Ein Framework zur Ausnahmebehandlung in mehrschichtigen Softwaresystemen

Vortrag auf den NET.OBJECT DAYS 2000 vom 9.-12.10.00 in Erfurt

Christoph Knabe

Technische Fachhochschule Berlin (University of Applied Sciences), FB VI,
Luxemburger Str. 10, D-13353 Berlin, Deutschland
http://public.beuth-hochschule.de/~knabe/
(Web-Adresse aktualisiert 2010-06-11)

Zusammenfassung. Es wird ein Framework vorgestellt, das die bequeme Erstellung von diagnosestarken fehlertoleranten Applikationen unterstützt, die in mehrere Schichten gegliedert sind. Das Framework kombiniert die neue Technik der Ursachenverkettung mit Automatischer Ausnahmebehandlung, Stack Traces, Ausnahmeparametern und Meldungstextverknüpfung. Die Realisierbarkeit in Java, C++ und Ada wird betrachtet. Erfahrungen im industriellen Bereich werden berichtet.

1 Einleitung

Das Konzept der Automatischen Ausnahmebehandlung, in seinen Anfängen betriebssystemnah angesiedelt, setzt sich zunehmend auch in höheren Programmiersprachen durch, die für die Anwendungsentwicklung verwendet werden. So in Ada 1983, Eiffel 1988, C++ 1990 und Java 1996. Für den Einsatz Automatischer Ausnahmebehandlung wird eine „best practice“ in Form des Frameworks [MulTEx] vorgestellt. Dieses ermöglicht es, fehlertolerante und diagnosestarke Applikationen arbeitsteilig und bequem zu entwickeln. Das Framework basiert auf den Erfahrungen des Autors als industrieller Software-Entwickler und Berater.

2 Qualitätskriterien für Produkt und Prozeß

Ein gutes Softwareprodukt sollte gemäß [Balzert82] u.a. die Qualitätskriterien der *Fehlertoleranz* und *Selbsterklärung* erfüllen. In [Meyer90] wird eine arbeitsteilige Entwicklung als *programming by contract* gefordert. Und spätestens seit ISO 9000 wird auch der Qualität des Entwicklungsprozesses mehr Gewicht beigemessen. Denn es kommt in der von Termindruck geprägten Praxis entscheidend darauf an, wie *bequem* es den Produktentwicklern gemacht wird, die produktbezogenen Qualitätskriterien zu erfüllen.

Die angestrebte *Fehlertoleranz* eines Softwaresystems wird durch die schon mit [Ada83] verbreitete Automatik des Abbruchs bei unbehandelter Ausnahme gefördert, da sie eine Fehlerverschleppung äußerst mühsam macht. C++ und Java unterstützen diese Automatik gleichermaßen.

Der arbeitsteilige *programming by contract*-Stil setzt voraus, daß ein Dienst vollständig spezifiziert ist, d.h. neben dem Normaleffekt auch mit allen auslösbaren Ausnahmen.

Bsp. in Java: `String readLine() throws EndOfFile;`

Die Selbsterklärung im Fehlerfall (*Diagnosestärke*) ist das Hauptziel des vorzustellenden Frameworks. Ein damit erstelltes Produkt klärt den Benutzer kurz, aber aussagekräftig über Fehlerfälle auf und stellt dem Support vollständige Diagnoseinformationen zur Verfügung.

Ein weiteres Ziel ist die möglichst *bequeme Benutzbarkeit* des Frameworks für die Anwendungsprogrammierer.

3 Die Diagnosekonzepte

Ein Softwaresystem fehlertolerant und diagnosestark zu machen, erfordert große Anstrengungen, wie z.B. die Bereitstellung einer spezialisierten Softwareproduktionsumgebung wie CeDRE [Knabe90]. Jedoch sind die in [Knabe87] und [Knabe89] beschriebenen Erfahrungen mit diagnosestarker Ausnahmebehandlung in der industriellen Softwareproduktion derart positiv, daß man versuchen sollte, sie auf die heutigen Plattformen zu übertragen und weiter zu entwickeln. Daher sollen hier zunächst die einzusetzenden Diagnosekonzepte erläutert werden.

