

- Software Richtlinien
- Typisierung
- Vertragsbasierte Programmierung
- ➔ ■ Fehlertolerante Programmierung
- Portabilität
- Dokumentation




Fehlertolerante Programmierung:

Verbesserung des Reaktionsverhaltens eines Programms im Fall eines Fehlers.

Die Bedeutung der fehlertoleranten Programmierung hängt von der Anwendung ab. Ein Textverarbeitungssystem kann auch mal abstürzen, Flugzeugsteuerungssoftware sollte das nicht.



- 
- Software Redundanz
 - Selbstüberwachende Systeme
 - Ausnahmebehandlung



- **Funktionale Redundanz**

Erweiterung um zusätzliche Funktionen, die ausschließlich der Erhöhung der Fehlertoleranz dienen.

- **Informationelle Redundanz**

Nutzdaten werden um zusätzliche Informationen angereichert.



- **Temporale Redundanz**
Zeitanforderungen werden übererfüllt, so dass ggfs. eine Wiederholung stattfinden kann.
- **Strukturelle Redundanz**
Mehrfachauslegung von Komponenten.
→ Details nächste Seiten



Ausprägungen:

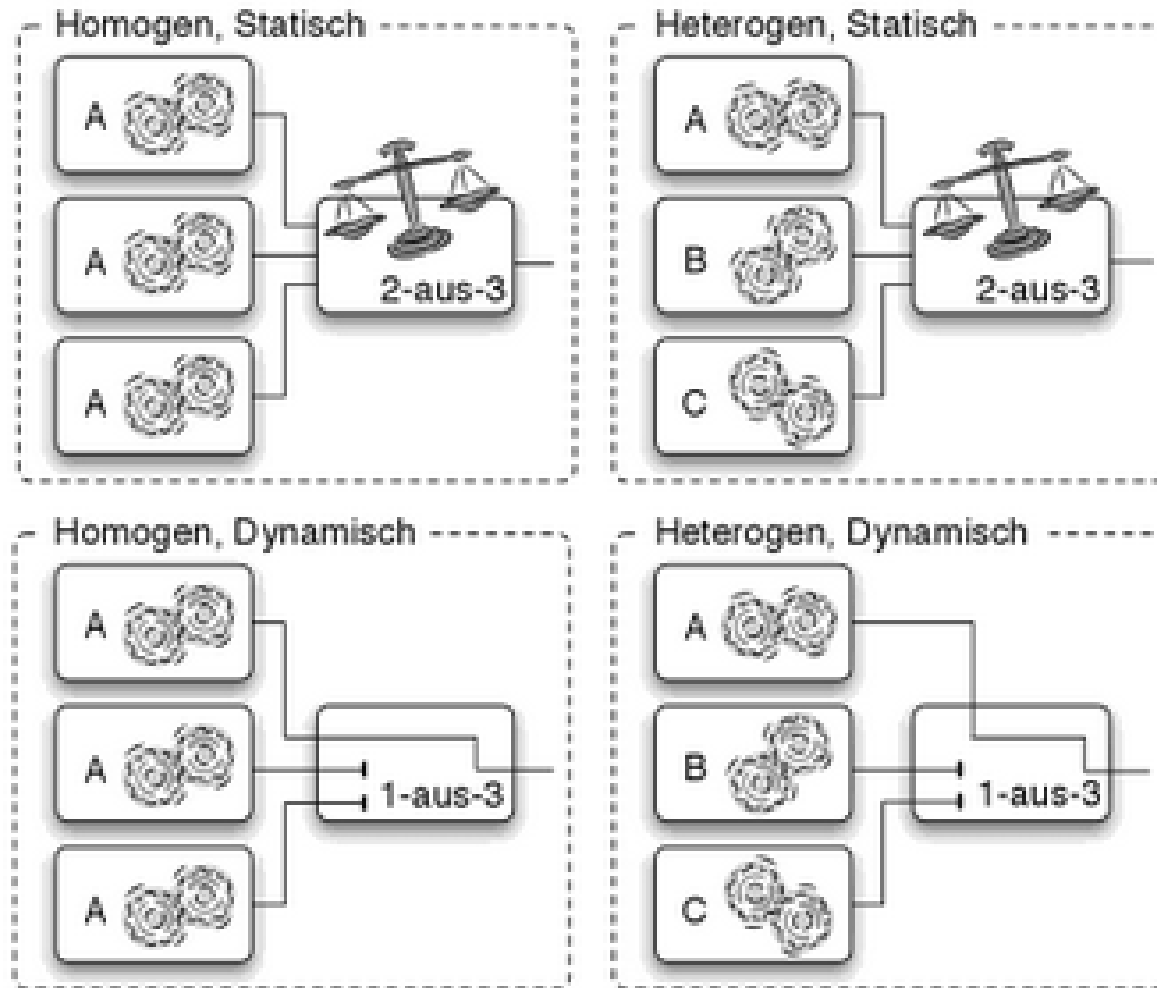
- Homogene Redundanz (n Komponenten gleicher Bauart)
Ausschließlich im Hardware Bereich
- Heterogene Redundanz (n Komponenten unterschiedlicher Bauart)

Ausprägungen:

- Statische Redundanz (alle Komponenten aktiv, Ergebnisse werden durch Voter verglichen)
- Dynamische Redundanz (Ersatzkomponenten, die bei Bedarf aktiviert werden)



