Testgetriebene Entwicklung mit Access

André Minhorst, Duisburg und Uwe Schäfer, Essen

Die Schlagwörter Extreme Programming (XP), Unit-Testing, Test Driven Development, Refactoring oder Pair Programming geistern durch die Entwicklerwelt. Dabei ist Extreme Programming der Oberbegriff für die anderen und fasst diese und mehr zu einer neuartigen Philosophie der Softwareentwicklung zusammen. Ziel der dahinter stehenden Konzepte sind Projekte, die von kleinen Entwicklerteams durchgeführt werden. Da die meisten Leser dieses Beitrags vermutlich allein entwickeln, stellt dieser Beitrag ein elementares Konzept von XP heraus: das Test Driven Development (TDD), zu deutsch testgetriebene Entwicklung.

Inhalt

1 Warum testgetrieben entwickeln? 1
2 Testgetriebene Entwicklung in der Praxis . 4
3 Beispielanwendung 7
4 Zusammenfassung und Ausblick 13

1 Warum testgetrieben entwickeln?

Extreme Programming ist ein Thema, mit dem man leicht mehrere hundert Seiten füllen könnte – wenn man nur die theoretischen Aspekte berücksichtigt.

Da dieser Platz leider nicht zur Verfügung steht, greifen wir die Teilbereiche auf, die auch Ein-Mann-Teams bei der Entwicklung von Access-Datenbanken Gewinn bringend nutzen können. Der Kern ist die „testgetriebene Entwicklung", eng damit verbunden sind die Begriffe „Unit Test" und „Refactoring".

Der vorliegende Beitrag soll möglichst praxisnah die Vorteile der testgetriebenen Entwicklung beschreiben. Dennoch sind einige einführende Worte erforderlich.

Hinweis

Im Internet und in der Literatur finden Sie eine Menge theoretischer Abhandlungen über diesen Themenkomplex. Wir möchten Ihnen neben den theoretischen Grundlagen ein Tool vorstellen, das Sie für die testgetriebene Entwicklung mit Access einsetzen können; außerdem lernen Sie, wie Sie dieses installieren und wie Sie Ihre ersten Schritte mit der testgetriebenen Entwicklung damit durchführen. ■

1.1 Test a little, code a little

Diese Entwicklungsmethode erfordert vom Programmierer eine Menge Disziplin, da sie voraussetzt, dass für jede Funktion einer Anwendung zunächst ein Test geschrieben wird.

Damit Sie sehen, dass der Test funktioniert – was der erste und wichtigste Schritt bei der testgetriebenen Entwicklung ist – schreibt man einen Test, der beim ersten Start scheitert.

Erst dann implementieren Sie die eigentliche Funktion.

Durch erneuten Start des Tests wird dann verifiziert, dass die Implementierung den Anforderungen des Tests genügt, dieser also nicht mehr fehlschlägt.

Hinweis

Schreiben Sie niemals mehr als einen neuen Test gleichzeitig! Vermutlich kennen Sie das Problem, Funktionen zu einer Anwendung hinzufügen oder ändern zu wollen, die Anpassungen an mehr als einer Stelle erfordern. Die wiederum beeinflussen andere Programmfunktionen oder machen diese gar untauglich. Wenn Sie jeweils nur einen Test gleichzeitig hinzufügen oder ändern, halten Sie auch den durch diese Anforderungen verursachten Aufwand minimal. ■

Kleine Schritte, einfache Wege

Jeder Test soll auf möglichst einfache Weise erfüllt werden. Wenn ein Test fordert, dass eine Funktion den Eingangswert „andre" in die Zeichenkette „Andre" umwandelt, dann schreiben Sie einfach eine Funktion, die den gewünschten Wert hartcodiert zurückgibt – das reicht für den ersten Ansatz, denn damit ist ja der erste Test erfüllt! Wenn der zweite Test die Umwandlung eines zweiten, anderen Wertes einfordert, müssen Sie die Funktion natürlich anpassen, was Sie in dem Fall leicht mit einer bestimmten VB-Funktion tun können. Auf diese Weise stellen Sie sicher, dass die Definition der Anforderungen (durch den Test) möglichst vollständig ist und nicht von Ihrem Verständnis der Implementierung abhängt.

Einmal testen, immer testen

Natürlich bringt das ganze Testen nicht viel, wenn Sie einen Test nach erfolgreichem Bestehen aus den Augen verlieren und sich direkt dem nächsten Test zuwenden.

Deshalb fügen Sie jeden neuen Test zu den bereits erfüllten Tests hinzu und führen mit jedem neuen Test alle bestehenden Tests erneut durch. Auf diese Weise stellen Sie sicher, dass bereits erfüllte Anforderung durch neuen Code oder Codeänderungen unberührt bleiben.

Hinweis

Unit-Testing-Frameworks wie **accessUnit** bieten durch sogenannte Testsuites die Möglichkeit, Tests nach beliebigen Gesichtspunkten zusammenzufassen. So können Sie etwa alle Tests, die nicht den gerade in Arbeit befindlichen Code betreffen, zusammenfassen und beispielsweise einmal am Tag ausführen, um unvorhergesehene Defekte der Software frühzeitig zu erkennen. Die Tests, auf deren Basis Sie gerade entwickeln, fassen Sie ebenfalls zusammen. Da Sie damit häufig testen (was dem Grundprinzip der testgetriebenen Entwicklung entspricht), sollten diese Test möglichst schnell abgearbeitet werden. Je schneller ein Test abläuft, desto geringer ist die Wahrscheinlichkeit, dass Sie ihn einmal aus „Zeitnot" auslassen. ■

1.2 Automatisierung ist Trumpf

Nach den ersten Abschnitten fragen Sie sich vermutlich wie jeder andere, der sich erstmalig mit dieser Thematik auseinandersetzt, wie die Tests überhaupt ablaufen. Die Antwort ist: Sie werden – genau wie normale Anwendungen auch – programmiert, und zwar als Abfolge von Prüfungen bestimmter Ausdrücke.

