

# 12 Trigger

Trigger lassen sich mit gespeicherten Prozeduren vergleichen. Sie werden auf ähnliche Weise angelegt und geändert und können eine oder mehrere SQL-Anweisungen kombiniert mit T-SQL-Befehlen und Strukturen wie Bedingungen oder Variablen enthalten. Ebenso wie bei den gespeicherten Prozeduren erstellt der SQL Server bei der ersten Ausführung eines Triggers einen Ausführungsplan und speichert diesen im Prozedurcache, sodass bei nachfolgenden Aufrufen der gespeicherte Ausführungsplan wiederverwendet wird und die Ausführung des Triggers einen Performancevorteil erhält.

Damit enden die Gemeinsamkeiten aber auch schon. Denn im Gegensatz zu gespeicherten Prozeduren sind Trigger keine eigenständigen Objekte und lassen sich dementsprechend auch nicht manuell aufrufen. Ein Trigger ist vielmehr Bestandteil der Definition einer Tabelle und legt fest, welche Aktionen bei den verschiedenen Datenoperationen ausgelöst werden.

## 12.1 Funktionsweise von Triggern

Wer bislang nur mit Access programmiert hat, erhält ein besseres Verständnis von der Funktionsweise von Triggern, wenn er diese mit den Ereignisprozeduren von Formularen oder Steuerelementen vergleicht. In Access können Sie Ereignisprozeduren hinterlegen, die durch verschiedene Ereignisse eines Formulars oder Steuerelements ausgelöst werden – etwa beim Laden oder Schließen des Formulars, beim Anzeigen oder Löschen eines neuen Datensatzes oder beim Betätigen einer Schaltfläche. Ähnlich ist es mit den Tabellen einer SQL Server-Datenbank. Auch hier lassen sich Prozeduren definieren, die durch verschiedene Ereignisse ausgelöst werden.

Welche Ereignisse können dies sein? Die folgende Übersicht fasst die möglichen Trigger zusammen:

- » *Aktualisieren eines Datensatzes:* Wird ein Datensatz einer Tabelle geändert, löst dies den sogenannten Update-Trigger aus.
- » *Löschen eines Datensatzes:* Wird ein Datensatz einer Tabelle gelöscht, löst dies den Delete-Trigger aus.
- » *Anlegen eines Datensatzes:* Auch beim Anlegen eines Datensatzes wird ein Trigger ausgelöst, in diesem Fall der Insert-Trigger.

Dabei ist wichtig, dass es keine Rolle spielt, auf welchem Weg ein Datensatz der Tabelle hinzugefügt, geändert oder gelöscht wird – ob im Frontend, mittels einer gespeicherten Prozedur oder durch die direkte Eingabe einer entsprechenden Anweisung im SQL Server Management Studio. Existiert an der Tabelle für diese Aktionen ein Trigger, wird dieser auch ausgelöst.

## 12.2 Pro und Contra

Trigger sind wegen ihrer Unbestechlichkeit nahezu perfekt zur Gewährleistung der Datenkonsistenz und zur Einhaltung von Geschäftsregeln. Nachfolgend finden Sie einige praktische Einsatzzwecke:

- » *Durchsetzen referenzieller Integrität:* Referenzielle Integrität inklusive Lösch- und Aktualisierungsweitergabe sind fester Bestandteil des Funktionsumfangs vom SQL Server. Dennoch kann es sein, dass Sie zusätzliche Anforderungen an die Prüfung referenzieller Integrität oder an die Lösch- und Aktualisierungsweitergabe haben. Diese lassen sich mit Triggern realisieren.
- » *Definition von Standardwerten:* SQL Server bietet die Möglichkeit, Standardwerte für Felder auf Basis von Konstanten sowie einfacher T-SQL-Funktionen und eigener Skalarfunktionen zu definieren. Alternativ wäre dies auch mit Triggern möglich.
- » *Definition von Einschränkungen:* Um eine Datenmanipulation zu verhindern, die gegen definierte Regeln verstößt, bieten Tabellen die *Check Constraints* (zu deutsch Einschränkungen). Ähnlich der Standardwerte können Sie zur Prüfung der Eingaben Konstanten sowie einfache T-SQL-Funktionen und eigene Skalarfunktionen nutzen. Die Prüfung der Eingaben kann aber auch von Triggern übernommen werden.
- » *Protokollieren von Datenänderungen:* Jede Änderung an den Daten einer Tabelle lässt sich über Trigger protokollieren. Auf diese Weise wäre nachvollziehbar, wer wann welchen Datensatz hinzugefügt, geändert oder gelöscht hat.
- » *Redundante Datenhaltung:* Redundante Daten werden gerne zur Performance-Steigerung verwendet. Um zum Beispiel die Rechnungssumme nicht für jede Auswertung erneut aus den Summen der Rechnungspositionen ermitteln zu müssen, könnte diese ebenso gut im Rechnungskopf gespeichert werden. Es muss jedoch gewährleistet sein, dass bei jeder Änderung der Positionen die Summe im Rechnungskopf angepasst wird. Wenn schon Redundanz, dann sollte diese auch korrekt und konsistent sein. Diese Konsistenz könnte ein Trigger sicherstellen. Egal ob eine neue Position hinzugefügt, eine bestehende geändert oder gelöscht wird, der Trigger übernimmt die Übertragung der neuen Summe in den Rechnungskopf.

Dies sind nur ein paar Beispiele für die Verwendung von Triggern. Beispiele, die auch gleichzeitig gute Argumente für den Einsatz von Triggern liefern. Nun gibt es aber auch durchaus einige Argumente gegen die Verwendung von Triggern.

Nicht umsonst wird oft über das Für und Wider von Triggern diskutiert. Wir möchten Ihnen die Entscheidung überlassen und führen nun auch die Gegenargumente auf:

- » *Keine Transparenz:* Die Trigger einer Datenbank und deren Auswirkungen sind nicht einfach zu überblicken. Trigger sind fest mit einer Tabelle verbunden und deshalb auch nur dort zu

finden. Um die Trigger einer Datenbank zu sehen, müssen Sie sich also mühsam durch alle Tabellen klicken oder aber Sie fragen die entsprechenden Systemtabellen der Datenbank ab und erstellen sich so eine Aufstellung aller Trigger. Aber auch mit einer Übersicht aller Trigger bleiben deren Auswirkungen weiterhin im Verborgenen. Schließlich könnte das Hinzufügen eines Datensatzes in Tabelle A durch einen Trigger eine Datenänderung an Tabelle B auslösen, die wiederum einen Trigger auslöst, der in Tabelle C und in Tabelle D einen Wert ändert. Es wäre sogar möglich, dass eine dieser Änderungen einen anderen Datensatz in Tabelle A löscht, worauf dort erneut ein Trigger aktiv wird. Solche rekursiven Aufrufe sind bis zu einer Tiefe von 32 Ebenen möglich, zum Glück nicht ohne vorherige Aktivierung.