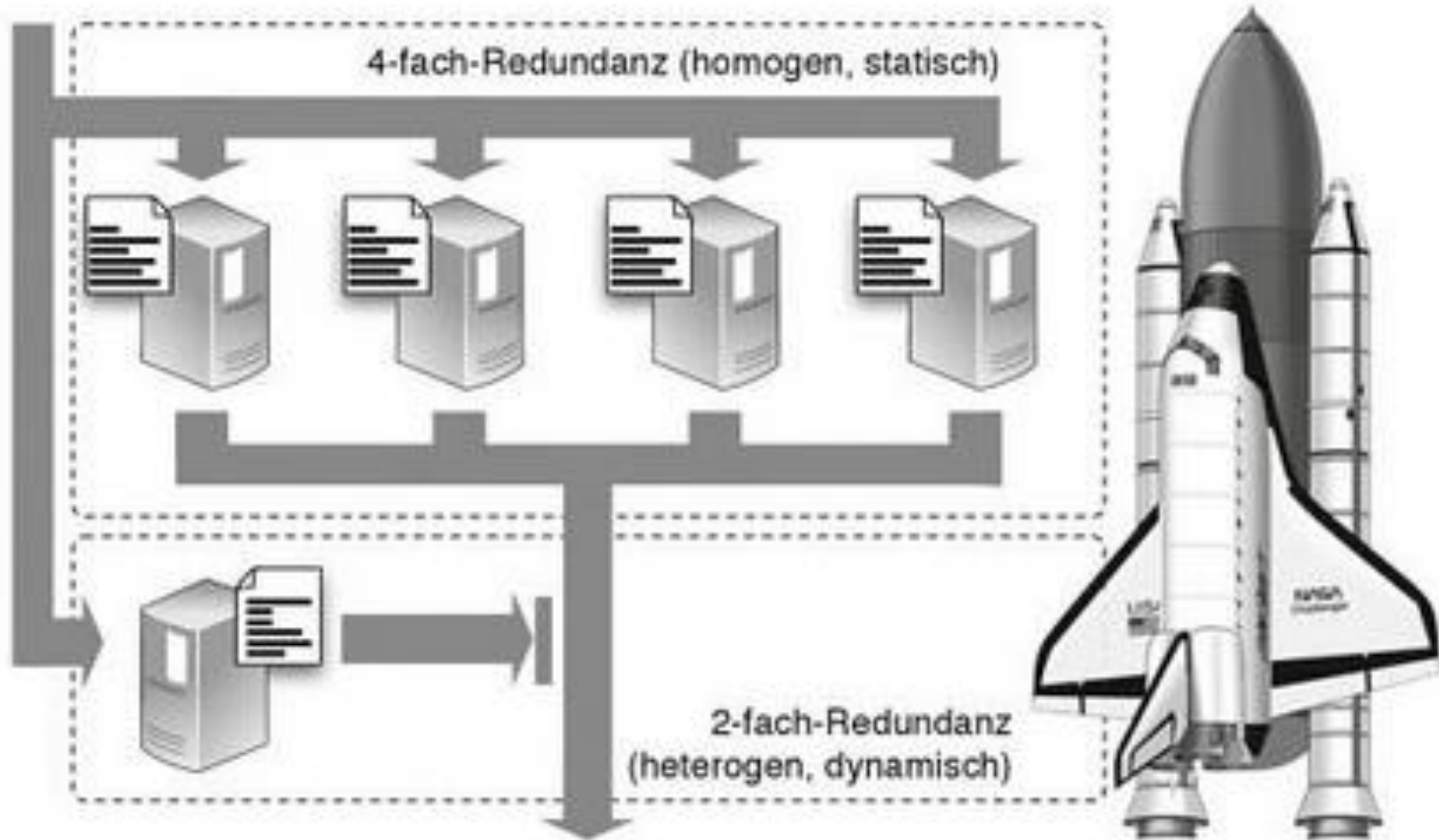
Homogene und Heterogene Redundanz



Quelle der Abb.: D.
Hoffmann: *Software
Qualität*



SW Redundanz, Bsp. Space Shuttle



Fehlertolerante Programmierung

- Software Redundanz
- Selbstüberwachende Systeme
- Ausnahmebehandlung



Reaktionsszenarien

- **Fail-Safe Reaktion**

Wechsel in einen sicheren Zustand (z.B. Selbstabschaltung)
Bsp: Selbstaktivierende Notbremse eines Aufzugs,
Sitzplatzverriegelung einer Achterbahn.

- **Selbstreparatur**

Selbstdiagnose und Reparatur
Bsp: automatische Wiederherstellung von Dateien

- **Reaktivierung**

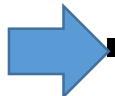
Watchdog Logik, Bsp: Explorer Prozess in Windows XP



- Software Redundanz
- Selbstüberwachende Systeme
- Ausnahmebehandlung



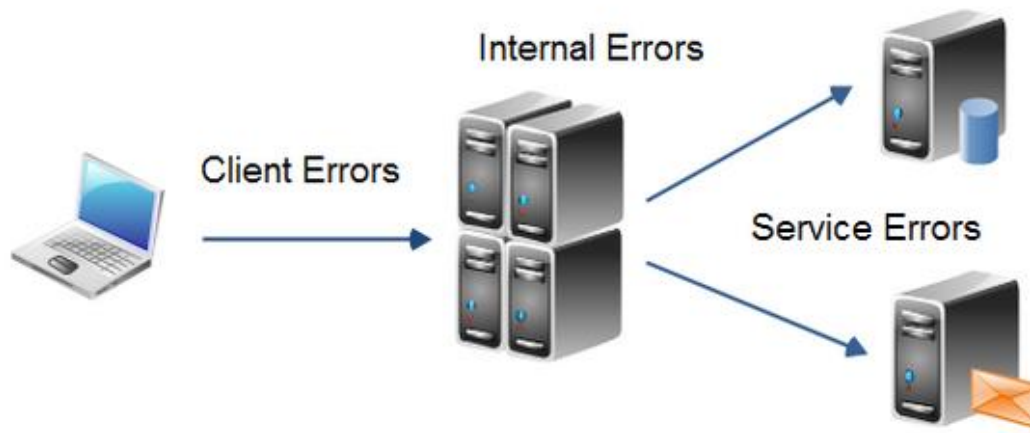
Ausnahmebehandlung

- 
- Einleitung
 - Strukturierte Ausnahmebehandlung
 - Java Checked Exceptions
 - Exception Handling Strategie



Einteilung von Exceptions

- Benutzer Fehler – Client Errors
- Programmierfehler – Internal errors
- Fehler bei Zugriffen auf Ressourcen – Service Errors



Typische Reaktionen

Client Errors:

- Aktion abbrechen
- Ressourcen schließen
- Speicher freigeben
- Benutzer informieren
- Error loggen

Service Errors:

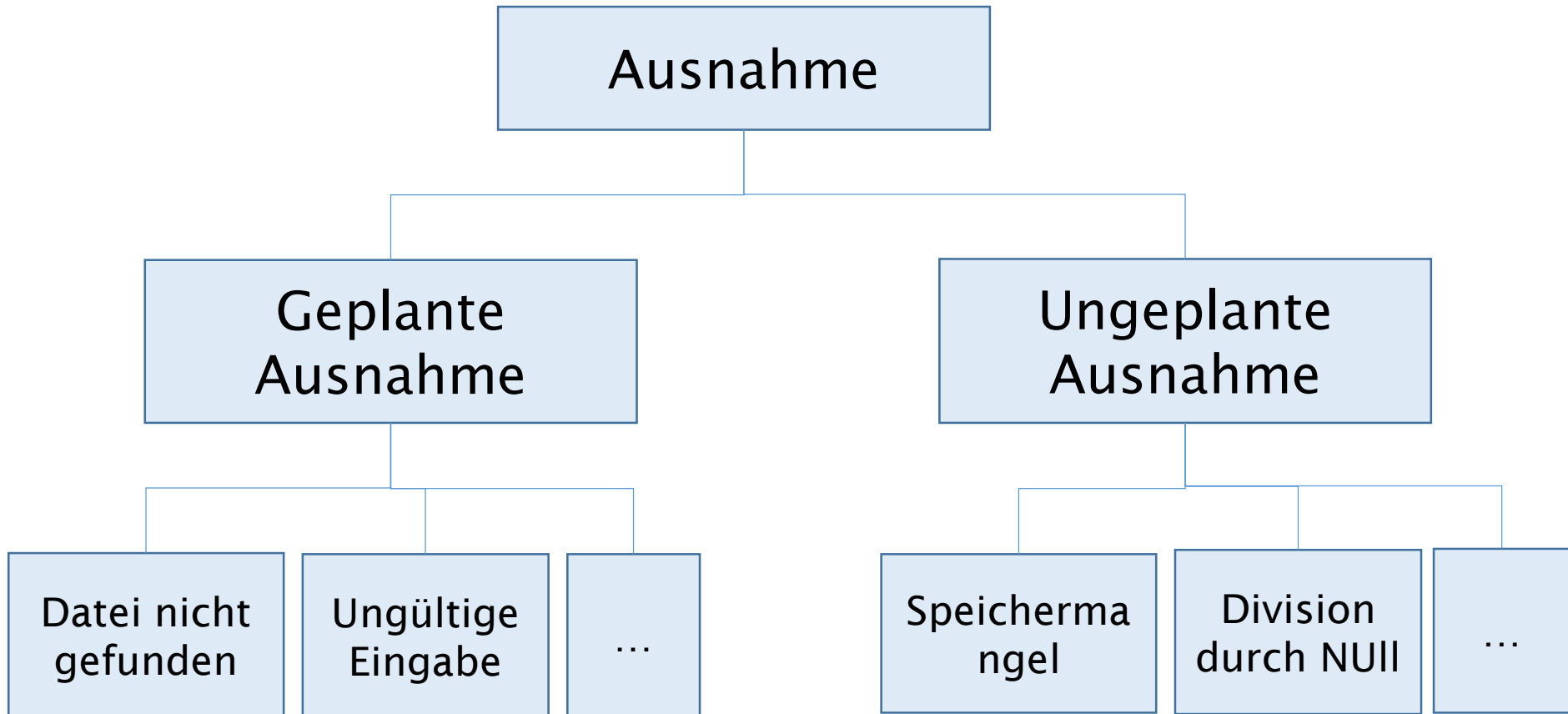
- Aktion abbrechen
- Ggfs später nochmal versuchen
- Ressourcen schließen
- Speicher freigeben
- Benutzer informieren
- Error loggen
- Nachricht an Admin

