Ein Framework zur Ausnahmebehandlung in mehrschichtigen Softwaresystemen

von
Christoph Knabe

www.tfh-berlin.de/~knabe

Technische Fachhochschule Berlin

Inhalt

1. Motivation

Praxis-Problem:

> Word kann diese Datei weder speichern noch erstellen.
> Eventuell ist der Datenträger schreibgeschützt.
> (D:\APPLEXC.WW6)

Hilfetext: Ca. 20 mögliche Ursachen

- **Datenträger schreibgeschützt, Datenträger voll, Datenträger defekt**
- **Zu viele Fenster offen**
- **...**

Lösung: keine

Bewertung: Miserable Diagnoseverwaltung

Vortragsinhalt: Wie sieht gute Fehlerbehandlung aus?

2. Qualitätsziele

Software-Produkt

- **Fehlertoleranz**
- **Selbsterklärung im Fehlerfall (Diagnosestärke)**

Entwicklungsprozeß

- **Programming by contract (Arbeitsteilung)**
- **Bequemlichkeit**

Qualitätsziel	Erreichbar durch
Fehlertoleranz	Automatischer Abbruch bei unbehandelter Ausnahme (ab Ada'83)
programming by contract	Dienst spezifiziert seine Ausnahmefälle (ab Eiffel'88) **String readLine() throws EndOfFile**
Diagnosestärke	MulTEx: Multi-Tier Exception Handling Framework

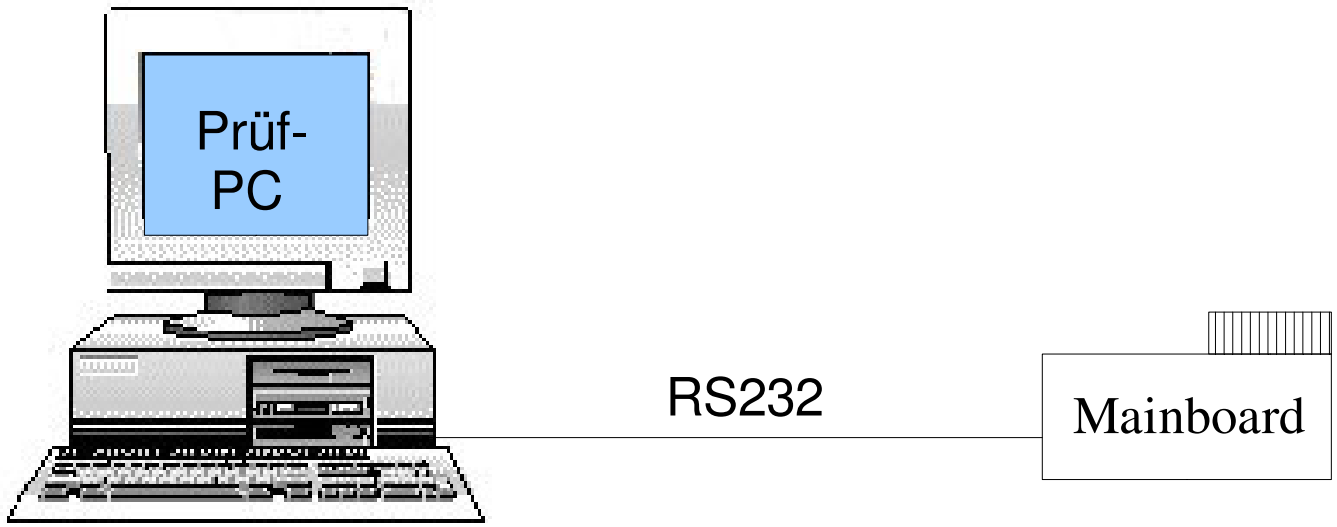
3. Diagnosekonzepte in MulTEx

3.1 Ursachenkette

SW-Architektur:

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

Anwendung: Prüfprogramm für Mainboards, kommuniziert über Serielle Schnittstelle



noch 3.1 Ursachenkette

Verbindungsaufnahme

b-Menüpunkt connect → f-Schicht

f-Ausnahme ConnectFailure ⇒

b-Meldung

Cannot connect to the monitor mainboard to be tested

für die Fehlerlokalisierung absolut unzureichend

noch 3.1 Ursachenkette

Mögliche Ursachen

- **Serielle Schnittstelle inexistent / falsch konfiguriert**
- **Fehler beim Senden der Initialisierungs-Botschaft**
- **Fehler beim Empfangen der Botschaftsquittung**
- **Ressourcenmangel**

Fazit: Fehlerursache in unteren Schichten bekannt,
muß erfasst und gemeldet werden!

noch 3.1 Ursachenkette

Bsp.: Serielle Schnittstelle inexistent:

Schicht	Operation		Schichtadäquate Ausnahme	
b	handleConnect	↓	*keine (Meldungsausgabe)*	
f	connect	↓	ConnectFailure	↑
x	open	↓	OpenFailure	↑
javax	getPortIdentifier		NoSuchPortException	↑

Ursachenkette: Kette der verursachenden Ausnahmen erfassen und mit melden.

Strategie für alle indirekt verursachten Ausnahmen

ansonsten nur vereinzelt: java.rmi.RemoteException, ab JDK 1.4 auch in Throwable

3.2 Weitere Diagnoseinformationen

Stack-Trace

unverzichtbar zur Fehlerlokalisierung
Ortsangaben der Aufrufhierarchie jeweils:

- **Klassenname**
- **Methodenname**
- **Quelldateiname**
- **Zeilennummer**

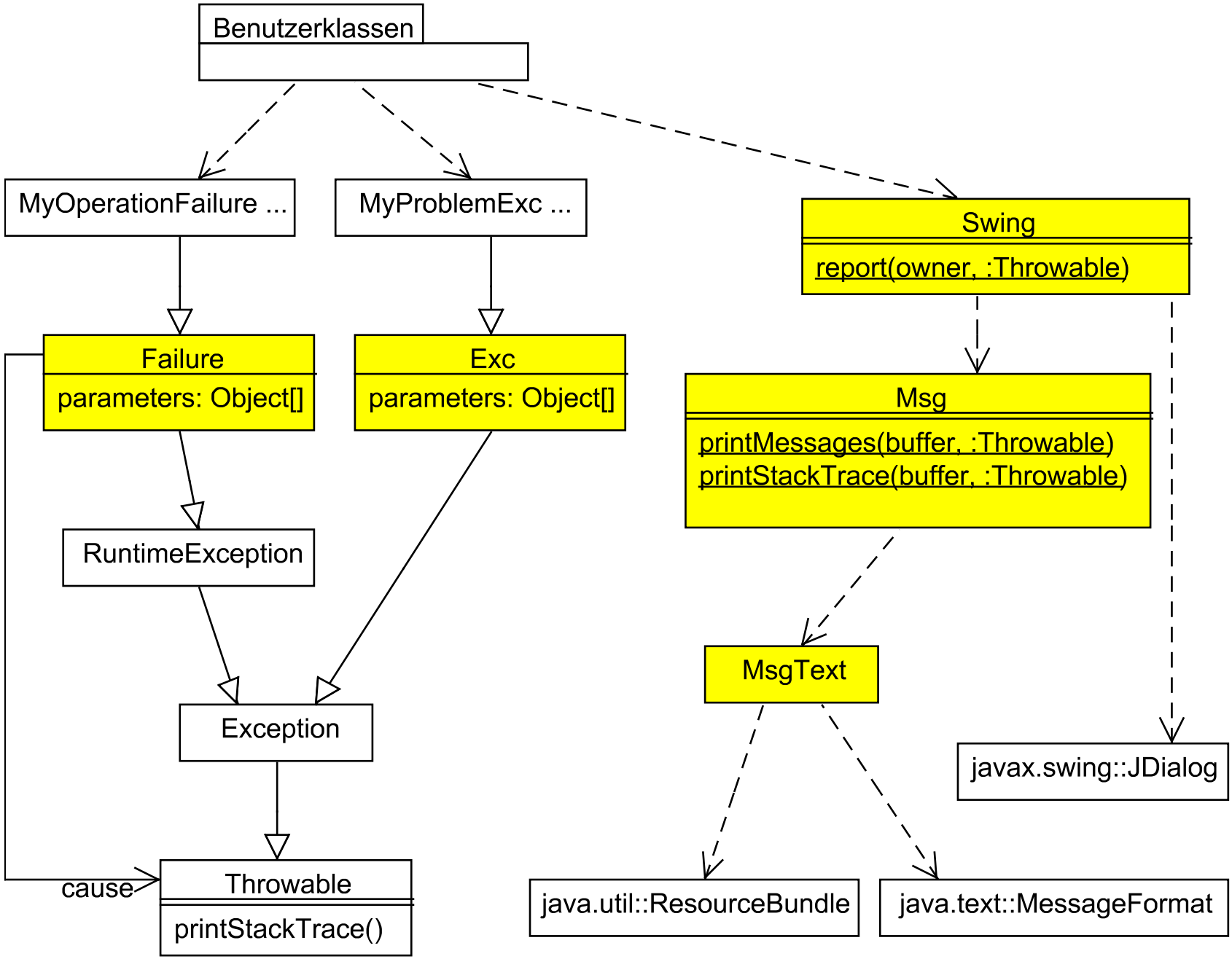
Ausnahmeparameter, Bsp.:

- **Name des SerialPort**
- **Kommunikationseinstellungen (baud, Bitzahl...)**

Meldungstextverknüpfung

- **Ausnahmen der oberen Schichten mit Meldungstext versehen**
- **Internationalisierbare Texte, Parameterreihenfolge und –formate**

4. Benutzung des Frameworks MulTEx



Das Framework in seinem Kontext

4.1 Erfassung von Ursache und Parametern einer Ausnahme

Ausnahme deklarieren

Konstruktor

```
Failure(
    String defaultMessagePattern,
    Exception cause,
    Object... parameters
)
```

Java: Konstruktor nicht vererbbar, nur Methoden.
Daher: Minimale Deklaration,
Parametrierung später über Fabrikmethode `create`.

Bsp. davon abgeleitet: x.SerialPort.OpenFailure:

```
static final class OpenFailure extends multex.Failure {}
```

noch 4.1 Erfassung von Ursache und Parametern

Ursache erfassen und Ausnahme auslösen

Operation x.SerialPort.open kann versagen mit

- **NameExc (originär festgestellt) bei falschem Portnamen**
- **OpenFailure (indirekt verursacht)**
  bei von unten kommenden Ausnahmen:
  NoSuchPortException,
  PortInUseException,
  UnsupportedCommOperationException,
  IOException

Folgende Seite:

Code zur Erfassung von Ursache und Parametern einer Ausnahme

```
public void open(
   String portName, int baudRate, int databits, int stopbits, int parity
) throws NameExc, OpenFailure {
  if(!portName.startsWith("COM")){throw create(NameExc.class, 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 ex) {
    …… //free resources
    //redefine exception:
    throw create(OpenFailure.class,
        ex, portName, baudRate, databits, stopbits, parity);
  } //catch
} //open
```

noch 4.1 Erfassung von Ursache und Parametern

In API-Schichten typischer Operationsrumpf:

```
if(Vorbedingung1 nicht erfüllt){
  throw create(Problem1Exc.class, parameter ...);
}
if(Vorbedingung2 nicht erfüllt){
  throw create(Problem2Exc.class, parameter ...);
}
...
try {
  Eigentlicher Algorithmus mit Aufruf von Diensten
} catch(Exception ex){
  throw create(OperationFailure.class, ex, parameter … );
}
```

