# Kapitel 8

jetzt lerne ich

# Gespeicherte Prozeduren mit Transact-SQL

Im vorangegangenen Kapitel haben Sie gelernt, wie die Sprache Transact-SQL aufgebaut ist, woraus sie besteht und welcher Syntax sie folgt. In diesem Kapitel werden wir diese Sprache zum Erstellen von Datenbankobjekten benutzen.


Sie sollten Kapitel 7, in welchem die jetzt zum Einsatz kommenden Techniken erklärt wurden, nicht übersprungen haben, wenn Sie sich nun diesem Kapitel widmen.

Gespeicherte Prozeduren sind kleine Programme, die direkt auf dem Server gespeichert sind und dort ausgeführt werden. Diese können lediglich Daten zurückgeben (wie eine normale SELECT-Anweisung), Daten manipulieren (über die Verwendung von DML-Befehlen) oder auch vielfältige komplexe Aufgaben übernehmen. Jedenfalls sollten Sie bei der Entwicklung von C/S-Datenbanken darauf achten, so viel Funktionalität wie möglich – weg von der Client-Programmierung – in gespeicherte Prozeduren zu übertragen, weil

- gespeicherte Prozeduren am Server bekannt sind und nach dem Aufruf sofort ausgeführt werden können, sodass Performancevorteile gegenüber ad hoc abgesendeten SQL-Statements realisierbar sind.
- Programmlogik, die in gespeicherten Prozeduren abgebildet ist, nur einmal ausprogrammiert werden muss, unabhängig davon, wie viele verschiedene Frontend-Applikationen darauf zugreifen. Jede Programmlogik, die im Frontend programmiert wird, muss in jedem neuen Frontend neu ausprogrammiert werden.

Optimalerweise wird die Programmlogik also immer im Backend implementiert, das Frontend greift nur darauf zu. Das heißt nicht, dass beispielsweise VBA für die Programmierung nicht mehr verwendet werden soll. Primär sollte es aber bei datenbezogenen Operationen dafür verwendet werden, auf dem Server gespeicherte Prozeduren aufzurufen und deren Rückgabewerte auszulesen.


Sie benötigen eine Funktion, die die in der Tabelle *tblProjektarbeitszeit* eingetragenen Arbeitszeiten verbucht. Programmieren Sie diese in einem Frontend wie z.B. VBA oder VB aus, müssen Sie die gesamte Logik noch einmal ausprogrammieren, wenn Sie später für ein Web-Frontend dieselbe Funktionalität noch einmal benötigen.

Erstellen Sie aber für diese Aufgabe eine gespeicherte Prozedur, der als Parameter die für die Zeitverbuchung relevanten Daten wie Personal- oder Teamnummer, Projekt und Zeitaufwand übergeben werden, muss nur ein einziges Programm erstellt werden. Sie müssen in jedem möglichen Frontend nur noch den Aufruf der fertigen Prozedur umsetzen. Der Vorteil wird vor allem dann klar, wenn man den Aufwand berücksichtigt, der in jedem Programm allein für die Fehlerbehandlung benötigt wird.

Weitere Vorteile von gespeicherten Prozeduren gegenüber der Verwendung von separaten SQL-Anweisungen:

- Reduzierung des Netzwerkverkehrs: Statt viele einzelne Anweisungen über das Netzwerk zum Server zu schicken, können Benutzer für eine komplexe Aufgabe mit dem Aufruf einer einzigen Prozedur auskommen. Auf diese Art und Weise wird die Anzahl der zwischen dem Server und dem Client übermittelten Anforderungen stark reduziert.
- Besserer Überblick für Benutzer: Benutzer haben es bei der Erledigung ihrer Aufgaben nicht mit einer unüberschaubaren Vielzahl an Tabellen zu tun, sondern kommen beispielsweise mit einem Satz an Prozeduren aus, die sie gezielt einsetzen. Dadurch ist ein direkter Zugriff auf Tabellen nicht erforderlich.
- Zugriffssicherheit: Sie können einem Benutzer auch dann die Berechtigung geben, eine Prozedur auszuführen, wenn er auf die Tabellen und Sichten, auf die in der Prozedur verwiesen wird, keine Berechtigungen besitzt. Er kann damit lediglich jene Aktionen starten, die über die Prozeduren vorgesehen sind. Es kann zu keinem Direktzugriff auf Daten und damit zu keiner ungewollten Bearbeitung kommen.

# 8.1 Aufbau einer gespeicherten Prozedur

Eine gespeicherte Prozedur besteht aus beliebigen Transact-SQL-Anweisungen. In Kapitel 7 haben wir Transact-SQL-Anweisungsblöcke erzeugt und vom Query Analyzer aus gestartet. Diese Anweisungen sind aber auf dem Server nicht gespeichert worden. Wenn sie gespeichert werden, dann erfolgt das in einer SQL-Scriptdatei auf einem Client.

Diese Anweisungsblöcke können nun zu Prozeduren zusammengefasst und auf dem Server gespeichert werden.

Die Vorteile dieser Methode gegenüber eigenständigen Blöcken ist, dass

- Prozeduren jederzeit von einem Client aus aufgerufen werden können,
- Sie in programmierte Client-Server-Applikationen fix integriert werden können und
- da sie in einer Art kompilierter Form am Server gespeichert sind, so schneller ausgeführt werden als neu an den Server übertragene Anweisungsblöcke.

Prozeduren werden über das Kommando `CREATE PROCEDURE` erzeugt. Wenn benötigt, wird diese Anweisung von der Definition der Übergabeparameter gefolgt. Im Anschluss an das Schlüsselwort `As` werden die eigentlichen Programmblöcke der Prozedur geschrieben. In der Regel wird die Prozedur durch die Anweisung `Return` beendet. Mit `Return` wird die Prozedurausführung unmittelbar beendet. Diese Anweisung kann aber auch – beispielsweise in einem Bedingungsblock – innerhalb einer Prozedur verwendet werden, um die Ausführung vorzeitig zu beenden. Es wirkt dann wie ein `Exit Sub` in VB.

```
CREATE PROCEDURE prozedurname
    @var1 datentyp,
    @var2 datentyp,
    ...
As
    Anweisungen
Return
```


`Return` ist für das Beenden der Prozedur nicht notwendig, macht aber den Prozedurcode besser lesbar, vor allem wenn Sie mehrere Prozeduren über den Query Analyzer anlegen.

## 8.2 Erzeugen einer gespeicherten Prozedur

Eine gespeicherte Prozedur kann sowohl über ein SQL Server-Tool wie den Enterprise Manager oder den Query Analyzer als auch über verschiedene andere Tools erzeugt werden, die in der Lage sind, SQL-Anweisungen an den SQL Server zu übertragen. Theoretisch können Sie auch Notepad dazu verwenden. Jedoch haben die SQL Server-eigenen Editoren Funktionalitäten, die Ihnen das Schreiben von Prozeduren erleichtern.

In der Regel werden Sie für das Erstellen einer gespeicherten Prozedur den Enterprise Manager oder den Query Analyzer – wie wir ihn auch im vorherigen Kapitel eingesetzt haben – verwenden.

### 8.2.1 Eine gespeicherte Prozedur über den Enterprise Manager anlegen

Eine übersichtliche Methode zur Erstellung von gespeicherten Prozeduren ist die Verwendung des Enterprise Managers. Hier haben Sie alle gespeicherten Prozeduren sowie auch alle anderen Datenbankobjekte im Überblick und im direkten Zugriff. Sie können hier allerdings eine gespeicherte Prozedur nicht testen.

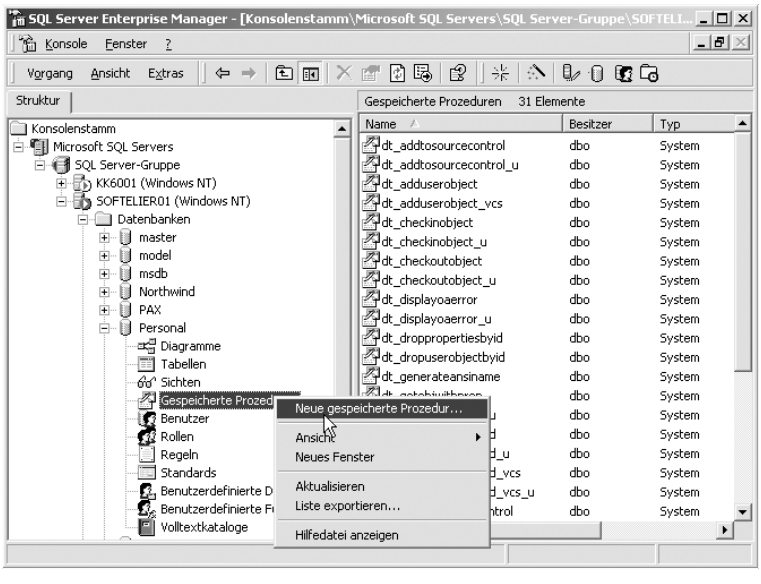



*Abb. 8.1: Gespeicherte Prozedur über den Enterprise Manager anlegen*

1. Starten Sie dazu den Enterprise Manager und erweitern Sie die Ordnerstruktur bis hin zur gewünschten Datenbank. Wählen Sie dort den Eintrag

*Gespeicherte Prozeduren* aus. Bereits gespeicherte Systemprozeduren werden im rechten Fenster angezeigt.

2. Klicken Sie mit der rechten Maustaste auf den Eintrag *Gespeicherte Prozeduren* und wählen Sie im Kontextmenü den Befehl NEUE GESPEICHERTE PROZEDUR ... aus.

Es erscheint das Eingabefenster für gespeicherte Prozeduren. Auch hier ist der SQL-Befehl für die Anlage der Prozedur `CREATE PROCEDURE [OWNER].[PROCEDURE NAME] AS` bereits erfasst.

Wenn Sie an dieser Stelle gespeicherte Prozeduren erstellen möchten, vergrößern Sie sinnvollerweise das standardmäßig sehr kleine Fenster.

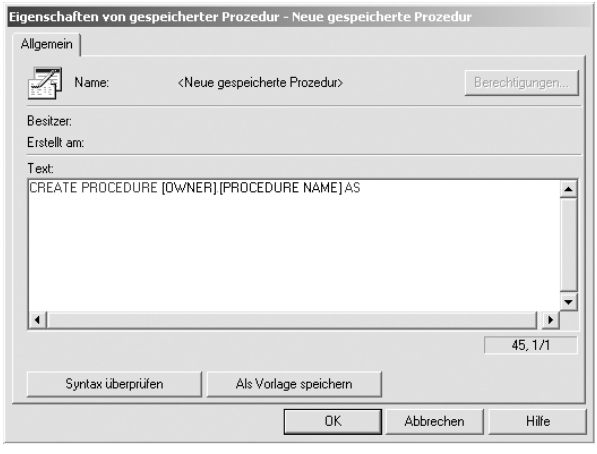



*Abb. 8.2: Eingabefenster für gespeicherte Prozeduren im Enterprise Manager*

Neben dem Erfassen der Prozedur stellt Ihnen dieser Dialog folgende Möglichkeiten zur Verfügung:

- *Syntax überprüfen*: Über diese Schaltfläche können Sie prüfen, ob die Syntax der Prozedur fehlerfrei ist. Hier werden nur Formalismen überprüft. Etwa, ob für jedes `Begin` auch ein `End` gesetzt ist, Spaltennamen in den angegebenen Tabellen richtig sind, Befehle und Funktionen sowie SQL-Anweisungen die korrekte Schreibweise und die richtige Anzahl an Parametern haben und Ähnliches.