Internal Errors:

- Aktion abbrechen
- Ressourcen schließen
- Speicher freigeben
- Benutzer informieren
- Error loggen
- Nachricht an Admin
- Entwickler informieren



Ausnahmen



Unterschiedliche Philosophien

- Delegation der Ausnahmebehandlung ans Betriebssystem.
- Behandlung der Ausnahmen in dem Konzept der Programmiersprache verankert.



Ausnahmebehandlung

- In C: Ausnahmebehandlung mit geschachtelten If Strukturen und return codes
- In Java: Ausnahmebehandlung mit Exceptions



Auswirkungen fehlerhaften Exceptionshandlings

- GUI: Sanduhr verschwindet nicht.
- Dateien werden nicht geschlossen.
- Datenbankverbindungen bleiben offen.

Auswirkungen oftmals nicht sofort, sondern erst nach einer gewissen Laufzeit.



Beschreibung der Funktion `readFile()`:

1. Datei wird geöffnet.
2. Dateigröße wird ermittelt.
3. Speicherplatz belegt.
4. Dateiinhalt einlesen.
5. Datei schließen.

Bei jedem Schritt können Fehler passieren.
Realisierung in C und in Java?



Geschachtelte If-Strukturen

```
int readFile(char *name)
{
    int error_code;
    <Datei öffnen>
    if(<Datei erfolgreich geöffnet>){
        <ermittle Dateigröße>
        if(<Dateigröße erfolgreich ermittelt>){
            <Belege Speicher>
            if(<Speicher erfolgreich belegt>){
                <Lese Datei Inhalt>
                if(<Dateiinhalt erfolgreich gelesen>)
                {
                    error_code = 0;
                }else{
                    error_code = 1;
                }
            }else
            {
                error_code = 2;
            }
        }else{
            error_code = 3;
        }
        <Schließe Datei>
        if(<Datei nicht schließbar>){
            error_code = 4;
        }
    }else{
        error_code = 5;
    }

    return error_code;
}
```




Exception Handling in Java

```
void readFile(String name){  
    try{  
        <Öffne Datei>  
        <Ermittle Dateigröße>  
        <Belege Speicher>  
        <Lese Dateinhalt>  
        <Schließe Datei>  
    }catch(DateiÖffnenFehler dfe){  
        <korrekte Fehlerreaktion>  
    }catch(DateiGrößenFehler dgf){  
        <korrekte Fehlerreaktion>  
    }catch(Speicherbelegenfehler sbe){  
        <korrekte Fehlerreaktion>  
    }catch(DateilesenFehler dle){  
        <korrekte Fehlerreaktion>  
    }catch(DateiSchließenFehler dsf){  
        <korrekte Fehlerreaktion>  
    }  
}
```

Fachliche Logik

Fehlerbehandlung



Ausnahmebehandlung

- Einleitung
- ➔ ▪ Strukturierte Ausnahmebehandlung
- Java Checked Exceptions
- Exception Handling Strategie



Strukturierte Ausnahmebehandlung

Idee: Code zur Ausnahmebehandlung wird vom Anwendungscode getrennt.

1. **Loggen** der Ausnahme nahe der Stelle, wo sie passiert ist.
2. Ggfs. an den Aufrufer **weiterreichen**.
3. **Behandlung** der Ausnahme an der Stelle, wo die Konsequenzen erkannt werden können und eine angemessene Reaktion erfolgen kann.
4. Angemessene **Reaktion**:
 - Benutzer wird zu erneuter Eingabe aufgefordert.
 - Datei wird geschlossen.
 - Programm wird beendet.
 - Rollback von Aktionen
 - ...

Vorteile der SEH

- Trennung Anwendungscode von Ausnahmebehandlungen
- Exceptions können nicht ignoriert werden. (Rückgabewerte schon)
- Fehler können einfach die Aufrufhierarchie hochpropagiert werden.
- Fehler können sauber gruppiert und differenziert werden.

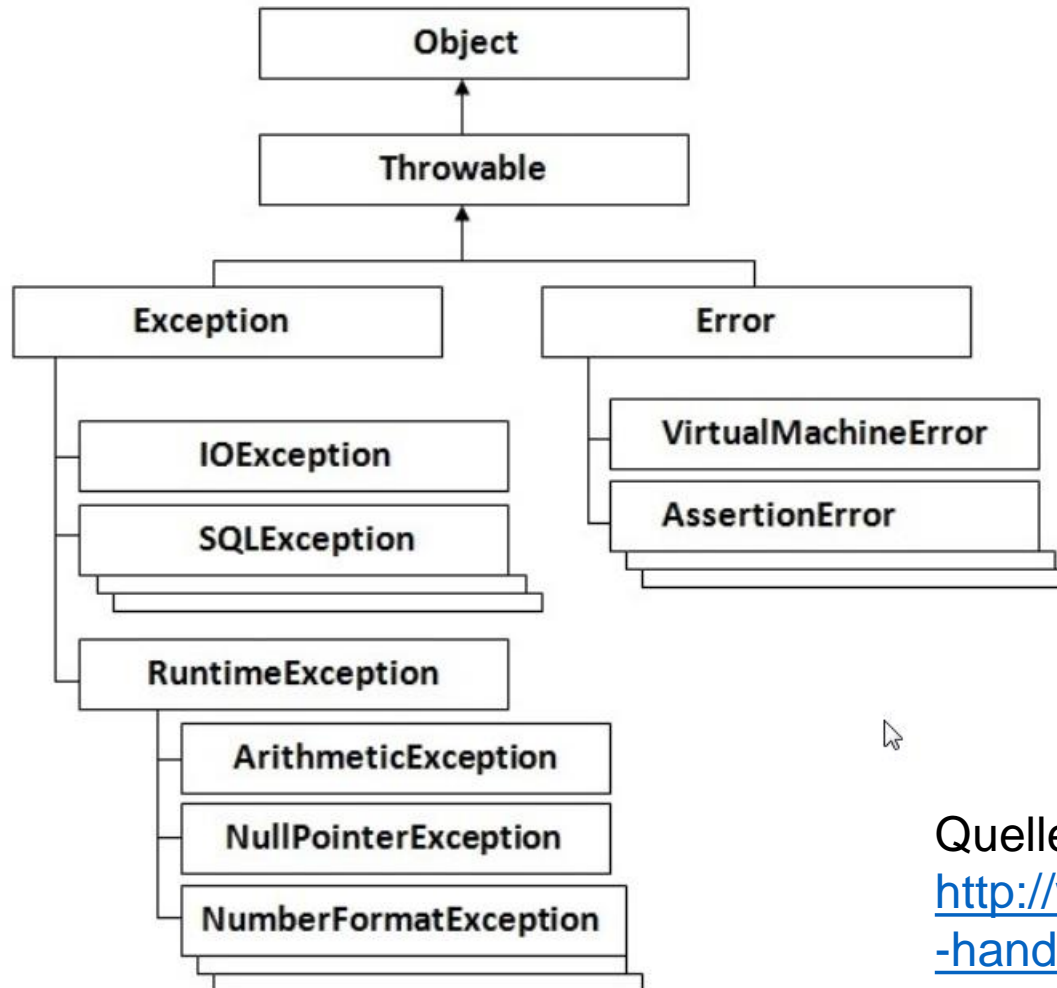
Programmiersprachen mit Ausnahmebehandlung: Bsp: Java, C#, Ruby, Eiffel, Ada ...