Wenn Sie beispielsweise eine Funktion testen möchten, die zwei Zahlen addiert, dann vergleichen Sie einfach das Ergebnis dieser Funktion mit dem zu erwartenden Ergebnis. Und damit Sie sich nur um die Festlegung dieser Tests und die Eingabe der erwarteten Ergebnisse kümmern müssen, gibt es so genannte Test-Frameworks. Mehr darüber erfahren Sie später im praktischen Teil dieses Beitrags.

1.3 Refactoring – Alles bleibt besser

Der Begriff „Refactoring" ist eng mit der testgetriebenen Entwicklung verbunden. Refactoring ist eine Veränderung, Anpassung oder Verbesserung des Designs. Dabei müssen natürlich bestehende, durch Tests definierte Anforderungen auch nach dem Refactoring noch erfüllt werden.

Ein Ad-hoc-Programmierstil, der aus dem immer höheren Zeit- und Erfolgsdruck entsteht und möglicherweise auch im ersten Schritt zu einer lauffähigen Anwendung führt, garantiert großen Aufwand, wenn nachträglich zu behebende Fehler und/oder sich während der Entwicklung ändernde Anforderungen auftreten; auch die nachträgliche Optimierung einer Anwendung, die nicht die gewünschte Performance aufweist, führt sicher zu Kopfschmerzen beim Entwickler(team).

Die testgetriebene Entwicklung bietet wesentlich mehr Möglichkeiten, den bestehenden Code ohne Angst anzufassen: nämlich immer, wenn alle bis dato vorhandenen Tests zuverlässig laufen. Da Sie mit jedem Testlauf den neuen und alle bereits bestehenden Tests durchführen, erfahren Sie nicht nur, ob der neue Test erfolgreich ist oder scheitert, sondern auch, ob alles andere noch wie gewünscht funktioniert.

So können Sie den bestehenden und regelmäßig getesteten Code nach Lust und Laune refaktorisieren, solange – ja, solange die Änderungen nicht bewusst ein anderes Ergebnis für einen beliebigen Test zurückliefern sollen. Das fiele dann nicht mehr unter den Begriff „Refactoring"; stattdessen heißt die Devise: Erst den Test schreiben beziehungsweise anpassen und dann die Funktionalität ändern.

Wenn Sie beispielsweise einen Vorgang, der in mehreren getesteten Routinen auftritt, in eine eigene Funktion auslagern und von den jeweiligen Routinen aus aufrufen möchten, können Sie das natürlich, ohne die Tests zu ändern, denn Sie lagern ja nur ein paar Zeilen in eine Funktion aus (Mathematiker würden hier von „Ausklammern" sprechen).

Noch besser wäre allerdings, Sie würden vorher Tests schreiben, welche die ausgelagerte Funktion auf Herz und Nieren prüfen. Damit wären Sie wieder bei der kleinsten Einheit – der „Unit".

1.4 Unit Test – was heißt das?

Der Begriff „Unit Test" ist so eng mit der testgetriebenen Entwicklung verknüpft, weil beide sich auf die kleinstmögliche Einheit beziehen. Wenn Sie kleinste Einheiten testen möchten, dann ist damit nicht eine Anwendung, auch kein Teil einer Anwendung wie ein Formular oder eine Klasse gemeint, sondern ein elementarer Bestandteil davon – eine Eigenschaft, eine Methode oder ein Ereignis, kurz: die „Unit under Test".

Je kleiner die Einheiten sind, die Sie testen, desto schneller und leichter finden Sie fehlerhafte Stellen. Zumindest aber sollte es für jede testbare Schnittstelle Ihrer Klassen und Objekte einen oder mehrere Tests geben, die deren Funktionalität jederzeit sicherstellen können. Nur auf diese Weise können Sie sich auf das im vorherigen Abschnitt beschriebene „Refactoring" stürzen.

1.5 Alles auf einmal?

Bei jeder größeren Änderung oder Erweiterung sollten Sie alle vorhandenen Tests Ihrer Anwendung durchführen. Wichtig ist, dass jeder Aspekt Ihrer Anwendung für sich allein testbar ist, und zwar in beliebiger Reihenfolge, um Wechselwirkungen auszuschließen.

1.6 Dummys

Natürlich können Sie mit der testgetriebenen Entwicklung nicht nur Einheiten, sondern auch deren Interaktion testen – man spricht hier von Integrationstests. Das entspricht allerdings nicht dem Grundprinzip der testgetriebenen Entwicklung. Um dennoch die Wechselwirkung zwischen Klassen testen zu können, verwendet man verschiedene Arten von Dummies.

Das Testen ohne Wechselwirkung ist in manchen Fällen nicht so einfach, da auch die Interaktion zwischen Klassen getestet werden muss. Dabei gibt es zwei Varianten:

Im ersten Fall benötigt die erste Klasse eine Eigenschaft oder Funktion der zweiten Klasse, um einen bestimmten Wert zu ermitteln. Im Idealfall lässt sich die zweite Klasse dabei durch eine Dummy-Implementierung ersetzen, die den gewünschten Wert liefert

Im zweiten Fall löst die Interaktion der beiden Klassen die Änderung einer Eigenschaft oder Verhaltensweise der zweiten Klasse aus, die für den Test der ersten Klasse wichtig ist. Will man für die zweite Klasse einen Dummy verwenden, reicht es nicht aus, wenn dieser einfach auf Anfrage einen bestimmten Wert liefert. Stattdessen muss man die Auswirkung der Interaktion zwischen den Klassen prüfen können. Ein solcher Dummy ist ein wenig komplizierter und heißt in der Fachsprache „Mock".

Hinweis

Mocks und Stubs werden im Rahmen dieses Beitrags nicht weiter erläutert. ■

1.7 Testdaten

Elementar wichtig für Tests sind Testdaten. Optimal wäre natürlich ein „echter" Testdatenbestand; wenn es sich um eine neue Anwendung handelt, ist dieser aber in der Regel nicht verfügbar. Um für alle Tests die gleiche Ausgangspositi-

on zu schaffen, sollten Sie die vorhandenen Daten vorher auf einen fest definierten Stand bringen – am besten jedes Mal neu.