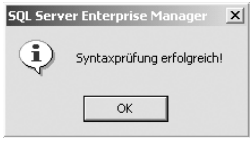
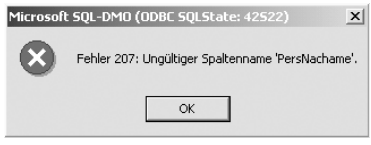



*Abb. 8.3: Erfolgreiche Syntaxüberprüfung*


Verlassen Sie sich aber nicht darauf, dass eine erfolgreiche Syntaxüberprüfung auch der Garant für eine fehlerfreie Prozedur ist. Nicht nur, dass Fehler in der Programmlogik selbstverständlich nicht überprüft werden können. Es werden auch nicht alle Fehler entdeckt. Wird beispielsweise richtig erkannt, dass ein Spaltenname ungültig ist, so evtl. nicht, wenn ein Tabellenname ungültig ist. Und dann wird natürlich auch der logischerweise nicht gültige Spaltenname nicht kritisiert.

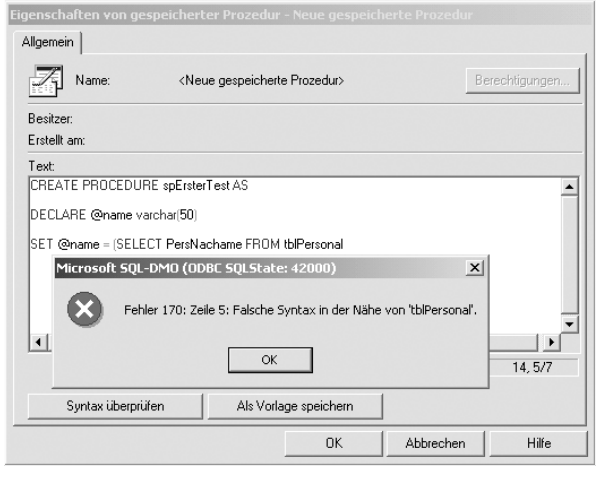
Ein nicht gültiger Spaltenname wird erkannt und als Fehler gemeldet. Leider wird hier nicht wie bei anderen Fehlermeldungen angezeigt, in welcher Zeile der Prozedur sich der Fehler befindet.

*Abb. 8.4: Ungültiger Spaltenname*




Wird ein echter Syntaxfehler entdeckt – im nachfolgenden Beispiel fehlt die rechte schließende Klammer der Unterabfrage – wird er, wie die nachfolgende Grafik zeigt, mit der Zeilennummer angegeben. Aufgrund dieser Zeilennummer kann der Fehler – auch wenn er nicht immer leicht erkannt wird – zumindest sehr einfach örtlich eingegrenzt werden. Wenn Sie sich mit dem Cursor im Prozedur-Eingabefenster bewegen, werden die aktuelle Zeile sowie die Gesamtanzahl an Zeilen im rechten unteren Fenstereck angezeigt.

*Abb. 8.5: Fehlermeldung mit Zeilenanzeige*



- *Als Vorlage speichern*: Mit dieser Option können Sie die Standardvorlage, wonach beim Erstellen einer neuen Prozedur der Prozedurheader wie beschrieben angezeigt wird, durch eine andere ersetzt werden. Hier können Sie bereits Festlegungen treffen, die Sie in einer Prozedur ohnehin sonst jedes Mal manuell eingeben würden. Zum Beispiel möchten Sie fix den *dbo* als Besitzer, das Namespräfix *sp* sowie die Option `set nocount on` und das schließende `Return` in der Vorlage integriert haben.

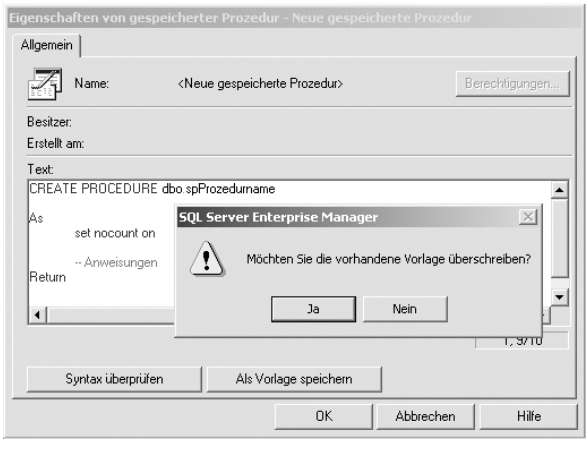



*Abb. 8.6: Vorlage für gespeicherte Prozedur anpassen*

Diese Anpassung gilt nicht nur für eine Datenbank, sondern für alle Prozeduren, die Sie mit dem Enterprise Manager, egal in welcher Datenbank und auch egal auf welchem Server, anlegen.

- *Prozedur speichern*: Eine neue Prozedur wird gespeichert, wenn Sie diese mit der Schaltfläche OK übernehmen. Bearbeiten Sie eine gespeicherte Prozedur, können Sie Änderungen auch mit ÜBERNEHMEN speichern. Dabei erfolgt jeweils die Syntaxüberprüfung, auch wenn Sie von Ihnen nicht extra angewählt worden ist.


Eine Prozedur kann – anders, als Sie es vielleicht von anderen Programmiersprachen her gewohnt sind – nur gespeichert werden, wenn sie keine Syntaxfehler enthält. Das bedeutet, dass vor allem Kontrollstrukturen immer vollständig ausprogrammiert werden müssen. Sehen Sie ggf. Dummyzeilen im Code vor, um unfertige Prozeduren dennoch speichern zu können. Oder möchten Sie Ihren bisherigen Code ungespeichert lassen, nur weil eine nette Kollegin mit Ihnen in die Kantine einen Kaffee trinken gehen möchte?

- *Berechtigungen vergeben*: Sobald Sie eine Prozedur mit OK gespeichert haben, können Sie Berechtigungen für diese vergeben. Das ist die Berechtigung *Execute*, mit der jeder direkt oder indirekt ausgestattet sein muss, der diese Prozedur ausführen möchte.

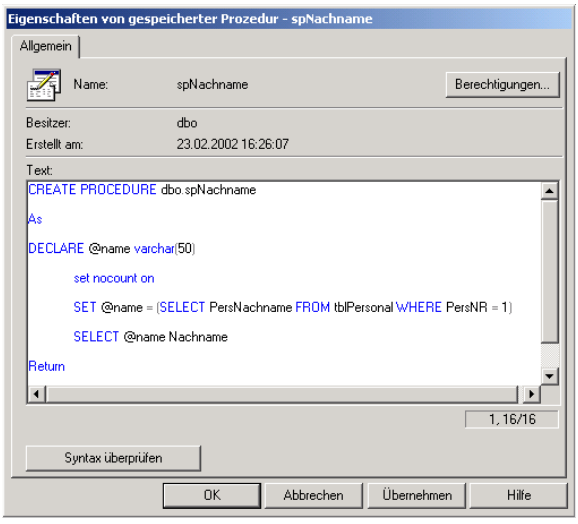



*Abb. 8.7: Fertige gespeicherte Prozedur*

Ist die Prozedur einmal gespeichert, kann sie nicht mehr als Vorlage umgesetzt werden. Über die Schaltfläche BERECHTIGUNGEN... gelangen Sie in den Dialog OBJEKTEIGENSCHAFTEN. Hier können Sie definieren, welchen Datenbankbenutzern bzw. Datenbankrollen Sie das Recht geben möchten, die Prozedur auszuführen. Auch hier gibt es die Möglichkeit, dies explizit zu erlauben oder auch zu verbieten. Dann darf der Benutzer diese Prozedur nicht ausführen, auch wenn er eine Rolle besitzt, die das schon darf.

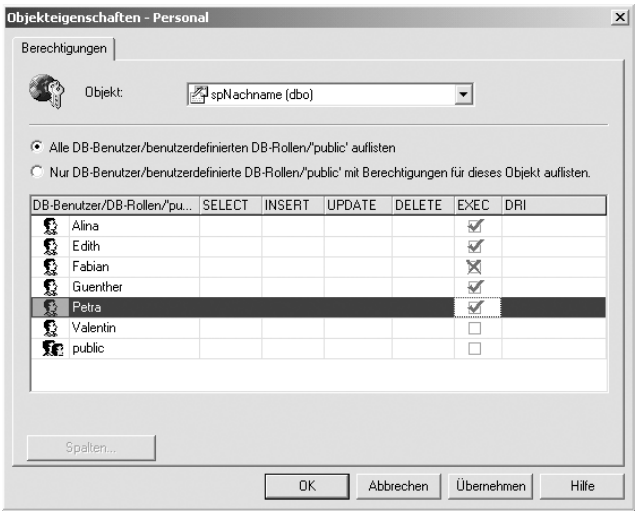



*Abb. 8.8: Berechtigungen für gespeicherte Prozeduren vergeben*

## 8.2.2 Eine gespeicherte Prozedur mit dem Query Analyzer anlegen

Als zweites SQL Server-eigenes Tool verwenden Sie wie gewohnt den Query Analyzer, um Prozeduren zu schreiben und diese danach auf dem Server zu speichern. Auch hier haben Sie ja die farbliche Kennzeichnung verschiedener Codeteile. Dies erleichtert das Lesen des Programmcodes ungemein.

Was unterscheidet das Anlegen einer Prozedur im Query Analyzer gegenüber der Variante mit dem Enterprise Manager?

- Sie schreiben die gesamte Prozedur inklusive der im Enterprise Manager vorgegebenen Anweisungen wie beispielsweise `CREATE PROCEDURE`. Wenn Sie aber nicht alles von Hand eingeben möchten, können Sie im Objektkatalog auf das Register VORLAGEN wechseln. Dort finden Sie Vorlagen für die verschiedensten Aufgaben mit Transact-SQL. Erweitern Sie den Ordner CREATE PROCEDURE und ziehen Sie den Eintrag *Create Procedure Basic Template* in das Abfragefenster rechts daneben. Neben dem Code für den Prozedurheader wird auch darüber das Codegerüst erstellt, mit dem Sie vor dem Erstellen der Prozedur über die Systemtabelle *sysobjects* auslesen, ob die Prozedur mit dem angegebenen Namen bereits existiert. Ist dies der Fall, wird sie vor dem neuerlichen Erstellen gelöscht. Außerdem wird nach dem Prozedurteil jener Code erstellt, mit dem Sie die fertige Prozedur später starten und damit testen können.