3.1 Das Konzept der Ursachenkette

Ein typisches Beispiel einer Mehrschichtenarchitektur ist die Trennung in Benutzeroberfläche, Funktionalität und Datenhaltung + Dienste. Als Kürzel und Paketnamen dafür verwenden wir **`b`**, **`f`** und **`x`**.

Benutzeroberfläche	**`b`**
Funktionalität	**`f`**
Datenhaltung + Dienste	**`x`**

Die Operationen werden zu den oberen Schichten hin immer abstrakter, dementsprechend auch die Ausnahmen, die ihr Versagen kennzeichnen.

Als Beispielapplikation dient uns das in Java geschriebene Prüfprogramm für die Mainboards von umweltanalytischen Monitoren [LAR]. Sie nimmt über die serielle Schnittstelle Kontakt mit dem Monitor-Mainboard auf und examiniert es dann.

Auf der Benutzeroberfläche wählt man dafür zunächst den Menüpunkt `connect`, welcher diesen Auftrag an die **`f`**-Schicht weitergibt. Die schichtadäquate Ausnahme heißt dann `Connect-Failure` (≡ Meldung "Cannot connect to the monitor mainboard to be tested").

Solch eine Meldung wäre jedoch für die Fehlerlokalisierung absolut unzureichend. Es gibt zu viele Möglichkeiten, warum eine Verbindung nicht zustande kommen könnte, z. B.:

- Serielle Schnittstelle ist nicht existent / falsch konfiguriert
- Fehler beim Senden der Initialisierungsbotschaft
- Fehler beim Empfangen der Botschaftsquittung
- Ressourcenmangel

In den unteren Schichten ist die jeweilige Ursache genau bekannt und es kommt darauf an, diese gezielt zu erfassen, zu sammeln und zugänglich zu machen.

Wir greifen den Fall heraus, daß die serielle Schnittstelle (engl. *serial port*) nicht existiert und geben in den verschiedenen Schichten jeweils die wichtigste Operation und die von ihr ausgelöste Ausnahme an. Zu den 3 Schichten unseres Softwaresystems kommt hier noch die verwendete Plattform [Comm00] hinzu.

Paket.Klasse	**Operation**	**Schichtadäquate Ausnahme**
b.MmbCheck	`handleConnect` ↓	*keine (Meldungsausgabe)*
f.MmbCom	`connect` ↓	`ConnectFailure` ↑
x.SerialPort	`open` ↓	`OpenFailure` ↑
javax.comm. CommPortIdentifier	`getPortIdentifier`	`NoSuchPortException` ↑

Das neue Konzept der *Ursachenkette* besagt, daß bei Meldung einer Ausnahme der oberen Schichten auch die Kette der verursachenden Ausnahmen mit gemeldet wird. Um das zu ermöglichen, muß jede Ausnahme eine Referenz `cause` auf ihre verursachende Ausnahme enthalten.

Eine *einstufige Ursachererfassung* gab es schon bei CeDRE [Knabe90]. In der Java-Welt gab es auch schon im Einzelfall (z. B. `java.rmi.RemoteException`) die Kapselung einer verursachenden Ausnahme in die ausgelöste Ausnahme. Bei Einsatz dieses Frameworks ist dies jedoch eine Strategie, die auf alle indirekt verursachten Ausnahmen in einem Softwaresystem angewendet wird.

3.2 Weitere Diagnoseinformationen

Stack-Trace
Für die Fehlersuche durch den Programmierer benötigt man nicht nur die Ursachenkette (die ist auch für den Benutzer interessant), sondern auch die exakten Ortsangaben über die Aufrufhierarchie beim Auftreten des Fehlers unter Angabe von jeweils Klasse, Operation und Zeilennummer im Quelltext.

Ausnahmeparameter
Es sollen alle dem Programmierer bekannten Informationen, die im Zusammenhang mit dem Versagen einer Operation stehen, als Ausnahmeparameter erfaßt werden. Weitere Log-Dateien oder Debug-Level hält der Autor dann für überflüssig. Die typischerweise darin stehenden Informationen der unteren Schichten finden sich hier als Parameter in den Ausnahmen der unteren Schichten.

