



# Machen Sie mehr aus Ihren Fehlern!

In jeder Anwendung treten Fehler auf. Darauf sollte der Entwickler seine Anwendung vorbereiten. Der Benutzer muss eine möglichst aussagekräftige Meldung erhalten. Detaillierte Informationen über die Umstände, unter denen der Fehler aufgetreten ist, erleichtern dem Entwickler die spätere Analyse. Der Artikel stellt verschiedene Möglichkeiten der Fehleranalyse vor. Eine selbst entwickelte Hilfsklasse erleichtert die Weitergabe von Fehlerinformationen und eine generische Dialogklasse stellt die Fehler dar.

\_\_\_ Entwickler verbringen viel Zeit mit der Analyse von Programmfehlern. Dies kann eine mühselige Aufgabe sein, besonders dann, wenn ein Fehler in einem Produktivsystem auftritt und von einem Anwender gemeldet wird. Häufig sind die Informationen über den Fehler spärlich und vermitteln nur einen vagen Eindruck von der möglichen Ursache. Die Situation wird noch schwieriger, wenn eine Anwendung aufgrund einer unbehandelten Ausnahme beendet wird. Eine klare Strategie bei der Fehlerbehandlung ist daher von Vorteil.

Im Folgenden werden einige Informationsquellen behandelt, die bei der Fehleranalyse in .NET-Anwendungen nützlich sein können. Im Anschluss wird eine generische Fehler-Reporting-Klasse vorgestellt, die alle relevanten Informationen sammelt. Darauf baut ein Fehlerdialog auf, der es Anwendern und Entwicklern ermöglicht, sich ein klares Bild vom aufgetretenen Fehler zu verschaffen und die notwendigen Schritte einzuleiten.

## Das Exception-Objekt

Die wichtigste Informationsquelle bei der Fehleranalyse ist das *Exception*-Objekt. Es liefert Informationen über die Art und Ursache des Fehlers in Form von Eigenschaften.

- *Message* enthält den Fehlertext.
- *Source* gibt den Namen der Anwendung zurück, in der die Ausnahme aufgetreten ist.
- *TargetSite* gibt den Namen der Methode zurück, in der die Ausnahme aufgetreten ist.

- *HelpFile* gibt den URL zur Hilfedatei an, in der ein Hilfetext zum aufgetretenen Fehler hinterlegt ist.
- *InnerException* gibt ein eingebettetes *Exception*-Objekt zurück.

## StackTrace-Analyse

Zur Analyse des aufgetretenen Fehlers bietet es sich außerdem an, die Aufrufkette der Methoden zurückzuverfolgen. Dazu stellt das *Exception*-Objekt die schreibgeschützte Eigenschaft *StackTrace* zur Verfügung. Diese enthält eine Liste aller Methodenaufrufe, die zwischen dem Code liegen, der die Ausnahme ausgelöst hat, und dem, der sie abgefangen hat.

Hierbei ist jedoch zu beachten, dass der Startpunkt der Aufrufkette durch das Weiterleiten einer Ausnahme beeinflusst werden kann. Im folgenden Beispiel ruft die Funktion *Calculate* die Funktion *Calculate2* auf. Tritt ein Fehler auf, leitet sie im *catch*-Block die Ausnahme mit *throw* an den Aufrufer weiter.

```
void Calculate()
try {
    Calculate2();
}
catch (Exception e) {
    throw e;
}
```

Auf diese Weise wird die Aufrufkette zurückgesetzt. Das hat zur Folge, dass die Aufrufkette *Calculate* als Ursprung der Ausnahme anzeigt und nicht *Calculate2*. Wird die Ausnahme jedoch ohne Angabe eines *Exception*-Objekts weitergeleitet, bleibt die Aufrufkette unverändert.

```
void Calculate()
try {
    Calculate2();
}
catch {
    throw;
}
```

Eine Zeile aus der *StackTrace*-Zeichenkette hat folgenden Aufbau:

```
at Namespace.Klassenname.Methodenname([Parameter]) in Dateiname: Line
Zeilennummer
```

## SUMMARY

### Auf einen Blick

Der Artikel baut eine konsequente Strategie für die Behandlung von Fehlern auf. Eine generische *Exception*-Klasse sammelt die nötigen Informationen und ein generischer Fehlerdialog stellt die Informationen übersichtlich dar.