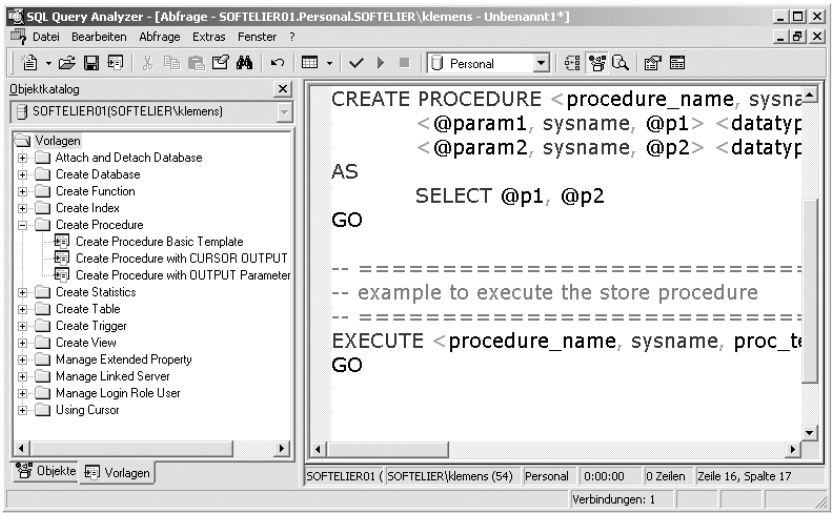
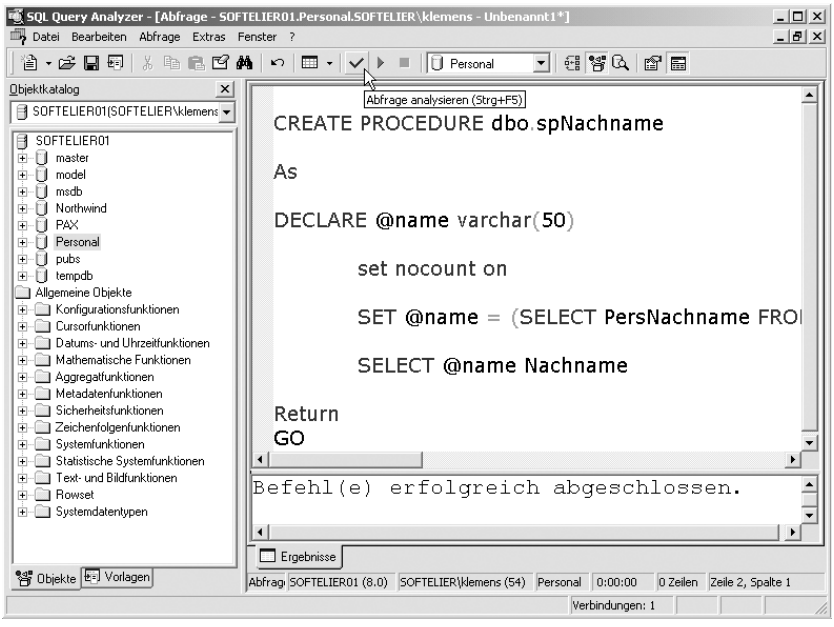



*Abb. 8.9: Prozedur über Vorlage erstellen*


Als enormer Vorteil des Query Analyzers gegenüber dem Enterprise Manager kann die Fähigkeit gewertet werden, Prozeduren über die Anweisung EXECUTE auszuführen. Damit wird ein unmittelbarer Test der soeben erstellten Prozedur ermöglicht.

- Auch im Query Analyzer haben Sie die Möglichkeit, die Syntax zu überprüfen. Wählen Sie dazu das Symbol mit dem blauen Haken oder verwenden Sie die Tastenkombination Strg + F5. Die Erfolgs- oder Misserfolgsmeldung wird sofort im Ergebnisfenster angezeigt.

*Abb. 8.10: Analysieren der Prozedur-Syntax*




- Das Speichern der Prozedur erfolgt wie gewohnt über die Schaltfläche F5 oder das Symbol mit dem grünen Rechtspfeil. Achten Sie dabei darauf, dass der Prozedurtext im Fenster markiert ist, wenn noch weitere Anweisungen davor oder danach erfasst sind und Sie nur die Prozedur anlegen möchten.


Wenn Sie eine bereits angelegte Prozedur verändern, können Sie diese natürlich nicht mehr über die Anweisung `CREATE PROCEDURE` ablegen. Dies führt zu einem Fehler, da das Objekt bereits besteht. Ändern Sie daher die Anweisung in `ALTER PROCEDURE`.


Ein Fehler, der sehr häufig gemacht wird – und ab und zu auch einem Profi passiert –, ist, dass man vergisst, die richtige Datenbank auszuwählen, und dann die Prozedur in der falschen Datenbank anlegt. Dies ist vorzugsweise die *master*, denn diese ist standardmäßig nach dem Öffnen des Query Analyzer ausgewählt. Denn oft sind Entwickler in der Praxis auch Mitglieder der Systemrolle *System Administrators* und haben damit auch das Recht dazu, in der *master* neue Objekte anzulegen. Damit Sie nicht vergessen, in der Symbolleiste die richtige Zieldatenbank auszuwählen, können Sie in Ihr Script zu Beginn den Eintrag `use Datenbankname` aufnehmen.

Es ist nicht wie bei Tabellen möglich, den Datenbanknamen in die `CREATE`-Anweisung aufzunehmen. `CREATE PROCEDURE Personal.dbo.spProzedur` führt zu einem Fehler.

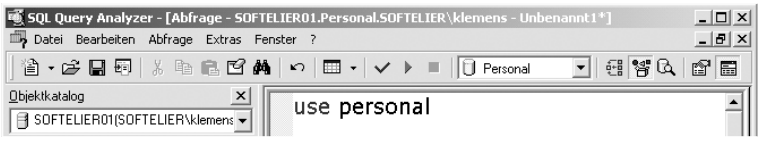



*Abb. 8.11: Richtige Datenbank auswählen*

- Ein meiner Meinung nach immenser Vorteil der Erstellung über den Query Analyzer ist, dass Sie den Prozedurcode jederzeit als SQL-Script speichern können. Sie können damit auch unfertige Codefragmente zwischendurch speichern. Und auf dem Server wird die Prozedur erst angelegt, wenn sie fertig ist oder zumindest eine vollständige Syntax besitzt. Außerdem haben Sie damit zugleich ein Script, mit dem Sie die Prozedur jederzeit in jeder beliebigen Datenbank auf jedem beliebigen Server ohne Aufwand neu anlegen können.

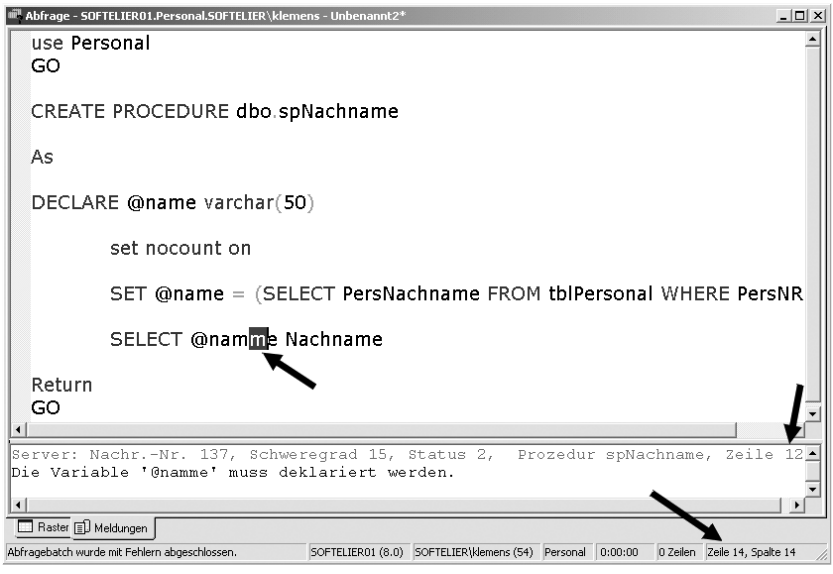

In meiner Praxis hat es sich bewährt, (eilige) Änderungen an der Datenbank über ein Script dem Kunden per E-Mail zu schicken. Den Query Analyzer zu starten, eine Scriptdatei zu öffnen und diese über einen Klick auszuführen, ist auch einem ungeschulten Benutzer mit entsprechender Anweisung zumutbar. Und Sie haben sich schon wieder eine Autofahrt erspart.

- Die aktuelle Zeile wird im rechten unteren Eck des Abfragefensters angezeigt. Dies hilft auch nach einer Fehlermeldung, die fehlerhafte Zeile aufzuspüren, da die Zeilennummer in einer Fehlermeldung stets mitangeführt wird.

  Sie sollten allerdings darauf achten, dass sich die Zeilenangabe im rechten unteren Eck immer auf die aktuelle Cursorposition in der Scriptdatei, die Zeilennummer in der Fehlermeldung sich allerdings auf die Zeile innerhalb

der Prozedur bezieht. Diese Angaben sind nicht identisch, wenn in der Scriptdatei vor der Prozedur noch andere Anweisungen vorkommen. Dennoch können Sie mit der Zeilenangabe der Fehlermeldung immer die betroffene Codezeile finden, auch wenn die Scriptdatei sehr umfangreich ist. Betrachten wir uns dazu das Beispiel in der nachfolgenden Grafik:

*Abb. 8.12: Zeilenanzeige im SQL-Script*




Vor der eigentlichen Prozedur steht die Anweisung, die Datenbank *Personal* auszuwählen. Danach muss die Anweisung `GO` verwendet werden, um den Batch zu beenden. Denn das Erzeugen der Prozedur muss stets in einem eigenen Batch innerhalb des Scripts erfolgen. Die Prozedur enthält einen Tippfehler in der Variablenbezeichnung `@name`. Wird das Script ausgeführt, wird ein Fehler in der Zeile 12 angezeigt. Zeile 12 ist von jener Position gerechnet, in der der Batch beginnt. Dies ist im Beispielsfall die Leerzeile nach der ersten `GO`-Anweisung. Sie gehen mit dem Cursor nun einfach eine Zeile, bevor der Batch beginnt, und stellen die Zeilennummer fest. (Hier ist es Zeile 2.) Diese Zeilennummer addieren Sie zur Zeilennummer in der Fehlermeldung und erhalten so die Zeile im Script, in der der Fehler steht. Sie bewegen den Cursor nun so lange, bis diese Zeilennummer rechts unten angezeigt wird. Dann haben Sie die Zeile gefunden. In unserem Beispiel ist dies die Zeile 14.

