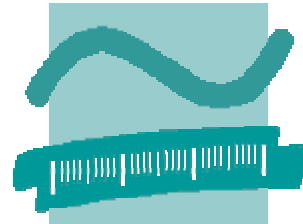


# **Ein Framework zur Ausnahmebehandlung in mehrschichtigen Softwaresystemen**

---

**von**  
**Christoph Knabe**  
[\*\*www.tfh-berlin.de/~knabe\*\*](http://www.tfh-berlin.de/~knabe)



**Technische Fachhochschule Berlin**

## **1. Motivation**

## **2. Qualitätsziele**

## **3. Diagnosekonzepte**

## **4. Benutzung des Frameworks**

- Erfassung der Diagnoseinfos**

- Ausnahmen melden**

## **5. Realisierbarkeit dieses Frameworks in Java, C++, Ada**

## **6. Erfahrungen / Ausblick**

### **A. Verbesserungen seither**

### **B. Ursachenkette ab JDK 1.4**

# 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

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

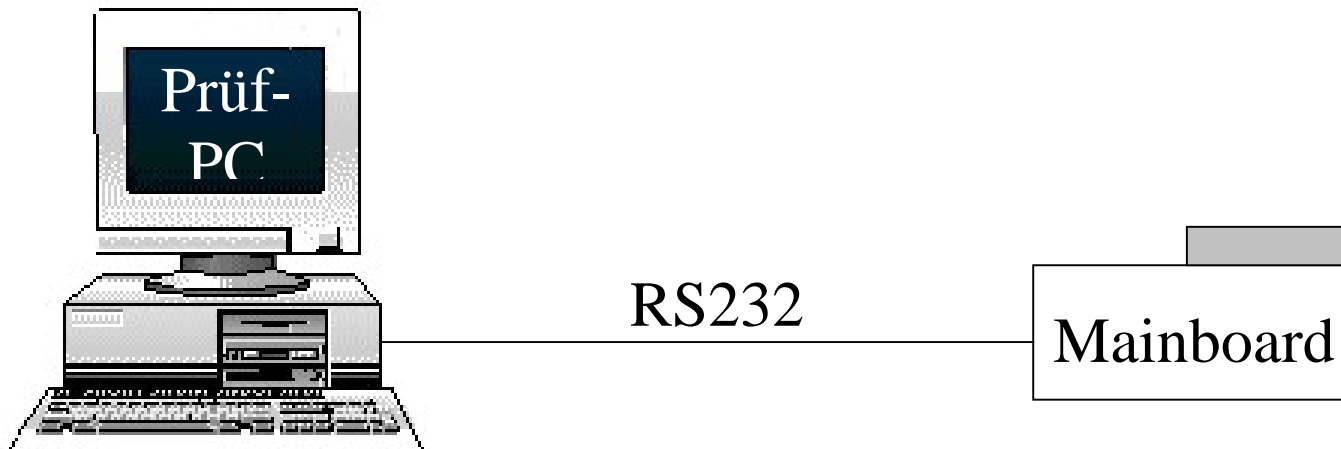
### 3. Diagnosekonzepte in MulTEx

---

#### 3.1 Ursachenkette

<b>SW-Architektur:</b>	<u>Benutzeroberfläche</u>	<b>b</b>
	<u>Funktionalität</u>	<b>f</b>
	<b>Datenhaltung + Dienste</b>	<b>x</b>

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:**

<u>Schicht</u>	<u>Operation</u>		<u>Schichtadäquate Ausnahme</u>
b	handleConnect	↓	<i>keine (Meldungsausgabe)</i>
f	connect	↓	ConnectFailure ↑
x	open	↓	OpenFailure ↑
java	getPortIdentifizier		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**

