Zentrales Ausnahmemelden

Der komfortable Notausgang

Publiziert im JavaMagazin 11/2007, Seiten 23-27, unter dem Titel „Ausnahmen bestätigen die Regel“

Christoph Knabe, Siamak Haschemi

Zusammenfassung: 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 Oberflächenrahmenwerke Struts und Swing Vorgehensweisen dafür beschrieben. Auch aspektorientierte Techniken helfen dabei. Konsequent 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 zentral abzufangen, in jedem Oberflächenrahmenwerk verschieden und oft nur schwer in der Dokumentation zu finden. Daher beschreiben wir für typische Rahmenwerke, wie man das Ausnahmemelden 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 man muss 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-programmiertes Ausnahmemelden ist oft nur ein *printStackTrace*. Die Ausgabe ist nicht benutzerfreundlich, erreicht oft nicht den Benutzer und kann 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 und ob geprüfte Ausnahmen geworfen werden sollten [2]. Trotz allen Streits darum 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.

1. Strategie: Ausnahme mit Diagnoseparametern und Meldungstext

Nach unserem Verständnis sollte eine Ausnahme, da sie ja an den Benutzer zu melden ist, einen parametrierten, internationalisierbaren Meldungstext haben. Dies war schon z.B. in den 1980er Jahren in dem Betriebssystem VMS [3] 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, mit Meldungstext und Parametern versehene MulTEx-Ausnahme sähe dann z.B. wie folgt aus:

```
/**Phone number {0} contains illegal characters. Allowed are only ''{1}''*/
class PhoneNumberExc extends multex.Exc {}
```

Die bei Auftreten dieser Ausnahme erscheinende Meldung würde z.B. lauten:
Phone number 030$36409775 contains illegal characters. Allowed are only '0123456789 ()/+-'

2. Strategie: 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 Demo-Persistenz in Klasse *db.Persistence* im Beispielprojekt *excrep* [4]:

```
import static multex.MultexUtil.create;
…
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 create(LoadFailure.class, e, FILENAME);
}
```

Ausnahmeerzeugung

Zu vorigem Beispiel gehört die entsprechende ungeprüfte Ausnahme:

```
/**Failure loading persistence from file {0}*/
public static class LoadFailure extends multex.Failure {}
```

Die statische Methode `MultexUtil.create` erzeugt ein neues `LoadFailure`-Objekt und versieht es mit seiner Ursache und weiteren Parametern. Dadurch lässt sich eine parametrierte Ausnahme ohne eigenen Konstruktor und damit sehr bequem deklarieren.

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 (2002). 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 Ausnahmemelden

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, Request-Id, Benutzername.

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

Leider ist es bei den populären Oberflächenrahmenwerken 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* geleitet.

Beispiel-Oberfläche Struts

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

```
<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 Ausnahmemelden bei Struts wird durch folgende Festlegung in der Datei *struts-config.xml* vereinbart:

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

Dies bedeutet, dass alle Ausnahmen, die Instanzen von *java.lang.Exception* sind, dem *CentralExceptionReporter* 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 *CentralExceptionReporter* die der Ausnahme entsprechenden Meldungen als Parameter 0 und den Link zur *errorPage.jsp* als Parameter 1 einsetzt:

```
struts_ui.CentralExceptionReporter.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.