Dazu gibt es zwei Möglichkeiten:

- Sie erstellen die Daten mit jedem Test durch geeignete SQL-Skripte neu und löschen diese anschließend wieder. Testframeworks enthalten geeignete Methoden, um die notwendigen Anweisungen unterzubringen.

- Wenn die zu entwickelnden Klassen selbst Methoden enthalten, um die notwendigen Daten anzulegen, stellen Sie die Testdaten doch einfach im Rahmen der Tests der entsprechenden Klassen zusammen! Vermutlich finden sich auch Methoden zum Löschen von Daten in den Klassen, die Sie zum Entfernen der Testdaten verwenden können.

1.8 Zusammenspiel und Vorzüge

Die vorhergehenden Abschnitte machen bereits deutlich, dass testgetriebene Entwicklung, Unit Tests und Refactoring ein eingespieltes „Team" sein müssen, wenn sie zum Erfolg führen sollen.

Zusammengefasst haben Sie die folgenden Vorzüge kennen gelernt:

- Vorausschauend planen: Wenn Sie vor jedem Programmierschritt einen Test erstellen, setzen Sie sich intensiver mit dem Ziel auseinander.

- Die Wahrscheinlichkeit, nach der Erstellung einigen Codes festzustellen, dass Sie eigentlich am Ziel vorbeiprogrammiert haben, ist geringer.

- Schritt für Schritt statt Entwicklung im Multitasking-Stil: Erst wenn der vorherige Test positiv ausfällt (und die damit verbundene Code-Änderung keine älteren Tests scheitern lässt), dürfen Sie einen neuen Test und neue Funktionalität hinzufügen. Vorteil: Sie arbeiten immer nur an einer Baustelle; wenn ein oder mehrere Tests durch neuen Code fehlschlagen, wissen Sie sofort, woran es liegt.

- Absicherung: Dadurch, dass Sie mit jedem neuen Test auch alle anderen Tests ausführen, sind Sie immer sicher, dass Sie durch Hinzufügen neuer Funktionen oder Refactoring nichts Funktionierendes zerstören.

- Durch den automatisierten Ablauf der Tests können Sie sich jederzeit davon überzeugen, dass noch alles entsprechend der Spezifikation funktioniert.

Hinzu kommen die folgenden Vorteile:

- Mit jedem Test stellen Sie sich eine neue Aufgabe, die Sie schnell erfüllen können – außer, Sie haben den Anspruch an den Test zu hoch angesetzt.

- Sie haben eine Menge kleiner Erfolgserlebnisse.

- Sie können jederzeit, wenn Sie einen Test erfolgreich durchgeführt haben, Pause oder Feierabend machen in dem Gefühl, dass die Anwendung im aktuellen Zustand wie gewünscht läuft.

- Tests sind eine sehr genaue Formulierung von Anfordungen. Sie können mit ihnen sehr schnell feststellen, ob die tatsächlichen Anforderungen umgesetzt wurden.

- Tests sind Dokumentation: Wenn Sie für alle Methoden, Eigenschaften und Ereignisse einer Klasse Tests schreiben, können Entwickler, die sich anschließend mit Weiterentwicklungen oder Änderungen der Anwendung beschäftigen, diese Tests als Dokumentation heranziehen.

2 Testgetriebene Entwicklung in der Praxis

Nachdem die theoretischen Grundlagen Ihr Interesse geweckt haben, lernen Sie nun den praktischen Ablauf kennen.

Dazu sind einige Vorbemerkungen erforderlich: Die testgetriebene Entwicklung wurde zuerst in Zusammenhang mit objektorientierten Sprachen

eingesetzt. Sie können diese Entwicklungsmethode natürlich auch für die Entwicklung mit prozeduralen Sprachen heranziehen. Es ist aber zu empfehlen, sich direkt mit der objektorientierten Entwicklung im Rahmen der Möglichkeiten von VBA auseinanderzusetzen (s. Beitrag **Objektorientierte Entwicklung mit Access**).

Die meisten Quellen zum Thema testgetriebene Entwicklung enthalten in der Regel keine Hinweise zum Testen von Benutzeroberflächen. In gewisser Weise können Sie die testgetriebene Entwicklung aber dennoch dazu verwenden: Formulare, die ja den größten Teil der Benutzeroberfläche ausmachen, sind eigentlich ebenfalls Objekte mit Methoden, Eigenschaften und Ereignissen. Der einzige Unterschied zu einem aus einer herkömmlichen Klasse erzeugten Objekt ist, dass es eine Benutzeroberfläche hat. Wie Sie Formulare testgetrieben entwicklen, erfahren Sie in einem der Update-Magazine zu diesem Werk.

2.1 Das Werkzeug: accessUnit

Die Werkzeuge zum Durchführen von Unit Tests heißen Testframework. Die Namen der entsprechenden Testframeworks für die unterschiedlichen Programmiersprachen sind immer nach dem gleichen Muster aufgebaut und enden auf Unit. Für Java gibt es unter anderem **JUnit**, für .NET **NUnit** und für Visual Basic **vbUnit**. Das nachfolgend vorgestellte Testframework ist einer der jüngeren Vertreter, aber ähnlich aufgebaut wie die anderen: **accessUnit**.

