

# Zentrales Ausnahmemelden

## Der komfortable Notausgang

**Christoph Knabe, Siamak Haschemi**

Damit Ausnahmen nicht in schwer auffindbaren Log-Dateien verschwinden oder gar unterdrückt werden, ist ein zentrales Ausnahmemelden in Verbindung mit konsequenter Ursachenerfassung einzuführen. Es werden für die repräsentativen Oberflächen-Rahmenwerke Swing und Struts Techniken dafür beschrieben. Auch Aspektorientierte Techniken helfen dabei. Konsequenter eingesetzt, erhält man robuste und diagnosestarke Applikationen, die bequem zu programmieren sind.

Die Zentralisierung des Meldens von Ausnahmen bewirkt, dass bereits erkannte Fehler nicht „verlorengehen“ und dass alle Fehlermeldungen in einem einheitlichen, benutzerfreundlichen Format erfolgen. Leider sind die Mittel, Ausnahmen zu melden, in jedem Oberflächenrahmenwerk verschieden und oft nur schwer in der Dokumentation zu finden. Daher beschreiben wir für einige Rahmenwerke, wie man die Ausnahmemeldung zentralisiert, als auch wie dieses ohne Kenntnis der Tiefen eines Rahmenwerks mit AspectJ möglich ist.

Daneben werden übliche Fehler und gute Praktiken in Bezug auf Ausnahmebehandlung besprochen.

## Warum zentrales Ausnahmemelden?

Das saubere Erfassen, Behandeln und Melden von Ausnahmen ist durchaus kompliziert und kann leicht die Hälfte des Quellcodes einnehmen. Es ist bei Programmierern auch unbeliebt, da es nicht direkt zum Funktionsumfang eines Systems beiträgt. Durch konsequente Zentralisierung wird jedoch der Code wartungsfreundlicher, robuster und diagnosestärker. Daneben erhält der Benutzer öfter eine Systemreaktion, die er versteht, selbst wenn das System seinen Auftrag nicht durchführen kann oder will.

## Schlechte Ausnahmebehandlung

Als Grundlage für die weiteren Konzepte wollen wir typische Fehler in der Ausnahmebehandlung zusammenstellen, die auch im Quellcode renommierter Hersteller leider immer noch vorkommen.

- Ausnahme ohne Diagnoseparameter werfen: Nichts ist frustrierender, als z.B. die Meldung *FileNotFoundException* zu erhalten, ohne gesagt zu bekommen, welche Datei denn fehlt. Die Suche nach der fehlenden Datei kann sehr langwierig werden.
- Ausnahme unterdrücken: Ein `try { ... } catch (IrgendwasException e) {}` findet sich leider in vielen Lehrbüchern, wenn es nicht gerade um Ausnahmebehandlung geht, und in manchem Quellcode. Der Programmierer ist zwar momentan die Sorge los, wohin mit der eigentlich unerwarteten Ausnahme. Wenn Sie aber dennoch einmal auftritt, erfährt keiner etwas davon und muss man Detektiv bei den Folgefehlern spielen.
- Ausnahme loggen, ohne Auftraggeber zu informieren: Ein `try { ... } catch (IrgendwasException e) {log.warn(e);}` ist nur eine etwas mildere Form der Ausnahmeunterdrückung. Weder die aufrufende Methode noch der Programmbenutzer erfahren von dem Fehler. Der Benutzer bemerkt das Versagen erst bei Folgefehlern und muss dann mühsam suchen, in welchen Log-Dateien sich relevante Ausnahmen finden könnten.

- Ursachenausnahme unterdrücken: Ein `try { ... } catch(IrgendwasException e) {throw new MyException();}` informiert zwar den Auftraggeber unverzüglich vom Fehlschlag, unterdrückt aber die eigentliche Ursache und führt daher ebenfalls zu langwieriger Fehlersuche.
- Unterschiedliche Meldungswege: Ad-hoc-programmierte Ausnahmemeldungen sind oft nur ein `printStackTrace`. Sie sind nicht benutzerfreundlich, erreichen den Benutzer oft nicht und können Legacy-Ausnahmen (wie die `SQLException`) nicht mit allen Diagnoseinfos wiedergeben.

Aus diesen typischen Fehlern bei der Ausnahmebehandlung leiten wir folgende Strategien ab.