So ist z. B. die Ausnahme `x.SerialPort.OpenFailed` parametriert mit dem Namen der betroffenen seriellen Schnittstelle sowie den Kommunikationseinstellungen.

Meldungstextverknüpfung und Internationalisierung
Der Benutzer erwartet Fehlermeldungen als Fließtext mit eingestreuten Meldungsparametern, **Bsp**.:

Datei „protokoll.txt“ konnte nicht nach „A:“ kopiert werden.

Mindestens die Ausnahmen der oberen Schichten, die an den Benutzer gerichtet sind, sollten daher mit einem Meldungstext verknüpft werden. Sowohl die Meldungstexte als auch die Reihenfolge und Darstellungsformate der Meldungsparameter sollten internationalisierbar sein.

4 Benutzung des Frameworks

Das vorgestellte Framework besteht aus den Ausnahme-Basisklassen `Exc` und `Failure` sowie den Dienstklassen `Msg` und `MsgText`.

Originär in einer Operation erkannte Fehlerfälle werden als Ausnahmen von `Exc` abgeleitet und folgen dem Namensschema *Problem*`Exc`. Dadurch verfügt jede *Problem*`Exc` über ihre Ausnahmeparameter `params`.

Fehlerfälle im Ablauf einer Operation aufgrund unerwarteter Ausnahmen aus der nächsttieferen Schicht werden als Ausnahmen von `Failure` abgeleitet und folgen dem Namensschema *Operation*`Failure`. Dadurch verfügt jede *Operation*`Failure`-Ausnahme über ihre verursachende Ausnahme `cause` und ihre Ausnahmeparameter `params`. Die `Failure`-Ausnahmen sind eine Weiterentwicklung der `failure`-Ausnahme von CLU [Black83].

Die Klasse `Msg` ist zuständig für die Ausgabe einer Ausnahmemeldung mit Ursachenkette. Dazu bedient sie sich der Klassen

- `MsgText` für die Bereitstellung (mittels `ResourceBundle`) und internationalisierte Parametrierung (mittels `MessageFormat`) der Meldungstexte
- `java.awt.Dialog` für die Ausgabe.

In untenstehender Fig. 1 ist dieses Framework als UML-Klassendiagramm dargestellt.

4.1 Erfassung von Ursache und Parametern einer Ausnahme

Die abstrakte Klasse `Failure` besitzt einen Konstruktor

```
Failure(Exception cause, Object[] params)
```

Beim Erzeugen eines Ausnahmeobjekts, um eine `Failure`-Ausnahme auszulösen, kann man also die verursachende Ausnahme an `cause` sowie die weiteren Diagnoseinformationen an den polymorphen `Object`-Array `params` übergeben.

