Peter Meinl

Dem Fehler auf der Spur

Saubere Fehlerbehandlung mit VB in allen Situationen

Mindestens 30% einer professionellen Anwendung sind Code zum Reagieren auf Fehler und Ausnahmesituationen. Es gibt unterschiedliche Strategien, die Fehlerbehandlung zu gestalten, und beim Implementieren kann man einiges falsch machen: Soll man Inline-Code oder Error-Handler verwenden? Wie sieht ein Standard-Error-Handler aus? Welche Stolpersteine gibt es? Wozu sind Zeilennummern gut? Wie protokolliert und lokalisiert man Fehler?

Ob Anwender ein System akzeptieren, hängt wesentlich von der Qualität der Fehlerbehandlung ab. Wir setzen voraus, daß eine Anwendung funktioniert und ausreichende Performance hat. Aber gute Software unterscheidet sich von schlechter Software wesentlich dadurch, wie souverän sie mit Fehlersituationen umgeht. Erwartete Fehler sind zu behandeln, unerwartete Fehler müssen dokumentiert und toleriert werden. Deshalb sollten Sie eine klare Strategie zur Fehlerbehandlung haben und diese konsequent umsetzen.

Aufgaben der Fehlerbehandlung

Fehlerarten

Bei der Fehlerbehandlung unterscheidet man zwischen erwarteten und unerwarteten Fehlern:

- *Erwartete Fehler* sind Anwendungsfehler (wie „Kreditlimit überzogen“) oder Fehler, die man beim Entwickeln schon bedacht hat (wie „Laufwerk schreibgeschützt“). Auf erwartete Fehler reagiert man mit spezifischem Code, z.B. indem man eine Meldung anzeigt oder eine bereits vorhandene Datei löscht.
- *Unerwartete Fehler* sind alle anderen Fehler. Es ist nicht möglich, völlig fehlerfreien Code zu schreiben, und weder sinnvoll noch praktikabel, alle möglichen Eingangsdaten und Bedienungsmöglichkeiten einer Anwendung bei der Entwicklung zu bedenken. Deshalb reagiert man auf unerwartete Fehler mit einer standardisierten Fehlerbehandlung.

Anforderungen

Die Fehlerbehandlung einer professionellen Anwendung muß folgenden Anforderungen genügen:

- Die Anwendung darf nicht abstürzen. Auch unerwartete oder schwere Fehler sind zu behandeln, oder die Anwendung ist kontrolliert zu beenden. Ein kontrolliertes Beenden der Anwendung dient vor allem dazu, die Applikationsdaten in einem konsistenten Zustand zu hinterlassen.
- Die Anwendung muß Fehler tolerieren. Auch nach einem Fehler muß die Anwendung in einem konsistenten Zustand bleiben, d.h. Datenänderungen dürfen nicht nur teilweise ausgeführt worden sein. In Datenbankanwendungen sind daher Transaktionen ein Grundwerkzeug bei der Fehlerbehandlung.
- Unerwartete Fehler müssen dokumentiert werden. Damit unerwartete Fehler behoben werden können, sind Auslöser und Ort solcher Fehler zu protokollieren. Als Ort sollte dabei nicht nur der Name einer ausführbaren Datei genannt sein, sondern Code-Modul bzw. Klasse, Prozedurname und möglichst auch eine Code-Zeilennummer.
- Der Anwender muß angemessen informiert werden. Anwender müssen ihren Erwartungen entsprechend über Fehler informiert werden und über die Möglichkeiten, sie zu beheben. Das kann durch eine *MessageBox* oder durch eine EMail- oder SMS-Nachricht geschehen.

Die Voraussetzung für das Programmieren von Code zur Fehlerbehandlung ist ein fundiertes Verständnis davon, wie VB Fehler auslöst, weiterleitet und behandelt. Auch erfahrene Programmierer sollten ihr Wissen von Zeit zu Zeit auffrischen, z.B. indem sie die Online-Hilfe erneut lesen. Wenn nämlich der Code nur ungenügend mit Fehlerbehandlungsroutinen instrumentiert ist, wird das immer mit einem

ÜBERblick

Thema
Die Behandlung von Fehlern und Ausnahmesituationen ist ein wesentlicher Bestandteil jedes Programms.
Dieser Bericht erläutert unterschiedliche Strategien zur Fehlerbehandlung mit VB und gibt Ratschläge zu ihrer Implementierung.
Mit diesen Tips ausgerüstet, sollten Sie Ihre Anwendungen robuster und benutzerfreundlicher machen können.

VB-Versionen: VB4, VB5, VB6

Zielgruppe
Anfänger und Fortgeschrittene gleichermaßen.

Totalabsturz bezahlt: Falls man auf der obersten Ebene der Aufrufhierarchie von VB-Prozeduren keinen eigenen Error-Handler codiert, verwendet VB automatisch den Default-Error-Handler. Er gibt eine Fehlermeldung aus und beendet die Anwendung ohne Möglichkeit zum Eingriff, indem er ein *End* ausführt. Dabei werden die Ereignisprozeduren *Form_Unload* von aktiven Formularen und *Class_Terminate* von aktiven Objekten nicht durchlaufen! Ihre Anwendung hat also keine Chance, Daten in einen sicheren Zustand zu überführen.

Strategien zur Fehlerbehandlung

Es gibt verschiedene Möglichkeiten, Fehler auszulösen und zu behandeln. Man sollte sich bewußt für eine Strategie entscheiden und entsprechende Standards zur Programmierung festlegen.

Jede Komponente sollte alle Fehler selbst behandeln, die in ihr entstehen oder die von einer Komponente ausgelöst werden, die sie verwendet. Meist ist es nicht sinnvoll, solche Fehler einfach zum Aufrufer weiterzuleiten. Das gilt auch für unerwartete Fehler. Am besten dokumentiert man diese im *Log* und *Trace* (s.u.) und gibt einen allgemeinen Fehler an den Aufrufer zurück.

Möglichkeiten der Fehlermeldung

Zum Melden von Fehlern an aufrufende Prozeduren gibt es folgende Möglichkeiten:

- **Rückgabewert** – Fehler werden im Rückgabewert von Funktionen codiert. Im einfachsten Fall geben Sie einer Funktion den Typ *Boolean* und liefern im Fehlerfall *False* zurück.
- **Fehlerfunktion** – Eine spezielle Funktion gibt Auskunft über den zuletzt aufgetretenen Fehler (z.B. *GetLastError*). Diese Technik kann die Meldung eines Fehlers per Rückgabewert unterstützen. Im Fehlerfall stehen dann umfangreiche Informationen zur Verfügung – ohne das Interface, d.h. die Funktionsdefinition, zu belasten.
- **Err.Raise** – Der Programmablauf wird unterbrochen, und der Aufrufer erhält Informationen zum Fehler im Standard-VB-Objekt *Err*. Ein mit *Err.Raise* erzeugter Fehler unterscheidet sich formal nicht von einem intern ausgelösten (z.B. Division durch 0). Er kann vom Aufrufer mit einem Error-Handler behandelt werden.