Zusammenfassend sind die Vorteile des Query Analyzer, dass man die Prozeduren zusätzlich auch als SQL-Script abspeichern kann. Damit hat man sie jederzeit (zusätzlich zur Datenbanksicherung) verfügbar, ohne sie am Ende des Entwicklungsprozesses erst generieren zu müssen. Nicht zu unterschätzen ist der Vorteil, dass Prozeduren im Query Analyzer auch getestet werden kön-

nen. Die Übersichtlichkeit kann auch dadurch erhöht werden, dass man Prozeduren, die sich gegenseitig aufrufen oder inhaltlich zu einem Aufgabenkomplex gehören, in einer Scriptdatei übersichtlich zusammenfassen kann.

### 8.2.3 Eine gespeicherte Prozedur mit einem externen Programm anlegen

Eine gespeicherte Prozedur kann wie erwähnt nicht nur mit den beim SQL Server mitgelieferten Programmen erzeugt werden. Auch andere Programme, die mit dem SQL Server kommunizieren können, sind dazu in der Lage. Ein gutes Beispiel dafür ist Access. Hier können Prozeduren ab der Version 2000 über so genannte Access-Projekte erzeugt werden. Diese Access-Projekte sind direkt mit einer SQL Server-Datenbank zu verbinden. Auch in älteren Access-Versionen könnten Sie das theoretisch über Pass-Through-Abfragen über ODBC, aber das wäre wohl etwas umständlich.

In einer Access-Projektdatei, die mit der SQL-Server-Datenbank verbunden ist, können Sie eine Prozedur direkt wie ein anderes Access-Datenbankobjekt erstellen. In der Version Access 2000 finden Sie direkt im Datenbankfenster in der linken Leiste den Eintrag *Gespeicherte Prozeduren*. Nun können Sie rechts daneben direkt auf den Eintrag *Erstellt eine Prozedur unter Verwendung des Designers* oder die Schaltfläche NEU klicken, um eine neue Prozedur zu erstellen. In der Eingabemaske werden der Prozedurheader sowie auskommentiert ein Variablenübergabeblock und das Deaktivieren der Ergebniszeilenanzeige mit `set nocount on` bereits angezeigt.

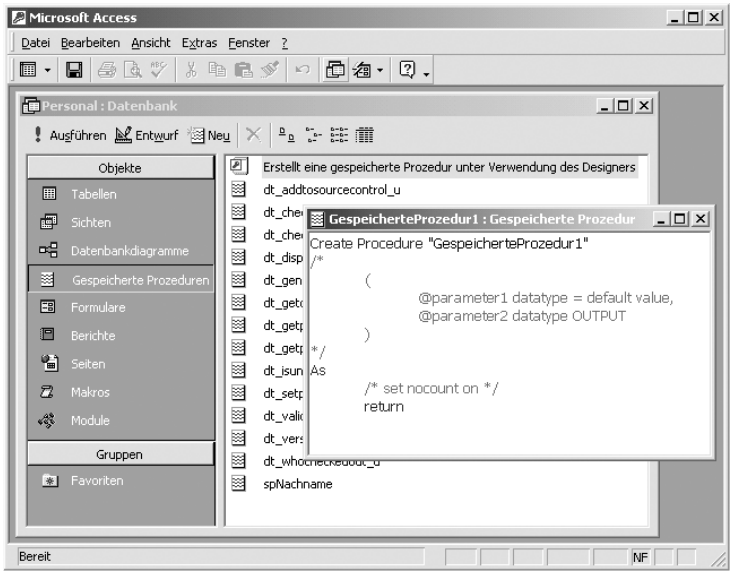



*Abb. 8.13: Gespeicherte Prozedur mit Access 2000-Projekt anlegen*

Auch hier gilt, dass Sie wie im Enterprise Manager nur syntaktisch korrekte und komplette Prozeduren speichern können. In der Version 2000 werden wie im Enterprise Manager und im Query Analyzer die einzelnen Syntaxelemente in verschiedenen Farben angezeigt. Dies macht die Eingabe ebenso leicht wie in den vorangegangenen Werkzeugen.


Wenn Sie einmal eine Installation mit dem »Mini-SQL Server« MSDE ohne die Werkzeuge Enterprise Manager und Query Analyzer vor sich haben, dann können Sie auf diese Variante mit Access zurückgreifen. Access kann auch manchmal die Rettung sein, wenn die erwähnten SQL Server-Tools aus einem anderen Grund nicht zur Verfügung stehen.


Die im nächsten Kapitel erläuterten und mit Prozeduren verwandten benutzerdefinierten Funktionen können mit Access 2000 noch nicht angelegt werden. Das ist dadurch erklärt, dass die erst mit SQL Server 2000 neu hinzugekommenen Funktionen zur Geburtsstunde von Access 2000 noch unbekannt gewesen sind.

Mit der aktuellen Access-Version 2002 hat Microsoft das Look and Feel einer Projektdatei (**.adp*) an eine herkömmliche Access-Datenbank (**.mdb*) angeglichen. Sichten, gespeicherte Prozeduren und benutzerdefinierte Funktionen sind unter der Objektbezeichnung *Abfragen* zu finden, und nicht mehr eigene Kategorien. Microsoft wollte damit wohl die Berührungsängste von Access-Benutzern mit dem SQL Server reduzieren. Diese Neuerung hat aber für Entwickler zwei Nachteile:

- Beim Erstellen einer neuen Prozedur über den Designer gelangen Sie zuerst in einen dem Access-Abfragefenster ähnlichen Editor. Um in die Texteingabe zu gelangen, wählen Sie keine Tabelle aus und wählen im Menü ANSICHT den Befehl SQL-ANSICHT.
- In der Angleichung des Editors hat Microsoft auf die für Entwickler so angenehmen Farbunterscheidungen im Editor vergessen, die bei der Vorversion noch implementiert gewesen sind. Der Entwickler sieht sich mit einer einheitlich schwarzen Schrift im Editor konfrontiert.

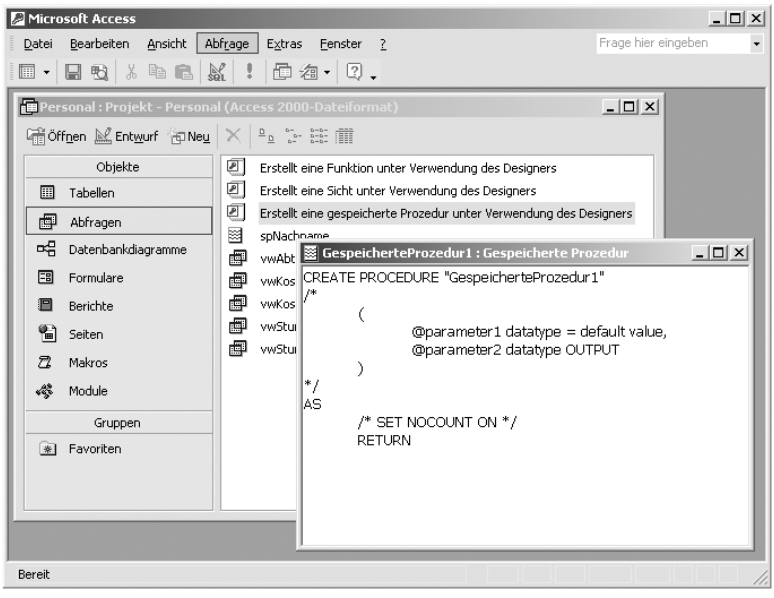



*Abb. 8.14: Gespeicherte Prozedur mit Access 2002-Projekt anlegen*

Der erste Nachteil lässt sich jedoch umgehen, wenn man nicht auf den Eintrag *Erstellt eine gespeicherte Prozedur unter Verwendung des Designers* klickt, sondern die links oberhalb positionierte Schaltfläche NEU bevorzugt. Denn dann kann im Dialog NEUE ABFRAGE direkt der Eintrag *Text-Gespeicherte Prozedur* ausgewählt werden, um den grafischen Editor zu umgehen.

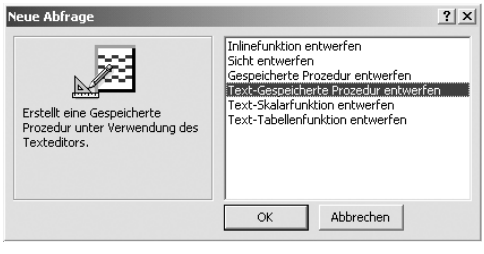



*Abb. 8.15: Gespeicherte Prozedur direkt im Texteditor anlegen*

Zusammenfassend sei gesagt, dass Access 2000/2002-Projekte eine sinnvolle Alternative zur Anlage von gespeicherten Prozeduren sind, wenn keine SQL Server-Werkzeuge zur Verfügung stehen.

## 8.3 Einfache gespeicherte Prozeduren

Wenn Sie sich nach diesem Überblick nun entschieden haben, welches Tool Sie für die Erstellung Ihrer Prozeduren verwenden möchten, können wir uns einigen Beispielen widmen und, beginnend mit leichten Anwendungsfällen, uns immer komplexeren Aufgabenstellungen nähern. Wir werden für die vorgestellten Beispiele aufgrund der gewählten Vorteile den Query Analyzer verwenden.


Auf der Buch-CD finden Sie zu den Kapiteln 7 bis 9 jeweils eine SQL-Script-Datei, die alle verwendeten Beispiele enthält.

Gespeicherte Prozeduren können aus einfachen SQL-Anweisungen bestehen und so Zeilen aus Tabellen als Ergebnis liefern.