- » *Schlechte Performance:* Abhängig von der Programmierung eines Triggers führt dieser seine Verarbeitung zu jedem Datensatz aus, der gerade hinzugefügt, geändert oder gelöscht wird. Je aufwendiger die Verarbeitung des Triggers ist, umso schlechter wird die Performance der Datenverarbeitung. Dies macht sich gerade dann bemerkbar, wenn Sie mit einer Anweisung gleich mehrere Datensätze verarbeiten.
- » *Komplexe Programmierung:* Zwar verwenden Sie zur Programmierung von Triggern ebenfalls T-SQL, jedoch ist die Vorgehensweise und Logik hierbei etwas anders. Mehr dazu später im Abschnitt »Trigger erstellen«, Seite 292.
- » *Für das Frontend unsichtbar:* Eine Datenänderung über eine SQL-Anweisung im VBA oder direkt in einer eingebundenen Tabelle weist Sie in keiner Weise auf einen möglichen Trigger hin. Dies bedeutet, dass Sie im Quellcode des Frontends von den Aktionen eines Triggers nichts sehen. Bilden Sie hingegen die Logik des Triggers mit gespeicherten Prozeduren ab, sehen Sie im Quellcode den Aufruf der jeweiligen gespeicherten Prozeduren.
- » *Gespeicherte Prozeduren:* Jegliche Aktion, die ein Trigger vor oder nach einer Datenänderung ausführen soll, können Sie auch in einer gespeicherten Prozedur realisieren. Eingabeprüfungen zum Beispiel führen Sie in einer gespeicherten Prozedur in entsprechenden *IF*-Anweisungen aus, worauf in Abhängigkeit vom Ergebnis die Aktion ausgeführt wird oder eben nicht. Oder soll beispielsweise beim Ändern einer Rechnungsposition die Rechnungssumme im Kopf angepasst werden, sind der *INSERT*- und *UPDATE*-Befehl lediglich in einer Transaktion auszuführen. Gespeicherte Prozeduren bieten Triggern gegenüber einen entscheidenden Vorteil: Anstelle der verborgenen Aktionen eines Triggers, die zudem noch eine wilde Verkettung weiterer Trigger zur Folge haben kann, sehen Sie in einer gespeicherten Prozedur alle notwendigen SQL-Anweisungen – lesbar und einfach nachvollziehbar.

Natürlich kommt es bei der Entscheidung für den Einsatz von Triggern nicht nur auf die oben genannten Argumente an. Viel entscheidender sind die Umstände des Projekts. In einer bestehenden und bereits installierten Client-/Server-Applikation werden Sie sich nicht unbedingt für die Variante mit gespeicherten Prozeduren entscheiden. Dies würde entscheidende Änderungen am Frontend nach sich ziehen, müssten Sie dort doch die jeweiligen Aufrufe der gespeicherten Prozeduren programmieren. In einem solchen Fall lässt sich die Logik schneller mit Triggern abbilden, da Sie hierbei das Frontend nicht zu ändern brauchen. Nur als Tipp: Dokumentieren

Sie Ihre Trigger und deren Auswirkungen auf andere Tabellen und deren Trigger von Anfang an ausführlich. Das erspart Ihnen später Zeit bei einer Fehlersuche. Sind Sie jedoch noch am Anfang des Projekts, sollten Sie den gespeicherten Prozeduren den Vorzug geben und auf Trigger verzichten – und genau dies ist der Fall, wenn Sie nach der Migration einer Access-Datenbank zum SQL Server auch die Logik von Access nach SQL Server verlagern. Technisch spricht nichts dagegen, Trigger und gespeicherte Prozeduren gemeinsam zu verwenden. Sie sollten jedoch gerade bei dieser Konstellation den Tipp zur ausführlichen Dokumentation berücksichtigen.

### 12.3 Zwei Arten von Triggern

SQL Server bietet Ihnen zwei verschiedene Trigger an. Beide reagieren auf die Aktionen *INSERT*, *UPDATE* und *DELETE*, jedoch zu unterschiedlichen Zeitpunkten:

- » *INSTEAD OF-Trigger*: wird vor der Datenmanipulation ausgelöst und ersetzt die eigentliche Aktion.
- » *AFTER-Trigger*: wird nach der Datenmanipulation ausgelöst.

Um zu verstehen, wann die beiden Trigger ausgeführt werden, schauen wir uns einmal den gesamten Prozess einer Datenmanipulation am Beispiel einer *INSERT*-Anweisung an:

- » Erstellen der virtuellen Tabellen *inserted* und *deleted*: Mehr dazu im Anschluss.
- » Auslösen des *INSTEAD OF*-Triggers: Die Logik des Triggers wird verarbeitet. Beinhaltet diese beispielsweise eine Prüfung der Eingabe, könnte der Trigger bei einem negativen Ergebnis die *INSERT*-Anweisung bereits an dieser Stelle abbrechen.
- » Prüfen der Einschränkungen (Check Constraints) und der referenziellen Integrität: Schlägt eine der Prüfungen fehl, wird die Verarbeitung der *INSERT*-Anweisung abgebrochen.
- » Erzeugen von Standardwerten: Zu jeder Spalte, die nicht durch die *INSERT*-Anweisung einen Wert erhält, wird gemäß der Spaltendefinition der Standardwert erstellt.
- » Manipulation der Daten: Der Datensatz wird der Datenseite und somit der Tabelle hinzugefügt.
- » Auslösen des *AFTER*-Triggers: Die im Trigger enthaltene Logik wird verarbeitet. Gibt es hierbei einen Fehler, wird der komplette *INSERT*-Vorgang abgebrochen und die Tabelle auf den Zustand vor der Aktion zurückgesetzt – es wird ein *ROLLBACK* ausgeführt.
- » Bestätigen der Datenmanipulation: Die *INSERT*-Anweisung wird mit einem *COMMIT* bestätigt.

Interessant bei diesem Prozess ist der *AFTER*-Trigger. Obwohl dieser erst nach der eigentlichen Aktion ausgeführt wird, ist er dennoch ein fester Bestandteil dieser Aktion. Ein Fehler im *AFTER*-Trigger würde in unserem Beispiel die *INSERT*-Anweisung verhindern, auch wenn es beim eigentlichen Hinzufügen des Datensatzes kein Problem gibt. Ein Umstand, den es bei der

Programmierung von *AFTER*-Triggern zu beachten gilt. Dort sollten keine komplexen und fehleranfälligen Verarbeitungen erfolgen.

### Die virtuellen Tabellen *inserted* und *deleted*

Kommen wir auf die virtuellen Tabellen *inserted* und *deleted* zurück. Beide Tabellen werden bei jeder Datenmanipulation erstellt und enthalten die davon betroffenen Daten.

Abhängig von der Aktion sind die Daten wie folgt auf die beiden Tabellen verteilt:

- » *INSERT*: Die Tabelle *inserted* beinhaltet die neuen Datensätze, die Tabelle *deleted* ist leer.
- » *UPDATE*: Die Tabelle *inserted* beinhaltet die neuen Werte der Datensätze, die Tabelle *deleted* die Werte vor der Änderung.
- » *DELETE*: Die Tabelle *inserted* ist leer und die Tabelle *deleted* beinhaltet die gelöschten Datensätze.

Im Trigger können Sie nun mit den Daten dieser beiden Tabellen arbeiten. Die Daten lassen sich per *SELECT* auswerten und dabei mit anderen Tabellen verknüpfen sowie aggregieren, sortieren und filtern. Nur das Ändern der Daten ist nicht möglich.

Ob Sie die Daten der beiden virtuellen Tabellen im Trigger überhaupt benötigen, ist abhängig von der dort abgebildeten Logik. Ein Trigger reagiert auf die Aktion, für die er erstellt wurde und nicht unbedingt auf die dabei geänderten Daten. Sie könnten zum Beispiel mit einem *AFTER*-Trigger lediglich das Löschen von Datensätzen protokollieren.

Dabei wird nur der Zeitpunkt und der Benutzer in einem Protokoll gespeichert, nicht aber die gelöschten Daten. Möchten Sie hingegen die entfernten Datensätze ebenfalls protokollieren, müssen Sie im Trigger die Daten der virtuellen Tabelle *deleted* auswerten und diese im Protokoll speichern. Wie Sie die Daten der virtuellen Tabellen im Trigger verwenden, sehen Sie in den kommenden Beispielen.

### Mehrere Trigger an einer Tabelle

Eine Tabelle kann maximal einen *INSTEAD OF*-Trigger enthalten, jedoch mehrere *AFTER*-Trigger. Es wäre also beispielsweise möglich, einen *AFTER*-Trigger für die Aktionen *INSERT*, *UPDATE* und *DELETE* zur Protokollierung von Datenänderungen zu definieren sowie einen weiteren *AFTER*-Trigger für einen besonderen und eher seltenen Fall einer *UPDATE*-Anweisung. In einem solchen Fall würden beim Ändern eines Datensatzes beide Trigger ausgelöst.

