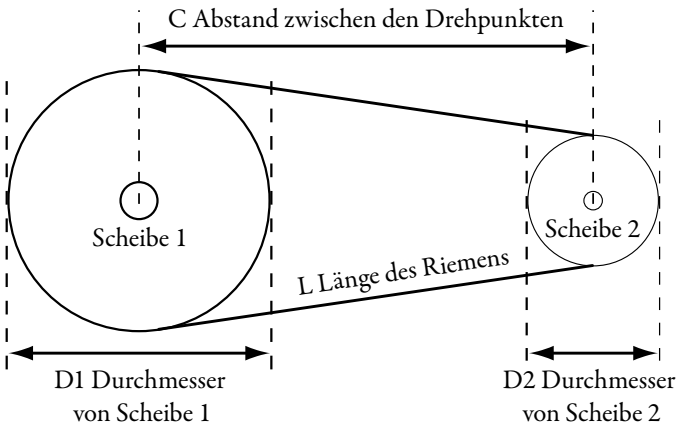


Abbildung 3.1: Riemenantriebssystem

Ein Riemen der Länge L befindet sich auf den Scheiben 1 und 2. D1 bezeichnet den Durchmesser von Scheibe 1 und D2 den Durchmesser von Scheibe 2. Der Abstand zwischen den Drehpunkten von Scheibe 1 und Scheibe 2 ist mit C angegeben.

Unsere Riemenantriebsfirma erfasst in einer Datenbank, welche Arten von Scheiben und Riemen auf Lager vorhanden sind. Die Datenbank enthält eine Tabelle zur Verwaltung der Scheiben, wobei jede Scheibe mit der Teilenummer und dem Durchmesser in Zoll aufgelistet wird. Weiterhin enthält sie eine Tabelle der Riemen, in der jeder Riemen mit der Teilenummer und der Länge in Zoll aufgeführt ist. Jede Kombination aus zwei Scheiben und einem Riemen lässt sich für eine Transmission verwenden, vorausgesetzt, dass sich die Scheiben weit genug voneinander entfernt anordnen lassen und sich der Riemen noch um beide legen lässt.

Die Firma möchte in einer Sicht alle möglichen Transmissionen anzeigen, die sich aus den auf Lager befindlichen Teilen zusammenstellen lassen. Die Transmissionssicht sollte die Teilenummern für die Scheiben und den Riemen zeigen sowie den Abstand der Scheibenmittelpunkte angeben. Natürlich muss diese Sicht alle Kombinationen von Scheiben und Riemen eliminieren, bei denen sich die Scheiben überlappen.

Der Abstand zwischen den Scheiben lässt sich über geometrische Beziehungen berechnen. Abbildung 3.2 zeigt eine Funktion, die den angenäherten Abstand (3) zwischen den beiden Scheiben einer Riementransmission berechnet, wobei die Scheibengrößen und die Riemenlänge gegeben sind. Die Methode `Pulley-Distance` (1) in der C#-Klasse `Pulley` (Scheibe) erfüllt die Anforderungen einer CLR-Methode, die in SQL Server 2005 verwendet wird. Die Berechnung des Abstands (3) basiert auf einer Formel aus dem *Machining and Metalworking Handbook* von Ronald Walsh (McGraw Hill, 1994). Per Definition ist der Abstand `distance` (4) gleich 0, wenn die Riemenlänge `BeltLength` zum Überlappen der Scheiben führt. Die Berechnung wird auf eine Dezimalstelle gerundet zurückgegeben (5).

```
public class Pulley {
① public static double PulleyDistance(
   double Pulley1Diameter, double Pulley2Diameter,
   double BeltLength) {
   ② double distance = 0, a = 2.0;
      double b = BeltLength - 1.57 *
         (Pulley1Diameter + Pulley2Diameter);
      double c = Math.Pow(Pulley1Diameter
③        - Pulley2Diameter, 2.0);
      double b1 = Math.Pow(b, 2.0) - (4 * a * c);
      if (b1 > 0) {
         distance = (b + Math.Sqrt(b1)) / (2 * a);
         if (distance <
            ((Pulley1Diameter + Pulley2Diameter) / 2)){
         ④ distance = 0;
         }  }
   ⑤ return Math.Round(distance, 1);
   }
};
```

Abbildung 3.2: Berechnung für den Scheibenabstand

Eine CLR-Methode muss in SQL Server 2005 geladen werden, bevor man sie verwenden kann, wie es in Abbildung 3.3 zu sehen ist. Die Assembly mit der Methode muss wie in Kapitel 2 beschrieben katalogisiert sein. Wenn die Assembly in den Katalog aufgenommen worden ist, muss die Funktion `PulleyDistance` in ähnlicher Weise erstellt werden, wie es bei einer T-SQL-Funktion geschieht, nämlich mit einer `CREATE FUNCTION`-Anweisung.

Anstelle des Funktionskörpers, den eine benutzerdefinierte T-SQL-Funktion haben würde, steht hier eine `EXTERNAL NAME`-Klausel. Der externe Name besteht aus drei Teilen, die durch Punkte getrennt sind. Der erste Namensbestandteil (1) dient zum Katalogisieren der Assembly. Die im zweiten Teil (2) genannte Klasse enthält die CLR-Methode, die die Funktion implementiert. Der dritte Teil (3) gibt den Namen der CLR-Methode an, die die Funktion implementiert.

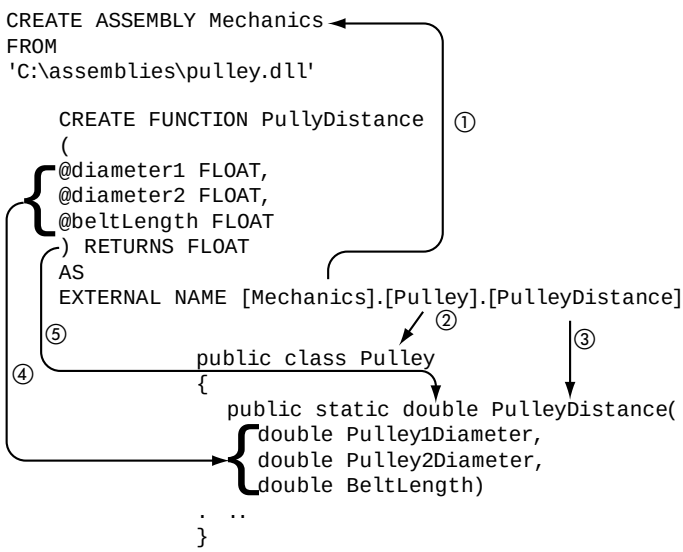


Abbildung 3.3: Die Funktion erzeugen

Der Name der T-SQL-Funktion muss nicht mit dem Namen der CLR-Methode, die die Funktion implementiert, übereinstimmen. Der Assembly-Name (1) (siehe Abbildung 3.3) ist nicht von der Groß-/Kleinschreibung abhängig, während beim Namen der Klasse (2) und der Methode (3) die Groß-/Kleinschreibung zu beachten ist. Auch für Sprachen wie Visual Basic, die Groß-/Kleinschreibung nicht unterscheiden, muss die Schreibweise im externen Namen für die Klasse und die Methode genau der, wie sie im Visual Basic-Quellcode verwendet wurde, entsprechen.

Beachten Sie, dass die einzelnen Teile von `EXTERNAL NAME` in eckige Klammern eingeschlossen sind. Diese Teilnamen müssen wie jeder Name in SQL Server behandelt werden. Im Beispiel sind die Klammern nicht notwendig, doch CLR-Sprachen unterstützen Namespaces. In diesen Fällen kann der Klassenname selbst Punkte enthalten und die eckigen Klammern verhindern, dass SQL Server die Punkte als Trennzeichen zwischen den Bestandteilen von `EXTERNAL NAME` interpretiert.

Der `CREATE FUNCTION`-Ausdruck muss genauso viele Parameter aufweisen wie die entsprechende CLR-Methode. Die Namen der Parameter brauchen nicht mit denen in der CLR-Methode (4) übereinzustimmen, doch die T-SQL-Datentypen müssen exakt kompatibel mit denen in der CLR-Methode sein – d.h., es gibt keine Umwandlung zwischen kompatiblen Typen. Analog hierzu muss der Rückgabewert (5) der CLR-Methode genau mit dem `RETURNS`-Typ der Funktion übereinstimmen.

Listing 3.1 zeigt den vollständigen Code für die C#-Datei *pulley.cs*, die die Funktion `PulleyDistanceFunction` implementiert. Sie muss als CLR-Bibliothek erstellt werden. Aus Abbildung 3.4 geht hervor, wie man *pulley.cs* mit dem C#-Befehls-

zeilen-Compiler `csc` in eine CLR-Bibliotheks-Assembly kompiliert. Listing 3.2 zeigt, wie die Assembly und die Funktion `PulleyDistanc` in einer Datenbank katalogisiert werden. Anschließend führt der Code die Funktion `PulleyDistance` aus.

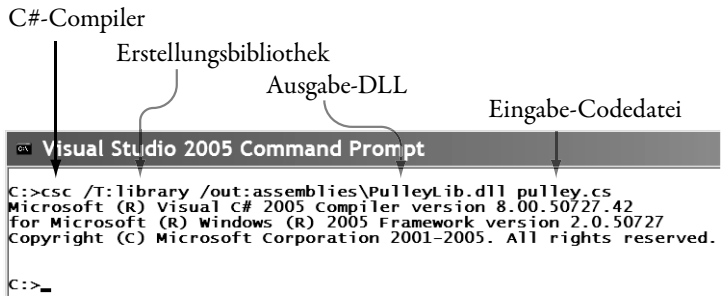


Abbildung 3.4: Assembly erstellen

```
// pulley.cs
using System;

public class Pulley
{
  public static double PulleyDistance(
    double Pulley1Diameter,
    double Pulley2Diameter,
    double BeltLength)
  {
     double length = 0, a = 2.0;
     double b = BeltLength - 1.57 *
       (Pulley1Diameter + Pulley2Diameter);
     double c = System.Math.Pow(Pulley1Diameter - Pulley2Diameter, 2.0);
     // Wenn dies nicht positiv ist, können die Scheiben nicht passen.
     double b1 = (b * b) - (4 * a * c);
     if (b1 > 0)
     {
       length = (b + Math.Sqrt(b1)) / (2 * a);
       // Prüfen, ob Scheiben passen
       if (length < ((Pulley1Diameter + Pulley2Diameter) / 2))
        {
           // 0 zurückgeben, wenn Scheiben nicht passen
           length = 0;
        }
      }
     // Eine Dezimalstelle genügt
     return System.Math.Round(length, 1);
    }
};
```

Listing 3.1: Der Pulley-Code

```
CREATE ASSEMBLY Mechanics
FROM 'C:\Assemblies\PulleyLib.dll'
GO

CREATE FUNCTION PulleyDistance
(@diameter1 FLOAT,
@diameter2 FLOAT,
@beltLength FLOAT
) RETURNS FLOAT
AS EXTERNAL NAME
[Mechanics].[Pulley].[PulleyDistance]
GO

SELECT dbo.PulleyDistance(3,2,100)
```

Listing 3.2: Die Funktion PulleyDistance katalogisieren

Eine benutzerdefinierte Funktion, die auf einer CLR-Methode basiert, verwendet man genauso wie eine benutzerdefinierte Funktion, die auf T-SQL basiert. Praktisch lässt sich kein Unterschied feststellen. Abbildung 3.5 zeigt eine `SELECT`-Anweisung (1), die die Funktion `PulleyDistance` verwendet. Die Ergebnisse (2) zeigen, dass für eine 3-Zoll-Scheibe, eine 2-Zoll-Scheibe und einen 100-Zoll-Riemen der Abstand der Scheibenmittelpunkte 46,1 Zoll beträgt.

① `SELECT dbo.PulleyDistance(3.0, 2.0, 100.0)`

②

Results | Messages

	(No column name)
1	46.1

Abbildung 3.5: Eine CLR-Funktion verwenden

Hinsichtlich der Verwendung gibt es keinen Unterschied zwischen T-SQL- und CLR-basierten benutzerdefinierten Funktionen und gespeicherten Prozeduren. Die Sicht `INFORMATION_SCHEMA.ROUTINES` unterscheidet jedoch zwischen ihnen. Die Abfrage (1) in Abbildung 3.6 gibt die Namen der Routinen und deren Implementierung zurück. Aus den Ergebnissen geht hervor, dass `PulleyDistance` (2) extern (d.h. nicht in T-SQL) und die Routine `SalesByMedianByState` (3) mit T-SQL implementiert ist.

```
① SELECT ROUTINE_NAME, ROUTINE_BODY
  from INFORMATION_SCHEMA.ROUTINES
```

Results | Messages

	ROUTINE_NAME	ROUTINE_BODY
② 96	PullyDistance	EXTERNAL
③ 97	SalesMedianByState	SQL

Abbildung 3.6: INFORMATION_SCHEMA.ROUTINES

Die Funktion `PulleyDistance` könnte in T-SQL implementiert sein, wie es in Abbildung 3.7 zu sehen ist. Eine CLR-Sprache setzt man unter anderem deshalb in SQL Server 2005 ein, weil in manchen Fällen die CLR-basierte Version eine bessere Leistung zeigt. Die Stärke von T-SQL liegt bei Mengenoperationen und nicht bei numerischen Berechnungen. Man hat schon komplexe finanzmathematische Berechnungen von T-SQL auf eine CLR-Sprache portiert und dabei beträchtliche Leistungsverbesserungen feststellen können.

```
CREATE FUNCTION PulleyDistanceTSQL
(@Diameter1 FLOAT, @Diameter2 FLOAT, @BeltLength FLOAT)
RETURNS FLOAT AS
BEGIN
DECLARE @distance FLOAT
SET @distance = 0
DECLARE @a FLOAT
SET @a = 2.0
DECLARE @b FLOAT
SET @b = @BeltLength  -
   1.57 * (@Diameter1 + @Diameter2)
DECLARE @c FLOAT
SET @c = (@diameter1 - @diameter2)
   *(@diameter1 - @diameter2)
DECLARE @b1 FLOAT
SET @b1 = (@b * @b) - (4 * @a * @c)
IF @b1 > 0
BEGIN
   SET @distance = (@b + SQRT(@b1)) / (2 * @a)
   IF @distance < ((@diameter1 + @diameter2) / 2)
      SET @distance = 0
   END
return ROUND(@distance, 1)
END
```

Abbildung 3.7: Die Funktion PulleyDistance in T-SQL

Die einfache Schleife (1) in Abbildung 3.8 bestimmt, wie lange es ungefähr dauert, die Funktion `PulleyDistance` 1 Million Berechnungen ausführen zu lassen. Die Ausführung dieser Schleife dauert auf unserem Testsystem rund 18 Sekunden (2). Ersetzt man die Funktion `PulleyDistance` (3) durch die Funktion `PulleyDistanceTSQL`, wächst die Zeit für die Schleifenausführung auf 30 Sekunden an (4).

