Veröffentlichung: 12/2004.

Rekonstruktionsfähigkeit

Software wird in ihrer Betriebsphase häufig unerwartet teuer: schwer zu reproduzierende Produktionsfehler und aufwendige Fehleranalysen führen zu hohen Wartungskosten. Dieser Artikel zeigt, welche Vorkehrungen bereits in der Entwicklungsphase einer Software getroffen werden sollten, um spätere Fehleranalysen beschleunigen und Wartungskosten reduzieren zu können.

Manu Carus

Bei Betrachtung des gesamten Lebenszyklus einer Software, von der Unterzeichnung des Projektauftrags bis zur Deinstallation des Produkts, verursacht allein die Wartung über 70% der Kosten einer Software. Während die Entwicklungskosten mit Hilfe einer detaillierten Planung relativ genau abgeschätzt werden können, bleiben die später entstehenden Wartungskosten unkalkulierbar. Wer kann voraussehen, welche Fehler in der Betriebsphase der Software auftreten werden? Wie häufig treten Fehler auf? Wie aufwendig wird es sein, die Ursache solcher Fehler zu ermitteln? Welche Komponenten, Datenbankprozeduren und Eingabeformulare müssen zur Bereinigung des Fehlers angepasst werden? Ist eine Migration erforderlich, um eventuell entstandene Dateninkonsistenzen zu bereinigen? All diese Unwägbarkeiten inkl. der Erfordernis, eine neue Programmversion erstellen, testen und ausliefern zu müssen, führen dazu, dass die Wartungskosten einer Software ein wesentliches betriebswirtschaftliches Risiko darstellen können.

Fehleranalyse - Ein Beispiel

Ein Anwender ruft den Kundensupport an und meldet, dass in der Portalapplikation bei der Änderung seiner Bankverbindung ein unerwarteter Fehler mit der Nummer „4711“ aufgetreten ist. Der Support-Mitarbeiter kann dem Fehlerprotokoll aus der Datenbank die in Tabelle 1 aufgeführten Informationen entnehmen.

Fehler-ID:	4711
Datum:	22.08.2004
Uhrzeit:	09:45
Exception.Message:	Object reference not set to an instance of an object.
Exception.StackTrace:	at de.iCommit.BlzAccountValidator.Validate(String strBlz, String strAccount) at de.iCommit.Portal.Payment.ChangeBankData(String strCustomerId, String strBlz, String strAccount, String strAccountOwner) at ChangeBankDataPage.btnSubmit_Click(Object objSender, EventArgs objEventArgs)
Exception.Source:	ChecksumValidation
Exception.TargetSite:	Void Validate(System.String, System.String)

Tabelle 1: Fehlerprotokoll (Beispiel)

Was ist passiert? Folgende Erkenntnisse lassen sich gewinnen:

- Die Applikation hat zur Laufzeit auf ein nicht-existentes Objekt zugegriffen („Object reference not set to an instance of an object.“).
- Der Fehler tritt in der Methode *Void Validate(System.String, System.String)* der Klasse *de.iCommit.BlzAccountValidator* auf, die der Assembly *ChecksumValidation* zugeordnet ist.
- Laut Stack-Trace wurden die folgenden Methoden ausgeführt: *btnSubmit_Click()* auf der ASP.NET-Seite *ChangeBankDataPage*, dann Aufruf der Methode *ChangeBankData()* der Klasse *de.iCommit.Portal.Payment*, abschließend Aufruf der Methode *Validate()* der Klasse *de.iCommit.BlzAccountValidator*, in der letztlich die Exception ausgelöst wurde.

Der First-Level-Support wird den Fehler anhand dieser Informationen kaum identifizieren können. Folglich steht der Second-Level-Support, gegebenenfalls ein Administrator oder die Entwickler der Applikation selbst vor dem Problem, mit Hilfe der oben aufgeführten, in der Datenbank protokollierten Informationen das Problem nachzustellen, den Fehler aufzufinden und zu bereinigen. Leider sind die protokollierten Fehlerinformationen nicht aussagekräftig genug:

- Auf welche Variable, die zum Zeitpunkt des Fehlers eine null- bzw. Nothing-Referenz hatte, wurde zugegriffen? Handelte es sich um einen formalen Parameter? Um eine Hilfsvariable? Oder um eine Property oder Membervariable der Klasse?
- Wie lauteten die Werte der beiden Parameter, die an die fehlerauslösende Methode Validate() übergeben wurden?
- Wie lauteten die Werte lokaler Variablen innerhalb der Methode Validate() zum Zeitpunkt des Fehlers?
- Welche Werte wurden an die anderen Methoden entlang des Call-Stacks übergeben?

Folgende Faktoren führen aufgrund langwieriger Fehleranalyse zu hohen Wartungskosten:

- Der fehlersuchende Entwickler hat keinen Zugriff auf die konkreten Parameter- und Variablenwerte der aufgerufenen Methoden zum Zeitpunkt des Fehlers.
- Die spezielle Datenkonstellation, die zu dem Fehler führte, ist nicht bekannt. Offensichtlich handelt es sich um einen bisher nicht getesteten Sonderfall, ansonsten hätte der Fehler nicht auftreten können.
- Der Entwickler muss gegebenenfalls viele unterschiedliche Testfälle durchführen, um den aufgetretenen Fehler reproduzieren zu können.
- Hierzu muss er sich in den bestehenden Code einarbeiten, den meist nicht er selbst, sondern ein Kollege vor längerer Zeit implementiert hat.
- Besonders aufwändig wird es, wenn der Fehler nur in Verbindung mit bestimmten Konfigurationsdaten der Anwendung auftritt, und sich diese Konfigurationsdaten zwischen der Produktions- und Testumgebung unterscheiden (z.B. ein Bankleitzahlenverzeichnis zur Überprüfung auf gültige BLZs).