Ausnahmebehandlung

- Einleitung
- Strukturierte Ausnahmebehandlung
- ➔ ▪ Java Checked Exceptions
- Exception Handling Strategie



Java Exceptions



Quelle der Abbildung:
<http://www.javatpoint.com/exception-handling-in-java>



Java checked Exception: Compiler prüft, ob alle Stellen, wo diese Exception auftreten kann, durch Code zum Abfangen abgedeckt sind.

Idee: Der Entwickler wird gezwungen, alle Ausnahmen, auf die man angemessen reagieren kann, auch wirklich zu behandeln.

Außerdem in Java: **Unchecked Exceptions:** Können auch behandelt werden, müssen es aber nicht.

Errors: Unchecked Exceptions, die auf ein wirklich ernsthaftes Problem hinweisen und nicht gefangen werden sollten.



Gutes Exceptions Handling →

- Einfachere Entwicklung
- Einfachere Wartung
- Weniger bugs
- Einfachere Anwendung

Schlechtes Exceptions Handling →

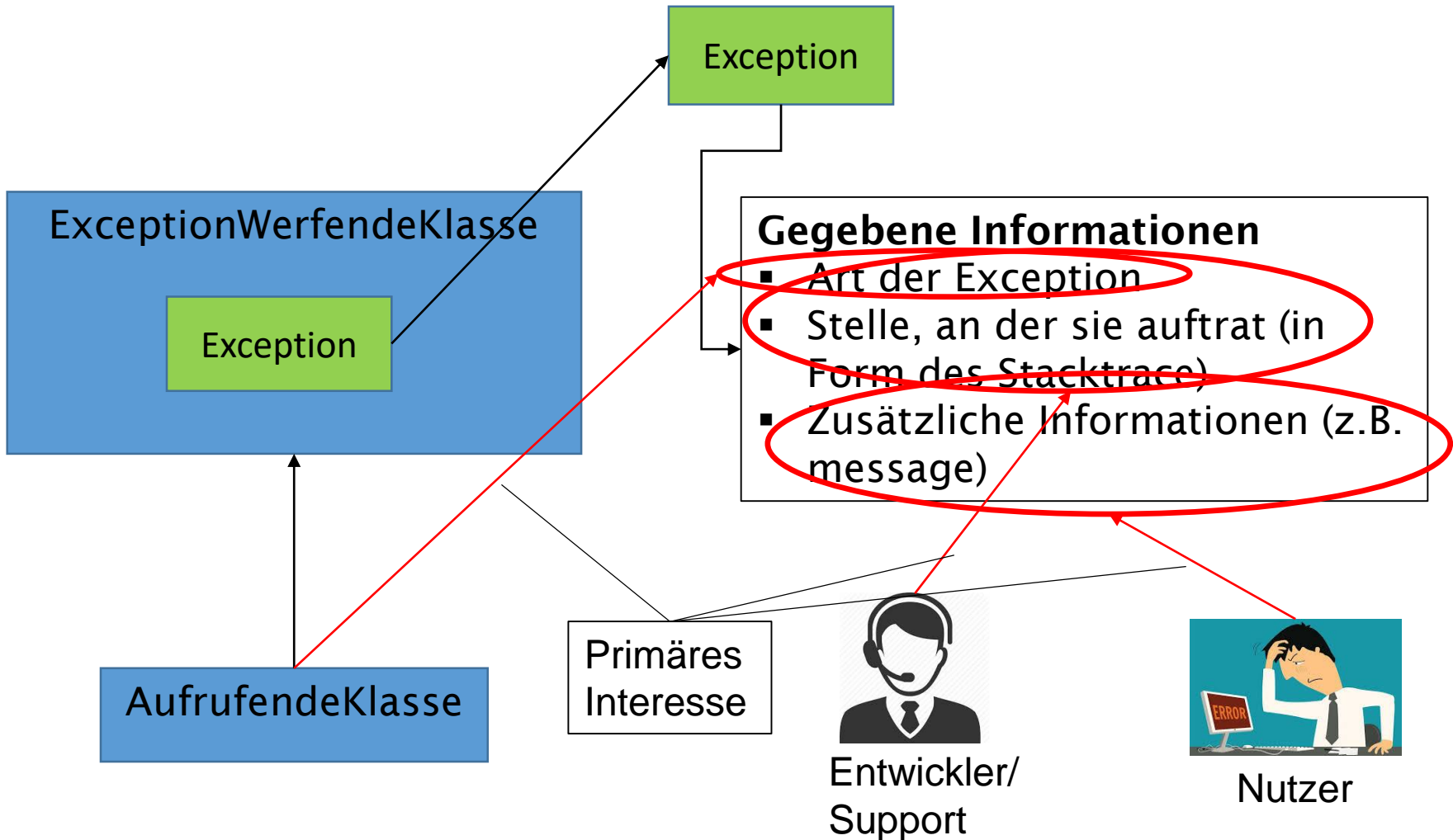
- Verwirrung der Benutzer
- Schwere Wartung

→ Exceptions Handling und error recovery muss im Design Prozess berücksichtigt werden!

- Die Klasse, die die Exception wirft.
- Die Klasse, die die Exception fängt.
- Der Benutzer, der mit dem Resultat umgehen muss.
- Entwickler/Support



Perspektiven des Exception Handlings



Die geworfenen Exception Types einer Klasse gehören zum Design.

- **Ausgangspunkt:** Der Aufrufer, nicht der aufgerufene.
 - Welche Typen kann der Aufrufer behandeln, welche reicht er vermutlich weiter (an Aufrufer oder Benutzer)?
 - Normalerweise **NICHT**: Deklaration aller möglichen Exceptions im throws.
 - Gründe:
 - Bewahrung der Designhoheit
 - Offenlegung von Implementierungsaspekten im Interface
- **Stabile throws clauses**
 - Kleine Anzahl von Exceptions auf höheren Abstraktionsleveln führt zu stabileren Methodensignaturen (Aufrufer werden es danken).



