TSQL mit NUnit testen

Einfacher testen

Unit-Tests für TSQL – geht das so einfach wie bei normalem C#-Code? Dieser Artikel gibt einen kurzen Überblick über die vorhandenen Tools und erklärt, wie Sie TSQL-Code mit NUnit testen können.

Auf einen Blick

Dipl.-Ing. **Martin Kortstiege** arbeitet als Senior Software Engineer bei der adesso AG in Dortmund. Er entwickelt seit 20 Jahren Software mit Microsoft-Technologien. Sie erreichen ihn unter martinkortstiege@web.de.

Christian Vogt arbeitet zurzeit als freier Softwarearchitekt. Er entwickelt seit rund 30 Jahren Software, über die Anfänge mit Assembler auf einem Apple IIe bis hin zu modernen Multicore-Prozessoren und C#. Sie erreichen ihn unter polarkreis@web.de.

Inhalt

› Kostenlose und kommerzielle Tools für TSQLUnit-Tests.

› Anleitung zum Testen von TSQL-Code mit NUnit.

dnpCode

A1012TestTSQL

Für Programmiersprachen innerhalb des .NET Frameworks stehen verschiedene kostenlose und kommerzielle Test-Frameworks zur Verfügung. TSQLUnit [1] ist eine Möglichkeit, Unit-Tests direkt in die Datenbank zu integrieren. TSQLUnit bietet jedoch keine grafische Oberfläche für die Testausführung und verändert die Datenbankstruktur. Tests müssen mühsam in TSQL erstellt werden. Es werden Prozeduren und Tabellen in der Datenbank angelegt. Auch für die Darstellung der Ergebnisse ist keine grafische Oberfläche vorhanden.

Für uTSQL [2] gilt das Gleiche wie schon bei TSQLUnit beschrieben.

Kommerzielle Tools

Visual Studio Database Tests sind eine komfortable Alternative. Für alle .NET-Entwickler ist hier der Einarbeitungsaufwand sehr gering. Tests können in Visual Studio erstellt und ausgeführt werden. Nachteil dieser Lösung: Die Datenbanktests sind nur mit der Team Foundation Database Edition von Visual Studio möglich.

Für alle, denen die freien Tools zu umständlich sind und die nicht über die Visual Studio Team Foundation Database Edition verfügen, zeigt dieser Artikel eine Alternative mit dem ebenfalls kostenlosen NUnit [3]. Diese Methode erlaubt es Ihnen, weitgehend in der vertrauten Umgebung zu arbeiten. Selbstverständlich können Sie für die Unit-Tests auch andere .NET-Testframeworks wie zum Beispiel xUnit [4] oder MS-Test verwenden.

Das Beispiel

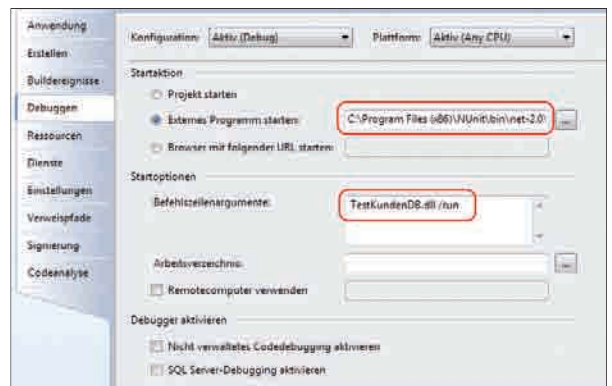
Die Beispieldaten (Abbildung 1) sind bewusst einfach gehalten. Natürlich sollten in einem echten Szenario die Informationen nicht im Klartext vorliegen und ebenso die TANs einen besseren Mix aufweisen.

Die Datenbank besteht aus zwei Tabellen und drei gespeicherten Prozeduren (stored procedures), die getestet werden sollen (Listing 1).