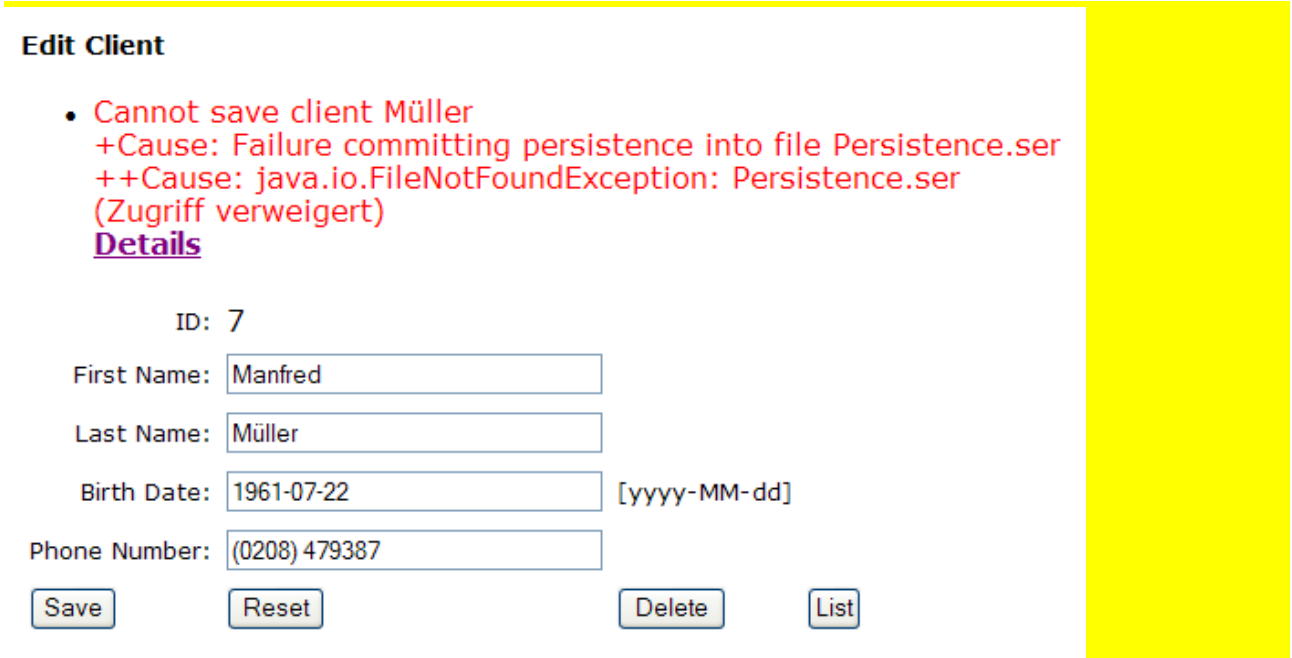
Der *CentralExceptionReporter* muss die Struts-Klasse *ExceptionHandler* erweitern und die Methode *execute* überschreiben. Diese stellt neben den üblichen *execute*-Parametern wie *mapping* und *request* auch die zu meldende Ausnahme *ex* zur Verfügung. Siehe Listing 1.

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 String contextRelativePath = "/system/errorPage.jsp";
  final String absolutePath = request.getContextPath() + contextRelativePath;
  final ResourceBundle bundle = getRequestLocaleBundle(BASE_NAME, request);
  final ActionMessages errors = new ActionErrors();
  errors.add( ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
    ae.getKey(),
    getMessagesAsHtml(ex, bundle), absolutePath
  ));
  serviceAction.mySaveErrors(request, errors);
  //Forward to messages display:
  if(mapping.getInput()==null){return new ActionForward(contextRelativePath);}
  return mapping.getInputForward();
}
```

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. in Abb. 1 bei unerwarteter Schreibsperre der Persistenzdatei.



>>knabe_ausnahmemelden_1.tif<<

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 final 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 ev) throws Exc {
    //Code, der das Objekt löscht
  }
};
```

```
final JButton deleteButton = new Jbutton(deleteAction);
```

>>knabe_ausnahmemelden_2.tif<<

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 mit viel schematischem Schreibaufwand verbunden.

Einfangen unbehandelter Ausnahmen

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 [5]. Heinz Kabutz erzeugt in seinem ab Java 1.4 funktionierenden Ansatz [6] 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 abfangen. Eine oder mehrere dieser Notfalllösungen sollte man zusätzlich zu den beschriebenen Ansätzen aktivieren.