Reaktionsmöglichkeiten

Abhängig davon, wie ein Fehler gemeldet wurde, gibt es verschiedene Möglichkeiten, auf ihn zu reagieren:

- **Inline** – Mit *If RevtVa*l = oder *If Err Then* nach jeder wesentlichen Aktion prüfen, ob ein Fehler aufgetreten ist:

```
On Error Resume Next
Set xlApp = GetObject(, "Excel.Application")
If Err = ERR_EXCEL_NOTRUNNING Then
   Set xlApp = CreateObject("Excel.Application")
Else
...
```

- **Error-Handler** – Mit *On Error Goto Error-Handler* eine Routine zur Fehlerbehandlung definieren und Fehler dort behandeln. Der Error-Handler kann Fehler der Routine behandeln, in der er definiert ist, und/oder auf unbehandelte Fehler in tieferen Aufrufschichten reagieren. Der nachfolgende Code zeigt die Arten der Behandlung: den Fehler an eine höhere Aufrufschicht weitergeben *(Err.Raise),* den Fehler tolerieren/beheben *(Resume)* und den Fehler an eine allgemeine Fehlerbehandlungsroutine weiterleiten *(itcUnexpectedErrorHandle):*

```
Option Explicit
Private Const mModuleName = ":CbuTest."
Private Const ComponentErrorBase = 1000

Public Enum genuDemo
  edeMinGTMax = CompanyErrorBase + _
    ComponentErrorBase + vbObjectError
  edeFileNotExists
  '...
End Enum

Private Function Demo()
  Const ProcedureName = "Demo"
  Dim fso As New Scripting.FileSystemObject

  On Error GoTo ErrorHandler
  ...
  If Min > Max Then Err.Raise edeMinGTMax
  ...
  fso.MoveFile PathFrom, PathTo
  ...
  Exit Function

ErrorHandler:
  Select Case Err.Number
    Case edeMinGTMax
      Err.Raise Err.Number, mModuleName & _
        ProcedureName, "Min ist größer Max!"
    Case evbFileExists
      'Ignore
    Case evbPathNotFound
      If fso.FolderExists(PathFrom) Then
        fso.CreateFolder PathTo
        Resume
      Else
        ...
      End If
    Case Else
      itcUnexpectedErrorHandle Err, Erl, _
        mModuleName & ProcedureName
      Err.Raise ecoComponentError, mModuleName _
        & ProcedureName, "Handled component error"
  End Select
End Function
```

Jede Bewertung von Strategien zur Fehlerbehandlung ist sicher subjektiv und von der Funktionalität abhängig, die programmiert werden soll. Der Autor favorisiert es, Fehler mit *Err.Raise* auszulösen und mit Error-Handlern zu behandeln. Auch VB-Statements, Prozeduren und Controls sowie die meisten Custom Controls verwenden diese Strategie. Man sollte das jedoch nicht zu dogmatisch betrachten. Es gibt in jedem Projekt Algorithmen, bei denen es besser ist, Fehler inline zu behandeln. Auch Erfahrung und Gewohnheit von Programmierern oder gleichzeitiges Programmieren in unterschiedlichen Programmiersprachen können ein Grund sein, VB-Fehler inline zu behandeln.

Hilfe bei der Qual der Wahl

Hier noch einige Aspekte, die Ihnen bei der Entscheidung für eine eigene Strategie helfen sollen:

- Windows-API-Funktionen geben Fehler als Rückgabewerte zurück. Deshalb muß man hier immer mit inline-Fehlerbehandlung arbeiten, also nach jedem API-Aufruf das *Err*-Objekt und je nach API-Spezifikation auch *App.LastDLLError* prüfen.
- Die Codierung von Fehlern in Rückgabewerten von Funktionen birgt einige Gefahren (s.u. *Probleme mit Rückgabewerten*). Wenn man sie verwendet, sollte man Werte ungleich 0 (= *True*) für Erfolg und 0 (= *False*) für Fehler verwenden, und für die Unterscheidung der Fehler ein Konstrukt analog der Windows-API-Funktion *GetLastError*.
- Mit Fehlercodes als Rückgabewerten von Funktionen nimmt man sich die Möglichkeit, den Rückgabewert fachlich zu verwenden, z.B. *strFarbe = GetFarbe(lngFarbCode)*. Man hat also im Prinzip nur noch *Sub*-Prozeduren zur Verfügung.
- Oft wird argumentiert, die Verwendung von Error-Handlern unterbreche den normalen Programmfluß. Meist ist das Gegenteil der Fall. Error-Handler erlauben es, den Normalfall geradlinig zu codieren und die Behandlung von Ausnahmen an Error-Handler zu delegieren, z.B. *If Min > Max Then Err.Raise edmMinGTMax*.

Ein Beispiel-Error-Handler

Anhand des obigen längeren Listings wollen wir verschiedene Aspekte der Fehlerbehandlung mit VB diskutieren. Das Beispiel zeigt eine Prozedur, die erwartete und unerwartete Fehler in einem Error-Handler behandelt.

Unerwartete Fehler

Unerwartete Fehler *(Case Else)* werden mit der selbstdefinierten Standardmethode *itcUnexpectedErrorHandle* behandelt (siehe Code auf der Heft-CD). Die *Instancing*-Eigenschaft ihrer Klasse *CError* ist *Global Multiuse*. Damit ersparen wir uns, explizit eine Instanz zu erzeugen. Das Präfix „itc“ soll Namenskollisionen mit globalen Methoden anderer Hersteller vermeiden. *itcUnexpectedErrorHandle* protokolliert den unerwarteten Fehler im Log *(Abbildung 1)*, schreibt ihn ins Trace *(Abbildung 2)* und kann eine Meldung an den Anwender ausgeben (gesteuert über einen optionalen Parameter). Falls der Quellcode Zeilennummern enthält, wird mit *Erl* auch die Nummer der Zeile, die den Fehler ausgelöst hat, dokumentiert (s.u. *Zeilennummern verwenden*).