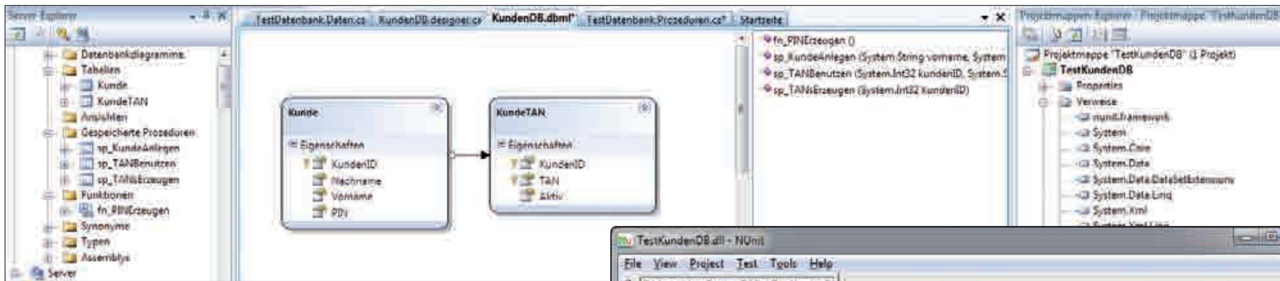
Für die Beispielanwendung *Kundendatenbank* werden folgende Regeln vereinbart:

- Datenänderungen erfolgen nur über gespeicherte Prozeduren.
- Die Anwendung authentifiziert sich mit den Anmeldedaten eines SQL-Benutzers. Benutzerrechte werden in der Applikation verwaltet.
- Für Tabellen bestehen nur Leseberechtigungen.
- Die Datenbankstruktur darf für einen Unit-Test nicht verändert werden.
- Die Inhalte der Tabellen dürfen durch die Unit-Tests nicht verändert werden, damit die Tests auch auf dem Produktivsystem durchgeführt werden können. Anmerkung: Die Kopie einer Produktionsdatenbank ist als Spielwiese dem Original selbstverständlich stets vorzuziehen.

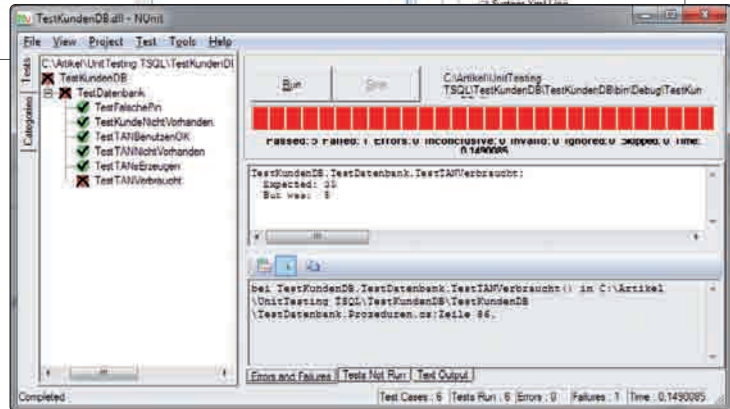
[Abb. 1] Datenbankdiagramm des Beispiels.



[Abb. 2] Die Debug-Einstellungen.



[Abb. 3] Tabellen, Prozeduren und Funktionen werden per Drag-and-drop hinzugefügt.



[Abb. 4] Die Benutzeroberfläche von NUnit.

Vorgehensweise

Legen Sie in Visual Studio ein neues *Classlibrary Projekt* an. Dem Projekt fügen Sie eine Referenz auf das Testframework hinzu, im Beispiel also eine Referenz auf *nunit.framework.dll*. Nun müssen Sie noch die Debug-Einstellungen anpassen, damit Sie die Tests direkt mit F5 starten können (Abbildung 2). Aktivieren Sie die Option *Externes Programm starten*, und tragen Sie den Pfad zur *NUnit.exe* ein. Im Feld *Befehlszeilen-Argumente* wird der Assembly-Name (inklusive der Endung *.dll*), gefolgt vom Parameter */run* eingetragen.