Der Vorteil einer solchen Prozedur gegenüber einer zur Laufzeit eingegebenen SQL-Anweisung liegt in der schnelleren Ausführung der Prozedur, da die Analyse des Statements entfällt.

```
CREATE PROCEDURE spMitarbeiterliste
AS
    set nocount on

    SELECT p.PersNr, p.PersNachname, p.PersVorname,
    p.PersGeschlecht,p.PersEintritt, a.AbtName
    FROM tblPersonal p INNER JOIN tblAbteilungen a
    ON p.PersAbteilung = a.AbtNr
Return
```

Wird in einer Prozedur eine `SELECT`-Anweisung verwendet, gibt diese das Ergebnis zurück. Eine Ausnahme bildet eine `SELECT`-Anweisung zum Befüllen von Variablen, wie im nachfolgenden Beispiel verwendet wird. Hier werden die Personalnummer, der Nachname, der Vorname, das Eintrittsdatum und der Abteilungsname jenes Mitarbeiters, der das höchste Gehalt bezieht, in Variablen gespeichert. Diese `SELECT`-Anweisung ist als Ergebnis direkt nicht sichtbar. Der Mitarbeiter mit dem höchsten Gehalt wird in diesem Beispiel mit Hilfe einer Unterabfrage ermittelt.

```
CREATE PROCEDURE spBestverdiener
AS
DECLARE @nr int
DECLARE @nachname varchar(50), @vorname varchar(50)
DECLARE @eintritt smalldatetime, @abteilung varchar(30)
    set nocount on
```

```
    SELECT @nr = p.PersNr,
           @nachname = p.PersNachname,
           @vorname = p.PersVorname,
           @eintritt = p.PersEintritt,
           @abteilung = a.AbtName
    FROM tblPersonal p INNER JOIN tblAbteilungen a
    ON p.PersAbteilung = a.AbtNr
    WHERE p.PersMonatsgehalt = (SELECT
                                MAX(PersMonatsgehalt)
                                FROM tblPersonal)

    SELECT @nr As Nr, @nachname As Nachname,
           @vorname As Vorname, @eintritt As Eintritt,
           @abteilung As Abteilung
Return
```

Eine Prozedur wird über die Anweisung `EXECUTE Prozedurname` ausgeführt. `EXECUTE` kann durch `EXEC` abgekürzt werden. Diese ist vergleichbar mit dem Schlüsselwort `TRANSACTION`, das ebenfalls mit `TRAN` abgekürzt werden kann. Der Anweisung `EXECUTE` folgt der Name der Prozedur. Wie bei der Angabe eines Tabellennamens in der `FROM`-Klausel einer SQL-Anweisung kann der Name der Prozedur mit dem vollen Bezeichner angegeben werden. Dies ist dann notwendig, wenn sich die Prozedur in einer anderen Datenbank als der gerade verbundenen befindet. Sie können zum Starten der Beispielprozedur eine der beiden nachfolgenden Anweisungen verwenden:

```
EXEC spBestverdiener
EXEC Personal.dbo.spBestverdiener
```

Die Prozedur liefert folgendes Ergebnis:

```
Nr  Nachname   Vorname   Abteilung        Eintritt
--- ---------- --------- ---------------- -----------
101 Obermann   Gernot    Geschäftsführung 01.10.1998
```

## 8.4 Gespeicherte Prozeduren mit Eingabeparametern

In der Praxis kommen Prozeduren, die ohne Eingabeparameter auskommen, eher selten vor. In der Regel benötigt eine Prozedur einen oder mehrere Parameter, um ihre Aufgabe erledigen zu können. Dies ist mit Funktionen vergleichbar, die es auch mit und ohne Eingabeparameter gibt.


Oft werden Eingabeparameter auch als Übergabeparameter bezeichnet, weil sie einer Prozedur oder Funktion beim Aufruf übergeben werden. Da es bei gespeicherten Prozeduren sowohl Eingabe- als auch Ausgabeparameter gibt, wird meist der Begriff eines Eingabeparameters zur exakten Bestimmung verwendet.

Manche Funktionen kommen ohne Eingabeparameter aus, Sie wissen auch so, was sie zu tun haben. Beispiele dafür sind die Funktion `GETDATE()` oder auch `CURRENT_USER`. Was die augenblickliche Systemzeit ist, ist auch ohne weitere Zusatzinformation eindeutig bestimmt, weshalb die Funktion `GETDATE()` keine Information mehr benötigt. Ebenso wird mit `CURRENT_USER` der Name des aktuellen Benutzers zurückgegeben. Auch hier ist keine weitere Information erforderlich, die von der Funktion benötigt wird, um ihre Aufgabe zu lösen.

Andere Funktionen, wie z.B. die Funktion `DATEPART()`, können nur dann ein Ergebnis liefert, wenn man ihr beim Aufruf mitteilt, welchen Teil von welchem Datum man benötigt.

Nach demselben logischen Schema können einer Prozedur ein oder mehrere Eingabeparameter übergeben werden. Diese werden am Beginn der Prozedur zwischen dem Prozedurnamen und dem Schlüsselwort `As`, nach dem die eigentliche Prozedur beginnt, angegeben. Die Angabe gleicht der Deklaration von Variablen. Auch die Namen von Eingabeparametern beginnen mit einem `@`. Bei der Definition von Eingabeparametern wird lediglich das Schlüsselwort `DECLARE` weggelassen.


Achten Sie darauf, dass Sie bei der Verwendung mehrerer Eingabeparameter diese durch Kommata voneinander trennen. Dies wird leicht übersehen, da oft jeder Parameter in einer eigenen Zeile definiert wird.

Ein Eingabeparameter wird wie folgt definiert:

```
@parametername datentyp [=standardwert]
```

Im nachfolgenden Beispiel wird einer Prozedur, welche Mitarbeiterinformationen bereitstellen soll, die Personalnummer jenes Mitarbeiters übergeben, über den wir diese Informationen haben möchten. Nur durch diesen Eingabeparameter »weiß« die Prozedur, welchen Mitarbeiter sie auswählen soll.

```
CREATE PROCEDURE spMitarbeiterinfo
    @nr int
AS
DECLARE @nachname varchar(50), @vorname varchar(50)
DECLARE @eintritt smalldatetime, @abteilung varchar(30)
    set nocount on
```

```
    SELECT @nachname = p.PersNachname,
           @vorname = p.PersVorname,
           @eintritt = p.PersEintritt,
           @abteilung = a.AbtName
    FROM tblPersonal p INNER JOIN tblAbteilungen a
    ON p.PersAbteilung = a.AbtNr
    WHERE p.PersNr = @nr

    SELECT @nr As Nr, @nachname As Nachname,
           @vorname As Vorname, @abteilung As Abteilung,
           CONVERT(varchar, @eintritt, 104)  As Eintritt
Return
```

Innerhalb der Prozedur kann ein Eingabeparameter wie jede andere Variable verwendet und eingesetzt werden. Auch die Zuweisung eines neuen Wertes oder die Veränderung des alten Wertes ist möglich. In diesem Beispiel wird die übergebene Personalnummer in der `WHERE`-Klausel der `SELECT`-Anweisung verwendet, um damit den Mitarbeiter auszuwählen. Rufen Sie die Prozedur auf und übergeben Sie ihr die Personalnummer *755*.

```
EXEC spMitarbeiterinfo 755
```

liefert:

```
Nr  Nachname   Vorname   Abteilung        Eintritt
--- ---------- --------- ---------------- -----------
755 Prügger    Mathias   Vertrieb         01.03.1998
```


Da in der Prozedur die Option `SET NOCOUNT ON` verwendet wird, fehlt im Ergebnis der altbekannte Hinweis `(x row(s) affected)`.


Verwenden Sie in Prozeduren immer die Option `SET NOCOUNT ON`, da dies – wie im vorangegangenen Kapitel beschrieben – Performancevorteile bringt.

Eingabeparameter können mit Default-Werten versehen werden, wenn häufig dieselben Werte übergeben werden und Sie sich diese folglich im Aufruf ersparen möchten. Default-Werte werden den Eingabeparametern mit einem Gleichheitszeichen, in der Reihenfolge nach dem Datentyp, zugewiesen.

```
CREATE PROCEDURE spParameterTest
    @var1 int,
    @var2 int = 20,
    @var3 decimal(2,1) = 5.5
As
...
```

Diese Prozedur kann nun ohne Angabe von Werten für die zweite und dritte Variable aufgerufen werden:

```
EXEC spParameterTest 50
```

Möchten Sie für den dritten Parameter nun dennoch einen Wert übergeben, müssen Sie eine der folgenden Varianten für den Aufruf verwenden:

- Übergabe von Werten durch Position: Sie vergeben auch für den zweiten Parameter einen Wert, um danach für den dritten in der richtigen Reihenfolge einen Wert eingeben zu können:

  ```
  EXEC spParameterTest 50, 20, 8
  ```

- Verwenden von `DEFAULT`: Um einen danach kommenden Parameter vergeben zu können und dennoch für den vorangegangenen Parameter den Standardwert zu verwenden, benutzen Sie das Schlüsselwort `DEFAULT`.

  ```
  EXEC spParameterTest 50, DEFAULT, 8
  ```

- Übergabe von Werten durch Parameternamen: Um nicht an die definierte Reihenfolge gebunden zu sein, können Sie den Parameterwert auch direkt dem Parameternamen zuweisen:

  ```
  EXEC spParameterTest @var1 = 50, @var3 = 8
  ```

  Dadurch können Sie Parameter mit Default-Werten auslassen, auch wenn in der Reihenfolge nach diesen vorkommende Parameter mit Werten versorgt werden müssen.


Um den Prozeduraufruf und die Parameterübergabe so weit wie möglich zu vereinfachen, definieren Sie Parameter mit Standardwerten in der Prozedur immer zuletzt.

## 8.5 Ergebnisrückgabe von Prozeduren

Prozeduren können Ergebnisse auf verschiedene Art an den, der sie aufruft, zurückliefern. Das kann eine Benutzereingabe, ein externes Programm oder eine andere gespeicherte Prozedur sein. Welche Variante dabei zum Einsatz kommt, hängt auch von der Art des Aufrufs vom Frontend aus ab. Denn nicht jede Aufrufvariante kann jeden Rückgabewert auch entgegennehmen.

Folgende Rückgabevarianten sind möglich:

- `RETURN`-Anweisung
- `PRINT`-Anweisung
- `SELECT`-Anweisung
- `OUTPUT`-Parameter

Diese vier Varianten möchten wir Ihnen nun im Detail vorstellen und dabei auch auf ihre möglichen Einsatzbereiche eingehen.

### 8.5.1 Return

Die RETURN-Anweisung haben wir schon kennen gelernt, um einen Anweisungsblock oder eine Prozedur sofort zu beenden. Alle nach RETURN stehenden Anweisungen werden nicht mehr ausgeführt. RETURN beendet aber eine Prozedur nicht nur, sondern liefert einen ganzzahligen Wert. Wird nach RETURN kein Wert explizit angegeben, wird 0 zuückgeliefert. Der Rückgabewert kann entweder direkt oder indirekt über eine Variable zurückgegeben werden:

```
RETURN 5
RETURN @nummer
```

Der von der Prozedur zurückgelieferte Wert muss in einer Variablen aufgefangen werden. Diese Variable muss direkt beim Aufruf angegeben werden:

```
EXEC @ergebnisvariable = prozedurname param1, paramn
```

Das nachfolgende Beispiel zeigt die Verwendung von RETURN, um ein Ergebnis zurückzuliefern. Der Prozedur *spAnzInAbteilung* wird eine Abteilungsnummer übergeben. Die Prozedur stellt fest, wie viele Mitarbeiter in dieser Abteilung arbeiten, und gibt das Ergebnis über RETURN zurück.

```
CREATE PROCEDURE spAnzInAbteilung
    @abtnr int
AS
DECLARE @anz int
    set nocount on
    SET @anz = (SELECT COUNT(*)
                FROM tblPersonal
                WHERE PersAbteilung = @abtnr)
RETURN @anz
```

Um diese Prozedur aufzurufen, deklarieren Sie im Query Analyzer eine Variable vom Typ *Integer*. Diese geben Sie beim Aufruf über EXECUTE an. Sie enthält nach Beendigung der Prozedur den Ergebniswert, den Sie mit SELECT anzeigen können.

```
DECLARE @ergebnis int
EXEC @ergebnis = spAnzInAbteilung 10
SELECT @ergebnis As Anzahl
```

Das Beispiel liefert folgendes Ergebnis:

```
Anzahl
-----------
2
(1 row(s) affected)
```


Die Anzeige der betroffenen Zeilen erfolgt in diesem Beispiel, da die Anweisung `SET NOCOUNT ON` nur innerhalb der Prozedur gilt. Die Anzeige, dass eine Zeile betroffen ist, rührt aber von der außerhalb der Prozedur bei deren Aufruf verwendete Anweisung `SELECT @ergebnis As Anzahl` her.