Unerwartete Fehler werden mit *Err.Raise ecoComponentError* zum Aufrufer weitergegeben. Wird *ecoComponentError* im Aufrufer wieder als unerwarteter Fehler behandelt (das ist meistens der Fall), dann schreibt *itcUnexpectedErrorHandle* den Fehler ins Trace, aber nicht mehr ins Log. Damit wird erreicht, daß ein Fehler genau einmal im Log auftaucht, und vermieden, daß die Anwendung das Log monopolisiert. Im Trace ist sichtbar, wo der Fehler entstanden ist und auf welchem Weg er sich über die Aufrufhierarchie nach oben hangelt. Man sieht also selbst bei einem minimalen Trace-Level *trlErr* (s.u. *Log & Trace*) im Fehlerfall trotzdem die gesamte Aufrufhierarchie zum Fehlerort.

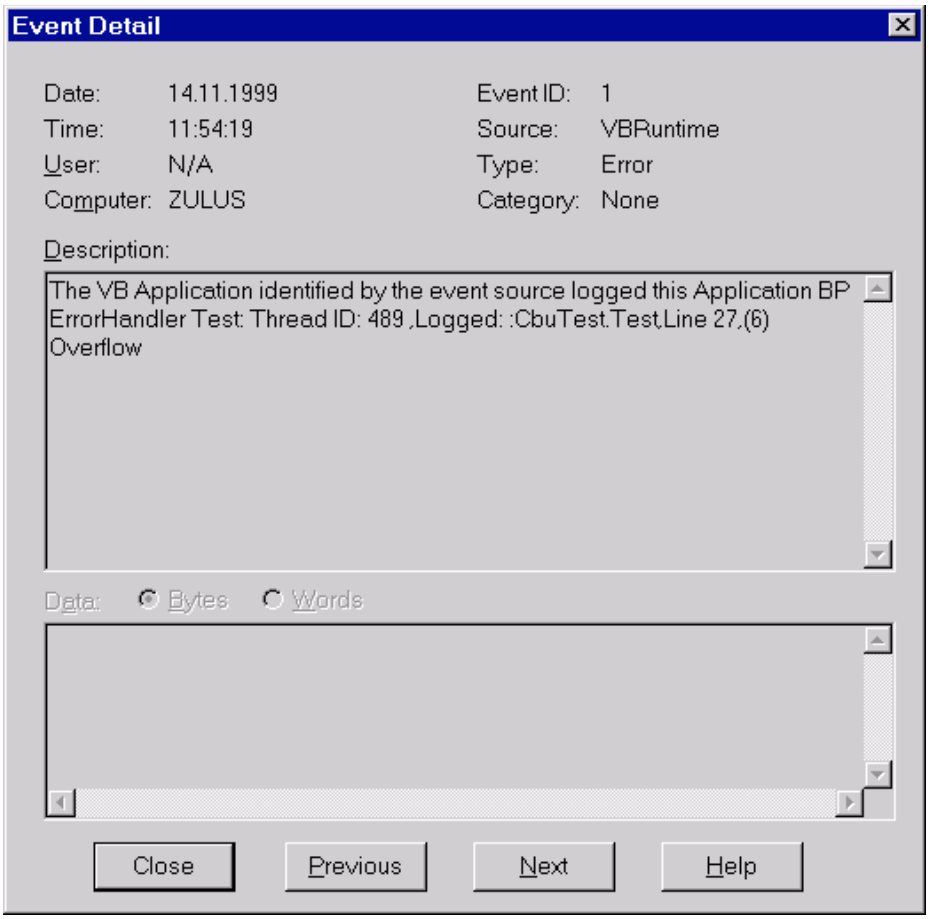

Abbildung 1:
Ein mit App.LogEvent erzeugter Eintrag im Ereignis-Log. Um die Fehlersuche zu erleichtern, ist die Nummer der fehlerhaften Zeile im VB-Quellcode (Zeile 27) angegeben.

Lokal erzeugte Fehler

Lokal erzeugte Fehler, *edeMinGTMax* im Beispiel, müssen im Error-Handler mit *Err.Raise* erneut ausgelöst werden, um sie an den Aufrufer weiterzugeben.

Alle Fehler, die eine Komponente nach außen gibt, werden am besten in einer globalen *Enum* definiert. Dadurch sind die möglichen Fehler im Objekt-Browser sichtbar, und die Fehlerkonstanten können vom Aufrufer in seinem Error-Handler verwendet werden. Hierbei ist zu beachten, daß diese *Enum* nur im direkten Aufrufer bekannt ist. Fehler, die weiter nach oben durchgereicht werden (häufig sind das die Standard-Datenbank-Fehler *edbDuplicateRecord, edbUpdateConflict, edbRecordNotExists*) sollten in einer globalen DLL definiert werden.

Eigene Fehlernummern müssen außerhalb des für VB reservierten Nummernbereichs 0 bis 512 liegen. Damit verschiedene Fehler in unterschiedlichen DLLs nicht dieselbe Nummer haben, sollte man für jede DLL eine eigene Basis-Nummer *ComponentErrorBase* festlegen.

Damit außerdem auch die DLLs verschiedener Hersteller nicht dieselbe Nummer liefern, sollte jeder Hersteller eine eigene Basis-Nummer *CompanyErrorBase* verwenden. Die Basis-Nummer der ISTEC GmbH ist z.B. 12000, die Basis-Nummer des Codes zum Buch „Hardcore Visual Basic“ ist 13000. Bitte verwenden Sie diese Nummern nicht, und machen Sie Ihre eigenen leicht änderbar! Alle Fehler, die über COM generiert werden, sollten mit *vbObjectError* maskiert werden. Faßt man all diese Anforderungen zusammen, dann hat die erste Fehlerkonstante einer Fehler-*Enum* den Wert *CompanyErrorBase* + *ComponentErrorBase* + *vbObjectError*.

Fehler ignorieren/beheben

evbFileExists zeigt, wie man Fehler ignoriert, indem man sie fängt, aber nicht erneut auslöst.

evbFolderNotExists zeigt, wie man einen Fehler fängt, behebt und mit *Resume* an der auslösenden Stelle im Code weitermacht.

Tips zur Fehlerbehandlung

Ein Error-Handler in jede Prozedur

Jede Prozedur sollte einen Error-Handler haben. Bei Prozeduren, in denen nur unerwartete Fehler auftreten können, ist das zwar nur auf oberster Ebene zwingend notwendig, damit VB die Anwendung nicht beendet. Beim Debuggen ist es aber sehr hilfreich, immer einen Error-Handler zu haben, da man sonst bei einem Fehler im Error-Handler des Aufrufers landet und die lokalen Variablen, die zum Fehler geführt haben, nicht mehr auswerten kann. Nur bei sehr einfachen, stabilen Prozeduren auf unterster Ebene sollte man sich erlauben, ganz auf Error-Handler zu verzichten.