### Eingesetzte Anwendungen

Visual Studio .NET

### CD-Code

Basics02

### Autor

Jörg Neumann ist Programmierer bei der Firma KEEP IT SIMPLE GmbH in Hamburg. Bei Fragen und Anregungen erreichen Sie ihn unter [Joerg.Neumann@KEEPITSIMPLE.de](mailto:Joerg.Neumann@KEEPITSIMPLE.de).



Neben der *StackTrace*-Eigenschaft des *Exception*-Objekts gibt es noch weitere Möglichkeiten, die Aufrufkette einer Anwendung zu analysieren. Dies kann erforderlich sein, wenn auch solche *StackTrace*-Informationen benötigt werden, die vor dem Auslösen einer Ausnahme liegen. Hierfür lässt sich zum Beispiel die *StackTrace*-Eigenschaft der statischen Klasse *Environment* auswerten. Diese enthält die aktuelle Aufrufkette der Anwendung.

Zu beachten ist, dass der Aufrufer beim Zugriff auf das *Environment*-Objekt uneingeschränkte Rechte vom Typ *EnvironmentPermission* benötigt. Dies ist standardmäßig der Fall.

Noch mehr Möglichkeiten bietet die *StackTrace*-Klasse aus dem Namespace *System.Diagnostics*. Sie ermöglicht das individuelle Erstellen einer Aufrufkette. Über die verschiedenen Konstruktoren kann ein *StackTrace*-Objekt für den aktuellen Kontext, einen Thread, eine Ausnahme oder ein angegebenes *StackFrame*-Objekt erstellt werden. Darüber hinaus besteht die Möglichkeit, über den *skipFrames*-Parameter eine definierte Anzahl von Elementen der Aufrufkette, also von *StackFrame*-Objekten, zu überspringen. Dies ist sinnvoll, wenn zum Beispiel der Name der Methode, die den *StackTrace* auswertet, nicht mit ausgegeben werden soll. Nach dem Erstellen eines *StackTrace*-Objekts kann zum Beispiel über die Eigenschaft *FrameCount* die Anzahl der enthaltenen *StackFrame*-Objekte ermittelt werden.

Um Zugriff auf die Frames der Aufrufkette zu erhalten, können mit der *GetFrame*-Methode einzelne *StackFrame*-Objekte erzeugt werden. Diese liefern über die Methoden *GetMethod*, *GetFileName* und *GetFileLineNumber* nicht nur die bekannten Infor-

mationen der *StackTrace*-Zeichenfolge, sondern sie geben zum Beispiel über *GetFileColumnNumber* auch die Spaltennummer im Code zurück.

*GetMethod* gibt hierbei neben dem Namen der Methode ein Objekt vom Typ *System.Reflection.MethodBase* zurück, welches detaillierte Informationen über die Methode und ihre Parameter bereitstellt. Außerdem kann mit *GetILOffset* und *GetNativeOffset* die Position im IL- beziehungsweise im nativen JIT-kompilierten Code ermittelt werden, was für eine Code-Analyse mit einem Disassembler hilfreich ist.

In Listing 1 wird eine Aufrufkette auf der Grundlage einer Ausnahme erstellt. Daraus werden Informationen wie Klasse, Methode, Parameter sowie Zeilen-/Spaltennummer, IL-Offset und der Dateiname der Methode ermittelt, in der die Ausnahme ausgelöst wurde. Eine Ausgabe könnte etwa so aussehen:

```
Fehler in ExceptionHandling.frmMain.Calculate2(Int32 i, Int32 q)
in Zeile 164, Spalte 5, ILOffset: 7, Datei:
"c:\exceptionhandling\frmmain.cs".
```

Zu beachten ist, dass Dateiname, Zeilennummern und Spaltennummer standardmäßig nicht in der Release-Version der Anwendung zur Verfügung stehen, da diese aus den Debug-Symbolen extrahiert werden.

