Kapitel 14

Gespeicherte Prozeduren

In diesem Kapitel:

Konzepte: Die Wahrheit über gespeicherte Prozeduren

Dieses Kapitel stellt Ihnen die wichtigsten Server-Objekte für Transact-SQL-Entwickler vor: die gespeicherten Prozeduren. *Gespeicherte Prozeduren* (*Stored Procedures*) stellen in SQL Server das Pendant zu Subroutinen in prozeduralen Programmiersprachen dar und haben vieles mit diesen gemeinsam. Vor allem bieten sie eine einfache Möglichkeit, Geschäftlogik serverseitig in einer Datenbank zu realisieren. Obwohl T-SQL Ihnen nicht den Komfort einer »richtigen« Programmiersprache bietet, haben gespeicherte T-SQL-Prozeduren doch einen entscheidenden Vorteil: Sie werden unglaublich schnell ausgeführt! Wirklich komplexe Algorithmen können Sie bei Bedarf jederzeit in Prozeduren schreiben, die auf .NET-Assemblies basieren. Da wir uns aber gerade in dem Buchteil befinden, in welchem es um die Arbeit mit Transact-SQL geht, spielt dies (noch) keine Rolle. Der etwas seltsam anmutende Name »gespeicherte Prozedur« ist natürlich der Tatsache geschuldet, dass der Programmcode in der Datenbank gespeichert wird und so ein fester Bestandteil der Datenschicht wird. Analog dazu werden Sichten bisweilen auch als gespeicherte Abfragen bezeichnet.

In einigen Programmiermodellen sollten Sie unbedingt mit gespeicherten Prozeduren arbeiten – in anderen ist es von Ihrer Herangehensweise, dem Projektumfang oder sogar von den Vorlieben und Kenntnissen Ihrer Teammitglieder abhängig, ob Sie mit gespeicherten Prozeduren arbeiten möchten oder nicht. Stellt Ihr Projekt eine klassische Client/Server-Datenbankapplikation dar, dann können Sie mit gespeicherten Prozeduren auf einfache Art eine saubere Datenzugriffsschicht zwischen der Darstellungslogik und der Datenbank selbst einbauen. So können Sie in einem *ASP*.NET Webprojekt in vielen Fällen darauf verzichten, eigene Datenzugriffsklassen zu entwickeln, in Entwicklungsumgebungen, die wenig mit .NET zu tun haben, oder den Access SQL Server-Projekten ist der Einsatz gespeicherter Prozeduren beinahe obligatorisch. Durch die Kapselung der Datenzugriffslogik in Serverprozeduren können Sie Ihre Anwendung portabel halten. Selbst wenn Sie T-SQL so richtig ausreizen, um die maximale Leistung in den Datenbankzugriffen zu erzielen und Sprachfeatures verwenden, die es auf anderen Datenbankservern wie Oracle oder DB2 nicht gibt. Natürlich müssen die gespeicherten Prozeduren beim »Umzug« Ihrer Anwendung auf einen anderen Server dann nachimplementiert werden, aber das stellt normalerweise wegen der Einfachheit des serverseitigen Codes keine riesige Hürde dar. Sämtliche Datenbankmanagementsysteme – inzwischen auch MySQL – kennen jedenfalls das Datenbankobjekt »gespeicherte Prozedur« und Datenbankentwickler machen regen Gebrauch davon.

Für Applikationen, die sehr hohe Anforderungen an das Zeitverhalten des Datenzugriffs stellen – in der Regel sind das Echtzeitanwendungen – und gleichzeitig nicht nur triviale Datenbankoperationen (wie einfache *INSERT*- oder *SELECT*-Anweisungen) durchführen, sind gespeicherte Prozeduren notwendig, um die Geschäftslogik mit maximaler Geschwindigkeit ablaufen zu lassen. Die hohe Ausführungsgeschwindigkeit der Programmtexte von gespeicherten Prozeduren ist darin begründet, dass sich deren Binärcode (ganz im Widerspruch zum Namen *gespeicherte* Prozedur) ständig im Speicherpool von SQL Server befindet und ohne Umschweife bei Bedarf in dessen Prozessraum ausgeführt werden kann. Die vorbereitenden Schritte, die normalerweise vor der Ausführung von T-SQL-Texten auf einem Prozessor durchlaufen werden, entfallen. Darum wird es in diesem Kapitel noch ausführlich gehen.

Es gibt es noch weitere wichtige Argumente dafür, sich mit den Grundlagen gespeicherter Prozeduren auseinanderzusetzen. Selbst, wenn Sie nie vorhaben, gespeicherte Prozeduren in T-SQL zu schreiben,

sondern dazu eine .NET-Programmiersprache einsetzen möchten, und selbst, wenn Sie vorhaben, *nie* mit gespeicherten Prozeduren zu arbeiten, so hilft das Durcharbeiten dieses Kapitels Ihnen beim Verständnis der Funktionsweise dieses Programmierobjektes. Und dies wiederum ist nützlich, wenn es darum geht, die Abläufe in der Kommunikation von Datenzugriffsbibliotheken wie OLEDB, ODBC, ADO oder ADO.NET mit SQL Server zu untersuchen, bei denen häufig Techniken eingesetzt werden, die auf dem Einsatz gespeicherter Prozeduren beruhen.

In diesem Kapitel stelle ich zunächst die Grundlagen von gespeicherten Prozeduren vor, danach geht es um die Fehlerbehandlung und abschließend kommt als nicht ganz unwichtiger Teil eine Betrachtung von Performanceaspekten. Eine kleine Warnung vorne weg: Während in bestimmten Situationen die bloße Verwendung gespeicherter Prozeduren tatsächlich einen Leistungsgewinn bringt, kann bei unüberlegter Verwendung auch das genaue Gegenteil eintreten – Ihre Prozeduren machen den Datenbankzugriff langsamer! Um zu erfahren, wie es dazu kommt, sollten Sie dieses Kapitel bis zum Schluss lesen.

Damit Sie sich eine erste Meinung zum Einsatz gespeicherter Prozeduren bilden können, sind an dieser Stelle einige der wichtigsten Argumente für deren Einsatz zusammengestellt.

- **Implementierung der Datenzugriffsschicht**

  Es gibt gute Gründe, die im Applikationsentwurf für die Implementierung einer dedizierten Datenzugriffsschicht sprechen. Mithilfe gespeicherter Prozeduren können Sie diese direkt im Datenbankserver realisieren. Dadurch wird Ihre Anwendung unabhängig von den Implementierungsdetails der Datenbank. Tabellen und Feldnamen können beliebig geändert werden – solange die Schnittstellen der gespeicherten Prozeduren sich nicht ändern, berührt das die Clientprogrammierung nicht. Im Programmcode der serverseitigen Prozeduren können zusätzlich Optimierungen durchgeführt werden, falls einzelne Zugriffe zu langsam erscheinen. Der clientseitige Code wird von umfangreichen (und oft unübersichtlichen) T-SQL-Befehlsstrings befreit und nicht zuletzt lässt sich Ihre Applikation leichter auf einen anderen Datenbankserver portieren.

- **Kapselung der Business-Logik**

  Gespeicherte Prozeduren sind in Client/Server-Applikationen eine hervorragende Möglichkeit zur klaren Trennung von Programmlogik und Darstellung. Die Geschäftslogik wird in Prozeduren gekapselt und der Client übernimmt vor allen Dingen die Bereitstellung des Benutzer-Interface. Die Wartung des Programmcodes für die Geschäftslogik vereinfacht sich.

