

- Einführung
- Software Fehler
- Konstruktive Qualitätssicherung
- Software Test
- Statische Analyse

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

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

Richtlinien regeln den Gebrauch einer Programmiersprache über die eigenen syntaktischen und semantischen Regeln hinaus.

Motivation:

- Vereinheitlichung
- Fehlerreduktion

Software Richtlinien

- [Notationskonventionen](#)
- [Sprachkonventionen](#)

Software Richtlinien

- [Notationskonventionen](#)
- [Sprachkonventionen](#)

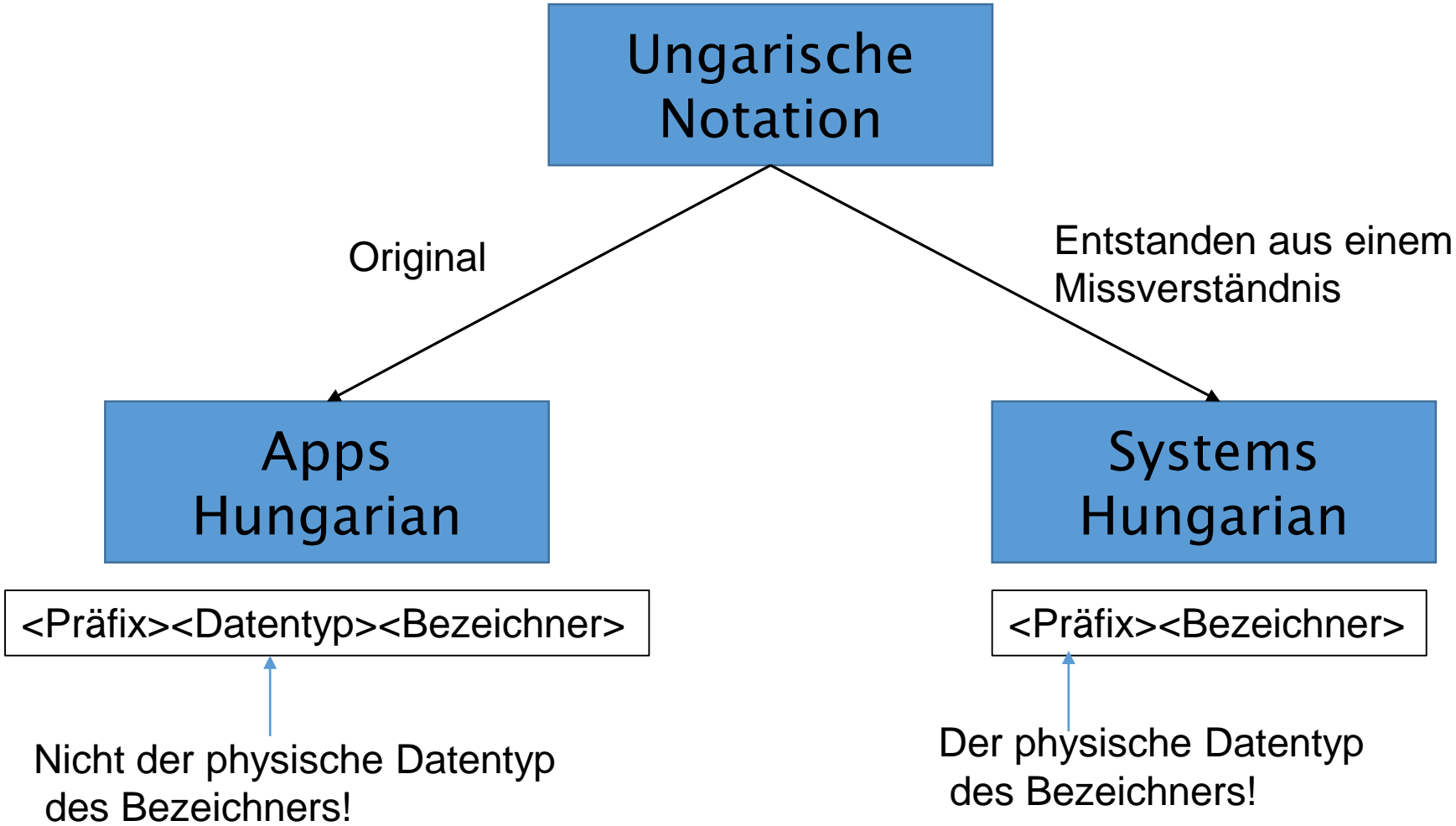
Notationskonventionen werden auf verschiedenen Ebenen definiert (Projekt, Sprache, Betriebssystem,..)

Typischerweise betroffen:

- Auswahl und Schreibweise von Bezeichnern
- Einrückungen, Verwendung von Leerzeichen
- Aufbau von Kontrollstrukturen
- Dokumentation

- **Pascal Case:** Bezeichner, sowie jedes enthaltene Wort startet mit Großbuchstaben.
- **Camel Case:** Bezeichner startet mit Kleinbuchstaben, jedes weitere Wort groß.
- **Uppercase:** Komplett in Großbuchstaben
- **Lowercase:** Komplett in Kleinbuchstaben

Ungarische Notation



Jeder Variablenname besteht aus zwei Teilen:

- **Präfix:** abkürzende Schreibweise für den Datentyp
- **Qualifier:** frei gewählter Name

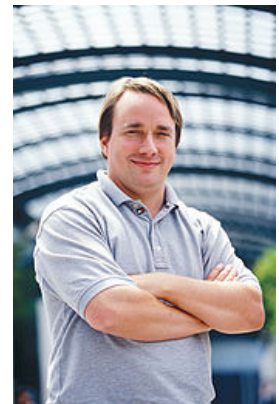
Bsp:

Button butOK;
ListBox lbColorSelector;
CheckBox cbRemindMe;

Problem: Verstoß
gegen das Single
Source Prinzip

Linus Torvalds:

Encoding the type of a function into the name (so called Hungarian notation) is brain-damaged – the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder Microsoft makes buggy programs.



Präfix: Nimmt Bezug auf die Funktion der Variablen

Beispiele:

i – index in einem Array

p – Pointer

h – handle (pointerpointer)

c – Anzahl von Elementen (z.B. in einem Array)

rg – Ein durch integer indiziertes Array

gr – Verbund von Variablen (z.B. bei einer struct)

Siehe z.B. https://de.wikipedia.org/wiki/Ungarische_Notation

Datentyp: Einige Basetypes definiert

Beispiele:

f – Boolescher Datentyp (Bedeutung- nicht der physische Datentyp)

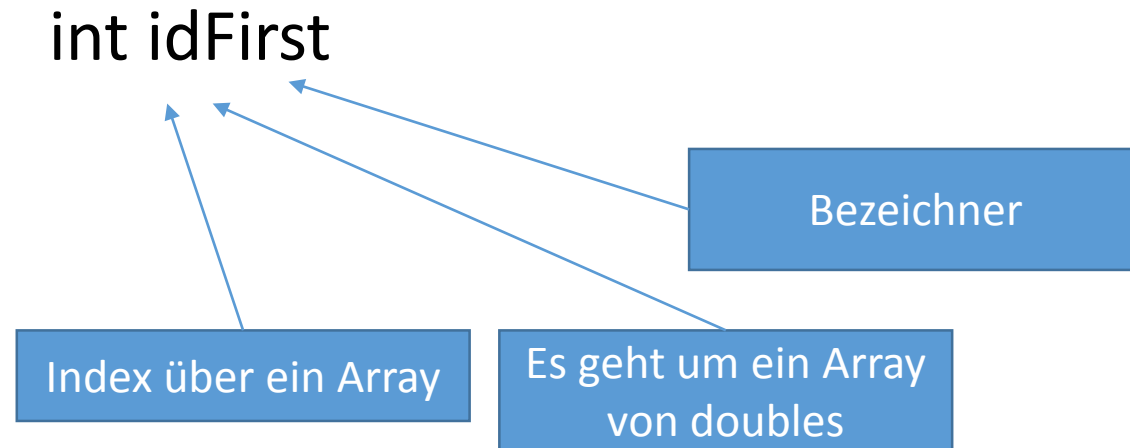
ch – ein Ein-Byte Zeichen

sz – ein Null terminierter String

Siehe z.B.

https://de.wikipedia.org/wiki/Ungarische_Notation

Beispiel



Coding Style

Die folgenden Coding Styles enthalten Notationskonventionen, gehen aber deutlich darüber hinaus.

C# Coding Style:

- siehe <https://msdn.microsoft.com/de-de/library/ff926074.aspx>

Java Coding Style: verschiedene Varianten: z.B.

- <https://google.github.io/styleguide/javaguide.html>
- <http://www.ambyssoft.com/essays/javaCodingStandards.html>
- Siehe auch: *Michael Inden, Der Weg zum Java Profi, Seite 1160*

GNU Coding Standards for C:

- https://www.gnu.org/prep/standards/html_node/Writing-C.html#Writing-C

Software Richtlinien

- [Notationskonventionen](#)
- [Sprachkonventionen](#)

Bisher: Layout und Syntax,

Jetzt: Semantische Besonderheiten einer Sprache

Bsp: MISRA-C: Programmierstandard ➔ <http://www.misra-c.com>

(MISRA: Motor Industry Software Reliability Association)

Der MISRA-C-Programmierstandard definiert eine Untermenge des Sprachumfangs von C, d.h. er umfasst Richtlinien die zu einer Qualitätssteigerung (insbesondere der Softwarequalitätsaspekte der Zuverlässigkeit und Wartbarkeit) in der Software-Entwicklung führen sollen.

Weite Verbreitung von C gerade auch in sicherheitskritischen Bereichen,

ABER

- Verhalten teils undefiniert.
- C macht es leicht, die Sprache falsch zu gebrauchen.
- C erlaubt schwer verständliche Konstrukte.
- C überlässt dem Entwickler die Fehlerbehandlung zur Laufzeit.

➔ **MISRA unterstützt Entwickler bei der Entwicklung speziell sicherheitskritischer Systeme**

Vision von MISRA

The MISRA C Guidelines define a subset of the C language in which the opportunity to make mistakes is either removed or reduced.

1. Empfehlungen zu

- Tool Selection
- Projekt Aktivitäten
- Implementierung der MISRA Compliance

2. Guidelines (Directives and Rules)

Beispiele für MISRA Richtlinien

- Konstanten in einem vorzeichenlosen Kontext müssen mit einem U-Suffix versehen werden.
- Variablen vom Typ float (Gleitkommazahlen) sollen nicht mit den Vergleichsoperatoren == oder != getestet werden.
- goto soll nicht verwendet werden.
- magic numbers vermeiden und stattdessen sinnvoll benannte Konstanten verwenden: #define MAXSIZE 12.
- Division durch null verhindern: if (b!=0) a/=b;
- Compilerunabhängigkeit sicherstellen, z. B. shiften neg. Zahlen: $-3 \ll 4 \implies -3 * (1 \ll 4)$
- Operatorrangfolgen sind nicht trivial, daher Klammern verwenden: $(a \ \&\& \ b \ || \ c) \implies ((a \ \&\& \ b) \ || \ c)$.
- Rekursion darf in keiner Form auftreten (weder indirekt noch direkt).

MISRA Regelkategorien

Regeln	Kategorie	Regeln	Kategorie
(1.1) - (1.5)	Übersetzungsumgebung	(12.1) - (12.13)	Ausdrücke
(2.1) - (2.4)	Spracherweiterungen	(13.1) - (13.7)	Kontrollstrukturen
(3.1) - (3.6)	Dokumentation	(14.1) - (14.10)	Kontrollfluss
(4.1) - (4.2)	Zeichensatz	(15.1) - (15.5)	Switch-Konstrukt
(5.1) - (5.7)	Bezeichner	(16.1) - (16.10)	Funktionen
(6.1) - (6.5)	Datentypen	(17.1) - (17.6)	Pointer und Arrays
(7.1)	Konstanten	(18.1) - (18.4)	Struct und Union
(8.1) - (8.12)	Deklarationen und Definitionen	(19.1) - (19.17)	Präprozessor
(9.1) - (9.3)	Initialisierung	(20.1) - (20.12)	Standardbibliotheken
(10.1) - (10.6)	Typkonversion (Arithmetik)	(21.1)	Laufzeitfehler
(11.1) - (11.5)	Typkonversion (Pointer)		

Auszug aus dem MISRA Regelsatz

Regel	Beschreibung
(1.4)	"The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers."
(2.2)	"Source code shall only use <code>/* . . . */</code> style comments."
(3.2)	"The character set and the corresponding encoding shall be documented."
(4.2)	"Trigraphs shall not be used."
(5.1)	"Identifiers (internal or external) shall not rely on the significance of more than 31 characters."
(6.4)	"Bit fields shall only be defined to be of type unsigned int or signed int ."
(7.1)	"Octal constants (other than zero) and octal escape sequences shall not be used."
(8.5)	"There shall be no definitions of objects or functions in a header file."
(9.1)	"All automatic variables shall have been assigned a value before being used."
(10.6)	"A U-suffix shall be applied to all constants of unsigned type."
(11.3)	"A cast should not be performed between a pointer type and an integral type."
(12.3)	"The sizeof -Operator shall not be used on expressions that contain side effects."
(13.3)	"Floating-point expressions shall not be tested for equality or inequality."
(14.1)	"There shall be no unreachable code."
(15.3)	"The final clause of a switch statement shall be the default class."
(16.1)	"Functions shall not be defined with variable numbers of arguments."
(17.4)	"Array indexing shall be the only allowed form of pointer arithmetic."
(18.4)	"Unions shall not be used."
(19.6)	" #undef shall not be used."
(20.12)	"The time handling functions of library <code><time.h></code> shall not be used."
(21.1)	"Minimisation of run-time failures shall be ensured by the use of at least one of: a) static analysis tools/techniques; b) dynamic analysis tools/techniques; c) explicit coding of checks to handle run-time faults."