Ende Kastentext

Oberflächenneutrale Lösung mit Aspektorientierung

Die Umsetzung zentralen Ausnahmemeldens hängt stark von dem verwendeten Oberflächenrahmenwerk ab. Im günstigen Fall bietet das Rahmenwerk entsprechende Mechanismen an (z.B. Struts), im ungünstigen Fall aber nicht (z.B. Swing). In diesem Abschnitt wird ein Ansatz vorgestellt, welcher Aspektorientierung verwendet und sich für jedes Rahmenwerk einsetzen lässt.

In Abb. 3 sind die typischen 3 Schichten einer Anwendung dargestellt. Eine Benutzeraktion wird vom Oberflächenrahmenwerk an die Darstellungsschicht der Anwendung weitergereicht. Diese wiederum ruft aus der Logikschicht eine Aktion auf, welche in einer Ausnahme resultiert. Um diese Ausnahme in Form einer anwendungsspezifischen Fehlerseite anzuzeigen, darf die Ausnahme nicht an das Oberflächenrahmenwerk weitergereicht werden. Auch könnte dadurch unter Umständen das Rahmenwerk in einen undefinierten Zustand überführt werden. Stattdessen muss diese Ausnahme abgefangen und wie zuvor beschrieben gemeldet werden. Dieses Verhalten muss für jede Benutzeraktion sichergestellt werden und stellt daher einen Querschnittsbelang dar.

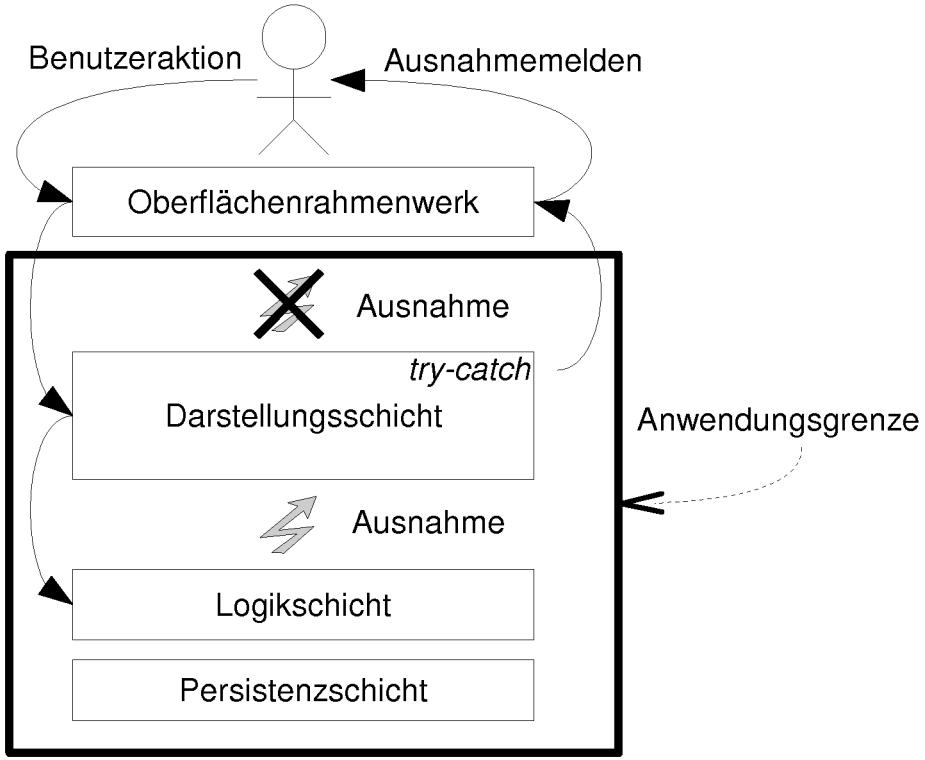


Abb. 3: Benutzeraktion und Ausnahmemelden >>haschemi_ausnahmemelden_3.tif<<

Unser Ansatz ist es, diesen Belang in einem Aspekt zu kapseln und damit sowohl eine Trennung von technischem (zentrales Ausnahmemelden) und fachlichem Code als auch eine Minimierung der Redundanz zu erreichen.