Die in der Praxis meist angewandten Fehlerbehandlungsroutinen sind nicht ausreichend, um Fehler schnell und effizient analysieren zu können und ihre Ursache ausfindig zu machen. Wenn die Applikation lediglich die aufgetretene Exception abfängt und in der Datenbank die Properties dieser Exception protokolliert, stehen First-Level- und Second-Level-Support, im Zweifelsfall die Entwickler der Applikation selbst, genau vor den oben aufgeführten. Problemen, nämlich vor der Situation, ausgehend von der Fehlermeldung „Object reference not set to an instance of an object.“ den Produktionsfehler auffinden zu müssen. Selbst dem protokollierten Stack-Trace können dabei keine wesentlichen Informationen entnommen werden. Der Stack-Trace bildet den zum Zeitpunkt

des Fehlers entstandenen Programmpfad ab (Call Stack). Aber dieser Programmpfad wird durch eine statische Code-Analyse, die bei der Analyse des Fehlers durchgeführt werden muss, eh abgeleitet.

Rekonstruktionsfähigkeit

In der Betriebsphase einer Software auftretende Fehler können nur dann schnell und effizient analysiert werden, wenn die Applikation so viele Informationen wie möglich über die Fehlersituation bereitstellen kann: über die Benutzer der Applikation, und insbesondere darüber, welche Interaktionen sie durchgeführt haben. Bei Auftreten eines Fehlers ist es dann zu jedem späteren Zeitpunkt möglich, den Zustand der Applikation zum Zeitpunkt dieses Fehlers zu rekonstruieren.

Diese Eigenschaft einer Software wird als **Rekonstruktionsfähigkeit** bezeichnet. Gemeint ist damit die Fähigkeit der Applikation, zu jedem Zeitpunkt einen vollständigen Überblick über den Zustand der Applikation vermitteln zu können.

Wie wäre es, wenn ein Produktionsfehler auftritt und sich der First-Level-Support mit Hilfe einer einfachen Auswertung einen lückenlosen Überblick über die Session des betroffenen Benutzers verschaffen kann? Es könnte nachvollzogen werden, welche Interaktionen der Anwender durchgeführt hat, welche ASP.NET-Seiten aufgerufen wurden, welche Eingaben getätigt wurden, und letztlich welche Werte die zum Zeitpunkt des Fehlers definierten Session-Variablen hatten.

Listing 1 zeigt einen Call-Stack, dem zusätzlich die Inhalte von Parametern, lokalen Variablen, Properties und Konfigurationsdaten zugeordnet wurden. Hier wurden nicht nur die Standard-Properties der .NET-Basisklasse *System.Exception* berücksichtigt, sondern eigene, benutzerdefinierte Properties hinzugefügt, wie beispielsweise ein Beschreibungsfeld, das alle wesentlichen Variablenwerte der aufgerufenen Methode zum Zeitpunkt des Fehlers aufführt. Exceptions wurden miteinander verkettet, damit nicht nur Informationen über die fehlerauslösende Methode protokolliert werden können, sondern Informationen über sämtliche Methoden, die entlang des entstandenen Call-Stacks aufgerufen wurden. Zudem wurde jedem Fehler ein eindeutiger Fehler-Code zugeordnet, über den der Systemadministrator in der Datenbank auswerten kann, ob der Fehler bereits mehrfach bisher aufgetreten ist. Die allgemeinen Fehlerinformationen am Anfang des Call-Stacks referenzieren die Session des betroffenen Benutzers sowie den Server, auf dem der Fehler aufgetreten ist. Wird der Fehler beispielsweise immer nur auf einem bestimmten Server festgestellt, so legt dies in einer Web-Farm, bestehend aus mehreren Web-Servern, den Verdacht eines Konfigurations- bzw. Deployment-Problems nahe.

Listing 1: Erweiterter Call-Stack

Allgemeine Informationen

Fehler-ID:	0815
Datum:	22.08.2004
Uhrzeit:	09:45
Session:	5cf2e2cc-edc4-4d44-a343-63b7885ff3fe
Server:	**PORTAL002**

Exception

Interne ID:	4711
Fehler-Code:	**02059**
Meldung:	Object reference not set to an instance of an object.
Typ:	System.NullReferenceException
Komponente:	de.iCommit.BlzAccountValidator

Methode:	Void Validate(System.String, System.String)
Beschreibung:	**BlzAccountValidator [strBlz = '73362500', strAccount = '1234567890', this.BlzFilename = 'c:\bundesbank\blz.xml', strChecksum = null, strValidate = null]**

Innere Exception

Interne ID:	4712
Fehler-Code:	**03160**
Meldung:	change bank data failed
Typ:	de.iCommit.Portal.Common.PortalException
Komponente:	de.iCommit.Portal.Payment
Methode:	Void ChangeBankData(String strCustomerId, String strBlz, String strAccount, String strAccountOwner)
Beschreibung:	**Payment [strCustomerId = '123456', strBlz = '73362500', strAccount = '1234567890', strAccountOwner = 'Max Mustermann']**

Innere Exception

Interne ID:	4713
Fehler-Code:	**05382**
Meldung:	change bank data failed
Typ:	de.iCommit.Portal.Common.PortalException
Komponente:	/aspx/Contractor/ChangeBankData.aspx
Methode:	Void btnSubmit_Click(Object objSender, EventArgs objEventArgs)
Beschreibung:	**ChangeBankDataPage[User = '123456', Role = 'Customer', strCustomerId = '123456', txtBlz = '73362500', txtAccount = '1234567890', txtAccountOwner = 'Max Mustermann']**

Tabelle 2 zeigt ein Protokoll der bisherigen Benutzerinteraktionen. Diesem Protokoll kann genauestens entnommen werden, welche ASP.NET-Seiten aufgerufen wurden und welche Eingaben der Benutzer getätigt hat. Beispielsweise wurde in dem Rechnungsportal erfolgreich ein Login durchgeführt (Pos. 3) und die Rechnungsübersicht aufgerufen (Pos. 6); anschließend hat der Benutzer seine Juli-Rechnung im PDF-Format abgerufen (Pos. 8), die Telefonverbindungen dieses Monats ausgewertet (Pos. 11), seine E-Mail-Adresse für den nächsten Rechnungsversand geändert (Pos. 15) und seine Bankverbindung geändert (Pos. 18). Bei den durchgeführten Änderungen wurden sogar die bisherigen Werte festgehalten, um eine höhere Transparenz zu schaffen (Pos. 15).