Hinweis

Sie finden die bei Drucklegung aktuelle Version von **accessUnit** auf der beiliegenden CD. Um neuere Versionen und Updateinformationen zu erhalten, besuchen Sie einfach im Internet die Seite **www.accessunit.de**. ■

accessUnit bietet eine grafische Benutzeroberfläche zur Darstellung des Ablaufs der Tests sowie der im Anschluss vorliegenden Testergebnisse (s. **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Hinweis

Das Testframework **accessUnit** funktioniert in der zum Zeitpunkt der Drucklegung dieses Textes vorliegenden Version mit Access 2000 und höher. ■

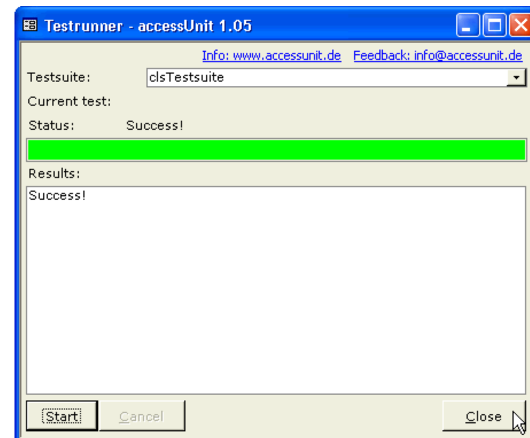

Abb. 1: Das Testframework accessUnit im Einsatz

2.2 Installation von accessUnit

accessUnit liegt in Form eines Formulars, eines Makros und einiger Klassenmodule vor, die in der Datenbankdatei **accessUnit.mdb** zu finden sind.

Sie können das Unit-Testing-Framework nachträglich in eine Datenbank einbinden oder eine neue Datenbank damit entwickeln.

Im ersten Fall importieren Sie einfach alle Objekte der Datenbank **accessUnit.mdb** in die Zieldatenbank. Das Framework steht dann sofort zur Verfügung.

Falls Sie eine neue Datenbank mit Hilfe des Unit-Testing-Frameworks entwickeln möchten, erstellen Sie einfach eine Kopie der Datenbank **accessUnit.mdb** und speichern Sie diese unter dem gewünschten Namen.

2.3 Elemente von accessUnit

Vor dem ersten Beispiel sollen Sie noch kurz die wichtigsten Elemente des **accessUnit**-Frameworks kennenlernen. Die Benutzeroberfläche besteht aus dem Formular **frmTestrunner**, das eine Schaltfläche zum Starten der Tests und Steuerelemente zur Ausgabe der Testergebnisse liefert.

Charakteristisch für Unit-Testing-Frameworks mit Benutzeroberfläche ist dabei je nach Testergebnis die Anzeige eines roten oder grünen Balkens. Der rote Balken bringt in der Regel eine oder mehrere Meldungen mit sich, die auf den oder die gescheiterten Tests hinweisen.

Das in der Datenbank enthaltene Autoexec-Makro enthält eine Anweisung, die der VBA-Entwicklungsumgebung ein Menü mit einer Schaltfläche zum Aufrufen des **Testrunner**-Formular hinzufügt. So können Sie den Testrunner komfortabel von dort aus aufrufen.

```
Option Compare Database
Option Explicit
Public Sub Suite(objTestsuite As Object)
    objTestsuite.AddTest New clsSampleTest
End Sub
```

Quellcode 1

```
Public Function TestsuiteWrapper(strTestsuitename _
    As String) As Object
    Select Case strTestsuitename
        Case "clsTestsuite"
            Set TestsuiteWrapper = New clsTestsuite
    End Select
End Function
```

Quellcode 2

```
Option Compare Database
Option Explicit

Public Sub Setup()
End Sub

Public Sub Teardown()
End Sub

Public Property Get Fixturename() As String
    Fixturename = "clsSampleTest"
End Property

Public Sub Test1(objTestcase As aUTestcase)
    On Error GoTo RunTest_Err
    objTestcase.Assert "Sample assertion 1a", True
    objTestcase.Assert "Sample assertion 1b", True
    Exit Sub
RunTest_Err:
    objTestcase.Assert "#Error in " & Me.Fixturename,
False
    Resume Next
End Sub
```

Quellcode 3

Neben dem Formular und dem Makro benötigt das Framework einige Module mit der Funktionalität: das Standardmodul **aUMenu** und die Klassenmodule **aUMenuEvents**, **aUModule**, **aUTestcase**, **aUTestsuite** und **aUTestsuites**. Die Modulnamen enthalten das Präfix **aU**, um die **accessUnit**-Module leicht von den anderen Modulen unterscheiden zu können.

Die übrigen Klassen der Datenbank **accessUnit.mdb** beinhalten die Tests. Sie benötigen auf jeden Fall eine Testsuite.

Sie enthält die Aufrufe der einzelnen Testcases, die in eigenen Klassen untergebracht sind. Eine solche Testsuite-Klasse sieht etwa wie in Quellcode 1 aus.

Eine Testsuite enthält nur eine Methode namens **Suite**. Diese Methode fügt der Testsuite mit der **AddTest**-Methode eine oder mehrere Testklassen hinzu.

Die Testsuite dieses Beispiels sorgt für die Ausführung der in dem Klassenmodul **clsSampleTest** enthaltenen Tests.

Damit Sie eine Testsuite über den Testrunner aufrufen können, müssen Sie einen Eintrag wie in dem Code aus Quellcode 2 in der Klasse **aUTestsuites** anlegen.

Fehlt noch der eigentliche Test. Jeder Test wird in einer Testklasse untergebracht. Eine einfache Testklasse sieht wie in Quellcode 3 aus.

Einen Test bringt man in je einer Methode unter, deren Methodenname mit „Test" beginnen muss. Ein Test besteht aus einer oder mehreren Assertions (deutsch: Absicherung), die Werte von (Funktions-)Methoden oder Eigenschaften der zu testenden Klasse überprüfen. Eine Assertion hat zwei Parameter: eine aussagekräftige Bezeichnung dessen, was getestet wird, sowie einen bool'schen Ausdruck als Ergebnis der Assertion.

Einer oder mehrere Tests, die sich in der gleichen Testklasse befinden und denselben Aspekt einer Klasse testen – etwa eine Methode oder Eigenschaft - nennt man Testcase.

Die verschiedenen Tests einer Testklasse erfordern häufig die gleiche Startkonfiguration, und wenn es sich nur um das Instanzieren der zu testenden Klasse handelt. Oft kommen noch weitere Vorbereitungen wie beispielsweise das Anlegen von Testdaten hinzu. Damit man die entsprechenden Anweisungen nicht in jede einzelne Test-Methode einbauen muss, verwenden alle Tests einer Testklasse eine gemeinsame Methode, die alle notwendigen Vorbereitungen enthält. Diese Methode heißt **Setup**.