Derartigen Vergleichen sollte man allerdings nicht blind vertrauen. Manche Anwendungen verbringen nur sehr wenig Zeit mit numerischen Berechnungen und manchmal gibt es wesentlich bessere Lösungen, die sehr wenig Rechenaufwand erfordern. Das Beispiel zeigt aber, dass CLR-Sprachen bei numerischen Berechnungen im Allgemeinen die Nase vorn haben.

```
① DECLARE @pass INT
  DECLARE @start DATETIME
  SET @start = GETDATE()
  DECLARE @result FLOAT
  SET @pass = 1000000
  redo:                                ③
  SET @result =  dbo.PulleyDistance(
  @pass % 5, @pass % 4, @pass % 100)
  SET @pass = @pass - 1
  IF @pass <> 0 GOTO redo
  print DATEDIFF(second, @start, GETDATE())

    PulleyDistance    PulleyDistanceTSQL
      ② Results          ④ Results
         18                 30
```

Abbildung 3.8: Vergleich von PulleyDistance-Funktionen

CLR-Methoden schneiden auch bei Zeichenfolgenoperationen gut ab. Eine gute Datenbank akzeptiert Zeichenfolgenwerte nicht unbesehen. Bevor eine Zeichenfolge übernommen wird, sollte sie auf geeignete Formatierung hin überprüft werden.

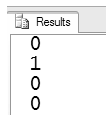
Eine US-Sozialversicherungsnummer (SSN) besteht aus drei Ganzzahlen, die durch Bindestriche getrennt sind, zum Beispiel 123-45-6789. Mit T-SQL lässt sich feststellen, ob eine Zeichenfolge wirklich eine SSN darstellt. Die in Abbildung 3.9 gezeigte benutzerdefinierte Funktion `ValidateSSNTSQL` (1) ist für diesen Zweck vorgesehen. Sie überprüft (2), ob ausschließlich numerische Zeichen und Bindestriche an den vordefinierten Positionen verwendet werden. In diesem Fall gibt sie 1 zurück, andernfalls (3) den Rückgabewert 0.

```
①CREATE FUNCTION ValidateSSNTSQL(@ssn CHAR(11))
  RETURNS BIT
  BEGIN
  RETURN CASE
   ② WHEN @ssn LIKE
        '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]'
     THEN 1
   ③ ELSE 0
  END
  END
```

Abbildung 3.9: Gültigkeitsprüfung einer US-Sozialversicherungsnummer (SSN) mit T-SQL

Abbildung 3.10 zeigt, dass `ValidateSSNTSQL` bei einer gültigen SSN (1) eine 1 (2) als Ausgabe und andernfalls eine 0 produziert. Es gibt durchaus bessere Implementierungen der `ValidateSSNTSQL`-Funktion, doch im Allgemeinen zerlegt sie die Eingabezeichenfolge und testet die einzelnen Teile.

```
   print dbo.ValidateSSNTSQL('asdf')
① print dbo.ValidateSSNTSQL('123-45-6789')
   print dbo.ValidateSSNTSQL('123-45.6789')
   print dbo.ValidateSSNTSQL('123-45-6a89')
```



②

Abbildung 3.10: Die ValidateSSNTSQL-Funktion verwenden

Die CLR kann ähnlich wie `LIKE` mit regulären Ausdrücken umgehen. Die C#-Funktion in Abbildung 3.11 prüft eine Zeichenfolge, ob sie eine korrekt formatierte SSN darstellt. Diese Funktion definiert das Format einer SSN mit einem regulären Ausdruck (1) und nicht mit imperativem Code. Die Funktion `ValidateSSN` prüft mit der Methode `Match` (2), ob die als Parameter übergebene Zeichenfolge korrekt formatiert ist.

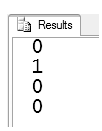
```
public class ValidationFunctions
{
   static readonly Regex SSNRegex;
   static ValidationFunctions()
   {
   ① SSNRegex = new Regex(@"\d{3}-\d{2}-\d{4}");
   }
   public static int ValidateSSN(string ssn)
   {
   ② return SSNRegex.Match(ssn).Success ? 1 : 0;
   }
};
```

Abbildung 3.11: CLR-basierte Gültigkeitsprüfung

Aus Abbildung 3.12 geht hervor, dass die `ValidateSSN`-Funktion (2) die gleichen Ergebnisse (3) wie die Funktion `ValidateSSNTSQL` produziert. Wie jede CLR-basierte benutzerdefinierte Funktion muss sie in die Datenbank aufgenommen werden (1), bevor man sie verwenden kann.

```
① CREATE FUNCTION ValidateSSN(@ssn NCHAR(11))
   RETURNS int
   AS EXTERNAL NAME
   ValidationAsm.ValidationFunctions.ValidateSSN

② print dbo.ValidateSSN('asdf')
   print dbo.ValidateSSN('123-45-6789')
   print dbo.ValidateSSN('123-45.6789')
   print dbo.ValidateSSN('123-45-6a89')
```



④

Abbildung 3.12: Die Funktion ValidateSSN verwenden

In der `ValidateSSN`-Funktion von Abbildung 3.11 fällt auf, dass sie die SSN nicht mit einer Codesequenz überprüft, sondern ihr Format mit einem regulären Ausdruck definiert. Um diese Technik zu verwenden, müssen Sie sich natürlich mit regulären Ausdrücken auskennen. In der MSDN-Dokumentation zum .NET Framework sind reguläre Ausdrücke beschrieben. Einen guten Ausgangspunkt bildet auch das Buch »Mastering Regular Expressions« von Jeffrey E.F. Friedl (O'Reilly). Und schließlich sei die Webseite *http://www.regexlib.com* genannt, auf der eine Online-Bibliothek von regulären Ausdrücken verwaltet wird.

```
public partial class UserDefinedFunctions
{                              ①
   static readonly Regex ssnRegex
   = new Regex(                ②
@"^(?!078-05-1120)(?!000)\d\d\d-(?!00)\d\d-(?!0000)\d\d\d\d$"
   ,RegexOptions.Compiled | RegexOptions.Singleline
   );

   public static bool CheckSSN(string ssn)
   {
      if(ssn == null) return false;
      Match m = ssnRegex.Match(ssn);
      return m.Success;③
   }
};
```

Abbildung 3.13: Verbesserte SSN-Gültigkeitsprüfung

Wir erweitern nun die Funktion `ValidateSSN` etwas, um zu zeigen, wie nützlich reguläre Ausdrücke sind. Zunächst einmal ist festzustellen, dass in einer SSN nicht alle Zahlen zulässig sind. Zum Beispiel stellt 078-05-1120 keine gültige SSN dar, weil sie im Jahre 1938 in einer Werbeanzeige verwendet wurde! Einzelheiten hierzu finden Sie unter *http://en.wikipedia.org/wiki/Social_security_number*. Weiterhin darf keiner der drei Zahlenbestandteile ausschließlich Nullen enthalten. So ist zum Beispiel 0000-45-9789 keine gültige SSN. Schließlich war der in der CLR-Methode `ValidateSSN` verwendete reguläre Ausdruck nicht korrekt, weil zusätzliche Zeichen auftreten konnten. Wir erstellen nun eine verbesserte Version von `ValidateSSN` namens `ValidateSSNFull`, die all diese Dinge überprüft. Diese neue Version verwenden und erweitern wir, um auch für die übrigen Features zu zeigen, wie man SQL Server 2005 mit der CLR erweitert.

Abbildung 3.13 zeigt die `ValidateSSNFull`-Implementierung, die einen verbesserten regulären Ausdruck (2) verwendet. Eine vollständige Beschreibung von regulären Ausdrücken würde den Rahmen dieses Buchs sprengen. Hier sei auf die genannten Quellen verwiesen. In diesem Fall stellen die Zeichen ^ und $ sicher, dass die Zeichenfolge immer die richtige Länge aufweist. Der Teil `(!!078-05-1120)` des Ausdrucks testet, ob es sich nicht um die ungültige SSN aus dem

Jahre 1938 handelt. Andere Teile des Ausdrucks stellen auf diese Weise fest, ob es sich durchgängig um Nullen handelt. Die Methode vergleicht den regulären Ausdruck mit der Eingabezeichenfolge (3) und gibt bei Übereinstimmung `true` zurück.

Der reguläre Ausdruck selbst wird in einer Variablen namens `ssnRegex` (1) gespeichert. Die Variable ist statisch, weil ein regulärer Ausdruck vor seiner Verwendung kompiliert werden muss. Stellt man ihn in eine statische Variable, passiert das nur einmal, selbst wenn der Ausdruck immer wieder verwendet wird.

Die Kapitel 2 und 6 betonen, dass Assemblys nach Möglichkeit mit `WITH PERMISSION_SET = SAFE` zu laden sind. Um dies zuzulassen, ist die Variable `ssnRegex` als `readonly` markiert. Jede Assembly, die eine veränderbare statische Variable enthält, muss mit `WITH PERMISSION_SET = UNSAFE` geladen werden.

Jetzt wird der Eingabeparameter getestet, um festzustellen, ob er null ist (2). Wenn ja, gibt `ValidateSSNFull` das Ergebnis `false` zurück. Das ist möglicherweise keine gute Wahl, doch kommen wir später in diesem Kapitel zu weiteren SQL-Typen, die immer `NULL` sein können, und CLR-Typen, die es manchmal nicht sein können. Und wir zeigen, wie man die beiden Typsysteme unter einen Hut bringt.

Die T-SQL-Implementierung des SSN-Checkers können wir leicht auf diese zusätzlichen Tests erweitern, indem wir einfach noch andere `WHEN`-Klauseln in den Code von Abbildung 3.9 einbauen. Letztlich besitzen reguläre Ausdrücke und T-SQL ungefähr die gleichen Möglichkeiten, um die Gültigkeit einer Zeichenfolge zu prüfen. Warum machen wir uns die Mühe, hierfür die CLR zu verwenden?

Es zeigt sich, dass die Implementierung mit CLR und regulärem Ausdruck schneller als die T-SQL-Implementierung ist, wenn auch nicht sehr viel schneller. Der eigentliche Wert der regulären Ausdrücke besteht darin, dass es sich um deklarative Ausdrücke handelt, die man außerhalb von T-SQL verwenden kann. Reguläre Ausdrücke werden auf breiter Front unterstützt. Eine Anwendung der mittleren Schicht oder eine Clientanwendung kann den gleichen regulären Ausdruck wie diese Funktion verwenden und erhält dieselben Ergebnisse. Wenn Sie mit derartigen regulären Ausdrücken in SQL Server 2005 arbeiten möchten, sollten Sie sie auch für Clients von der Datenbank aus selbst zugänglich machen und nicht nur eine Spezifikation angeben. Listing 3.3 zeigt ein Beispiel für eine Methode, die sich in die Implementierung von Abbildung 3.13 einfügen und als Funktion in SQL Server 2005 katalogisieren lässt. Clients greifen darauf zurück, um mit dem regulären Ausdruck, der für die Überprüfung von SSNs verwendet wird, synchron zu bleiben.

```
public static string SSNRegex()
{
  return ssnRegex.ToString();
}
```

Listing 3.3: Regulärer Ausdruck für das Überprüfen einer SSN

Handelt es sich um eine von der CLR unterstützte Sprache, kann sie sogar dieselbe Assembly verwenden, die in SQL Server katalogisiert wurde! Es gibt dann keine Fehlinterpretationen eines Zeichenfolgenformats mehr. Server und Client können über einen regulären Ausdruck denselben Code für die Gültigkeitsprüfung verwenden.

Es sind noch weitere Dinge zu testen, doch was wir in `ValidateSSNFull` realisiert haben, genügt zunächst einmal, um zu zeigen, dass reguläre Ausdrücke in der CLR sehr nützliche Ergänzungen zu SQL Server 2005 sind.

3.3 Details zu den CLR-Erweiterungen

Bislang haben wir uns die Grundlagen angesehen, wie sich mit der CLR die Funktionalität von SQL Server 2005 erweitern lässt. Wir haben gezeigt, wie man eine öffentliche statische Methode einer öffentlichen Klasse als Funktion verfügbar macht, auf die man über T-SQL zugreifen kann. Jetzt dringen wir tiefer in die Details ein.

Die Assembly und die Methoden in den vorherigen Funktionsbeispielen enthalten nicht genügend Informationen, dass SQL Server 2005 sie in der gleichen Weise wie T-SQL-Funktionen verwalten kann. Abbildung 3.14 zeigt zum Beispiel eine triviale T-SQL-Funktion, die die an sie übergebene Zeichenfolge zurückgibt. Das `INFORMATION_SCHEMA` für diese Funktion verzeichnet eine Eigenschaft, die die Funktion als deterministisch ausweist. In SQL Server liefert eine *deterministische Funktion* für eine gegebene Eingabe immer das gleiche Ergebnis zurück, wie es hier bei der trivialen Funktion der Fall ist. Es gibt einige Fälle, wann eine Funktion deterministisch sein muss – wenn sie zum Beispiel dazu dient, den Wert einer persistent gespeicherten berechneten Spalte einer Tabelle zu ermitteln.

Nichts in der Funktionsdefinition von Abbildung 3.14 weist darauf hin, dass die Funktion deterministisch ist. Woran also erkennt SQL Server dies? SQL Server analysiert den Code und stellt fest, dass er keine Funktionen aufruft, die nicht deterministisch sind. Außerdem hängt der Rückgabewert der Funktion nicht vom Zustand der Datenbank ab.

```
CREATE  FUNCTION Same(@s CHAR(50))
RETURNS CHAR(50)
WITH SCHEMABINDING
AS
BEGIN
RETURN @s
END

select ROUTINE_NAME, IS_DETERMINISTIC
from INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_NAME='Same'
```

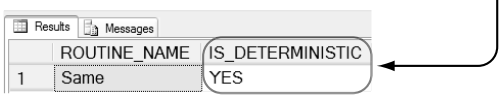


Abbildung 3.14: Deterministische Funktion

In CLR-Methoden analysiert SQL Server nicht den Code, sondern stützt sich stattdessen auf Metadaten der Assembly. Das liefert die Informationen, die SQL Server beim Analysieren einer T-SQL-Funktion erhalten würde. Dazu wird ein Feature der CLR – ein so genanntes Attribut – verwendet. Weitere Informationen hierzu finden Sie in der MSDN-Dokumentation für das .NET Framework zum Thema »Attribute-Klasse«.