Nr.	Zeit	Aktion	Information
1	22.08.2004 09:41:07	ASP.NET-Page	GET: https://icommit2/portal/
2	22.08.2004 09:41:08	ASP.NET-Page	POST: https://icommit2/portal/browser/private/en/aspx/login/Login.aspx
3	22.08.2004 09:41:08	Login	
4	22.08.2004 09:41:09	ASP.NET-Page	GET: https://icommit2/portal/browser/private/en/aspx/Customer/Welcome.aspx
5	22.08.2004 09:41:15	ASP.NET-Page	GET: https://icommit2/portal/browser/private/en/aspx/Customer/InvoiceSummary.aspx
6	22.08.2004 09:41:15	Invoice Summary	
7	22.08.2004 09:41:48	ASP.NET-Page	GET: https://icommit2/portal/browser/private/en/aspx/Customer/Invoice.pdf
8	22.08.2004 09:41:48	Invoice PDF	Month = '07/2004'
9	22.08.2004 09:41:57	ASP.NET-Page	GET: https://icommit2/portal/browser/private/en/aspx/Customer/AnalyzeCdrs.aspx
10	22.08.2004 09:42:15	ASP.NET-Page	POST: https://icommit2/portal/browser/private/en/aspx/Customer/AnalyzeCdrs.aspx
11	22.08.2004 09:42:15	Analyze Cdrs	Filter: [Month = '07/2004', DateFrom = '20040715', DateTo = '20040722', Zone = 'Internet'] OrderBy: ['Euro']
12	22.08.2004 09:43:22	ASP.NET-Page	GET: https://icommit2/portal/browser/private/en/aspx/Customer/Welcome.aspx
13	22.08.2004 09:43:26	ASP.NET-Page	GET: https://icommit2/portal/browser/private/en/aspx/Customer/ChangeMail.aspx
14	22.08.2004 09:43:49	ASP.NET-Page	POST: https://icommit2/portal/browser/private/en/aspx/Customer/ChangeMail.aspx
15	22.08.2004 09:43:50	Change Mail	Old Value = 'mustermann@t-online.de' New Value = 'max@mustermann.de'
16	22.08.2004 09:44:21	ASP.NET-Page	GET: https://icommit2/portal/browser/private/en/aspx/Customer/ChangeBankData.aspx
17	22.08.2004 09:44:38	ASP.NET-Page	POST: https://icommit2/portal/browser/private/en/aspx/Customer/ChangeBankData.aspx
18	22.08.2004 09:44:39	Change Bank Data	txtBlz = '73362500', txtAccount = '1234567890', txtAccountOwner = 'Max Mustermann'

Tabelle 2: Benutzerinteraktionen

I.d.R. speichern Portalapplikationen bestimmte Informationen und Zustände in sog. Session-Variablen, die ihrerseits für den Hergang eines Fehlers relevant sein können. Tabelle 3 zeigt die Werte solcher Session-Variablen zum Zeitpunkt des Fehlers auf, beispielsweise die ID und Rolle des angemeldeten Benutzers sowie ein zwischengespeichertes XML-Dokument mit den bisherigen Stammdaten des Benutzers.

Session-Variable	Wert
SESSION_ID	"5cf2e2cc-edc4-4d44-a343-63b7885ff3fe"
USER_ID	"123456"
USER_ROLE	"Customer"
USER_LANGUAGE	"en"
USER_LAST_LOGIN_DATE	"20040718"
USER_LAST_LOGIN_TIME	"14:38:01"
USER_EMAIL	"max@mustermann.de"
USER_START_URL	"https://icommit2/portal/browser/ private/en/aspx/Customer/Welcome.aspx"
USER_FIRST_LOGIN	"false"
USER_CRM_XML	<?xml version="1.0" encoding="ISO-8859-1"?> <customer id="123456"> <contact> <first-name>Max</first-name> <last-name>Mustermann</last-name> </contact> <address> <street>Musterstr.</street> <house>1a</house> <zipcode>12345</zipcode> <city>Musterstadt</city> </address> <payment> <german-bank-account> <blz>37050198</blz> <account>1234567897</account> <owner>Max Mustermann</owner> </german-bank-account> </payment> </customer>

Tabelle 3: Session-Variablen

Mit Hilfe dieser Informationen wird eine sehr hohe Transparenz in der Applikation geschaffen. Der First-Level-Support wird in die Lage versetzt, die Session eines Benutzers ausführlich und lückenlos nachvollziehen zu können; er ist nicht auf Angaben durch den Benutzer selbst angewiesen, welche oft ungenau und unvollständig beschrieben werden. Der Second-Level-Support sowie Administrator und Entwickler der Applikation selbst können sich anhand der Informationen, die dem Call-Stack beigefügt wurden, wesentlich schneller und genauer einen Überblick über die entstandene Fehlersituation verschaffen. Im Zweifelsfall ist es mit Hilfe dieser Protokolle gar nicht erforderlich, sich in den bestehenden Code einarbeiten zu müssen: die Hilfswerte der in Listing 1 aufgeführten Exception, [strBlz = '73362500', strAccount = '1234567890', this.BlzFilename = 'c:\bundesbank\blz.xml', strChecksum = null, strValidate = null], legen den Verdacht nahe, dass die eingegebene Bankleitzahl 73362500 in dem Bankleitzahlen-Verzeichnis *c:\bundesbank\blz.xml* entweder nicht vorhanden ist, oder der mit ihr verknüpfte Prüfziffernalgorithmus für die Kontonummer 1234567890 dort nicht eingetragen ist, weshalb die Variablen *strChecksum* und *strValidate* vermutlich kein gültiges Objekt referenzieren.

In fünf Schritten zum Ziel

Mit Hilfe der Rekonstruktionsfähigkeit können die Wartungskosten einer Software empfindlich gesenkt werden, da langwierige, statische Code-Analysen bei der Fehlersuche entfallen oder zumindest deutlich verkürzt werden können.

Wie kann dieses Ziel erreicht werden? Indem zusätzlicher Entwicklungsaufwand geleistet wird, um im Fehlerfall ausführliche Informationen über den Hergang des Fehlers liefern zu können. Indem alle Benutzerinteraktionen lückenlos protokolliert werden. Und indem gleichzeitig sichergestellt wird, dass keine Daten im System gelöscht werden können, falls, fachlich bedingt, nicht anders erforderlich.