Bsp: Klasse ResourceLoader lädt Ressourcen aus File, DB, remote Server – je nach Initialisierung.

➔ Wie sieht die Methodensignatur aus?

➔ Welche Exceptions werden deklariert?



Negativ Beispiel für throws

Bsp: Klasse ResourceLoader lädt Ressourcen aus File, DB, remote Server – je nach Initialisierung.

Methodensignatur

```
public Resource getResource() throws SQLException, IOException, RemoteException
```

1. **Frage:** Ist die Detail-Information der Exceptions für den Aufrufer relevant?
==> Normalerweise nicht. Relevant ist die Information, dass die Ressource nicht verfügbar ist.
2. Es wird Detail Information der Implementierung offenbart (DB Zugriff, File Zugriff, Remote Zugriff)

➔ Besser: Siehe nächste Folie



Bessere Signatur

```
public Resource getResource() throws ResourceLoadException{  
  
    try{  
        //access resource  
    }  
    catch(SQLException sqle){  
        throw new ResourceLoadException();  
    }  
    catch(RemoteException re){  
        throw new ResourceLoadException();  
    }  
    catch(IOException ioe){  
        throw new ResourceLoadException();  
    }  
}
```

Problem: Es geht Information verloren, die interessant ist für Entwickler/Support.



Java Multicatch Feature

```
public Resource getResource() throws ResourceLoadException{  
  
    try{  
        //access resource  
    }  
    catch(SQLException | RemoteException | IOException exc){  
        throw new ResourceLoadException();  
    }  
}
```

Code ist jetzt aufgeräumter.

Exceptions, die nicht (weiter) geworfen werden sollen

Bsp:

```
public class ResourceLoader {  
    public getResource(String name) throws ResourceLoadException {  
        try {  
            // try to load the resource from the database  
            ...  
        }  
        catch (SQLException e) {  
            throw new ResourceLoadException(e.toString());  
        }  
    }  
}
```

Aufrufer bekommt die Information, an der er interessiert ist.

Information, welche Exception die Ursache war. (StackTrace geht verloren)

Exception chaining

Vorheriges Beispiel führt dazu, dass wesentliche Information der ursprünglichen Exception verloren geht. Dem wirkt das Exception chaining entgegen.

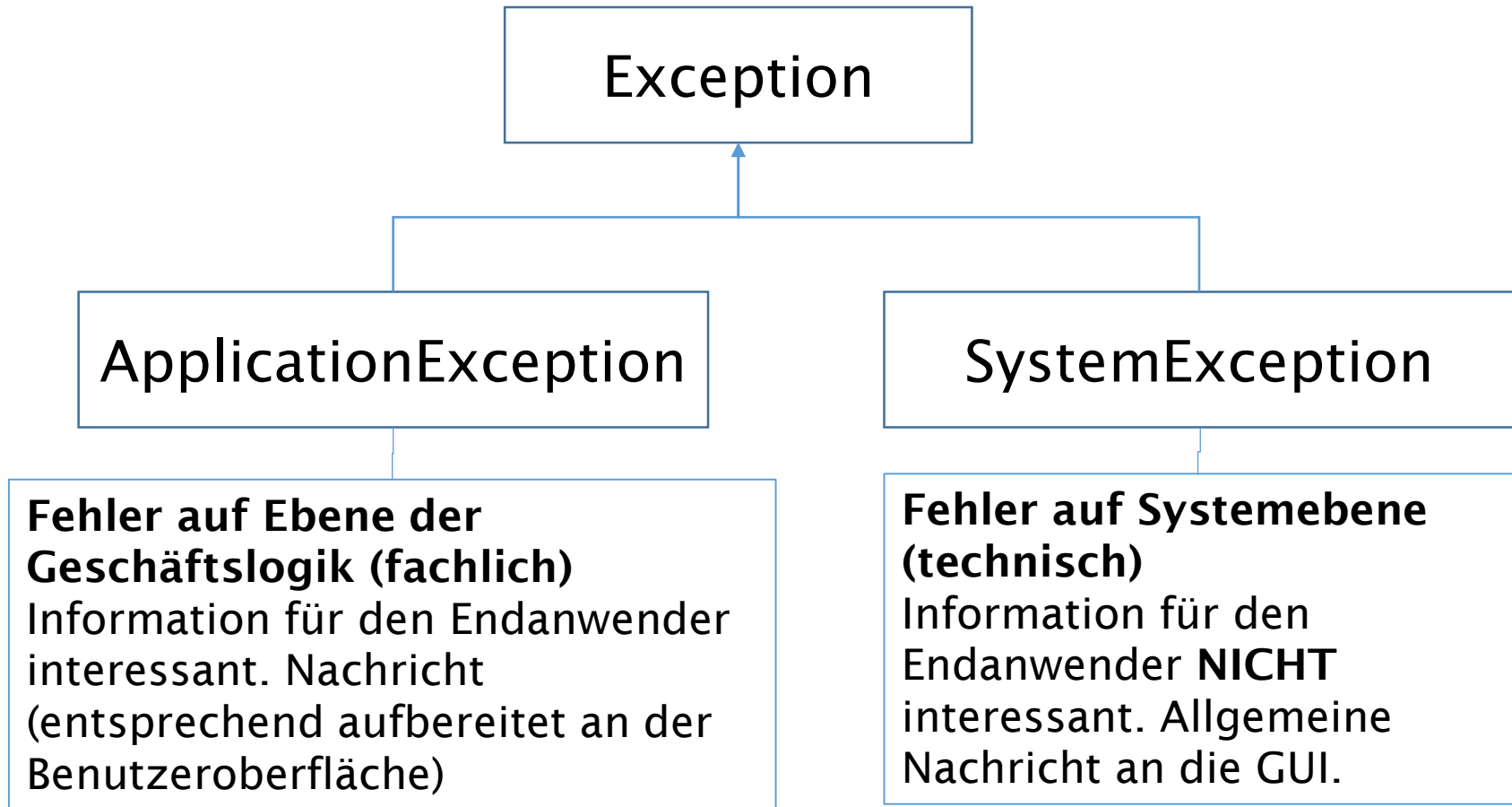
→ **Exception chaining:** Die ursprüngliche Exception wird in der neu geworfenen Exception als Ursache gespeichert.

Methoden und Konstruktoren dazu:

- Throwable getCause()
- Throwable initCause(Throwable)
- Throwable(String, Throwable)
- Throwable(Throwable)



Unterscheidung von Exceptions in der Anwendung



Exceptions für den Benutzer

- Der Benutzer ist normalerweise nicht an technischen Informationen interessiert.
- ➔ Ihm wird eine sprechende Meldung angezeigt, aus der er schließen kann, wie er weiter vorgehen soll.
- ➔ ggfs. lokalisiert
- ➔ ggfs Fehlercode für Kommunikation mit dem Support.