Grundsätzlich gibt es für die Ausführung mehrerer *AFTER*-Trigger keine Reihenfolge, es sei denn Sie legen diese mit der Systemprozedur *sp\_SetTriggerOrder* fest. Allerdings definieren Sie hiermit lediglich die beiden *AFTER*-Trigger, die als Erstes und als Letztes ausgeführt werden.

Die Ausführungsreihenfolge für alle anderen *AFTER*-Trigger lässt sich nicht beeinflussen. Haben Sie also an einer Tabelle vier *AFTER*-Trigger, ist nur die Ausführungsreihenfolge für den als ersten

## Kapitel 12 Trigger

und den als letzten definierten Trigger festlegt. Die Ausführung der beiden anderen Trigger erfolgt willkürlich.

Grundsätzlich sollten Sie sich gut überlegen, ob Sie die Logik eines *AFTER*-Triggers in einem Trigger abbilden oder ob Sie diese auf mehrere Trigger aufteilen. Wie so oft, gilt auch hier der Tipp: *KISS* – Keep It Simple And Stupid.

### 12.4 Trigger erstellen

Trigger werden wie die anderen SQL Server-Objekte mit der *CREATE*-Anweisung angelegt – hier mit *CREATE TRIGGER*. Das SQL Server Management Studio bietet zudem wie bei gespeicherten Prozeduren und Funktionen eine Vorlage mit der Syntax zum Anlegen eines Triggers.

Um diese Vorlage zu erhalten, navigieren Sie im Objekt-Explorer zu der Tabelle, an der der Trigger angelegt werden soll. Dort wählen Sie im Kontextmenü des Elements *Trigger* den Eintrag *Neuer Trigger*. Die Vorlage wird dann in einem neuen Abfragefenster geöffnet (siehe Abbildung 12.1).

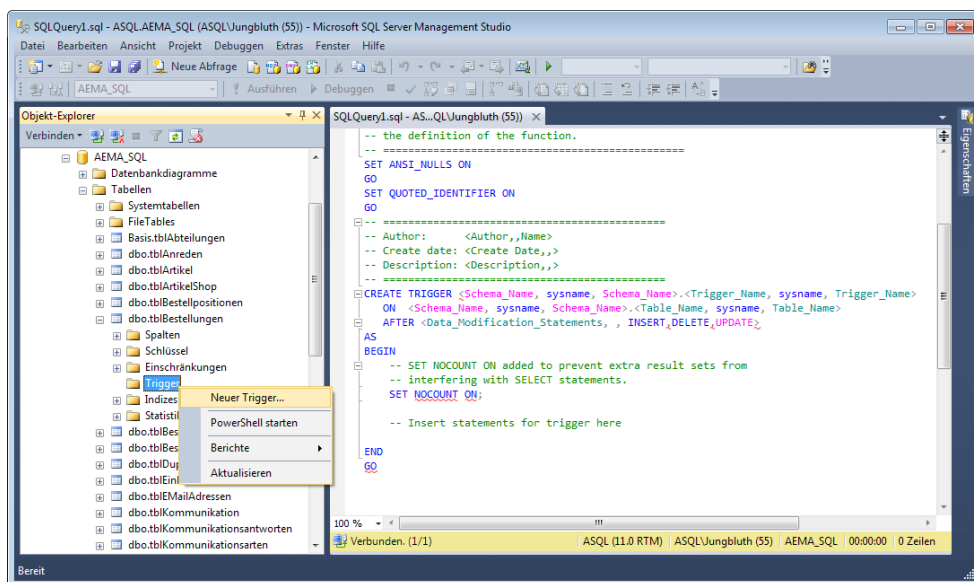


Abbildung 12.1: Einen neuen Trigger anlegen

Leider ist auch diese Vorlage nicht sonderlich hilfreich. Deshalb zeigen wir Ihnen in den folgenden Beispielen, wie Sie Trigger ohne die Vorlage anlegen.

Ein Trigger ist in seiner Struktur vielschichtiger als eine gespeicherte Prozedur, eine Sicht oder eine Funktion. Infolgedessen ist auch seine Syntax etwas komplexer. Bevor wir also zu den ei-

gentlichen Beispielen kommen, möchten wir Ihnen zunächst die Grundstruktur eines Triggers vorstellen.

Los geht es mit der *CREATE TRIGGER*-Anweisung. Es folgt der Name des Schemas, dem Sie den Trigger zuordnen möchten. Anschließend geben Sie dem Trigger einen Namen. Bei der Namensvergabe empfiehlt sich wieder die Verwendung eines Präfix, etwa *tr\_*.

Ebenso empfehlenswert ist es, bereits im Triggernamen darauf hinzuweisen, zu welcher Tabelle der Trigger gehört, um welchen Typ es sich handelt und bei welchen Aktionen er ausgelöst wird. Hier ein Beispiel für den Namen eines Triggers, der als *AFTER*-Trigger auf *UPDATE*-Anweisungen an der Tabelle *tblBestellungen* reagiert:

```
dbo.tr_tblBestellungenAfterUpdate
```

Nach dem Schema und der Bezeichnung folgt mit dem Schlüsselwort *ON* die Zuweisung zur Tabelle. Anschließend wird die Art des Triggers mit *AFTER* oder *INSTEAD OF* angegeben und die Aktionen aufgelistet, bei denen der Trigger ausgelöst werden soll.

```
CREATE TRIGGER dbo.tr_tblBestellungenAfterUpdate  
ON dbo.tblBestellungen AFTER Update
```

Mit dem Schlüsselwort *AS* leiten Sie dann den eigentlichen Programmcode ein, der auch hier wieder in einem *BEGIN...END*-Block enthalten ist.

```
CREATE TRIGGER dbo.tr_tblBestellungenAfterUpdate  
ON dbo.tblBestellungen AFTER Update  
AS  
BEGIN  
    SET NOCOUNT ON;  
    -- Ihr Programmcode  
END
```

Im obigen Skript sehen Sie die Anweisung *SET NOCOUNT ON*. Diese geben Sie in jedem Trigger aus den gleichen Gründen wie bei den gespeicherten Prozeduren an.

Innerhalb des *BEGIN...END*-Blocks können Sie mit T-SQL, Variablen, *IF*-Anweisungen und Schleifen arbeiten. Dabei sollten Sie jedoch nie die Performance außer Acht lassen. Bedenken Sie, dass die hier abgebildete Logik bei jeder Aktion ausgelöst wird, für die der Trigger definiert ist.

Die Trigger-Programmierung bietet eine besondere Spezialität: die Funktion *UPDATE()*. Hiermit prüfen Sie, ob sich bei der aktuellen Datenverarbeitung der Wert einer bestimmten Spalte geändert hat. Zur Prüfung geben Sie dazu den Spaltennamen als Parameter an:

```
IF UPDATE(<spaltenname>)
```

Wurde der Inhalt der Spalte geändert, liefert die Funktion den Wert *True*, ansonsten *False*. Auf diese Weise können Sie die weitere Verarbeitung des Triggers abhängig von der Änderung einer bestimmten Spalte steuern. Im Abschnitt »*INSTEAD OF*-Trigger erstellen«, Seite 294, erstellen sehen Sie den Einsatz dieser Funktion.

Neben den gespeicherten Prozeduren sind Trigger die einzigen SQL Server-Objekte, mit denen Sie Daten in Tabellen hinzufügen, ändern und löschen können. Jedoch sollten Sie in einem Trigger die SQL-Befehle *INSERT*, *UPDATE* und *DELETE* vorsichtig einsetzen und vorab prüfen, ob diese Aktionen nicht weitere Trigger an den betroffenen Tabellen auslösen.

Dies gilt auch für gespeicherte Prozeduren, die Sie in Triggern verwenden. Auch hier sind vorher die Auswirkungen zu prüfen.

### 12.4.1 INSTEAD OF-Trigger erstellen

Ein *INSTEAD OF*-Trigger wird vor der eigentlichen Datenmanipulation ausgeführt. Er ist also prädestiniert für Prüfungen. Bei einem negativen Ergebnis der Prüfung kann im Trigger die eigentliche Aktion abgebrochen oder auch korrigiert werden.