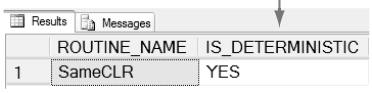
Einfach ausgedrückt greift SQL Server auf Attribute zurück, um Name/Wert-Paare mit der CLR-Methode, die eine benutzerdefinierte Funktion implementiert, zu verbinden. SQL Server liest diese Name/Wert-Paare, wenn die Funktion katalogisiert wird. Attribute in der CLR ähneln konzeptionell und bezüglich der Verwendung den erweiterten Eigenschaften in SQL Server. Damit lassen sich Datenbankobjekten beliebige Name/Wert-Paare hinzufügen.

Auf CLR-Methoden, die gespeicherte Prozeduren, Funktionen oder Trigger implementieren, können mehrere Attribute angewandt werden: `SqlProcedure`, `SqlFunction` bzw. `SqlTrigger`. Alle diese Attribute befinden sich im Namespace `Microsoft.SqlServer.Server`. SQL Server verwendet nur eines dieser Attribute, nämlich `SqlFunction`. Später in diesem Kapitel erfahren Sie, wie die anderen verwendet werden.

Abbildung 3.15 zeigt eine C#-Implementierung der trivialen T-SQL-Funktion von Abbildung 3.3. Durch das Attribut `SqlFunction` werden der Funktion `SameCLR` Metadaten hinzugefügt, die anzeigen, dass die Funktion deterministisch ist. Wenn wir diese Funktion in den Katalog von SQL Server 2005 aufnehmen und dann einen Blick auf die Funktion im `INFORMATION_SCHEMA` werfen, zeigt sich, dass SQL Server anhand des Attributs `SqlFunction` erkannt hat, dass diese Funktion deterministisch ist.

```
[Microsoft.SqlServer.Server.SqlFunction(
  IsDeterministic=true)]
public static string SameCLR(string s)
{
  return s;
}

select ROUTINE_NAME, IS_DETERMINISTIC
from INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_NAME='Same'
```



	ROUTINE_NAME	IS_DETERMINISTIC
1	SameCLR	YES

Abbildung 3.15: Das Attribut SqlFunction verwenden

SQL Server verwendet fünf Eigenschaften im `SqlFunction`-Attribut: `DataAccess`, `SystemDataAccess`, `IsDeterministic`, `IsPrecise` und `FillRowMethodName`. Später gehen wir ausführlich auf diese Eigenschaften ein (auf `SystemDataAccess` und `DataAccess` erst in Kapitel 4).

Listing 3.1, Abbildung 3.4 und Listing 3.2 zeigen, wie eine Funktion erzeugt und katalogisiert wird. Visual Studio 2005 ist eng mit SQL Server integriert und kann Funktionen, gespeicherte Prozeduren und Trigger direkt in SQL Server 2005 bereitstellen – d.h., Sie brauchen weder etwas auf der Befehlszeile zu kompilieren, wie es in Abbildung 3.4 zu sehen ist, noch ein Installationsskript zu verwenden, wie es Listing 3.2 gezeigt hat. In Visual Studio 2005 drücken Sie einfach F5 wie beim Debuggen eines C#- oder Visual Basic-Programms. Visual Studio kompiliert die Funktion und stellt sie in SQL Server 2005 bereit. Daraufhin können Sie mit dem Debuggen des Codes beginnen und SQL Server sogar im Einzelschritt abarbeiten.

Visual Studio benötigt hierfür zusätzliche Informationen. Diese kommen von den Attributen `SqlFunction`, `SqlProcedure` und `SqlTrigger` sowie von einigen anderen Attributen, auf die wir in Kapitel 5 noch eingehen. Anhang C gibt einen Überblick über die Verwendung von Visual Studio 2005 mit SQL Server.

3.4 System.Data.SqlTypes

Es gibt einige Einschränkungen hinsichtlich der Parametertypen in CLR-Methoden, die SQL Server 2005 erweitern. Allgemein gilt, dass sie für SQL Server 2005 »sinnvoll« sein müssen. Außerdem existieren einige Beschränkungen zum Design der Klasse, die die Methoden implementiert, die über T-SQL verwendet werden. Mit diesen Fragen setzen wir uns jetzt auseinander. In diesem Zusam-

menhang beschäftigen wir uns auch mit einigen neuen CLR-Klassen aus dem Namespace `System.Data.SqlTypes`, die den Typen in SQL Server 2005 besser entsprechen als es bei anderen CLR-Typen der Fall ist.

Die CLR-Funktion `CheckSSN` aus Abbildung 3.13 im vorherigen Abschnitt ist zwar nützlich, doch besitzt sie nicht alle Fähigkeiten einer nativen T-SQL-Funktion. Abbildung 3.16 zeigt, was passiert, wenn man `NULL` an `CheckSSN` übergibt (1). Wie erwartet, gibt die Funktion eine 0 zurück (2). Doch ist das wirklich sinnvoll? In einer Datenbank stellt man mit `NULL` einen fehlenden oder ungeeigneten Wert dar. Wenn der Wert fehlt oder ungeeignet ist, lässt sich nicht bestimmen, ob er gültig ist oder nicht.

```
① print dbo.CheckSSN(NULL)
② Messages
  0
```

Abbildung 3.16: Zweifelhaftes Ergebnis, das bei Übergabe von NULL entsteht

3.4.1 Ein paar Worte zu Nullwerten

Die meisten Sprachen bieten irgendeine Möglichkeit, um einen so genannten Nullwert darzustellen. Diesen können Sie einsetzen, wenn ein Wert verlangt wird, aber kein Wert vorhanden ist. Was diese Sprachen darzustellen versuchen, bezeichnen wir hier als *Nullwert*, weil der Begriff »null« oftmals in anderer Bedeutung verwendet wird. So setzt sowohl `SET @a = NULL` in einem T-SQL-Skript als auch `a = null;` in einem C#-Programm eine Variable auf einen Nullwert.

Dies kann zu Verwirrung führen, weil ein Nullwert in der Mengenlehre ein sehr spezifisches Verhalten aufweist. Das wird zwar von relationalen Datenbanken wie SQL Server 2005 unterstützt, normalerweise aber nicht von Programmiersprachen wie C# oder Visual Basic. In einer Datenbank zum Beispiel ergibt ein Vergleich auf einen Nullwert normalerweise `false`, selbst wenn beide Werte Nullwerte sind. In C# liefert der Vergleich eines Nullwerts mit einem anderen normalerweise `true`. C# bietet aber auch mehrere Möglichkeiten, um einen Nullwert darzustellen.

Die Datentypen im Namespace `System.Data.SqlTypes` sollen die Unterschiede überbrücken, wie Nullwerte in SQL Server 2005 und in CLR-Sprachen normalerweise behandelt werden. Sehen wir uns an, warum wir die `System.Data.SqlTypes`-Typen brauchen.

Die Integration von T-SQL und der CLR-Sprachen ist in SQL Server 2005 recht nahtlos realisiert – bis auf einen Bereich: die Verwendung von Nullwerten. Wie ein Nullwert behandelt wird, hängt vom Kontext ab, in dem er auftritt. Abbildung 3.17 zeigt einen der Unterschiede, wie T-SQL und C# mit Nullwerten umge-

hen. Auf den ersten Blick scheint es sich um dasselbe Programm zu handeln. Beide erzeugen zwei Zeichenfolgen, von denen eine ein Nullwert ist, und verketten sie dann. Das Ergebnis der Verkettung ist in T-SQL (1) ein Nullwert. Dagegen behandelt das C#-Programm den Nullwert so, als wäre er eine Zeichenfolge der Länge null, und produziert einen echten Wert als Ergebnis (2).

Das C#-Programm von Abbildung 3.17 würde in Visual Basic geschrieben die gleichen Ergebnisse liefern. Das heißt aber nicht, dass C# und Visual Basic Nullwerte in der gleichen Weise verarbeiten. Abbildung 3.18 zeigt Visual Basic- und C#-Programme, die denen von Abbildung 3.17 ähneln. Der Unterschied in den Programmen von Abbildung 3.18 besteht darin, dass weder das Visual Basic- (1) noch das C#-Programm (2) der zweiten Zeichenfolge einen Nullwert speziell zuweisen. Das Visual Basic-Programm behandelt die zweite Zeichenfolge als Nullwert, das C#-Programm unterbindet es von vornherein, nicht zugewiesene Variablen zu verwenden.

```
DECLARE @s1 VARCHAR(MAX)
SET @s1 = 'asdf'
DECLARE @s2 VARCHAR(MAX)
SET @s2 = NULL
DECLARE @s3 VARCHAR(MAX)
SET @s3 = @s1 + @s2
SELECT @s3 AS [string + NULL]
```

① →

Results | Messages

	string + NULL
1	NULL

```
static void Main(string[] args)
{
   string s1 = "asdf";
   string s2 = null;
   string s3 = s1 + s2;
   Console.WriteLine(s3);
}
```

② →

```
Visual Studio 2005 Command Prompt
C:>UsingNullString.exe
asdf
C:>
```

Abbildung 3.17: Nullwerte in T-SQL und C#

Aus den Abbildungen 3.17 und 3.18 lässt sich mitnehmen, dass es sehr wenige Gemeinsamkeiten bei der Behandlung von Nullwerten unter T-SQL und CLR-Sprachen gibt. Wenn Sie mit T-SQL vertraut sind und jetzt Routinen einer CLR-Sprache in Ihre Anwendung integrieren möchten, sollten Sie sich unbedingt mit den Besonderheiten beschäftigen, wie die CLR-Sprache Ihrer Wahl mit Nullwerten umgeht. Wenn Sie umgekehrt in einer CLR-Sprache zu Hause sind und jetzt diese Sprache in SQL Server 2005 integrieren möchten, sollten Sie verstehen, wie SQL Server Nullwerte behandelt.

```
Module Module1
Sub Main()
  Dim s1 As String
① Dim s2 As String
  s1 = "asdf"
  Dim s3 As String
  s3 = s1 + s2
  Console.WriteLine(s3)
End Sub
End Module
```

```
class Program
{
  static void Main(string[] args)
  {
    string s1 = "asdf";
  ② string s2;
    string s3 = s1 + s2;      ③
    Console.WriteLine
  }                  Use of unassigned local variable 's2'
}
```

Abbildung 3.18: Behandlung einer nicht zugewiesenen Variablen in Visual Basic und C#

Es ginge über den Rahmen dieses Buchs hinaus, die Behandlung von Nullwerten in T-SQL oder den CLR-Sprachen erschöpfend zu erläutern. Dieses Buch folgt allerdings einer Konvention beim Verweisen auf Nullwerte. Der Begriff *Nullwert* bezeichnet hier das generische Konzept eines Nullwerts, unabhängig von seiner Herkunft. Der Begriff *NULL* steht für einen Nullwert in SQL Server. Und wenn schließlich *null* allein erscheint, ist ein Nullwert von C# gemeint, während sich *nothing* auf einen Nullwert von Visual Basic bezieht.

3.4.2 SqlTypes verwenden

Die skalaren Datentypen in SQL Server 2005 unterscheiden sich manchmal von denen in der CLR, wie sie physisch in SQL Server dargestellt werden, und darin, wie Nullwerte dargestellt und verwendet werden. Ein neuer Satz von Datentypen im Namespace `System.Data.SqlTypes` bildet die physische Darstellung und die Darstellung bei Operationen mit null in SQL Server-Datentypen nach.

`System.Data.SqlTypes` stellt die Typdefinitionen für Datentypen bereit, die man normalerweise in Datenbanken verwendet. Diese werden auf CLR-Typen abgebildet und unterstützen über zusätzliche Eigenschaften das Konzept einer `NULL`, wie es in SQL Server 2005 üblich ist.

Zu den Neuigkeiten der CLR, die auf den ersten Blick für das Nachbilden von Datentypen in SQL Server 2005 nützlich scheinen, gehört die Unterstützung von Vorlagen, die in der MSDN unter »Vorlagen« dokumentiert sind. Kurz gesagt stellt eine Vorlage eine Möglichkeit dar, einem vorhandenen Typ Methoden hinzuzufügen, ohne von ihm abzuleiten. Und mit diesem Feature lassen sich neue Typen erzeugen, die sich wie Werttypen verhalten, aber auch null sein können.

Zum Beispiel kann man eine Variable vom Typ `System.Nullable<int>` deklarieren. Diese Variable ist ein Werttyp, kann aber auf null gesetzt werden. In C# lässt sich das Synonym `int?` für `System.Nullable<int>` verwenden.

Auch wenn sich die neuen Nullable Types der CLR in vielerlei Hinsicht wie die Typen in `System.Data.SqlTypes` verhalten, müssen wir darauf eingehen, weil man sie nicht als Parameter oder Rückgabewerte bei Funktionen und gespeicherten Prozeduren verwenden kann. Denn diese Typen entsprechen keinem SQL Server 2005-Typ. Wenn nullbare Typen notwendig sind, müssen Sie hierfür die Typen von `System.Data.SqlTypes` verwenden.

Abbildung 3.19 zeigt die SSN-Validierungsfunktion von Abbildung 3.13 in einer neuen Version, die mehr auf T-SQL ausgerichtet ist. Sie verwendet den Typ `SqlBoolean` (1) aus dem Namespace `System.Data.SqlTypes` für den Rückgabewert und den Typ `SqlString` (aus demselben Namespace) für den Eingabeparameter `ssn`. Die Funktion testet `ssn`, ob der Wert `null` ist, und gibt in diesem Fall auch `null` (2) zurück. Ist `ssn` nicht `null`, testet die Funktion diesen Wert gegen einen regulären Ausdruck – den gleichen wie in Abbildung 3.13 – und gibt `true` zurück, wenn eine Übereinstimmung gefunden wird.

```
static readonly Regex SSNRegex = new Regex(
@"^(?!078-05-1120)(?!000)\d\d\d-(?!00)\d\d-(?!0000)\d\d\d\d$"
     ,RegexOptions.Compiled | RegexOptions.Singleline
     );
public static SqlBoolean①
   ValidateSSNFullNull(SqlString ssn)
{
   if (ssn.IsNull)
   {
      return SqlBoolean.Null;②
   }
   Match match = SSNRegex.Match(ssn.Value);
   return match.Success;③
}
```

Abbildung 3.19: SSN-Validierung im Stil von T-SQL

Die T-SQL-Funktion `ValidateSSNTSQL` in Abbildung 3.9 könnte ebenfalls auf `NULL` testen und eine `NULL` zurückgeben. T-SQL-Funktionen lassen sich auch mit der Funktionsoption `RETURNS NULL ON NULL INPUT` erstellen, damit Sie Eingaben nicht auf `NULL` testen müssen. Diese Funktionsoption können Sie auch für `ValidateSSNFullNull` verwenden, wenn Sie die Funktion in SQL Server 2005 katalogisieren, doch sollten Sie weiterhin auf eine null-Eingabe testen. Es handelt sich ja um CLR-Code, der auch von anderem Code und nicht nur von T-SQL aus aufgerufen werden kann.