### Weitere nützliche Informationen

Neben den Eigenschaften des *Exception*-Objekts und der Aufrufkette gibt es weitere Informationen, die bei der Fehleranalyse von Interesse sein können. So lässt sich etwa über die *WorkingSet*-Eigenschaft der *Environment*-Klasse der Umfang des Speichers ermitteln, der dem aktiven Prozess zugewiesen wurde. Eine weitere nützliche Information ist der Name der *AppDomain*, in der gerade gearbeitet wird. Dieser wird wie folgt ermittelt:

```
string appDomain = AppDomain.CurrentDomain.FriendlyName;
```

Bei Multithreading-Anwendungen sind neben den üblichen Fehlerangaben Informationen über einen bestimmten Thread von Vorteil. Aber auch bei Zugriffsproblemen im Zusammenhang mit Code Access Security können diese Informationen nützlich sein. Nachfolgend einige Eigenschaften aus dem *System.Threading*-Namespace, die es lohnt auszulesen:

- *Thread.CurrentThread.CurrentUICulture* liefert ein *CultureInfo*-Objekt zurück, das Informationen über das eingestellte Gebietsschema enthält, unter dem der Prozess läuft.
- *Thread.CurrentThread.Name* gibt den Namen des aktuellen Threads zurück, sofern er benannt wurde.
- *Thread.CurrentPrincipal.Identity.Name* gibt den Namen des Benutzers des aktuellen Threads zurück.
- *System.AppDomain.GetCurrentThreadId()* gibt die ID des aktuellen Threads zurück.

### Systeminformationen

Sehr nützlich können auch Systeminformationen des Rechners sein. Über *Environment.OSVersion* kann ein *OperatingSystem*-Objekt angefordert werden, das Auskunft über das installierte Betriebssystem gibt. Die Eigenschaft *Platform* liefert den ID-Wert des Betriebssystems und *Version* stellt ein *Version*-Objekt

#### Listing 1 Informationen zur Ausnahme sammeln.

```
...
// StackTrace aus Exception erstellen
StackTrace trace = new StackTrace(ex, true);

// StackFrame aus StackTrace ermitteln
StackFrame frame = trace.GetFrame(0);

// Klassen- und Funktionsnamen aus StackFrame ermitteln
MethodBase method = frame.GetMethod();
string traceString = "Fehler in " +
method.DeclaringType + "." +
method.Name + "(";

// Methoden-Parameter ermitteln
ParameterInfo[] paramInfos = method.GetParameters();
string methodParams = "";
foreach (ParameterInfo paramInfo in paramInfos)
{
    methodParams += " " + paramInfo.ParameterType.Name + " " +
paramInfo.Name;
}
if (methodParams.Length > 2)
    traceString += methodParams.Substring(2);

// Zeile, Spalte, ILOffset und Datei ermitteln
traceString +=
    ") in Zeile " + frame.GetFileLineNumber().ToString() +
    ", Spalte " + frame.GetFileColumnNumber().ToString() +
    ", ILOffset: " + frame.GetILOffset() +
    ", Datei: \" + frame.GetFileName() + "\" +
    Environment.NewLine;

// Fehlerzeile ausgeben
Console.WriteLine(traceString);
...
```

zur Verfügung. Zur Darstellung kann auch die überschriebene *ToString*-Methode verwendet werden.

Möglicherweise ist auch die Version des installierten .NET Framework von Interesse. Diese lässt sich über die Eigenschaft *Version* der *Environment*-Klasse ermitteln. Sie liefert ein *Version*-Objekt, welches die Version des .NET Framework enthält, unter dem die Anwendung läuft. Auch hier kann die überschriebene *ToString*-Methode zur Darstellung verwendet werden.