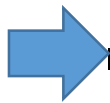
Exceptions – Best Practices

- Verwenden Sie spezifische Exceptions – keine toplevel Exceptions fangen.
- Fangen Sie die Exception erst dann, wenn Sie sie auch ordentlich behandeln können.
- Schließen Sie alle Ressourcen im finally Block (oder automatisch mit Java ARM)
- Loggen Sie alle Exceptions. Aber nur einmal!
- Verwenden Sie Multicatch um Ihren Code übersichtlicher zu machen.
- Dokumentieren Sie alle Exceptions (javadoc @throws)
- Verwenden Sie keine Exceptions für den Kontrollfluss.
- Keine Exceptions ignorieren.

Ausnahmebehandlung

- Einleitung
- Strukturierte Ausnahmebehandlung
- Java Checked Exceptions
- Exception Handling Strategie

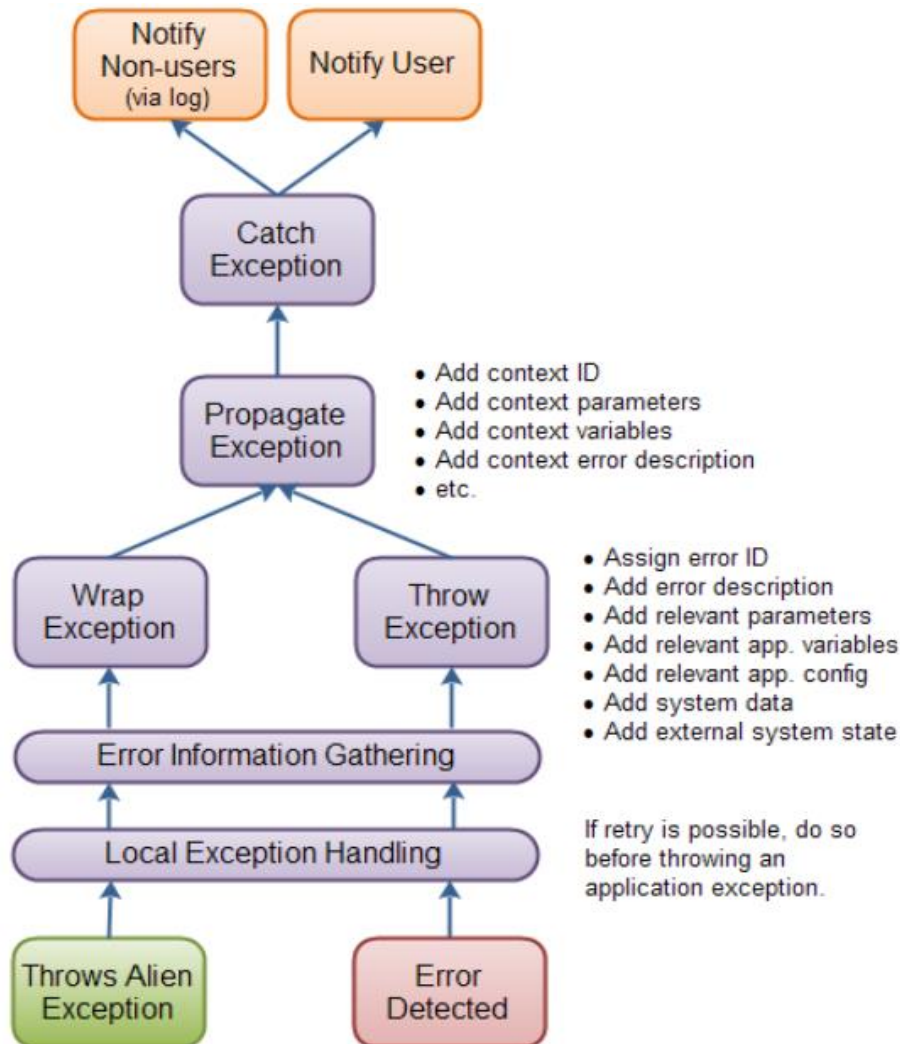




- Überblick
- Anforderungen
- Error Location und Error Context
- Fehlertypen und Reaktionen
- Strategie Elemente
- Mögliches Template



Exception Handling Strategie - Überblick



Quelle:

<http://tutorials.jenkov.com/exception-handling-strategies/overview.html>



Exception Handling Strategie

- Überblick
- Anforderungen
- Error Location und Error Context
- Fehlertypen und Reaktionen
- Strategie Elemente
- Mögliches Template



Primär:

- Überleben der Applikation
- Information der relevanten Stellen
- Fehlerdiagnose und Reproduktion

Sekundär:

- Abstraktion
- Les- und Wartbarer Code



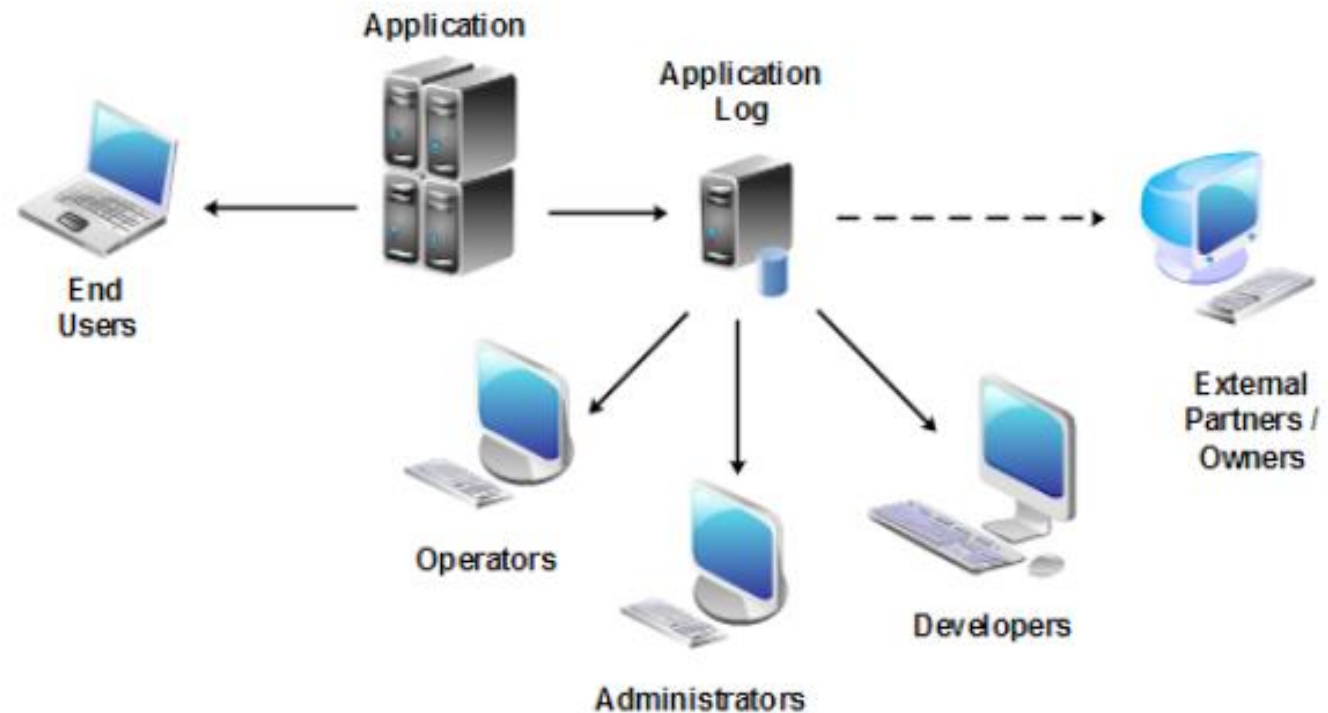
1. Fehler fangen und behandeln

- Im Idealfall: Applikation läuft weiter als wäre der Fehler nicht passiert
- Sonst. Sauberes Beenden der Applikation

2. Ressourcen sauber schließen



Information der relevanten Stellen



Wer? Typisch:

- Endnutzer
- Betreiber
- Administrator
- Entwickler
- Application owner



Diagnose: Relevante Informationen:

- Wo genau im Code ist die Exception aufgetreten?
- Kontext, in dem der entsprechende Code aufgerufen wurde
- Fehlerbeschreibung incl Variablen Werte, Zustände etc.