- Sprachkonventionen in Java Programmen
 - In Firmen
 - In Projekten
- Beispiel. Google Java Style:
 - <https://google.github.io/styleguide/javaguide.html>

Empfehlung: keine neuen Styles erfinden,
sondern bestehende verwenden.

Soziale Aspekte bei der Durchsetzung von Konventionen durch Codereviews

- Diejenigen, die die Konventionen am besten kennen, sind die Buhmänner.
- Der Gereviewte fühlt sich persönlich angegriffen.
- Der Reviewer will deswegen nichts mehr anmerken.

➔ Empfehlungen zum Vorgehen siehe nächste Seite

Durchsetzung von Konventionen

- Alle Beteiligten an der Erarbeitung der Konventionen mitarbeiten lassen.
- Konventionen gemeinsam weiterentwickeln.
- Begründete Ausnahmen akzeptieren (und dokumentieren).
- Toolgestützt prüfen (sowohl am Arbeitsplatz als auch beim Build Prozess).
- Regeln *zu Beginn* vereinbaren.
- Bestehende Regeln nur mit Bedacht ändern.

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

Typisierte Sprachen (Sprachen, die ein Typsystem besitzen) kategorisieren Daten und fassen gleichartige Objekte zu einem Datentyp zusammen.

Heute: Die modernen Programmiersprachen unterstützen ausgefeilte Typsysteme.

Früher (Lisp, Fortran, Cobol,..) oftmals nur rudimentäre Typsysteme.

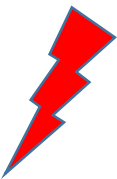
Typsystem:

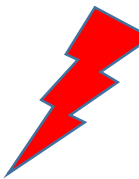
Der Teil eines Compilers oder einer Laufzeitumgebung, der ein Programm auf die korrekte Verwendung der Datentypen überprüft.

Nutzen:

1. Frühzeitiges Erkennen von Inkonsistenzen
2. Verständlichkeit des Programms

Bsp (zu 1.): Syntaktische Bildung von unsinnigen Ausdrücken wird bereits zur Compile Zeit verhindert:

&42 

"41"++ 

Nutzen der Verwendung von Datentypen

- Frühzeitige Erkennung von Inkonsistenzen
- Verständlichkeit der Codes/Lesbarkeit
- ➔ Letztlich Vermeidung von Laufzeitfehlern.

Bestandteile eines Typsystems

- Typen (entweder in der Sprache verankert oder mittels Typdefinitionen erzeugt).
- Möglichkeit, Variablen, Funktionsparameter etc. mit einem bestimmten Typ zu deklarieren.
- Regeln, nach denen die Werte von Ausdrücken einem bestimmten Typ zugeordnet werden.
- Regeln zur Prüfung der Zuweisungskompatibilität von Typen.
- Optional weitere Sprachbestandteile (typbezogene Operatoren, Reflection etc.)

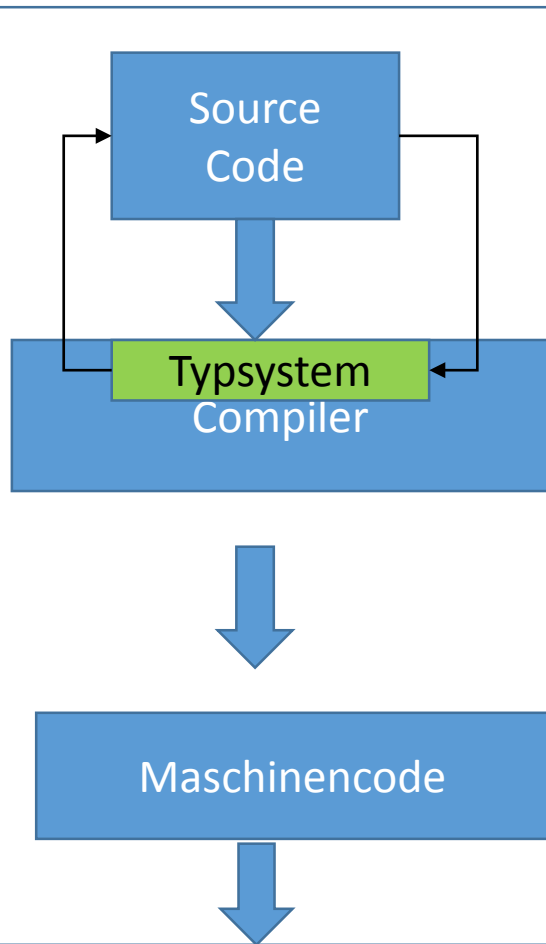
Aufgaben eines Typsystems

- Erkennen von Typverletzungen
- Typumwandlungen

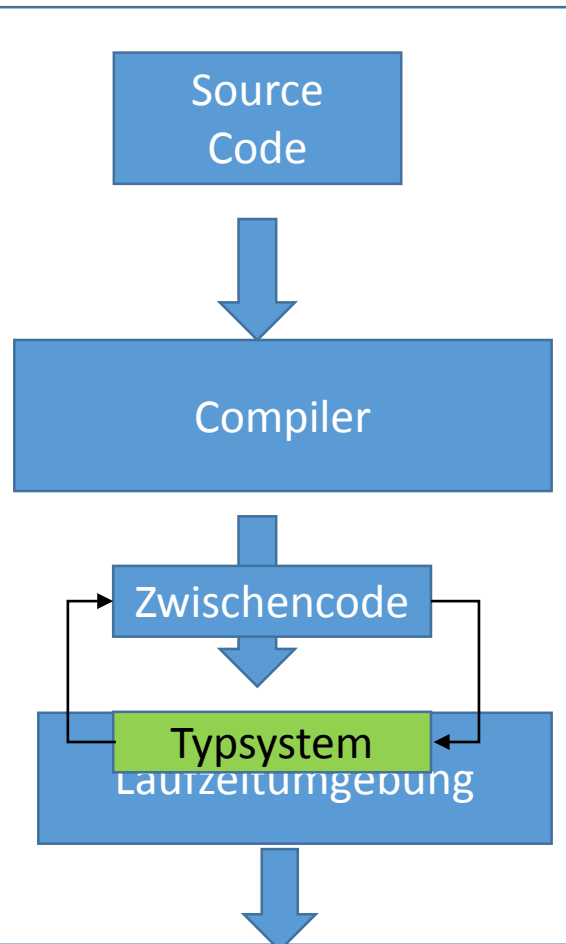
Typprüfung

- Statische Typprüfung (static type checking): Prüfung zur Compilezeit.
- Dynamische Typprüfung (dynamic type checking): Prüfung während der Programmausführung.

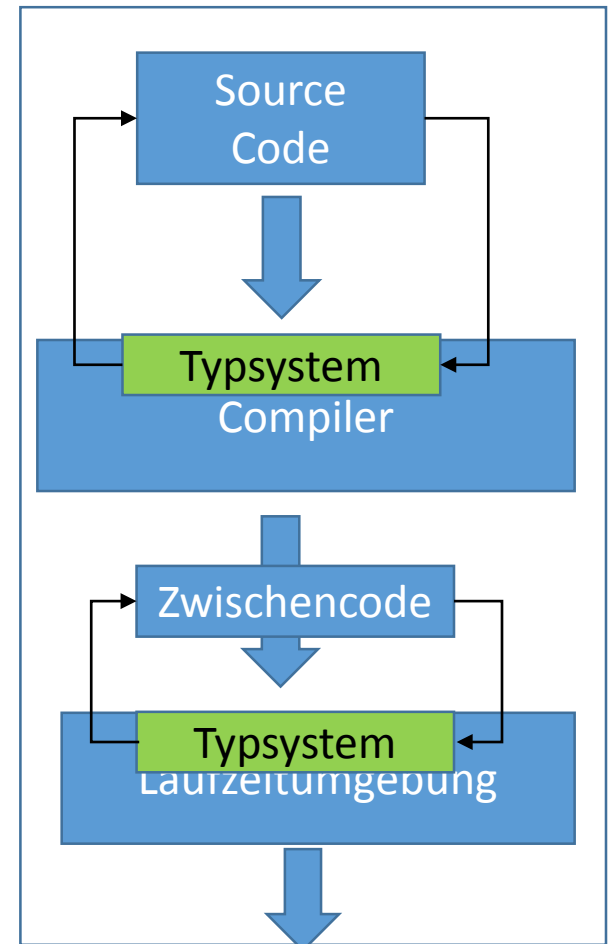
Typprüfung



Statische Typprüfung



Dynamische Typprüfung



Kombinierte Typprüfung

Dynamische Typprüfung (die zwei rechten Bilder der Vorfolie) erhöht die Sicherheit des Programms und kostet Performance.

- ➔ Dynamische Typprüfung wird hauptsächlich bei Programmiersprachen der höheren Abstraktionsebene.
- ➔ Verlass auf statische Typprüfung bei hardwarenaher Programmierung.

Einteilung der Typisierung nach Prinzipien:

- **Statische Typisierung (static typing):**
Variablen werden zusammen mit ihrem Datentyp im Quelltext deklariert.
Bsp: C, C#, Java
- **Dynamische Typisierung (dynamic typing):**
Variablen sind nicht an einen bestimmten Datentyp gebunden.
Bsp: Smalltalk, PHP, Python

PHP: dynamic typing

```
<?php
    $mystring = "12";
    $myinteger = 20;
    print $mystring + $myinteger;
?>
```

PHP konvertiert den String „12“ in den Integer 12 → Ergebnis ist 32.

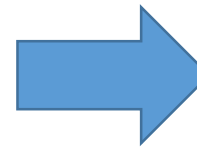
Aber:

\$mystring = “Hans“;

Konvertierung in Integer 0 → Ergebnis 20 ohne Fehlermeldung.