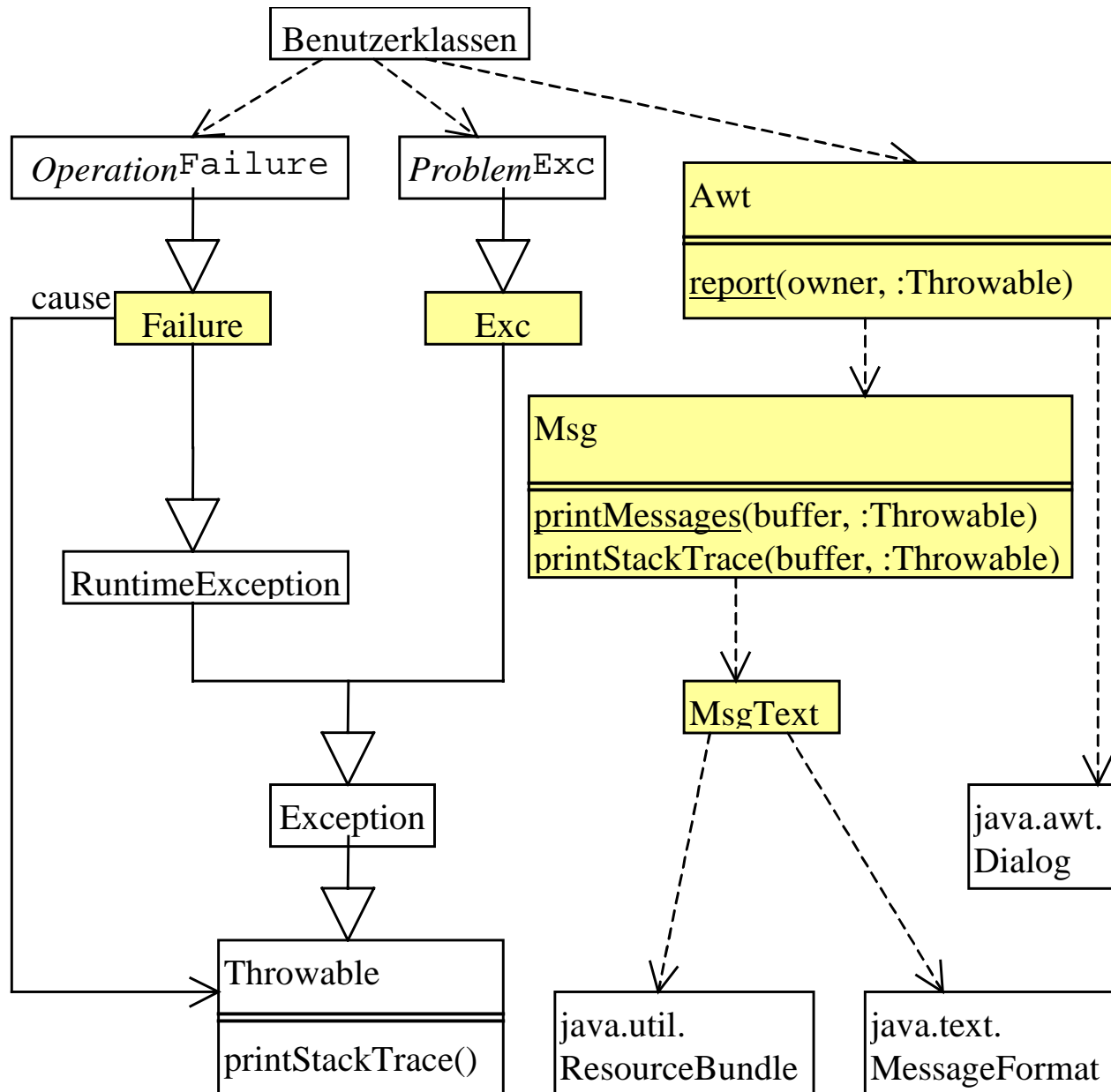
Ausnahmeparameter, Bsp.:

- **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 messagePattern, Exception cause, Object param, ...)**

Bsp. davon abgeleitet: **x.SerialPort.OpenFailure:**

```
class OpenFailure extends Failure {  
    public OpenFailure(  
        Exception cause, String portName,  
        int baudRate, int databits, int stopbits, int parity  
    ){ super(  
        "Serielle Schnittstelle {0} konnte nicht mit Einstellungen {1} geöffnet werden.",  
        cause, portName, ""+baudRate+', '+databits+', '+stopbits+', '+parity  
    );}  
}
```