Alle Datentypen aus dem Namespace `SqlTypes` besitzen drei Eigenschaften: `Null`, `IsNull` und `Value`. Die Eigenschaft `IsNull` entspricht der `IS NULL`-Klausel in T-SQL und die Eigenschaft `Null` dem Schlüsselwort `NULL` in T-SQL. Die Eigenschaften `IsNull` und `Null` erlauben es einer CLR-basierten Methode, Nullwerte in der gleichen Weise zu behandeln wie in T-SQL (siehe Abbildung 3.19).

Jeder Datentyp in `System.Data.SqlTypes` besitzt einen zugrunde liegenden CLR-Datentyp. Auf den zugrunde liegenden Datentyp greift man über die `Value`-Eigenschaft zu. Wenn `IsNull` für eine Instanz eines `System.Data.SqlTypes`-Typs `true` zurückgibt, löst der Zugriff auf seine `Value`-Eigenschaft eine Ausnahme aus, wie es beim Zugriff auf jeden Nullwert in der CLR geschieht.

Der von der `Null`-Eigenschaft eines `System.Data.SqlTypes`-Typs zurückgegebene Wert ist selbst typisiert – d.h., er unterscheidet sich von `NULL` in SQL Server 2005 oder den `null`- und `nothing`-Werten in C# bzw. Visual Basic. Sowohl `SqlBoolean b2 = SqlInt32.Null;` als auch `SqlBoolean b3 = null;` erzeugen zur Kompilierzeit Fehler. Sie müssen `SqlBoolean.Null` zurückgeben, um `null` darzustellen, wenn der Rückgabetyp `SqlBoolean` ist. Es gibt keine generische Null wie in T-SQL und den CLR-Sprachen.

Beachten Sie, dass `System.Data.SqlTypes`-Typen gewöhnlich größer sind (mehr Speicher belegen) als ihre zugrunde liegenden CLR-Typen.

Per Konvention entspricht das Suffix eines `System.Data.SqlType`-Typnamens dem zugrunde liegenden CLR-Typ. Viele Sprachen, einschließlich C# und Visual Basic 2005, verwenden einen Alias für den CLR-Typnamen. Zum Beispiel heißt der zugrunde liegende Datentyp für `SqlInt32` im CLR-Typsystem `System.Int32` und in C# `int`. Tabelle 3.1 zeigt (von links nach rechts) die Zuordnung zwischen einem Typ von `System.Data.SqlTypes`, dem nativen Typ, den er in SQL Server repräsentiert, dem zugrunde liegenden Typ in der CLR, dem Alias für den Typ in C# und schließlich dem Typ, der für einen ADO.NET-`SqlParameter` verwendet wird.

SqlTypes	SQL Server	Zugrundeliegender CLR-Typ	C#-Alias	ADO.NET
SqlBinary	BINARY VARBINARY	Byte[]	byte[]	Binary VarBinary TimeStamp Image
SqlInt64	BIGINT	Int64	Long	BigInt
SqlInt32	INT	Int32	Int	Int
SqlInt16	SMALLINT	Int16	Short	SmallInt
SqlByte	TINYINT	Byte	Byte	Byte

SqlTypes	SQL Server	Zugrundeliegender CLR-Typ	C#-Alias	ADO.NET
SqlChars	NVARCHAR(MAX)	String	String	NVarChar
SqlString	NVARCHAR	String	String	Char VarChar
	NCHAR			NChar
SqlDateTime	DATETIME	DateTime	DateTime	DateTime
	SMALLDATETIME			SmallDateTime
SqlDecimal	DECIMAL	Decimal	Decimal	Decimal
	NUMERIC			Numeric
SqlDouble	FLOAT	Double	Double	Float
SqlSingle	REAL	Single	Float	Real
SqlMoney	MONEY	Decimal	Decimal	Money
	SMALLMONEY			SmallMoney
SqlGuid	UNIQUE IDENTIFIER	Guid	Guid	UniqueId
SqlBoolean	BIT	Boolean	Bool	Boolean

Tabelle 3.1: SqlTypes-Abbildungen auf andere Typen

Wenn Sie in T-SQL arbeiten, können Sie recht oft den Typ einer Variablen ignorieren. Zum Beispiel zeigt Abbildung 3.20, dass sich der Zeichenfolgentyp auf einen ganzzahligen Wert setzen lässt (1). Auch kann man einen Zeichenfolgenwert zu einer Ganzzahlvariablen (2) und sogar eine Ganzzahl zu einer Zeichenfolge addieren (3). Hier findet eine so genannte *implizite Typumwandlung* statt.

```
   DECLARE @i INT
   DECLARE @s VARCHAR(10)
①  SET @s = 3
②  SET @i = '1' + @s
   SELECT @i
③  SET @i = 1 + '2'
   SELECT @i
```

Abbildung 3.20: Typen in T-SQL verwenden

Manchmal entspricht die von T-SQL vorgenommene implizite Umwandlung nicht dem, was Sie wirklich beabsichtigen. Abbildung 3.21 zeigt ein Beispiel, bei dem T-SQL zwei Zeichenfolgen verkettet hat (1). Wenn Sie eigentlich eine arithmetische Addition im Sinn haben, müssen Sie eine der beiden Zeichenfolgen mit `CAST` in einen `INT` konvertieren (2). Bei einer derartigen *expliziten Typumwandlung* legen Sie genau fest, wie die Umwandlung erfolgen soll.

```
   SELECT
① '1' + '3',
② CAST('1' AS INT) + '3'
```

Results | Messages

	(No column name)	(No column name)
1	13	4

Abbildung 3.21: Typen explizit umwandeln

Die meisten CLR-Sprachen erlauben implizite und explizite Umwandlungen ähnlich denen in T-SQL. Tabelle 3.2 zeigt die Möglichkeiten der Umwandlung zwischen `SqlTypes` und ihren zugeordneten CLR-Typen. Es sind nicht alle impliziten Umwandlungen aufgeführt; es gibt eine implizite Umwandlung vom zugrunde liegenden CLR-Typ wie in Tabelle 3.1 zu seinem zugeordneten `SqlTypes`-Typ, außer bei `SQLBinary`, `SqlBytes` und `SqlChars`.

In Tabelle 3.2 steht ein I für eine implizite Umwandlung in den Typ, der links vom I genannt ist, aus dem Typ, der über dem I steht. Ein E kennzeichnet eine explizite Umwandlung in den Typ links vom E aus dem Typ über dem E.

Wie der Name sagt, werden die unterstützten impliziten Typumwandlungen von den CLR-Sprachen automatisch vorgenommen. Jede CLR-Sprache, die explizite Umwandlungen unterstützt, verwendet hierfür allerdings eine eigene Syntax. Abbildung 3.22 zeigt ein Beispiel für Typumwandlungen in C#. Zuerst findet eine implizite Umwandlung von einem `int` in einen `SqlInt32` statt (1). Darauf folgt eine explizite Umwandlung von einem `SqlInt32` in einen `int` (2). Da es keine implizite Umwandlung von `SqlInt32` nach `int` (3) gibt, erzeugt diese Anweisung einen Fehler zur Kompilierzeit. Viele Typen von `SqlTypes` lassen sich explizit in einen `SqlString` konvertieren (4).

```
static void Main(string[] args)
{
① SqlInt32 i1 = 9;
② int i2 = (int)i1;
③ int i3 = i1;
④ SqlString s = (SqlString)i1;
}
```

Error List

1 Error | 0 Warnings | 0 Messages

	Description
1	Cannot implicitly convert type 'System.Data.SqlTypes.SqlInt32' to 'int'. An explicit conversion exists (are you missing a cast?)

Abbildung 3.22: CLR-Konvertierungen

Von Nach	Sql Binary	Sql Byte	Sql Bytes	Sql Boolean	Sql Chars	Sql DateTime	Sql Decimal	Sql Double	Sql Guid	Sql Int16	Sql Int32	Sql Int64	Sql Money	Sql Single	Sql String
SqlBinary	E								E						
SqlByte	E	E					E	E		E	E	E	E	E	E
SqlBytes	E		E												
SqlBoolean	E	E					E	E		E	E	E	E	E	E
SqlChars	E														E
SqlDataTime						E									E
SqlDecimal				E			E	E						E	E
SqlDouble		I		E				E		I	I	I	I	I	E
SqlGuid	E								E						E
SqlInt16		I		E			E	E		E	E	E	E	E	E
SqlInt32		I		E			E	E		I	E	E	E	E	E
SqlInt64		I		E			E	E		I	I	E	E	E	E
SqlMoney		I		E			EI	E		I	I	I	E	E	E
SqlSingle		I		E			I	E		I	I	I	I	E	E
SqlString			E	E		E	E	E	E	E	E	E	E	E	E

Tabelle 3.2: SqlTypes-Umwandlungen

Datentypen in der CLR unterstützen implizite und explizite Umwandlungen mithilfe der bekannten Methoden `op_Implicit` und `op_Explicit`. Zum Beispiel implementiert `SqlInt32` die Methoden `op_Implicit(Int32)`, `op_Implicit(SqlByte)` und `op_Implicit(SqlInt16)`, wie es die MSDN-Dokumentation zum Thema »SqlInt32.op_Implicit« angibt. Deshalb zeigen die Spalten mit den Überschriften `SqlByte`, `SqlInt16` und `SqlInt32` von Tabelle 3.2 ein I in der Zeile, in der links `SqlInt32` steht.

Ein `SqlString` lässt sich explizit in die meisten `SqlTypes` konvertieren, doch sollten Sie dies mit Vorsicht angehen. Abbildung 3.23 zeigt den Versuch einer expliziten Umwandlung (1) von einem `SqlString`, der keine Zahl repräsentiert, in einen `SqlInt32`. Das Ergebnis ist eine Laufzeitausnahme. Jeder `System.Data.SqlTypes`-Typ besitzt eine statische `Parse`-Methode, mit der man ebenfalls einen `System.String` in diesen Datentyp umwandeln kann (2). Beide Verfahren zur Umwandlung von Text in einen `System.Data.SqlTypes`-Typ führen zu einer Laufzeitausnahme, wenn die Zeichenfolge nicht geeignet formatiert ist.

```
static void Main(string[] args)
{
   SqlString s = "A123";
① SqlInt32 i1 = (SqlInt32)s;
② SqlInt32 i2 = SqlInt32.Parse(s.Value);
   SqlInt32 i3;
   Int32 i;
③ if (Int32.TryParse(s.Value, out i))
   {
      i3 = i;
   }
}
```

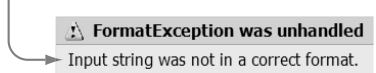


Abbildung 3.23: SqlString konvertieren

Viele numerische CLR-Typen implementieren eine statische Methode `TryParse`, mit der sich Text auf korrekte Formatierung überprüfen lässt, ohne dass bei falschem Format eine Ausnahme ausgelöst wird. Die `SqlTypes`-Typen implementieren keine derartige Methode. Abbildung 3.23 zeigt eine Alternative, wie sich mit der `TryParse`-Methode vom zugrunde liegenden CLR-Typ des `SqlTypes`-Typs der Text prüfen lässt (3), bevor er in einen `System.Data.SqlTypes`-Typ konvertiert wird.

In einigen Fällen verhalten sich die Typen im Namespace `System.Data.SqlTypes` etwas anders als ihre zugrunde liegenden CLR-Datentypen. So können `SqlDouble` und `SqlSingle` keine Werte der Art NaN (Not a Number, keine Zahl) ausdrücken. Dabei handelt es sich um eine zugrunde liegende Byte-Darstellung, die weder eine Zahl noch einen unendlichen Wert verkörpert. Die CLR-Gleitkommatypen kennen

diese speziellen Werte. Versucht man, einen `SqlDouble` oder `SqlSingle` auf einen dieser Werte zu setzen, führt das zu einem Laufzeitfehler.

Abbildung 3.24 zeigt einen `SqlDouble`, der implizite Typumwandlung verwendet, damit ihm ein `System.Double` – d.h. ein `double`-Typ in C# – zugewiesen werden kann (1). Ein `System.Double` (oder `double` in C#) kann Gleitkommawerte für Unendlichkeit (2) und NaN darstellen, während das bei `SqlDouble` und `SqlSingle` nicht möglich ist. Setzt man einen `SqlDouble` oder `SqlSingle` auf `System.Double`, der einen unendlichen Wert (2) oder NaN verkörpert, wird eine Laufzeitausnahme ausgelöst (3).

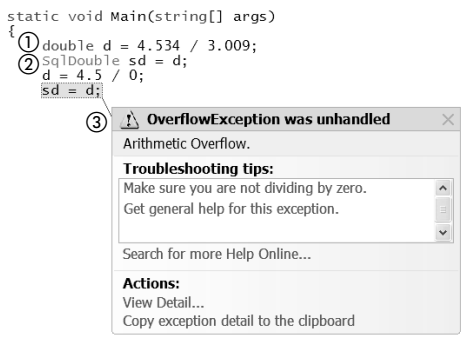


Abbildung 3.24: Überlauf bei SqlTypes

Der zugrunde liegende CLR-Datentyp für `SqlDecimal` umfasst nicht den gleichen numerischen Bereich wie seine korrespondierenden T-SQL-Datentypen `DECIMAL` und `NUMERIC`. Das Beispiel in Abbildung 3.25 weist einer `DECIMAL`-Variablen von T-SQL einen sehr großen Wert zu (1). Versucht man, eine `SqlDecimal`-Variable mit dem gleichen Wert zu initialisieren (2), führt das zu einem Kompilierfehler (3), weil die Zahl außerhalb des Bereichs für den zugrunde liegenden dezimalen Datentyp fällt. Wenn man den großen Wert in eine Variable schreibt und diese zuweist, entsteht eine Laufzeitausnahme.

```
DECLARE @d DECIMAL(38)
SET @d = ① 79228162514264337593543950336
SELECT @d

SqlDecimal sqlDecimal =
   new SqlDecimal( ② 79228162514264337593543950336M);

③ Error     1 Floating-point constant is outside the
range of type 'decimal'
```

Abbildung 3.25: Bereichsbeschränkungen von SqlDecimal

Die meisten CLR-Sprachen definieren spezielle Operatorsymbole für gebräuchliche Operationen: + für Addition, - für Subtraktion usw. Wenn eine CLR-Sprache – zum Beispiel C# – ein Pluszeichen zwischen zwei Werten entdeckt, muss sie eine Methode finden und anwenden, die diese beiden Werte addiert. In der CLR gibt es keine +-Methode und es ist in jedem Fall möglich, dass die Datentypen auf beiden Seiten des Pluszeichens in unterschiedlichen CLR-Sprachen implementiert sind.