Um den Zustand einer Benutzersession zu jedem späteren Zeitpunkt rekonstruieren zu können, muss wie folgt verfahren werden:

1. Die auf dem Web-Server erzeugten Sessions werden grundsätzlich in der Datenbank verwaltet.
2. Die gesamte Verwaltung der Session-Variablen wird vom Web-Server in die Datenbank verlagert.
3. Benutzer müssen sich authentifizieren, bevor sie das Portal betreten können, so dass zum frühestmöglichen Zeitpunkt im System bekannt ist, welcher Benutzer Daten liest, ändert, löscht.
4. Die Interaktionen des Benutzers werden in der Datenbank aufgezeichnet.
5. Datenlöschungen werden unterbunden.
   Datenänderungen werden protokolliert.

Das Entity-Relationship-Diagramm (ERD) in Abbildung 1 zeigt, wie diese fünf Schritte in einem Datenmodell umgesetzt werden können.

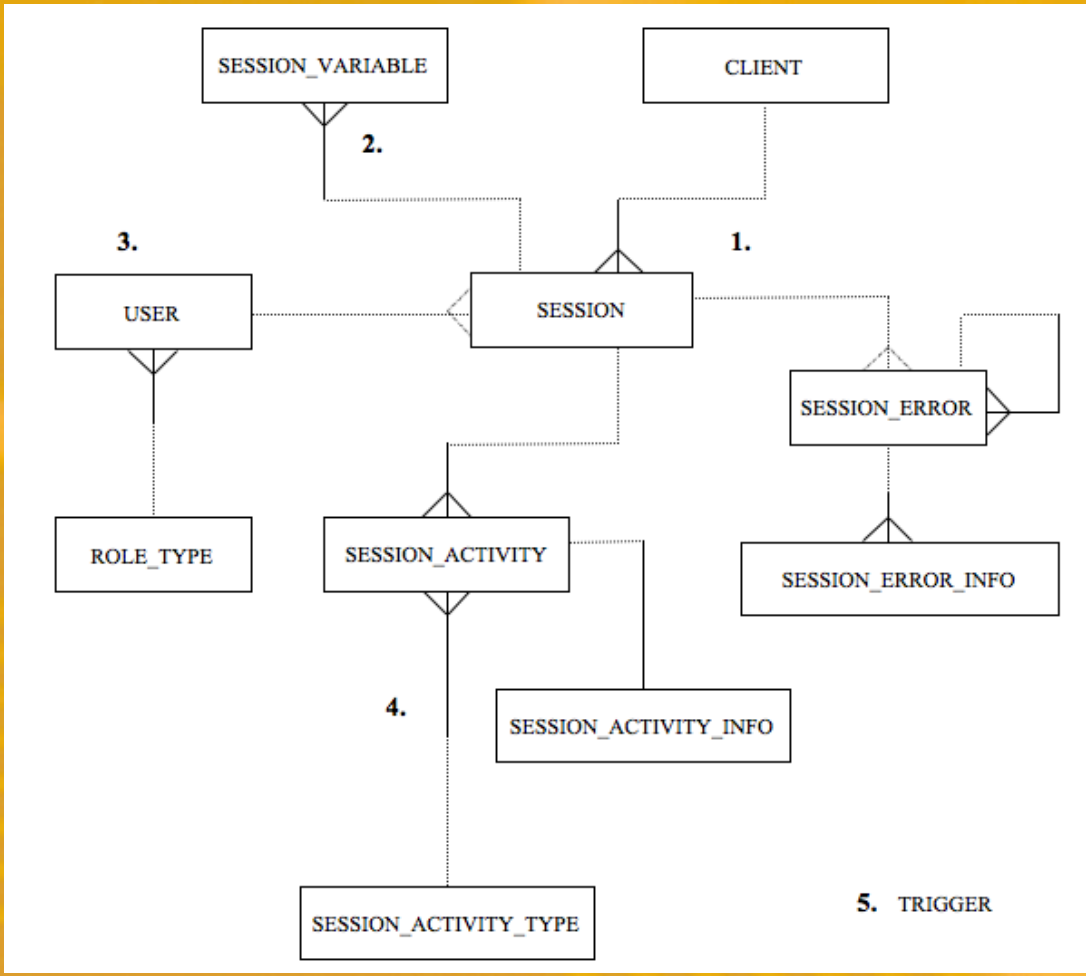

Abb. 5: Entity-Relationship-Diagramm

Schritt 1: Verwaltung von Sessions in der Datenbank

Das zentrale Datenbankobjekt ist die Session. Ohne vorher eine Entität vom Typ *SESSION* in der Datenbank zu erzeugen, kann keine Session-Variable gespeichert, keine Benutzerinteraktion protokolliert und kein Benutzer authentifiziert werden. Die Session wird durch einen eindeutigen Key identifiziert (z.B. 5cf2e2cc-edc4-4d44-a343-63b7885ff3fe); darüber hinaus werden die Start- und Endezeit der Session sowie die IP-Adresse des Benutzers festgehalten.

Über die Tabelle *CLIENT* werden der Session weitere Informationen über den verwendeten Browser des Benutzers zugeordnet, beispielsweise welcher Browser (z.B. „IE“), welche Version (z.B. „6.0“), welches Betriebssystem (z.B. „WinXP“), und ob der Browser grundsätzlich JavaScript und/oder Cookies unterstützt. Die Tabelle *CLIENT* wächst dynamisch, d.h. ist der Browser des Benutzers dort bereits eingetragen, wird eine Relation zu der *SESSION* gesetzt, ansonsten muss vorher eine neue Entität vom Typ *CLIENT* in der Datenbank erzeugt werden. Clients werden durch den HTTP-User-Agent, den der Browser bei jedem HTTP-Request an den Web-Server sendet, eindeutig identifiziert, beispielsweise „Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.0.3705; .NET CLR 1.1.4322)“.

Exceptions werden mit den folgenden Informationen in der Tabelle SESSION_ERROR protokolliert (vgl. Listing 1):

- ein allgemeiner Text (Property *System.Exception.Message*)
- eine eigenentwickelte, detaillierte Beschreibung des Fehlers einschl. der Werte aller relevanten Variablen, Parameter, Properties, Hilfsvariablen und Konfigurationsdaten,
- die Komponente oder ASP.NET-Seite, in der der Fehler aufgetreten ist (Property *System.Exception.Source*), beispielsweise *de.iCommit.BlzAccountValidator* oder */aspx/Contractor/ChangeBankData.aspx*,
- die Methode, in der der Fehler aufgetreten ist (Property *System.Exception.TargetSite*), beispielsweise *Void Validate(System.String, System.String)*,
- der Typ der Exception (Property *System.Exception.GetType().FullName*), beispielsweise *System.NullReferenceException*,
- der Stack-Trace (Property *System.Exception.StackTrace*),
- ein systemweit eindeutiger Fehler-Code, über den das Mehrfachauftreten dieses Fehlers administrativ ausgewertet werden kann, beispielsweise „02059“,
- Datum und Uhrzeit, wann der Fehler aufgetreten ist,
- der Computername oder die IP-Adresse des Web-Servers, auf dem der Fehler aufgetreten ist (z.B. „PORTAL002“).