  In SQL Server 2005 wird dieses Programmiermodell durch die Möglichkeit der Verwendung einer .NET-Programmiersprache in der CLR-Laufzeitumgebung natürlich um ein Vielfaches attraktiver, als es bei einer serverseitigen T-SQL-Programmierung der Fall ist. In C# oder VB.NET lassen sich Business-Objekte naturgemäß viel geschmeidiger entwickeln als im kargen Transact-SQL.

- **Verarbeitung auf dem Server**

  Gespeicherte Prozeduren werden immer komplett auf dem Server verarbeitet. Dadurch steht der Prozedur die vollständige Leistungsfähigkeit einer großen Datenbankmaschine zur Verfügung. Bei komplexen Abfragen und Berechnungen bedeutet dies eine wesentliche Steigerung der Abfrageleistung. Dieses Argument spielt bei einem in einer .NET-Programmiersprache geschriebenen Client allerdings eine untergeordnete Rolle, da hier die Ausführung des SQL-Codes immer auf dem Server passiert, bei Datenbankclients wie Visual FoxPro, Crystal Reports oder Access sind Sie durch den Einsatz von gespeicherten Prozeduren immer auf der sicheren Seite. Verwenden Sie die Reporting Services von SQL Server für Ihre Berichte, dann werden Ihre Datenquellen übersichtlicher.

- **Verringerung von Netzwerkverkehr**

  Da der Client nur den Prozeduraufruf zum Server schickt und anschließend die fertigen Ergebnisse zu sehen bekommt, verringert sich der Netzwerkverkehr zwischen Client und Server. Es gibt diverse Anwendungsfälle bei denen dies deutlich zu spüren ist.

- **Bessere Test- und Optimierbarkeit**

  Das Testen Ihrer Programmierung wird durch die Verwendung gespeicherter Prozeduren verbessert. Sie benötigen keinerlei Oberfläche für das Ausführen von gespeicherten Prozeduren. Stattdessen können Sie SQL Server-Skripte vorbereiten, über die eine Prozedur unter gesicherten Bedingungen getestet werden kann. Dies ermöglicht ein schnelles und vor allen Dingen sehr einfach wiederholbares Testen. Aufwändigerer T-SQL-Code lässt sich in einer gespeicherten Prozedur einfacher optimieren. Falls Sie auf den Einsatz gespeicherter Prozeduren verzichten, dann kann man nur empfehlen, die Datenzugriffe auf dem Client in Klassen zu kapseln, damit Sie die genannten Vorteile ebenfalls erlangen können.

- **Parametrisierung von Abfragen**

  SQL Server-Sichten können keine Parameter verarbeiten, dies ist nur in benutzerdefinierten Funktionen oder eben Prozeduren möglich. Gespeicherte Prozeduren sind gegenüber Funktionen etwas einfacher zu implementieren (und zu testen), haben aber den Nachteil, dass die von ihnen gelieferten Ergebnismengen auf dem Server nicht ohne weiteres weiterverarbeitet werden können.

- **Schnellere Ausführung**

  Da eine gespeicherte Prozedur nach der ersten Ausführung in kompilierter Form im Cache von SQL Server vorliegt, wird keine Zeit für das Vorbereiten der darin enthaltenen Befehle mehr benötigt. Wie schon in der Einleitung erwähnt, geht das aber nicht in jedem Fall gut. Warum das so ist und wie Sie Probleme vermeiden können, erläutere ich am Ende dieses Kapitels.

Mit gespeicherten Prozeduren arbeiten

Sie können gespeicherte Prozeduren in den Entwicklungsumgebungen von Visual Studio oder des SQL Server Management Studios entwerfen. Da die Unterstützung in beiden Fällen aber sehr rudimentär ist, kommen Sie nicht um die Aufgabe herum, sich mit den Möglichkeiten vertraut machen, die T-SQL dem Entwickler bietet, um gespeicherte Prozeduren anzulegen und zu bearbeiten. Letzten Endes bieten nämlich beide Entwicklungsumgebungen einfach nur die Möglichkeit an, das Erstellungs- oder Änderungsskript für eine Prozedur auszuführen. Der Rest ist Ihre Sache. Visual Studio erlaubt zusätzlich das Debuggen gespeicherter Prozeduren (bietet aber ansonsten weniger Möglichkeiten als das Management Studio – es ist ein Kreuz, sich zwischen den Umgebungen entscheiden zu müssen).

HINWEIS Der Wegfall des Debuggers im neuen Abfrageeditor ist natürlich eine bittere Pille für T-SQL-Entwickler, die gewohnt waren, ihre Prozeduren im Query Analyzer von SQL Server 2000 zu entwickeln und zu testen. So schön die neuen Möglichkeiten für das projektorientierte Arbeiten mit T-SQL-Code im Management Studio auch sind – beim intensiven Entwickeln gespeicherter Prozeduren benötigt man einfach einen Debugger. Hier ist Umdenken angesagt und die Gewöhnung an das Visual Studio – auch für T-SQL-Freaks.

Gespeicherte Prozeduren anlegen

Da die Schritte beim Anlegen von gespeicherten Prozeduren in den beiden Arbeitsumgebungen nahezu identisch sind, beschränke ich mich bei den Erklärungen auf das Management Studio. Um eine neue gespeicherte Prozedur anzulegen, können Sie im Ordner *Gespeicherte Prozeduren* des Objekt-Explorers unterhalb von *Programmierbarkeit* den Befehl *Neue gespeicherte Prozedur* aus dem Kontextmenü wählen. Im T-SQL-Editor wird daraufhin ein leeres Fenster mit einer passenden Schablone angelegt. Neben den allgemeinen Funktionen des Editors wie dem integrierten Abfragedesigner, die in Kapitel 5 vorgestellt wurden, gibt es keine spezielle Unterstützung für das Programmieren einer gespeicherten Prozedur. Abbildung 14.1 zeigt solch eine Prozedurschablone.

Da sich der Komfort bei dieser Vorgehensweise in Grenzen hält, können Sie das Anlegen einer neuen gespeicherten Prozedur genauso gut mit einem leeren Abfragefenster starten und das *CREATE PROCEDURE* plus Programmtext direkt eingeben. Eine gute Idee ist die Verwendung einer selbst definierten Schablone, die einen vorgegebenen Prozedurkopf mit Kommentarblock für die Dokumentation enthält.

SHIVA\YUKON.n... SQLQuery1.sql | Zusammenfassung

```
-- ============================================
-- Template generated from Template Explorer using:
-- Create Procedure (New Menu).SQL
--
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the procedure.
-- ============================================
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Author:      <Author,,Name>
-- Create date: <Create Date,,>
-- Description: <Description,,>
-- =============================================
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
    -- Add the parameters for the stored procedure here
    <@Param1, sysname, @p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, , 0>,
    <@Param2, sysname, @p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, , 0>
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    -- Insert statements for procedure here
    SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

Abbildung 14.1 Gespeicherte Prozedur im Management Studio anlegen

Für das Ändern einer vorhandenen Prozedur *müssen* Sie, wenn Sie mit einem Sourcecode-Kontrollsystem arbeiten, auf jeden Fall auf das Erstellungsskript zurückgreifen und nach dessen Auschecken eine neue Version von diesem erstellen. Der Objekt-Explorer bietet alternativ die Möglichkeit, die Definition einer vorhandenen gespeicherten Prozedur aus einer Datenbank zu extrahieren und als Skript anzuzeigen. Diese Möglichkeit ist nun tatsächlich sehr bequem und steht über das Kontext-Kommando *Ändern* zur Verfügung, nachdem Sie eine Prozedur markiert haben. Das entsprechende *ALTER*-Skript wird in einem neuen Editorfenster angezeigt. Nachdem Sie die Änderungen im Programmtext vorgenommen und das Skript ausgeführt haben, liegt die neue Version der Prozedur in der Datenbank vor.

Gespeicherte Prozeduren in T-SQL bearbeiten

Mit den entsprechenden T-SQL-DDL-Befehlen können Sie gespeicherte Prozeduren anlegen, ändern und ausführen. Für das Erstellen einer neuen Prozedur wird das folgende SQL-Kommando verwendet:

```
CREATE PROC [ EDURE ] [schema_name.] procedure_name [ ; number ]
   [ { @parameter [ type_schema_name. ] data_type  }
   [ VARYING ] [ = default ] [ OUT [ PUT ]]
```

```
   [ ,...n ]
[ WITH { RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION } ]
AS
sql_statement [ ...n ]
```

Lässt man alle optionalen Anteile weg, dann kann eine Prozedur im einfachsten Fall so angelegt werden, wie es hier im ersten Beispiel vorgeführt wird:

```
CREATE PROC OrdersGetAll
AS
SELECT * FROM Orders
ORDER BY [ID]
```

Listing 14.1 Anlegen einer gespeicherten Prozedur

Der Prozedurkopf besteht in diesem Fall nur aus dem Namen der Prozedur. Unter diesem wird der Prozedurtext in der Datenbank abgelegt. Ein Prozedurname kann, wie jeder andere Objektname, innerhalb einer Datenbank nur ein einziges Mal vergeben werden. Eine Ausnahme stellt die Möglichkeit dar, dass die gleiche Prozedurbezeichnung in zwei verschiedenen Schemata verwendet werden kann, also etwa *ProductDepartment.OrdersGetAll* und *SalesDepartment.OrdersGetAll.* Zusätzlich kennt T-SQL noch den etwas obskuren Fall, dass eine Prozedur unter dem gleichen Namen in verschiedenen Versionen existieren kann. Dies wird über eine Nummerierung im Prozedurkopf – genannt *Gruppennummer* – erreicht. Mit den folgenden Befehlen können Sie zwei Versionen der *OrdersGetAll*-Prozedur anlegen:

```
CREATE PROC OrdersGetAll; 1
AS …
CREATE PROC OrdersGetAll; 2
AS …
```

Im Objekt-Explorer werden die verschiedenen Prozeduren gleichen Namens und unterschiedlicher Gruppennummer als *eine* Prozedur unter dem gemeinsamen Namen angezeigt. Sie merken erst, was los ist, wenn Sie sich die Prozedurdefinitionen über den Kontextbefehl *Ändern* anzeigen lassen. Dann werden nämlich die Quelltexte aller Varianten angezeigt. In der wirklichen Welt habe ich so ein Vorgehen bis jetzt allerdings noch nie gesehen – vielleicht fällt Ihnen ja eine kluge Anwendung dazu ein.

Eine Prozedurdefinition reicht im T-SQL-Editor entweder bis zum Ende der Skriptdatei oder bis zum nächsten *GO,* also dem Ende eines Batches. Genau wie bei den Sichten wird bei Prozeduren nach dem Ausführen von *CREATE* die Definition des neuen Objekts in den Systemtabellen abgelegt. Und genau wie bei den Sichten können Sie den Quelltext in Ihrer Datenbank verbergen, wenn Sie die Option *WITH ENCRYPTION* verwenden. Die weiteren Optionen *EXECUTE AS* und *WITH RECOMPILE* werden weiter unten in den Abschnitten zur Sicherheit und zur Performance erläutert.

Die in den Metadaten hinterlegten Informationen zu einer gespeicherten Prozedur können Sie über die Katalogsichten *sys.procedures*, *sys.sql_modules*, *sys.parameters* und *sys.sql_dependencies* ausfindig machen. In *sys.procedures* befindet sich der Katalog der in einer Datenbank vorhandenen Prozeduren – sowohl der vom System, als auch der vom Benutzer definierten. Die Sicht *sys.sql_modules* liefert

die Texte, *sys.parameters* die Parameterdefinitionen und *sys.sql_dependencies* die Abhängigkeiten zu Tabellen und anderen Datenbankobjekten.

Durch das nachfolgende T-SQL-Skript werden nacheinander die Definition der Prozedur *OrdersGetAll*, die Parameter (es gibt keine) und die Abhängigkeiten zu anderen Objekten angezeigt.

```
DECLARE @object_id int

-- der Prozedurtext
SET @object_id =
  ( SELECT object_id
     FROM sys.procedures
     WHERE [Name] = 'OrdersGetAll' )
SELECT
   definition
FROM
   sys.sql_modules sm
WHERE
   @object_id  = object_id

-- die Parameter
SELECT
   sp.[name] AS ParameterName, st.[name] AS TypeName
FROM
   sys.parameters sp
INNER JOIN
   sys.types st
ON
   sp.system_type_id = st.system_type_id
WHERE
   @object_id  = object_id

-- referenzierte Objekte
SELECT
   so.[name] AS ObjectName, type_desc AS ObjectType
FROM
   sys.sql_dependencies sd
INNER JOIN
   sys.objects so
ON
   sd.referenced_major_id = so.object_id
WHERE
   @object_id  = sd.object_id
```

Listing 14.2 Anzeige von Informationen zu einer gespeicherten Prozedur via T-SQL Skript

Über das Auslesen der Spalte *definition* gelangt man in der Sicht *sys.modules* zum gespeicherten T-SQL-Text der Prozedur. Aus *sys.parameters* kann man alle möglichen Informationen zu den Parametern abrufen. Das Beispiel beschränkt sich auf die Bezeichnung des Parameters und den Datentyp. Um den Datentyp im Klartext auszugeben, wird ein *JOIN* auf die Systemsicht *sys.types* gemacht. Diese enthält die Spezifikationen der von SQL Server bereitgestellten Datentypen. Der bei der dritten Abfrage verwendete Trick besteht in der Verknüpfung mit der Katalogsicht *sys.objects*. In dieser werden *alle*

SQL Server-Objekte dargestellt. Für die Verknüpfung auf die Sicht *sys.sql_dependencies*, die die Abhängigkeiten von Objekten anzeigt, wird die Spalte *referenced_major_id* angeboten.

Ist eine Prozedur einmal angelegt, so wird zum Ändern ihrer Definition das *ALTER*-Kommando eingesetzt.

```
ALTER PROC OrdersGetAll
AS
SELECT * FROM Orders
ORDER BY OrderDate
```

Listing 14.3 Ändern der Prozedurdefinition mit T-SQL

Das Löschen geht wie üblich ganz einfach über *DROP*. Etwaige Abhängigkeiten anderer Objekte von der zu löschenden Prozedur werden nicht geprüft. Wenn Sie auf Nummer sicher gehen wollen, können Sie sich über die Funktion *Abhängigkeiten Anzeigen* im Objekt-Explorer vergewissern, ob es Auswirkungen des Löschens gibt, oder Sie fragen in T-SQL, analog zum Listing 14.2, die Systemsicht *sys.sql_dependencies* ab. Es folgt der Befehl für das endgültige Entfernen des Objekts *OrdersGetAll*:

```
DROP PROCEDURE OrdersGetAll
```

Listing 14.4 Löschen einer gespeicherten Prozedur

In Skripten, die dem Anlegen von SQL Server-Objekten dienen, finden Sie häufig Programmzeilen, mit denen geprüft wird, ob ein Objekt gleichen Namens bereits in der Datenbank existiert – so beispielsweise in Erstellungsskripten, die von Visual Studio vorgeschlagen werden oder in Skripten, die Sie mit dem Skriptgenerator des Management Servers erzeugen können. Eine Möglichkeit, eine bereits vorhandene Prozedur gleichen Namens aus der Datenbank zu entfernen, bevor diese neu angelegt wird, zeigt der folgende Programmschnipsel:

```
IF EXISTS (SELECT * FROM sys.procedures WHERE [name] = OrdersGetAll)
   DROP PROC OrdersGetAll
GO
```

Sind in einer Datenbank bereits Rechte für die Benutzung der Prozedur vergeben worden, dann dürfen Sie nicht vergessen, diese neu zu setzen, da diese durch das *DROP* natürlich ebenfalls gelöscht wurden. Beim Ändern einer Definition mit *ALTER* ist dies nicht notwendig.

Das Aufrufen von gespeicherten Prozeduren in T-SQL selbst erfolgt über das *EXEC*-Kommando:

```
EXEC[UTE] prozedurname
```

Die Prozedur *OrdersGetAll* wird also folgendermaßen aufgerufen:

```
EXEC OrdersGetAll
```

Listing 14.5 Aufruf einer gespeicherten Prozedur

Verschachtelte Aufrufe gespeicherter Prozeduren sind möglich. Allerdings lässt SQL Server maximal 32 Aufrufe innerhalb einer *EXECUTE*-Kette zu. Auf rekursive Algorithmen wird man also verzichten müssen. Das ist in seltenen Fällen durchaus schade, wird aber, zumindest was das Abfragen von Daten angeht, durch die Möglichkeit der rekursiven *SELECT*-Kommandos mit Common Tables Expressions (CTEs) wettgemacht. Der Server erkennt das Überschreiten der Schachtelungstiefe, noch bevor der erste Befehl ausgeführt wird, und verweigert dann mit einer Fehlermeldung die Ausführung. Dadurch wird zumindest verhindert, dass ein Programmabschnitt nur halb ausgeführt wird.

Parametrisierte gespeicherte Prozeduren

Selbstverständlich können gespeicherte Prozeduren auch mit Parametern versorgt werden. Dies können Ein- oder Ausgabeparameter (und beides zugleich) sein. Zusätzlich kann einer Prozedur ein optionaler Rückgabecode mitgegeben werden. Die Teilsyntax für eine einzelne Parameterdeklaration hat die folgende Form:

```
@parameter data_type [ = default ] [ OUTPUT ]
```

Parameter werden direkt hinter dem Prozedurkopf und vor dem Schlüsselwort *AS* angegeben. Genau wie eine lokale T-SQL-Variable wird ein Parameter durch ein vorangestelltes @-Zeichen gekennzeichnet. Ein Parameter kann im Prozedurtext denn auch exakt wie eine lokale Variable eingesetzt werden – also in einem Ausdruck, speziell in einer *SELECT*-Liste oder einer *WHERE*-Klausel. Auf den Parameternamen folgt in der Definition ein SQL Server-Datentyp, der für ein Argument dieses Parameters verwendet werden soll. Das kann einer der SQL Server-Basistypen sein, ein Benutzerdatentyp, der in T-SQL definiert wurde, oder ein Benutzertyp, der in einer *.CLR*-Sprache implementiert wurde.

Möchten Sie einen Standardwert für einen Parameter vorsehen, für den beim Prozeduraufruf kein Wert angegeben wird, so geben Sie diesen hinter einem Gleichheitszeichen gleich mit an. Dieser Wert wird für den Parameter verwendet, falls beim Aufruf der Prozedur kein aktuelles Argument übergeben wird. Dummerweise sind nur Konstanten oder der Wert *NULL* erlaubt. Der folgende Programmtext stellt das Erstellungsskript für die *netShop*-Prozedur *GetOrders* dar, die mit einem einzelnen Eingabeparameter versehen ist:

```
CREATE PROC GetOrders
   @PayingMethodID int = 1
AS
SELECT * FROM Orders
WHERE PayingMethodID = @PayingMethodID
ORDER BY [ID]
```

Listing 14.6 Gespeicherte Prozedur mit Parameter

PayingMethodID steht in der *netShop*-Datenbank für die Zahlweise des Kunden. Die Prozedur erlaubt in der neuen Form durch die Parametrisierung ein Filtern der Aufträge nach der Bezahlart. Dazu wird der Parameter in der *WHERE*-Klausel verwendet. Wird die Prozedur ohne die Übergabe eines Wertes für *@PayingMethodID* aufgerufen, dann wird automatisch der Standardwert 1 im Kriterium verwendet (dies steht in der Standard-Testdatenmenge des *netShop* für »Bankeinzug«). Die Übergabe eines Parameters in einem Prozeduraufruf sieht in T-SQL so aus:

```
EXEC GetOrders 2
```

Listing 14.7 Prozeduraufruf mit Parameter

Da die Prozedur einen Standardwert für den Parameter *@PayingMethodID* besitzt, kann man das Argument auch weglassen:

```
EXEC GetOrders
```

Listing 14.8 Prozeduraufruf mit Standardwert

Jetzt wird nach dem Wert 1 für die Bezahlmethode gefiltert. Bitte beachten Sie: Wenn Sie bei der Prozedurdeklaration auf die Angabe eines Standardwertes für einen Parameter verzichten, dann muss beim Aufruf der Prozedur auch tatsächlich ein Argument übergeben werden. Ansonsten bricht SQL Server die Verarbeitung mit einem Laufzeitfehler ab.

Werden mehrere Parameter benötigt, so bildet man eine durch Kommata separierte Liste. Diese Liste ist im Prinzip beliebig lang (na ja, ehrlich gesagt darf sie 2.100 Einträge umfassen, aber damit kommt man normalerweise aus). Unsere Beispielprozedur kann daher wie folgt um neue Parameter erweitert werden:

```
CREATE PROC GetOrders
   @PayingMethodID int = 1,
   @ShippingMethodID int = 1,
   @OrderDate smalldatetime,
   @PayingCosts smallmoney = 0
AS
SELECT * FROM Orders
WHERE
   PayingMethodID = @PayingMethodID AND
   ShippingMethodID = @ShippingMethodID AND
   OrderDate >= @OrderDate AND
   PayingCosts >= @PayingCosts
ORDER BY [ID]
```

Listing 14.9 Prozedur mit mehreren Parametern

Für den Aufruf einer Prozedur mit mehreren Parametern gibt es in T-SQL verschiedene Möglichkeiten. Die erste Art der Parameterübergabe ist die Angabe der aktuellen Werte in der korrekten Reihenfolge (positionale Übergabe):

```
EXEC GetOrders 2, 1, '2005-09-01', 1
```

Listing 14.10 Positionale Parameterübergabe

Im Gegensatz zum Prozeduraufruf in manchen Programmiersprachen können Sie Argumente nicht so einfach weglassen und »leere« Kommata stehen lassen. Sie dürfen die Argumentenliste nur von hinten nach vorne *ohne Lücken* verkürzen. Das sieht dann so aus:

```
EXEC GetOrders 2, 1, '2005-09-01'
```

Listing 14.11 Verkürzte Parameterliste

Soll ein Wert aus der Mitte der Liste entfallen, dann können Sie explizit den Wert *NULL* übergeben oder mit benannten Parametern arbeiten. Die Übergabe erfolgt dabei durch die Angabe des Parameternamens mit einer Wertzuweisung. Bei dieser Art des Prozeduraufrufs können Sie die Argumente in einer beliebigen Reihenfolge angeben und auch beliebige Lücken lassen. Denken Sie aber an die Standardwerte für ausgelassene Parameter! Das nächste Beispiel demonstriert dies.