Die CLR verwendet übliche Namen für Methoden, die Standardoperationen implementieren. Alle diese Namen beginnen mit dem Präfix `op_`. Zum Beispiel lautet der Name für eine Additionsmethode `op_Addition`. Diese Methoden sind in der MSDN-Dokumentation beschrieben. Suchen Sie im Index einfach nach »op_Addition« – auf der entsprechenden Seite sind alle Methoden aufgeführt.

Es ist recht komfortabel, einfache Operatoren auf `System.Data.SqlTypes`-Datentypen zu verwenden, und viele Operatoren sind für diese Typen implementiert. Abbildung 3.26 zeigt die Signatur für die statische Methode `op_Addition` (1), die die Addition für den Typ `SqlInt32` implementiert. Die Sprache C# implementiert mit dieser Methode den +-Operator (2).

```
① public static op _ addition(SqlInt32 x, SqlInt32 y);

   SqlInt32 i1= 10;
   SqlInt32 i2 = 9;
② SqlInt32 sum = i1 + i2;
```

Abbildung 3.26: Addition mit SqlTypes

Bei diesen Operatoren wird das Ergebnis durch den Datentyp selbst definiert. Aber nicht alle `System.Data.SqlTypes`-Datentypen implementieren sämtliche Operatoren. Zum Beispiel berechnet der `op_addition`-Operator für `SqlInt32` die arithmetische Summe, während derselbe Operator für `SqlString` eine Verkettung ausführt. Tabelle 3.3 listet die Operatormethoden auf, die von den Datentypen im Namespace `System.Data.SqlTypes` implementiert werden. Ein X in der Tabelle gibt an, dass die darüber aufgeführte Methode durch den links vom X stehenden `System.Data.SqlTypes`-Typ implementiert wird. Informieren Sie sich in der MSDN-Dokumentation über die Besonderheiten, die für jeden Operator gelten. In den meisten Fällen jedoch bewirkt der Operator das, was man intuitiv erwartet.

Zum Vergleichen von `SqlBoolean`-Typen gibt es die Operatoren `op_Equality` und `op_Inequality`. Selbst wenn der zugrunde liegende Datentyp `System.Boolean` ist, sind die Ergebnisse dieser Vergleiche nicht dieselben wie bei Vergleichen mit `System.Boolean`.

	op_Addition	op_BitwiseAnd	op_BitwiseOr	op_Division	op_Equality	op_ExclusiveOr	op_False	op_GreaterThan	op_GreaterThanOrEqual	op_Inequality	op_LessThan	op_LessThanOrEqual	op_LogicalNot	op_OnesComplement	op_Modulus	op_Multiply	op_True	op_Subtraction	op_UnaryNegation
SqlBinary	X				X			X	X		X	X							
SqlByte	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	
SqlBytes																			
SqlBoolean		X	X		X	X	X	X	X	X	X	X	X	X			X		
SqlChars																			
SqlDateTime	X				X			X	X	X	X	X						X	
SqlDecimal	X			X	X			X	X	X	X	X				X		X	X
SqlDouble	X			X	X			X	X	X	X	X				X		X	X
SqlGuid					X			X	X	X	X	X							
SqlInt16	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	X
SqlInt32	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	X
SqlInt64	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	X
SqlMoney	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	X
SqlSingle	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	X
SqlString	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	X
SqlBinary	X	X	X	X	X	X		X	X	X	X	X		X	X	X		X	X

Tabelle 3.3: Von SqlTypes-Typen unterstützte CLR-Operationen

Der C#-Code in Abbildung 3.27 zeigt den Einsatz von `SqlBoolean`. Versucht man, das Vergleichsergebnis von zwei `SqlBoolean`-Werten an einen `bool`-Typ zuzuweisen (1), entsteht ein Kompilierfehler, weil das Ergebnis ein `SqlBoolean` ist und keine implizite Umwandlung von `SqlBoolean` nach `System.Boolean` existiert.

```
static void Main(string[] args)
{
   SqlBoolean b1 = SqlBoolean.True;
   SqlBoolean b2 = SqlBoolean.False;
① bool compare1 = b1 == b2;
② bool compare2 = (b1 == b2).Value;
③ bool compare3 = (bool)(b1 == b2);
}
```

	Description
1	Cannot implicitly convert type 'System.Data.SqlTypes.SqlBoolean' to 'bool'. An explicit conversion exists (are you missing a cast?)

Abbildung 3.27: SqlBoolean verwenden

Der zugrunde liegende Typ für das Vergleichsergebnis von zwei `SqlBoolean`-Werten ist ein `System.Boolean`, sodass sich seine `Value`-Eigenschaft einem `System.Boolean` zuweisen lässt (2). Und da das Ergebnis ein `SqlBoolean` ist, kann man es explizit in einen `System.Boolean` umwandeln (3).

CLR-Typen und `SqlTypes`-Typen gehen unterschiedlich mit null um. Das C#-Programm in Abbildung 3.28 zeigt diese Unterschiede. Es vergleicht zunächst zwei Instanzen des `int?`-Typs von C#. Dabei handelt es sich um einen nullfähigen `System.Nullable<System.Int32>`-Typ, der in gewisser Hinsicht den `SqlTypes`-Typen ähnelt, da er `null` sein kann, auch wenn jeder CLR-Typ, der sich auf null setzen lässt, das gezeigte Verhalten an den Tag legt. Beachten Sie, dass ein Vergleich von zwei `int?`-Instanzen auf Gleichheit (1) das Ergebnis `true` liefert, was die Ausgabe des Codes auch dokumentiert.

```
static void Main(string[] args)
{
  int? a = null;
  int? b = null;
①if (a == b) Console.WriteLine("int? ==");
  if (a != b) Console.WriteLine("int? !=");

  SqlInt32 c = SqlInt32.Null;
  SqlInt32 d = SqlInt32.Null;
②if (c == d) Console.WriteLine("SqlInt32 ==");
  if (c != d) Console.WriteLine("SqlInt32 !=");
}
```

```
C:\WINDOWS\system32\cmd.exe
int? equal
Press any key to continue . . .
```

Abbildung 3.28: Vergleichen von null-Werten

Vergleicht man jedoch die beiden `SqlInt32`-Instanzen auf Gleichheit, ist das Ergebnis `false` und der Code gibt nichts aus. Ebenso ist das Ergebnis `false`, wenn zwei `SqlInt32`-Instanzen auf Ungleichheit getestet werden. Das entspricht dem Verhalten von SQL Server 2005, dass ein Vergleich mit mindestens einem

NULL-Wert immer false liefert. Allerdings kann man SQL Server 2005 so konfigurieren, dass sich ein Verhalten wie bei CLR-Vergleichen mit NULL ergibt.

In T-SQL ist ein Datentyp SQL_VARIANT verfügbar. Mit diesem Typ unterstützt man normalerweise Datentypen, die im Microsoft Component Object Model (COM) verwendet werden. Es gibt keinen System.Data.SqlTypes-Datentyp, der dem SQL_VARIANT äquivalent ist. CLR-Methoden stellen einen SQL_VARIANT von T-SQL durch einen System.Object-Datentyp dar. In einer SQL_VARIANT-Variablen kann man die meisten Datentypen speichern.

Abbildung 3.29 zeigt die in C# geschriebene Funktion ObjectType, die einen einzelnen Eingabeparameter übernimmt und den Namen des als SqlString übergebenen Objekttyps zurückgibt (1). Diese Funktion wird von T-SQL aus aufgerufen. Im ersten Fall (2) wird die Zahl 1 an ObjectType übergeben und die Funktion liefert als Ergebnis die Zeichenfolge »System.Data.SqlTypes.SqlInt32« – d.h. den Typ der übergebenen Zahl 1.

```
public static SqlString ObjectType(object obj)
{
① return new SqlString(obj.GetType().ToString());
}
```

```
DECLARE @f FLOAT
SET @f = 7
DECLARE @i INT
DECLARE @v SQL _ VARIANT
SET @v = 'abc'
SELECT                  ②
dbo.ObjectType(1) AS Col1,
                        ③
dbo.ObjectType(@i) AS Col2
                        ④
dbo.ObjectType(@v) AS Col3
```

Col1
System.Data.SqlTypes.SqlInt32

Col2
System.DBNull

Col3
System.Data.SqlTypes.SqlString

Abbildung 3.29: System.Object verwenden

Im zweiten Fall (3) wird eine nicht initialisierte T-SQL-Variable vom Typ INT übergeben. Die Funktion ObjectType liefert hier das Ergebnis »System.DBNull«. Dieser Datentyp dient in der CLR dazu, eine Datenbank-NULL darzustellen. Schließlich wird eine SQL_VARIANT-Variable als Zeichenfolge initialisiert und übergeben (4). Beachten Sie, dass ObjectType hier nicht erkennen kann, dass es sich um einen SQL_VARIANT handelt. Die Funktion gibt »System.Data.SqlTypes.SqlString« zurück – d.h. den Typ des Werts, mit dem der SQL_VARIANT initialisiert wurde.

Wenn man mit System.Object als Eingabeparameter arbeitet, muss man im Allgemeinen den übergebenen Typ bestimmen. Die in C# implementierte Beispielfunktion von Abbildung 3.30 verarbeitet einen Eingabeparameter des Typs System.Object und gibt einen SqlString zurück. Die Art der Verarbeitung hängt

vom übergebenen Objekttyp ab. Stellen Sie sich das als überladene Funktion vor. Wenn Sie einen `SqlInt32` übergeben, führt sie etwas anderes aus, als wenn ein `SqlBoolean` übergeben wird.

```
public static SqlString ProcessObject(object obj)
{
   if (obj is SqlInt32)
   {
   ① return (((SqlInt32)obj).Value + 1).ToString();
   }
   if (obj is SqlBoolean)
   {
      return (!((SqlBoolean)obj)).ToString();
   }
   if (obj is SqlString)
   {
      return ((SqlString)obj).Value.ToUpper();
   }
② return SqlString.Null;
}
```

```
DECLARE @b bit
SET @b = 1                          ③
select dbo.ProcessObject(1) AS Col1,
dbo.ProcessObject(@b) AS Col2,
dbo.ProcessObject('abc') AS Col3
```

Col1
2

Col2
False

Col3
ABC

Abbildung 3.30: System.Object verarbeiten

Wenn Sie der Funktion einen `SqlInt32` übergeben, inkrementiert sie seinen Wert um eins und gibt das Ergebnis als Zeichenfolge zurück (1). Ist der Eingabeparameter vom Typ `SqlBoolean`, negiert die Funktion den Wert und gibt ihn als Zeichenfolge zurück, usw. Handelt es sich beim Eingabeparameter um einen nicht unterstützten Typ, gibt die Funktion null zurück (2).

Ruft man diese Funktion von T-SQL aus auf, wie es Abbildung 3.30 ebenfalls zeigt, und übergibt den numerischen Wert 1, wird 2 zurückgegeben (3). Und wenn man die Variable `b` vom Typ `BIT` mit 1 initialisiert übergibt, lautet der Rückgabewert `False`.

Der Datentyp `SqlString` verdient eine besondere Betrachtung. Die CLR unterstützt nur Unicode-Zeichenfolgen. Nimmt man also eine CLR-Methode in den Katalog von SQL Server 2005 auf, muss man den Datentyp `NCHAR` oder `NVARCHAR` für Zeichenfolgenparameter und Rückgabewerte verwenden. Die Typen `CHAR` und `VARCHAR` sind nicht möglich. Der Typ `NVARCHAR(MAX)` kommt in der CLR dem Datentyp `System.String` am nächsten, in C# wird oft die Kurzform `string` verwendet.

Bei `NVARCHAR` mit Größeneinschränkung – zum Beispiel `NVARCHAR(10)` als Rückgabewert oder Ausgabeparameter – löst die CLR eine Ausnahme aus, wenn

die betreffende Methode versucht, eine Zeichenfolge mit mehr Zeichen zurückzugeben, als es die Obergrenze von `NVARCHAR` angibt. Bei Rückgabewerten oder Ausgabeparametern, die mit `NCHAR` spezifiziert sind, schneidet die CLR die Ergebnisse ab und löst keine Ausnahme aus.

Beachten Sie auch, dass eine von Visual Studio 2005 bereitgestellte Funktion oder gespeicherte Prozedur den Typ `NVARCHAR(4000)` für Zeichenfolgen verwendet.

Zusammenfassend lässt sich zu `SqlTypes` sagen, dass es sich bei einem `SqlTypes`-Typ um einen Typ handelt, der einen Nullwert in ähnlicher Weise darstellen kann wie ein Nullwert in einer Datenbank dargestellt wird, sich im Übrigen aber eher wie sein zugrunde liegender CLR-Typ verhält. Einen `SqlTypes`-Typ verwendet man vor allem, wenn man das Verhalten eines SQL Server-Typs nachbilden möchte. Ein `SqlTypes`-Typ besitzt eine `Value`-Eigenschaft, die den Wert in Form des zugrunde liegenden CLR-Typs speichert. Man kann einen `SqlTypes`-Typ in einen anderen `SqlTypes`-Typ und in einen CLR-Typ konvertieren und bestimmte CLR-Typen lassen sich in einen `SqlTypes`-Typ umwandeln.