Liefert die Prüfung ein positives Ergebnis, bedeutet dies aber nicht, dass der Trigger die tatsächliche Aktion ausführt. *INSTEAD OF* steht nun mal für *Anstelle dessen*, weshalb auch die im Trigger enthaltene Logik anstelle der eigentlichen SQL-Anweisung ausgeführt wird. Soll also im positiven Fall der Prüfung die tatsächliche Aktion verarbeitet werden, muss der Quellcode des Triggers die SQL-Anweisung der tatsächlichen Aktion auch enthalten.

Ein *INSTEAD OF*-Trigger kann die eigentliche SQL-Anweisung ebenso komplett mit einer anderen Verarbeitung ersetzen. So wäre es beispielsweise möglich, dass ein Trigger anstelle einer ursprünglichen *DELETE*-Anweisung eine *UPDATE*-Anweisung ausführt. Auf diese Weise könnten Sie das Löschen von Datensätzen vermeiden und stattdessen diese als inaktiv kennzeichnen.

Im Gegensatz zum *AFTER*-Trigger erweitert ein *INSTEAD OF*-Trigger also nicht den Vorgang der eigentlichen Datenverarbeitung, sondern ersetzt diesen durch seine eigene Logik. Für die Verarbeitung der Daten ist dann nicht mehr die ursprüngliche Aktion maßgebend, sondern die SQL-Anweisungen des *INSTEAD OF*-Triggers.

In unserem ersten Beispiel wollen wir dies nutzen und mit einem *INSTEAD OF*-Trigger das Löschen von Datensätzen in der Tabelle *tblRechnungen* vermeiden. Da hier eine bestimmte Datenverarbeitung kategorisch ausgeschlossen werden soll, ist dies ein klassischer Fall für einen *INSTEAD OF*-Trigger.

Für den neuen Trigger öffnen Sie zunächst ein neues Abfragefenster über den Eintrag *Neue Abfrage* aus dem Kontextmenü der Beispieldatenbank *AEMA\_SQL*. Dort geben Sie nun die *CREATE TRIGGER*-Anweisung ein, gefolgt vom Schema und dem Namen für den Trigger.

```
CREATE TRIGGER dbo.tr_tblRechnungenInsteadOfDelete
```

Das es sich um einen Trigger an der Tabelle *tblRechnungen* handelt, der zudem auf die Aktion *DELETE* reagieren soll, definieren Sie mit den folgenden Parametern:

```
ON dbo.tblRechnungen INSTEAD OF Delete
```



Es folgt das Schlüsselwort *AS* und darauf die eigentliche Logik im *BEGIN...END*-Block. Die Logik beginnt wie bereits erwähnt immer mit *SET NOCOUNT ON*.

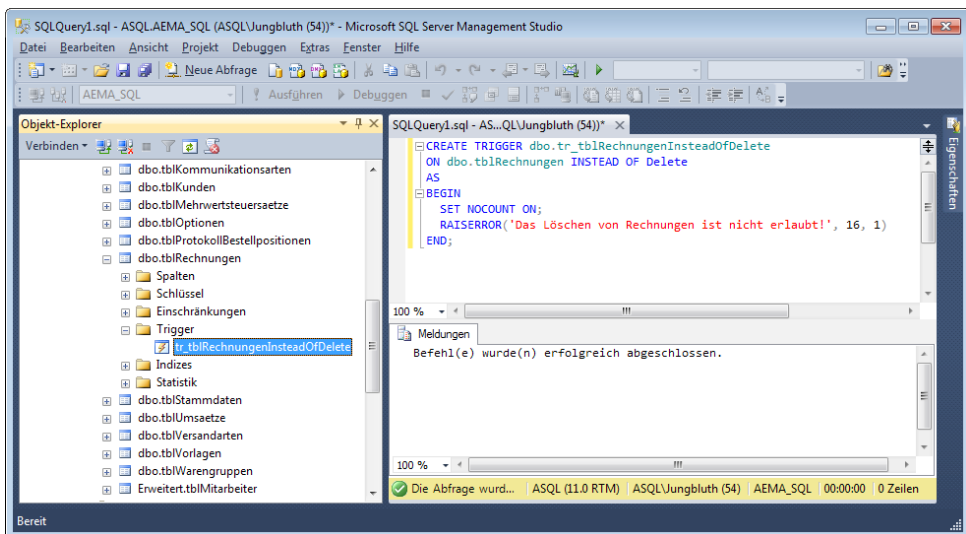
Da dieser Trigger letztendlich nur das Löschen von Datensätzen vermeiden soll, besteht der Quellcode lediglich aus einer Fehlermeldung, die mit dem T-SQL-Befehl *RAISERROR* ausgelöst wird.

Der komplette Trigger sieht folgendermaßen aus:

```
CREATE TRIGGER dbo.tr_tblRechnungenInsteadOfDelete
ON dbo.tblRechnungen INSTEAD OF Delete
AS
BEGIN
    SET NOCOUNT ON;
    RAISERROR('Das Löschen von Rechnungen ist nicht erlaubt!', 16, 1)
END;
```

Nachdem Sie den Quellcode im Abfragefenster eingegeben haben, führen Sie diesen mit der Taste *F5* aus. Der Trigger wird angelegt, wodurch die im Trigger enthaltenen SQL-Anweisungen nun ein fester Bestandteil der Tabelle sind.

Sie finden den Trigger im Element *Trigger* der Tabelle *tblRechnungen*. Möglicherweise müssen Sie die Anzeige im Objekt-Explorer zunächst aktualisieren (siehe Abbildung 12.2).



**Abbildung 12.2:** Der neue Trigger

Den Trigger können Sie in einem neuen Abfragefenster mit der folgenden *DELETE*-Anweisung testen:

```
DELETE FROM dbo.tblRechnungen WHERE RechnungID = 2;
```

## Kapitel 12 Trigger

Als Ergebnis erhalten Sie die Meldung des Triggers, dass das Löschen von Rechnungen nicht erlaubt ist (siehe Abbildung 12.3). Diese Meldung wird auch ausgegeben, wenn Sie mehrere Rechnungen mit einer SQL-Anweisung löschen.

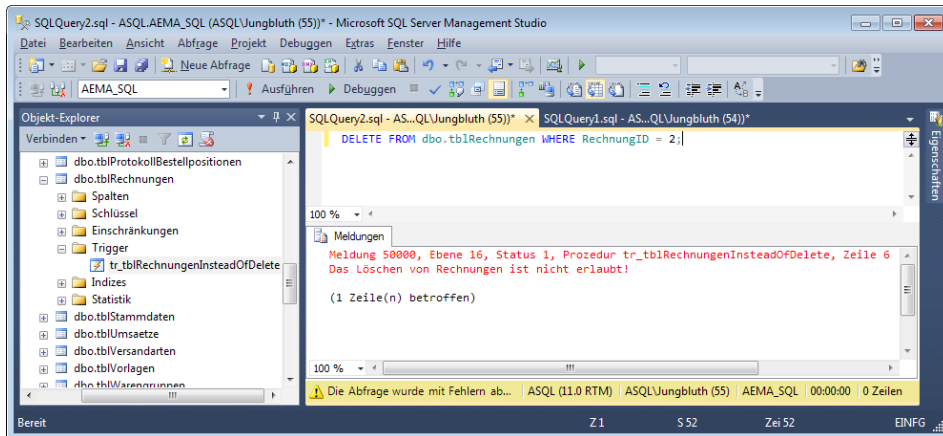


Abbildung 12.3: Der neue Trigger in Aktion

Dieses Beispiel zeigt nicht nur einen klassischen Einsatz eines *INSTEAD OF*-Triggers, Sie sehen hier auch, dass die Verarbeitung eines Triggers unabhängig von den verarbeiteten Datensätzen sein kann.