Häufig wird diese Variante der Wertrückgabe nicht dazu verwendet, Daten aus der Datenbank auszugeben, sondern um eine Art Statusbericht zu liefern, ob die der Prozedur übertragenen Aufgaben erfolgreich erledigt werden konnten.

Diese Variante demonstriert das nächste Beispiel: Werden in der Tabelle *tblProjektarbeitszeit* Arbeitszeiten erfasst, hat dies Auswirkungen auf die für ein Projekt aufgewendeten Ist-Stunden sowie kalkulatorischen Projektkosten. Es müssen also einerseits die erfassten Stunden den Projektstunden zugebucht werden, andererseits diese Stunden mit dem kalkulatorischen Stundenlohn des jeweils betroffenen Mitarbeiters multipliziert und den kalkulatorischen Projektkosten zugeschlagen werden.

Nach der Eingabe ist die Prozedur aufzurufen. Als Parameter werden ihr die Projektnummer des Projektes, für das gearbeitet worden ist, die Mitarbeiternummer sowie die geleistete Stundenanzahl übergeben. Der kalkulatorische Stundenlohn des Mitarbeiters wird zunächst über eine Unterabfrage aus der Tabelle *tblPersonal* ausgelesen. Danach wird sofort geprüft, ob dieser *Null* ist. Ist dies der Fall, ist die übergebene Personalnummer ungültig oder der kalkulatorische Stundenlohn ist nicht erfasst. In beiden Fällen kann eine korrekte Verbuchung von Zeit und Kosten in der Tabelle *tblProjekte* nicht vorgenommen werden. Die Prozedur wird in diesem Fall beendet.

In diesem Beispiel wird als Ergebnis der Wert -1 zurückgegeben. Dieses Rückgabeergebnis sagt dem Benutzer oder dem aufrufenden Programm, dass der zuvor beschriebene Fall eingetreten ist. Konnte der Stundenlohn erfolgreich eruiert werden, wird der Projektdatensatz in der Tabelle *tblProjekte* upgedatet. Die Ist-Stunden werden um die geleisteten Stunden erhöht und die Projektkosten um den mit den Stunden multiplizierten Stundenlohn erhöht. Über die globale Systemvariable `@@rowcount` wird die Anzahl der von der `UPDATE`-Anweisung betroffenen Datensätze ausgelesen und der Variablen`@ergebnis` übergeben. Der Inhalt dieser Variablen wird mit `RETURN` zurückgeliefert. War die Prozedur erfolgreich, dann muss sie als Ergebnis den Wert 1 liefern. Liefert sie aber 0, dann war das Update nicht erfolgreich. Das kann z.B. daran liegen, dass die Projektnummer ungültig ist. Wichtig ist es jedoch, die Information zu haben, ob ein Vorgang erfolgreich abgeschlossen werden konnte, um richtig darauf reagieren zu können. Wir geben dieser Prozedur den Namen *spZeitbuchung*.

```
CREATE PROCEDURE spZeitbuchung
    @projekt int,
    @persnr int,
    @stunden decimal(3,1)
AS
DECLARE @stdsatz smallmoney
DECLARE @ergebnis int
    set nocount on

    SET @stdsatz = (SELECT PersKalkStdLohn
                    FROM tblPersonal
                    WHERE PersNr = @persnr)
    IF @stdsatz Is Null
        RETURN -1

    UPDATE tblProjekte
    SET ProjStundenIst = ProjStundenIst + @stunden,
        ProjKalkKosten = ProjKalkKosten + @stdsatz * @stunden
    WHERE ProjNr = @projekt

    SET @ergebnis = @@rowcount
RETURN @ergebnis
```

Um diese Prozedur zu testen, lesen wir den momentanen Wert der Ist-Stunden und der kalkulatorischen Kosten für das Projekt mit der Nummer *102* aus der Tabelle *tblProjekte* aus.

```
SELECT ProjBezeichnung, ProjStundenIst, ProjKalkKosten
FROM tblProjekte
WHERE ProjNr = 102
```

liefert:

```
ProjBezeichnung           ProjStundenIst  ProjKalkKosten
------------------------- --------------- ---------------
Standortplanung Berlin    718.0           57267.0000
(1 row(s) affected)
```

Nun rufen wir die neue Prozedur *spZeitbuchung* auf und übergeben ihr die Projektnummer 102, die Mitarbeiternummer 755 und 8 Stunden.

```
DECLARE @ergebnis int
EXEC @ergebnis = spZeitbuchung 102, 755, 8
SELECT @ergebnis As Anzahl
```

liefert:

```
Anzahl
-----------
1
(1 row(s) affected)
```

Der Ergebniswert 1 zeigt uns an, dass alles ordnungsgemäß funktioniert hat. Wir überprüfen das, indem wir die Werte für das Projekt 102 nochmals abrufen:

```
ProjBezeichnung           ProjStundenIst  ProjKalkKosten
------------------------- --------------- ---------------
Standortplanung Berlin    726.0           57907.0000
(1 row(s) affected)
```


Testen Sie diese Prozedur, indem Sie absichtlich eine ungültige Personalnummer oder eine ungültige Projektnummer übergeben. Kontrollieren Sie, ob für diese Fälle wirklich die Werte –1 und 0 zurückgegeben werden.

Abschließend lässt sich über die Ergebnisrückgabe mit RETURN Folgendes bemerken:

- Diese Methode ist nur anwendbar, wenn sich das Ergebnis durch eine ganze Zahl ausdrücken lässt.
- Die Variante kann verwendet werden, wenn der Aufruf der Prozedur über den Query Analyzer erfolgt oder über eine andere Prozedur.
- Verschiedene Frontend-Werkzeuge wie Visual Basic, VBA und Access können diesen Wert nicht entgegennehmen.
- Sie können immer nur einen einzelnen Wert als Ergebnis zurückliefern.

### 8.5.2 Print

Eine weitere Variante, die nur für den direkten Prozeduraufruf über den Query Analyzer einen Sinn macht, ist die Verwendung von `PRINT`. Die `PRINT`-Anweisung haben wir ja bereits im vorangegangenen Kapitel kennen gelernt.

Wir behalten das Beispiel bei und verändern es so, dass das Ergebnis über die `PRINT`-Anweisung ausgegeben wird. Dabei spielt es keine Rolle, ob Sie das Ergebnis über einen Zahlencode wie vorhin oder durch eine Meldung in Klartext ausgeben. Zur leichteren Unterscheidung haben wir der Prozedur den Namen *spZeitbuchungPrint* gegeben.

```
CREATE PROCEDURE spZeitbuchungPrint
    @projekt int,
    @persnr int,
    @stunden decimal(3,1)
AS
DECLARE @stdsatz smallmoney
DECLARE @ergebnis int
    set nocount on
```

```
    SET @stdsatz = (SELECT PersKalkStdLohn
                    FROM tblPersonal
                    WHERE PersNr = @persnr)

    IF @stdsatz Is Null
    BEGIN
        PRINT 'Die Personalnummer ' + @persnr +
              ' ist ungültig.'
        RETURN
    END

    UPDATE tblProjekte
    SET ProjStundenIst = ProjStundenIst + @stunden,
        ProjKalkKosten = ProjKalkKosten + @stdsatz * @stunden
    WHERE ProjNr = @projekt

    IF @@rowcount = 1
        PRINT 'Zeit und Kosten erfolgreich verbucht.'
    ELSE
        PRINT 'Projektnummer ungültig oder Verbuchung nicht
erfolgreich.'
RETURN
```

Der Vorteil dieser Variante liegt darin, dass Sie für den Aufruf mit der einfachen `EXECUTE`-Anweisung auskommen und keine Variable für die Aufnahme des Ergebnisses benötigen.

```
EXEC spZeitbuchungPrint 105, 182, 6
```

liefert:

```
Zeit und Kosten erfolgreich verbucht.
```

Vertippen wir uns absichtlich, und rufen die Prozedur mit der Personalnummer 181 auf, erhalten wir folgendes Ergebnis angezeigt:

```
Die Personalnummer 181 ist ungültig.
```

Testen Sie, ob Sie mit einer ungültigen Projektnummer die erwartete Fehlermeldung erhalten.


### 8.5.3 Select-Anweisung

Eine sehr beliebte Variante, vor allem wenn die Prozedur von einem programmierten Frontend aus aufgerufen wird, ist die Rückgabe des Ergebnisses mit einer `SELECT`-Anweisung.

Hier enthält die Prozedur – in der Regel ist dies meist die letzte Anweisung innerhalb der Prozedur – eine `SELECT`-Anweisung, die entweder beliebig viele

Zeilen und Spaltenwerte aus der Datenbank liefert oder den Inhalt einer oder mehrerer Variablen ausgibt.

Diese Variante haben wir bereits bei den Einstiegs-Beispielen im Abschnitt »Einfache gespeicherte Prozeduren« verwendet.

Das Zeitbuchungsbeispiel haben wir beibehalten und nun auf die Variante mit der `SELECT`-Ausgabe umgearbeitet. Dabei wird der Rückgabetext in der Variablen `@ergebnis` eingefügt. Am Ende der Prozedur wird der Variableninhalt mit `SELECT` ausgegeben. Diese Variante der Prozedur trägt für die leichtere Unterscheidbarkeit den Namen *spZeitbuchungSelect*.

```
CREATE PROCEDURE spZeitbuchungSelect
    @projekt int,
    @persnr int,
    @stunden decimal(3,1)
AS
DECLARE @stdsatz smallmoney
DECLARE @ergebnis varchar(100)
    set nocount on

    SET @stdsatz = (SELECT PersKalkStdLohn
                    FROM tblPersonal
                    WHERE PersNr = @persnr)

    IF @stdsatz Is Null
      SET @ergebnis = 'Die Personalnummer ' + CONVERT(varchar, @persnr)
+ ' ist ungültig.'
    ELSE
    BEGIN
        UPDATE tblProjekte
        SET ProjStundenIst = ProjStundenIst + @stunden,
            ProjKalkKosten = ProjKalkKosten + @stdsatz * @stunden
        WHERE ProjNr = @projekt

        IF @@rowcount = 0
            SET @ergebnis = 'Zeit und Kosten erfolgreich verbucht.'
        ELSE
         SET @ergebnis = 'Projektnummer ungültig oder Verbuchung nicht
erfolgreich.'

    END
    SELECT @ergebnis As Ergebnis
RETURN
```

Rufen wir diese Beispielvariante mit der Anweisung

```
EXEC spZeitbuchungSelect 106, 387, 4.5
```

auf, erhalten Sie folgende Anzeige im Query Analyzer:

```
Ergebnis
-----------------------------------------
Zeit und Kosten erfolgreich verbucht.
```


Testen Sie auch diese Prozedurvariante wieder mit verschiedenen Übergabewerten.