```
EXEC GetOrders @OrderDate= '2002-09-01', @PayingMethodID = 2
```

Listing 14.12 Parameterübergabe per Namen

Dass als Standardwerte für Parameter gespeicherter Prozeduren in T-SQL nur Konstante und keine Ausdrücke verwendet werden dürfen, stellt natürlich eine deftige Einschränkung dar. Es gibt aber einen »klassischen« Workaround, über den Sie feststellen können, ob beim Prozeduraufruf ein bestimmter Parameter nicht gefüllt wurde, und dann im Programm darauf reagieren. Dieses häufig benutzte Verfahren basiert auf dem Vorgehen, dass für nicht benutzte Parameter ein spezieller Standardwert vorgesehen wird – häufig der Wert *NULL.* Mit diesem Trick lässt sich die Prozedur *GetOrders* so umbauen, dass beim Aufruf auch der Parameter *@OrderDate* weggelassen werden kann und dann ein berechnetes Datum als Startwert für die Selektion verwendet wird (ein statisches Datum macht hier ganz offensichtlich wenig Sinn). Mit der nächsten Version der Prozedur werden alle Aufträge selektiert, die höchstens ein Jahr zurückliegen, wenn kein Datum in `@OrderDate` übergeben wird.

```
CREATE PROC GetOrders
   @PayingMethodID int = 1,
   @ShippingMethodID int = 1,
   @OrderDate smalldatetime = NULL,
   @PayingCosts smallmoney = 0
AS

SET @OrderDate = ISNULL(@OrderDate, DATEADD(yy, -1, GETDATE()))

SELECT * FROM Orders
WHERE
   PayingMethodID = @PayingMethodID AND
   ShippingMethodID = @ShippingMethodID AND
   OrderDate >= @OrderDate AND
   PayingCosts >= @PayingCosts
ORDER BY [ID]
```

Listing 14.13 Prozedur mit dynamischem Standardwert für Parameter

Probieren Sie die neue Form der Prozedur einmal mit dem folgenden Aufruf aus:

```
EXEC GetOrders 2, 1
```

Listing 14.14 Aufruf der neuen Prozedur

Die Parameter einer gespeicherten Prozedur sind standardmäßig Eingabeparameter. In üblichen Programmiersprachen wird diese Art der Parameterübergabe Call by Value genannt. Man kann Werte

an die Prozedur übergeben und die Prozedur arbeitet mit diesen Werten. Es können aber keine Ergebniswerte an den Aufrufer zurückgegeben werden. Möchte man Werte aus der Prozedur à la Call by Reference zurückgeben, so werden dafür so genannte *OUTPUT*-Parameter verwendet. Die Deklaration eines *OUTPUT*-Parameters sieht aus, wie im nächsten Beispiel gezeigt:

```
@FirstOrderDate smalldate OUTPUT
```

Den Wert eines Ausgabeparameters legt man durch die Zuweisung eines Ausdrucks an den Parameternamen fest. In der nächsten Ausbaustufe der Beispielprozedur *GetOrders* werden zwei zusätzliche Parameter verwendet, die den jüngsten und ältesten Auftrag zu einem gegebenen Satz von Argumenten zurückgeben. Die Prozedur liefert in dieser erweiterten Form also sowohl Werte in Form einer Ergebnismenge und gleichzeitig Ergebnisse in den *OUTPUT*-Parametern zurück.

```
CREATE PROC GetOrders
   @PayingMethodID int = 1,
   @ShippingMethodID int = 1,
   @OrderDate smalldatetime = NULL,
   @PayingCosts smallmoney = 0,
   @FirstOrderDate smalldatetime OUTPUT,
   @LastOrderDate  smalldatetime  OUTPUT

AS

SET @OrderDate = ISNULL(@OrderDate, DATEADD(yy, -1, GETDATE()))

SELECT
   @FirstOrderDate = MIN(OrderDate),
   @LastOrderDate  = MAX(OrderDate)
FROM
   Orders
WHERE
   PayingMethodID = @PayingMethodID AND
   ShippingMethodID = @ShippingMethodID AND
   OrderDate >= @OrderDate AND
   PayingCosts >= @PayingCosts

SELECT * FROM Orders
WHERE
   PayingMethodID = @PayingMethodID AND
   ShippingMethodID = @ShippingMethodID AND
   OrderDate >= @OrderDate AND
   PayingCosts >= @PayingCosts
ORDER BY [ID]
```

Listing 14.15 Prozedur mit OUTPUT-Parametern

Die Ausgabeparameter beziehen ihre Werte aus einer zweiten Abfrage, die eine identische *WHERE*-Klausel besitzt. Zur Erinnerung – in Kapitel 8 finden Sie die Details: Sie können über ein *SELECT*-Kommando *entweder* lokale Variablen (oder eben Parameter) füllen *oder* eine Ergebnismenge generieren, die von SQL Server zum Client geschickt wird. Beides zugleich ist nicht möglich – daher *zwei* Abfragen. Soll eine gespeicherte Prozedur mit Ausgabeparametern in T-SQL aufgerufen werden, so

muss eine spezielle Syntax verwendet werden. Für die Ergebnisparameter sind beim Aufruf natürlich lokale Variable vorzusehen. Diese müssen, wie die entsprechenden Ausgabeparameter in der Prozedurdeklaration selbst, durch das Schlüsselwort *OUTPUT* gekennzeichnet werden, wenn eine Werterückgabe stattfinden soll. Ein Programmausschnitt, der die gespeicherte Prozedur *GetOrders* verwendet, muss damit dem folgenden Beispiel entsprechen. Diese Art eines Prozeduraufrufs ist sicherlich nicht übermäßig elegant, dafür ist der T-SQL-Code aber wieder einmal sehr effektiv.

```
DECLARE @theFirstOrderDate smalldatetime
DECLARE @theLastOrderDate smalldatetime

EXEC
   GetOrders 2, 1, '2005-09-01', 1, @theFirstOrderDate OUTPUT, @theLastOrderDate OUTPUT

PRINT @theFirstOrderDate
PRINT @theLastOrderDate
```

Listing 14.16 Aufruf einer Prozedur mit Rückgabeparametern

Neben den eigentlichen Parametern können Sie einer gespeicherten Prozedur, ähnlich einer Funktion, zusätzlich einen Ergebniswert mitgeben. Dieser wird häufig dafür eingesetzt, das Ergebnis der Ausführung, also einen Return-Code, zu transportieren. Die Verwendung eines Rückgabewertes muss nicht besonders deklariert werden. Sie können das Schlüsselwort *RETURN* an jeder Stelle des Prozedurrumpfes einsetzen, um die Ausführung einer Prozedur zu beenden und gleichzeitig einen Return-Code zu generieren. Dieser ist immer vom Typ *int*. Der folgende (verkürzte) Code-Abschnitt demonstriert den Einsatz des Schlüsselwortes *RETURN* in der *GetOrders*-Prozedur:

```
CREATE PROC GetOrders
   …
AS

SET @OrderDate = ISNULL(@OrderDate, DATEADD(yy, -1, GETDATE()))

SELECT
   @FirstOrderDate = MIN(OrderDate),
   @LastOrderDate  = MAX(OrderDate)
FROM
   Orders
WHERE
   …

IF @FirstOrderDate IS NULL OR @LastOrderDate IS NULL
   -- keine Werte gefunden: Fehlercode zurück
   RETURN -1
ELSE
    -- die Ausführung wird erfolgreich sein
   SELECT * FROM Orders
   WHERE
   …
   RETURN 0
```

Listing 14.17 Gespeicherte Prozedur mit Rückgabewert

Der Aufruf einer gespeicherten Prozedur mit *RETURN*-Wert wird in T-SQL naturgemäß noch einmal ein bisschen klobiger. Da Sie aber in der Regel eine Prozedur via ADO.NET vom Client aus aufrufen werden, kann Ihnen das eigentlich egal sein. Ein Beispiel für den T-SQL-Aufruf folgt auf dem Fuße:

```
…
DECLARE @theReturnCode int

EXEC @theReturnCode = GetOrders 2, 1, '2007-09-01', 1, @theFirstOrderDate OUTPUT, @theLastOrderDate
OUTPUT

PRINT @theReturnCode
…
```

Listing 14.18 Aufruf einer gespeicherten Prozedur mit Rückgabewert

Wie Sie gespeicherte Prozeduren mit oder ohne Parameter und mit gar keiner, einer oder gleich mehreren Ergebnismengen von ADO.NET aus am besten verwenden – das erfahren Sie in Kapitel 26.

Gespeicherte Prozeduren und Sicherheit

Um eine gespeicherte Prozedur auszuführen, benötigt der Benutzer die *EXECUTE*-Berechtigung für diese Prozedur. Welche Berechtigungen für die in einer Prozedur angesprochenen Objekte gelten, ist vom Ausführungskontext abhängig, der zum Zeitpunkt des Aufrufs gilt. Dies ist *nicht* der Kontext, in welchem der Benutzer in seiner Session mit dem Server arbeitet, sondern – zumindest im Großen und Ganzen – der Kontext, der für den Ersteller der Prozedur gilt. Das können Sie sehr einfach überprüfen, indem Sie einen neuen Benutzer in der *netShop*-Datenbank einrichten, dem Sie überhaupt keine Berechtigungen in der Datenbank geben. Dieser wird per *SELECT* nichts aus der *Orders*-Tabelle lesen dürfen, er kann aber problemlos die im ersten Beispiel des Kapitels angelegte Prozedur verwenden, um die Daten zu lesen. Um es Ihnen einfacher zu machen, folgen hier zwei Testskripte. Führen Sie als erstes das Skript für das Anlegen eines neuen Logins auf dem Server und eines neuen Benutzers in der Datenbank aus.

```
CREATE LOGIN Pit
WITH
PASSWORD = 'sesam'
USE netShop
CREATE USER Pit FOR LOGIN Pit
GRANT EXECUTE ON OrdersGetAll TO Pit
```

Listing 14.19 Anlegen des neuen Benutzers Pit

Öffnen Sie nun ein neues Abfragefenster und melden Sie sich mit dem Benutzernamen »Pit« und dem Passwort »sesam« an. Testen Sie den Zugriff auf die Tabelle *Customers* mit einem entsprechenden *SELECT*-Befehl.

```
SELECT * FROM Customers
```

Listing 14.20 Zugriff auf die Customers-Tabelle

Pit kann nicht zugreifen. Durch das Ausführen der Prozedur *OrdersGetAll* kann er aber die Daten der Tabelle *Customers* lesen. Sie werden sich sicher gewundert haben, warum vorhin die Wendung »im Großen und Ganzen« verwendet wurde und es nicht einfach hieß: »Der Kontext entspricht dem Kontext des Erstellers.« Für das Ausführen in der Prozedur vorhandener DML-Kommandos auf bestehende Objekte ist das tatsächlich auch so einfach. Objektberechtigungen werden vom Ersteller einer gespeicherten Prozedur geerbt. Etwas anders sieht es mit weiteren Datenbankberechtigungen wie *CREATE TABLE* aus (also DDL-Kommandos). Diese stehen nicht automatisch zur Verfügung. Daran muss man denken, wenn in einer Prozedur beispielsweise temporäre Objekte angelegt werden sollen.

Ansonsten gilt, was bei den Sichten (Kapitel 13) ausführlich erklärt wurde: Entsteht durch das Aufrufen weiterer Prozeduren oder Sichten in der ausgeführten Prozedur eine Besitzrechtskette, dann werden die Berechtigungen für die unterliegenden Objekte nur dann geprüft, wenn ein Besitzerwechsel stattfindet.

In SQL Server 2005 können Sie bei Bedarf den Sicherheitskontext, der bei der Ausführung einer gespeicherten Prozedur verwendet werden soll, ganz präzise festlegen. Entweder beim Aufruf durch einen expliziten Kontextwechsel über das *Kommando EXECUTE AS* oder fest verdrahtet in der Prozedurdefinition über die *Klausel WITH EXECUTE AS.* Das zweite Verfahren wird auch als impliziter Kontextwechsel bezeichnet. Es stehen die folgenden Varianten zur Verfügung:

- **CALLER:** Der Kontext des Aufrufers wird benutzt.
- **SELF:** Der Kontext des Erstellers wird benutzt.
- **OWNER:** Der Kontext des aktuellen Besitzers wird benutzt.
- **»login_name«:** Es wird explizit ein Server-Benutzerkontext ausgewählt.

Im folgenden Beispiel wird die *OrderGetAll*-Prozedur so angelegt, dass beim Aufruf der Benutzerkontext des aktuell an der Sitzung angemeldeten Benutzers verwendet wird.

```
CREATE PROC OrdersGetAll
WITH EXECUTE AS CALLER
AS
SELECT * FROM Customers
ORDER BY [ID]
```

Listing 14.21 Prozedur mit implizitem Kontextwechsel

Einen expliziten Kontextwechsel stellt das Listing 14.22 vor. Vor dem Ausführen des *CREATE TABLE*-Befehls wird der Kontext auf denjenigen des Datenbankbesitzers eingestellt. Anschließend hebt der *REVERT*-Befehl diesen Kontext wieder auf und es gilt der ursprüngliche.

```
CREATE PROC OrdersGetAll
AS
EXECUTE AS USER = 'dbo'

CREATE TABLE #Customers ( ID int , Name_1 varchar(50), Name_2 varchar(50), City varchar(50) )