Im nächsten Beispiel wird der Trigger die durch die Aktion geänderten Daten verarbeiten. Dieser Trigger vermeidet in der Tabelle *tblBestellpositionen* die Eingabe von Datensätzen mit einem zu hohen Rabatt. Durch eine fiktive Geschäftsregel sind Rabatte von mehr als 10 € nicht erlaubt und sollen auf jeden Fall vermieden werden. Gibt es beim Hinzufügen oder Ändern von Datensätzen auch nur einen Datensatz mit einem Rabatt von mehr als 10 €, wird die Aktion direkt abgebrochen. Ein Fall für einen *INSTEAD OF*-Trigger. Dieses Beispiel zeigt Ihnen drei neue Aspekte bei der Programmierung eines Triggers:

- » Die Verwendung der Funktion *UPDATE()*
- » Der Bezug auf die verarbeiteten Datensätze
- » Die Besonderheit eines *INSTEAD OF*-Triggers

Beginnen wir von vorne und geben dem Trigger zunächst einen Namen, sowie die Zuordnung zur Tabelle und den Aktionen:

```
CREATE TRIGGER dbo.tr_tblBestellpositionenInsteadOfInsertUpdate  
ON dbo.tblBestellpositionen INSTEAD OF Insert, Update
```

Um die Rabattvergabe beim Neuanlegen wie auch beim Ändern eines Datensatzes zu prüfen, wird der Trigger für die Aktionen *INSERT* und *UPDATE* definiert. Bevor es danach an die tat-

sächliche Logik geht, ist wieder das Schlüsselwort *AS* und der *BEGIN...END*-Block anzugeben. Die eigentliche Verarbeitung des Triggers beginnt erneut mit der Anweisung *SET NOCOUNT ON*. Danach wird in einer *IF*-Anweisung mit der Funktion *UPDATE()* geprüft, ob bei der Datenmanipulation der Wert der Spalte *Rabatt* geändert wurde. Liefert dies den Wert *True*, folgt die weitere Verarbeitung der Datensätze mit geänderten Rabattwerten. Ist das Ergebnis der Prüfung jedoch *False*, bedeutet dies, dass die Spalte *Rabatt* nicht von der Datenmanipulation betroffen war.

Betrachten wir zunächst den Fall, dass der Inhalt der Spalte verändert wurde. Hier folgt dann in einer weiteren *IF*-Anweisung die Prüfung, ob einer der vergebenen Rabatte über 10 € liegt. Dazu wird in der Tabelle *inserted* der größte Wert der Spalte *Rabatt* ermittelt und verglichen. Die Tabelle *inserted* beinhaltet im Fall einer *INSERT*-Aktion die neuen Werte und im Fall einer *UPDATE*-Aktion die Werte der Änderung.

Liegt der höchste vergabene Rabatt über dem Grenzwert, wird mit der *RAISERROR*-Anweisung eine Meldung ausgegeben und die gesamte Datenverarbeitung per *RETURN* beendet.

```
IF UPDATE(Rabatt)
BEGIN
    IF (SELECT Max(Rabatt) FROM inserted) > 10
        BEGIN
            RAISERROR ('Es wurde ein Rabatt von mehr als 10 Euro vergeben!', 16, 1)
            RETURN
        END
END
END;
```

Soweit die Verarbeitung der Datensätze mit einem zu hohen Rabatt. Wurde jedoch weder die Spalte *Rabatt* verändert, noch ein Rabatt über der angegebenen Grenze vergeben, können die Datensätze der Tabelle hinzugefügt oder dort geändert werden.

Ein *INSTEAD OF*-Trigger führt aber per Definition nicht die ursprüngliche Aktion aus, sondern ersetzt diese durch seine eigene Verarbeitung. Also muss der Quellcode des Triggers die hierfür notwendigen *INSERT*- und *UPDATE*-Anweisungen enthalten. Wobei die *UPDATE*-Anweisung natürlich nur bei einer *UPDATE*-Aktion und die *INSERT*-Anweisung im Fall einer *INSERT*-Aktion ausgeführt werden darf.

Woher aber kommt die Information, ob der Trigger nun über eine *INSERT*-Aktion oder durch eine *UPDATE*-Aktion ausgelöst wurde? Die Antwort liefert uns in diesem Fall die virtuelle Tabelle *deleted*. Diese Tabelle enthält bei einer *UPDATE*-Aktion die Werte vor der Änderung und im Fall einer *INSERT*-Aktion ist sie leer. Wir müssen also lediglich prüfen, ob die Tabelle *deleted* Daten enthält. Ist dies der Fall, handelt es sich um eine *UPDATE*-Aktion, ansonsten um eine *INSERT*-Aktion:

```
IF (SELECT Count(*) FROM deleted) = 0
BEGIN
    INSERT dbo.tblBestellpositionen
        (BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
        SELECT BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt FROM
        inserted;
```

## Kapitel 12 Trigger

```
END
ELSE
BEGIN
    UPDATE dbo.tblBestellpositionen
    SET BestellungID = inserted.BestellungID,
        ArtikelID = inserted.ArtikelID,
        Einzelpreis = inserted.Einzelpreis,
        Mehrwertsteuersatz = inserted.Mehrwertsteuersatz,
        Menge = inserted.Menge,
        Rabatt = inserted.Rabatt
    FROM inserted INNER JOIN dbo.tblBestellpositionen
        ON inserted.BestellpositionID = dbo.tblBestellpositionen.
        BestellpositionID;
END
```

Das Hinzufügen der neuen Datensätze erfolgt mit den Daten der virtuellen Tabelle *inserted*. Auch für die *UPDATE*-Anweisung sind die Daten der Tabelle *inserted* maßgebend. Diese werden hierfür mit den Daten der Tabelle *tblBestellpositionen* verknüpft. Mit diesen Anweisungen ist der Quellcode des Triggers auch schon komplett. Der gesamte Quellcode sieht wie folgt aus:

```
CREATE TRIGGER dbo.tr_tblBestellpositionenInsteadOfInsertUpdate
ON dbo.tblBestellpositionen INSTEAD OF Insert, Update
AS
BEGIN
    SET NOCOUNT ON;
    IF UPDATE(Rabatt)
    BEGIN
        IF (SELECT Max(Rabatt) FROM inserted) > 10
        BEGIN
            RAISERROR ('Es wurde ein Rabatt von mehr als 10 Euro vergeben!', 16, 1)
            RETURN
        END
    END
    IF (SELECT Count(*) FROM deleted) = 0
    BEGIN
        INSERT dbo.tblBestellpositionen
        (BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
        SELECT BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt FROM
inserted;
    END
    ELSE
    BEGIN
        UPDATE dbo.tblBestellpositionen
        SET BestellungID = inserted.BestellungID,
            ArtikelID = inserted.ArtikelID,
            Einzelpreis = inserted.Einzelpreis,
            Mehrwertsteuersatz = inserted.Mehrwertsteuersatz,
            Menge = inserted.Menge,
            Rabatt = inserted.Rabatt
        FROM inserted INNER JOIN dbo.tblBestellpositionen
            ON inserted.BestellpositionID = dbo.tblBestellpositionen.BestellpositionID;
    END
END
```

Um den Trigger zu testen, verwenden Sie die folgende Anweisung:

```
UPDATE dbo.tblBestellpositionen SET Rabatt = 11 WHERE BestellpositionID = 279;
```

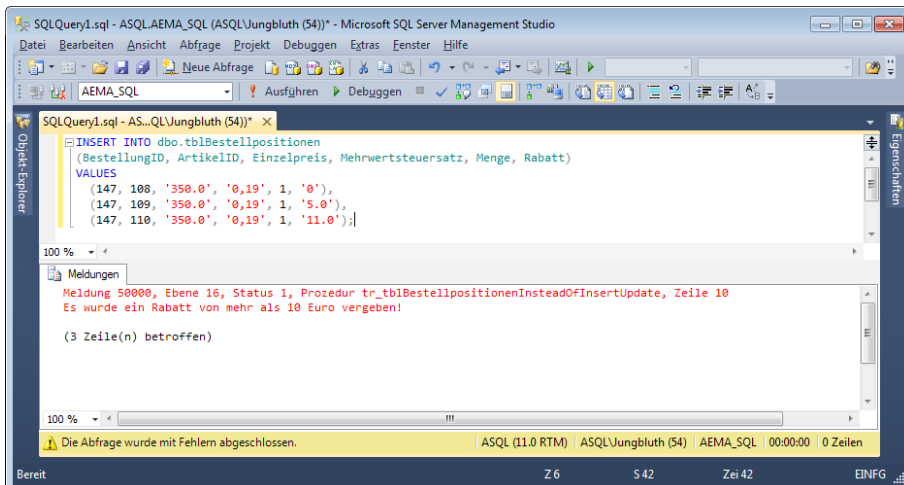
Die Änderung der Bestellposition mit der ID 279 wird mit der Fehlermeldung des Triggers abgewiesen. Ein Blick in die Tabelle zeigt, dass der neue Rabatt von 11 € dort nicht gespeichert wurde.