Die vor und während des Testens angefallenen Testdaten sollen auch wieder entfernt werden, was in einer Methode passiert, die automatisch nach jedem Aufruf einer Testmethode aufgerufen wird: die Methode **Teardown**. Diese beiden Methoden nennt man zusammenfassend „Fixture".

3 Beispielanwendung

Als Beispielanwendung erstellen wir nachfolgend eine Klasse, die zur Kontrolle und gegebenenfalls zur Korrektur der Groß- und Kleinschreibung von Namen dient. Sie soll sicherstellen, dass ein eingegebener Benutzername einen führenden Großbuchstaben und nachfolgende kleingeschriebene Buchstaben enthält. Die Methode soll also beispielsweise die Eingabe **andRe** in **Andre** umwandeln.

Desweiteren soll die Methode **Capitalize** heißen und in einer Klasse namens **clsStringfunctions** enthalten sein. Die Klasse erhält diesen Namen, weil später vielleicht noch weitere Zeichenkettenfunktionen hinzugefügt werden sollen.

Hinweis

Natürlich haben Sie Recht, wenn Sie sagen, dass die Funktion **strConv** mit dem Parameter **vbProperCase** das erledigen kann. Aber lesen Sie doch weiter: Wie üblich, gibt es aber auch hier Spezialfälle... ■

Ohne Test kein neuer Code – diese Regel haben Sie bereits weiter oben kennengelernt. Das gilt im Übrigen für jeden Schritt, den Sie mit dem Testframework gehen.

Sie schreiben einen Test, bevor Sie Quellcode schreiben, und genauso fügen Sie dem Testframework erstmal eine Testklasse und den entsprechenden Aufruf hinzu.

Hinweis

Die nachfolgenden beiden Schritte sind für jeden neuen Test erforderlich; beachten Sie diese daher besonders aufmerksam. ■

3.1 Neue Testsuite erstellen

Für das Erstellen der benötigten Testsuite kopieren Sie einfach die enthaltene Beispieltestsuite clsTestsuite in eine neue Klasse namens **clsTestsuite_StringFunctions**. Öffnen Sie das Klassenmodul und passen Sie es wie in Quellcode 4 an.

```
Public Sub Suite(objTestsuite As Object)
    objTestsuite.AddTest _
        New clsStringfunctionsTest
End Sub
```

Quellcode 4

Testsuite im Testrunner verfügbar machen

Damit die Testsuite im Testrunner-Formular ausgewählt werden kann, fügen Sie den folgenden Eintrag zu der **Select Case**-Anweisung der Methode **TestsuiteWrapper** der Klasse **aUTestSuites** hinzu (s. Quellcode 5).

```
Public Function TestsuiteWrapper(strTestsuitename As _
    String) As Object
    Select Case strTestsuitename
        '... weitere Testsuites
        Case "clsStringfunctionsTest"
            Set TestsuiteWrapper = _
                New clsStringfunctionsTest
    End Select
End Function
```

Quellcode 5

```
Option Compare Database
Option Explicit

Public Sub Setup()
End Sub

Public Sub Teardown()
End Sub

Public Property Get Fixturename() As String
    Fixturename = "clsStringfunctionsTest"
End Property

Public Sub Test1(objTestcase As aUTestcase)
    On Error GoTo RunTest_Err
    'Add assertions
    Exit Sub
RunTest_Err:
    objTestcase.Assert "#Error in " & Me.Fixturename, _
        False
    Resume Next
End Sub
```

Quellcode 6

Wenn Sie nun den Testrunner öffnen möchten, erhalten Sie vermutlich eine Meldung, die Sie zum Kompilieren der Anwendung auffordert.

Das passiert zu Recht, denn der nachfolgende Kompiliervorgang, den Sie beispielsweise über den Menüeintrag **Debuggen → Kompilieren von ...** starten, wird auf die fehlende Klasse **clsStringfunctionsTest** hinweisen.

Testklasse anlegen

Das Gerüst der Testklasse erstellen Sie genauso wie im Fall der Testsuite: Kopieren Sie die Klasse **clsNoTest** in eine neue Klasse und nennen Sie diese **clsStringfunctionsTest**. Diese sollte nun den Inhalt aus Quellcode 6 haben.

Passen Sie die Eigenschaft **Fixturename** wie folgt an, indem Sie **clsNoTest** durch **clsStringfunctionsTest** ersetzen. Kompilieren Sie erneut und öffnen Sie den Testrunner. Das Kombinationsfeld **Testsuite** (s. Abb. 2) sollte nun unter anderem den Eintrag **clsTestsuite_StringfunctionsTest** anzeigen.

3.2 Testen mit dem Testrunner

Da noch keine Assertions vorhanden sind, sollte der Test erfolgreich verlaufen, da keine negativen Ergebnisse zu erwarten sind. Ein Klick auf die Schaltfläche **Start** zeigt, dass die Annahme richtig war – es erscheint der grüne Balken.

Nach der Auswahl der Testsuite **clsTestsuite_Stringfunctions** zeigt der Testrunner an, dass die Suite einen Test enthält. Dabei handelt es sich um den (noch) leeren **Test1** (s. Quellcode 6).