## Strategien zur Ausnahmebehandlung

Als erstes ist zu empfehlen, dass Sie bewährte Lösungen zur Ausnahmebehandlung einsetzen. Leider gibt es wenige quelloffene Rahmenwerke dazu. Die meisten Lösungen sind hausinterne Entwicklungen. Auch das „Multi-Tier Exception Handling Framework“ MulTex [1] des Autors Knabe geht auf derartige Erfahrungen aus der Privatwirtschaft zurück. MulTex wird in den Beispielen dieses Artikels verwendet, auch wenn es nicht im Mittelpunkt steht. Die Strategien sind auch auf andere Frameworks übertragbar.

Daneben gibt es eine Debatte, wie häufig Ausnahmen geworfen werden sollten. Trotz allen Streits um die Performanz von Ausnahmen bleibt unzweifelhaft, dass die automatische Ausnahmebehandlung sehr zur Robustheit von Java-Applikationen beiträgt, dass in Bibliotheken Ausnahmen häufig eingesetzt werden und man daher auf jeden Fall für eine saubere Meldung geworfener Ausnahmen sorgen muss.

## Ausnahme mit Diagnoseparametern und Meldungstext

Nach unserem Verständnis sollte eine Ausnahme, da sie ja an den Benutzer zu melden ist, einen parametrisierten, internationalisierbaren Meldungstext haben. Dies war schon z.B. in den 1980er Jahren in dem Betriebssystem VMS [2] gegeben, ist auf der Java-Plattform aber leider nicht vorbereitet.

MulTex kombiniert dazu die Standardklassen `java.text.MessageFormat` und `java.util.ResourceBundle` und verwendet den Klassennamen jeder Ausnahme als Schlüssel für den zugehörigen Meldungstext. Der Meldungstext in der Entwicklungssprache kann dabei als Javadoc-Kommentar oder als String-Literal in der Ausnahmeklasse definiert werden. Eine typische MulTex-Ausnahme sähe dann z.B. wie folgt aus:

```
/**Phone number {0} contains illegal characters. Allowed are only '{1}'*/
class PhoneNumberExc extends multex.Exc {
    public PhoneNumberExc(String phoneNumber, String allowedCharacters) {
        super(null, phoneNumber, allowedCharacters);
    }
}
```

Die bei Auftreten dieser Ausnahme erscheinende Meldung würde z.B. lauten:

*Phone number 030\$36409775 contains illegal characters. Allowed are only '0123456789 ()/+-'*

## Ursachenerfassung

Sehr häufig will oder kann man eine geprüfte Ausnahme nicht einfach durch Aufnahme in die eigene `throws`-Klausel nach oben weiterreichen (propagieren). Einerseits würden dadurch die `throws`-Klauseln der obersten Methoden sehr lang und wartungsfeindlich oder sehr nichtssagend wie `throws Exception`. Häufig ist es auch beim Überschreiben einer Rahmenwerk-Methode wie z.B. des `Swing-actionPerformed` mit leerer `throws`-Klausel gar nicht möglich, diese zu erweitern.

All dies spricht dafür, nur Vorbedingungsverletzungen durch geprüfte Ausnahmen zu beantworten. Technisches Versagen einer Methode sollte hingegen durch Verpacken jeder unerwartet von unten kommenden Ausnahme in eine ungeprüfte Ausnahme der eigenen Schicht mit zusätzlichen Diagnoseparametern und eigenem Meldungstext propagiert werden. Als Beispiel dafür siehe den Code zum Laden der „Spielpersistenz“ in Klasse *db.Persistence* im Beispielprojekt *excrep* [3]:

```
try {
    final FileInputStream f = new FileInputStream(FILENAME);
    final ObjectInputStream o = new ObjectInputStream(f);
    lastId = o.readLong();
    result = (Set<Client>)o.readObject();
    o.close();
} catch (Exception e) {
    throw new LoadFailure(e, FILENAME);
}
```

Dazu gehört die entsprechende ungeprüfte Ausnahme:

```
/**Failure loading persistence from file {0}*/
public static class LoadFailure extends multex.Failure {
    public LoadFailure(Throwable cause, String filename) {
        super(null, cause, filename);
    }
}
```

Dieses Vorgehen kann zu einer Kette von Ursachenausnahmen führen und wird deshalb auch „Exception Chaining“ genannt. Es ist in MulTex seit 1998 enthalten, in Java seit dem JDK 1.4. MulTex geht noch einen Schritt weiter, indem auch eine *Collection* von Ursachenausnahmen an einen *Failure* übergeben werden kann, was in der Konsequenz zu einem Ursachenbaum führt.

## Zentrales Ausnahmemeldern

Aus obiger Argumentation folgt schon, dass das Melden einer Ausnahme keine triviale Sache ist. Es muss Folgendes leisten.