Als Nächstes sollen drei Datensätze auf einmal an die Tabelle angefügt werden, einer davon mit einem zu hohen Rabatt. Hierfür verwenden wir die seit SQL Server 2008 mögliche Syntax des *Table Row Constructors*.

```
INSERT INTO dbo.tblBestellpositionen
(BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
VALUES
(147, 108, '350.0', '0,19', 1, '0'),
(147, 109, '350.0', '0,19', 1, '5.0'),
(147, 110, '350.0', '0,19', 1, '11.0');
```

Auch der *INSERT*-Vorgang wurde abgebrochen und keiner der drei Datensätze in der Tabelle gespeichert. Den Grund hierfür liefert der letzte Datensatz, der einen Rabatt von 11 € aufweist. Den Hinweis hierzu zeigt die Meldung aus Abbildung 12.4.

Vielleicht finden Sie die Vorgehensweise des Triggers zu rigoros und Sie möchten wenigstens die Datensätze mit korrektem Rabatt speichern. Wobei der Benutzer natürlich darüber informiert werden sollte, wenn es Datensätze mit zu hohem Rabatt gibt und diese deshalb nicht angelegt oder geändert wurden. Auch diese Anforderung lässt sich mit einem *INSTEAD OF*-Trigger realisieren.



**Abbildung 12.4:** Die Eingabeverweigerung eines *INSTEAD OF*-Triggers

Die neue Logik unterscheidet sich nur an zwei Stellen von der ersten Variante. Der erste Unterschied besteht in der Verarbeitung der Datensätze mit zu hohem Rabatt. Liefert die Prüfung

## Kapitel 12 Trigger

auf den höchsten vergebenen Rabatt bei dieser Version einen Wert größer 10 €, wird die Anzahl der Datensätze mit zu hohen Rabatten in der Tabelle *inserted* ermittelt und das Ergebnis in einer Variablen namens *@AnzahlZuHoherRabatt* gespeichert. Diese Variable ist die Basis für die darauf folgende Meldung, die mit der Anweisung *RAISERROR* ausgegeben wird.

```
IF (SELECT Max(Rabatt) FROM inserted) >= 10
BEGIN
    -- Speichern Anzahl Datensätze mit zu hohen Rabatt
    SELECT @AnzahlZuHoherRabatt = COUNT(*) FROM inserted WHERE Rabatt > 10;
    -- Ausgabe der Meldung
    SET @Meldung = N'Es wurden ' + CAST(@AnzahlZuHoherRabatt as nvarchar(5)) +
        N' Bestellposition(en) nicht gespeichert, da
diese einen zu hohen Rabatt aufweisen!';
    RAISERROR (@Meldung, 16, 1);
END
```

Den zweiten Unterschied gibt es bei der Verarbeitung der tatsächlichen SQL-Anweisung. Hier werden jetzt nur die Datensätze mit korrektem Rabatt per *INSERT* oder *UPDATE* verarbeitet. Da Sie an einer Tabelle nur einen *INSTEAD OF*-Trigger definieren können, müssen Sie die erste Variante mit dem neuen Quellcode überschreiben. Hierfür verwenden Sie wieder den *ALTER*-Befehl:

```
ALTER TRIGGER dbo.tr_tblBestellpositionenInsteadOfInsertUpdate
ON dbo.tblBestellpositionen INSTEAD OF Insert, Update
AS
BEGIN
    SET NOCOUNT ON;
    -- Variable für Anzahl Datensätze mit ungültigem Rabatt
    DECLARE @AnzahlZuHoherRabatt int = 0;
    -- Variable für die Meldung bei Datensätzen mit ungültigem Rabatt
    DECLARE @Meldung nvarchar(255);
    -- Rabatt geändert?
    IF UPDATE(Rabatt)
    BEGIN
        -- Ausgabe Datensätze mit zu hohen Rabatt
        IF (SELECT Max(Rabatt) FROM inserted) > 10
        BEGIN
            -- Speichern Anzahl Datensätze mit zu hohen Rabatt
            SELECT @AnzahlZuHoherRabatt = COUNT(*) FROM inserted WHERE Rabatt > 10;
            -- Ausgabe der Meldung
            SET @Meldung = N'Es wurden ' + CAST(@AnzahlZuHoherRabatt as nvarchar(5)) +
                N' Bestellposition(en) nicht gespeichert, da
diese einen zu hohen Rabatt aufweisen!';
            RAISERROR (@Meldung, 16, 1);
        END
    END
    -- INSERT oder UPDATE
    IF (SELECT Count(*) FROM deleted) = 0
    BEGIN
        -- Nur die mit guten Rabatt einfügen
        INSERT dbo.tblBestellpositionen
        (BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
        SELECT BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt
```

```

FROM inserted
  WHERE inserted.Rabatt <= 10;
END
ELSE
BEGIN
  -- Nur die mit gutem Rabatt ändern
  UPDATE dbo.tblBestellpositionen
  SET BestellungID = inserted.BestellungID, ArtikelID = inserted.ArtikelID,
      Einzelpreis = inserted.Einzelpreis, Mehrwertsteuersatz = inserted.
Mehrwertsteuersatz,
      Menge = inserted.Menge, Rabatt = inserted.Rabatt
  FROM inserted INNER JOIN dbo.tblBestellpositionen
      ON inserted.BestellpositionID = dbo.tblBestellpositionen.
BestellpositionID
  WHERE inserted.Rabatt <= 10;
END
END

```

Testen Sie diesen Trigger wieder mit der bereits eben verwendeten *INSERT*-Anweisung, werden Sie feststellen, dass nur ein Datensatz nicht im Ergebnis enthalten ist (siehe Abbildung 12.5 und Abbildung 12.6).

## 12.4.2 AFTER-Trigger erstellen

Der *AFTER*-Trigger wird nach der eigentlichen Datenmanipulation ausgeführt. Er beinhaltet in der Regel Ergänzungen zur Datenverarbeitung, beispielsweise das Speichern von Änderungsinformationen, bestehend aus dem Zeitpunkt der Änderung und dem Anmeldenamen des Benutzers, der die Änderung durchgeführt hat. Genau dieses Szenario wird unser Beispiel für den *AFTER*-Trigger sein. Eine neue Geschäftsregel schreibt vor, dass in der Tabelle *tblBestellpositionen* Informationen zur letzten Änderung zu speichern sind. Die Tabelle wird hierzu mit zwei neuen Spalten ergänzt:

- » *LetzteAenderungVon* vom Datentyp *nvarchar(255)* speichert den Anmeldenamen des Benutzers, der die Änderung durchgeführt hat.
- » *LetzteAenderungAm* vom Datentyp *datetime* speichert den Zeitpunkt der letzten Änderung.

Sie können die Tabelle *tblBestellpositionen* entweder manuell im Entwurfsmodus mit den beiden Spalten ergänzen oder aber Sie führen in einem neuen Abfragefenster diese Anweisung aus:

```

ALTER TABLE dbo.tblBestellpositionen
ADD LetzteAenderungVon nvarchar(255), LetzteAenderungAm datetime;

```

Die Werte zu den Änderungsinformationen liefert ein *AFTER*-Trigger. Dieser soll sowohl bei einer *INSERT*- wie auch bei einer *UPDATE*-Aktion die Änderungsinformationen in die beiden Spalten eintragen.