3.5 Parameter und Rückgabewert

Es empfiehlt sich, für Parameter und den Rückgabewert von CLR-Methoden, mit denen man gespeicherte Prozeduren und Funktionen in T-SQL implementiert, die Typen aus dem Namespace `System.Data.SqlTypes` zu verwenden. Neben diesen Typen kann man alle zugrunde liegenden CLR-Typen, die in der entsprechenden Spalte von Tabelle 3.1 aufgeführt sind, zusammen mit `System.Object` aus dem CLR-Typsystem und jedem benutzerdefinierten Typ einsetzen. Kapitel 5 befasst sich ausführlich mit benutzerdefinierten Typen.

Auf weitere Einschränkungen zu Parametern und Rückgabewerten, die spezifisch für benutzerdefinierte Funktionen und gespeicherte Prozeduren sind, gehen die Abschnitte »Benutzerdefinierte Funktionen« und »Gespeicherte Prozeduren« später in diesem Kapitel ein.

3.6 Benutzerdefinierte Funktionen

SQL Server 2005 unterstützt sowohl benutzerdefinierte Skalarfunktionen als auch benutzerdefinierte Tabellenwertfunktionen, die in der CLR implementiert sind. Benutzerdefinierte Funktionen in T-SQL besitzen einige Eigenschaften, mit denen SQL Server 2005 ihre Verwendung in verschiedenen Aufgaben festlegt. Zwei wichtige Eigenschaften sind `IsDeterministic` und `IsPrecise`.

`IsDeterministic` bedeutet, dass das Ergebnis für eine bestimmte Eingabe immer gleich ist. Deterministisch ist zum Beispiel eine Funktion, die zwei Zahlen addiert und ihre Summe zurückgibt; 1 + 2 ist immer gleich 3. Als `IsPrecise` gelten Funktionen, die weder für Parameter oder Rückgabetypen noch für interne Berechnungen die Datentypen `REAL` oder `FLOAT` verwenden. Es ginge über den Rahmen dieses Buchs hinaus, zu diskutieren, warum diese Datentypen unpräzise sind. Jedenfalls hat es damit zu tun, wie Gleitkommaberechnungen über die Hardware implementiert werden.

SQL Server 2005 kann eine auf T-SQL basierende benutzerdefinierte Funktion analysieren und ermitteln, ob sie deterministisch und/oder genau ist. Abbildung 3.31 zeigt eine T-SQL-Funktion namens `Same`, die eine ihr übergebene Zeichenfolge zurückgibt (1). Diese Funktion ist deterministisch, weil der Rückgabewert für eine bestimmte Eingabe immer gleich ist.

Die zweite T-SQL-Funktion in Abbildung 3.31, `AddUser` (2), gibt ebenfalls die ihr übergebene Zeichenfolge zurück, fügt aber an diese den Namen des aktuellen Benutzers an. Diese Funktion ist nicht deterministisch, weil ihr Rückgabewert vom aktuellen Benutzer abhängt.

```
   CREATE FUNCTION Same(@s CHAR(50)) RETURNS CHAR(50)
   WITH SCHEMABINDING AS BEGIN
①  RETURN @s
   END
   CREATE FUNCTION AddUser(@s CHAR(50)) RETURNS CHAR(50)
   WITH SCHEMABINDING AS BEGIN
②  RETURN @s + USER
   END
③  Select routine_name, is_deterministic
   from INFORMATION_SCHEMA.ROUTINES
   where routine_name = 'Same'
   OR routine_name = 'AddUser'
```

④

Results | Messages

	routine_name	is_deterministic
1	Same	YES
2	AddUser	NO

Abbildung 3.31: Berechnungseigenschaften von T-SQL-Funktionen

SQL Server 2005 implementiert die ANSI-Sicht `INFORMATION_SCHEMA.ROUTINES`. Damit lässt sich feststellen, wie SQL Server 2005 diese Funktionen in Bezug auf `IsDeterminstic` kategorisiert (3). Wie sich zeigt, ist die Funktion `Same` tatsächlich deterministisch, die Routine `AddUser` dagegen nicht (4). SQL Server 2005 hat den Code dieser Funktionen analysiert und ermittelt, dass die Funktion `Same` ständig den gleichen Wert liefert, während die Funktion `AddUser` das Schlüsselwort `USER` verwendet und somit bei jedem Aufruf ein anderes Ergebnis zurückgeben kann.

Den Code einer benutzerdefinierten Funktion, die in einer CLR-Sprache implementiert ist, analysiert SQL Server 2005 nicht und verlässt sich stattdessen auf Informationen, die im Attribut `Microsoft.SqlServer.Server.SqlFunction` übergeben werden. Mit diesem Attribut kann man eine Methodendefinition optional versehen. Abbildung 3.32 zeigt Funktionen, die in der CLR implementiert sind und den Funktionen von Abbildung 3.31 ähneln. Die erste CLR-Funktion in Abbildung 3.32, `SameCLR`, identifiziert sich selbst mithilfe des `SqlFunction`-Attributs als deterministische Funktion. Analog zeigt die zweite CLR-Funktion, `AddUserCLR`, mit dem `SqlFunction`-Attribut an, dass sie keine deterministische Funktion ist (2). Mit der `INFORMATION_SCHEMA.ROUTINE`-Sicht wird wie in Abbildung 3.31 festgestellt, wie SQL Server 2005 diese Funktionen klassifiziert (3). Die Ausgabe (4) spiegelt die Informationen des korrespondierenden `SqlFunction`-Attributs wider.

①
```
[SqlFunction(IsDeterministic = true)]
public static SqlString SameCLR(SqlString s)
{
   return s;
}
```
②
```
[SqlFunction(IsDeterministic = false)]
public static SqlString AddUserCLR(SqlString s)
{
   return s + System.Environment.UserName;
}
```

③
```
Select routine_name, is_deterministic
from INFORMATION_SCHEMA.ROUTINES
where routine_name = 'SameCLR'
OR routine_name = 'AddUserCLR'
```

④ Results | Messages

	routine_name	is_deterministic
1	AddUserCLR	NO
2	SameCLR	YES

Abbildung 3.32: Berechnungseigenschaften von CLR-Funktionen

T-SQL- und CLR-Funktionen gehen nicht genau gleich mit `IsDeterministic` und `IsPrecise` um. Eine T-SQL-Funktion wird immer als nicht deterministisch oder unpräzise klassifiziert, wenn sie ohne die Funktionsoption `WITH SCHEMABINDING` erstellt wurde, und zwar unabhängig vom Inhalt des Funktionskörpers. Da es nicht zulässig ist, die Anweisung `CREATE FUNCTION` für eine CLR-Funktion mit der Funktionsoption `WITH SCHEMABINDING` zu verwenden, wird eine derartige Funktion immer entsprechend den Eigenschaften `IsDeterministic` und `IsPrecise` des `SqlFunction`-Attributs klassifiziert.

Es ist wichtig, die korrekten `IsDeterministic`- und `IsPrecise`-Eigenschaften des `SqlFunction`-Attributs zu spezifizieren. Gibt man eine dieser Eigenschaften nicht korrekt an – indem man etwa behauptet, dass eine Funktion `IsPrecise` ist, obwohl sie Gleitkommaoperationen ausführt –, kann das zu subtilen, schwer auffindbaren Fehlern in den Daten führen.

Das `SqlFunction`-Attribut besitzt eine Reihe von Eigenschaften, die in Tabelle 3.4 zusammengefasst sind. Einige davon werden von SQL Server 2005 verwendet und einige von Visual Studio 2005, wenn es die Funktion in einer Instanz von SQL Server 2005 bereitstellt. Anhang C beschäftigt sich mit Visual Studio 2005-Projekten, die eine CLR-Erweiterung direkt in SQL Server 2005 bereitstellen können.

SqlFunction-Eigenschaft	Verwendung	Standardwert
`DataAccess`	greift auf Benutzerobjekte zu	`None`
`IsDeterministic`	spezifiziert die Eigenschaft `IsDeterministic`	`False`
`IsPrecise`	spezifiziert die Eigenschaft `IsPrecise` – d.h., ob die Funktion Gleitkommaarithmetik verwendet	`False`
`FillRowMethodName`	Spaltendecoder für Tabellenwertfunktion (TVF)	`Null`
`Name`	Name für Bereitstellung (nur Visual Studio)	`Null`
`TableDefinition`	Tabellendefinition für Tabellenwertfunktion (nur Visual Studio)	`Null`
`SystemDataAccess`	greift auf Systemobjekte zu	`None`

Tabelle 3.4: SqlFunction-Eigenschaften

Die Eigenschaften `DataAccess` und `SystemDataAccess` des `SqlFunction`-Attributs besitzen zwei mögliche Aufzählungswerte: `None` und `Read`. Dabei bedeutet `DataAccess = None`, dass die Funktion auf keinerlei Benutzerobjekte in SQL Server zugreift, und `SystemDataAccess = None` heißt, dass die Funktion auf keine Systemobjekte in SQL Server zugreift. Wenn man die `Read`-Enumeration für diese Eigenschaften verwendet, ist der Zugriff auf die jeweiligen Objekte möglich. Dagegen verhindert `None` den Zugriff auf die betreffenden Objekte. Kapitel 4 befasst sich ausführlich mit dem Zugriff auf SQL Server aus einer CLR-basierten benutzerdefinierten Funktion. Beachten Sie, dass eine CLR-benutzerdefinierte Funktion in keinem Fall den Zustand einer Datenbank ändern kann – zum Beispiel eine `INSERT`-Anweisung ausführen –, genau wie es für eine benutzerdefinierte T-SQL-Funktion gilt.

Auf die `Name`-Eigenschaft greift Visual Studio 2005 zurück, wenn die Methode bereitgestellt wird. Der Wert der `Name`-Eigenschaft dient dazu, auf die Funktion in T-SQL zu verweisen, falls er nicht null ist.

Die Eigenschaften `TableDefinition` und `FillRowMethodName` werden von benutzerdefinierten Tabellenwertfunktionen verwendet und später in diesem Kapitel besprochen.

Es gibt zwei Arten von benutzerdefinierten Funktionen: Skalarfunktionen und Tabellenwertfunktionen. Eine benutzerdefinierte Skalarfunktion gibt einen Skalarwert zurück. Alle Beispiele in diesem Kapitel haben bislang Skalarfunktionen verwendet, wie zum Beispiel die Funktion `ValidateFullNull` in Abbildung 3.19. Die allgemeine Form aller CLR-basierten benutzerdefinierten Funktionen verlangt, dass die CLR-Methode eine statische öffentliche Methode einer öffentlichen Klasse auf oberster Ebene ist. Methodennamen dürfen nicht überladen werden. Das `SqlFunction`-Attribut ist für benutzerdefinierte Skalarfunktionen, die weder deterministisch noch genau sind und die nicht auf Datenbankobjekte zugreifen, optional. Das `SqlFunction`-Attribut ist jedoch erforderlich, wenn die Funktion durch Visual Studio 2005 bereitgestellt wird.

Abbildung 3.33 zeigt in einer C#-Klasse CLR-Methoden, die sich als benutzerdefinierte Skalarfunktionen verwenden lassen. Die öffentliche Klasse `Math` (1) enthält zwei öffentliche Methoden: `AddInt32` (3) und `AddSingle` (4). Die Methode `AddInt32` ist mit einem `SqlFunction`-Attribut versehen (2), das sie als präzise und deterministisch markiert. Fehlt dieses Attribut, nimmt SQL Server 2005 an, dass diese Funktion weder präzise noch deterministisch ist. Das würde ihre Verwendung einschränken, sie also zum Beispiel nicht für Indexe zulassen.

```
① public class Math
   {
   ② [SqlFunction(IsPrecise=true,
      IsDeterministic=true)
   ③ static public SqlInt32 AddInt32
         (SqlInt32 a, SqlInt32 b)
      {
         return a + b;
      }
   ④ static public SqlSingle AddSingle
         (SqlSingle a, SqlSingle b)
      {
         return a + b +
            System.Environment.UserName.Length;
      }
   }
```

Abbildung 3.33: SqlFunction verwenden

Die Funktion `AddSingle` in Abbildung 3.33 ist tatsächlich weder deterministisch noch genau und greift auch nicht auf Datenbankobjekte zu. Deshalb ist für sie auch kein `SqlFunction`-Attribut erforderlich, um korrekt in SQL Server 2005 geladen zu werden. Es gehört aber zum guten Programmierstil, immer das `SqlFunction`-Attribut anzugeben, um die Absicht deutlich zu machen.

Die CLR unterstützt das Verschachteln von Klassendefinitionen, wie in Abbildung 3.34 gezeigt. Die Klasse `Math` (1) enthält die Definition der Klasse `Numeric` (2). Beide Klassen sind öffentlich. Die Klasse `Numeric` enthält eine statische öffentliche Methode `AddInts` (3). Versucht man, die `AddInts`-Methode als T-SQL-Funktion zu laden, führt das zu einer Fehlermeldung (4).

```
① public class Math
  {
  ② public class Numeric
     {
        [SqlFunction(IsPrecise = true,
           IsDeterministic = true)]
     ③ static public SqlSingle AddInts(
           SqlSingle a, SqlSingle b)
        {
           return a + b;
        }
     }
  }
```

④

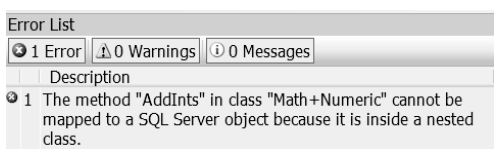


Abbildung 3.34: Verschachtelte Klassen

SQL Server 2005 unterstützt kein Überladen von Methoden, auch wenn nur eine dieser Methoden als T-SQL-Funktion verwendet wird. Abbildung 3.35 zeigt zwei C#-Methoden einer Klasse, die grammatikalisch korrekt ist. Die Methoden sind überladen und haben beide den gleichen Namen: `Add`. Eine Methode übernimmt `SqlInt32`-Parameter (1) und die andere `SqlInt16`-Parameter (2). Will man eine dieser Methoden laden, erscheint eine Fehlermeldung, dass überladene Methoden nicht unterstützt werden (3), selbst wenn die andere Methode nicht als T-SQL-Funktion geladen wird.