Zeilennummern verwenden

Bei der Fehlersuche ist es sehr hilfreich, wenn Fehlermeldungen die Nummer der Zeile enthalten, die den Fehler ausgelöst hat. Während der Entwicklung sind Zeilennummern aber meist nur störend und fast unmöglich zu verwenden, da sie nicht automatisch gepflegt werden. Man kann jedoch vor der Auslieferung mit einem kleinen VB-Add-In oder einem Werkzeug (z.B. Builder von GridLinx Software auf der Heft-CD) die Codezeilen numerieren und zusammen mit dem übersetzten Code ablegen. Im Error-Handler gibt man dann die Fehlerzeile mit *Erl* aus. Diese kleine Maßnahme kann stundenlanges Suchen ersparen.

Texte konfigurieren

Bei vielen Anwendungen müssen die Fehlertexte konfigurierbar sein, um sie leicht ändern zu können oder die Anwendung mehrsprachig zu machen. Dazu kann man Ressourcendateien, *Resource-Only*-Komponenten oder Werkzeuge verschiedener Hersteller verwenden (vgl. [1]).

Eine neue Möglichkeit eröffnen ADOs persistente Recordsets. Mit den *Recordset*-Methoden *Save* und *Load* kann man Recordsets ohne Verbindung zu einer Datenbank in Dateien schreiben und von Dateien lesen. Damit ist es leicht möglich, eine kleine Anwendung zum Verwalten der Texte zu entwickeln und die Text-Tabellen als ADO-Dateien auszuliefern. Recordsets haben gegenüber Ressourcendateien insbesondere den Vorteil, daß man zur Identifizierung von Texten nicht auf Nummern beschränkt ist, sondern sprechende Schlüssel (z.B. „txtBenutztKz“ o.ä.) verwenden kann.

Fail-Fast statt Rollback

Bei unerwarteten Fehlern ist es oft besser, die Prozedur möglichst direkt zu verlassen und sich z.B. auf den Transaktionsschutz einer Datenbank zu verlassen, statt zuerst aufzuräumen oder ein eigenes Rollback zu schreiben, um bereits getätigte Aktionen rückgängig zu machen. Siehe dazu auch „Robuster VB-Code“ in [2].

Verläßt man ein Formular nach einem unerwarteten Fehler, sollte man im *Form_Unload* mit *Set <FormName> = Nothing* das versteckte *Form*-Objekt zerstören, das VB beim *Show* automatisch erzeugt (siehe Beispiel auf der Heft-CD, vgl. auch den Bericht über *Set ... = Nothing* in dieser Ausgabe). Dadurch werden automatisch alle global im Formular definierten Objekte zerstört und durchlaufen ihr *Class_Terminate*.

Windows-API LastDLLError

Viele Funktionen des Windows-API liefern weitere Fehlerinformationen in *Err.LastDLLError*. Den zugehörigen Text ermittelt man mit der API-Funktion *FormatMessage:*

```
Public Function itcApiErrorText(ByVal LastDllError _
  As Long) As String
  Dim MsgLen As Long
  Dim MsgText As String

  MsgText = String$(256, 0)
    MsgLen = FormatMessage_
      (FORMAT_MESSAGE_FROM_SYSTEM Or _
      FORMAT_MESSAGE_IGNORE_INSERTS, _
      0&, LastDllError, 0&, MsgText, _
      Len(MsgText), ByVal pNull)
  If MsgLen Then itcApiErrorText = Left$(MsgText, _
    MsgLen)
End Function
```

Die Funktion *FormatMessage* ist in der Typelib *win.tlb* aus [3] definiert, die Sie ebenfalls auf der Heft-CD finden.

Exceptions sind abfangbar

Einige Fehler sind normalerweise in VB nicht abfangbar, z.B. allgemeine Zugriffsverletzungen (GPF). Sie führen zu einer Windows-Fehlermeldung, und die Anwendung wird beendet. Je nach Konfiguration des PC wird automatisch Dr. Watson gestartet.

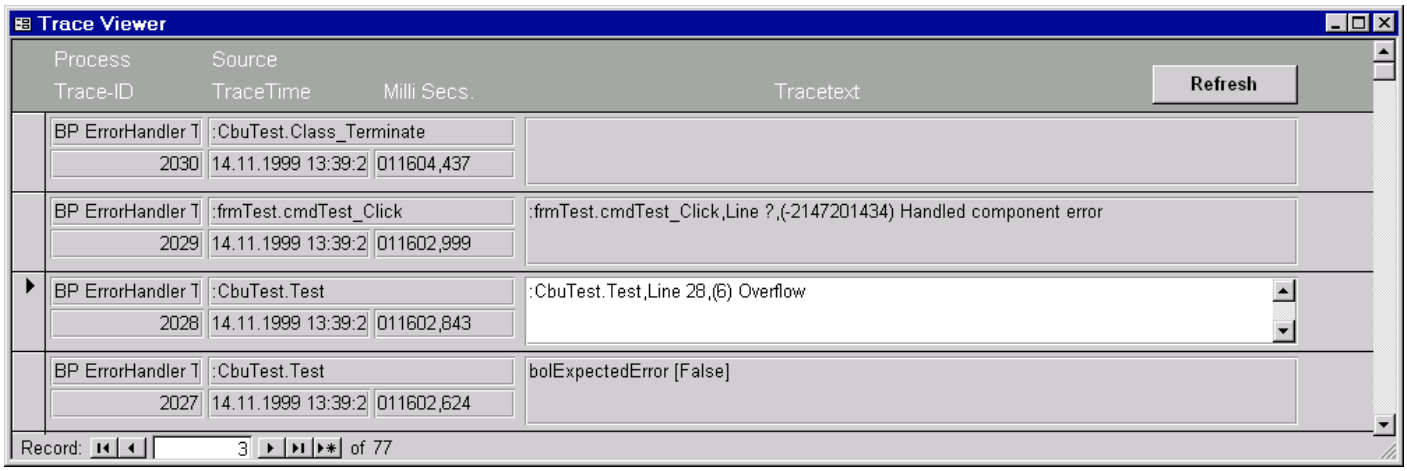

Abbildung 2:
Anzeige eines Trace. Man sieht, daß die Variable bolExpectedError den Wert False hatte, ein Fehler „(6) Overflow“ in CbuTest.Test in Zeile 28 aufgetreten ist, der Aufruf von frmTest.cmdTest_Click kam und das Objekt am Ende sein Class_Terminate durchlaufen hat.
Die Trace-Informationen wurden mit der Methode CTrace.itcTrace in eine Tabelle geschrieben. Die Tabelle und ein Formular zur Anzeige finden Sie auf der Heft-CD in trace.mdb.