Zur späteren Fehleranalyse sollten auch die wichtigsten Programminformationen gespeichert werden. Dazu zählen:

- Programmtitel (*Application.ProductName*),
- Version (*Application.ProductVersion*) und
- Arbeitsverzeichnis (*Application.ExecutablePath*).

### Debugger nutzen

Um den aufgetretenen Fehler zur Laufzeit analysieren zu können, bietet es sich an, aus der Anwendung heraus eine Debug-Sitzung zu starten. Hierfür stellt der *System.Diagnostics*-Namespace die statische Klasse *Debugger* zu Verfügung. Mit ihr kann ein installierter Debugger gestartet und gesteuert werden. Das Starten eines Debuggers wird durch die *Launch*-Methode initiiert. Diese sucht in der Registry nach installierten Debuggern und stellt diese in einem Dialog zur Auswahl, wie er in Abbildung 1 zu sehen ist.

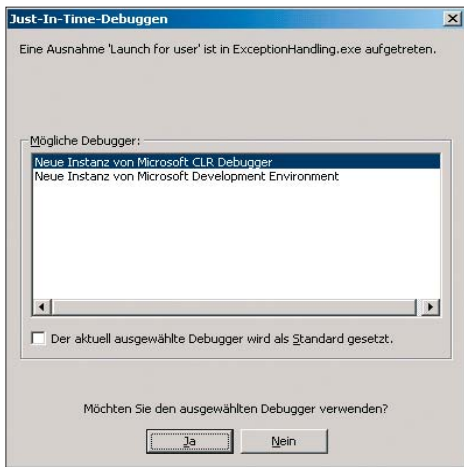


Abbildung 1 | Zur Laufzeit kann aus der Anwendung heraus ein Debugger gestartet werden.

Der gewählte Debugger klinkt sich direkt in die laufende Anwendung ein. Handelt es sich um eine Debug-Version, kann der Quellcode inspiziert werden. Wurde die Anwendung hingegen im Release-Modus kompiliert, ist nur der Blick auf den kompilierten IL- beziehungsweise auf den JIT-kompilierten Code möglich.

Über die Eigenschaft *IsAttached* der Klasse *Debugger* lässt sich zur Laufzeit prüfen, ob sich ein Debugger in die Anwendung eingeklinkt hat. Ist dies der Fall, kann mit der Methode *Break* die Ausführung unterbrochen und die Kontrolle an den Debugger übergeben werden. Eine Ausgabe von Debug-Informationen ist über die Methode *Log* möglich. Mit der Methode *IsLogging* kann man prüfen, ob das Logging im Debugger aktiviert ist. Folgendes Beispiel verdeutlicht die Verwendung.

```
if (Debugger.IsAttached) {
    If (Debugger.IsLogging())
        Debugger.Log(0, "Kategorie", "Meine Meldung.");
}
```

In der Methode *Log* werden vor der eigentlichen Meldung die Dringlichkeitsstufe und die Kategorie bestimmt.

### Erstellen einer generischen Exception-Klasse

Um möglichst einheitlich die oben genannten Informationen zu ermitteln und bereitzustellen, bieten sich zwei Vorgehensweisen an:

- Erstellen einer Basis-*Exception*-Klasse, die alle benötigten Informationen ermittelt und von der alle Anwendungs-*Exceptions* abgeleitet werden;
- Implementieren einer generischen *Exception*-Informationsklasse, die alle Informationen bereitstellt.

Der erste Ansatz hat den Nachteil, dass alle Ausnahmen, die nicht von der Basis-*Exception*-Klasse abgeleitet sind, nicht über die allgemeinen Informationen verfügen. Zwar ließe sich dies durch Abfangen aller Systemausnahmen und deren Umwandlung in Anwendungsausnahmen beheben. Die Ausnahmen enthielten dann aber keine aussagekräftigen Informationen mehr.

Der zweite Ansatz ist flexibler. Hier wird eine generische Klasse definiert, die im Konstruktor ein Objekt vom Typ

Tabelle 1 | Eigenschaften der *ExceptionInfo*-Klasse.