Diese Variante können Sie nicht verwenden, um Werte von einer Prozedur an eine andere zu übergeben.

Die SELECT-Variante ist in folgenden Situationen zum Einsatz geeignet:

- Die Prozedur soll mehrere Zeilen aus einer Datenbank als Ergebnis zurückgeben.
- Die Prozedur wird von einem Frontent-Programmiertool aufgerufen, das das Erstellen und Auslesen von Recordsets unterstützt. Das sind insbesondere Programmiertools wie VB, VBA, C und auch Java. Aber auch häufig im Web eingesetzte Tools wie ASP unterstützen dies.
- Sie können in Access-Projektdateien direkt mit Doppelklick gestartet werden und zeigen ein Ergebnis an.

### 8.5.4 Output-Parameter

Die wohl am universellsten einsetzbare Variante ist die Verwendung von Output-Parametern:

- Output-Parameter können über den Query Analyzer ausgelesen und ausgegeben werden.
- Über Output-Parameter können Prozeduren untereinander Werte übergeben.
- Output-Parameter können über Programmiertools, wie z.B. VB und VBA über Verwendung von ADO (ActiveX Data Objects) festgelegt werden.
- Über Output-Parameter können Sie keine Datenzeilen zurückliefern und deshalb kann das Ergebnis in Access-Projekten auch nicht direkt angezeigt werden.

Output-Parameter werden in einer Prozedur wie Eingabeparameter definiert, gefolgt vom Schlüsselwort OUTPUT. Verwenden Sie mehrere Output-Parameter, müssen Sie OUTPUT bei jedem Ausgabeparameter ergänzen.


Sie können Eingabe- und Ausgabeparameter in der Reihenfolge beliebig mischen. Jedoch ist es von der Übersicht her sinnvoller, zuerst die Eingabe- und danach die Ausgabeparameter zu definieren.

Sie können einen Output-Parameter mit einem Einkaufskorb vergleichen, den Sie jemandem zum Einkauf mitgeben, mit der Bitte, ihn mit bestimmten Dingen zu füllen. Nach dem Einkauf bekommen Sie ihn zurück und haben die gewünschten Artikel im Korb. Genauso funktioniert es mit Output-Parametern: Sie übergeben der Prozedur für jeden Output-Parameter eine Variable. Die Prozedur schreibt die Ergebniswerte in diese Variablen, und somit stehen diese Werte nach Beendigung der Prozedur zur Verfügung.

Unser Zeitverbuchungsbeispiel haben wir nun auch auf die Variante mit dem Output-Parameter umgebaut und als *spZeitbuchungOutput* angelegt.

```
CREATE PROCEDURE spZeitbuchungOutput
    @projekt int,
    @persnr int,
    @stunden decimal(3,1),
    @ergebnis varchar(100) OUTPUT
AS
DECLARE @stdsatz smallmoney
    set nocount on

    SET @stdsatz = (SELECT PersKalkStdLohn
                    FROM tblPersonal
                    WHERE PersNr = @persnr)

    IF @stdsatz Is Null
      SET @ergebnis = 'Die Personalnummer ' + @persnr + ' ist ungültig.'
    ELSE
    BEGIN
        UPDATE tblProjekte
        SET ProjStundenIst = ProjStundenIst + @stunden,
            ProjKalkKosten = ProjKalkKosten + @stdsatz * @stunden
        WHERE ProjNr = @projekt

        IF @@rowcount = 0
            SET @ergebnis = 'Zeit und Kosten erfolgreich verbucht.'
        ELSE
         SET @ergebnis = 'Projektnummer ungültig oder Verbuchung nicht
erfolgreich.'

    END
RETURN
```

Für den Aufruf dieser Prozedur muss eine Variable deklariert werden. Diese wird beim Aufruf übergeben wie Eingabeparameter auch. Jedoch genügt es nicht, dass der entsprechende Parameter in der Prozedur als Output-Parameter definiert worden ist. Auch die beim Aufruf übergebene Variable muss mit `OUTPUT` gekennzeichnet werden.

```
DECLARE @ergebnis varchar(100)
EXEC spZeitbuchungOutput 103, 602, 7, @ergebnis OUTPUT
SELECT @ergebnis As Ergebnis
```

liefert:

```
Ergebnis
----------------------------------------
Zeit und Kosten erfolgreich verbucht.
(1 row(s) affected)
```

## 8.6 Cursor in gespeicherten Prozeduren nutzen

In der Tabelle *tblProjektarbeitszeit* wird nicht nur die Stundenaufzeichnung von Einzelpersonen erfasst. Wenn das gesamte Projektteam gemeinsam – z.B. in einer Projektsitzung – Arbeitszeit in ein Projekt einbringt, wird die Zeit nicht für jeden Mitarbeiter einzeln, sondern für das gesamte Team einmal erfasst.

Das bedeutet, dass bei der Verbuchung der Arbeitszeit in den Projekten die Teamzeiten auf die einzelnen Mitarbeiter aufgeschlüsselt werden müssen. Dies ist notwendig, weil nicht alle Teammitglieder denselben kalkulatorischen Stundenlohn haben. Für diese Aufschlüsselung kann ein Cursor eingesetzt werden, da mehrere Datensätze dabei durchlaufen werden müssen.

Prinzipiell muss die Prozedur zuerst einmal feststellen, ob es sich um die Zeit einer Einzelperson oder eines Teams handelt. Die Verbuchung der Zeit einer Einzelperson läuft nach dem Schema der vorangegangenen Beispiele ab. Bei einem Team werden die Mitglieder eingelesen und in einer Schleife die Buchung für jedes einzelne Teammitglied vorgenommen. Die Verbuchung für die einzelnen Teammitglieder muss in eine Transaktion eingeschlossen werden. Denn alle Buchungen müssen rückgängig gemacht werden, wenn die Verbuchung für ein einziges Teammitglied fehlschlägt.

Der Prozedur wird als Eingabeparameter die ID der erfassten Arbeitszeit übergeben.

```
CREATE PROCEDURE spProjektzeitbuchung
    @id int,
    @ergebnis int OUTPUT
AS
DECLARE @projekt int
DECLARE @pers int, @team int
DECLARE @stunden decimal(3,1)
DECLARE @stdsatz smallmoney
```

Da für die Verbuchung die Projektnummer, die Personal- oder Teamnummer sowie die geleistete Stundenanzahl benötigt werden, werden diese anhand der übergebenen ID aus der Tabelle *tblProjektarbeitszeit* ausgelesen und in den zuvor deklarierten Variablen gespeichert.

```
    set nocount on
    SELECT @projekt = PazProjekt,
           @pers = PazPersonal,
           @team = PazTeam,
           @stunden = PazStunden
    FROM tblProjektarbeitszeit
    WHERE PazID = @id
```

Aufgrund der Definition der Tabelle *tblProjektarbeitszeit* kann nur entweder eine Personal- oder eine Teamnummer pro Datensatz eingetragen sein. Daher kann durch die Prüfung dieser Felder auf Null festgestellt werden, ob die zu verbuchende Zeit sich auf eine Einzelperson oder ein ganzes Projektteam bezieht. Ist die `@pers` nicht Null, handelt es sich um einen einzelnen Mitarbeiter. Der Stundensatz wird ausgelesen und danach erfolgt die Verbuchung.

```
    IF @pers Is Not Null
    BEGIN
        SET @stdsatz = (SELECT PersKalkStdLohn
                        FROM tblPersonal
                        WHERE PersNr = @pers)

        UPDATE tblProjekte
        SET ProjStundenIst = ProjStundenIst + @stunden,
            ProjKalkKosten = ProjKalkKosten + @stdsatz
            * @stunden
        WHERE ProjNr = @projekt
```

Als Prozedurergebnis wird `@@rowcount` dem Ausgabeparameter zugewiesen. War die Verbuchung erfolgreich, ist dies 1 sonst 0.

```
        SET @ergebnis = @@rowcount
    END
```

Nun kommen wir zum Teil der Teamverbuchung. Dafür wird ein Cursor definiert, der den kalkulatorischen Stundenlohn für jedes Teammitglied enthält.

```
IF @team is Not Null
BEGIN
    DECLARE team_cursor INSENSITIVE CURSOR
    FOR
        SELECT PersKalkStdLohn
        FROM tblPersonal p INNER JOIN
        tblTeammitglieder t
        ON p.PersNr = t.PersNr
        WHERE t.TeamNr = @team
        FOR READ ONLY
```

Der Cursor wird geöffnet und der Stundenlohn des ersten Teammitglieds wird eingelesen.

```
        OPEN team_cursor
        FETCH NEXT FROM team_cursor INTO @stdsatz
```

Bevor die erste Verbuchung innerhalb der Schleife erfolgt, wird eine Transaktion gestartet.

```
        BEGIN TRANSACTION
        WHILE @@fetch_status = 0
        BEGIN
            UPDATE tblProjekte
            SET ProjStundenIst = ProjStundenIst +
            @stunden,ProjKalkKosten = ProjKalkKosten
            + @stdsatz * @stunden
            WHERE ProjNr = @projekt
```

Nach jeder Einzelverbuchung wird über `@@rowcount` geprüft, ob diese erfolgreich gewesen ist. Liefert `@@rowcount` den Wert 0, ist die Verbuchung nicht erfolgt. In diesem Fall wird der Ergebniswert auf 0 gesetzt, der Cursor geschlossen, die Transaktion zurückgerollt und die Prozedur mit `RETURN` verlassen.

```
        IF @@rowcount = 0
        BEGIN
            SET @ergebnis = 0
            CLOSE team_cursor
            DEALLOCATE team_cursor
            ROLLBACK TRANSACTION
            RETURN
        END
```

Nach jeder erfolgreichen Verbuchung wird die nächste Zeile aus dem Cursor abgerufen, solange bis alle verbucht sind. Konnten alle Zeilen erfolgreich verbucht werden, wird das Prozedurergebnis auf 1 gesetzt, die Transaktion kommittiert und der Cursor geschlossen.

```
            FETCH NEXT FROM team_cursor INTO @stdsatz
        END
        SET @ergebnis = 2
        COMMIT TRANSACTION
        CLOSE team_cursor
        DEALLOCATE team_cursor
    END
RETURN
```

Die Prozedur liefert 0, wenn die Verbuchung nicht erfolgreich gewesen ist, 1 wenn eine erfolgreiche Einzelverbuchung zu verzeichnen ist, und 2, wenn ein Team erfolgreich verbucht werden konnte.