Mit ein paar API-Aufrufen können aber auch solche Fehler abgefangen und wie normale Fehler behandelt werden (s. [4]). Diese Möglichkeit ist jedoch mit Vorsicht zu genießen, denn nach solchen Fehlern können Code und Daten eines Prozesses in einem unerwarteten Zustand sein, und die Resultate sind manchmal unvorhersehbar.

Datenbankfehler (RDO, ADO)

Mit RDO wertet man bei einem Datenbank-Fehler (*Err.Number* = 40002) am besten den SQL-Status des letzten Fehlers in der *rdoErrors*-Collection aus:

```
rdoErrors(rdoErrors.Count).SQLState
```

ADO liefert in *Err.Number* spezifischere Datenbankfehler als RDO. Das macht das Behandeln von Datenbankfehlern einfacher. Meist kommt man mit den Fehlern *edbDuplicateRecord* = -2147217900, *edbUpdateConflict* = -2147217864 und *edbRecordNotExists* = -2147217885 aus. Um lesbarere Fehlernummern zu erhalten, könnte man die langen negativen ADO-Fehlernummern um *vbObjectError* demaskieren (man erhält dann z.B. 3021 statt -2147218483). Leider funktioniert das aber nicht bei allen ADO-Fehlern; deshalb lebt man der Konsistenz wegen besser mit den langen Zahlen.

Das Behandeln von Fehlern beim *UpdateBatch* von ADO batch-optimistischen Recordsets ist etwas schwieriger, da ja mehrere unterschiedliche Fehler gleichzeitig auftreten können. Darüber hinaus müssen Fehler bei neu angelegten Records gesondert behandelt werden, und die Methode *Resync* erzeugt überflüssige Fehler, die zu ignorieren sind. Eine ausführliche Beschreibung von batch-optimistischen Recordsets finden Sie in [5].

Fehlerbehandlung ausschalten

Beim Entwickeln ist es oft hilfreich, die Fehlerbehandlung zeitweilig auszuschalten. Dazu kann man unter Extras|Optionen|Allgemein| Fehlerbehandlung eine entsprechende Option wählen. Diese Optionen schalten jedoch alle Error-Handler aus, also z.B. auch Konstrukte mit *On Error Resume Next,* die man meist nicht ausschalten möchte. Hier kann man sich mit einem globalen Schalter behelfen und nur ausgewählte Error-Handler bedingt codieren:

```
If Not bolDebug Then On Error Goto ErrorHandler
```

Error-Handler als Hilfskonstrukt

Error-Handler kann man auch einsetzen, um einfach kurzen und schnellen Code zu schreiben. Der folgende Code ermittelt, ob ein Feld in einer ADO *Fields*-Collection existiert, ohne eine Schleife zu benutzen:

```
Public Function FieldExists(ByVal rst As _
  ADODB.Recordset, ByVal Fieldname As String) _
  As Boolean
  On Error GoTo ErrorHandler
  Call rst.Fields(Fieldname).Name
  FieldExists = True
  Exit Function

ErrorHandler:
  FieldExists = False
End Function
```

Der Error-Handler ersetzt in solchen Fällen oft zeitaufwendige Iterationen oder Prüfungen, die den Zweck des Code weniger leicht erkennbar machen und in den meisten Fällen überflüssig sind, weil keine Sondersituation vorliegt. Ein Beispiel dafür ist folgende Implementierung eines selbstexpandierenden Feldes:

```
Sub MySub(ByRef mabytCol() As Byte)
  ...
  On Error GoTo DimErr
  ColLen = ColLen + 1
  mabytCol(ColLen) = byt
  Exit Sub

DimErr:
  ReDim Preserve mabytCol(1 To ColLen _
    + mChunkSize)
  Resume

End Function
```

Der Code nimmt an, daß die Zuweisung *mabytCol(ColLen)* in den meisten Fällen klappt. Um den seltenen Fall einer Zuweisung jenseits des Array-Endes korrekt zu behandelt, benutzt er jedoch keine explizite Prüfung wie:

```
if ColLen <= UBount(mabytCol) then
  mybytCol(ColLen) = byt
else
  ...
end if
```

Stattdessen „läßt er es darauf ankommen“ und reagiert nur bei einem Fehler. Im Error-Handler vergrößert er dann das Feld nicht nur um 1, sondern um eine größere Anzahl Zellen, damit nicht so schnell wieder ein Überlauf stattfindet.

Cleanup-Konstrukte

Cleanup-Code sollte auch im Fehlerfall durchlaufen werden. Dabei kann man mit einem *Resume* auf eine Cleanup-Marke Redundanzen vermeiden:

```
  ...
Cleanup:
  On Error Resume Next
  cnx.close
  Exit Sub

ErrorHandler:
  Select Case Err
    ...
  End Select
  Resume Cleanup
End Sub
```

Hierbei ist zu beachten, daß auch beim Cleanup Fehler auftreten können, z.B. wenn man versucht, ein nicht geöffnetes Recordset zu schließen. Das kann zu schwer lokalisierbaren Problemen führen, da ein Fehler im Error-Handler einen Fehler im Aufrufer auslöst. Um hier umständliche Konstrukte wie *If Not cnx.ActiveConnection Is Nothing Then...* zu vermeiden, kann man solche Fehler mit einem *On Error Resume Next* unterdrücken.

Vorsicht! Wenn man im Error-Handler mit *Err.Raise* Fehler auslöst, kann man *Resume Cleanup* nicht verwenden, da der Code nach *Err.Raise* nicht ausgeführt wird.

Vermeiden Sie auch überflüssigen Code im Cleanup. Der Autor teilt nicht die weit verbreitete Ansicht, es sei guter VB-Programmierstil, alles, was man erzeugt hat, auch zu zerstören. VB verfügt über eine automatische Garbage Collection. Die VB-Dokumentation sagt: *„All object variables are automatically cleared when they go out of scope“* (siehe dazu auch den Bericht zum Thema *Set ... = Nothing* in diesem Heft). Das Aufräumen mag ja in einem Programmierbeispiel ganz schön aussehen, aber die Erfahrungen aus vielen Code-Reviews und Fehlersuchen in produktiven Systemen zeigen, daß solche Konstrukte nur Ärger machen. Es ist praktisch unmöglich, Cleanup-Code vollständig zu testen. Investieren Sie Ihre Energie lieber in anwendungsspezifischen Code.

Tücken bei der Fehlerbehandlung

Die folgenden Hinweise sollen helfen, einige Stolpersteine bei der Implementierung zu vermeiden.

Probleme mit Fehlercodes als Rückgabewert

Es gibt verschiedene Möglichkeiten, Fehlercodes in Rückgabewerten zu codieren.