- Meldungsziel bestimmen: Ausnahmen bei Aktionen, die vom Bediener angestoßen wurden, an diesen melden. Andere z.B. in Dauer-Threads können nur protokolliert werden. Schwergewichtige Ausnahmen sollten eventuell speziell per E-Post an den Systemadministrator gemeldet werden.
- Internationalisierung und Parametrierung der Ausnahmen-Meldungstexte.
- Verfolgen von Ursachenkette oder -baum eventuell mit weitergehender Detaillierung erst auf Mausklick.
- Verfolgen von Legacy-Ursachen wie *getRootCause()* bei der *javax.servlet.ServletException*.
- Anreichern der Ausnahmemeldung um allgemeine Kontextinformation wie z.B. Datum/Uhrzeit, Threadname, Benutzername.

Daher ist eine strikte Zentralisierung des Meldungsvorgangs von Ausnahmen zu empfehlen.

Leider ist es bei den populären Oberflächen-Rahmenwerken nicht einfach, eine zentrale Ausnahmebehandlung einzusetzen, die alle propagierten Ausnahmen meldet. Teils gibt es gar keinen Mechanismus; teils fehlen bei Nutzung des vorbereiteten Mechanismus nötige Kontextinformationen; teils ist er nur schwer zu finden. Daher wollen wir hier für zwei typische Rahmenwerke Anleitungen geben.

## Beispiel-Applikation Kundenverwaltung Logikkern

Alle Beispiele stellen eine Verwaltung von Kunden (Client) dar. Sie benutzen den Logikkern im Paket *lg*. Die Klasse *lg.Client* stellt einen Kunden dar mit den Attributen: *id*, *firstName*, *lastName*, *birthDate* und *phone*. Die Objektverwaltungsoperationen werden über das Interface *lg.Session* kanalisiert.

## Beispiel-Oberfläche Struts

Zunächst sollte eine allgemeine Fehlerseite in *web.xml* konfiguriert werden. Diese übernimmt auch die Detaildarstellung einer Ausnahme, wenn der Benutzer dies wünscht.

```
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/system/errorPage.jsp</location>
</error-page>
```

Die auf Struts 1.3.8 basierende Oberfläche ist im Paket *struts\_ui* enthalten. Das zentrale Ausnahmemeldungen bei Struts wird durch folgende Festlegung in der Datei *struts-config.xml* vereinbart:

```
<global-exceptions>
  <exception
    type="java.lang.Exception"
    handler="struts_ui.CentralExceptionHandler"
    key="struts_ui.CentralExceptionHandler.inPageErrorMessage"
  />
</global-exceptions>
```

Dies bedeutet, dass alle Ausnahmen, die Instanzen von *java.lang.Exception* sind, dem *CentralExceptionHandler* übergeben werden. Dies sind sowohl geprüfte als auch ungeprüfte Ausnahmen. Nur Instanzen von *java.lang.Error*, z.B. wegen korrupter virtueller Maschine, sind davon ausgeschlossen. Die *key*-Angabe für die Datei *MessageResources.properties* wählt einen Meldungsrahmen aus, in den der hier vorgestellte *CentralExceptionHandler* die der Ausnahme entsprechenden Meldungen als Parameter 0 und den Link zur *errorPage.jsp* als Parameter 1 einsetzt:

```
struts_ui.CentralExceptionHandler.inPageErrorMessage=<div style="color:red"> {0}
<br><a href="{1}"> <b>Details</b></a></div>
```

In jeder Eingabeseite muss *<html:errors/>* stehen, jede *<action>* in *struts-config.xml* sollte mit *input="eingabeseite.jsp"* angeben, auf welcher Seite eine eventuelle Fehlermeldung angezeigt werden soll.

Der *CentralExceptionHandler* muss die Struts-Klasse *ExceptionHandler* erweitern und die Methode *execute* überschreiben. Diese stellt neben den üblichen *execute*-Parametern wie *mapping* und *request* auch die zu meldende Ausnahme *ex* zur Verfügung. Siehe Listing 1.

### Listing 1

```
@Override public ActionForward execute(final Exception ex, final ExceptionConfig ae,
final ActionMapping mapping, final ActionForm form, final HttpServletRequest
request, final HttpServletResponse response) throws ServletException
{
  //Store exception for global error page:
  request.getSession().setAttribute(EXCEPTION_KEY, ex);
  //Store exception messages for Struts display on input page:
  final ActionForward globalErrorForward = mapping.findForward("error");
  final String globalErrorPage = globalErrorForward.getPath();
  final ResourceBundle bundle = getRequestLocaleBundle(BASE_NAME, request);
  final ActionMessages errors = new ActionErrors();
  errors.add( ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
    ae.getKey(),
    getMessagesAsHtml(ex, bundle), globalErrorPage
  ));
  serviceAction.mySaveErrors(request, errors);
  //Forward to messages display:
  final ActionForward inputForward = mapping.getInputForward();
  return inputForward!=null ? inputForward : globalErrorForward;
}
```

### Ende Listing 1

Hierdurch wird erreicht, dass jede von den unteren Schichten kommende Ausnahme, als auch jede nur in den oberen Schichten geworfene, zunächst auf der Eingabeseite und bei Anklicken des *Details*-Links ausführlich auf der *errorPage.jsp* gemeldet wird. So z.B. bei unerwarteter Schreibsperrung der Persistenzdatei.