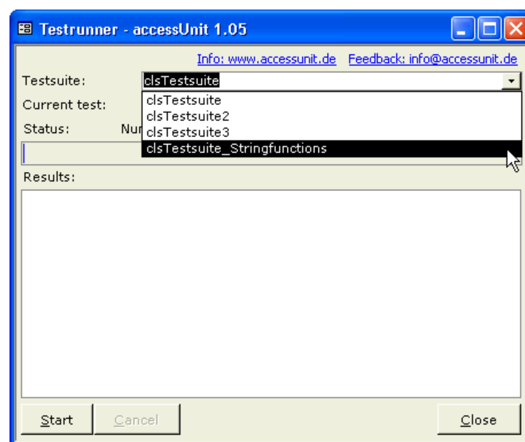

Abb. 2: Auswahl einer Testsuite

Hinweis

Schließen Sie den Testrunner jedesmal, wenn sie Änderungen am Code vornehmen und öffnen Sie ihn für weitere Tests erneut. Auf diese Weise macht der Testrunner Sie darauf aufmerksam, wenn Sie die Anwendung nach Änderung des Codes nicht kompiliert haben. Wenn Sie den Testrunner geöffnet lassen, können anderenfalls Laufzeitfehler auftreten. ■

Nun wollen wir zum ersten Test schreiben: Dazu legen wir zunächst im Kopf der Testklasse eine Objektvariable für die zu testende Klasse an:

```
Private mClass As clsStringfunctions
```

Zum Instanzieren der Klasse verwenden wir die Methode **Setup**:

```
Public Sub Setup()
    Set mClass = New clsStringfunctions
End Sub
```

Der Ordnung halber wird nach dem Test aufgeräumt:

```
Public Sub Teardown()
    Set mClass = Nothing
End Sub
```

Und jetzt folgt der große Moment. Wir legen den ersten Test an, der voraussetzt, dass ein Name, der aus beliebigen Groß- und Kleinbuchstaben besteht, in einen Namen mit führendem großem Buchstaben umgewandelt wird. Dazu verwenden wir verschiedene Assertions (s. Quellcode 7). Diese sechs Assertions überprüfen verschiedene Varianten für die Eingabe, darunter die Übergabe eines Leerstrings und einer Sonderzeichenkombination als „Randfälle".

Hinweis

Sie können jetzt natürlich fragen, wozu wir soviele Assertions benötigen, da die **strConv**-Funktion doch zuverlässig arbeitet. Damit begeben Sie sich in Gefahr: Wer sagt denn, dass keine Anforderungen hinzukommen, die viel besser mit anderen Mitteln, etwa mit regulären Ausdrücken erfüllt werden könnten? Und das ist ein wichtiger Punkt: Sie sollten sich beim Formulieren der Anforderungen in Form von Tests am besten von sämtlichen Implementierungsdetails befreien, die Sie schon im Kopf haben. Ein Test zuviel kostet ein kleines bisschen Zeit, aber einer zuwenig kann Ihnen Tage zusätzlicher Arbeit einbringen. ■

Bevor Sie den Testrunner erneut öffnen, kompilieren Sie die Anwendung und stellen fest, dass

```
Public Sub Test_SingleWord(objTestcase As aUTestcase)
    On Error GoTo RunTest_Err
    objTestcase.Assert "Andre -> Andre", StrComp(mClass.Capitalize("Andre"), _
        "Andre", vbBinaryCompare) = 0
    objTestcase.Assert "andre -> Andre", StrComp(mClass.Capitalize("andre"), _
        "Andre", vbBinaryCompare) = 0
    objTestcase.Assert "ANDRE -> Andre", StrComp(mClass.Capitalize("ANDRE"), _
        "Andre", vbBinaryCompare) = 0
    objTestcase.Assert "AnDrE -> Andre", StrComp(mClass.Capitalize("AnDrE"), _
         "Andre", vbBinaryCompare) = 0
    objTestcase.Assert "->", StrComp(mClass.Capitalize(""), "", vbBinaryCompare) = 0
    objTestcase.Assert "Sonderzeichen", StrComp(mClass.Capitalize("(/&$$=('§/§"), _
         "(/&$$=('§/§", vbBinaryCompare) = 0
    Exit Sub
RunTest_Err:
    objTestcase.Assert "#Error in " & Me.Fixturename, False
    Resume Next
End Sub
```

Quellcode 7

die zu testende Klasse **clsStringfunctions** noch nicht vorhanden ist.

Legen Sie ein leeres Klassenmodul namens **clsStringfunctions** an und kompilieren Sie erneut.

Der nächste Kompilierversuch bringt erwartungsgemäß zu Tage, dass die Methode **Capitalize** nicht vorhanden ist.

Legen Sie diese als leere Methode an:

```
Public Function Capitalize(strName _
    As String) As String

End Function
```

Die Kompilierung funktioniert nun. Öffnen Sie den Testrunner, wählen Sie die richtige Suite aus und starten Sie den Test. Wie Abb. 3 zeigt, scheitert der Test, weil fünf der sechs Assertions nicht erfüllt wurden.

Nur die leere Zeichenfolge wird als leere Zeichenfolge zurückgegeben und erfüllt die entsprechende Assertion.

In dieser Abbildung ist schön zu sehen, wie **accessUnit** von der Testsuite über den Test bis hin zum Assertion zeigt, wo es hapert. Passen wir also nun den Test so an, dass die Anforderungen erfüllt werden:

```
Public Function Capitalize(strName _
    As String) As String
    Capitalize = StrConv(strName, _
        vbProperCase)
End Function
```

Es folgt die übliche Prozedur: Kompilieren, Testrunner starten, Test durchführen. Erwartungsgemäß wird der Test nun erfüllt.

3.3 Anforderungen erweitern

Die Funktion soll aber nicht nur einzelne Namen anpassen, sondern auch Kombinationen aus Vor- und Nachname. Diese Anforderung formulieren wir in einem neuen Test, der folgendermaßen aussieht und wie erwartet erfolgreich verläuft (s. Quellcode 8).