PHP static typing

```
<?php
    $bool = true;
    print "Bool is set to $bool\n";
    $bool = false;
    print "Bool is set to $bool\n";
?>
```



```
Bool is set to 1
Bool is set to 0
```

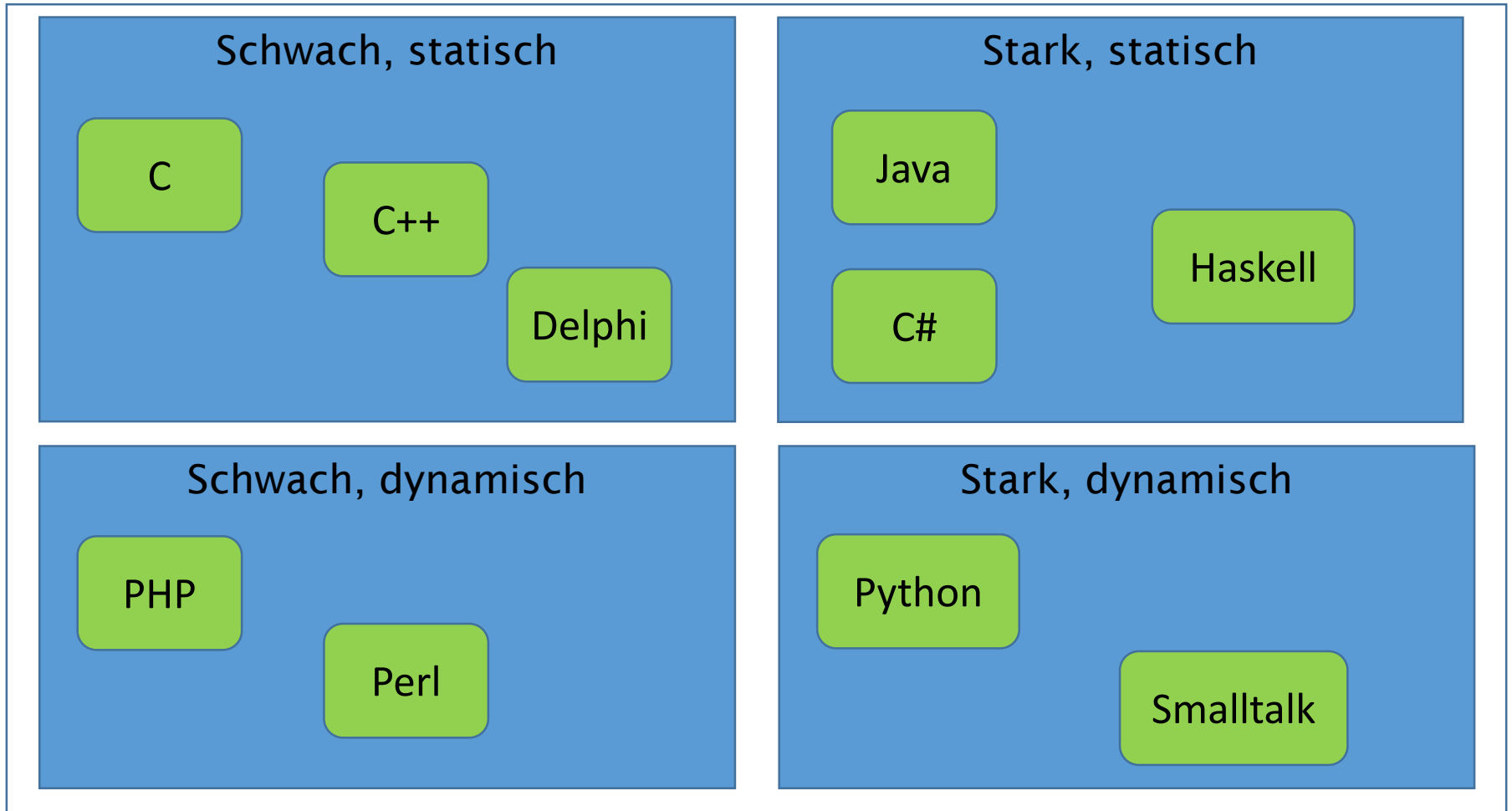
Lösung: expliziter Cast:

```
<?php
    $bool = true;
    print "Bool is set to $bool\n";
    $bool = false;
    print "Bool is set to ";
    print (int)$bool;
?>
```

- **Schwache Typisierung:**
Datentyp eines Objekts darf beliebig uminterpretiert werden. (mithilfe von Typecasts).
- **Starke Typisierung:**
Sicherstellung, dass der Zugriff auf alle Objekte und Daten stets typkonform erfolgt.

Diese Begriffe sind nicht absolut zu verstehen.

Programmiersprachen und Typsysteme



Klassifikation verschiedener Programmiersprachen
anhand ihrer Typsysteme

Beispiel in Java

```
public static Integer addMax(Object a, int b, int c){  
    int max= Math.max(b,c);  
    int sum = (int) a + max;  
  
    return sum;  
}
```

ClassCastException,
falls a nicht gecastet
werden kann.

Implizite Konvertierung
zu Integer.

NullPointerException,
falls a nicht definiert
ist.

Grenzen, Fallstricke: Bsp Java

```
public class FooBar {  
  
    public static void main(String[] args) {  
        // create list  
        List list = new ArrayList();  
        list.add(new String("foo"));  
        list.add(new String("bar"));  
        list.add(new Integer(42));  
  
        //print list  
        Iterator iterator = list.iterator();  
        while(iterator.hasNext()){  
            String item = (String)iterator.next();  
            System.out.println(item);  
        }  
    }  
}
```

foo
bar

Exception in thread "main" [java.lang.ClassCastException](#): java.lang.Integer cannot be cast to java.lang.String
at demo.FooBar.main([FooBar.java:19](#))

Lösung: Generics

```
public class FooBar_2 {  
  
    public static void main(String[] args) {  
        // create list  
        List<String> list = new ArrayList();  
        list.add(new String("foo"));  
        list.add(new String("bar"));  
        list.add(new Integer(42));  
  
        //print list  
        Iterator iterator = list.iterator();  
        while(iterator.hasNext()){  
            String item = (String)iterator.next();  
            System.out.println(item);  
        }  
    }  
}
```

Compile Fehler !

C-Bsp

```
#include <stdio.h>

int speed_limit()
{
    /*speed limit in mph*/
    int limit = 65;
    return limit;
}

int main()
{
    /*speed limit in km/h*/
    int limit;
    limit = speed_limit();

    printf("Speed limit = %d km/h \n", limit);

    getchar();

    return 1;
}
```

C-Bsp- Verbesserung

```
#include <stdio.h>
```

```
typedef int kmh;
```

```
typedef int mph;
```

```
mph speed_limit()
```

```
{  
    /*speed limit in mph*/  
    int limit = 65;  
    return limit;  
}
```

```
int main()
```

```
{  
    /*speed limit in km/h*/  
    kmh limit;  
    limit = speed_limit();  
  
    printf("Speed limit = %d km/h \n", limit);  
  
    return 1;  
}
```

Besser,
aber noch nicht gut

C-Bsp, Version 3

Fehler beim Kompilieren!

```
#include <stdio.h>

struct kmh
{
    int value;
};

struct mph
{
    int value;
};

typedef struct kmh kmeter_pro_stunde;
typedef struct mph miles_per_hour;

miles_per_hour speed_limit()
{
    /*speed limit in mph*/
    miles_per_hour limit;
    limit.value = 65;
    return limit;
}

int main()
{
    /*speed limit in km/h*/
    kmeter_pro_stunde limit;
    limit = speed_limit();

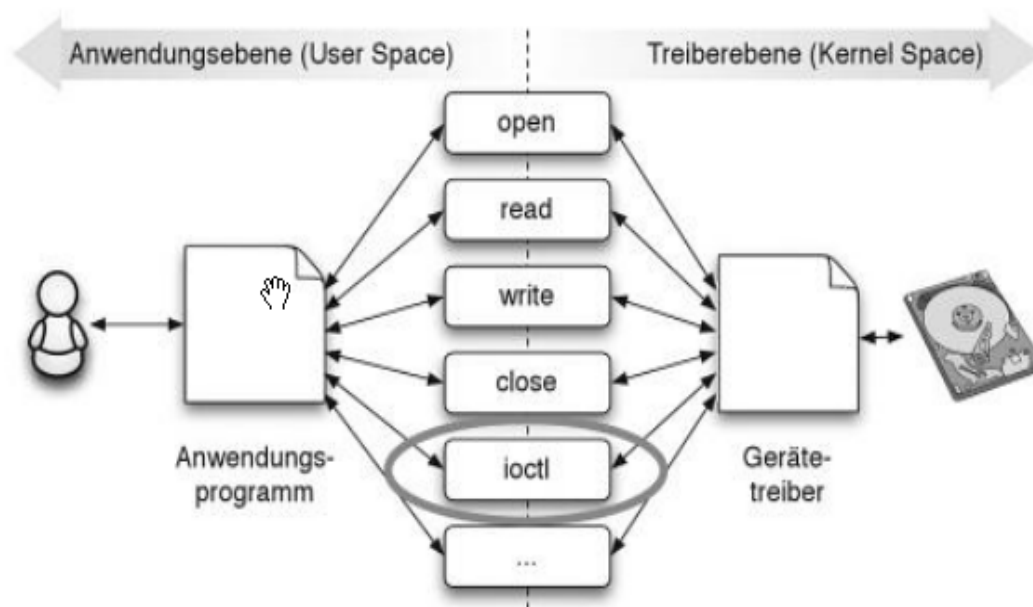
    printf("Speed limit = %d km/h \n", limit);

    return 1;
}
```

Generische Schnittstellen

In manchen Anwendungsfällen ist eine strenge Typsicherheit nicht gewünscht.

Bsp: Kommunikation mit Gerätetreibern über eine einheitliche Schnittstelle (für verschiedenste Geräte)



Generische Schnittstellen

Funktion `int ioctl(int file, int request, ... /*argument list*/);`

Variable Parameterliste ohne Definition der Typen!

Erhöhung der Typsicherheit durch: Verwendung von Container Typen.

Bsp: VARIANT (Bsp in C siehe nächste Seite) als Container für eine Vielzahl von fest definierten Datentypen.

Variant als selbstbeschreibender Datentyp

variant.c

```
typedef struct {  
    VARTYPE vt;  
    WORD    wReserved1;  
    WORD    wReserved2;  
    WORD    wReserved3;  
    union {  
        LONG    lVal;  
        BYTE    bVal;  
        SHORT   iVal;  
        FLOAT   fltVal;  
        DOUBLE   dblVal;  
        LONG*    plVal;  
        BYTE *   pbVal;  
        SHORT *  piVal;  
        LONG *   plVal;  
        FLOAT *  pfltVal;  
        DOUBLE * pdblVal;  
        ...  
    } value;  
} VARIANT;
```

vt hält die
Typinformation

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

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

Idee von DbC

- Definition von formalen und messbaren Vereinbarungen für Schnittstellen zwischen Modulen.
- Überprüfung der Vereinbarungen
- **Design** by Contract: Vor der Implementierung werden die Vereinbarungen festgeschrieben.

Gründer: Betrand Meyer im Zusammenhang mit Entwicklung der Programmiersprache Eiffel.

Es wird ein **Vertrag** zwischen Aufrufer und Aufgerufenem vereinbart, der Vor- und Nachbedingungen sowie Invarianten festlegt.

The Design by Contract theory, then, suggests associating a specification with every software element. These specifications (or contracts) govern the interaction of the element with the rest of the world.

Quelle: <https://www.eiffel.com/values/design-by-contract/introduction/>

Vorbedingungen:

Zusicherungen, die der Aufrufer zu beachten hat.

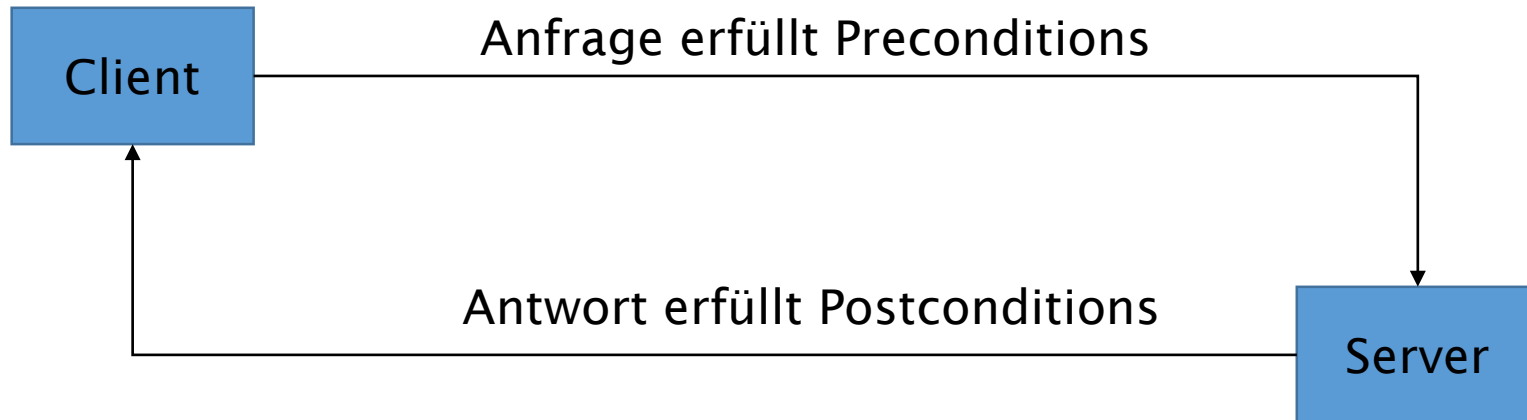
Nachbedingungen:

Zusicherungen, die der Aufgerufene zu beachten hat.

Invarianten:

Gesundheitszustand der Klasse. (logische Aussagen, die für alle Instanzen einer Klasse über den gesamten Objektlebenszyklus hinweg gelten)

DbC – Pre- und Postconditions



Verletzung der Constraints führt zu Programmabbruch.

- Steigerung der Qualität durch genaue Spezifikation der Schnittstellen
- Contracts als Dokumentation
- Unterstützung beim Testen/Fehlerfinden.