Öffnen Sie also ein neues Abfragefenster, um das Skript zum Erstellen eines *AFTER*-Triggers einzugeben. Das Skript beginnt wieder mit der Anweisung *CREATE TRIGGER*, der Angabe des

## Kapitel 12 Trigger

Schemas und der Namensvergabe, gefolgt von der Zuweisung zur Tabelle. Als Typ geben Sie nun *AFTER* ein und ergänzen die Anweisung noch mit der Angabe der Aktionen, bei denen der *AFTER*-Trigger ausgelöst werden soll.

Das Schlüsselwort *AS* leitet wie gewohnt die Programmlogik ein, die auch hier in einem *BEGIN...END*-Block enthalten ist. Mal abgesehen von der Standardanweisung *SET NOCOUNT ON*, besteht die Logik dieses Triggers lediglich aus einer einzigen *UPDATE*-Anweisung.

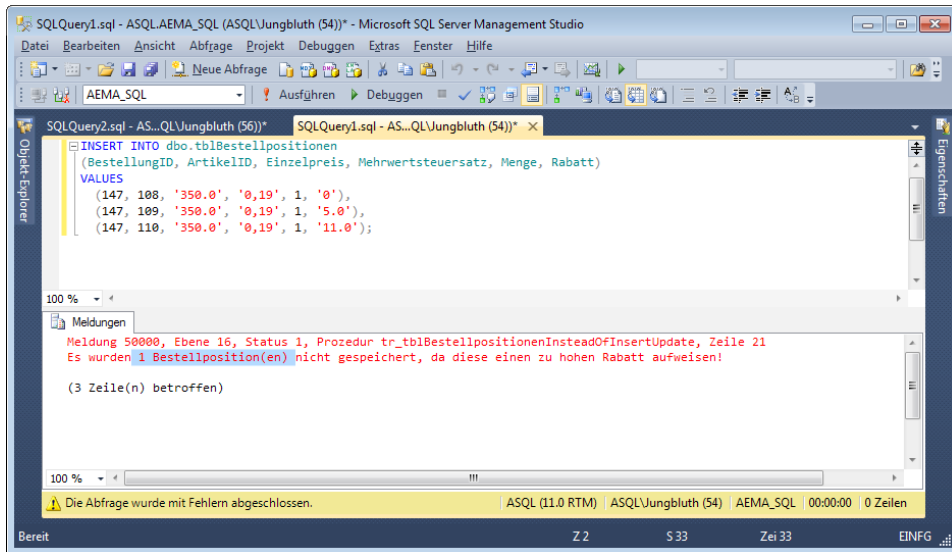


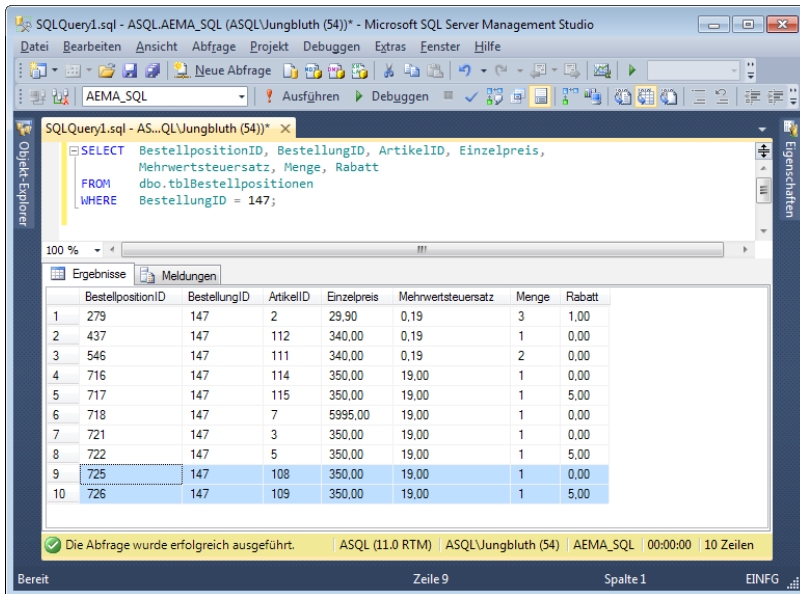
Abbildung 12.5: Die Schlechten ins Kröpfchen ...

Diese aktualisiert die verarbeiteten Datensätze der Tabelle *tblBestellpositionen* mit den Änderungsdaten, wobei die entsprechenden Datensätze über die virtuelle Tabelle *inserted* ermittelt werden.

```
CREATE TRIGGER dbo.tr_tblBestellpositionenAfterInsertUpdate
ON dbo.tblBestellpositionen AFTER Insert, Update
AS
BEGIN
    SET NOCOUNT ON;
    UPDATE dbo.tblBestellpositionen
    SET LetzteAenderungVon = SUSER_SNAME(), LetzteAenderungAm = GETDATE()
    FROM inserted INNER JOIN dbo.tblBestellpositionen
    ON inserted.BestellpositionID = dbo.tblBestellpositionen.
    BestellpositionID;
END
```

Den Zeitpunkt der letzten Änderung liefert Ihnen die bereits bekannte Systemfunktion *GETDATE()*. Auch die Information zum Benutzer ist das Ergebnis einer Systemfunktion: *SUSER\_SNAME()* ermittelt den Anmeldenamen des angemeldeten Benutzers.





SQLQuery1.sql - AS...QLVungbluth (54)) - Microsoft SQL Server Management Studio

Objekt-Explorer

SQLQuery1.sql - AS...QLVungbluth (54))

```

SELECT BestellpositionID, BestellungID, ArtikelID, Einzelpreis,
Mehrwertsteuersatz, Menge, Rabatt
FROM
dbo.tblBestellpositionen
WHERE
BestellungID = 147;

```

100 %

Ergebnisse Meldungen

	BestellpositionID	BestellungID	ArtikelID	Einzelpreis	Mehrwertsteuersatz	Menge	Rabatt
1	279	147	2	29,90	0,19	3	1,00
2	437	147	112	340,00	0,19	1	0,00
3	546	147	111	340,00	0,19	2	0,00
4	716	147	114	350,00	19,00	1	0,00
5	717	147	115	350,00	19,00	1	5,00
6	718	147	7	5995,00	19,00	1	0,00
7	721	147	3	350,00	19,00	1	0,00
8	722	147	5	350,00	19,00	1	5,00
9	725	147	108	350,00	19,00	1	0,00
10	726	147	109	350,00	19,00	1	5,00

Die Abfrage wurde erfolgreich ausgeführt. ASQL (11.0 RTM) ASQLVungbluth (54) AEMA\_SQL 00:00:00 10 Zeilen

Bereit Zeile 9 Spalte 1 EINFÜG

Abbildung 12.6: die Guten ins Töpfchen!

Drücken Sie nun die Taste *F5*, um das T-SQL-Skript auszuführen und den Trigger anzulegen. Anschließend geben Sie in einem neuen Abfragefenster diese SQL-Anweisungen ein:

```

UPDATE dbo.tblBestellpositionen SET Menge = 3 WHERE BestellpositionID = 279;
SELECT BestellpositionID, BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz,
Menge, Rabatt,
        LetzteAenderungVon, LetzteAenderungAm
FROM dbo.tblBestellpositionen WHERE BestellpositionID = 279;

```

Der Datensatz der Bestellposition mit der ID 279 wurde nicht nur mit der Menge 3 aktualisiert, sondern auch durch den Trigger mit den Änderungsdaten ergänzt (siehe Abbildung 12.7).