Eigenschaften	Beschreibung
Exception	Ausnahme, für die Informationen ermittelt werden sollen
AppDomainName	Name der AppDomain, in der die Ausnahme aufgetreten ist
AssemblyName	Name der aufrufenden Assembly
ThreadId	ID des Threads
ThreadUser	Thread-Benutzer
ProductName	Titel der Anwendung
ProductVersion	Version der Anwendung
ExecutablePath	Ausführungsverzeichnis der Anwendung
CompanyName	Firmenname
OperatingSystem	Informationen über das installierte Betriebssystem
FrameworkVersion	Versionsinformation des installierten .NET Framework
WorkingSet	Umfang des physischen Speichers, der dem aktiven Prozess zugewiesen wurde

Tabelle 2 | Methoden der *ExceptionInfo*-Klasse.

Methode	Beschreibung
Save	Serialisiert das <i>ExceptionInfo</i> -Objekt unter dem angegebenen Dateinamen
Open	Deserialisiert das <i>ExceptionInfo</i> -Objekt vom angegebenen Dateinamen
GetMethodName	Methodenname
GetClassName	Klassenname
GetFileName	Dateiname
GetFileColumnNumber	Spaltenposition des Fehlers
GetFileLineNumber	Zeilenposition des Fehlers
GetILOffset	IL-Offset im Code aus der Aufrufkette
GetNativeOffset	Nativer Offset im Code
GetStackTrace	Aufrufkette der aktuellen Ausnahme
GetClassNameFromStack	Extrahiert die Klassennamen aus einem StackTrace-String.
GetMethodNameFromStack	Extrahiert den Methodennamen aus einem StackTrace-String.
GetFileNameFromStack	Extrahiert den Dateinamen aus einem StackTrace-String.
GetLineNumberFromStack	Extrahiert die Zeilennummer aus einem StackTrace-String.

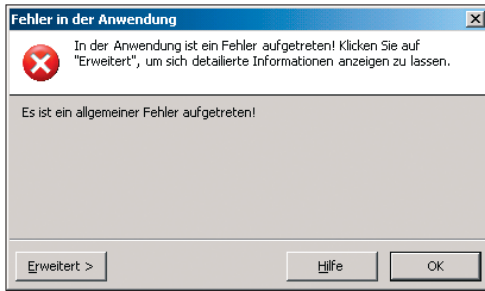


Abbildung 2 | Der generische Fehlerdialog kann alle Arten von Ausnahmen darstellen.

*System.Exception* entgegennimmt, alle benötigten Informationen sammelt und über Eigenschaften zur Verfügung stellt. Über die Methoden *Open* und *Save* kann das Objekt serialisiert beziehungsweise deserialisiert werden, was eine einfache Weitergabe der Informationen ermöglicht.

Zusätzlich stellt die Klasse einige Methoden zur Analyse der Ausnahme bereit, die zum Beispiel in einem Fehlerdialog von Nutzen sein können. Als Beispiel für diese Vorgehensweise dient die Klasse *ExceptionInfo*, die Teil des Beispielprojekts *ExceptionHandling* ist, das sich auf der Heft-CD befindet.

Tabelle 1 führt alle Eigenschaften und Tabelle 2 alle Methoden der Klasse *ExceptionInfo* auf.

### Erstellen eines generischen Fehlerdialogs

Um eine einheitliche Fehlerdarstellung und -Behandlung zu gewährleisten bietet es sich an, einen generischen Fehlerdialog zu erstellen, der im Falle eines Fehlers aufgerufen wird. Einen solchen Dialog sehen Sie in Abbildung 2. Er visualisiert die Informationen des *ExceptionInfo*-Objekts, das ihm übergeben wird.

Nach dem Aufruf erscheint im oberen Teil des Fensters eine allgemeine Meldung, die den Benutzer darauf hinweist, dass ein Fehler aufgetreten ist. Darunter wird der eigentliche Fehlertext ausgegeben. Ist der Text länger als der Dialog, wird automatisch eine vertikale Bildlaufleiste eingeblendet. Durch Anklicken der *Erweitert*-Schaltfläche expandiert der Dialog und zeigt Detailinformationen über den Fehler an (siehe Abbildung 3).