Hinweis

Mit diesem „Ergänzungstest" haben Sie in dem Wissen, das der Test funktionieren würde, eine zusätzliche Sicherheit eingebaut. Wenn einmal ein Test funktioniert, von dem Sie es nicht erwarten, sollten Sie kontrollieren, ob nicht der Test selbst eventuell fehlerhaft ist - möglicherweise wird er ja gar nicht aufgerufen. Wenn Sie unsicher sind, bauen Sie einfach eine Assertion ein, die fehlschlägt – damit finden Sie auf jeden Fall heraus, ob der Test überhaupt aufgerufen wird. Das ginge etwa mit folgendem Ausdruck: **objTestcase.Assert „Test the test", False.** ■

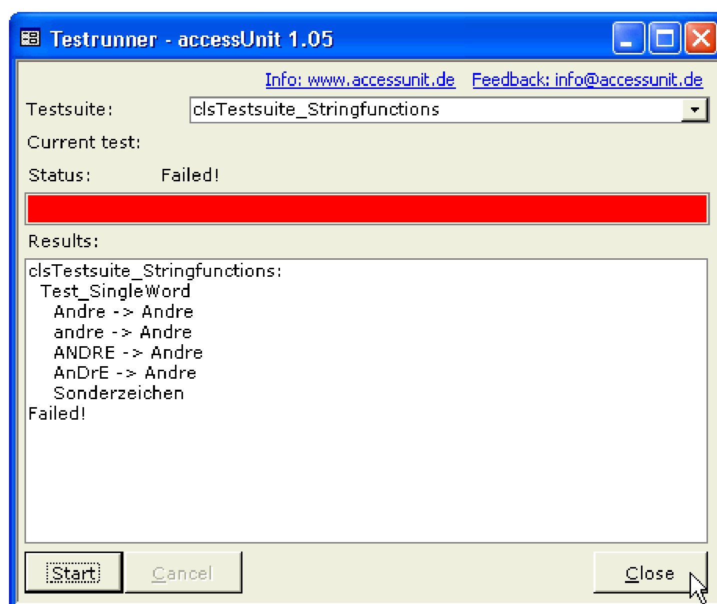

Abb. 3: Ein Test mit nicht erfüllten Anforderungen

Doppelnamen

Die Methode gibt nun sowohl einzelne Namen als auch Kombinationen aus Vor- und Nachname in der gewünschten Form zurück. Nachdem wir diese Klasse nun beispielsweise in einer Web-Anwendung zum

```
Public Sub Test_FirstAndLastname(objTestcase As aUTestcase)
    On Error GoTo RunTest_Err
    objTestcase.Assert "Andre Minhorst -> Andre Minhorst", _
        mClass.Capitalize("Andre Minhorst") = "Andre Minhorst"
    objTestcase.Assert "andre minhorst -> Andre Minhorst", _
        mClass.Capitalize("andre minhorst") = "Andre Minhorst"
    objTestcase.Assert "ANDRE MINHORST -> Andre Minhorst", _
        mClass.Capitalize("ANDRE MINHORST") = "Andre Minhorst"
    objTestcase.Assert "AnDrE MiNhOrSt -> Andre Minhorst", _
        mClass.Capitalize("AnDrE MiNhOrSt") = "Andre Minhorst"
    Exit Sub
RunTest_Err:
    objTestcase.Assert "#Error in " & Me.Fixturename, False
    Resume Next
End Sub
```

Quellcode 8

```
Public Sub Test_MinusBetweenNames(objTestcase As aUTestcase)
    On Error GoTo RunTest_Err
    objTestcase.Assert "Müller-Lüdenscheid -> Müller-Lüdenscheid", _
        StrComp(mClass.Capitalize("Müller-Lüdenscheid"), "Müller-Lüdenscheid", _
        vbBinaryCompare) = 0
    objTestcase.Assert "müller-lüdenscheid -> Müller-Lüdenscheid", _
        StrComp(mClass.Capitalize("müller-lüdenscheid"), "Müller-Lüdenscheid", _
        vbBinaryCompare) = 0
    Exit Sub
RunTest_Err:
    objTestcase.Assert "#Error in " & Me.Fixturename, False
    Resume Next
End Sub
```

Quellcode 9

„Geradebiegen" von Namen bei der Anmeldung etwa für einen Newsletter verwenden, stellt sich früher oder später heraus, dass die Anforderungen nicht alle Fälle abgedeckt haben.

Meldet sich dort jemand mit einem Doppelnamen an, der durch Bindestrich getrennt ist, wird der zweite Teil des Namens klein weitergeschrieben.

Schreiben wir also einen Test für diesen Fall. Dieser Test enthält nur zwei Assertions: Eine, die überprüft, dass ein großer Buchstabe hinter einem Bindestrich auch groß bleibt, und eine, die überprüft, ob ein kleiner Buchstabe hinter dem Bindestrich in einen großen umgewandelt wird (s. Quellcode 9).

Beide Tests schlagen wie fehl, weil die Buchstaben hinter dem Minus-Zeichen durch die **strConv**-Funktion verkleinert werden.

Passen wir also die **Capitalize**-Funktion an. Dazu fügen wir eine **Do While**-Schleife ein, die solange durchlaufen wird, bis keine Minus-Zeichen mehr enthalten sind (s. Quellcode 10). Normalerweise ist das zwar maximal eines, aber mit dieser Vorgehensweise gehen wir auf Nummer Sicher.

Der Test verläuft erfolgreich: Die Methode erfüllt nun alle vorläufig bekannten Anforderungen. Natürlich gibt es noch einige Varianten, die hier nicht berücksichtig sind – beispielsweise Namenszusätze wie „de", „van" oder „von". Diese

```
Public Function Capitalize(strName As String) As String
    Dim posMinus As Integer
    Capitalize = StrConv(strName, vbProperCase)
    posMinus = InStr(1, Capitalize, "-")
    Do While posMinus > 0
        If Mid(Capitalize, posMinus + 1, 1) Like "[a-z]" Then
            Capitalize = Mid(Capitalize, 1, posMinus) & StrConv(Mid(Capitalize, _
                posMinus + 1, 1), vbUpperCase) & Mid(Capitalize, posMinus + 2)
        End If
        posMinus = InStr(posMinus + 1, Capitalize, "-")
    Loop
End Function
```

Quellcode 10

```
Public Function Capitalize(strName As String) As String
    Dim posMinus As Integer
    Dim strCapitalize As String
    strCapitalize = StrConv(strName, vbProperCase)
    posMinus = InStr(1, strCapitalize, "-")
    Do While posMinus > 0
        If Mid(strCapitalize, posMinus + 1, 1) _
            Like "[a-z]" Then
            strCapitalize = Mid(strCapitalize, 1, _
                posMinus) & StrConv(Mid(strCapitalize, _
                posMinus + 1, 1), vbUpperCase) & _
                Mid(strCapitalize, posMinus + 2)
        End If
        posMinus = InStr(posMinus + 1, strCapitalize, "-")
    Loop
    Capitalize = strCapitalize
End Function
```

Quellcode 11

würden von der Methode im aktuellen Zustand wie alle anderen Bestandteile groß geschrieben. Für Beispielzwecke reicht uns allerdings die jetzige Form der Methode.