Besonders zu berücksichtigen bei der Protokollierung von Fehlern sind die nachfolgenden Aspekte.

Verkettung von Exceptions

Exceptions können über die Property *System.Exception.InnerException* miteinander verkettet werden. Wird in jeder einzelnen Methode einer jeden Klasse ein Exception-Handler implementiert (try/catch), der

- die aufgetretene Exception abfängt,
- eine neue Exception erzeugt (beispielsweise durch Implementation einer eigenen, von *System.Exception.ApplicationException* abgeleiteten Klasse namens *PortalException*),
- dieser Exception detaillierte Informationen wie beispielsweise aktuelle Variablen- und Parameterwerte in Form einer einfache Zeichenkette zuordnet (Beschreibungsfeld)
- und diese beiden Exceptions über die Property *System.Exception.InnerException* miteinander verknüpft,

so ergibt sich eine lineare Liste von Exceptions, die den bisher entstandenen Call-Stack exakt widerspiegelt.

Abbildung 2 veranschaulicht die Analogie zwischen Call-Stack und Exception-Kette. Die linke Seite stellt den entstandenen Call-Stack dar, also den in der Property *System*.Exception.StackTrace aufgeführten Informationen, beginnend mit der innersten, zuerst aufgetretenen Exception in der Komponente *de.iCommit.BlzAccountValidator*, über die Komponente *de.iCommit.Portal.Payment* bis in die Code-Behind-Klasse *ChankgeBankDataPage* der ASP.NET-Seite */aspx/Contractor/ChangeBankData.aspx*. Die rechte Seite zeigt die zugehörige Exception-Kette: in jeder Methode wurde eine Exception abgefangen, eine neue, detaillierte Exception erzeugt und mit der abgefangenen Exception verknüpft. Aufgrund der 1:1-Beziehung zwischen Methode und Exception ergibt sich im Gesamtbild eine 1:1-Beziehung zwischen Call-Stack und Exception-Kette.

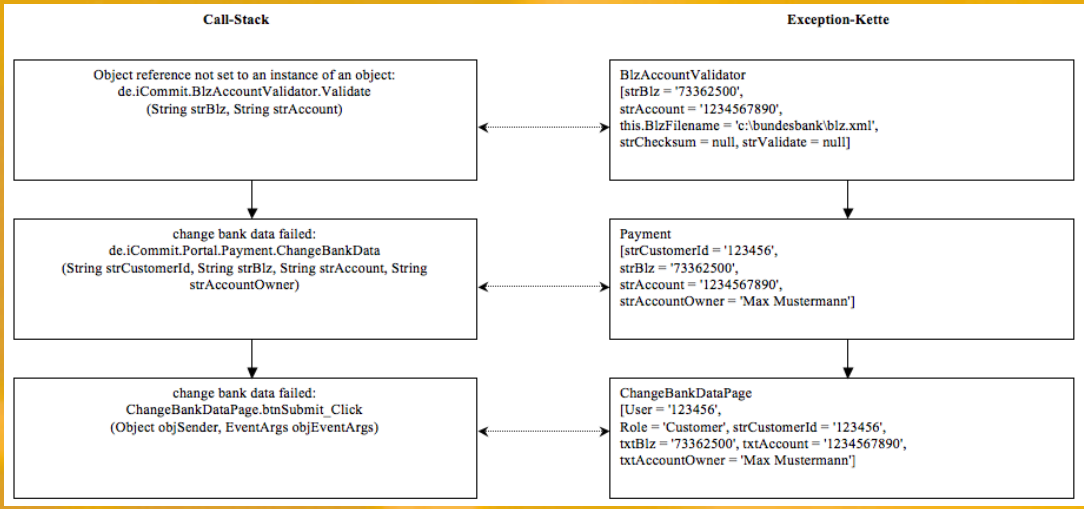

Abb. 2: Analogie zwischen Call-Stack und Exceptions

Diese Analogie ist nur dann möglich, wenn in jeder Methode ein Exception-Handler implementiert wird, der die oben aufgeführten. Aufgaben wahrnimmt. Listing 2 zeigt am Beispiel der Methode *ChangeBankData* die Implementierung eines solchen Exception-Handlers in VB.NET. In der Datenbank wird die Verkettung von Exceptions durch die Relation von *SESSION_ERROR* auf sich selbst ausgedrückt: jeder Eintrag zeigt auf die Wurzel der Exception-Kette, d.h. auf den zuerst innerhalb der Exception-Kette aufgetretenen Fehler.

Listing 2: Implementierung eines Exception-Handlers in VB.NET

```
' Payment.vb

Imports System
Imports de.iCommit

Namespace de.iCommit.Portal

  Public Class Payment
    ' ...

    Public Sub ChangeBankData(ByVal strCustomerId As String,
                              ByVal strBlz As String,
                              ByVal strAccount As String,
                              ByVal strAccountOwner As String)
      Try

        ' ...

        Dim objValidator As BlzAccountValidator = Nothing

        objValidator = New BlzAccountValidator()
        objValidator.Validate(strAccount, strBlz)

        ' ...

      Catch ex As Exception

        String strDescription = Me.ToString() +
                                " [strCustomerId = '" + strCustomerId + "', " +
                                "  strBlz = '" + strBlz + "', " +
                                "  strAccount = '" + strAccount + "', " +
                                "  strAccountOwner = '" + strAccountOwner + "']"

        Throw New PortalException("03160",                        // error code
                                  "change bank data failed",      // message
                                  strDescription,                 // description
                                  Request.Path,                   // component
                                  ex.TargetSite.ToString(),       // method
                                  ex)                             // inner exception
      End Try

    End Sub

    Public Overrides Function ToString() As String
      Return Me.GetType().Name
    End Function

  End Class

End Namespace
```