Der Zugriff auf die Datenbank erfolgt mit Link to SQL. Hierzu fügen Sie dem Projekt ein neues Link-to-SQL-Element hinzu. Mit Link to SQL sparen Sie sich viel Tipparbeit und bekommen die Typsicherheit quasi frei Haus. Die einzelnen Datenbanktabellen, Prozeduren und Funktionen werden im Visual Studio Datenbank Designer nun durch Drag-and-drop hinzugefügt, siehe Abbildung 3.

Als Nächstes erstellen Sie eine neue Klasse für die Tests. Die *Setup*-Methode wird einmalig vor dem Start der Unit-Tests aufgerufen. Hier wird die Verbindung zur Datenbank hergestellt und eine neue Transaktion eingerichtet (Listing 2). Die Transaktion gewährleistet, dass alle Datenänderungen nach der Testausführung wieder verworfen werden. Anschließend werden Testdaten erzeugt.

Hierdurch wird sichergestellt, dass die Tests mit konsistenten Daten stattfinden.

Nach der Durchführung aller Tests wird die *Teardown*-Methode (siehe Listing 1) aufgerufen. In dieser Methode platzieren Sie die Aufräumarbeiten.

```
KundenContext.Transaction.
  Rollback();
```

Listing 1

Zu testende Prozeduren.

```
ALTER PROCEDURE [dbo].[sp_TANBenutzen]
    @KundenID int,
    @PIN Varchar(10), @TAN Varchar(6)
AS
BEGIN
    DECLARE @result int
    DECLARE @dbPIN varchar(10)
    DECLARE @dbCount int

    SELECT @DBPIN = PIN FROM Kunde WHERE KundenID = @KundenID
    IF (@dbPIN is null)
      BEGIN
        RETURN 2
      END

    IF (not @dbPIN = @PIN)
      BEGIN
        RETURN 3
      END

    SELECT @dbCount = count(*) FROM KundeTAN
       WHERE [KundenID] = @KundenID
       AND [TAN] = @TAN
    IF ( @dbCount = 0)
      RETURN 4
    IF ((select top 1 [Aktiv] FROM KundeTAN
        WHERE [KundenID] = @KundenID AND [TAN] = @TAN ) = 0)
      RETURN 5
    UPDATE KundeTAN SET [Aktiv] = 0
        WHERE [KundenID] = @KundenID AND [TAN] = @TAN
    RETURN 1
END

ALTER PROCEDURE [dbo].[sp_TANsErzeugen]
    @KundenID int
AS
BEGIN
    DECLARE @I as int
    SET @I = 0
    WHILE @I <50
      BEGIN
        SET @I = @I+1
        INSERT INTO [dbo].[KundeTAN]
             ([KundenID], [TAN], [Aktiv])
        VALUES
             (@KundenID, convert(varchar(10) , @I), 1)
      END
  RETURN 1
END
```

Listing 2

FixtureSetup & Teardown.

```
using System;
using System.Linq;
using System.Reflection;
using NUnit.Framework;
using TestKundenDB;

namespace TestKundenDB
{
  [TestFixture]
  public partial class TestDatenbank
  {
    string _ConnectionString = "Data Source=
      (local);Initial Catalog=Kunden;User ID=
      UnitTest_Admin;Password=dotnetpro";

    KundenDBDataContext _KundenContext;
    int _testKundenID = 0;

    [TestFixtureSetUp]
    public void FixtureSetup()
    {
      _KundenContext = new KundenDBDataContext(
        _ConnectionString);
      _KundenContext.Connection.Open();
      _KundenContext.Transaction =
        _KundenContext.Connection.BeginTransaction();

      //Initial einen TestKunden und TANs anlegen
      _testKundenID = KundeErzeugen();
      TANsErzeugen(_testKundenID, false);
    }

    [TestFixtureTearDown]
    public void TearDown()
    {
      _KundenContext.Transaction.Rollback();
      _KundenContext.Connection.Close();
      _KundenContext.Dispose();
    }
  }
```