In manchen Sprachen nativ verankert: Eiffel, D

Design by Contract in Java:

- Nicht in der Sprache verankert.
- Konzept anwenden: javadoc (oder Äquivalent) nutzen um Vor-, Nachbedingungen und Invarianten zu definieren.
- Rudimentär: Arbeiten mit assert.
- Möglichkeit mit frameworks das Konzept zu implementieren (z.B. <http://www.valid4j.org/>)
- Bsp: [Contracts for Java](#)

Contract for Java

(<https://github.com/nhatminhle/cofoja>):

```
@Requires("x >= 0")  
@Ensures("result >= 0")  
static double sqrt(double x);
```

Default: keine Auswirkung,

Contracts angeschalten: Spezifische Laufzeitexceptions bei Verletzungen.
(PreconditionError, PostConditionError, InvariantError)

- PreConditions können in den vererbten Methoden gelockert werden.
- PostConditions können strenger gemacht werden.
- Invariants können strenger gemacht werden.
- Zusätzliche Einschränkungen sind bei den abgeleiteten Methoden möglich.

- 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

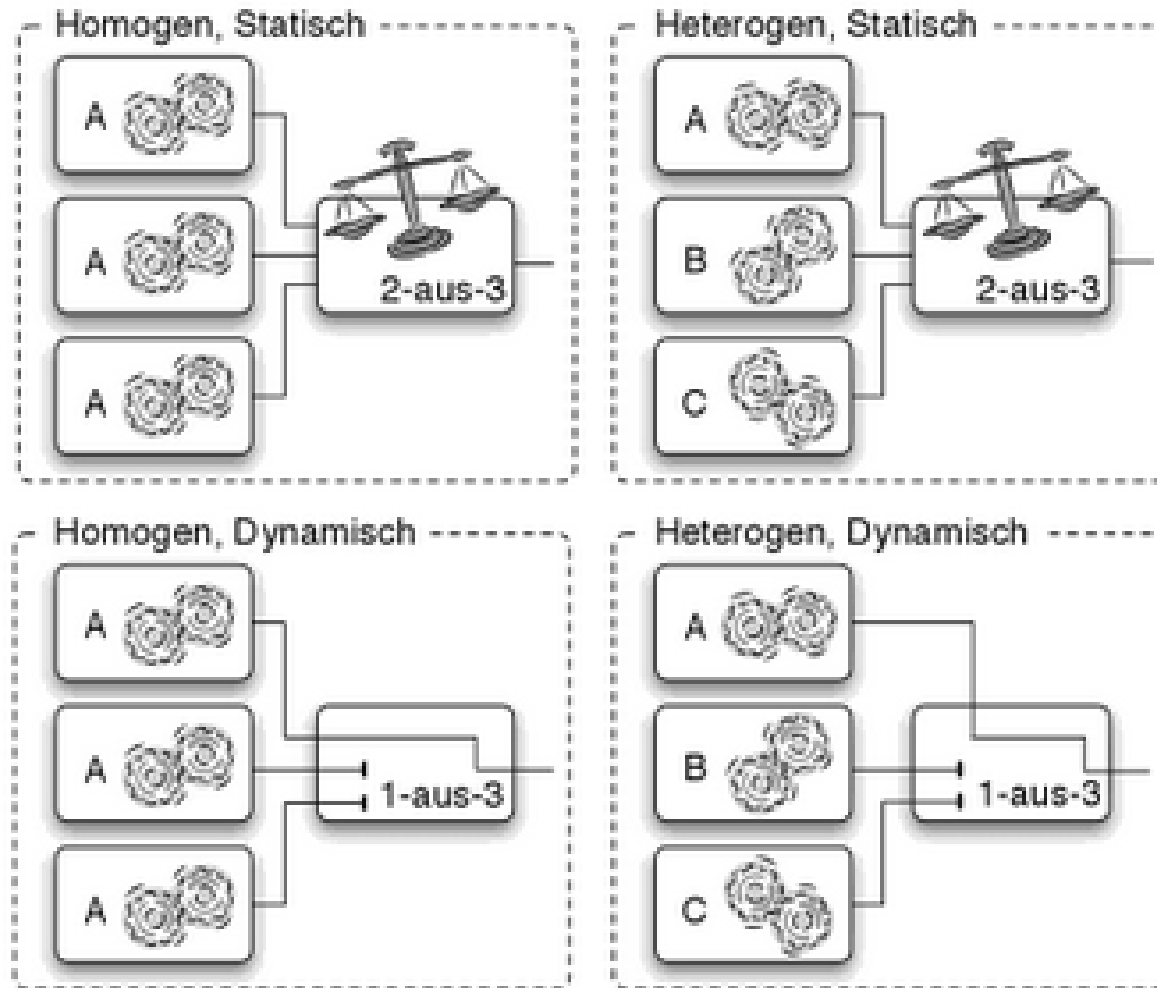
Ausprägungen:

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

Ausprägungen:

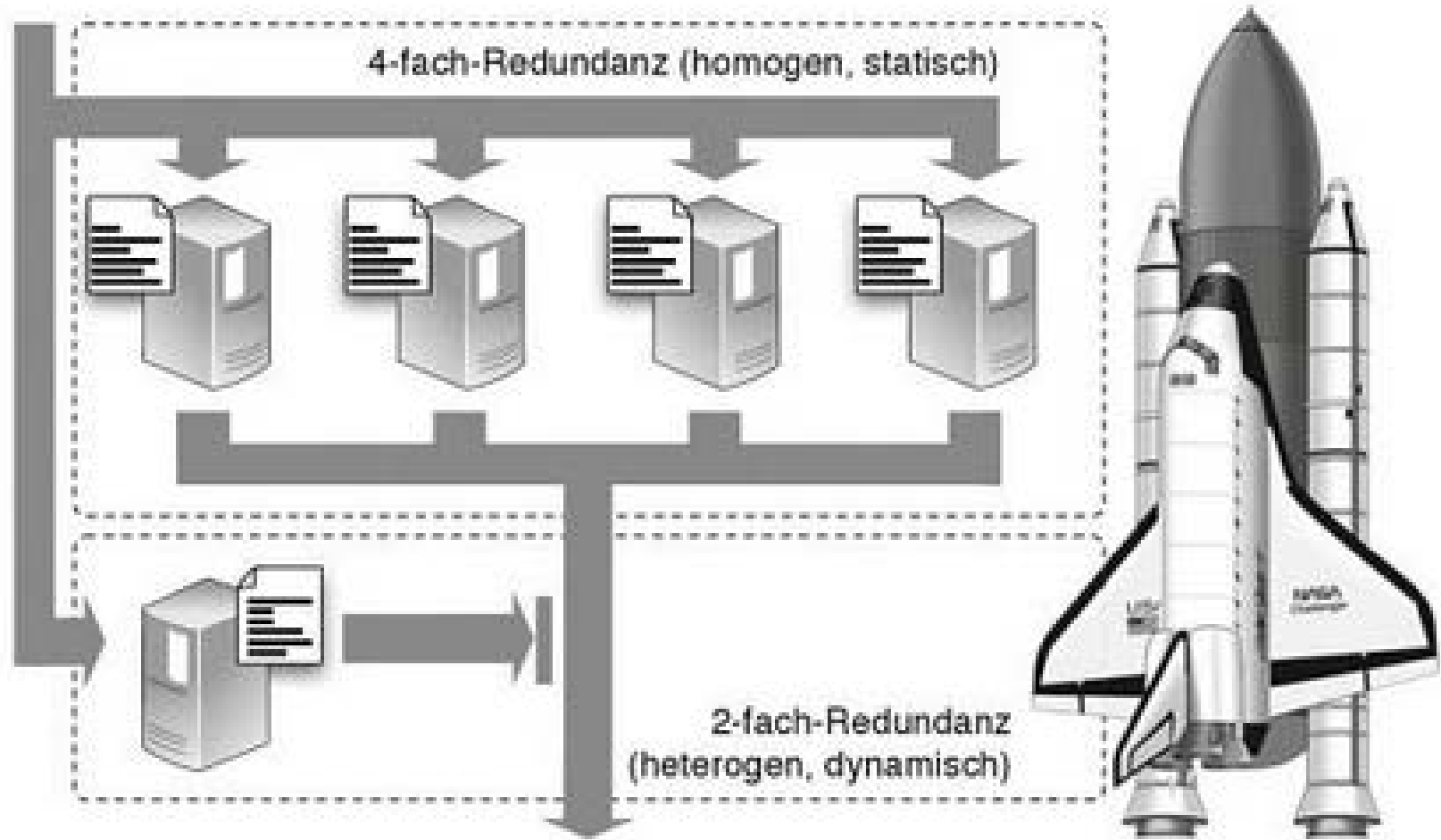
- Statische Redundanz (alle Komponenten aktiv, Ergebnisse werden durch Voter verglichen)
- Dynamische Redundanz (Ersatzkomponenten, die bei Bedarf aktiviert werden)

Homogene und Heterogene Redundanz



Quelle: D. Hoffmann:
Software Qualität

SW Redundanz, Bsp. Space Shuttle



- 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

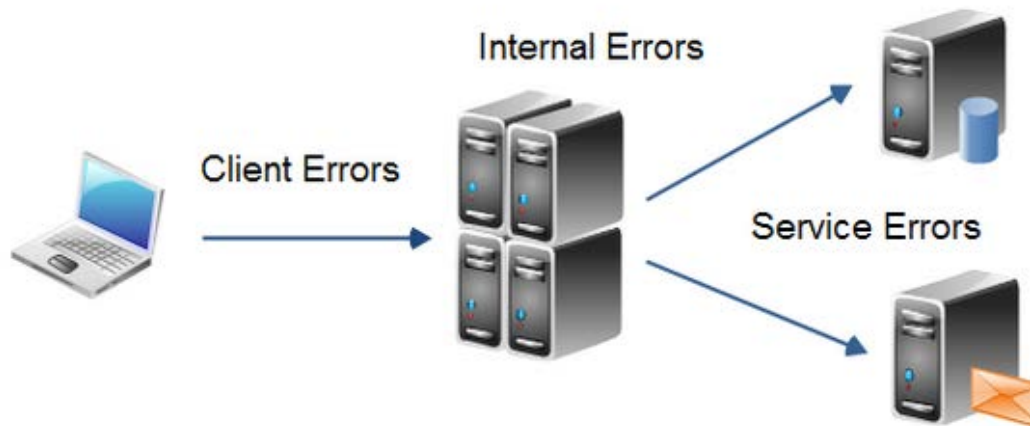
- Software Redundanz
- Selbstüberwachende Systeme
- Ausnahmebehandlung

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

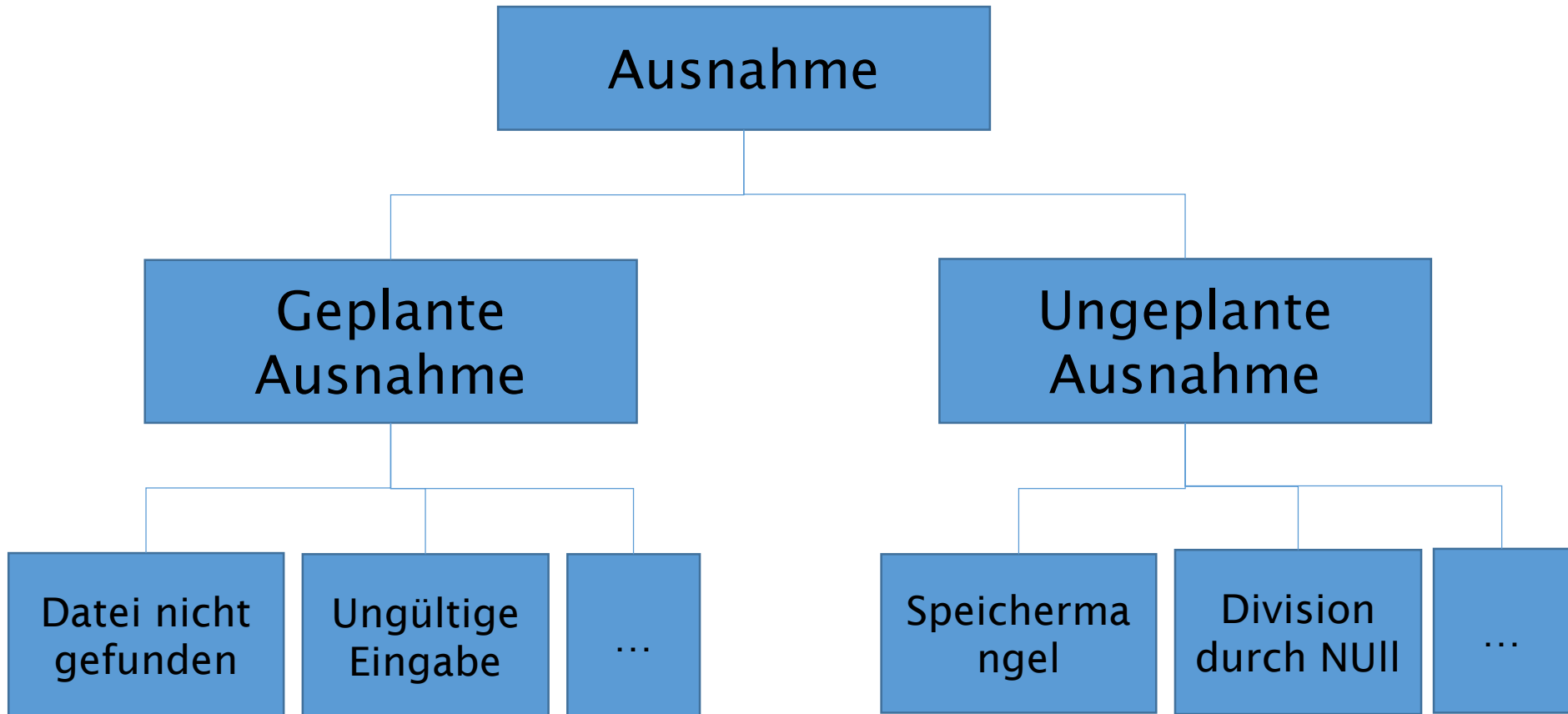
Service Errors:

- 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



- 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

```
void 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

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

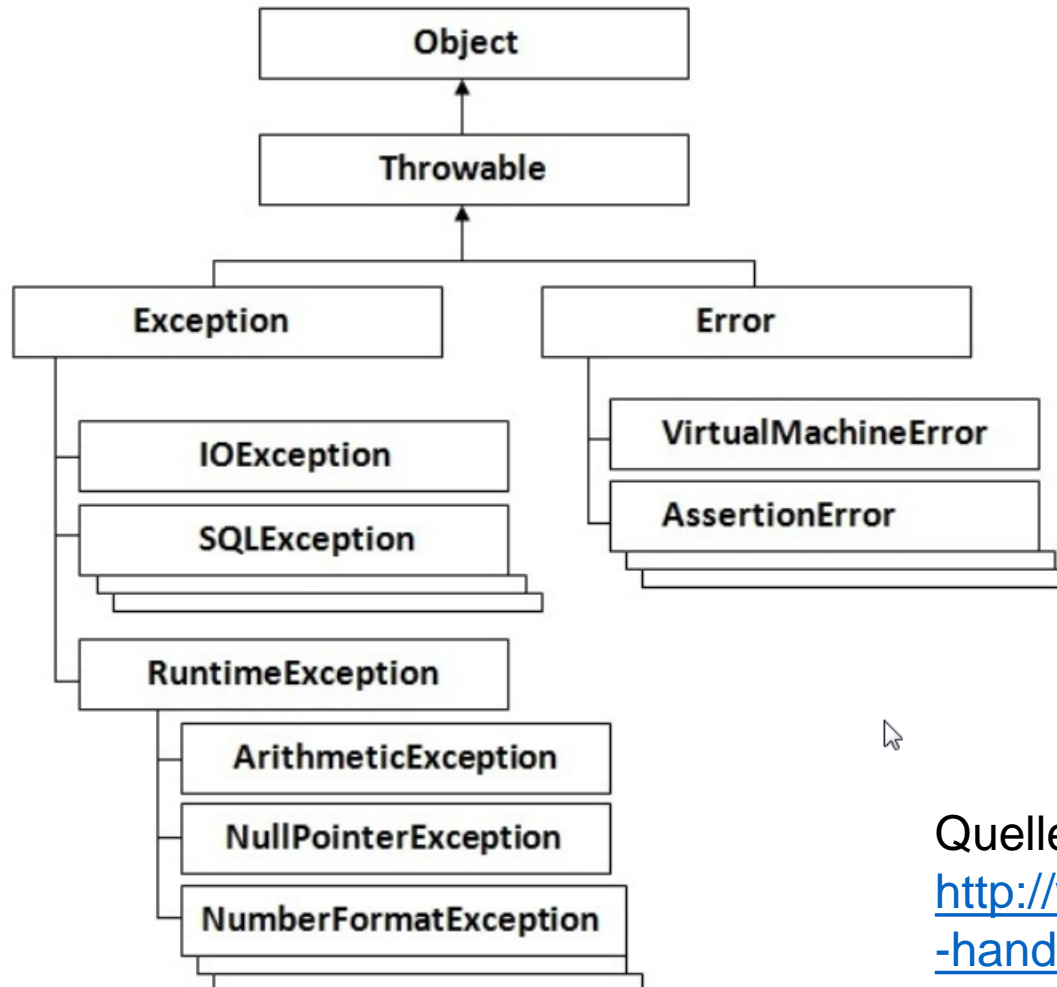
- Trennung Anwendungscode von Ausnahmebehandlungen
- Exceptions können nicht ignoriert werden. (Rückgabewerte schon)
- Fehler können einfach die Aufrufhierarchie hochpropagiert werden.
- Fehler können sauber gruppiert und differenziert werden.

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

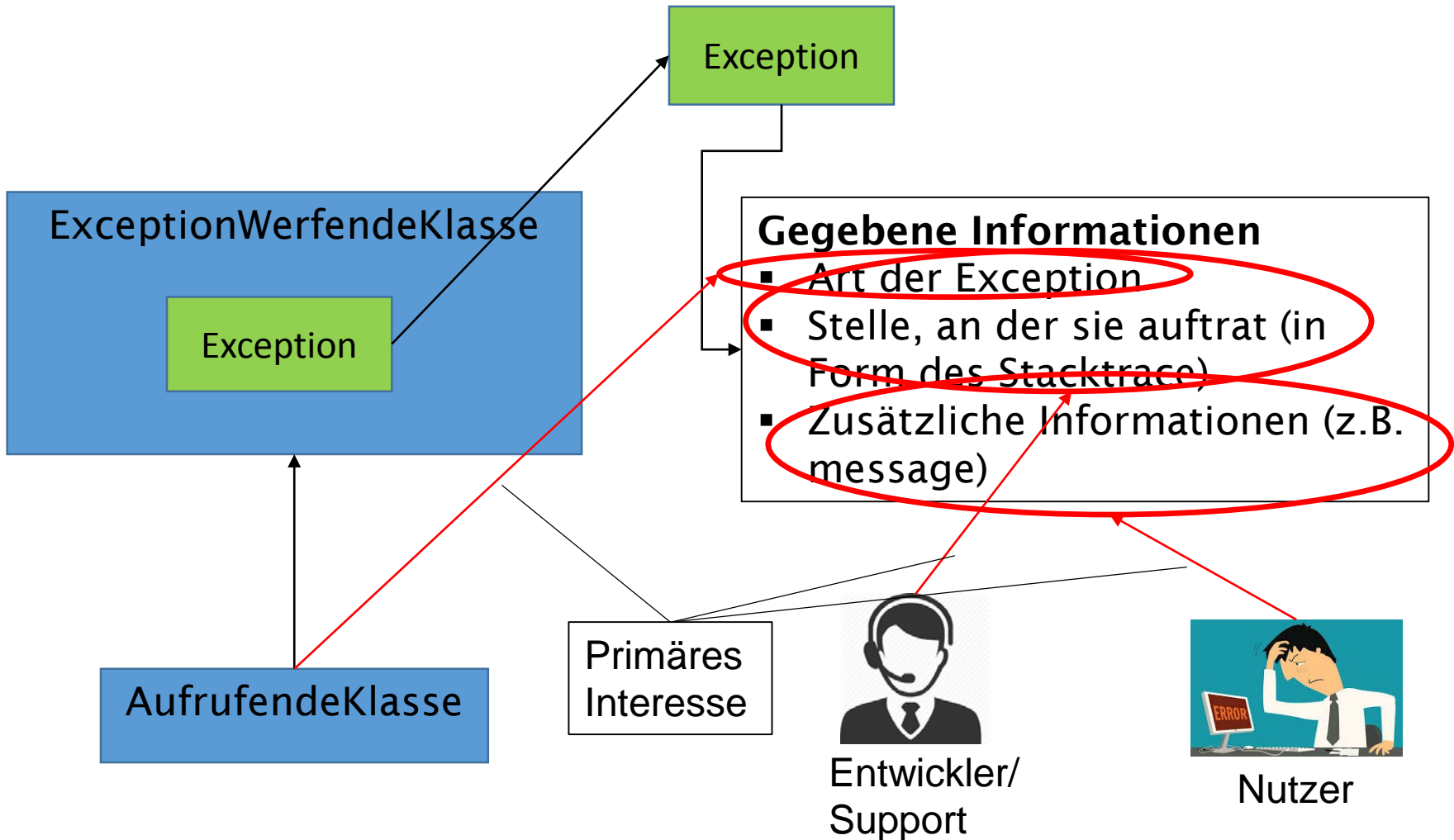
Schlechtes Exceptions Handling →

- 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

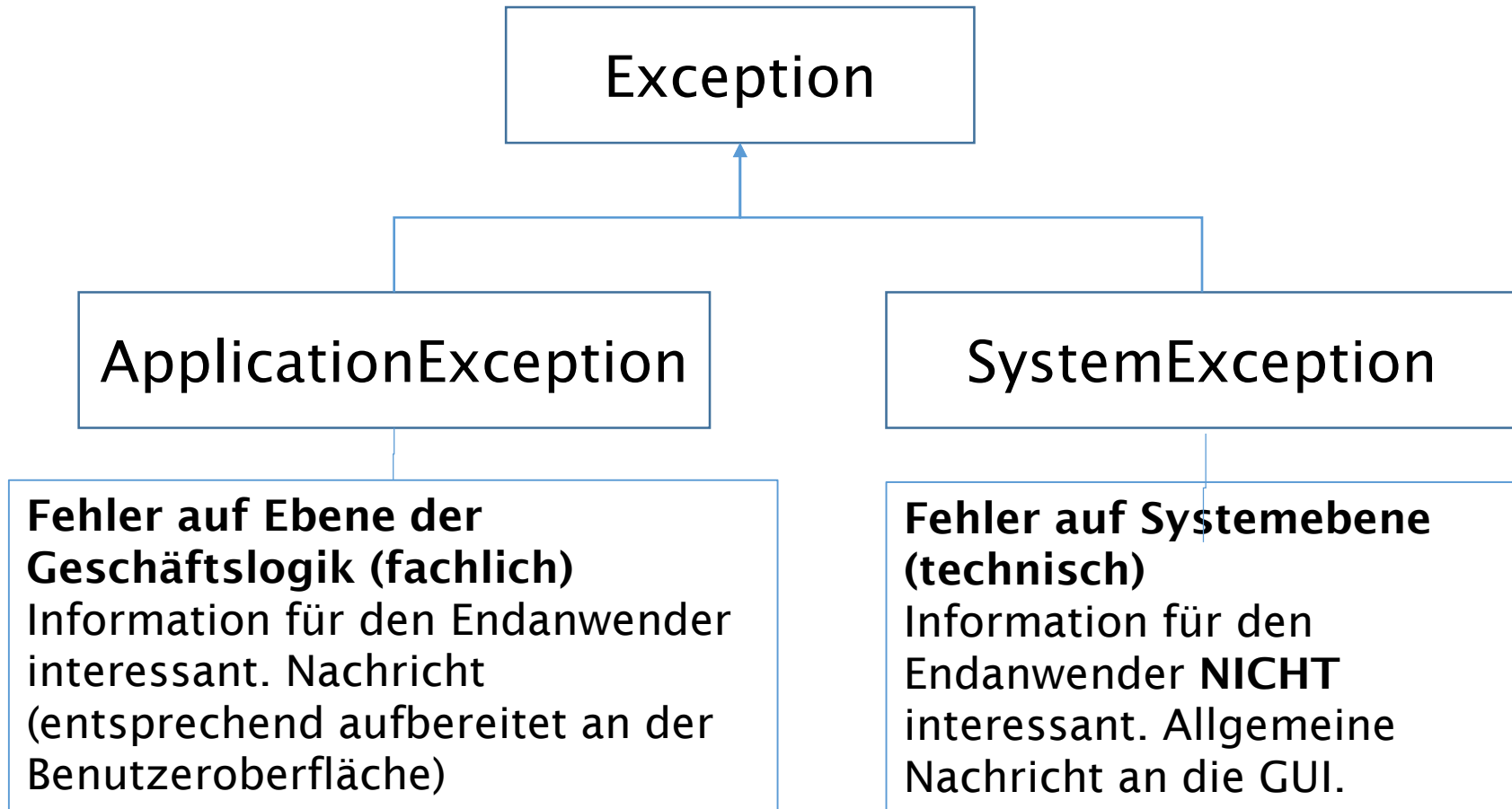
Vorheriges Beispiel führt dazu, dass wesentliche Information der ursprünglichen Exception verloren geht. Dem wirkt das Exception chaining entgegen.

→ **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

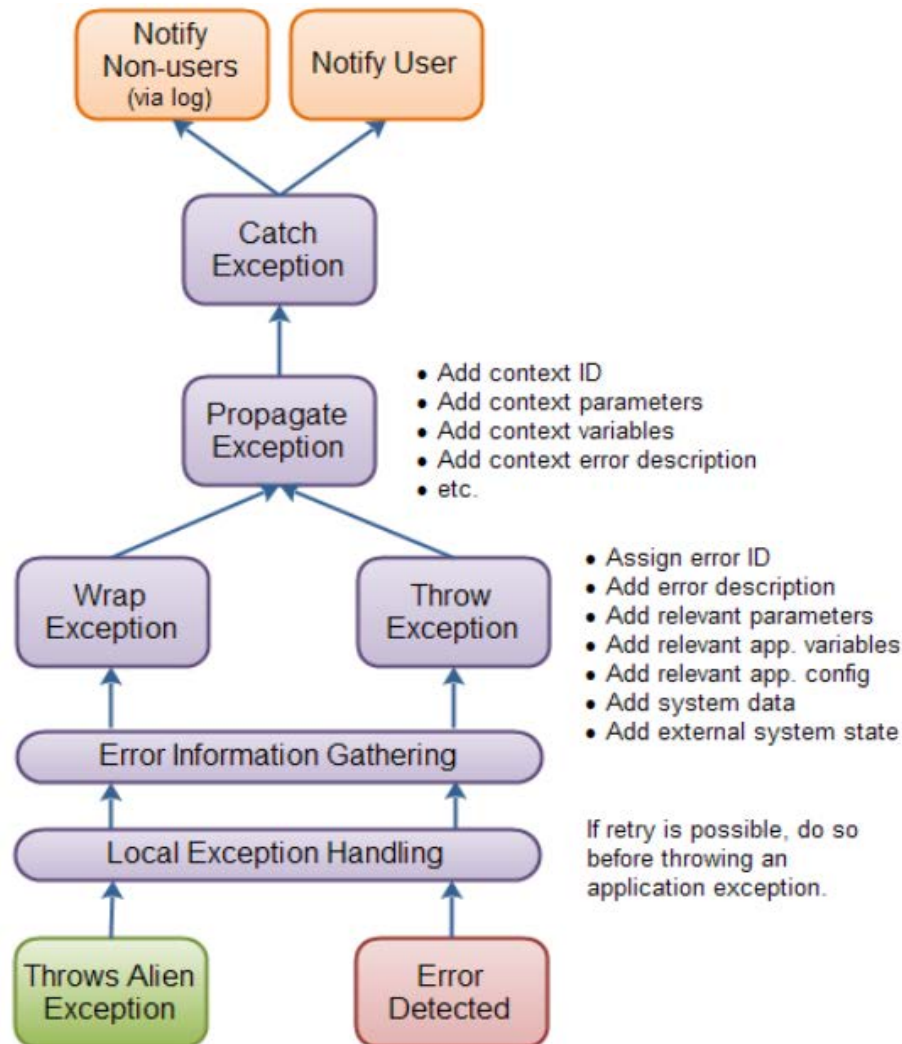
- Verwenden Sie spezifische Exceptions – keine toplevel Exceptions fangen.
- Fangen Sie die Exception erst dann, wenn Sie sie auch ordentlich behandeln können.
- Schließen Sie alle Ressourcen im finally Block (oder automatisch mit Java ARM)
- Loggen Sie alle Exceptions. Aber nur einmal!
- Verwenden Sie 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>

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

Anforderungen ans Exception Handling

Primär:

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

Sekundär:

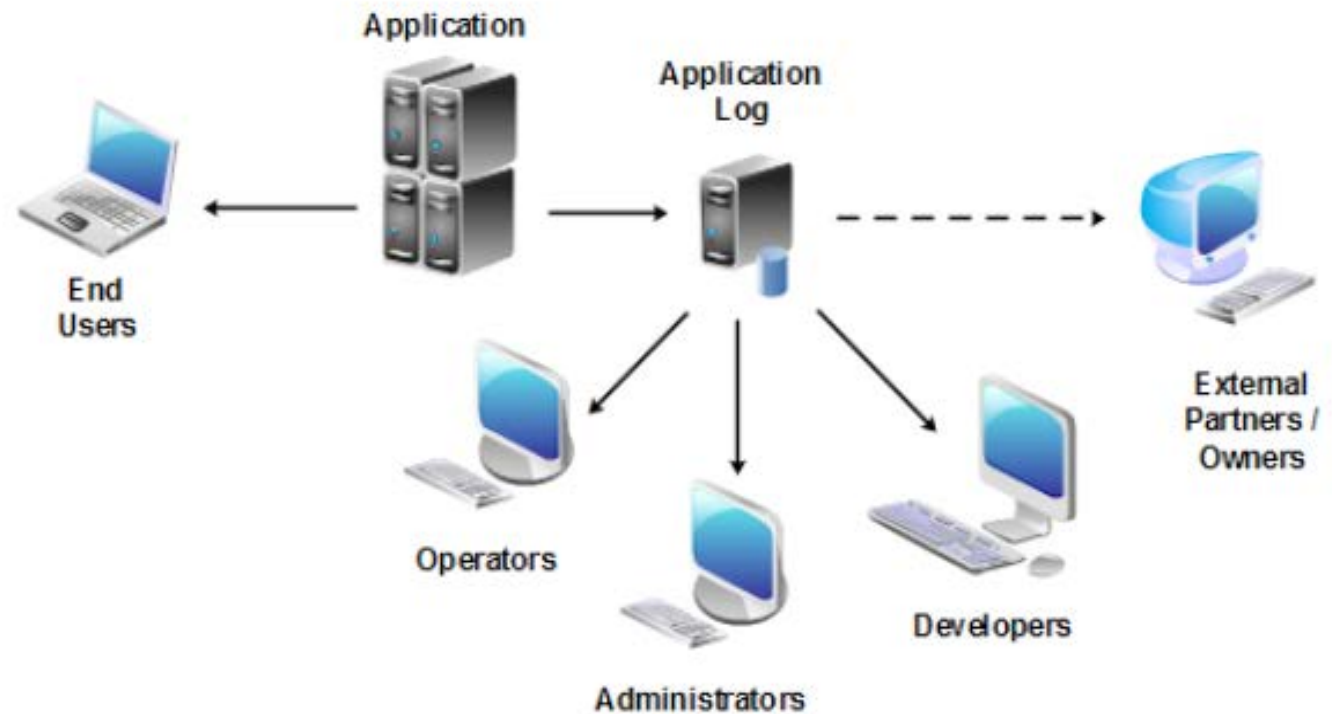
- Abstraktion
- Les- und Wartbarer Code

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
- Etnwickler
- 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

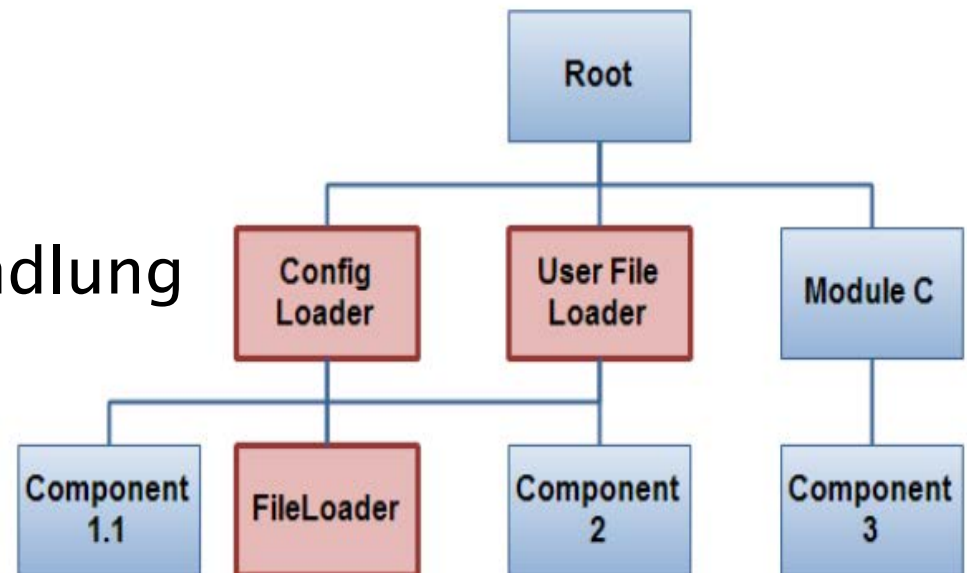