Als Datentypen der Parameter und Rückgabewerte von benutzerdefinierten Skalarfunktionen sind die Typen erlaubt, die der Abschnitt »Parameter und Rückgabewert« weiter vorn in diesem Kapitel beschrieben hat. Alle Parameter dürfen nur als Eingabeparameter verwendet werden, wie es bei benutzerdefinierten T-SQL-Funktionen der Fall ist. Dies ist auch der Standard für Parameter in C# und Visual Basic. Das heißt, dass man die Parameter in C# als `IN` oder überhaupt

nicht markieren muss und sie nicht als `OUT` oder `REF` gekennzeichnet sein dürfen. In Visual Basic darf man Parameter entsprechend nicht als `BY REF` deklarieren.

```
   [SqlFunction(IsPrecise = true,
      IsDeterministic = true)]
   static public SqlInt32
①     Add(SqlInt32 a, SqlInt32 b)
   {
      return a + b;
   }
   static public SqlInt32
②     Add(SqlInt16 a, SqlInt16 b)
   {
      return a + b;
   }
```

③

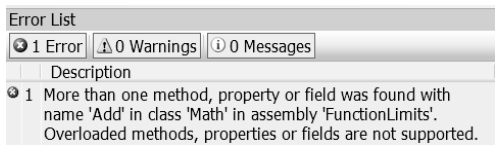


Abbildung 3.35: Überladene Methoden

Abbildung 3.36 zeigt eine C#-Methode mit einem `ref`-Parameter `a` (1). Über diesen `ref`-Parameter gibt die Methode einen zusätzlichen Wert an den Aufrufer zurück (2). In T-SQL lässt sich dieses Feature nicht nutzen. Man müsste den entsprechenden Parameter in der T-SQL-Funktionsdefinition mit dem Schlüsselwort `OUTPUT` deklarieren (3), was aber für Parameter von benutzerdefinierten Funktionen in T-SQL nicht zulässig ist.

```
[SqlFunction(IsPrecise = true,
   IsDeterministic = true)]
 static public SqlInt32 AddInt16(
① ref SqlInt32 a, SqlInt32 b)
 {
② a = a + 1;
   return a + b;
 }

CREATE FUNCTION [AddInt16]
(                              illegal
③ @a int OUTPUT,
   @b int
)
RETURNS [int]
AS EXTERNAL NAME [Math].[AddInt16]
```

Abbildung 3.36: Fehler bei Ausgabeparametern in Funktionen

3.7 Tabellenwertfunktionen

Eine Tabellenwertfunktion (Table-Valued Function, TVF) wird in der CLR durch eine Methode implementiert, die eine `IEnumerable`-Schnittstelle zusammen mit anderen Hilfskonstrukten zurückgibt. In T-SQL gibt eine Tabellenwertfunktion Zeilen an SQL Server 2005 zurück. In der CLR dient die `IEnumerable`-Schnittstelle dazu, eine `IEnumerator`-Schnittstelle abzurufen. Für SQL Server 2005 sind in der `IEnumerator`-Schnittstelle vor allem die Methode `MoveNext` und die Eigenschaft `Current` interessant. Die Methode `MoveNext` gibt `true` zurück, sofern ein weiteres Objekt existiert, und wechselt zu diesem Objekt. Gibt es keine Objekte mehr, lautet der Rückgabewert `false`. Die Eigenschaft `Current` gibt das aktuelle Objekt zurück. Enumeratoren zeigen anfangs immer vor das erste Objekt, sodass Sie zunächst `MoveNext` aufrufen müssen.

Führt SQL Server 2005 eine durch die CLR implementierte Tabellenwertfunktion aus, wird `MoveNext` aufgerufen und mit `Current` das aktuelle Objekt abgefragt, bis `MoveNext` das Ergebnis `false` liefert. Dabei taucht allerdings ein Problem auf: SQL Server 2005 benötigt die Spaltenmenge für die Zeile und kein Objekt. Die Implementierung der Tabellenwertfunktion muss SQL Server 2005 auch eine Hilfsmethode bereitstellen, mit der sich das von `Current` zurückgegebene Objekt in die von SQL Server 2005 benötigten Spaltenwerte konvertieren lässt. Diese Hilfsmethode heißt `FillRowMethod`; ihr Name wird in der `FillRowMethodName`-Eigenschaft des `SqlFunction`-Attributs an SQL Server 2005 übergeben – genauso wie SQL Server 2005 über die Eigenschaft `IsDeterministic` ermitteln kann, ob eine Funktion deterministisch ist.

Es ist nicht schwer, `IEnumerable` zu implementieren, aber recht ermüdend, da jede Menge Standardcode für die Verwaltung notwendig ist, damit das Ganze funktioniert. C# implementiert den gesamten Standardcode für Sie, wie Sie noch später in diesem Kapitel sehen werden.

Methoden, die die Schnittstelle `IEnumerable` zurückgeben, werden von einigen CLR-Sprachkonstrukten verwendet, um eine Enumeration von Werten zu durchlaufen – beispielsweise `foreach` in C# oder `For Each` in Visual Basic. SQL Server 2005 verwendet ein ähnliches Konstrukt, um die Methode wie einen schreibgeschützten Vorwärts-Cursor zu behandeln.

Abbildung 3.37 zeigt mit `EnumNumbers1to4` ein C#-Beispiel für eine Enumerator-Methode (1) und ihre typische Verwendung. Der Methodenkörper besteht aus einer Schleife, die von 1 bis 4 zählt. Die `yield return`-Anweisung wird bei jedem Schleifendurchlauf ausgeführt und gibt den von der Schleife erzeugten Wert zurück.

```
class SimpleEnum {
① static public IEnumerable EnumNumbers1to4()
   {
      for (int index = 1; index < 5; index++)
      {
      ② yield return index;
      }
   }
}
class Program {
   static void Main(string[] args)
   {
   ③ foreach(int i in SimpleEnum.EnumNumbers1to4())
      {
         Console.WriteLine(i);
      }
   }
}
```

④ Visual Studio

```
C>SimpleEnum.exe
1 2 3 4
C>
```

Abbildung 3.37: Einfache Enumeration

Durch die Anweisung `yield return` vereinfacht sich die Implementierung von `IEnumerable`. Setzt man ein `yield` vor ein `return`, werden der aktuelle Zustand – insbesondere alle lokalen Variablen – und die Position der `yield`-Anweisung für die betreffende Methode gespeichert, die Methode kehrt aber trotzdem zurück. Beim nächsten Aufruf der Methode setzt die Codeausführung mit der Zeile unmittelbar nach der `yield`-Anweisung fort und alle lokalen Variablen erhalten denselben Zustand wie im vorherigen Aufruf.

Dieses `yield` lässt sich in einer C#-`foreach`-Anweisung verwenden, wie es im Hauptprogramm von Abbildung 3.37 zu sehen ist (3). Der Körper der `foreach`-Anweisung wird einmal für jede `yield return`-Anweisung durchlaufen, die in der Methode `EnumNumbers1to4` ausgeführt wird. Als Ergebnis gibt das Programm die Zahlen 1 bis 4 auf dem Bildschirm aus (4).

Es ist nicht offensichtlich, welche Funktion aufgerufen wird, damit `yield` den lokalen Zustand speichert. Denn hinter den Kulissen hat C# eine Menge Standardcode geschrieben, um `IEnumerable` zu implementieren. Praktisch wird bei jedem Durchlauf der `foreach`-Schleife (3) in `Main` die `yield`-Anweisung in `EnumNumbers1to4` (2) aufgerufen.

Sehen wir uns nun eine Tabellenwertfunktion an, mit der sich ein Satz von logarithmischen Bereichswerten erzeugen lässt. Das Ziel besteht darin, einen Gesamtbereich von Werten in einen Satz von logarithmischen Bereichen zu zerlegen, sodass der erste logarithmische Bereich doppelt so groß wie der zweite ist usw.

Später verwenden wir diese Tabelle in einer Abfrage, um Werte entsprechend dem logarithmischen Bereich, in den sie fallen, zu klassifizieren.

Abbildung 3.38 verdeutlicht, wie ein Gesamtbereich von 10 in fünf logarithmische Bereiche zerlegt wird. Bereich 1 läuft von 0 bis 5,161. Bereich 2 überstreicht 7,742 – 5,161 = 2,581 oder etwa die Hälfte von Bereich 1. Analog geht es für die restlichen Bereiche weiter.

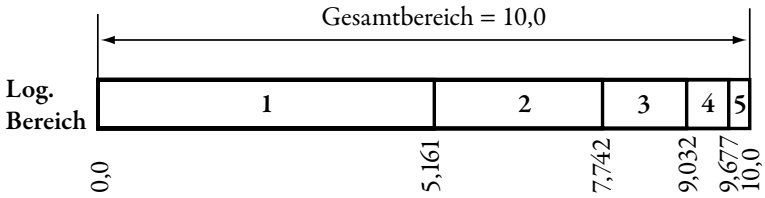


Abbildung 3.38: Logarithmische Bereiche

Wir erstellen eine Tabellenwertfunktion `LogRange`, die für einen Kleinstwert, einen Größtwert und eine Anzahl von logarithmischen Bereichen eine Tabelle erzeugt, in der Kleinst- und Größtwert für jeden Bereich angegeben sind. Die Tabelle in Abbildung 3.39 zeigt, was wir für den Kleinstwert 0, den Größtwert 10 und 5 logarithmische Bereiche erzeugen wollen. Der Wert von `MinValue` für die Bereichsnummer `RangeNumber` 1 ist die kleinste Zahl und der Wert `MaxValue` für `RangeNumber` 5 ist die größte Zahl. Außerdem wird `MaxValue` in der nächsten (außer in der letzten) Zeile als `MinValue` wiederholt.

MinValue	MaxValue	RangeNumber
-1.79769313486232E+308	5.16129032258065	1
5.16129032258065	7.74193548387097	2
7.74193548387097	9.03225806451613	3
9.03225806451613	9.67741935483871	4
9.67741935483871	1.79769313486232E+308	5

Abbildung 3.39: Tabelle mit logarithmischen Bereichen

Mit einer Tabelle wie in Abbildung 3.39 können wir leicht Werte logarithmisch von 1 bis 10 klassifizieren. Abbildung 3.42 zeigt die Implementierung der Funktion `LogRange`, die zu der in Abbildung 3.39 dargestellten Tabelle führt.

Für eine Tabellenwertfunktion ist etwas mehr Aufwand erforderlich als für eine Skalarfunktion, weil die `yield return`-Anweisung nur ein einzelnes Objekt zurückgibt. Abbildung 3.40 zeigt die C#-Struktur `Range` (1), die die drei Spalten der Tabellenwertfunktion aufnehmen kann. Jede Spalte wird durch ein Feld in der Struktur `Range` dargestellt (2). Für jede Spalte in der Tabelle muss ein Feld vorhanden sein. Der Konstruktor (3) für die Klasse `Range` initialisiert diese Felder.

```
① struct Range
   {
   ② public readonly double minValue;
      public readonly double maxValue;
      public readonly int rangeNumber;
   ③ public Range(double minValue,
         double maxValue, int rangeNumber)
      {
         this.minValue = minValue;
         this.maxValue = maxValue;
         this.rangeNumber = rangeNumber;
      }
   }
```

Abbildung 3.40: Die Klasse Range

SQL Server 2005 erhält für jede Zeile in der Tabelle eine Instanz der `Range`-Struktur. Es ist eine `FillRowMethod` erforderlich, um die Spaltenwerte aus einer Instanz der `Range`-Struktur zu extrahieren. Die `FillRowMethod` übernimmt ein Objekt als Eingabe und gibt einen Ausgabeparameter für jede Spalte in der Tabelle zurück. Abbildung 3.41 zeigt die Methode `GetRow` (1), die diesen Anforderungen entspricht. SQL Server 2005 ruft die Methode `GetRow` einmal für jede Zeile auf und übergibt ihr das vom Enumerator abgerufene Objekt. Die Ausgabeparameter (2) geben die Spaltenwerte zurück.

```
public class RangeFunctions
{
① static public void GetRow(Object obj,
②     out SqlDouble minValue,
      out SqlDouble maxValue,
      out SqlInt32 rangeNumber)
   {
   ③ Range r = (Range)obj;
      minValue = r.minValue;
      maxValue=r.maxValue;
      rangeNumber = r.rangeNumber;
    }
}
```

Abbildung 3.41: Die Methode GetRow

Das übergebene Objekt ist immer eine Instanz einer `Range`-Struktur (3) und wie Sie gleich sehen werden, wird dies auch durch die Implementierung garantiert. Diese Methode muss ein Member derselben Klasse sein, die die Methode für die Tabellenwertfunktion implementiert – in diesem Beispiel ein Member der Klasse `RangeFunctions`.

Wir besitzen nun die erforderlichen Grundbausteine, um die `LogRange`-Tabellenwertfunktion zu implementieren – eine Methode, die die Schnittstelle `IEnumerable` implementiert. Bei jedem Aufruf von `MoveNext` gibt `Current` eine Instanz der `Range`-Struktur zurück. Die Methode `GetRow` extrahiert die Spaltenwerte aus der `Range`-Struktur.

Abbildung 3.42 zeigt die Implementierung von `LogRange`. Das `SqlFunction`-Attribut, das ihr vorangestellt wurde, besteht aus zwei Teilen. Die `FillRowMethodName`-Eigenschaft (1) des `SqlFunction`-Attributs enthält den Namen der `FillRowMethod`-Methode, die SQL Server verwenden muss, um die Spalteninformation aus einer Instanz der `Range`-Struktur herauszuziehen. Die Eigenschaft `TableDefinition` (2) enthält den Namen und den Datentyp für jede Tabellenspalte, die von der Tabellenwertfunktion produziert wird. Diese Informationen weisen das gleiche Format auf, wie es in einer T-SQL-Tabellenwertfunktion eingesetzt werden würde. Wird die Funktion `LogRange` katalogisiert, erhält SQL Server 2005 den Namen der `FillRowMethod` vom `SqlFunction`-Attribut. SQL Server verwendet allerdings die `TableDefinition`-Eigenschaft nicht – nur Visual Studio 2005 greift darauf zurück, wenn es die Funktion bereitstellt.

```
public class RangeFunctions
{
[Microsoft.SqlServer.Server.SqlFunction
① (FillRowMethodName="GetRow",
② TableDefinition =
  "MinValue FLOAT, MaxValue FLOAT, RangeNumber INT"
  )]
public static IEnumerable LogRange(
  SqlDouble minValue,
  SqlDouble maxValue,
  SqlInt32 count)
  {
    double factor = (Math.Pow(2.0, count.Value) - 1)
      /Math.Pow(2.0, count.Value - 1.0);
    double lastTop = double.MinValue;
    double thisTop = minValue.Value + range/2;
  ③ yield return new Range(lastTop, thisTop, 1);
    lastTop = thisTop;
    range /= 2;
    for (int index = 1; index < (count-1); index++)
    {
      thisTop = lastTop + range / 2.0;
      yield return
      ④ new Range(lastTop,thisTop, index + 1);
      lastTop = thisTop;
      range /= 2.0;
    }
  ⑤ yield return new
      Range(lastTop, double.MaxValue, count.Value);
  }
}
```

Abbildung 3.42: CLR-Tabellenwertfunktion

Im Körper der `LogRange`-Methode stehen als Erstes einige arithmetische Operationen, um die logarithmischen Bereiche zu berechnen. Ausgegeben wird die erste Zeile der Tabelle mit einer `yield return`-Anweisung für eine `Range`-Struktur, die die kleinste Zahl, den größten Wert für den ersten logarithmischen Bereich und die Nummer des Bereichs enthält.