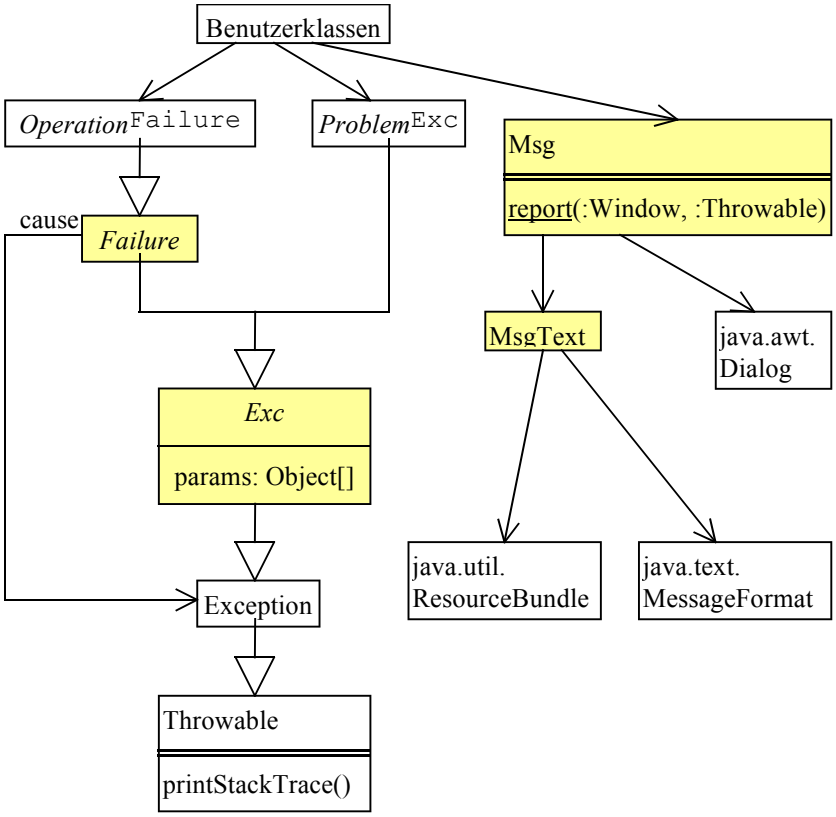

Fig. 1. Das Ausnahmebehandlungs-Framework in seinem Kontext (UML).

Ausnahmen deklarieren

Eine *Operation*`Failure`–Ausnahme mit Ursache und Parametern deklariert man durch Ableitung von der Klasse `Failure`. Dabei sollte man für ein bequemeres Auslösen der Ausnahme und zwecks besserer Dokumentation ihrer Parameter diese jeweils einzeln typsicher im Konstruktorkopf deklarieren und erst im Konstruktorrumpf in einen neu zu erzeugenden polymorphen `Object`-Array eintragen.

Als Beispiel sei die Ausnahme `x.SerialPort.OpenFailure` angegeben, die besagt, daß das Öffnen der seriellen Schnittstelle fehlschlug. Diese Ausnahme gehört aufgrund ihrer umfangreichen Parametrierung (Ursache, Portname und weitere 4 Kommunikationsparameter) zu den überdurchschnittlich komplizierten.

```
class OpenFailure extends Failure {
  public OpenFailure(
    Exception cause, String portName,
    int baudRate, int databits, int stopbits, int parity
  ){
    super(cause, new Object[]{portName,
      ""+baudRate+','+databits+','+stopbits+','+parity
    });
```

```
    }
  }
```

Erfassen der Ursache einer Ausnahme

Wir betrachten die Operation `x.SerialPort.open`. Sie soll die serielle Schnittstelle namens `portName` mittels `javax.comm.SerialPort` öffnen, sie als `sp` zugreifbar machen, die Kommunikationsparameter einstellen und als `is` sowie `os` je einen I/O-Stream für die Datenübertragung zur Verfügung stellen.

In den einzelnen Schritten dieser Operation können folgende Ausnahmen auftreten: `NoSuchPortException`, `PortInUseException`, `UnsupportedCommOperationException` sowie `IOException`. Die entsprechenden Stellen sind im abgedruckten Code jeweils dahinter in einem `//`-Kommentar angegeben.

Um diese systemnahen Ausnahmen zur schichtadäquaten Ausnahme `x.SerialPort.OpenFailure` zusammenzufassen, wird der gesamte verarbeitende Teil des Operationsrumpfs in einen Ausnahmebehandler eingefaßt. Darin wird jede von unten kommende Ausnahme in die Ausnahme `x.SerialPort.OpenFailure` umgedeutet und dieser als Ursache mitgegeben. Darüberhinaus werden der Schnittstellenname `portName` und weitere Kommunikationsparameter als Ausnahmeparameter erfaßt.

```
public void open( String portName,
  int baudRate, int databits, int stopbits, int parity
) throws NameExc, OpenFailure
{ if(!portName.startsWith("COM")){
    throw new NameExc(portName);
  }
try {
  this.portName = portName;
  final javax.comm.CommPortIdentifier portId
  = CommPortIdentifier.getPortIdentifier(portName);
    //NoSuchPortException
  sp = (javax.comm.SerialPort)portId.open(null,0);
    //PortInUseException
  sp.setSerialPortParams(
    baudRate, databits, stopbits, parity
  ); //UnsupportedCommOperationException
  os = sp.getOutputStream(); //IOException
  is = sp.getInputStream();  //IOException
} catch (Exception e) {
  ..... //free resources
  //redefine exception:
  throw new OpenFailure(e,
    portName, baudRate, databits, stopbits, parity
  );

}} //catch,open
```