REVERT

… u.s.w
```

Listing 14.22 Expliziter Kontextwechsel in einer gespeicherten Prozedur

Ausnahmebehandlung

Eines ist ja klar: Fehlerbehandlung muss sein – *oder*? Sind Sie sicher, dass Sie in Ihrem SQL Server-Code Ausnahmebehandlung betreiben müssen? Sicher nicht in einfachen Abfragen, eher schon in gespeicherten Prozeduren. Wenn es in diesen aber um die Implementierung von Geschäftslogik geht, die sich vollständig relational beschreiben lässt, dann kann es gut sein, dass Sie auf Ausnahmebehandlung verzichten können. Schließlich sollten Programmierung und Logik auf dem Server sorgfältig durchgeführt und gründlich getestet sein, bevor eine Prozedur freigegeben wird. Ausnahmebehandlung ist nicht dazu da, *logische Fehler* abzufangen, schon gar nicht auf Datenbankebene. Und Sie werden bei der SQL Server-Entwicklung wirklich ausgesprochen selten mit Ressourcenfehlern zu tun haben (der Server läuft und läuft und läuft…). Falls diese doch einmal auftreten, dann sind sie in der Regel so fatal, dass Ihre gesamte Anwendung »etwas davon hat«: die Serververbindung bricht ab, die Datenbank ist beschädigt. So etwas muss sowieso Ihre Client-Anwendung abfangen.

Dennoch gibt es Argumente für den Einsatz von Ausnahmebehandlung. Ich habe ein paar davon zusammengestellt.

- **Zugriff auf externe Resourcen:** Greifen Sie aus Ihrer Transact-SQL-Programmierung heraus auf externe Ressourcen zu, dann können immer Fehler auftreten. Schicken Sie beispielsweise Remoteabfragen auf einen anderen Datenbankserver, dann kann dieser gerade offline sein. Möchten Sie einen Datenimport per *BULK INSERT* durchführen, dann kann die Textdatei im Dateisystem noch nicht generiert worden sein. Hier lohnt sich der Einsatz von Ausnahmebehandlung auf jeden Fall.
- **Aufruf »fremder« Methoden:** Verwenden Sie von einem anderen Team geschriebenen Code oder setzen Sie eingekauften Code ein – sei es in Form von .NET-Assemblies oder erweiterter gespeicherter Prozeduren (so genannte XPs – Gott stehe Ihnen bei!) –, werden Sie sich besser fühlen, wenn Sie die Aufrufe mit Ausnahmebehandlung versehen. Vor allem dann, wenn Sie keinen Zugriff auf den Quelltext der eingesetzten Routinen haben.
- **Explizite Transaktionen:** Setzen Sie in Ihrer Datenbank explizite Transaktionen, dann ist es sehr wahrscheinlich, dass Sie gleichzeitig auch Ausnahmebehandlung einsetzen werden. Transaktionen fangen auf SQL Server per Definition nur schwere Systemfehler ab. Um alles andere müssen Sie sich selbst kümmern und dazu sollten Sie strukturierte Ausnahmebehandlung einsetzen.
- **Schnittstellen zur Clientprogrammierung:** Hier benötigen Sie weniger die Funktionen zum Abfangen von Ausnahmen, als eher Möglichkeiten selbst serverseitige Ausnahmen zu generieren. Setzen Sie gespeicherte Prozeduren und benutzerdefinierte Funktionen als Schicht zwischen einer Datenbank und der Clientprogrammierung ein, dann macht es Sinn, Ausnahmen zu *generieren*, falls beispielsweise irgend ein Programmierer versucht, Ihre Prozeduren nicht korrekt zu verwenden. So können Sie den Clients bei falsch parametrisierten Prozeduraufrufen mal ordentlich die Meinung sagen.

TRY-CATCH-Blöcke

An dieser Stelle erwartet Sie eine gute Nachricht: Wenn Sie in Ihrem serverseitigen Code Ausnahmen behandeln möchten, geht das in SQL Server 2005 endlich strukturiert. Goodbye Inline-Fehlerbehandlung, goodbye *GOTO*, danke Microsoft! Dies ist die grundlegende Syntax für die strukturierte Fehlerbehandlung:

```
BEGIN TRY
     { sql_statement | statement_block }
END TRY
BEGIN CATCH
     { sql_statement | statement_block }
END CATCH
```

Die Regeln für die Anwendung sind einfach und überschaubar. Die wichtigste lautet: *TRY-CATCH*-Blöcke dürfen nur in Batches, gespeicherten Prozeduren und Triggern eingesetzt werden (*tatsächlich* also nicht in benutzerdefinierten Funktionen). Zwischen dem *TRY*- und dem *CATCH*-Block dürfen keine weiteren Kommandos eingebaut werden und die *TRY-CATCH*-Konstruktion muss sich insgesamt in einem einzigen Batch befinden. Läuft der T-SQL-Programmcode im *TRY*-Block fehlerfrei, wird die Programmausführung mit den Befehlen hinter dem *END CATCH* ausgeführt, ansonsten erfolgt der Eintritt in den *CATCH*-Befehlsblock. Anschließend geht es auch dann mit den Befehlen hinter *END CATCH* weiter. In vielen Fällen wird die Prozedur in einem *CATCH*-Block allerdings über das *RETURN*-Kommando verlassen und dann wird der Rest des Codes natürlich nicht ausgeführt.

TRY-CATCH-Blöcke können ineinander verschachtelt sein. Tritt in einem inneren *CATCH*-Block ein Fehler auf, dann wird dieser zum übergeordneten *CATCH*-Block hochgereicht. Solch eine »Bubbling« tritt auch dann auf, wenn im *TRY*-Block eine Prozedur aufgerufen wird (die eine Prozedur aufruft, die eine Prozedur aufruft, ...), die selbst keine Fehlerbehandlung durchführt. In der ersten Prozedur *mit* Ausnahmebehandlung greift dann der *CATCH*-Block.

Es stellt sich die Frage, was denn überhaupt als behandlungswürdige Ausnahme gewertet wird. SQL Server liefert zu jedem (wirklich jedem) ausgeführten T-SQL-Kommando einen Ergebniscode zurück. Diese Codes sind in verschiedene Schwereklassen aufgeteilt. Tabelle 14.1 gibt eine Übersicht. Im Meldungsfenster des T-SQL-Editors sehen Sie bei einer aufgetretenen Ausnahme neben dem Fehlercode und der Beschreibung auch den Schweregrad – hier Ebene genannt:

```
Meldung 2745, Ebene 16, Status 2, Zeile 1
Der Prozess mit der ID 58 hat den Benutzerfehler 50000, Schweregrad 20, ausgelöst. Dieser Prozess
wird von SQL Server beendet.
Meldung 50000, Ebene 20, Status 1, Zeile 1
Schlimmer Fehler!
Meldung 0, Ebene 20, Status 0, Zeile 0
Für den aktuellen Befehl ist ein schwerwiegender Fehler aufgetreten. Löschen Sie eventuelle
Ergebnisse.
```

Diesem kleinen Beispiel kann man ein paar typische Fakten in Bezug auf SQL Server-Fehler entnehmen. Zum einen wird bei einem Fehler des Schweregrades 20 oder höher der Benutzerprozess – also die Datenbankverbindung – aufgehoben. Der Client muss diese bei Bedarf selbsttätig wiederherstellen.

Außerdem führen Fehler mit einem Schweregrad von 19 oder höher dazu, dass SQL Server die Verarbeitung des Befehlsstapels abbricht. Bereits ausgelieferte Teilergebnisse machen keinen Sinn und Transaktionen auf dem Server werden automatisch zurückgesetzt.

Und die Antwort auf die ursprüngliche Frage lautet: Bei Meldungen, die einen Schweregrad über 10 und kleiner als 20 aufweisen, wird der *CATCH*-Block aufgerufen. Ansonsten nicht. Bei Fehlern ab Schweregrad 20, welche die Verbindung beenden, wird es erst gar nicht mit *CATCH* versucht und Fehler unter 10 sind keine wirklichen Fehler sondern Informationsmeldungen.

Klasse	Bedeutung
0-10	Dies sind reine Informationsmeldungen.
11-16	Benutzerfehler: fehlerhafte Verwendung von T-SQL-Befehlen oder fehlerhafte Programme.
17	Mangelnde Ressourcen: Ein Befehl kann nicht ausgeführt werden, weil eine SQL Server-Ressource ausgeschöpft ist. Beispiele sind fehlender Speicherplatz oder keine Verfügbarkeit von Sperren (eine interne SQL Server-Ressource).
18	Mittelschwerer interner Fehler: Problem in der SQL Server-Software. Die Verarbeitung kann dennoch fortgesetzt werden.
19	Schwerer interner Fehler: Ein internes Problem führt dazu, dass die Verarbeitung abgebrochen wird.
20-25	Fataler Fehler: Dabei handelt es sich um schwere Integritätsprobleme in der Datenbank oder um einen schweren Prozessfehler (durch das Betriebssystem ausgelöst). Fatale Fehler führen dazu, dass die Verbindung zum Server unterbrochen wird.

Tabelle 14.1 Schwereklassen von SQL Server-Fehlermeldungen

Ein Beispiel für die Ausnahmebehandlung in einer gespeicherten Prozedur wird im nächsten T-SQL-Skript vorgestellt. Es geht darum, einen neuen Artikel in die Datenbank aufzunehmen. Dies geschieht über das Eintragen eines Wertes in die Tabelle *Articles*. Verschiedene Dinge könnten an dieser Stelle schiefgehen: Der Artikelcode ist bereits in der Datenbank vorhanden, die Katalog-*ID* existiert nicht, es wird ein ungültiger Mehrwertsteuersatz übergeben und so weiter. In der Prozedur wird daher das *INSERT* in eine Ausnahmebehandlung eingepackt.