Normalerweise verwendet man *False* (=0) für Fehler und *True* (=-1) für Erfolg. Damit lassen sich aber unterschiedliche Fehler nicht unterscheiden:

```
If TuWas() then
  'OK
Else
  'Fehler! Aber welcher?
End If
```

Alternativ kann man 0 für Erfolg und alle anderen Werte für Fehler verwenden. Dabei muß man aber den Rückgabewert zwischenspeichern, da er sonst im Fehlerfall nicht zur Verfügung steht:

```
lngRetVal = TuWas()
If lngRetVal = OK then
  'OK
Else
  ' Fehlerbehandlung...
  Select Case lngRetVal
    Case FILENAME_INVALID
    ...
  End Select
End If
```

Werte ungleich 0 für Fehler zu verwenden, kann leicht zu Mißverständnissen führen. Wenn jemand *If TuWas() then* statt *If TuWas()=OK then* schreibt, führt das zu unerwarteten Ergebnissen. Dokumentieren Sie daher Ihre Strategie genau und seien Sie konsequent bei der Anwendung.

Kein Code nach Err.Raise im Error-Handler

Code, der in einem Error-Handler nach einem *Err.Raise* steht, wird nie ausgeführt. Bei einem *Err.Raise* wird die Kontrolle an den Aufrufer zurückgegeben. In der VB-Literatur und der MS-Dokumentation [6] finden sich leider unzählige Beispiele für diesen Fehler:

```
Sample_Err:
  If Len(strTextToCheck) = 0 Then
    Err.Raise CUSTOM_ERROR, "ErrorExample3", _
      CUSTOM_ERR_DESC
    ErrorExample3 = False
    'Diese Zeile wird niemals ausgeführt
  ...
AddInvoice_Err:
  Err.Raise CUSTOM_ERROR, _
    „clsTest.AddInvoice", Msg
  Resume AddInvoice_End
  'Diese Zeile wird niemals ausgeführt
  ...
```

Err.raise in Class_Initialize erzeugt „Automation Error" in der IDE

Ein *Err.Raise* im Error-Handler der Prozedur *Class_Initialize* einer Klasse führt beim Debuggen in der IDE zu einem allgemeinen Fehler „*440 Automation Error*". Das hat zur Folge, daß Error-Handler im Aufrufer zu falschen Ergebnissen führen. Dieser allgemeine Fehler tritt aber nur auf, wenn man *Err.Raise* in einem Error-Handler verwendet. Er tritt nicht auf, wenn man *Err.Raise* außerhalb des Error-Handlers verwendet. Er tritt ebenfalls nicht auf, wenn man keinen Error-Handler definiert, also Fehler einfach zum Aufrufer durchreicht.

Deshalb sollte man im *Class_Initialize* entweder ganz auf einen Error-Handler verzichten, oder Fehler vollständig lokal behandeln.

Kein Err.Raise in Class.Terminate

Ein *Err.Raise* in der Prozedur *Class_Terminate* einer Klasse führt zu einem nicht abfangbaren Laufzeitfehler mit einer Standard-VB-Messagebox. Dieses Verhalten ist das gleiche wie bei einem *Err.Raise* in Ereignis-Prozeduren. Es ist in der Online-Hilfe dokumentiert.

Kein Err.Raise in Ereignis-Prozeduren

Verwendet man *Err.Raise* in einer Ereignis-Prozedur, führt das zu einem nicht abfangbaren Laufzeitfehler mit einer Standard-VB-Messagebox. Der Fehler wird erst nach Klicken der Schaltfläche „OK" an den Aufrufer weitergegeben. Dabei wird *Err.Description* durch den allgemeinen Text „*Application or Object defined Error*" ersetzt.

Unterdrückt man die Messagebox durch Ankreuzen von „Project Properties, unattended Execution", wird sie mit dem Typ „Info" ins Ereignis-Log umgeleitet, und es wird kein (!) Fehler im Aufrufer ausgelöst. Jeder weitere Aufruf des Objekts, in dem der Fehler aufgetreten ist, erzeugt den Fehler „*The callee (server [not server application]) is not available and disappeared; all connections are invalid. The call may have executed*".

Bei Ereignissen in Formularen verwundert dieses Verhalten nicht. Beim Codieren benutzerdefinierter Ereignisse oder z.B. beim Ereignis „MessageArrived" des MSMQ übersieht man es jedoch leicht. Fehler in Ereignis-Prozeduren müssen also immer vollständig lokal behandelt werden.

Kein Error-Handler in MTS ObjectControl_Deactivate

Ein Error-Handler im MTS-Ereignis *ObjectControl_Deactivate* führt bei einem *SetAbort* im Error-Handler des Aufrufers implizit ein *Err.Clear* aus und setzt damit *Err.Number* = 0. Um sich ein Zwischenspeichern des Error-Objekts zu ersparen, vermeidet man Error-Handler in *ObjectControl_Deactivate* am besten.

Unter COM+ benötigt man die *ObjectControl*-Ereignisse glücklicherweise nicht mehr.

Rückgabe-Parameter nach Err.Raise nicht verfügbar

Wenn man eine Prozedur mit *Err.Raise* verläßt, kann man keine Daten in Parametern zurückgeben. Wenn man solche Parameter braucht, muß man Rückgabewerte statt *Err.Raise* verwenden.

Kein Resume <Marke>

Ein *Resume <Marke>* ist nichts anderes als ein *Goto* rückwärts im Code – und solche Goto-Anweisungen benutzen wir ja nicht mehr, nicht wahr? Diese Konstrukte lassen sich immer vermeiden, indem man Code in einer eigenen Prozedur kapselt. Statt:

```
Public Sub DontDoThis()
  On Error GoTo ErrorHandler

  Do Until rstCust.EOF
    If rstCust!LastDate < Now() Then
      rstReceipt.AddNew ...
      ...
      rstLog.Delete
    End If

NextCustomer:
    rstCust.MoveNext
  Loop
  Exit Sub

ErrorHandler:
  Select Case Err.Number
    Case edbRecordAlreadyExists
      Resume NextCustomer
  ...
End Sub
```

schreiben Sie besser:

```
Public Sub DoThis()
  Do Until rstCust.EOF
    Call UpdateReceipt
    rstCust.MoveNext
  Loop
End Sub

Public Sub UpdateReceipt()
  On Error GoTo ErrorHandler
  If rst!LastDate < Now() Then
    rstReceipt.AddNew ...
    ...
    rstLog.Delete
  End If
  Exit Sub

ErrorHandler:
  Select Case Err.Number
    Case edbRecordAlreadyExists
      'Ignore
  ...
End Sub
```

Der Fehler wird lokal in *UpdateReceipt* behandelt. Die Hauptroutine *DoThis* konzentriert sich voll auf die Iteration und enthält keine unschönen, impliziten *Goto*-Anweisungen. Sie können sie aber natürlich um eine eigene Fehlerbehandlung ergänzen, die sich mit schweren Fehlern beschäftigt, die zum Abbruch der Schleife führen.