>>knabe\_ausnahmemeldung\_1.bmp<<

## Edit Client

- Cannot save client Müller  
+Cause: Failure committing persistence into file Persistence.ser  
++Cause: java.io.FileNotFoundException: Persistence.ser  
(Zugriff verweigert)  
[Details](#)

ID: 7

First Name:

Last Name:

Birth Date:  [yyyy-MM-dd]

Phone Number:

Abb. 1: Ausnahmemeldung mit Ursachenkette auf der Eingabeseite

Wenn nur eine Geschäftsregelverletzung vorliegt, z.B. ungültiges Telefonnummernzeichen, wird die Ursachenkette kürzer ausfallen, z.B.

*Cannot save client Müller*

*+Cause: Phone number (0208=479387 contains illegal characters. Allowed are only '0123456789 ()/+-'*

## Beispiel-Oberfläche Swing

In Swing gibt es im Gegensatz zu Struts keine vorbereitete Lösung zur Zentralisierung der Ausnahmemeldung. Daher müssen wir auf eine einfache Form des Entwurfsmusters *template method* zurückgreifen.

Normalerweise ist es sinnvoll, Buttons und Menüpunkte mit einem Unterklassenobjekt von *javax.swing.AbstractAction* zu erzeugen, welches sowohl den Namen der Aktion als auch die zugehörige Methode *actionPerformed* enthält. Dies sieht fürs Löschen z.B. wie folgt aus:

```
final Action deleteAction = new AbstractAction("Delete"){
    @Override public void actionPerformed(ActionEvent evt){
        try{
            //Code, der das Objekt löscht
        }catch(Exception ex){
            //Meldung der Ausname ex in Kenntnis der verursachenden Komponente:
            reportException(ev.getSource(), ex);
        }
    }
};
final JButton deleteButton = new JButton(deleteAction);
```

Selbst wenn man alle Feinheiten der Ausnahmemeldung in der Methode *reportException* versteckt hat, wird es dabei unzählige, im Hinblick auf ihre try-catch-Struktur redundante, *actionPerformed*-Methoden geben. Um den Aktionen zu erlauben, Ausnahmen zu werfen und um diese dann zentral zu melden, muss die Verwendung der *Swing-AbstractAction* durchgehend durch eine eigene *ExceptionReportingSwingAction* ersetzt werden. Diese enthält als „template method“ eine Methode *actionPerformed*, die die inhaltliche Aktion an die abstrakte Methode *actionPerformedWithThrows* delegiert, aber die Ausnahmebehandlung zentralisiert hat:

```
public abstract class ExceptionReportingSwingAction
extends javax.swing.AbstractAction {
    ...
    public void actionPerformed(final ActionEvent ev) {
        try{
            actionPerformedWithThrows(ev);
        }
```

```

    }catch(Exception ex){
        //Meldung der Ausname ex in Kenntnis der verursachenden Komponente:
        reportException(ev.getSource(), ex);
    }
}

public abstract void actionPerformedWithThrows(ActionEvent ev) throws Exception;
}

```

Durch die Zentralisierung der Ausnahmemeldung ist diese wartungsfreundlich und die Deklaration der *deleteAction* viel einfacher geworden. Eine Beispielmeldung sehen Sie in Abb. 2.

```

final Action deleteAction = new ExceptionReportingSwingAction("Delete"){
    @Override public void actionPerformedWithThrows(ActionEvent evt) throws Exc {
        //Code, der das Objekt löscht
    }
};
final JButton deleteButton = new JButton(deleteAction);

```



>>knabe\_ausnahmemeldung\_2.bmp<<

Abb. 2: Ausnahmemeldung mit Ursachenkette in Swing-Dialog

Dieser Ansatz hat den Vorteil, auf alle Rahmenwerke übertragbar zu sein, die selbst kein API zur Zentralisierung der Ausnahmebehandlung anbieten. Der Nachteil ist, dass man in der Benutzung des Rahmenwerks an sehr vielen Stellen vom Üblichen abweichen muss. Eine nachträgliche Einführung ist daher aufwändig.