4.2 Internationalisierbare Meldungstexte

Definition als Javadoc-Hauptkommentar jeder Ausnahmeklasse, Bsp.:

```
/**Cannot open the serial port "{0}"
  * with communication parameters "{1},{2},{3},{4}"
*/
static final class OpenFailure extends multex.Failure {}
```

Einsammeln durch das `ExceptionMesagesDoclet` in eine `ResourceBundle`-Datei, z. B. in

MsgText.properties:

```
x.SerialPort$OpenFailure = Cannot open the serial port "{0}"\
with communication parameters "{1},{2},{3},{4}"
```

Benutzt: java.text.MessageFormat

4.3 Arbeitsteilung und Benutzeroberfläche

Vorgehen: Erkannte Fehler als abfangbare Ausnahmen auslösen, erst inOberflächenschicht in Meldung umwandeln.

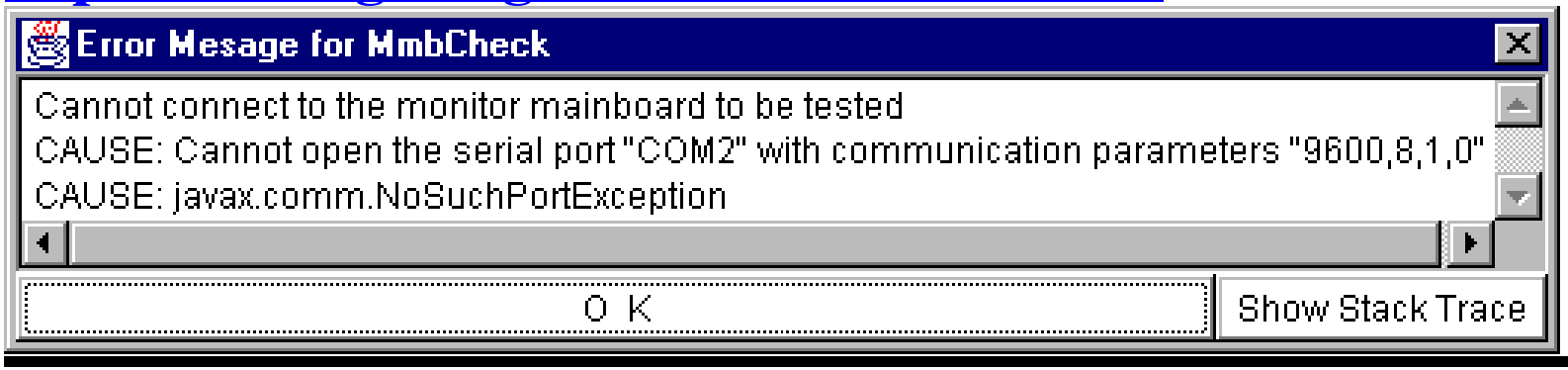
Bsp.: void connect() throws ConnectFailure

Meldungstext: Cannot connect to the monitor mainboard to be tested

In Oberflächenschicht:

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

Bsp.-Meldungsausgabe mit Ursachenkette:



Meldungsausgabe mit Ursachenkette:

- **Cannot connect to the monitor mainboard to be tested**
- **CAUSE: Cannot open the serial port "COM2" with communication parameters "9600,8,1,0"**
- **CAUSE: javax.comm.NoSuchPortException**

Bewertung:

- ***Verständlich*, da oberste Zeile das Wesentliche enthält**
- ***Diagnosestark*, da die Informationen der niedrigeren Schichten enthalten sind**
- ***Bequem* für Programmierer, da ohne Aufwand eine Benutzermeldung mit verschiedenen Ursachenmeldungen kombiniert wird.**

4.4 Stack-Trace und Ursachenkette

Fehlerlokalisierung:

Button „Show Stack Trace“ meldet:

- **Unverfälschte Ausnahmenamen, -parameter**
- **Aufruforte rückwärts (übliche Stacktrace-Reihenfolge)**
- **„WAS CAUSING:“ markiert Ausnahmenverursachung**

noch 4.4 MulTEx-Stacktrace mit Ursachenkette

```
javax.comm.NoSuchPortException
    at javax.comm.CommPortIdentifier.getPortIdentifier
       (CommPortIdentifier.java:105)
WAS CAUSING:
x.SerialPort$OpenFailure: {0}=COM2 {1}=9600 {2}=8 {3}=1 {4}=0
    at x.SerialPort.open(SerialPort.java:120)
    at x.SerialPort.<init>(SerialPort.java:53)
    at x.SerialPort.<init>(SerialPort.java:34)
WAS CAUSING:
f.MmbCom$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 ...  //Standardteil innerhalb von AWT/Swing
```

5. Realisierbarkeit dieses Frameworks

Notwendiges Feature	Java	C++	Ada
Ausnahme parametrierbar mit Ausnahmen	+	+	– String
Sammel-Handler für alle Ausnahmen möglich	+ Throwable	–	+ others
Ermitteln des Namens einer Ausnahme	+ Reflection	+ RTTI	+ Ada'95
Zugriff auf den Stack Trace	+	–	–
Spezifikation der Ausnahmen im Operationskopf	+ Pflicht	0 möglich	– unmöglich
Erben parametrierter Konstruktoren	–	–	–

6. Erfahrungen / Ausblick

Bisheriger Einsatz

- **LAR: Monitor-Mainboard-Prüfprogramm, Monitor-Steuersoftware**
- **Diplomarbeiten, viele Software-Projekte im Hauptstudium**
- **Software des Fachbereich VI-Webservers**

Positiv

+ **Strategie zur Ausnahmebehandlung vorgegeben**
+ **Hilfe gegen erzwungene Ausuferung von `throws`-Klauseln in den oberen Schichten**
+ **Einfache Meldungstextverknüpfung**
+ **Ausführliche Diagnoseinfos im Fehlerfall**

Bezug

www.tfh-berlin.de/~knabe/java/multex/

A. Verbesserungen in MulTEx seit der Erstversion 1998

- **Umbenennung: Failed → Failure [Ehre an CLU]**
- **Meldungsausgabedienste getrennt:**
  Msg.report(..., ex) → StringBuffer, Streams
  Swing.report(..., ex) → Swing-JDialog
- **Meldungstext im Javadoc-Kommentar:**
  ⇒ bequemere Vorbereitung für Internationalisierung.
- **Generische Fabrikmethode für Ausnahme-Erzeugung+Parametrierung**

B. Ursachenkette jetzt auch in JDK 1.4

Throwable wurde im JDK 1.4 um das „Chained Exception Facility“ erweitert:

- **Konstruktoren Throwable(Throwable) und Throwable(String, Throwable) erfassen Ursache einer Ausnahme.**
- **Alternativ kann Ursache auch ohne speziellen Konstruktor mittels Operation initCause(Throwable) nachträglich erfaßt werden, Bsp.:**

```
throw (IllegalArgumentException )
    new IllegalArgumentException(arg).initCause(ex);
```

- **Einheitlicher Zugriff auf verursachende Ausnahme mittels getCause()**
- **printStackTrace() meldet alle beteiligten Stack Traces von oben nach unten.**

B.1 JDK1.4: Ursachenkette im Stack Trace, Beispiel

Im Stacktrace des JDK 1.4 leider widersprüchliche Reihenfolge:

- **Ausnahmen von oben nach unten**
- **Programmzeilen von unten nach oben**

```
HighLevelException
    at Junk.a(Junk.java:13)
    at Junk.main(Junk.java:4)
Caused by: MidLevelException
    at Junk.c(Junk.java:23)
    at Junk.b(Junk.java:17)
    at Junk.a(Junk.java:11)
    ... 1 more
Caused by: LowLevelException
    at Junk.e(Junk.java:30)
    at Junk.d(Junk.java:27)
    at Junk.c(Junk.java:21)
    ... 3 more
```