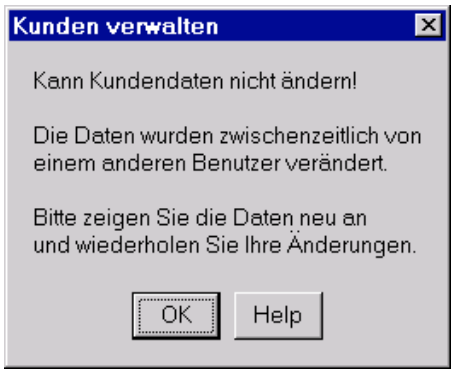

Abbildung 3:
Komplexe Fehlermeldungen sollten immer aus mehreren Teilen bestehen: Kurztext, Beschreibung und Hinweis zur Behebung.

Nicht mehrere Error-Handler in einer Prozedur

Vermeiden Sie mehrere Error-Handler in einer Prozedur. Meist ist es sinnvoller, den Code auf mehrere Prozeduren aufzuteilen.

Meldungen an die Anwender

Zum Thema, wie man Fehler dem Anwender meldet, kann ich Ihnen folgendes Zitat von Bruce McKinney einfach nicht vorenthalten [3]:

Put up a dialog box telling users they are filthy, unwashed idiots with ugly children. Tell them they have just committed error _2147211501 (&H80042713). Display the message „I am unworthy!“ and make them click OK. Then terminate the program, throwing all their wretched, unsaved work into the bit bin.

Sie verstehen: So sollte man es nicht machen. Art und Ton der Fehlermeldung sollten sich immer am Anwender orientieren. Formulieren Sie klar, kurz, positiv und konsistent.

Schreien sie nicht mit Icons, Farben, Großbuchstaben, Blinken oder gar mit Beeps.

Vermeiden Sie Worte wie „Falsch“, „Fehler“ und „Schwerwiegend“. Statt „Das eingegebene Datum ist falsch!“ besser „Bestelldatum ungültig“ oder „Bestelldatum muß später als Heute sein.“

Versuchen Sie nicht vorwegzunehmen, was der Anwender wahrscheinlich tun will, sondern lassen Sie ihm die Kontrolle – selbst wenn das einen Mausklick mehr bedeuten sollte.

Bei komplexeren Problemen empfiehlt sich eine Aufteilung von Meldungen in drei Teile: Kurztext, Beschreibung und Hinweis zur Behebung *(Abbildung 3)*. Der Hinweis zur Behebung kann hinter einer Hilfe-Schaltfläche verborgen sein.

Um ein konsistentes Erscheinungsbild aller Meldungen zu gewährleisten, empfiehlt es sich, die *MsgBox*-Anweisung in einer Funktion zu kapseln, etwa wie folgt:

```
Public Function itcMsgBox(ByVal DescriptionShort _
  As String, ByVal DescriptionLong As StringByVal _
  SolutionHint As String, ByVal intVbButton As _
  VbMsgBoxStyle, ByVal strTitle As String, _
  ParamArray Params()) As VbMsgBoxResult
```

Log & Trace

Fehler protokollieren

Ein *Log* dient zum Dokumentieren von Fehlern und wesentlichen Statusinformationen (z.B. Start und Stop) einer Anwendung. Mit VB bietet es sich an, das Windows-Ereignis-Log zu verwenden. Mit der VB-Anweisung *Appl.LogEvent* kann man leicht ins Ereignis-Log oder in ein eigenes Logfile schreiben (vgl. [7]). Es gibt Werkzeuge, die Ereignis-Logs auswerten (z.B. unterstützt Crystal Reports Ereignis-Logs als Datenquelle für Berichte), und Werkzeuge, die Log-Meldungen automatisch per EMail an Administratoren weiterleiten.

Den Anwendungsablauf verfolgen

Ein *Trace* dient dazu, den Ablauf einer Anwendung detailliert zu verfolgen und die Zustände während des Ablaufs (z.B. Werte von Parametern oder SQL-Kommandos) zu protokollieren. Die Aufrufe von *itcTrace* (siehe *BPTracer.vbp* auf der Heft-CD) im folgenden Code schreiben Einträge in die Trace-Tabelle in einer Access-Datenbank (siehe *trace.mdb* auf der Heft-CD und *Abbildung 2*). Dabei liefert *itcTrace* als Rückgabewert den Trace-Text mit ersetzten Platzhaltern. Dieser wird im Beispiel als *Err.Description* verwendet.

```
'Den Ablauf tracen
itcTrace trlproc, mModuleName & _
  ProcedureName, "Start"
...
lngRetVal = WaitForSingleObject(mhFileNotify, _
  INFINITE)

If lngRetVal <> 0 Then
  'Einen Fehler tracen
  Err.Raise mrcFileNotify, mModuleName & _
    ProcedureName, itcTrace(trlErr, mModuleName _
    & ProcedureName, "WaitForSingleObject failed _
    with error %1 on Folder %2.", Err.LastDllError, _
    mFolderName)
End If
...
'Einen Status tracen
itcTrace trlproc, mModuleName & _
  ProcedureName & mSubFolderName, _
  "Change in folder %1 notified.", _
  mFolderName
```

Viele Anwendungen lassen sich nicht mit einem interaktiven Debugger testen, z.B. verteilte oder zeitkritische Anwendungen. Oft braucht man auch in der Produktion einen permanent mitlaufenden Trace, um bei Problemen nachweisfähig zu sein oder Fehler nachvollziehen zu können. Viele Fehler sind nicht im Labor reproduzierbar.

Beim Trace ist es sehr hilfreich, wenn das Trace-Level von außen konfigurierbar ist, d.h. die Detailliertheit bzw. die Anzahl/Häufigkeit der Trace-Protokolleinträge. Dazu definiert jede Trace-Zeile über einen Parameter, bei welchen Trace-Levels sie ins Trace geschrieben werden soll: *Call itcTrace (CallLevel := trlErr, TraceText := „Fehler... “)*. Die Tracer-Funktion entscheidet bei jedem Aufruf, ob ein Eintrag ins Protokoll geschrieben werden soll, indem sie das *CallLevel* mit dem eingestellten Trace-Level vergleicht: *If (CallLevel And gTraceLevel) = CallLevel*. Damit kann man den Detaillierungsgrad der Trace-Infos steuern und Performance-Einbußen durch das Trace minimieren (in komplexen Algorithmen kann jede dritte Zeile ein Trace-Aufruf sein).