Abbildung 3 | In expandierter Form bietet der Fehlerdialog detaillierte Informationen.



Abbildung 4 | Anzeige aller Exception-Eigenschaften im PropertyGrid.

Das Feld *Ausnahmen* stellt einen Treeview dar, der die Ausnahme mit allen enthaltenen inneren Ausnahmen anzeigt. Durch Klicken auf eine Ausnahme ändern sich die Inhalte der entsprechenden Informationsfelder. Auf der Karteikarte *Aufrufkette* kann die Aufrufkette der gewählten Ausnahme analysiert werden. Die Karteikarte *Anwendung* zeigt allgemeine Informationen zur Anwendung, wie Anwendungsname, Version und weitere, an.

Auf der letzten Karteikarte (siehe Abbildung 4) werden alle Eigenschaften der Ausnahme in einem *PropertyGrid*-Steuerelement angezeigt. Diese Darstellung ist besonders dann sinnvoll, wenn die Ausnahme neben den allgemeinen Informationen auch eigene Eigenschaften implementiert hat.

### Aufruf des Fehlerdialogs

Um den Fehlerdialog anzuzeigen, muss zunächst eine Instanz der Klasse *ExceptionInfo* erstellt und ihr muss im Konstruktor das entsprechende *Exception*-Objekt übergeben werden. Im Anschluss kann eine Instanz von *frmExceptionInfo* erzeugt und im Konstruktor das zuvor erstellte *ExceptionInfo*-Objekt übergeben werden.

```
try {
    Calculate(4, 0);
}
catch (Exception ex) {
    ExceptionInfo exceptionInfo = new ExceptionInfo(ex);
    frmExceptionInfo exceptionDialog =
        new frmExceptionInfo(exceptionInfo);
    exceptionDialog.ShowDialog();
}
```

Hier wird die Methode *Calculate* aufgerufen, die einen Fehler erzeugt. Dieser wird im *Catch*-Block abgefangen und an eine neue Instanz von *ExceptionInfo* weitergeleitet. Diese wird anschließend an eine neue Instanz von *frmExceptionDialog* übergeben. Schließlich wird der Dialog mit *ShowDialog* angezeigt.

### Globale Fehlerbehandlung

Neben der normalen Fehlerbehandlung sollte eine Anwendung zusätzlich über einen globalen Error-Handler verfügen, der alle

unbehandelten Ausnahmen auswertet. Dazu könnte zum Beispiel ein *try/catch*-Block um den Aufruf von *Application.Run* in der *Main*-Methode der Anwendung eingefügt werden, der im Falle einer unbehandelten Ausnahme eine Fehlerbehandlung im *catch*-Block durchführt.

```
try {
    Application.Run(new frmMain());
}
catch (Exception ex) {
    // ... hier werden globale Fehler behandelt
}
```

Diese Vorgehensweise hat jedoch einen entscheidenden Nachteil: Im Falle einer *OutOfMemory*-Ausnahme wird der *catch*-Block nicht mehr ausgeführt!

Für das Abfangen von unbehandelten Ausnahmen bietet die *Application*-Klasse das *ThreadException*-Ereignis. Wenn hierfür ein entsprechender Handler implementiert wird, wird er beim Auftreten einer unbehandelten Ausnahme aufgerufen.