Dieses Verfahren ist typisch in der **f**-Schicht oder **x**-Schicht. Eine dortige Operation löst an Ausnahmen typischerweise mehrere (originäre) *Problem*`Exc`, aber nur eine (indirekt verursachte) *Operation*`Failure` aus. Der Operationsrumpf hat meistens den Aufbau:

```
if(Vorbedingung nicht erfüllt){
  throw new Problem Exc(parameter ...);
}
...
try {
  Eigentlicher Algorithmus mit Aufruf von Diensten
```

```
} catch(Exception e){
  throw new OperationFailure(e, parameter … );
}
```

4.2 Verknüpfung der Ausnahmen mit internationalisierten Meldungstexten

Alle Ausnahmen, die der Benutzer sinnvoll interpretieren könnte, sollten mit einem Meldungstext verknüpft werden.

In einer lokalisierbaren `ResourceBundle`-Datei gemäß [MF99], z. B. `MsgText_en.properties` wird für jede Ausnahme das zugehörige Meldungstextmuster (hier auf `en`glisch) definiert:

```
x.SerialPort$OpenFailure = Cannot open the \
  serial port "{0}" with communication parameters "{1}"
```

Dabei entspricht der Platzhalter `{`*i*`}` dem Element *i* des `Object`-Arrays mit den Ausnahmeparametern.

4.3 Arbeitsteilung und Benutzeroberfläche

Alle in der Funktionalitätsschicht erkannten Fehler sollten dem Aufrufer als abfangbare Ausnahmen geliefert werden. Auf diese Weise sind die Funktionalitätsklassen in verschiedensten Umgebungen einsetzbar. Als Beispiel diene die Operation `f.MmbCom.connect`, die die Verbindung zum Testling aufnimmt.

```
void connect() throws ConnectFailure;
```

Zu dieser Ausnahme gehört der Meldungstext

```
Cannot connect to the monitor mainboard to be tested
```

In der Oberflächenschicht wird man die meisten der bei Aufruf von Operationen der Funktionalitätsschicht ausgelösten Ausnahmen einfach melden wollen. Dazu benutzt man den Dienst

```
Msg.report(Window ownerWindow, Throwable throwable)
```

der die *Ausnahme* `throwable` und ihre *Ursachenkette* im Fließtextformat in einem modalen Dialog anzeigt (das `ownerWindow` wird solange blockiert). Siehe **Fig**. 2.

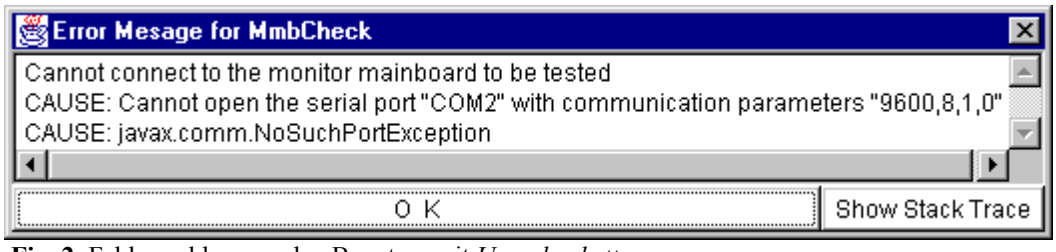

Fig. 2. Fehlermeldung an den Benutzer mit *Ursachenkette*

Der Autor hält diese Form einer Fehlermeldung an den Benutzer für

- *verständlich*, da die oberste Zeile das Wesentliche enthält,
- *diagnosestark*, da die Informationen der niedrigeren Schichten enthalten sind.

Zudem ist diese Form der Meldungszusammenstellung auch für die Programmierer äußerst *bequem*, da ohne weitere Mühe eine an den Benutzer gerichtete Meldung mit verschiedenen Ursachen kombiniert wird.