Physische Grenzen in der Datenbank

Stack-Trace und Beschreibungsfeld können die physischen Einschränkungen des eingesetzten Datenbanksystems überschreiten. Insbesondere wird bei der Fehlerprotokollierung eine Zeichenkette mit allen Properties aller Exceptions aus der in Abbildung 2 aufgeführten Exception-Kette zusammengesetzt. Ist für die Ablage dieses Beschreibungsfeldes in der Datenbank eine maximale Feldlänge von nur 4.000 Zeichen möglich, so darf der Inhalt der zu speichernden Information nicht einfach abgeschnitten werden. Stattdessen ist der gesamte Inhalt des Beschreibungsfeldes in 4 kB – Chunks aufzuteilen, d.h. in Form mehrerer, konsekutiver Datensätze zu speichern und in Relation zu setzen (siehe Entitätstyp *SESSION_ERROR_INFO*).

Fehler vor dem Aufbau einer Session

Bevor ein Benutzer das Portal betreten kann, also eine beliebige ASP.NET-Seite der Portalapplikation aufrufen oder sich authentifizieren kann, ist eine Session in der Datenbank zu erzeugen. Leider ist nicht auszuschließen, dass bei oder vor diesem Prozess ein Fehler auftreten kann. Daher müssen in der Datenbank auch solche Fehler protokolliert werden können, die nicht einer Session zugeordnet werden können. Dies wird im ERD durch die optionale Relation zwischen *SESSION_ERROR* und *SESSION* ausgedrückt (NULLable).

Schritt 2: Verwaltung von Session-Variablen

Session-Variablen können in der Datenbank mit Hilfe einer einfachen Name-Value-Zuordnung und einer Relation zu der *SESSION* verwaltet werden. Daher reichen in der Tabelle *SESSION_VARIABLE* die Attribute *NAME* und *VALUE* zur Darstellung der Inhalte von Session-Variablen vollkommen aus. Um lange Inhalte wie beispielsweise XML-Dokumente unter Beachtung der physischen Einschränkungen des Datenbanksystems ablegen zu können, ist auch hier wieder eine Aufteilung in 4 kB – Chunks denkbar.

Schritt 3: Authentifizierung von Benutzern

Die für die Nutzung des Portals zugelassenen Benutzer werden in der Tabelle *USER* verwaltet. Benutzer werden dort durch ihre Benutzerkennung (User-ID) identifiziert. Die Tabelle *USER* umfasst zudem Attribute wie beispielsweise einen Passwort-Hash zur sicheren Authentifizierung der Benutzer sowie einige allgemeine Informationen (E-Mail-Adresse, etc.).

Bei erstmaligem Aufruf einer ASP.NET-Seite der Portalapplikation wird der Benutzer mit Hilfe der .NET-FormsAuthentication auf die Login-Seite des Portals umgeleitet. Auf diese Weise wird der Benutzer gezwungen, sich zunächst zu authentifizieren, bevor er die Funktionalitäten des Portals nutzen kann. Viel wichtiger für spätere Fehleranalysen ist jedoch, dass von diesem Zeitpunkt an bekannt ist, welcher Benutzer im Rahmen der aufgezeichneten Session die Daten im System ändert oder löscht.

Schritt 4: Protokollierung der Benutzerinteraktionen

Werden alle Benutzerinteraktionen lückenlos und vollständig im System abgebildet, kann eine hohe Transparenz für Administratoren, First-Level-Support, Second-Level-Support und Entwickler einer Software geschaffen werden. Zu diesem Zweck wird im Datenmodell eine Grunddatentabelle *SESSION_ACTIVITY_TYPE* eingeführt, die eine Liste aller protokollierbaren Benutzerinteraktionen enthält, beispielsweise

ID	Protokolltyp	Bedeutung
1	ASP.NET Page	ASP.NET-Seite aufgerufen
2	Login	Benutzer (erfolgreich) authentifiziert
3	Invoice Summary	Rechnungsübersicht aufgerufen
4	Invoice PDF	Rechnungsdokument im PDF-Format abgerufen
5	Analyze Cdrs	Verbindungsdaten ausgewertet
6	Change Mail	E-Mail-Adresse geändert
7	Change Bank Data	Bankverbindung geändert

Bei Eintreten eines der in dieser Grunddatentabelle aufgeführten Ereignisses muss lediglich der entsprechende Protokolleintrag in der Tabelle *SESSION_ACTIVITY* erzeugt und mit dem jeweiligen Protokolltyp in *SESSION_ACTIVITY_TYPE* in Relation gesetzt werden. Zudem sind natürlich Datum und Uhrzeit des Protokolleintrags festzuhalten. Um zusätzliche Informationen zu der Interaktion speichern zu können, wird die Tabelle *SESSION_ACITIVITY_INFO* eingeführt. In dieser Tabelle kann in Relation zu dem jeweiligen Eintrag beispielsweise protokolliert werden, welche ASP.NET-Seite aufgerufen wurde, welches Rechnungsdokument abgerufen wurde, welche Filter- und Sortierkriterien bei der Auswertung von Verbindungsdaten angegeben wurden und wie die E-Mail-Adresse oder Bankverbindung des Benutzers vor der Durchführung einer Änderung lautete.

Schritt 5: Löschung und Änderung von Daten

Soll der Zustand einer Applikation zu jedem späteren Zeitpunkt rekonstruiert werden können, muss sichergestellt werden, dass keinerlei Daten in der Datenbank gelöscht oder geändert werden können. Daten, die fachlich bedingt geändert werden dürfen, sind nachvollziehbar zu ändern, d.h. durch Protokollierung des vorherigen Wertes. Beide Anforderungen können sehr wirkungsvoll durch Trigger und Stored Procedures implementiert werden.

Unterbindung von Datenlöschungen

Aus dem in Abbildung 5 aufgeführten Datenmodell geht beispielsweise hervor, dass Sessions nicht gelöscht werden dürfen, denn dies würde bedeuten, dass alle mit der Session verbundenen Informationen für die nachfolgende Fehleranalyse verloren gingen (Session-Variablen, Error-Logs, Benutzerinteraktionen). Daher werden Sessions beendet, indem einfach das Ende-Datum der Session gesetzt und der Datensatz ansonsten in der Datenbank erhalten bleibt.