➔ Verbergen von Informationen, die die höheren Schichten nicht zu interessieren hat.

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



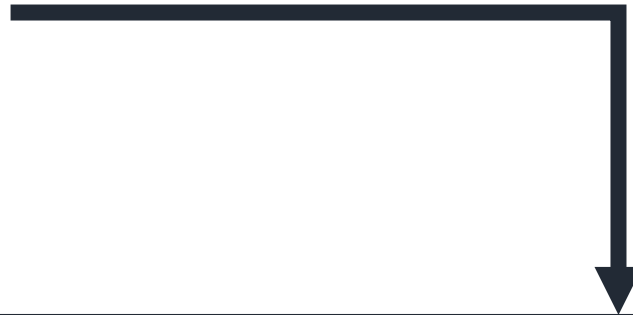
- Überblick
- Anforderungen
- Error Location und Error Context
- Fehlertypen und Reaktionen
- Strategie Elemente
- Mögliches Template

Fehlerarten und Reaktionen

Client Error

Service Error

Internal Error



Typische Reaktion:

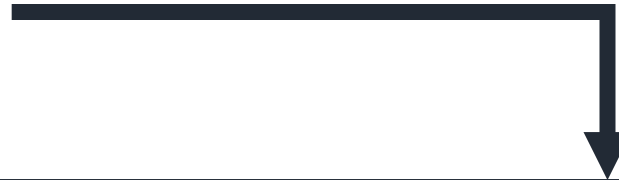
- Abbruch der aufgerufenen Aktion
- Schließen der geöffneten Ressourcen (connections, files, streams etc.)
- Freigeben der allokierten Ressourcen (memory buffers etc.).
- Nutzer benachrichtigen.
- Loggen des Fehlers

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

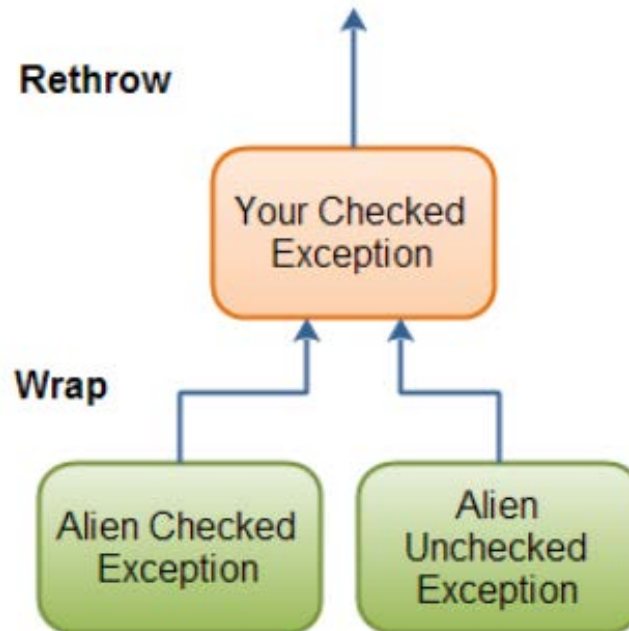
Werfen:

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 (MyAppException e) {  
  
    notifyUser(lookupErrorText(e));  
    notifyNonUsers(e);  
  
} catch (Throwable t) {  
  
    notifyUser(lookupErrorText(e));  
    notifyNonUsers(t);  
}
```

Exception Handling Strategie

- 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>

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

Portabilität entspricht Plattformunabhängigkeit.

Portierungsszenarien in der Praxis:

- Architekturportierung (auf andere Hardware)
- Betriebssystemportierungen
- Systemportierungen (Portierung auf andere Geräteklasse)
- Sprachportierungen

Es existieren verschiedene Ebenen, um die Portabilität eines SW Systems zu erhöhen:

- **Portabilität auf Implementierungsebene**
Wie kann ich ein portables Programm schreiben?
- **Auf Sprachebene**
Wie kann die Programmiersprache bzw der Compiler in Richtung höhere Portabilität bewegt werden?
- **Auf Systemebene**
Anpassung der Umgebung an das Programm

Hier nicht weiter behandelt. Details siehe z.B.