Die `for`-Schleife in Abbildung 3.42 gibt mit einer `yield return`-Anweisung (4) alle darauf folgenden Zeilen mit Ausnahme der letzten aus. Schließlich wird die letzte Zeile mit dem kleinsten Wert für den letzten logarithmischen Bereich und der größten Zahl für seinen größten Wert ausgegeben (5).

Listing 3.4 zeigt das T-SQL-Skript, um die `LogRange`-Funktion zu katalogisieren. Wie bei jeder CLR-Funktion müssen die als Parameter im T-SQL-Skript verwendeten Typen mit denen in der CLR-Implementierung übereinstimmen. Außerdem ist in einer Tabellenwertfunktion die Anweisung `RETURNS TABLE` erforderlich. Eine CLR-Tabellenwertfunktion muss die Namen und Typen der Tabellenspalten spezifizieren. Falls Visual Studio 2005 diese Funktion bereitstellt, holt es die Spezifikation der Tabellenspalten aus der `TableDefinition`-Eigenschaft des `SqlFunction`-Attributs.

```
CREATE ASSEMBLY [RangeTable]
FROM 'C:\assemblies\Range.dll'
GO

CREATE FUNCTION [LogRange]
(
  @minValue float,
  @maxValue float,
  @count int
)
RETURNS TABLE(
MinValue FLOAT, MaxValue FLOAT, RangeNumber INT
) AS
EXTERNAL NAME [RangeTable].[RangeFunctions].[LogRange]
```

Listing 3.4: Tabellenwertfunktionen in den Katalog aufnehmen

Abbildung 3.43 zeigt die Verwendung der `LogRange`-Funktion (1) und die von ihr produzierte Tabelle. Die erste Zeile der Tabelle (2) wird durch die erste `yield return`-Anweisung in der `LogRange`-Methode erzeugt. Die mittleren Zeilen (3) kommen von `yield return` in der `for`-Schleife der `LogRange`-Methode. Schließlich stammt die letzte Zeile (4) von der letzten `yield return`-Anweisung der `LogRange`-Methode.

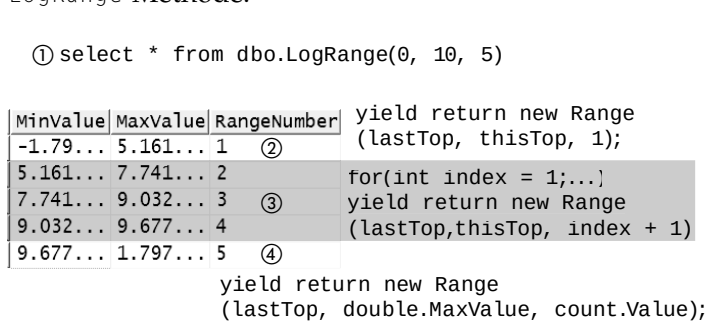


Abbildung 3.43: LogRange verwenden

Nachdem die `LogRange`-Funktion katalogisiert ist, verhält sie sich wie jede andere Tabellenwertfunktion in SQL Server 2005. Zum Beispiel kann man sie in einem `SELECT`-Ausdruck mit anderen Tabellen verknüpfen.

Abbildung 3.44 fasst zusammen, was man für eine Tabellenwertfunktion braucht:

- eine Klasse, hier `Range` (1), um die Spaltenwerte zu kapseln, die an SQL Server 2005 zurückgegeben werden,
- eine `FillRowMethod`-Methode, hier `GetRow` (4), die die Spaltenwerte aus Instanzen der `Range`-Klasse extrahieren kann,
- eine Funktion, die eine `IEnumerable`-Schnittstelle zurückgibt, hier `LogRange` (3), und
- ein `SqlFunction`-Attribut (2), das vor der `LogRange`-Methode angegeben wird und den Namen der `FillRowMethod` spezifiziert.

Die `TableDefinition`-Eigenschaft von `SqlFunction` dokumentiert die Tabellendefinition und wird von Visual Studio 2005 verwendet, wenn es diese Funktion bereitstellt.

```
① struct Range
  {
     ...
  }
  public class RangeFunctions
  {
  ② [Microsoft.SqlServer.Server.SqlFunction
        (FillRowMethodName="GetRow",
        TableDefinition = "MinValue ..."
        )]
  ③ public static IEnumerable LogRange(
        SqlDouble minValue,
        SqlDouble maxValue,
        SqlInt32 count)
     {
        ...
     }
  ④ static public void GetRow(Object obj,
        out SqlDouble minValue,
        out SqlDouble maxValue,
        out SqlInt32 rangeNumber)
     {
        ...
     }
  };
```

Abbildung 3.44: Zusammenfassung einer Tabellenwertfunktion

3.8 Gespeicherte Prozeduren

Eine gespeicherte T-SQL-Prozedur ist einer benutzerdefinierten T-SQL-Funktion dahingehend ähnlich, dass sie Eingabeparameter und einen Rückgabewert besitzt. Allerdings kann man mit den Eingabewerten flexibler arbeiten, ist dafür aber beim Rückgabewert eingeschränkter. Außerdem darf eine gespeicherte Prozedur Änderungen an einer Datenbank vornehmen. Kapitel 4 beschäftigt sich ausführlich mit der Realisierung von gespeicherten Prozeduren. In diesem Kapitel geht es vor allem um die technischen Details von CLR-basierten gespeicherten Prozeduren.

Parameter für eine gespeicherte Prozedur besitzen eine Richtungseigenschaft. Ein Parameter ist standardmäßig ein Eingabeparameter. Mit der `OUTPUT`-Klausel wird er zum Ein-/Ausgabeparameter wie bei `ref` in C# oder bei `By Ref` in Visual Basic.

Eine gespeicherte Prozedur gibt einen Statuswert zurück. Der Rückgabewert zeigt den Ergebnisstatus der gespeicherten Prozedur an, wobei 0 bedeutet, dass die Prozedur ordnungsgemäß gearbeitet hat. Der Statuswert ist nicht dafür vorgesehen, die Ergebnisse einer gespeicherten Prozedur zurückzugeben.

Abbildung 3.45 zeigt die C#-Methode `Div` (1), die man als gespeicherte Prozedur verwenden kann. Die Methode berechnet Quotient und Divisionsrest der an sie übergebenen Parameter. Der erste Parameter, `dividend` (2), fungiert als Eingabeparameter. Der zweite Parameter, `divisor_quotient` (3), ist ein Ein-/Ausgabeparameter, der sowohl den Divisor übernimmt als auch den Quotienten zurückgibt. Der letzte Parameter, `remainder`, ist ein Ausgabeparameter. Ist `dividend` oder `divisor_quotient` gleich null, gibt die gespeicherte Prozedur das Ergebnis 1 zurück (5) und zeigt damit an, dass sie nicht erfolgreich war. Beachten Sie, dass `remainder` vor der Rückkehr aus der Methode auf einen Wert gesetzt werden muss, da es sich um einen Ausgabeparameter handelt. Bei erfolgreicher Ausführung gibt `Div` den Wert 0 zurück (6).

Normalerweise gibt man mit einer gespeicherte Prozedur Resultsets zurück und/oder nimmt Änderungen am Zustand der Datenbank vor. Kapitel 4 zeigt, wie man das mithilfe der CLR bewerkstelligt. Das hier vorgestellte Beispiel gibt zwei Werte zurück, was bei einer gespeicherten Prozedur mit Skalarwert nicht möglich ist, außer wenn sie einen benutzerdefinierten Typ zurückgibt. Kapitel 5 ist dem Thema benutzerdefinierte Typen gewidmet.

```
public class Math {
① public static SqlInt32 Div(
②    SqlInt32 dividend,
③    ref SqlInt32 divisor_quotient,
④    out SqlInt32 remainder)
   {
      if (dividend.IsNull || divisor_quotient.IsNull)
      {
         remainder = SqlInt32.Null;
      ⑤ return 1;
      }
      remainder = dividend.Value
        % divisor_quotient.Value;
      divisor_quotient =
        dividend.Value / divisor_quotient.Value;
   ⑥ return 0;
   }
}
```

Abbildung 3.45: CLR-gespeicherte Prozedur

Abbildung 3.46 zeigt, wie die gespeicherte Prozedur `Div` katalogisiert wird. Die Syntax ist die gleiche wie für das Erstellen einer gespeicherten T-SQL-Prozedur (1), außer dass die `AS`-Klausel die Schlüsselwörter `EXTERNAL NAME` anstelle eines Körpers enthält. Der zweite (2) und der dritte Parameter (3) verwenden das Schlüsselwort `OUTPUT`, um den in der CLR-Implementierung verwendeten Schlüsselwörtern `ref` und `out` zu entsprechen.

```
CREATE ASSEMBLY Numeric
FROM 'C:\assemblies\math.dll'

①CREATE PROCEDURE [Div]
    @dividend int,
  ②@divisor_quotient int OUTPUT,
  ③@remainder int OUTPUT
AS
EXTERNAL NAME [Numeric].Math.Div
```

Abbildung 3.46: Eine gespeicherte Prozedur laden

Parameter in der CLR lassen sich nur für die Eingabe, nur für die Ausgabe oder für die Ein- und Ausgabe deklarieren. T-SQL unterstützt in gespeicherten Prozeduren lediglich Parameter, die nur für die Eingabe oder die Ein-/Ausgabe deklariert sind. In T-SQL werden dagegen keine Ausgabeparameter unterstützt, selbst wenn das Schlüsselwort `OUTPUT` dies impliziert. Als `OUTPUT` markierte T-SQL-Parameter sind immer Ein-/Ausgabeparameter.

Wie bei CLR-basierten Funktionen unterscheiden sich CLR-basierte gespeicherte Prozeduren hinsichtlich der Verwendung nicht von gespeicherten T-SQL-Prozeduren. In Abbildung 3.47 ist zu sehen, wie die gespeicherte Prozedur `Div`

ausgeführt wird (1). Die Ergebnisse zeigen, dass ein Quotient von 11 dividiert durch 3 gleich 3 mit dem Rest 2 ist.

```
   DECLARE @i INT
   DECLARE @b INT
   SET @b = 3
① EXEC Div 11, @b OUTPUT, @i OUTPUT
   SELECT @b, @i
```

②

Results | Messages

	(No column name)	(No col
1	3	2

Abbildung 3.47: Eine gespeicherte Prozedur verwenden

Einer CLR-Methode, die die gespeicherte Prozedur implementiert, kann man das Attribut `SqlStoredProcedure` hinzufügen. Es besitzt eine einzelne Eigenschaft: `Name`. Visual Studio 2005 verwendet dieses Attribut, wenn die gespeicherte Prozedur bereitgestellt wird. SQL Server 2005 greift auf die Informationen im `SqlStoredProcedure`-Attribut nicht zurück.

3.9 Trigger

Ein Trigger in SQL Server 2005 kann durch eine CLR-Methode implementiert werden. Methoden, die einen Trigger implementieren, geben `void` zurück und übernehmen keine Parameter. Abbildung 3.48 zeigt eine CLR-Methode in C# (1), die sich als Trigger eignet. Wie bei Funktionen und gespeicherten Prozeduren muss es sich um eine statische öffentliche Methode in einer öffentlichen Klasse handeln. Der Körper der Methode greift auf die Datenbank zu und modifiziert diese möglicherweise (2). Kapitel 4 erläutert, wie man in einem Trigger auf eine Datenbank zugreift, und zeigt Beispiele für den Einsatz von Triggern.

```
public class Triggers
{
① public static OnMyUpdate()
   {
   ② ... code that checks/modifies database
   }
}
```

Abbildung 3.48: CLR-Trigger

Ein CLR-basierter Trigger muss in SQL Server hinzugefügt werden. Abbildung 3.49 zeigt die Syntax zum Erstellen eines Trigger. Sie entspricht der Syntax für einen T-SQL-Trigger, außer dass auf die `AS`-Klausel eine `EXTERNAL NAME`-Klausel folgt (2).

Eine CLR-Methode, die einen Trigger implementiert, kann man optional mit dem `SqlTrigger`-Attribut versehen. SQL Server 2005 greift auf die Informationen in diesem Attribut nicht zurück. Das Attribut besitzt drei Eigenschaften, die SQL Server 2005 verwendet, um den Trigger bereitzustellen. Die `Name`-Eigenschaft legt einen Namen für den Trigger fest. Fehlt diese Angabe, wird der Name der CLR-Methode als Trigger-Name herangezogen. Die Eigenschaft `Target` benennt die Tabelle, zu der der Trigger hinzugefügt wird. Diese Eigenschaft ist das Äquivalent der `ON`-Klausel in der T-SQL-Anweisung `CREATE TRIGGER`. Mit der `Event`-Eigenschaft spezifiziert man den Typ des Trigger – zum Beispiel »`FOR UPDATE`«. Die `Name`-Eigenschaft ist optional, jedoch sind sowohl die `Target`- als auch die `Event`-Eigenschaft für Visual Studio 2005 erforderlich, damit der Trigger bereitgestellt werden kann.

```
① CREATE TRIGGER reminder
   ON Sales
   AFTER INSERT, UPDATE
   AS
② EXTERNAL NAME Rules.Triggers.OnMyUpdate
```

Abbildung 3.49: Einen Trigger laden

3.10 Zusammenfassung

Dieses Kapitel hat gezeigt, dass man mit CLR-Funktionen gespeicherte Prozeduren, Funktionen und Trigger für T-SQL implementieren kann. CLR-Sprachen sind normalerweise bei numerischen Berechnungen besser geeignet und bieten auch für Experten, die in der Regel keine T-SQL-Anwendungen schreiben, eine vertraute Programmierumgebung. Der Namespace `System.Data.SqlTypes` stellt Datentypen bereit, die man in der CLR einsetzen kann, die aber wie die korrespondierenden Datentypen in SQL Server arbeiten.