Testen wir die Prozedur mit der Verbuchung der Zeiterfassung mit der ID 3. Da es sich um eine Teamzeit handelt, erwarten wir als Ergebnis den Wert 2.

```
DECLARE @ergebnis int
EXEC spProjektzeitbuchung 3, @ergebnis OUTPUT
SELECT @ergebnis As Ergebnis
```

liefert:

```
Ergebnis
-----------
2
(1 row(s) affected)
```


Testen Sie diese Prozedur auch mit anderen Werten. Da aufgrund der referenziellen Integrität keine ungültigen Mitarbeiter und Teams sowie Teammitglieder vorkommen können, beschränkt sich die Fehlermöglichkeit auf Sperren oder andere externe Einflüsse.

Die Prozedur sollte in der Praxis sofort nach dem Anlegen eines neuen Datensatzes in der Tabelle *tblProjektarbeitszeit* ausgeführt werden.

## 8.7 Gespeicherte Prozeduren aus Client-Anwendungen aufrufen

Zum Abschluss dieses Kapitels möchten wir Ihnen noch einige Beispiele liefern, wie Sie auf dem SQL Server gespeicherte Prozeduren von einer Client-Anwendung aus aufrufen können.

### 8.7.1 Visual Basic und Visual Basic for Applications mit ADO

Wir beginnen mit zwei Funktionen, die Sie in einem VB-Projekt, aber auch in VBA einsetzen können. Auch wenn es andere Varianten gibt, empfiehlt sich der Einsatz von *ADO* (ActiveX Data Objects). Diese sind jener Teil von *OLE DB*, der für den Zugriff auf relationale Datenbanken zuständig ist.

ADO benötigt als Erstes eine Verbindung zur Datenquelle. Diese wird über einen Connect-String realisiert. Zu diesem gelangen Sie am einfachsten über eine so genannte Datenlinkdatei. Dies ist eine Textdatei mit der Dateierweiterung *UDL*. Um einen Datenlink anzulegen, klicken Sie mit der rechten Maustaste z.B. am Desktop und erstellen über den Befehl NEU eine neue TEXTDATEI. Diese benennen Sie von **.txt* nach **.udl* um. Danach klicken Sie doppelt auf diese Datei, um den Datenlink zu konfigurieren. Auf dem Register PROVIDER wählen Sie den *Microsoft OLE DB Provider for SQL Server* aus.

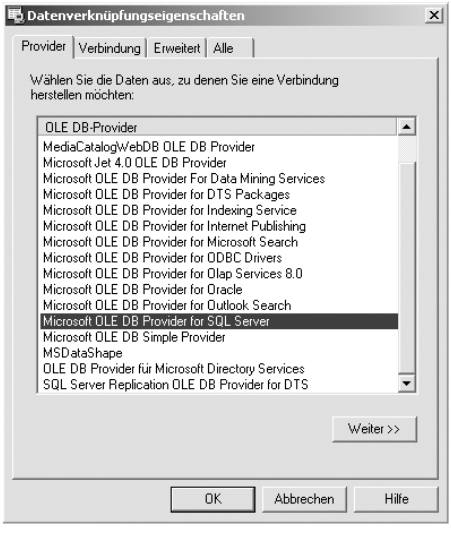



*Abb. 8.16: Provider auswählen*

Danach wechseln Sie auf das Register VERBINDUNG und wählen den Server, den Anmeldemodus sowie die Datenbank aus. Wenn möglich, sollten Sie die integrierte Sicherheit von Windows NT verwenden.

Ausführliche Informationen über die Benutzerverwaltung und die Anmeldemodi von SQL Server finden Sie in Kapitel 4.


*Abb. 8.17: Server, Anmeldung und Datenbank auswählen*

Testen Sie die Verbindung, und speichern Sie ihre Einstellungen und schließen Sie die Datei. Öffnen Sie die Datei nun mit dem Editor, um den Connect-String aus der Datei zu kopieren.

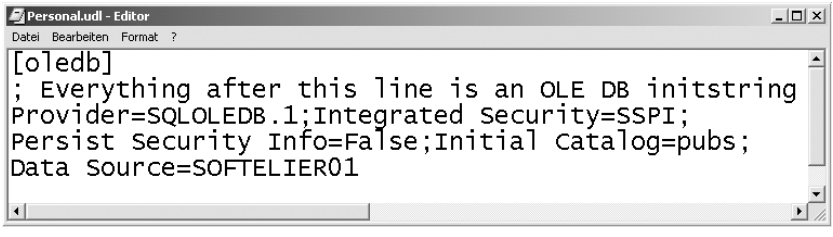



*Abb. 8.18: Connect-String in Datenlinkdatei*


Sie können die Datenlinkdatei auch gleich direkt zur Herstellung der Verbindung verwenden, jedoch macht es in der Praxis mehr Sinn, nur den Connect-String zu übernehmen und in den Code zu kopieren. Denn dann muss man nicht darauf achten, auch diese Datei immer mitzukopieren und den Pfad richtig anzugeben.

### Prozedur mit SELECT-Rückgabe

Die Funktion *ZeitbuchungSelect()* ruft die weiter vorne in diesem Kapitel erstellte gespeicherte Prozedur *spZeitbuchungSelect* auf. Die `EXECUTE`-Anweisung wird in der Variablen *sql* zusammengesetzt. Für die Verbindung wird ein

Connection-Objekt und für das Auslesen der zurückgegebenen Prozedurwerte ein Recordset-Objekt benötigt. Zuerst wird die Verbindung hergestellt und danach das Recordset mit der EXECUTE-Anweisung erzeugt.

Im Beispielsfall liefert die Prozedur immer eine Zeile mit einer Spalte. Diese wird aus dem Recordset ausgelesen und der Funktion als Ergebnis zugewiesen. Danach werden das Recordset sowie das Connection-Objekt wieder geschlossen.

```
Public Function ZeitbuchungSelect(ByVal projekt As Long, ByVal pers As
Long, ByVal std As Single) As String

Dim dbcon As ADODB.Connection, rs As ADODB.Recordset
Dim sql As String

    sql = "EXEC spZeitbuchungSelect " & projekt & ", "
    & pers & "," & Str(std)

    Set dbcon = New ADODB.Connection
    dbcon.ConnectionString = "Provider=SQLOLEDB.1;
    Integrated Security=SSPI;Initial
    Catalog=Personal;Data Source=SOFTELIER01"
    dbcon.Open
    Set rs = New ADODB.Recordset
    rs.Open sql, dbcon, adOpenStatic

    ZeitbuchungSelect = rs(0)

    rs.Close
    Set dbcon = Nothing
End Function
```

### Prozedur mit Output-Parametern

Um eine Prozedur mit Output-Parametern anzusprechen, wird ein Command-Objekt verwendet. Diesem werden sowohl die Eingabeparameter als auch die Ausgabeparameter »umgehängt«. Dem Command-Objekt werden außer den Parametern auch der Zieltyp (*CommandType*) und der Name der Prozedur (*CommandText*) zugewiesen. Sie können auch das Timeout (*CommandTimeout*) in Sekunden angeben. Beim Anhängen der Parameter werden der Datentyp sowie der Typ (Input/Output) definiert. Eingabeparametern muss ein Wert zugewiesen werden. Nachdem das Command-Objekt mit Execute ausgeführt worden ist, können Output-Parameter ausgelesen werden.

```
Public Function Projektzeitbuchung(ByVal id As Long) As Long
Dim dbcon As ADODB.Connection
Dim dbparam As ADODB.Parameter, dbcmd As ADODB.Command
```

```
    Set dbcon = New ADODB.Connection
    dbcon.ConnectionString = "Provider=SQLOLEDB.1;
    Integrated Security=SSPI;Initial
    Catalog=Personal;Data Source=SOFTELIER01"
    dbcon.Open

    Set dbcmd = New ADODB.Command
    dbcmd.CommandText = "spProjektzeitbuchung"
    dbcmd.CommandType = adCmdStoredProc
    dbcmd.CommandTimeout = 30

    Set dbparam = dbcmd.CreateParameter("@id", adInteger,
    adParamInput)
    dbparam.Value = id
    dbcmd.Parameters.Append dbparam

    Set dbparam = dbcmd.CreateParameter("@ergebnis",
    adInteger, adParamOutput)
    dbcmd.Parameters.Append dbparam

    Set dbcmd.ActiveConnection = dbcon
    dbcmd.Execute
    Projektzeitbuchung = dbcmd.Parameters("@ergebnis")
    Set dbcon = Nothing
End Function
```


Wenn Sie diese Funktion in einer Access-Projektdatei verwenden, kann das Connection-Objekt vereinfacht durch Zuweisung der Projektverbindung geöffnet werden:

```
Set dbcon = CurrentProject.Connection
```

### 8.7.2 Active Server Pagers / VBScript

Mit ähnlicher Syntax können Sie die Verbindung über VBScript herstellen und eine gespeicherte Prozedur aufrufen. Diese Syntax können Sie auch einsetzen, wenn Sie VBScript für Active Server Pages auf einem Internet-Server verwenden.

Wenn Sie den nachfolgenden Code in einer Textdatei mit der Erweiterung **.vbs* einsetzen, kann er direkt ausgeführt werden.

```
projekt = InputBox("Geben Sie die Projekt-ID ein:", "Zeitverbuchung",
"Projekt-ID")
pers = InputBox("Geben Sie die Personal-Nummer ein:", "Zeitverbuchung",
"Personal-Nr.")
std = InputBox("Geben Sie die Stundenanzahl ein:", "Zeitverbuchung",
"Stunden")
```

```
sql = "EXEC spZeitbuchungSelect " & projekt & ", " & pers & "," & std

Set dbcon = CreateObject("ADODB.Connection")
dbcon.open "Provider=SQLOLEDB.1;Integrated Security=SSPI;Initial
Catalog=Personal;Data Source=SOFTELIER01"
Set rs = CreateObject("ADODB.Recordset")
rs.Open sql, dbcon, 3, 1
ergebnis = rs(0)
rs.Close
Set dbcon = Nothing
MsgBox ergebnis, 64, "Zeitverbuchung"
```

Das Script finden Sie auf der Buch-CD unter dem Namen *Zeitbuchung.vbs*.


## 8.8 Fragen

1. Mit welchen SQL Server-Tools können Sie eine gespeicherte Prozedur erstellen?
2. Mit welcher Anweisung führen Sie eine gespeicherte Prozedur aus?
3. Welche Varianten der Wertrückgabe gibt es für gespeicherte Prozeduren?
4. Mit welcher Anweisung wird eine Prozedur sofort beendet?
5. Was bewirkt die Anweisung SET NOCOUNT ON in einer Prozedur?