Tracer kann man auf verschiedenste Weise implementieren, z.B. als Funktionen, die in Files oder Datenbanken schreiben, oder als COM-Server, die Trace-Infos in einem Dialog anzeigen. Welche Implementierung jeweils die beste ist, hängt wesentlich von den Anforderungen an Performance, Zuverlässigkeit und von der Architektur der Anwendung ab.

Trace in eine Datei ist ca. zehnmal schneller als in eine Datenbank. Eine Datenbank ist aber sicherer als ein File.

Eine verteilte Anwendung kann leicht in eine gemeinsame Datenbank-Tabelle schreiben, aber nicht in eine gemeinsame Datei.

In eine Datei kann man aber auch noch schreiben, wenn eine Datenbank nicht mehr verfügbar ist. Bei Datei I/O kann man nur dann sicher sein, daß beim Absturz eines Prozesses (und das ist oft genau die Situation, in der ein Trace wichtig ist) der letzte Eintrag auch wirklich geschrieben wurde, wenn man nach jedem Schreiben mit einem expliziten *Flush* einen I/O auf die Platte erzwingt.

COM-Server sind nur zuverlässig genug, wenn man sie über asynchrone Schnittstellen anspricht.

Von MuTek Solutions gibt es das Werkzeug *BugTrapper,* mit dem Programme ohne Modifikation im Code nachträglich mit einem Tracing versehen werden können. Das hört sich interessant an, allerdings hat der Autor bisher keine Erfahrung damit. Auf der beiliegenden CD finden Sie als Beispiel einen Tracer, der in eine Datenbank schreibt.

Traces können große Mengen an Plattenplatz belegen. Deshalb ist es wichtig, geeignete Mechanismen zur Begrenzung des Speicherplatzes (z.B. rollierende Files) zu implementieren.

Assertions

Um die Richtigkeit von Prozeduren möglichst vollständig sicherzustellen, kann man *Assertions* („Absicherungen“) verwenden. Dabei prüft man alle Eingangs- und Ausgangsdaten auf Gültigkeit.

In VB kann man dazu *Debug.Assert <Ausdruck>* verwenden. Ist der Wert des Ausdrucks *False,* wird der Code angehalten. Allerdings wird das in VB eingebaute *Debug.Assert* analog zu *Debug.Print* nur in der Entwicklungsumgebung ausgeführt. Beide Anweisungen sind also im übersetzten Code nicht vorhanden. Will man Assertions in übersetztem Code verwenden (z.B. bei einem Beta-Test), muß man eine eigene *Assert*-Prozedur einsetzen (vgl. [3]). Eine solche Prozedur sollte abschaltbar sein und Fehlerart und -ort dokumentieren:

```
...
BugAssert (DateMin < DateMax), “DateMin _
  < DateMax“, mModulenem & ProcedureName
...
Sub BugAssert(byval bolAssertion as Boolean, _
  byval strExpression as String, byval strSource _
  as String)
#If gbolDebug then
  If bolAssertion Then Exit Sub
  MsgBox “Assertion failed: “ & strExpression _
    & “ in “ strSource
  Stop
#End If
End Sub
```

Damit Größe und Geschwindigkeit des Codes nicht durch den aggressiven Einsatz von Assertions leiden, verwendet man sie meist nur während der Entwicklung und beim Testen. Fehler, die auch im Echtbetrieb auftreten können, sollten nicht als Assertions codiert werden *(BugAssert (DateMin < DateMax, ...)),* sondern Fehler auslösen: *If DateMin < DateMax then Err.Raise ...*

Fazit

Die Behandlung von Fehlern und Ausnahmesituationen ist ein wesentlicher Bestandteil jeder Anwendung. Legen Sie eine Strategie zur Fehlerbehandlung fest und definieren Sie Standards.

Die schönsten Standards nützen allerdings nichts, wenn sie nicht befolgt werden. Deshalb zeigen Sie Selbstdisziplin, sorgen Sie für Disziplin in Ihrem Team und setzen Sie Ihre Standards durch permanente gegenseitige Code-Reviews durch.

Schreiben Sie am besten Error-Handler immer sofort und nicht nachträglich.

Und machen Sie keine falschen Fehler! ;-) ▼

Ressourcen

[1] Ralf Westphal: *Sprachentwirrung mit VB;* BasicPro 5/99, Seite 64

[2] Peter Meinl: *Fehlertolerante VB-Systeme mit dem MS Cluster Server;* BasicPro 4/99, Seite 22

[3] Bruce McKinney: *Hardcore Visual Basic;* Microsoft Press 1997. Gehört in jedes Regal. Unter *www.vb-zone.com/upload/free/features/vbpj/1999/mckinney/mckinneytoc.asp* gibt es ein Update des Codes auf VB6.

[4] Jonathan Lunman: *No Exception Errors, My Dear Dr. Watson;* Visual Basic Programmer’s Journal 5/99, S. 108: Beschreibt, wie man Exceptions in VB abfangen und behandeln kann.

[5] Ralf Westphal: *Batch-optimistische Recordsets benutzen;* BasicPro 6/98, S. 44

[6] Falsch codierte Error-Handler findet man in der MS-Dokumentation z.B. in *„Microsoft Office 2000/Visual Basic Programmer’s Guide, Writing Error-Free Code“,* hier steht Code nach *Err.Raise;* oder in der Fitch & Mather Stocks-Beispiel-Anwendung *msdn.microsoft.com/library/techart/fmstocks_dal_sql.htm*: hier sieht man jede Menge überflüssige *Set... = Nothing,* aber das wichtige *cnx.Close* fehlt, das die Connection für den Pool freigibt.

[7] Stefan Henneken: *Zugriff auf das Event-Log von Windows NT;* BasicPro 6/98, Seite 56

Diplom-Informatiker (FH) Peter Meinl, Jg. 1957, studierte Allgemeine Informatik an der Fachhochschule Furtwangen. Von 1980 bis 1985 arbeitete er bei PSI und Bosch an der Entwicklung von Produktionsplanungssystemen auf der Basis fehlertoleranter Tandem-Rechner. Seit 1985 ist er Berater bei der ISTEC GmbH. Schwerpunkt seiner derzeitigen Tätigkeit ist das Design von kundenspezifischen Systemen in der Fertigungsindustrie auf der Basis von Microsoft-Produkten. Für Fragen und Anregungen erreichen Sie Peter Meinl per EMail an PeterM@BasicPro.de.