## 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 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 ex) {
        ..... //free resources
        //redefine exception:
        throw new OpenFailure(ex, portName, baudRate, databits, stopbits, parity);
    } //catch
} //open

```

## noch 4.1 Erfassung von Ursache und Parametern

---

### In f/x-Schichten typischer Operationsrumpf:

```
if(Vorbedingung nicht erfüllt){  
    throw new ProblemExc(parameter ...);  
}  
...  
try {  
    Eigentlicher Algorithmus mit Aufruf von Diensten  
} catch(Exception ex){  
    throw new OperationFailure (e, parameter ... );  
}
```

## 4.2 Internationalisierte Meldungstexte

---

Werden in **ResourceBundle**-Datei definiert,  
z. B. in

**MsgText\_en.properties:**

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

**Benutzt:** **java.text.MessageFormat**

## 4.3 Arbeitsteilung und Benutzeroberfläche

---

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

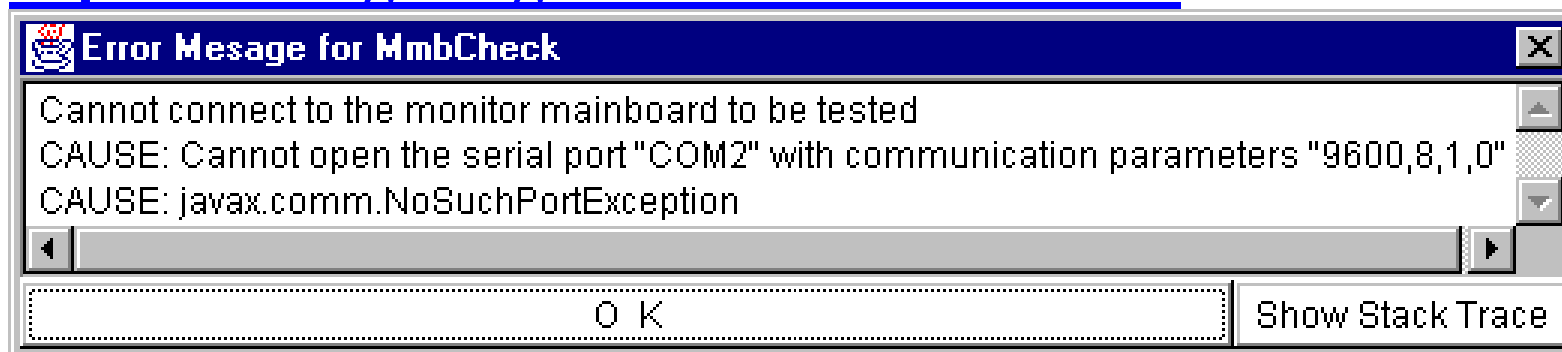
**Bsp.**: **void connect() throws ConnectFailure**

**Meldungstext**: Cannot connect to the monitor mainboard to be tested

**In Oberflächenschicht**:

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

**Bsp.-Meldungsausgabe mit Ursachenkette**:





## noch 4.3 Arbeitsteilung und Benutzeroberfläche

---

### 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}=9600,8,1,0

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 ... //Standardteil innerhalb des AWT

## 5. Realisierbarkeit dieses Frameworks

---

<u>Notwendiges Feature</u>	<u>Java</u>	<u>C++</u>	<u>Ada</u>
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**
- **Diplomarbeiten, Software-Projekte im Hauptstudium**

### 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/](http://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**  
**Awt.report(..., ex)** → **AWT-Dialog**
- **Default-Meldungstext** im Ausnahmeobjekt:  
bequemere Benutzung, wenn keine Internationalisierung nötig.
- **Bequemlichkeitskonstruktoren** mit 1..10 Ausnahmeparametern  
Benutzung ohne **new Object[]** möglich.

### Beispiel:

```
try { ... } catch(Exception ex){  
    throw new multex.Failure(  
        "Datei {0} konnte nicht nach {1} kopiert werden", ex, quelle, ziel  
    );  
}
```

## 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