Reproduktion:

- Welcher Weg hat zu dem Fehler geführt?
- ➔ Welche events werden geloggt?



Abstraktion

➔ Verbergen von Informationen, die die höheren Schichten nicht zu interessieren hat.

Bsp siehe das ResourceLoaderBeispiel von oben

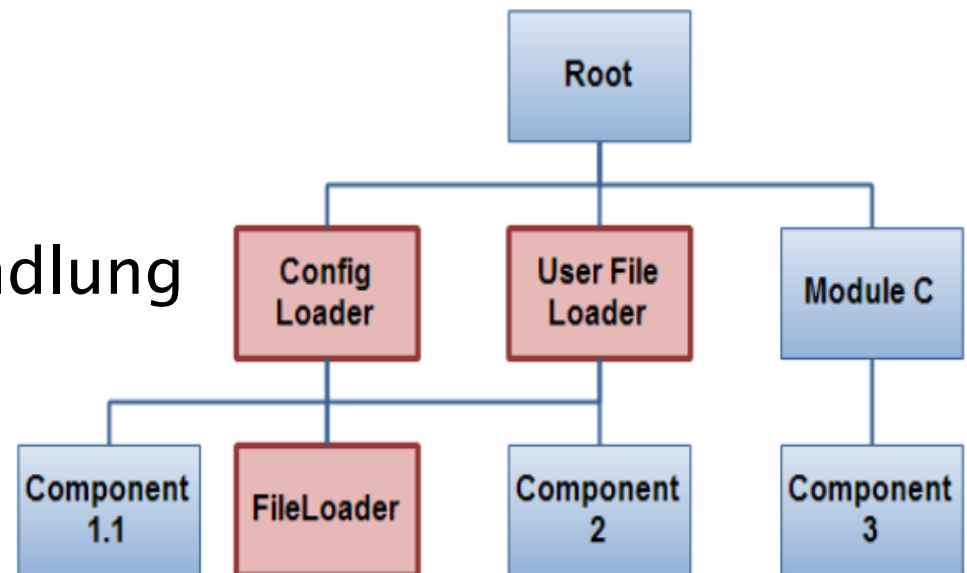


- Überblick
- Anforderungen
- Error Location und Error Context
- Fehlertypen und Reaktionen
- Strategie Elemente
- Mögliches Template



Error Location und Error Context

- Error Location:
 - Wo genau ist der Fehler aufgetreten?
- Error Context:
 - Ausführungspfad zur Errorlocation
 - Hat Auswirkung auf die Fehlerbehandlung



- Überblick
- Anforderungen
- Error Location und Error Context
- Fehlertypen und Reaktionen
- Strategie Elemente
- Mögliches Template

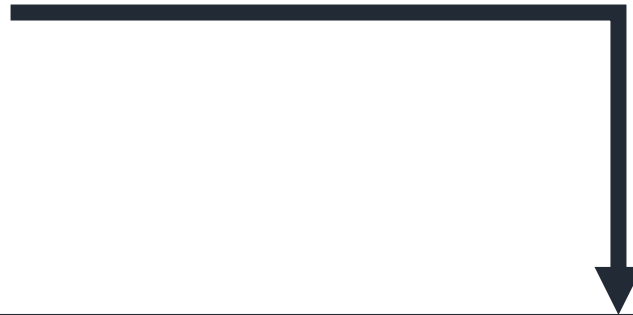


Fehlerarten und Reaktionen

Client Error

Service Error

Internal Error



Typische Reaktion:

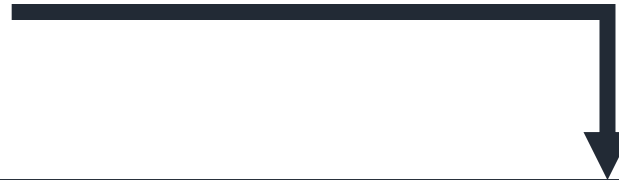
- Abbruch der aufgerufenen Aktion
- Schließen der geöffneten Ressourcen (connections, files, streams etc.)
- Freigeben der allokierten Ressourcen (memory buffers etc.).
- Nutzer benachrichtigen.
- Loggen des Fehlers

Fehlerarten und Reaktionen

Client Error

Service Error

Internal Error



Typische Reaktion:

- Abbruch der aufgerufenen Aktion oder neuer Versuch nach kurzer Zeit
- Schließen der geöffneten Ressourcen (connections, files, streams etc.)
- Freigeben der allokierten Ressourcen (memory buffers etc.).
- Nutzer benachrichtigen.
- Loggen des Fehlers
- Benachrichtigung des Betriebs, damit geeignete Maßnahmen ergriffen werden können.

Fehlerarten und Reaktionen

Client Error

Service Error

Internal Error



Typische Reaktion:

- Abbruch der aufgerufenen Aktion oder neuer Versuch nach kurzer Zeit
- Schließen der geöffneten Ressourcen (connections, files, streams etc.)
- Freigeben der allokierten Ressourcen (memory buffers etc.).
- Nutzer benachrichtigen.
- Loggen des Fehlers
- Benachrichtigung des Betriebs, damit geeignete Maßnahmen ergriffen werden können.
- Benachrichtigung der Entwickler zur Behebung.

- Überblick
- Anforderungen
- Error Location und Error Context
- Fehlertypen und Reaktionen
- ➔ ■ Strategie Elemente
- Mögliches Template

Eine Exception Handling Strategie umfasst typisch Design Überlegungen zu folgenden Elementen:

- Fehler entdecken
- Informationen zum Fehler sammeln
- Exception werfen
- Propagieren der Exception, ggfs. Kontextinformation hinzufügen
- Fangen und Reaktion:
 - Wenn möglich neuer Versuch.
 - Relevante Parteien informieren.