Nachdem der Namespace *System.Threading* eingebunden wurde, kann der Event-Handler in der *Main*-Methode der Anwendung registriert werden:

```
Application.ThreadException += new
ThreadExceptionHandler(GlobalExceptionHandler);
```

Nun wird eine entsprechende Methode implementiert:

```
public static void GlobalExceptionHandler(object sender,
ThreadExceptionEventArgs e)
{
    // ... hier werden globale Fehler behandelt
}
```

Hier könnte der Fehlerdialog aufgerufen werden. Für unbehandelte Ausnahmen bietet *ExceptionInfo* einige zusätzliche Eigenschaften. Da sich die Anwendung nach dem Auftreten eines unbehandelten Fehlers eventuell in einem inkonsistenten Zustand befindet, sollte sie nach der Anzeige des Fehlerdialogs beendet werden. Dies kann dem Benutzer zum Beispiel im Informationstext des Dialogs mitgeteilt werden. Die Klasse *ExceptionInfo* bietet hierfür die Eigenschaft *TitleText*. Außerdem sollte die Eigenschaft *UnhandledException* auf *true* gesetzt werden, womit der Dialog über den Zustand informiert wird.

Um dem Benutzer den Neustart der Anwendung nach dem Fehler zu ersparen, bietet der Dialog über das Kontrollkästchen *Anwendung neu starten* an, automatisch einen Neustart auszuführen. Die Eigenschaft *ShowAppRestartCheckBox* steuert, ob dieses Kontrollkästchen angezeigt wird oder nicht.

Da die Klasse *ExceptionInfo* serialisierbar ist, bietet es sich an, die Fehlerinformationen nach dem Auftreten einer unbehandelten Ausnahme an den Support zu senden. Durch das Setzen der Eigenschaft *ShowReportErrorLabel* auf *true* wird dem Benutzer ein *LinkLabel*-Steuerelement mit dem Text *Diesen Fehler an <Firmenname> melden* angezeigt. Im Event-Handler des *LinkLabels* könnte nun das *ExceptionInfo*-Objekt über die *Save*-Methode serialisiert und zum Beispiel per Mail versendet werden.

Über die Eigenschaft *ShowDebugButton* lässt sich bestimmen, ob die *Debug*-Schaltfläche im Dialog erscheint, der für das

Starten eines Debuggers zuständig ist. Der Aufrufer könnte zum Beispiel diese Schaltfläche nur in der Debug-Version der Anwendung anzeigen. Dies ist mithilfe des *Conditional*-Attributs möglich, wie das folgende Beispiel zeigt.

```
[Conditional("DEBUG")]
private static void ShowDebugButton(frmExceptionInfo exceptionDialog)
{
    exceptionDialog.ShowDebugButton = true;
}
```

Die Methode macht nichts weiter, als die *ShowDebugButton*-Eigenschaft des übergebenen *frmExceptionInfo*-Dialogs auf *true* zu setzen. Durch die Deklaration des *Conditional*-Attributs mit *DEBUG* wird erreicht, dass diese Funktion nur ausgeführt wird, wenn es sich um eine Debug-Version handelt.

Vor dem Anzeigen des Dialogs könnte der Aufrufer diese Methode zum Aktivieren der Debug-Schaltfläche ansteuern. Im Falle einer Release-Version würde dieser Aufrufcode vom Compiler ignoriert.

Über die *Hilfe*-Schaltfläche wird die Hilfedatei der Ausnahme aufgerufen. Die Schaltfläche wird jedoch nur dann angezeigt, wenn die Eigenschaft *HelpLink* der Ausnahme einen Wert enthält und die Eigenschaft *ShowHelpButton* auf *true* gesetzt wurde.

Der Dialog für eine unbehandelte Ausnahme sieht damit genauso aus wie ein Dialog für eine Ausnahme, die regulär abgefangen wurde (siehe Abbildung 3).

### Fehlerprotokollierung

Neben der Darstellung des Fehlers am Bildschirm bietet es sich an, bestimmte Fehler zu protokollieren. Dafür bieten sich zum Beispiel folgende Medien an:

- Textdatei
- Event Log
- Datenbank

Beim Implementieren dieser Protokollierungstypen sieht das .NET Framework die Verwendung von *TraceListener*-Objekten vor. Diese können sowohl in der Debug- als auch in der Release-Version einer Anwendung verwendet werden. Eine Einführung in die Tracing-Funktionalität von .NET finden Sie unter [1].

[1] Jörg Neumann, Tracing in .NET, BasicPro 1/2002, Seite 28 ff.