In Abb. 4 kann man sehen, wie sich dieser Aspekt zwischen der Darstellungsschicht und dem Oberflächenrahmenwerk einklinkt, alle Ausnahmen abfängt und sie meldet.

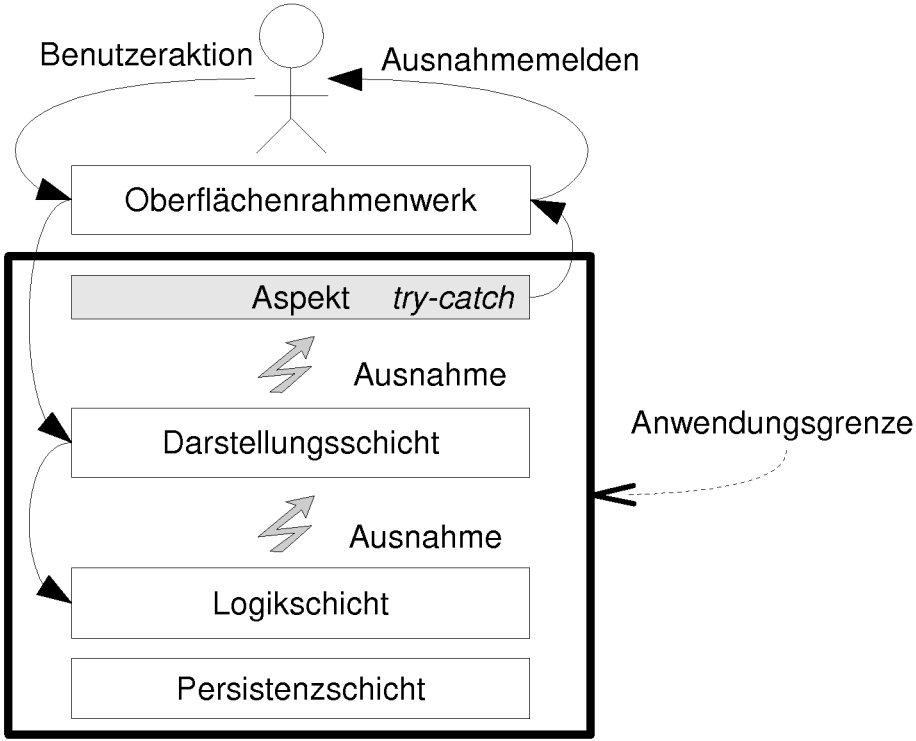


Abb. 4: Benutzeraktion und Ausnahmemelden mit einem Aspekt >>haschemi_ausnahmemelden_4.tif<<

Um diesen Aspekt formulieren zu können, müssen wir alle Methoden in der Darstellungsschicht identifizieren, die vom Oberflächenrahmenwerk aufgerufen werden. Zur Illustration werden wir das zentrale Ausnahmemelden des Swing-Beispiels (mit dem Ansatz des Entwurfsmusters *template method*) durch eine aspektorientierte Alternative ersetzten.

Listing 2

```
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;

public aspect SwingExceptionAspect {
  pointcut pointOfInterest(ActionEvent ev)
    : execution(void AbstractAction+.actionPerformed(ActionEvent))
       && args(ev);

  void around(ActionEvent ev) : pointOfInterest(ev) {
    try {
      proceed(ev);
    } catch (Exception ex) {
      CentralExceptionReporter.reportException(ev, ex);
    }
  }
}
```

Ende Listing 2

In Listing 2 ist der Aspekt, der das zentrale Ausnahmemelden durchführt, aufgelistet. Die verwendete Programmiersprache ist *AspectJ* [7,8,9], eine aspektorienierte Erweiterung von Java.

Mit dem Pointcut *pointOfInterest* werden alle *actionPerformed*-Methoden von Unterklassen der *AbstractAction* angesprochen. In dem *around*-Advice wird jede Ausführung einer *actionPerformed*-Methode in einen *try-catch*-Block eingefasst, der alle Ausnahmen abfängt und meldet.