Durch Implementation eines einfachen Datenbank-Triggers kann das unerwünschte Löschen von Entitäten in der Datenbank wirksam unterbunden werden. Listing 3 zeigt am Beispiel eines T-SQL-Triggers, wie ein Datenbankfehler bei dem Versuch einer DELETE-Anweisung erzeugt werden kann.

Listing 3: Verhindern von Datenlöschungen mit Hilfe von Triggern (T-SQL)

```
CREATE TRIGGER portal_owner.tr_session_delete ON portaldb.portal_owner.session
FOR DELETE AS
BEGIN
  RAISERROR('-20999:sessions must not be deleted', 16, 1) WITH SETERROR, NOWAIT
  RETURN
END
GO
```

Unterbindung von Datenänderungen

Der ändernde Zugriff auf Daten kann mit Hilfe von Triggern ähnlich effektiv unterbunden werden. Enthält die Tabelle *SESSION* die Attribute *ID*, *SESSION_KEY*, *STARTED_AT*, *ENDED_AT*, *IP_ADDRESS*, *CLIENT_ID* und *USER_ID*, so darf weder der Primary Key ID noch der Session-Key im nachträglich geändert werden. Auch die Startzeit der Session sowie die IP-Adresse des Clients und der verwendete Browser

(*CLIENT_ID*) dürfen nachträglich nicht mehr verändert werden. Lediglich die Ende-Zeit der Session kann nachträglich gesetzt werden. Ist der Benutzer authentifiziert, so darf auch die Referenz *USER_ID* zwischen Benutzer und Session im Datenmodell nachträglich gesetzt werden.

Listing 4 zeigt am Beispiel eines T-SQL-Triggers, wie Änderungen auf der Entität *SESSION* unterbunden werden können. Bei dem Versuch, ein Attribut außer *ENDED_AT* und *USER_ID* nachträglich zu ändern, wird eine Exception erzeugt. Wird das Ende-Datum der Session gesetzt, oder die Referenz auf den authentifizierten Benutzer, so wird zudem überprüft, ob der Wert von NULL auf NOT NULL gesetzt wird. In allen anderen Fällen wird ebenfalls eine Exception erzeugt.

Listing 4: Verhindern von Datenänderungen mit Hilfe von Triggern (T-SQL)

```
CREATE TRIGGER portal_owner.tr_session_update ON portaldb.portal_owner.session
FOR UPDATE AS
BEGIN

  /* Attributes in table SESSION are:

     # ID
     * SESSION_KEY
     * STARTED_AT
     o ENDED_AT
     * IP_ADDRESS
     * CLIENT_ID
     o USER_ID

  */

  IF NOT UPDATE(ended_at) AND NOT UPDATE(user_id)
  BEGIN
    RAISERROR('-20999:sessions must not be changed except for attributes "ended_at"
               and "user_id"', 16, 1) WITH SETERROR, NOWAIT
    RETURN
  END

  IF UPDATE(ended_at)
  BEGIN
    IF NOT EXISTS (SELECT 'True'
                   FROM   Deleted d, Inserted i
                   WHERE  d.id = i.id
                   AND    d.ended_at IS NULL
                   AND    i.ended_at IS NOT NULL
                  )
    BEGIN
      RAISERROR('-20999:attribute "ended_at" may only be changed from null to a
                        non-null value', 16, 1) WITH SETERROR, NOWAIT
      RETURN
    END
  END

  IF UPDATE(user_id)
  BEGIN
    IF NOT EXISTS (SELECT 'True'
                   FROM   Deleted d, Inserted i
                   WHERE  d.id = i.id
                   AND    d.portal_user_id IS NULL
                   AND    i.portal_user_id IS NOT NULL
                  )
    BEGIN
      RAISERROR('-20999:attribute "portal_user_id" may only be changed from null to
                        a non-null value', 16, 1) WITH SETERROR, NOWAIT
      RETURN
    END
  END

END
GO
```

Protokollierung von Datenänderungen

Wenn fachlich bedingte Datenänderungen im System erfolgen, so ist zu protokollieren:

- Welcher Benutzer hat die Änderung durchgeführt?
- Welches Datum wurde geändert?
- Wie lautete der vorherige Wert?
- Mit welchem neuen Wert wurde überschrieben?
- Wann wurde die Änderung durchgeführt?

Bei der Protokollierung von Änderungen ist zu beachten, dass die Protokollierungsmechanismen nicht explizit aufgerufen werden müssen. Würde beispielsweise eine Stored Procedure für das Logging implementiert, bestünde die Gefahr, dass die ASP.NET-Entwickler der Portalapplikation diese Stored Procedure vergessen aufzurufen. Das Ergebnis wäre ein lückenhaftes, unvollständiges Protokoll und damit nicht geeignet für eine spätere Fehleranalyse.

Das Ziel der Protokollierung muss vielmehr sein, die Protokollierung in die bestehenden Stored Procedures zur Änderung von Daten zu integrieren. Einerseits spart man sich dadurch den Aufwand, die Protokollmechanismen explizit aufzurufen, und man läuft nicht Gefahr, den Aufruf dieser Stored Procedures zu vergessen. Andererseits, und viel wichtiger in diesem Zusammenhang, ist die Tatsache, dass beide Aktionen in der Stored Procedure, das Ändern von Daten sowie die Protokollierung dieser Änderung gemeinsam transaktioniert werden. Keine Datenänderung ohne Benutzerprotokoll.

Listing 5 zeigt am Beispiel einer T-SQL-Stored-Procedure, wie die Änderung einer E-Mail-Adresse inkl. Protokollierung der geänderten Daten implementiert werden kann. Vor der Durchführung einer Änderung wird die bisherige E-Mail-Adresse des Benutzers selektiert und zwischengespeichert. Nach der Durchführung des UPDATEs wird die Änderung durch Aufruf einer Hilfsprozedur namens *LogSessionActivity()* in der Datenbank gespeichert. Dabei wird in dem Aktivitätenprotokoll sowohl die bisherige als auch die neue E-Mail-Adresse gespeichert. Zwecks Protokollierung der Datenänderung ist es erforderlich, der Stored Procedure einen weiteren, bisher nicht erforderlichen Parameter *@pSession* zu übergeben, um den erzeugten Protokolleintrag in Relation zu der Session des Benutzers setzen zu können.