```
CREATE PROCEDURE ArticleInsert
(
   @CatalogID int,
   @Code varchar(50),
   @Name varchar(200),
   @DescriptionShort varchar(2000),
   @Price smallmoney,
   @Tax decimal(4,2),
   @Freight smallmoney,
   @Active bit,
   @ID int OUTPUT
)
AS
```

```
SET NOCOUNT ON

BEGIN TRY

   INSERT Articles
      (CatalogID, Code, Name, DescriptionShort, Price, Tax, Freight, Active)
   VALUES
      (@CatalogID, @Code, @Name, @DescriptionShort, @Price, @Tax, @Freight, @Active)
   SET @ID = @@IDENTITY

END TRY

BEGIN CATCH

   SET @ID = -1

END CATCH

SET NOCOUNT OFF
```

Listing 14.23 Prozedur mit TRY-CATCH-Block

Der nachstehende Codeausschnitt ruft die Prozedur *ArticleInsert* auf. Dabei wird eine *Article-ID* übergeben, die nicht in der Datenbank vorhanden ist. Ohne Ausnahmebehandlung würde der durch das fehlgeschlagene *INSERT* generierte Fehler von der Prozedur an den umgebenden Block durchgereicht. Die Variable *@intNewID* behält den Wert *NULL,* daher liefert *PRINT* kein Ergebnis. Bei einem Aufruf der Prozedur aus ADO.NET heraus käme es zu einer entsprechenden Ausnahme auf dem Client. Die Ausnahmebehandlung mit *TRY-CATCH* bewirkt, dass sowohl der T-SQL-Code als auch eine ADO.NET-Programmierung ungeachtet eines fehlgeschlagenen *INSERT* weiterlaufen können und der Parameter *@ID* den Wert -1 erhält.

```
DECLARE @intNewID int
EXEC
dbo.ArticleInsert 200, '99999', 'Hokkaido', '...', 2.5, .16, 0.5, 1, @intNewID OUTPUT
PRINT @intNewID
```

Listing 14.24 Aufruf, der einen Fehler liefert

Eine Fehlerbehandlungsroutine im *CATCH*-Block benötigt Informationen darüber, was eigentlich schief-gegangen ist. T-SQL stellt ein paar Systemprozeduren zur Verfügung, mit denen man sich die entsprechenden Werte besorgen kann. Dies funktioniert *ausschließlich* im *CATCH*-Block.

- **ERROR_NUMBER()** liefert die Fehlernummer.
- **ERROR_SEVERITY()** liefert den Schweregrad.
- **ERROR_PROCEDURE()** liefert den Namen der betroffenen Prozedur (des Triggers).
- **ERROR_LINE()** liefert die Zeilennummer.
- **ERROR_MESSAGE()** liefert den Text der SQL Server-Meldung.

Mithilfe dieser Funktionen lässt sich der *CATCH*-Block der Beispielprozedur entsprechend erweitern, um über einen Returncode eine genauere Aussage zum aufgetretenen Fehler zu liefern.

```
   …
   DECLARE @ReturnCode int
   SET @ReturnCode = 0

   …

   BEGIN CATCH

      SET @ID = -1

      IF ERROR_NUMBER() = 515
         -- NOT NULL - Verletzung
         SET @ReturnCode = -1

      IF ERROR_NUMBER() = 547
         -- CONSTRAINT - Verletzung
         SET @ReturnCode = -2

      IF ERROR_NUMBER() = 8152
         -- Zeichenfolge zu lang - Verletzung
         SET @ReturnCode = -3

   END CATCH

   RETURN @ReturnCode
```

Listing 14.25 CATCH-Block zur Auswertung des aufgetretenen Fehlers

Für die Auswertung des Returncodes muss der Aufruf der Prozedur in T-SQL ein wenig angepasst werden.

```
DECLARE
   @intNewID int,
   @intRetCode int

EXEC
   @intRetCode =  dbo.ArticleInsert 1, '99999', 'Hokkaido', '...', 1, 0.16, 1, 1,
   @intNewID OUTPUT

PRINT @intRetCode
PRINT @intNewID
```

Listing 14.26 Aufruf mit Abfangen des Returncodes

Leider sind die T-SQL Fehlercodes in den meisten Fällen nicht differenziert genug, um eine präzise Analyse der aufgetretenen Fehlersituation zu ermöglichen. Einen mehr oder weniger (eher weniger) eleganten Workaround stellt die Untersuchung der Variablen *ERROR_MESSAGE()* dar. In dieser finden Sie mit etwas Glück die Namen der an einem Fehler beteiligten Objekte, Spaltennamen, Bezeichnungen der *CHECK*-Constraints und so weiter. Ein Beispiel dafür sehen Sie im folgenden T-SQL-Codeausschnitt, der die Ausnahmebehandlung in der Prozedur *ArticleInsert* noch einmal erweitert.

```
BEGIN CATCH
      SET @ID = -1
```

```
      IF ERROR_NUMBER() = 515
         -- NOT NULL – Verletzung
         BEGIN
            -- Code darf nicht NULL sein
            IF ERROR_MESSAGE() LIKE '%Code%' SET @ReturnCode = -101
            -- Name darf nicht NULL sein
            IF ERROR_MESSAGE() LIKE '%Name%' SET @ReturnCode = -102
            -- Price darf nicht NULL sein
            IF ERROR_MESSAGE() LIKE '%Price%' SET @ReturnCode = -103
            -- und so weiter...
         END

      IF ERROR_NUMBER() = 547
         -- CONSTRAINT - Verletzung
         BEGIN
            -- Ungültiger Katalog
            IF ERROR_MESSAGE() LIKE '%FK_Articles_Catalogs%' SET @ReturnCode = -201
            -- Ungültiger Steuersatz
            IF ERROR_MESSAGE() LIKE '%FK_Articles_Taxes%' SET @ReturnCode = -202
            -- Preis darf nicht negativ sein
            IF ERROR_MESSAGE() LIKE '%CK_Price%' SET @ReturnCode = -203
            -- und so weiter...
         END

      IF ERROR_NUMBER() = 8152
         -- Zeichenfolge zu lang – Verletzung (keine Details möglich…)
         SET @ReturnCode = -3
END CATCH
```

Listing 14.27 Untersuchung der Systemvariablen ERROR_MESSAGE()

ACHTUNG Falls Sie im TRY-Block dynamischen T-SQL-Code verwenden, der mit *EXEC*() ausgeführt wird, dann können Sie Ausnahmen aus diesem *nicht* im CATCH-Block behandeln!

RAISERROR

Nachdem Sie nun kennen gelernt haben, wie Sie mithilfe von *TRY-CATCH* Ausnahmen abfangen, die von SQL Server generiert werden, kehren wir nun den Spieß um. Es geht jetzt darum, wie Sie in T-SQL selbst Ausnahmen generieren können. Es ist von Ihrer Entwicklungsstrategie abhängig, auf welche Art und Weise Sie von SQL Server aus mit den Clients »über Probleme reden wollen«. Eine Strategie besteht in der Vermeidung von Ausnahmen, die an die Clientzugriffsschicht durchgereicht werden. Die Kommunikation findet über die Parameter und Returncodes von gespeicherten Prozeduren (T-SQL- oder .NET-basiert) statt. Eine andere Strategie besteht in der bewussten Verwendung von Ausnahmen für die Kommunikation mit den Clients. Anwendungen, die im .NET-Framework realisiert sind, legen die zweite Variante nahe, da die Ausnahmebehandlung von .NET sehr gut unterstützt wird.