Listing 3

Testdaten erzeugen.

```
private int TestKundeErzeugen()
{
  var testKundenID = _KundenContext.Kunde.Count() + 1;
  if (testKundenID != 1)
  {
    testKundenID = _KundenContext.Kunde.Max((p => p.KundenID)) + 1;
  }
  var c = new Kunde();
  c.KundenID = testKundenID;
  c.Vorname = "Theo";
  c.Nachname = "Waigel";
  c.PIN = "12345";

  _KundenContext.Kunde.InsertOnSubmit(c);
  _KundenContext.SubmitChanges();
  return testKundenID;
}

private void TestTANsErzeugen(int testKundenID, bool aktiv)
{
  for (var i = 1; i <= 5; i++)
  {
    var c = new KundeTAN();
    c.KundenID = testKundenID;
    c.TAN = i.ToString();
    c.Aktiv = aktiv;
    _KundenContext.KundeTAN.InsertOnSubmit(c);
  }
  _KundenContext.SubmitChanges();
}
```

Rollback verwirft alle durch die Tests durchgeführten Änderungen an den Daten. Jetzt noch die Verbindung schließen, und die Testausführung ist beendet.

Wie funktionieren die Tests?

Alle Testmethoden funktionieren nach dem gleichen Schema (Listing 3). Im ersten Schritt werden die Daten in einen definierten Zustand gebracht.

Im Beispiel wird hierzu die Methode *TestKundeErzeugen* und teilweise auch *TestTANsErzeugen* aufgerufen. Danach wird die zu testende stored procedure aufgerufen.

```
_KundenContext.sp_TANsErzeugen(_testKundenID);
_KundenContext.SubmitChanges();
```

Anschließend wird das Ergebnis überprüft.

In der Beispielanwendung wird exemplarisch die Anzahl der erzeugten Datensätze geprüft.

```
Assert.That(
  _KundenContext.KundeTAN.Where(
  p => p.KundenID == _testKundenID
  & p.Aktiv).Count(), Is.EqualTo(50));
```

In den anderen Tests werden die Rückgabewerte der Prozedur verglichen.

```
Assert.That(
  (ReturnWerte)_KundenContext.
  sp_TANBenutzen(_testKundenID,
  "12345", "4"), Is.EqualTo(
  ReturnWerte.KundeNichtVorhanden));
```

Tests ausführen

Mit einem Druck auf die Funktionstaste [F5] starten Sie den NUnit-Test-Runner. Die grafische Benutzeroberfläche von NUnit wird geöffnet, und die Tests werden automatisch ausgeführt (Abbildung 4).

Die Testergebnisse werden hier sehr anschaulich in einer TreeView (links im Bild) dargestellt. Details zu fehlgeschlagenen Tests werden im rechten Teil des Fensters angezeigt. Hier können Sie per Mausklick auch die Code-Ansicht einschalten – Sie sehen dann sofort die Stelle, an welcher der Test gescheitert ist.

Benutzer von ReSharper [7] können die Tests auch direkt in Visual Studio über das Kontextmenü starten.

Tests automatisieren

Nutzen Sie beispielsweise Nant [8] zum Automatisieren des Build-Prozesses, können Sie die Testjobs einfach ausführen lassen. Der Task *<NUnit2>* stellt sich mitunter als sperrig heraus, daher empfiehlt sich eine Lösung wie in Listing 4 beschrieben. Hier werden im Verzeichnis *TestResults* für jede Solution, die ausgeführt wird, drei Dateien angelegt, wobei zumindest die **-err.xml* möglichst 0 Byte groß sein sollte.

Damit die Tests unter .NET 4.0 funktionieren, muss das Runtime-Framework mit der Option */framework=4.0* versehen werden. Die Target-Definition ist leicht, jedoch umfangreich. Sie finden das vollständige Buildscript auf der Heft-DVD.