Dasselbe funktioniert mit der folgenden *INSERT*-Anweisung, wie Abbildung 12.8 beweist:

```

INSERT INTO tblBestellpositionen (BestellungID, ArtikelID, Einzelpreis,
Mehrwertsteuersatz, Menge)
VALUES (147, 110, '59.95', '0.19', 1);

```

Die Programmierung eines *AFTER*-Triggers ist nicht so aufwendig wie die eines *INSTEAD OF*-Triggers. Schließlich ersetzt er nicht die tatsächliche Aktion, sondern er ergänzt diese nur. In einem *INSTEAD OF*-Trigger sind Sie je nach Programmlogik gezwungen, die eigentliche Anweisung im Trigger nochmals zu programmieren – wie in unseren Beispielen zum *INSTEAD OF*-Trigger.

Wäre es da nicht sinnvoller, die Prüfungen vom *INSTEAD OF*-Trigger in den *AFTER*-Trigger zu verlagern? Ein *AFTER*-Trigger wird zwar nach der Datenverarbeitung ausgeführt, jedoch könnte

## Kapitel 12 Trigger

bei einem negativen Ergebnis einer Prüfung der Befehl *ROLLBACK* die Aktion abbrechen und die Tabelle in den Zustand vor der Verarbeitung der Daten zurücksetzen. Das Ergebnis wäre gleich einem Abbruch in einem *INSTEAD OF*-Trigger: Die Tabelle hat denselben Zustand wie vor der Datenverarbeitung.

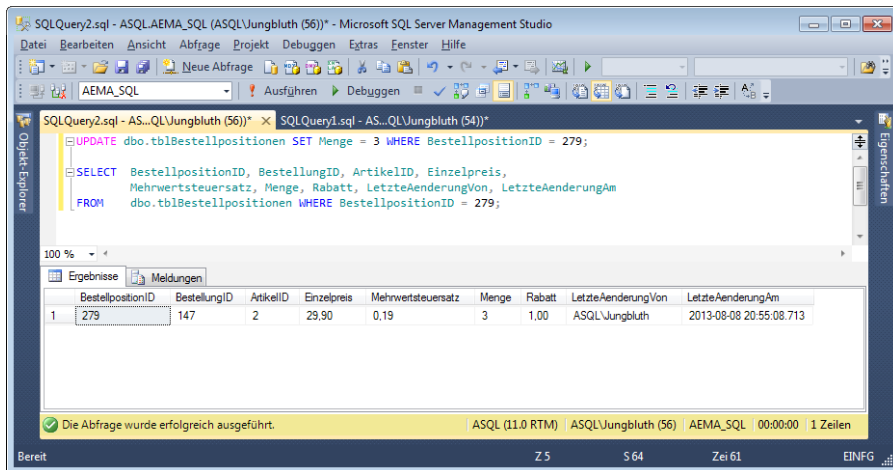


Abbildung 12.7: Der AFTER-Trigger bei einer UPDATE-Anweisung

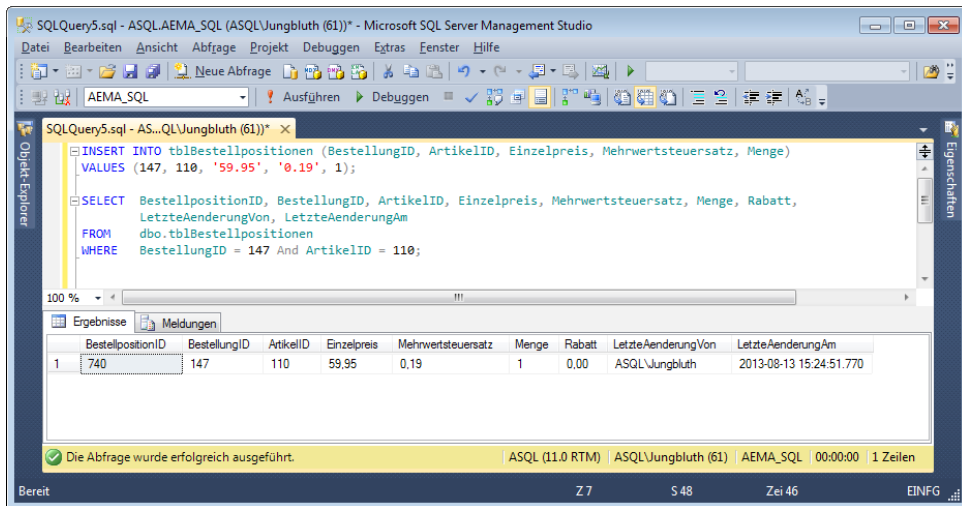


Abbildung 12.8: Der AFTER-Trigger bei einer INSERT-Anweisung

Beantworten wir die Frage anhand unseres *INSTEAD OF*-Triggers *tr\_tblBestellpositionenInstead-OfInsertUpdate*. Die Logik von Variante 1 könnte durchaus ein *AFTER*-Trigger übernehmen. Ist der Rabatt zu hoch, sorgt ein *ROLLBACK* dafür, dass die Verarbeitung abgebrochen und die Ta-

belle in den Zustand vor der Datenverarbeitung gesetzt wird. Die zweite Variante des *INSTEAD OF*-Triggers lässt sich jedoch nicht mit einem *AFTER*-Trigger realisieren. Hierbei sollten wenigstens die Datensätze mit korrektem Rabatt in der Tabelle gespeichert werden. Ein *ROLLBACK* wirkt sich jedoch immer auf die gesamte Aktion aus, weshalb die Tabelle am Ende keine der verarbeiteten Datensätze enthält.

Nicht nur die Logik ist entscheidend, ob Sie die Prüfungen im *INSTEAD OF*-Trigger oder im *AFTER*-Trigger durchführen. Ein weiterer wichtiger Faktor ist die Performance. Es macht durchaus einen Unterschied, ob Sie die Gültigkeit einer Datenverarbeitung am Anfang durch einen *INSTEAD OF*-Trigger oder erst am Ende mit einem *AFTER*-Trigger prüfen, um dann gegebenenfalls die Verarbeitung abubrechen.

## 12.5 Trigger bei MERGE und TRUNCATE TABLE

Seit SQL Server 2008 gibt es die SQL-Anweisung *MERGE*, mit der Sie die Daten einer Quelle mit den Daten einer Zieltabelle abgleichen können. Je nach dem Stand der Daten werden hierbei in der Zieltabelle Daten gelöscht, geändert oder der Tabelle hinzugefügt – mehr dazu in Kapitel »T-SQL-Grundlagen«, Seite 221.

Bleibt die Frage, ob *MERGE*-Anweisungen auch Trigger auslösen können. Nicht direkt, denn der *MERGE*-Befehl selbst ändert keine Daten. Er steuert lediglich, für welche Datensätze ein *INSERT*, *UPDATE* oder *DELETE* ausgeführt wird. Haben Sie also beispielsweise eine Tabelle mit einem *UPDATE*-Trigger und Sie führen an dieser Tabelle einen *MERGE*-Befehl aus, der unter anderem Daten ändert, löst dies den *UPDATE*-Trigger der Tabelle aus.

Die gesamten Daten einer Tabelle können Sie mit der *DELETE*-Anweisung löschen oder auch per *TRUNCATE TABLE*. Dabei sollten Sie bedenken, dass ein *DELETE*-Trigger bei der Anweisung *TRUNCATE TABLE* nicht ausgelöst wird. Die Anweisung *TRUNCATE TABLE* löscht nicht wie *DELETE* die Daten einer Tabelle, sondern lediglich die Zuordnungen der Datenseite zu einer Tabelle. Mehr zu *TRUNCATE TABLE* lesen Sie im Kapitel »T-SQL-Grundlagen«, Seite 221.

Mit den Triggern wäre nun auch das letzte verfügbare SQL Server-Objekt beschrieben. Trigger mögen nicht unumstritten sein. Es gibt gute Gründe für und gegen den Einsatz von Triggern. Die Vor- und Nachteile haben Sie in diesem Kapitel kennengelernt. Die Entscheidung, ob Sie Trigger verwenden oder nicht, liegt nun bei ganz Ihnen.