In der Oberflächenschicht kann dieses Verhalten durch folgenden typischen Aufbau eines Listener-Rumpfes erreicht werden:

```
try { Aufruf einer Operation der Funktionalitätsschicht; }
  catch(Exception e){Msg.report(ownerWindow, e);}
```

4.4 Der Stack-Trace mit Ursachenkette

Wenn der Benutzer im obigen Fehlermeldungsfenster den Button *Show Stack Trace* betätigt, wird der volle Stack-Trace mit den unverfälschten Ausnahme-Namen und -Parametern ausgegeben. Dies würde ein fehlersuchender Programmierer oder der Benutzer tun, wenn er sich an den Support wenden will.

In dem Stack-Trace erscheint die Ursachenkette dadurch, daß jede Stelle der Umdeutung einer Ausnahme in die einer höheren Schicht durch `WAS CAUSING:` markiert ist. Aufgrund der in einem Stack-Trace üblichen Reihenfolge beginnt dieser mit der Ausnahme der niedrigsten Programmschicht. Als Beispiel dient der Stack-Trace zu dem Fehlermeldungsfenster aus Fig. 2:

```
javax.comm.NoSuchPortException
 at javax.comm.CommPortIdentifier.getPortIdentifier(CommPortIdentifier.java:105)
WAS CAUSING:
x.SerialPort$OpenFailure: {0}=COM2  {1}=9600,8,1,0
  at x.SerialPort.open(SerialPort.java:120)
  at x.SerialPort.<init>(SerialPort.java:53)
  at x.SerialPort.<init>(SerialPort.java:34)
WAS CAUSING:
f.MmbCom$ConnectFailure
  at f.MmbCom.connect(MmbCom.java:160)
  at f.MmbCom.<init>(MmbCom.java:36)
  at f.MmbCheck.<init>(MmbCheck.java:31)
  at b.MmbCheck.handleConnect (MmbCheck.java:504)
  at b.MmbCheck.actionPerformed(MmbCheck.java:212)
  at ...
```

5 Realisierbarkeit dieses Frameworks

5.1 Realisierbarkeit in Java

Dieses Framework wurde in Java realisiert, welches sich im Großen und Ganzen als gut geeignet dafür erwies.

Jedoch gestaltet sich die Deklaration einer Ausnahme mit Ursache und Parametern noch abschreckend aufwendig. Es reicht leider nicht aus, einfach die Basisklasse `Failure` zu beerben wie in

```
class OperationFailure extends Failure {}
```

Um die Ursache und die Ausnahmeparameter an den Konstruktor von `Failure` durchzureichen, muß der Programmierer einen eigenen Konstruktor definieren, was deutlich aufwendiger ist.
Daher ergeben sich für die Sprachentwicklung von Java folgende Wünsche:

- Ohne viel Schreibaufwand *alle* Konstruktoren der Oberklasse erben zu können.
- Die Klasse `java.lang.Throwable` sollte die Eigenschaften von `Failure` haben und bei einem `throw` innerhalb eines `catch`-Zweiges sollte die verursachende Ausnahme automatisch eingetragen werden (ähnlich wie der Stack-Trace).

5.2 Realisierbarkeit in C++

Das Ausnahmebehandlungskonzept von C++ ist der Vorläufer von dem in Java. Da auch in C++ eine Ausnahmeklasse mit beliebigen Attributen (also auch einer Ausnahmenreferenz) versehen werden kann, ist das Konzept der Ursachenkette auch in C++ gut umsetzbar.
Mängel von C++ sind jedoch:

- Keine standardisierte Möglichkeit des Zugriffs auf den Stack-Trace
- Einer abgefangenen Ausnahme sieht man nicht an, ob sie per Variablendeklaration oder Allokation erzeugt wurde. Die Bereinigung bereitet daher große Probleme.
- Keine Spezifikationspflicht der auslösbaren Ausnahmen im Operationskopf.
- Kein Handler formulierbar, der beliebige Ausnahmen abfängt und meldet.

5.3 Realisierbarkeit in Ada

In Ada wäre dieses Framework nur unter größten Mühen umsetzbar, da die Sprache für unsere Zwecke folgende Mängel aufweist:

- Eine Ausnahme kann nur einen String als Attribut haben (ab Ada`95).
- Keine standardisierte Möglichkeit des Zugriffs auf den Stack-Trace
- Keine Spezifikationsmöglichkeit der auslösbaren Ausnahmen im Operationskopf.