3.4 Ein wenig Refactoring

Allerdings möchten wir noch einen kleinen Ausflug in die Welt des Refactoring unternehmen. Die Methode hat den kleinen Makel, dass die Funktionsbezeichnung nicht nur für die Zuweisung des Rückgabewertes, sondern als Hilfsvariable verwendet wird.
Da gerade alle Tests laufen, können wir ohne Sorge eine neue Hilfsvariable einbauen – der anschließende Test wird uns mitteilen, ob die Funktion der Methode durch den Umbau beeinträchtigt wurde.

Abb. 4 veranschaulicht die einzelnen Positionen, die man bei der testgetriebenen Entwicklung durchlaufen kann.

Derzeit haben wir die Wahl zwischen einem der beiden „Test bestanden"-Pfeile – und wählen den Weg zum Refactoring.

Wir bauen die Methode also wie in Quellcode 11 um und überprüfen direkt im Anschluss, ob die Tests noch durchlaufen. Bei positivem Ergebnis können wir beruhigt an die nächste Methode oder Klasse herangehen.

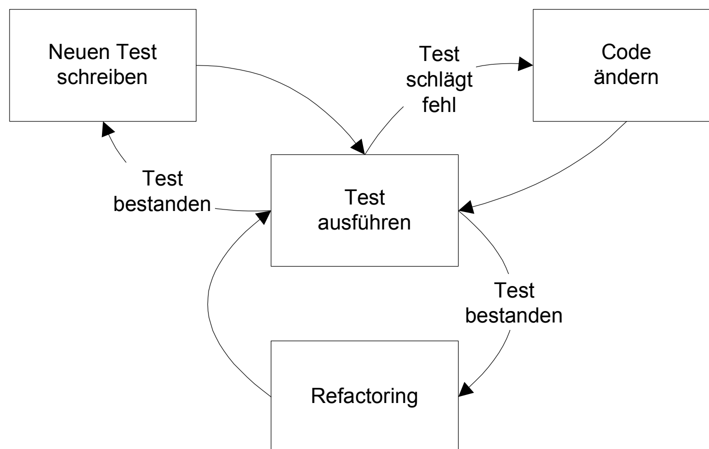

Abb. 4: Ablauf der testgetriebenen Entwicklung

3.5 Randfälle testen

Mit den hier beschreibenen Tests haben Sie relativ einfache Fälle abgedeckt. Dabei bestand kein großartiger Anlass, sich Gedanken um sogenannte Randfälle zu machen. Bei Randfällen handelt es sich um solche Anforderungen, die nicht triviale Fälle abdecken und deren Erfüllung spezielle Vorgehensweisen erfordern, weil durch das Eintreten des Randfalls beispielsweise ein Fehler ausgelöst würde. Im vorliegenden Test gibt es keinen solch schwerwiegenden Randfall, da die Methode ohnehin nur Werte mit dem Datentyp String akzeptiert. Ein Beispiel wäre aber etwa eine Funktion, die den Quotienten zweier Zahlen ermittelt und man mit solchen Werten füttert, dass eine Division durch 0 auftritt. Diesen Fall würde man im Test berücksichtigen, damit er bei der Ausführung entsprechend behandelt wird.

3.6 Wie geht es weiter?

Auf die gleiche Art wie im Beispiel können Sie komplette Anwendungen erstellen, indem Sie Klasse für Klasse testgetrieben entwickeln. Dabei werden Sie oft Testklassen erstellen, die sich jeweils auf nur eine zu testende Klasse beziehen; Sie werden aber auch Testklassen verwenden, die zum Testen der Interaktion zwischen Klassen dienen.

3.7 Wohin mit den Tests?

Sie haben mittlerweile gemerkt, dass Sie bei der testgetriebenen Entwicklung bestimmt noch mal soviel Code produzieren wie für die eigentliche Anwendung erforderlich ist.

Wenn die Anwendung fertig ist, möchten Sie diesen Code möglicherweise loswerden - vielleicht aber auch nicht, denn immerhin haben Sie eine Menge Arbeit hineingesteckt. Außerdem sollten Sie nicht vergessen: Die Tests sind eine Dokumentation der damit erstellten Klassen. Sie sollten die Tests auf jeden Fall in Ihrer eigenen Kopie der Anwendung behalten, und es gibt eigentlich auch keinen Grund, die Tests mit auszuliefern.

Dass Sie zusätzlich zu den Tests auch noch das Framework in der Anwendung belassen, kann ebenfalls Vorteile haben: Gegebenenfalls gibt es einmal neue Versionen des Frameworks, die mit mit den aktuell in der getesteten Datenbank vorhandenen Objekten nicht mehr zusammenarbeiten. Das enthaltene Framework wird seinen Dienst aber klaglos tun, solange Sie es nicht ersetzen.

4 Zusammenfassung und Ausblick

Die testgetriebene Entwicklung mag Ihnen nach der Lektüre dieses Beitrags sinnvoll erscheinen oder auch nicht – überzeugen können werden Sie sich von Ihr erst, wenn Sie es selbst einmal ausprobiert haben. Um sich den Einstieg zu erleichtern, nehmen Sie sich einfach eine möglichst kleine bestehende Anwendung und erstellen diese neu. Damit haben Sie schon eine Vorstellung davon, in welche Richtung die Anwendung gehen soll und können sich voll auf die Art der Entwicklungsmethode konzentrieren.

Die Internetseite **www.accessunit.de** beschäftigt sich mit dem Testframework selbst und mit den damit in Zusammenhang stehenden Techniken wie der testgetriebenen Entwicklung oder Refactoring; hier finden Sie regelmäßig neue Versionen und neue Informationen zum Unit-Testing mit Access sowie interessante Links.