Dirk W. Hoffmann: Software-Qualität, 2 Auflage, Springer Vieweg

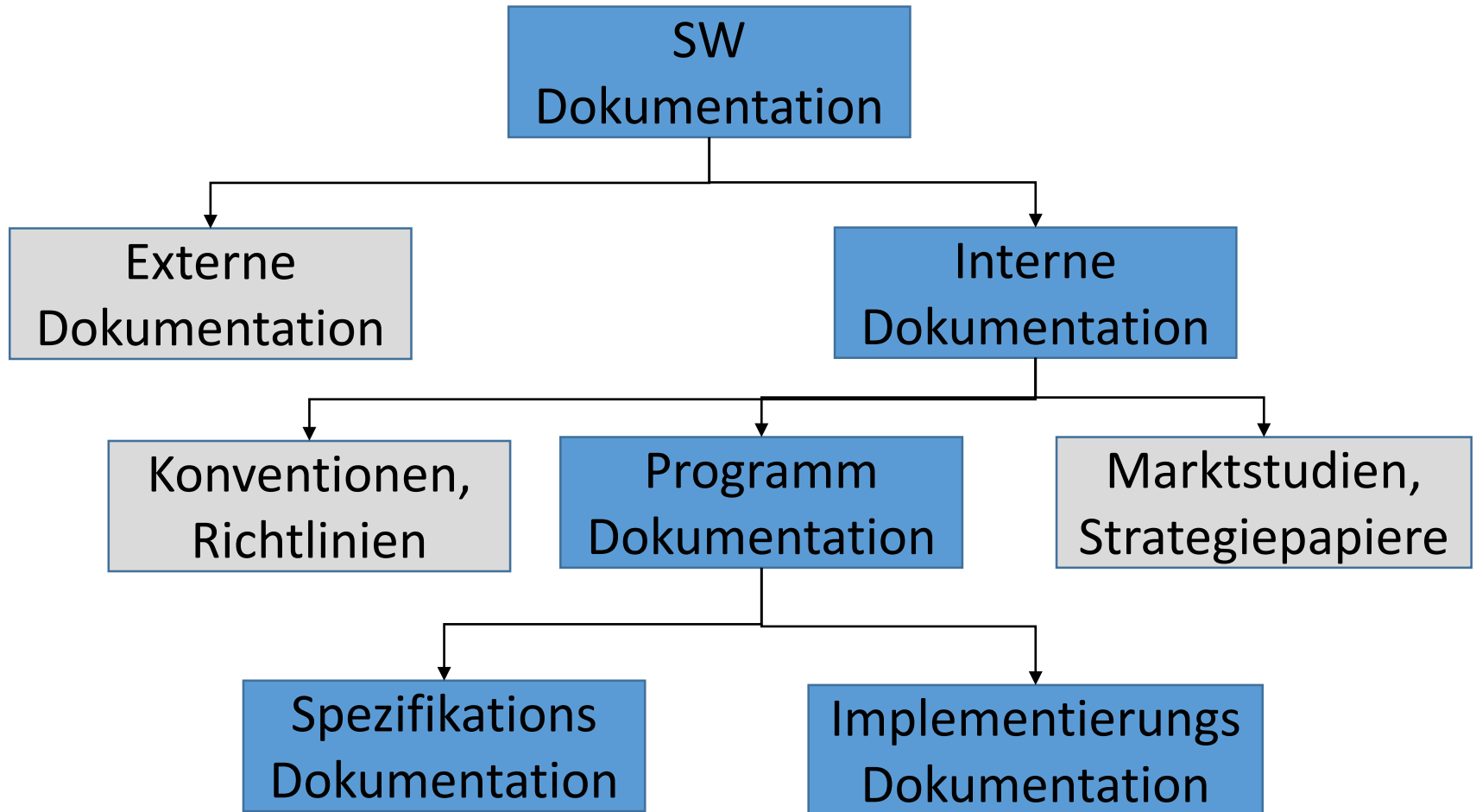
- Software Richtlinien
- Typisierung
- Vertragsbasierte Programmierung
- Fehlertolerante Programmierung
- Portabilität
- Dokumentation

- **Externe Dokumente**
werden an den Kunden ausgeliefert.
- **Interne Dokumente**
Nicht für den Kunden zugänglich.

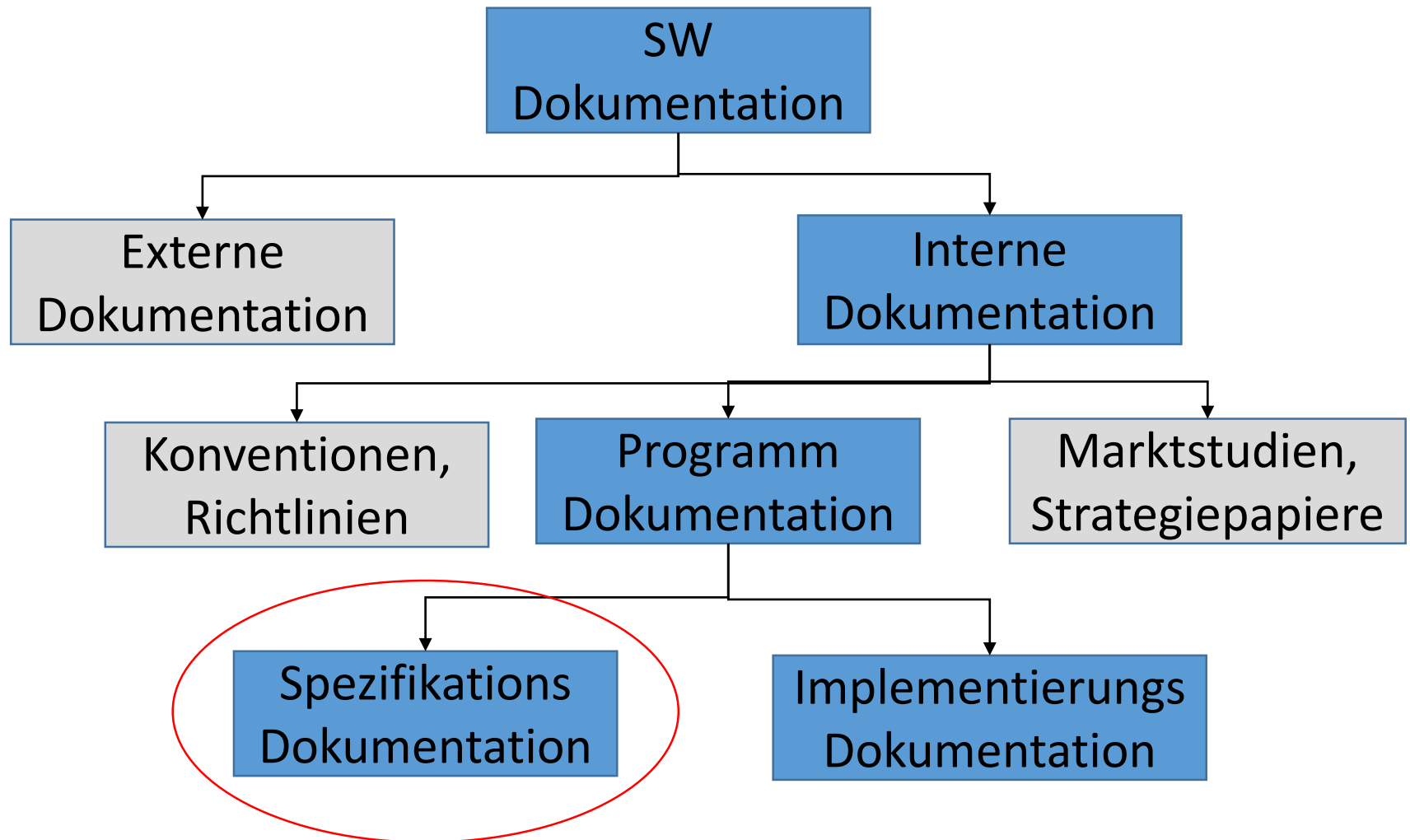
Hier besprochen: Programmdokumentation (Teil der internen Dokumentation)

NB: Terminologie ist nicht einheitlich: Hier wird die Terminologie von D. Hoffmann verwendet, der Spezifikationsdokumente auch als Dokumentation betrachtet.

Dichotomie der SW Doku - Ausschnitt



Dichotomie der SW Doku - Ausschnitt



Kriterien

- Vollständig
- Eindeutig
- Widerspruchsfrei
- Verständlich

Drei Varianten:

- Informal
- Semiformal
- Formal

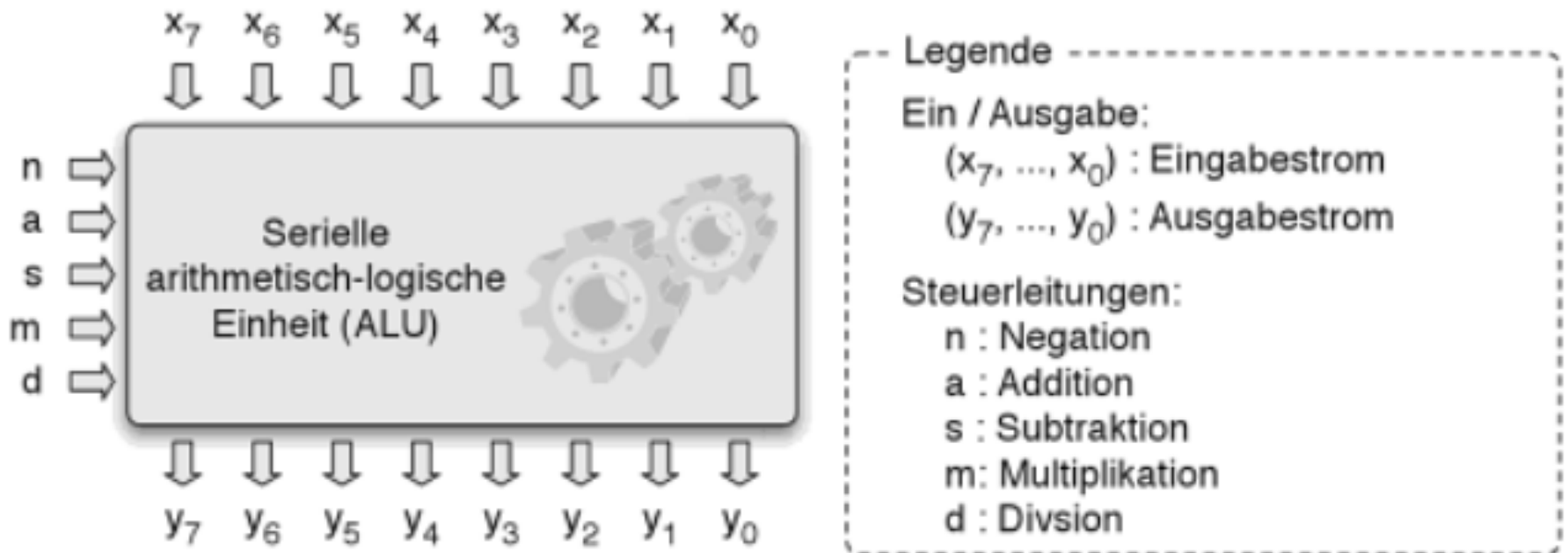
Umgangssprachlich formuliert.

Offene Fragen durch:

- Implizite Annahmen
- Ignorieren von Sonderfällen
- Sprachliche Ungenauigkeiten

Beispiel für informelle Spezifikation

Beispiel ALU



Informelle Spezifikation - Bsp

Die serielle arithmetisch-logische Einheit (ALU) berechnet aus dem seriellen Eingabestrom (x_7, \dots, x_0) den Ausgabestrom (y_7, \dots, y_0). Die von der ALU ausgeführte arithmetische Operation wird durch die Steuerleitungen n, a, s, m, d bestimmt. Für $n=1$ (negate) negiert die ALU den Eingabewert. Für $a=1$ (add) berechnet sie die Summe, für $s=1$ (subtract) die Differenz, für $m=1$ (multiply) das Produkt und für $d=1$ (divide) den Quotienten der letzten beiden Eingabewerte.

Beispiel – offene Fragen

1. Wie werden negative Werte dargestellt?
2. Wie verhält sich die Schaltung bei numerischen Überläufen?
3. Was passiert, wenn alle Steuerleitungen gleich 1 sind?
4. Was passiert, wenn alle Steuerleitungen gleich 0 sind?
5. Wie wird die Division durch 0 behandelt?
6. Welche Ausgabe liegt zum Zeitpunkt 0 an?
7. Welche Werte sind „die letzten beiden“ genau?
8. Was genau bedeutet „der Quotient“?