6 Erfahrungen mit diesem Framework

Das propagierte Konzept (Ausnahmen mit Ursache und Parametern) wurde von den Programmierern des Mainboard-Prüfprogramms bei [LAR] weitgehend akzeptiert.

- Es enthielten 53% der `throw`-Anweisungen eine Ursachenerfassung.
- In 42% der `throw`-Anweisungen wurden weder Ursache noch Parameter erfaßt.
- Zu 5% handelte es sich um ein einfaches Re-`throw` nach Aufräumarbeiten.

In nur 10% der `throw`-Anweisungen hätte man meiner Meinung nach sinnvoll Ausnahmeparameter ergänzen sollen, z. B. sollte man einer `TimeoutExc` noch die angewandte Timeout-Frist mitgeben.

Von den Programmierern wurde insbesondere begrüßt:

- Eine Strategie zur Ausnahmebehandlung vorgegeben zu bekommen
- Die einfache Meldungstextverknüpfung
- Hilfe gegen die ansonsten immer ausufernde Auflistung von Ausnahmenamen in den `throws`-Klauseln von Operationen der oberen Schichten, vgl. [ES90] p. 363
- Ausführliche Diagnoseinformationen im Fehlerfall

Insofern kann für die Entwicklung robuster Anwendungen Java und das hier beschriebene Ausnahmebehandlungskonzept sehr empfohlen werden. Ein Code-Beispiel steht unter [MulTEx].

Quellen

[Ada83] The Programming Language Ada Reference Manual. American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983, LNCS 155, Springer-Verlag Berlin (1983) 179

[Balzert82] Balzert H.: Die Entwicklung von Software-Systemen. Bibliographisches Institut, Mannheim (1982), (Reihe Informatik; Bd. 34) 13-14

[Black83] Black A. P.: Exception Handling: The Case Against. Dissertation, Dept. Of Computer Science, University of Washington, TR 82-01-02. 238pp., Washington (1983) 54. *Diese vergleichende, scharfe Kritik vieler Mechanismen zur Ausnahmebehandlung ist für jeden Liebhaber derselben äußerst interessant.*

[Comm00] http://java.sun.com/products/javacomm/ (2000) *The „Serial Port (COMM)“ optional package (javax.comm) provides platform-independent access to asynchronous communication over a RS232 port.*

[ES90] Ellis M. A., Stroustrup B.: The Annotated C++ Reference Manual, Addison-Wesley Reading, Massachusetts (USA), (1990)

[Knabe87] Knabe Ch.: The Impact of Exception Handling on Labour Division, Safety, and Error Diagnostics in an Industrial Software Engineering Environment. In: Nichols H.K., Simpson D. (eds.): ESEC'87. 1st European Software Engineering Conference. Proceedings. Lecture Notes in Computer Science, Vol. 289. Springer-Verlag, Berlin Heidelberg New York (1987) 369-376.

[Knabe89] Knabe Ch.: Die Praxis der Wiederverwendung mit einer BOIE-basierten Software-Produktions-Umgebung. In Haas W. R.: „Softwaretechnik in Automatisierung und Kommunikation. Wiederverwendbarkeit von Software. Proceedings der ITG/GI/GMA-Fachtagung 15./16. 11. 1989 in Ulm“, ITG-Fachbericht Nr. 109, vde-verlag Berlin (1989) 151-159.

[Knabe90] Knabe Ch.: CeDRE. C Design and Realization Environment. Benutzerhandbuch. PSI GmbH, Berlin (1990). http://www.psi.de/

[LAR]°http://www.lar.com/

[MF99] http://java.sun.com/docs/books/tutorial/i18n/format/messageFormat.html (1999)

[Meyer90] Meyer B.: Objektorientierte Softwareentw. Hanser-Verlag, München (1990) 120

[MulTEx] http://www.tfh-berlin.de/~knabe/java/multex/ *MulTEx – The Multi-Tier Exception Handling Framework.*