Es sei noch auf weitere Lösungen hingewiesen, die mehr einen Notfallcharakter haben, da sie keinen Zugriff auf die verursachende Oberflächenkomponente haben. Die inoffizielle System Property *sun.awt.exception.handler* erlaubt, einen Behandler für alle bei der Verarbeitung von Swing/AWT-Events auftretenden Ausnahmen einzusetzen [awtExc]. Heinz Kabutz [4] erzeugt in seinem ab Java 1.4 funktionierendem Ansatz das Swing-UI in einer eigenen *ThreadGroup* und überschreibt dabei die Methode *ThreadGroup.uncaughtException*. Seit Java 5 kann man auch mittels *Thread.setDefaultUncaughtExceptionHandler* alle überhaupt auftretenden unbehandelten Ausnahmen melden. Eine oder mehrere dieser Notfalllösungen sollte man zusätzlich zum Action-Ansatz in einer Swing-Applikation einsetzen.

- Dies ist eine Aufzählung. Wenn Sie nummeriert aufzählen m
- öchten, schreiben Sie bitte 1. 2. 3. davor.
- Dies ist eine Aufzählung. Wenn Sie nummeriert aufzählen möchten, schreiben Sie bitte 1. 2. 3. davor.
- Dies ist eine Aufzählung. Wenn Sie nummeriert aufzählen möchten, schreiben Sie bitte 1. 2. 3. davor.

Der eigentliche Artikel wird in der Absatzvorlage „Mengentext“ geschrieben. Der eigentliche Artikel wird in der Absatzvorlage „Mengentext“ geschrieben. Der eigentliche Artikel wird in der Absatzvorlage „Mengentext“ geschrieben. Der eigentliche Artikel wird in der Absatzvorlage „Mengentext“ geschrieben. Der eigentliche Artikel wird in der Absatzvorlage „Mengentext“ geschrieben.

```
Kurzer Quellcode Quellcode
Kurzer Quellcode Quellcode
Kurzer Quellcode Quellcode
//Quellcode bis zu zehn Zeilen kann zwischen dem Mengentext laufen
//Längere Listings bitte in einen Kasten ausgliedern (s.u.)
```

>>autorname\_thema\_1.tif<<

Abb. 1: Hier steht eine Bildunterschrift

### Überschrift Kastentext

Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten.

### Absatzheadline im Kasten

Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Hier steht ein Text für einen separierten Textkasten. Kapitälchen gibt es auch im Textkasten, ebenso Links@Kastentext und Kursive.

- Hier steht ein Text für einen separierten Textkasten.
- Hier steht ein Text für einen separierten Textkasten.
- Hier steht ein Text für einen separierten Textkasten.

### Ende Kastentext

### Listing

```
Lange Quellcodepassage Quellcodepassage
Lange Quellcodepassage Quellcodepassage
Lange Quellcodepassage Quellcodepassage
Lange Quellcodepassage Quellcodepassage
Lange Quellcodepassage Quellcodepassage
Lange Quellcodepassage Quellcodepassage
Lange Quellcodepassage Quellcodepassage
//Bitte im Mengentext auf den entsprechenden Listingkasten verweisen
//Bei mehreren Listingkästen diese durchnummerieren
```

### Ende Listing

Tabellenheadline	Tabellenheadline
------------------	------------------

Hier steht der Tabellentext.	Hier steht der Tabellentext.
Hier steht der Tabellentext.	Hier steht der Tabellentext.
Hier steht der Tabellentext.	Hier steht der Tabellentext.
Hier steht der Tabellentext.	Hier steht der Tabellentext.

**Tabelle 1: Hier steht eine Tabellenunterschrift**

*Hier steht eine kleine Autorinfo, die kursiv gesetzt wird. Hier steht eine kleine Autorinfo, die kursiv gesetzt wird Hier steht eine kleine Autorinfo, die kursiv gesetzt wird Hier steht eine kleine Autorinfo, die kursiv gesetzt wird.*

## **Links & Literatur**

---

[1] MuTEx

[2] VMS

[3] excrep

[awtExc] Dokumentiert im Quellcode der Methode `handleException` von `java.awt.EventDispatchThread`.

[j2ee] Exception management and error tracking in J2EE  
[http://www.theserverside.com/news/thread.tss?thread\\_id=35107](http://www.theserverside.com/news/thread.tss?thread_id=35107)

[4] <http://www.javaspecialists.co.za/archive/Issue081.html> Central Exception Reporting for Swing by `ThreadGroup.uncaughtException`