Fehler entdecken - Bsp

```
public void doSomething(int value, Employee targetEmployee){  
    if(value < 20){  
        throw new IllegalArgumentException("Value too low");  
    }  
    if(targetEmployee == null){  
        throw new IllegalArgumentException("Target employee was null");  
    }  
    ... do actual work.  
}
```

Fehler entdecken

NB: Die throw Clauses gehören zum „Fehler werfen“

Relevante Information: **Ursache** und **Location**

	Information	Interested Party
Cause	Technische Fehlerbeschreibung	Entwickler, Operator / Administrator
Cause	End User Fehlerbeschreibung	End User
Cause	Relevante input / Parameter / Variable Daten	End User, Entwickler, Operator / Administrator
Cause	Relevante System Daten in Konfigurationen, Datenbanken etc.	End User, Entwickler, Operator / Administrator
Cause	Relevante externe Bedingungen (ist System xy erreichbar?)	End User, Entwickler, Operator / Administrator
Location	Stack Trace	Entwickler
Location	Unique Error ID	Entwickler, Operator / Administrator

Werfen und Propagieren von Exceptions

Werfen:

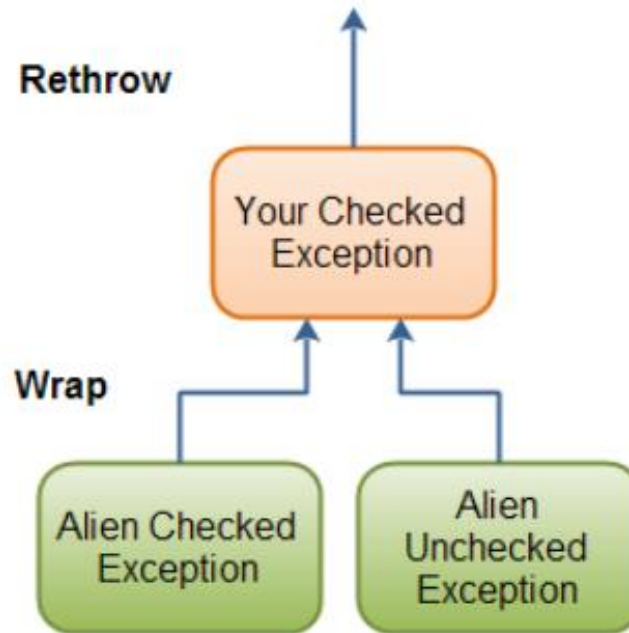
Klären: Welche Exceptions werfe ich und welche Information gebe ich der Exception mit?

➔ Siehe Folien zu Abstraktion und zum Exception Chaining



Propagieren von Exceptions

Rethrow fremder Exceptions (bei Verwendung von Checked Exceptions in der eigenen Applikation)



Exceptions Fangen und behandeln

Bsp:

```
try{  
  
    startTheWholeThing();  
  
}catch (MyAppAxception exc){  
  
    notifyUser(lookupErrorText(exc));  
    notifyNonUsers(exc);  
  
}catch(Throwable t){  
  
    notifyUserlookupErrorText(t));  
    notifyNonUsers(t);  
  
}
```



- Überblick
- Anforderungen
- Error Location und Error Context
- Fehlertypen und Reaktionen
- Strategie Elemente
- Mögliches Template



Lesbarer und Wartbarer Code

Eine saubere Strategie zur Exceptionbehandlung sorgt dafür, dass sich nicht viele identische try-catch-finally Blöcke im Code befinden.

➔ Ein Teil dieser Strategie:
Verwendung von Exception Templates, siehe z.B.

<http://tutorials.jenkov.com/java-exception-handling/exception-handling-templates.html>



Template für Exception Klasse

```
public class AppException extends Exception {  
  
    protected List<ErrorInfo> errorInfoList = new ArrayList<ErrorInfo>();  
  
    public AppException() {  
    }  
  
    public ErrorInfo addInfo(ErrorInfo info) {  
        this.errorInfoList.add(info);  
        return info;  
    }  
  
    public ErrorInfo addInfo() {  
        ErrorInfo info = new ErrorInfo();  
        this.errorInfoList.add(info);  
        return info;  
    }  
  
    public List<ErrorInfo> getErrorInfoList() {  
        return errorInfoList;  
    }  
}
```



Template für ErrorInfo

```
public class ErrorInfo {  
  
    protected Throwable cause                = null;  
    protected String    errorId              = null;  
    protected String    contextId            = null;  
  
    protected int       errorType            = -1;  
    protected int       severity             = -1;  
  
    protected String    userErrorDescription = null;  
    protected String    errorDescription    = null;  
    protected String    errorCorrection     = null;  
  
    protected Map<String, Object> parameters =  
        new HashMap<String, Object>();  
}
```

Getter und Setter sind nicht dargestellt



ErrorInfo Template

Field	Description
cause	The error cause, if an alien exception is caught and wrapped.
errorId	A unique id that identifies this error. The errorId tells what went wrong, like FILE_LOAD_ERROR. The id only has to be unique within the same context, meaning the combination of contextId and errorId should be unique throughout your application.
contextId	A unique id that identifies the context where the error occurred. The contextId tells where the error occurred (in what class, component, layer etc.). The contextId and errorId combination used at any specific exception handling point should be unique throughout the application.
errorType	The errorType field tells whether the error was caused by erroneous input to the application, an external service that failed, or an internal error. The idea is to use this field to indicate to the exception catching code what to do with this error. Should only the user be notified, or should the application operators and developers be notified too?
severity	Contains the severity of the error. E.g. WARNING, ERROR, FATAL etc. It is up to you to define the severity levels for your application.
userErrorDescription	Contains the error description to show to the user. Note: In an internationalized application this field may just contain a key used to lookup an error message in a text bundle, so the user can get the error description in his or her own language. Also keep in mind that many errors will be reported to the user with the same standard text, like "An error occurred internally. It has been logged, and the application operators has been notified". Thus, you may want to use the same user error description or error key for many different errors.
errorDescription	Contains a description of the error with all the necessary details needed for the application operators, and possibly the application developers, to understand what error occurred.
errorCorrection	Contains a description of how the error can be corrected, if you know how. For instance, if loading a configuration file fails, this text may say that the operator should check that the configuration file that failed to load is located in the correct directory.
parameters	A Map of any additional parameters needed to construct a meaningful error description, either for the users or the application operators and developers. For instance, if a file fails to load, the file name could be kept in this map. Or, if an operation fails which require 3 parameters to succeed, the names and values of each parameter could be kept in this Map.



Verwendung der AppException - Bsp

```
public byte[] loadFile(String filePath) throws AppException {  
  
    // Error Detection  
    if(filePath == null){  
  
        AppException exception = new AppException();  
  
        ErrorInfo info = exception.addInfo();  
  
        // Error Information Gathering  
        info.setErrorId("FilePathNull");  
        info.setContextId("FileLoader");  
  
        info.setErrorType(ErrorInfo.ERROR_TYPE_CLIENT);  
        info.setSeverity(ErrorInfo.SEVERITY_ERROR);  
  
        info.setErrorDescription("The file path of file to load was null");  
        info.setErrorCorrection("Make sure filePath parameter is not null.");  
  
        // Throw exception  
        throw exception;  
    }  
  
    ...  
}
```



Verwenden der AppException

- Verwendung Severity und ErrorType
Bei mehreren ErrorInfoObjekten: Welche sollen verwendet werden?
→ die letzten
- Beschreibungen für den Benutzer
 - Bei client errors: Beschreibung + Anleitung der Korrektur
 - Bei Internal errors: Standard Fehler Meldung ohne Details
- Beschreibung für Nicht-Benutzer (Entwickler etc.):
 - Z.B. Darstellung in XML Form

Exception Handling - Links

- <http://tutorials.jenkov.com/exception-handling-strategies/index.html>
- <http://tutorials.jenkov.com/java-exception-handling/index.html>
- <http://www.java-tutorial.org/exception-handling.html>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>
- <http://www.javaworld.com/article/2075476/core-java/exceptional-practices--part-1.html>
- <http://www.journaldev.com/1696/java-exception-handling-tutorial-with-examples-and-best-practices>