Fazit

TSQL-UnitTesting mit NUnit ist eine einfache Methode, die mit geringem Aufwand die Softwarequalität stark verbessern kann. Die Automatisierung der Tests in einem Entwicklungsprozess mit Continuous Integration schließt hier die Lücke einer ungetesteten Datenbank. **[bl]**

[1] TSQLUnit, www.dotnetpro.de/SL1012TestTSQL1
[2] uTSQL, www.dotnetpro.de/SL1012TestTSQL2
[3] NUnit, www.nunit.org
[4] xUnit, http://xunit.codeplex.com
[5] UnitTesting C#, www.dotnetpro.de/SL1012TestTSQL3
[6] Wiki-Übersicht, www.dotnetpro.de/SL1012TestTSQL4
[7] ReSharper, www.jetbrains.com/resharper
[8] Nant, http://nant.sourceforge.NET

Listing 4

Nant job.

```
<exec program="${nunit-console}">
  <arg value="/nologo" />
  <arg value="/nodots" />
  <arg value="/framework=4.0" />
  <arg value="/xml:../TestResults/${purefilename}-result.xml" />
  <arg value="/out:../TestResults/${purefilename}-out.xml" />
  <arg value="/err:../TestResults/${purefilename}-err.xml" />
  <arg value="${content}" />
</exec>
```

Grundsätzliches zum Testen von TSQL

Für das Testen sind einige grundsätzliche Punkte zu beachten. Datenbankeinstellungen können bewirken, dass gleiche SQL-Anweisungen zu unterschiedlichen Ergebnissen oder zu Fehlern führen. Beispielsweise kann eine Datenbank bezüglich der Groß- und Kleinschreibung unterschiedlich konfiguriert sein. Ist die Sortierreihenfolge gleich *SQL_Latin1_General_CP1_CI_AS*, liefert der Ausdruck *'a' = 'A' true* zurück, bei der Einstellung *SQL_Latin1_General_CP1_CS_AS* wird *false* zurückgegeben.

Die Sicherheitseinstellungen können einen weiteren Fallstrick für Unit-Tests darstellen. Entwickler greifen in der Regel mit Administratorrechten beziehungsweise als Database-Owner auf eine Datenbank zu. Allerdings sollten Tests mit den gleichen Rechten wie im Produktivbetrieb durchgeführt werden, damit auch Fehler wie falsch vergebene Berechtigungen durch die Tests aufgedeckt werden. Eventuell müssen hier unterschiedliche Tests für die verschiedenen Rollen erstellt werden. Insgesamt sollten Sie sorgfältig darauf achten, dass sämtliche Einstellungen der Testdatenbank denen aus dem Produktionsbetrieb gleichen.

Konkurrierende Zugriffe zu testen ist ebenfalls problematisch – den entsprechenden Synchronisationsansatz müssen Sie selbst entwickeln. Beispielsweise wollen Sie Daten schreiben und mit einem anderen „Client" (sprich einer anderen Verbindung in einem separaten Test) wieder lesen. Ohne Synchronisation können Sie nicht vorhersagen, ob das Lesen vor oder nach dem Schreiben ausgeführt wird – die Reihenfolge der Testdurchführung ist nicht festgelegt. Dies wiederum führt dazu, dass beim Abfragen der Testergebnisse nicht simpel mit *Assert.That* geprüft werden sollte – schließlich wissen Sie ja nicht, ob die alte oder die neue Version des Datensatzes vorliegt.

Da es zu den best practises gehört, jeden Test möglichst isoliert und unabhängig von allen anderen durchzuführen, raten die Autoren davon ab, eine testübergreifende Synchronisation zu programmieren. Ein besserer Ansatz wäre es sicher, innerhalb eines Tests mehrere Threads zu starten und in je einem Thread eine Verbindung zu nutzen. Weiterführende Grundsatzinformationen finden Sie unter [5] und [6].

Experts in agile software engineering

Professional Scrum Master
mit Ken Schwaber
8. + 9.2.2011 in Frankfurt

Softwareentwicklung
Java, C#, Testgetriebene Entwicklung
Sicherheit

Projektmanagement
Scrum Training und Coaching
Geschwindigkeit

Beratung und Training
Qualität, Architektur, Technologie, Prozess
Qualität

Migration und Sanierung
Reengineering von Altsystemen
Werterhaltung

Lösungen
Kreditwirtschaft, Wertpapiergeschäft
Nachhaltigkeit

www.andrena.de
andrena objects ag
Albert-Nestler-Straße 11
76131 Karlsruhe
Telefon 0721 6105-122
Telefax 0721 6105-140