Listing 5: Protokollierung von Änderungen (T-SQL)

```
------------------------
---   SetUserEmail   ---
------------------------

CREATE PROCEDURE portal_owner.SetUserEmail @pUser    NVarChar(25),
                                           @pEmail   NVarChar(50),
                                           @pSession NVarChar(36)
AS
BEGIN

  DECLARE @vUserEmail NVarChar(50)

  SELECT @vUserEmail = email
  FROM   portaldb.portal_owner.user
  WHERE  login = @pUser

  UPDATE portaldb.portal_owner.user
  SET    email = @pEmail
  WHERE  login = @pUser

  DECLARE @vSessionActivityId BigInt
  EXECUTE @vSessionActivityId = portal_owner.LogSessionActivity
                                          @pSession        = @pSession,
                                          @pActivityType   = 'change user email',
                                          @pAdditionalInfo = NULL,
                                          @pOldValue       = @vPortalUserEmail,
                                          @pNewValue       = @pEmail
END
GO
```

Fazit

Die Wartungskosten stellen den teuersten und aufwändigsten Anteil im Lebenszyklus einer Software dar. Die Ursache hoher Wartungskosten sind häufig aufwendige Fehleranalysen und –bereinigungen. Durch die Implementierung ausgefeilter Logging-Mechanismen und umfassender Fehlerbehandlungsroutinen können später in der Betriebsphase entstehende Wartungskosten eines Software-Produkts empfindlich gesenkt werden. Der Schlüssel hierzu ist die „Rekonstruktionsfähigkeit“, die Fähigkeit einer Software, ihren (Fehler-)Zustand zu jedem späteren Zeitpunkt rekonstruieren zu können, um einen vollständigen Einblick in Variableninhalte, Session-Variablen und Benutzerinteraktionen erhalten zu können.

Kritik

Die beschriebenen Mechanismen erfordern einen gewissen Implementierungsaufwand, der in herkömmlichen Entwicklungen bisher meist nicht berücksichtigt wird. Jedoch fällt dieser Entwicklungsaufwand nur einmalig an. Dem stehen wiederholte Situationen im Produktionsbetrieb der Software gegenüber, die weder zahlen- noch kostenmäßig eingeschätzt werden können. Plant das Projektteam die Entwicklung der beschriebenen Mechanismen in die Entwicklungsphase ein, so kann der zusätzlich entstehende Aufwand zeit- und kostenmäßig abgeschätzt werden. Produktionsfehler hingegen sind nicht einschätzbar und stellen somit ein nicht kalkulierbares Risiko dar.

Zudem kann man den beschriebenen Mechanismen entgegenhalten, dass ihre Anwendung hohe Speicherplatzanforderungen stellen. Sicherlich werden in den meisten Software-Systemen nicht derlei umfangreiche Interaktionsprotokolle und Fehlerbeschreibungen gespeichert, so dass bei einem Massenbetrieb

durchaus mehrere GB Plattenkapazität einzurechnen sind. Jedoch können Datenbank-Jobs implementiert werden, die die aufgezeichneten Sessions, Benutzerinteraktionen, Session-Variablen und Fehler rollierend beispielsweise nach drei Monaten löschen.

Letzter Kritikpunkt mag die Frage nach der Performance sein. Die in die Stored Procedures integrierten Protokollierungen werden schnell und effizient innerhalb der Datenbank durchgeführt. Lediglich im Fehlerfalle werden Exceptions abgefangen, aufbereitet, Informationen zusammengestellt und in der Datenbank protokolliert. Dabei kann die Performance einer Anwendung im Fehlerfall eher vernachlässigt werden: dem Benutzer wird eher an einer zügigen Behebung des Fehlers gelegen sein.

Nichts geht ohne Session...

Das in Abbildung 5 vorgestellte Datenmodell birgt eine weitere Idee: keine Funktionalität lässt sich innerhalb des Portals durchführen, ohne zuvor eine Session in der Datenbank anzulegen. Kein Login, kein Abruf von Rechnungsdokumenten, keine Änderung von Daten. Doch wer sagt, dass dieses Prinzip nur auf Portalapplikationen angewendet werden kann? Was spricht dagegen, bei Aufruf einer beliebigen Desktop-Applikation eine Session zu erzeugen, den aktuellen Benutzer (explizit oder implizit) zu authentifizieren und alle Interaktionen und Fehler in der oben beschriebenen Form zu protokollieren? Warum sollte nicht auch ein SOAP-Web-Service zunächst eine Session erzeugen und den Aufrufer auf die beschriebene Weise authentifizieren? Warum sollten Systemadministratoren Grunddaten oder Konfigurationsdaten im System ändern dürfen, ohne sich authentifizieren zu müssen, und ohne dass die durchgeführten Änderungen im System protokolliert werden? Warum sollten Migrationen auf dem Datenstamm einer Software durchgeführt werden dürfen, ohne dass nachträglich nachvollzogen werden kann, wer die Daten geändert hat und wie die Daten vorher beschaffen waren?

Die Rekonstruktionsfähigkeit einer Software schafft vor dem Hintergrund dieser Anwendungsbeispiele eine hohe Transparenz und deutlich reduzierte Wartungskosten.

Ausblick

Die Rekonstruktionsfähigkeit ist eine von mehreren Maßnahmen, die Entwickler zur Erhöhung der Qualität ihrer Software umsetzen sollten. Weitere Konzepte sind: defensive Programmierung, automatisierte Code-Reviews, Testfallautomatisierung, Sicherheit und Performance.

In der nächsten Ausgabe wird das Konzept der defensiven Programmierung für mehrschichtige Anwendungen vorgestellt. Am Beispiel einer Portalapplikation werden Programmiermuster und Source Codes in VB.NET und T-SQL gezeigt, mit deren Hilfe die Stabilität, Zuverlässigkeit und Sicherheit einer Software wesentlich erhöht werden kann.

Manu Carus ist unabhängiger Technologie-Berater und Software-Ingenieur der iCommit Integrationslösungen GmbH. Seine Schwerpunkte sind Software-Architektur, technische Spezifikationen, Integration sowie Sicherheits-, Performance- und Skalierungsaspekte. Sie erreichen ihn unter manu.carus@ethical-hacking.de.