Somit stellen sich gegenüber dem ersten Ansatz folgende Vorteile ein:

- Die Verwendung der *AbstractAction*-Unterklasse *ExceptionReportingSwingAction* und damit das Vermischen von technischem und fachlichem Code wurde verhindert.
- Bei Bedarf können weitere Listener-Methoden wie *menuSelected* in den Aspekt aufgenommen werden. Der Code in der Darstellungsschicht muss dazu nicht verändert werden.

Die wichtigsten Konzepte von AspectJ

Aspekt

Ein *Aspekt* ist eine erweiterte Klasse zur Umsetzung eines Querschnittsbelangs. Er kann *Pointcuts* und *Advices* enthalten.

Join Point

Ein *Join Point* ist eine spezifische Stelle innerhalb des Ablaufs einer Anwendung, an der ein *Advice* ausgeführt werden kann. Beispiele für *Join Points* sind:

- Methoden/Konstruktor-Aufruf
- Methoden/Konstruktor-Ausführung
- Initialisierung eines Objekts
- Lesen/Schreiben einer Instanzvariablen

Pointcut

Mit Hilfe eines *Pointcuts* kann man das Interesse an einer Menge von *Join Points* in der Anwendung verkünden. Das geschieht über ein Muster, in dem Wildcards verwendet werden können.

Advice

In einem *Advice* wird ausgedrückt, für welche *Pointcuts* welche Zusatzfunktionalität anzuwenden ist.

Ende Kastentext

Fazit

- Die Zentralisierung des Ausnahmemeldens macht Anwendungen robust und diagnosestark bei deutlich erhöhter Bequemlichkeit für die Anwendungsprogrammierer.
- Dieses Ziel ist mit AspectJ auch ohne tiefere Kenntnise des Oberflächenrahmenwerks erreichbar.
- Mit AspectJ kommt man dabei auch ohne spezielle Richtlinien für die Programmierung von Benutzeraktionen aus.

Christoph Knabe (http://public.beuth-hochschule.de/~knabe/) *ist Professor für Softwaretechnik/Programmierung an der Beuth-Hochschule für Technik Berlin. Zuvor war er Software-Entwickler bei der PSI GmbH in Berlin. Seine Interessen gelten der objekt-funktionalen und aspektorientierten Programmierung,, Ausnahmebehandlung, Qualitätssicherung und Performance-Optimierung.*

Dipl. Inf. (FH) Siamak Haschemi (siamak.haschemi@sdm.de) hat das Studium der Medieninformatik abgeschlossen und war 2007 Softwareingenieur für die Entwicklung von individuellen Softwarelösungen bei der sd&m AG – software design & management Berlin. Seine Schwerpunkte umfassten Enterprise Java (J2EE), Aspektorientierung, Modellgetriebene Softwareentwicklung (MDSD) und Rich-Internet-Applications (RIAs).

Links & Literatur

[1] MulTEx: http://public.beuth-hochschule.de/~knabe/java/multex/

[2] Exception management and error tracking in J2EE http://www.javaworld.com/article/2071961/java-web-development/exception-management-and-error-tracking-in-j2ee.html

[3] VMS: http://de.wikipedia.org/wiki/Virtual_Memory_System

[4] Central Exception Reporting sample project: https://www.assembla.com/code/excrep/git/nodes

[5] Dokumentiert im Javadoc-Text der privaten Methode *handleException* von *java.awt.EventDispatchThread*.

[6] Heinz M. Kabutz: Catching Exceptions in GUI Code: http://www.javaspecialists.co.za/archive/Issue081.html

[7] AspectJ: www.eclipse.org/aspectj

[8] Ramnivas Laddad: AspectJ in Action. Practical Aspect-Oriented Programming (Paperback). Manning-Verlag, 2003, ISBN 1-930-11093-6

[9] Oliver Böhm: Aspektorientierte Programmierung mit AspectJ 5. Einsteigen in AspectJ und AOP. dpunkt.verlag, 2005.