Spezifikationsdokumentation Semiformale Spezifikation - Bsp

- **Eingabe**
 - $x[t]$: Eingabe zum Zeitpunkt t in 8 Bit-Zweierkomplementdarstellung
- **Ausgabe**
 - $y[t]$: Ausgabe zum Zeitpunkt t in 8-Bit Zweierkomplementdarstellung
- **n, a, s, d, m ; Steuerleitungen**
- **Verhalten:**
 - $y[0] = \begin{cases} -x[0] & \text{für } n=1 \text{ und } a=s=m=d=0 \\ 0 & \text{sonst} \end{cases}$
 - $y[n+1] = \begin{cases} -x[n+1] & \text{für } n=1 \text{ und } a=s=m=d=0 \\ x[n]+x[n+1] & \text{für } a=1 \text{ und } n=s=d=m=0 \\ x[n]-x[n+1] & \text{für } s=1 \text{ und } a=n=d=m=0 \\ x[n] \times x[n+1] & \text{für } m=1 \text{ und } n=s=a=d=0 \\ x[n] / x[n+1] & \text{für } d=1 \text{ und } n=a=a=m=0 \\ 0 & \text{sonst} \end{cases}$

Weiteres Beispiel

Anforderung an ein Programm in einer Musikschule, um Leihinstrumente zu verwalten:

„Die Anzahl der Saiteninstrument ist doppelt so hoch wie die Anzahl der Blasinstrumente. Die Summe der Saiten- und der Blasinstrumente liegt zwischen 5 und 15.“

→ Formulierung in mathematischen Formeln:

$$\text{Anzahl}(\text{SaitenInstr.}) = 2 \times \text{Anzahl}(\text{Blasinstrumente})$$

$$5 < \text{Anzahl}(\text{SaitenInstr.}) + \text{Anzahl}(\text{Blasinstrumente}) < 15$$

Aufgabe: Programmieren Sie folgende Zahlenreihe bis zu Ihrem 8. Element:

Spezifikation:

- Die Zahlenreihe beginnt mit einer 3.
- Jede Zahl der Folge ist um 1 grösser als die Hälfte der nächsten Zahl.

Bessere Spezifikation:

$$Z(1) = 3$$

$$Z(n+1) = 2 \text{ mal } (Z(n) - 1)$$

Umsetzung in C

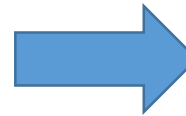
```
#include <stdio.h>

int main() {

    int zahl = 3;
    int ende = 0;
    int zaehler;
    printf("wie weit soll es denn gehen?\n");
    scanf("%d", &ende);
    printf("die %ite Zahl ist %i\n", 1 , zahl);

    for(zaehler =0; zaehler < ende-1; zaehler ++){

        zahl= 2 * (zahl-1);
        printf("die %ite Zahl ist %i\n", zaehler+2, zahl);
    }
    getchar();
    return 0;
}
```



```
wie weit soll es denn gehen?
8
die 1te Zahl ist 3
die 2te Zahl ist 4
die 3te Zahl ist 6
die 4te Zahl ist 10
die 5te Zahl ist 18
die 6te Zahl ist 34
die 7te Zahl ist 66
die 8te Zahl ist 130
```

Weiteres Beispiel

Das Laufwerk von Nicks Computer hat die doppelte Kapazität wie das von Alfs Computer. Zusammen haben sie 240 GB.

Schreiben Sie in Programm, das die die Kapazität der einzelnen Laufwerke berechnet.

Textuell schwer zu lösen. Mathematisch sehr einfach:

(1) $x + y = 240$

(2) $x = 2 * y$

→ aus (1): $3y = 240 \rightarrow y = 80$

In (2) → $x = 160$

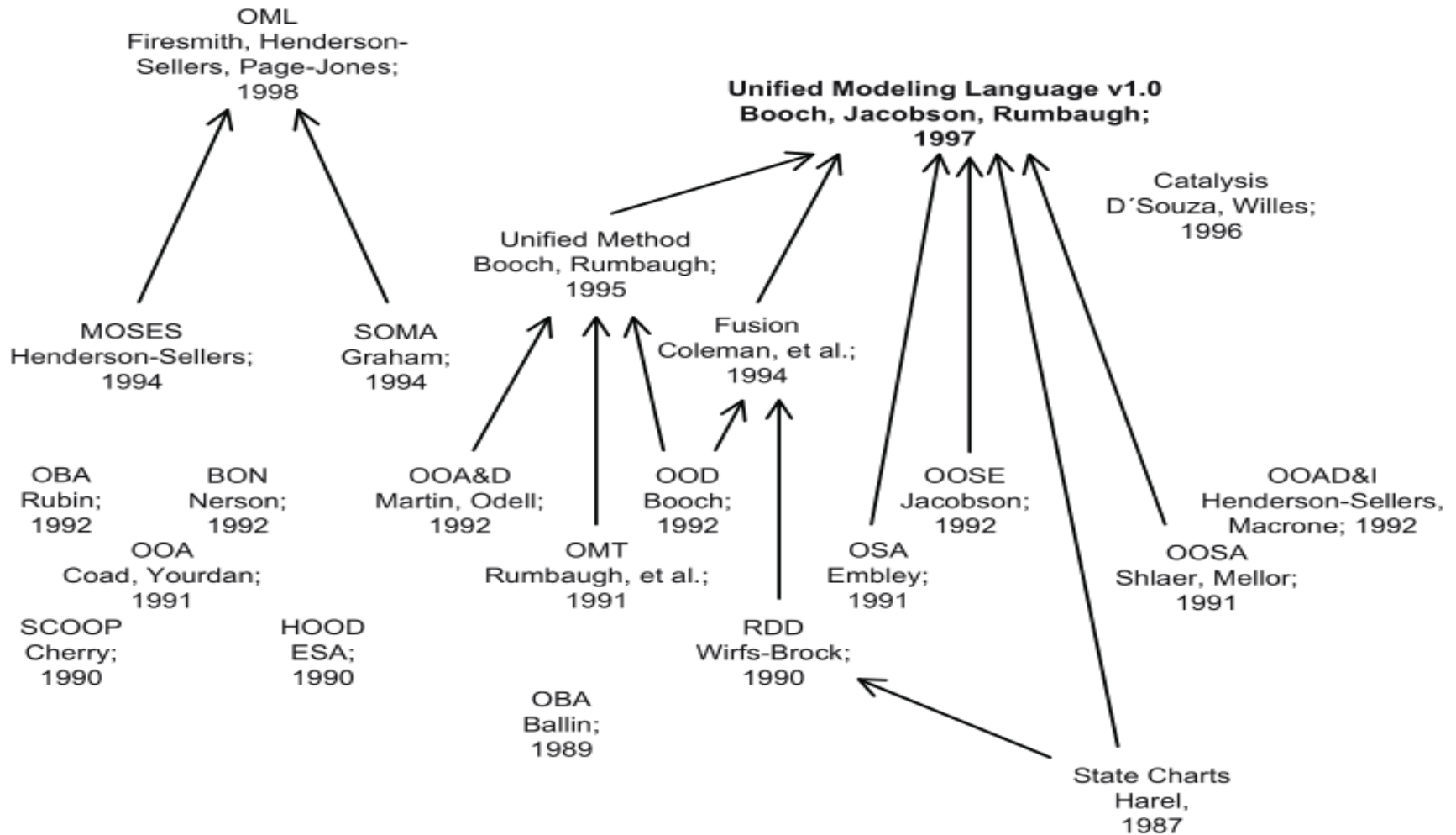
- Verwendung definierter Formalismen.
- Vorteil: Zweideutigkeiten vermieden, dennoch gut lesbar.

Prominentes Beispiel einer semi formalen Spezifikationssprache: **UML**

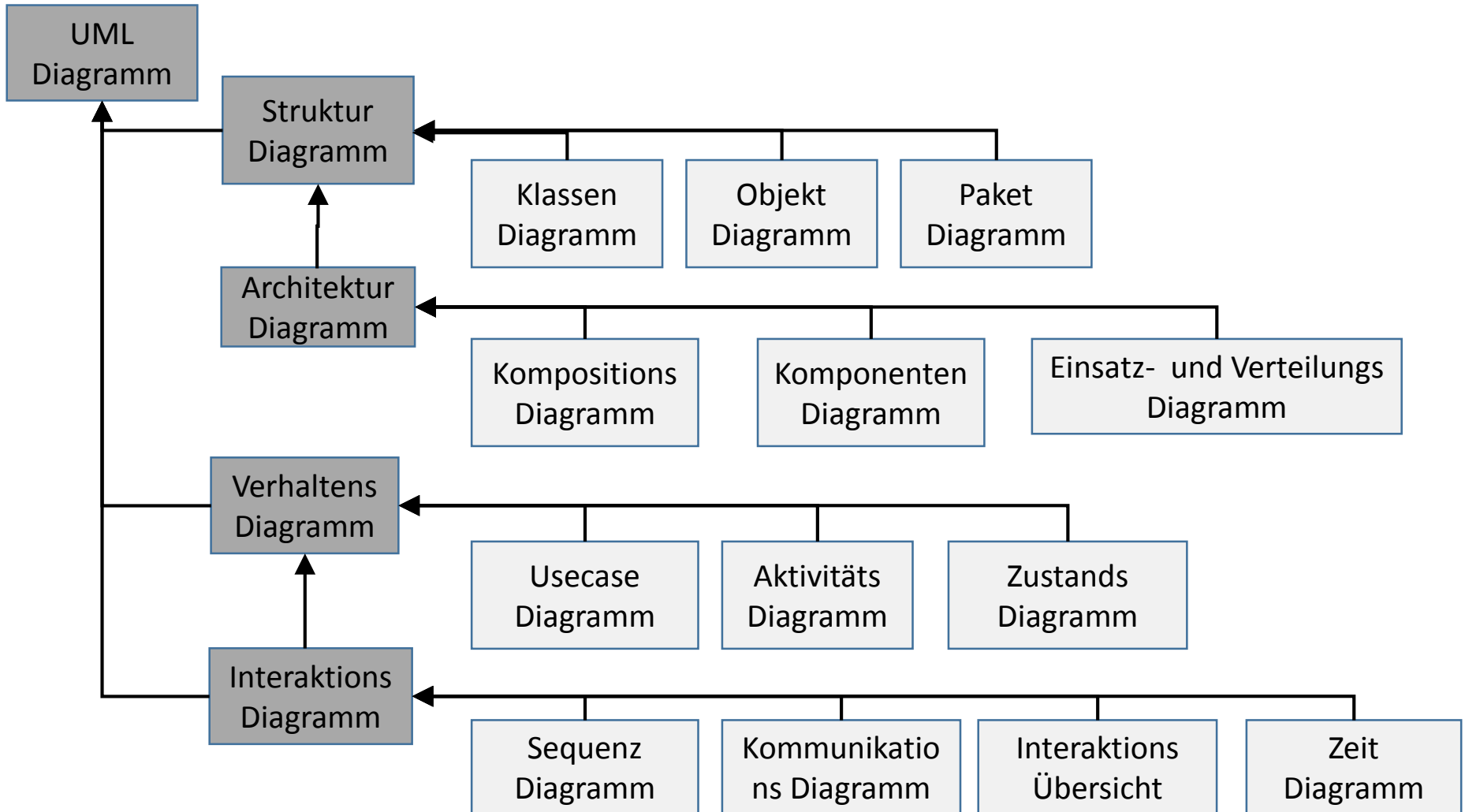
UML Diagramme

Die Unified Modeling Language, kurz UML, ist eine grafische Darstellungsform zur **Visualisierung, Spezifikation, Konstruktion** und **Dokumentation** von (Software) Systemen. Sie bietet ein Set an standardisierten Diagrammtypen, mit denen komplexe Sachverhalte, Abläufe und Systeme einfach, übersichtlich und verständlich dargestellt werden können.

Uml Historie



UML Diagrammtypen



- ➔ **Übungsbeispiel zu Activity Diagramm ➔**
Bsp. zu Materialwirtschaft.
- ➔ **Übungsbeispiel zu Usecase Diagramm ➔**
Bsp. zu Fahrradshop.
- ➔ **Übungsbeispiel Klassen Diagramm ➔ Bsp.**
Videoverleih.
- ➔ **Übungsbeispiel Zustandsdiagramm ➔ Bsp**
Tresor

- Lässt keinen Interpretationsspielraum.
- Wird schnell extrem komplex, sogar bei einfachen Sachverhalten.
- Spielt in der Praxis kaum eine Rolle.

Beispiele:

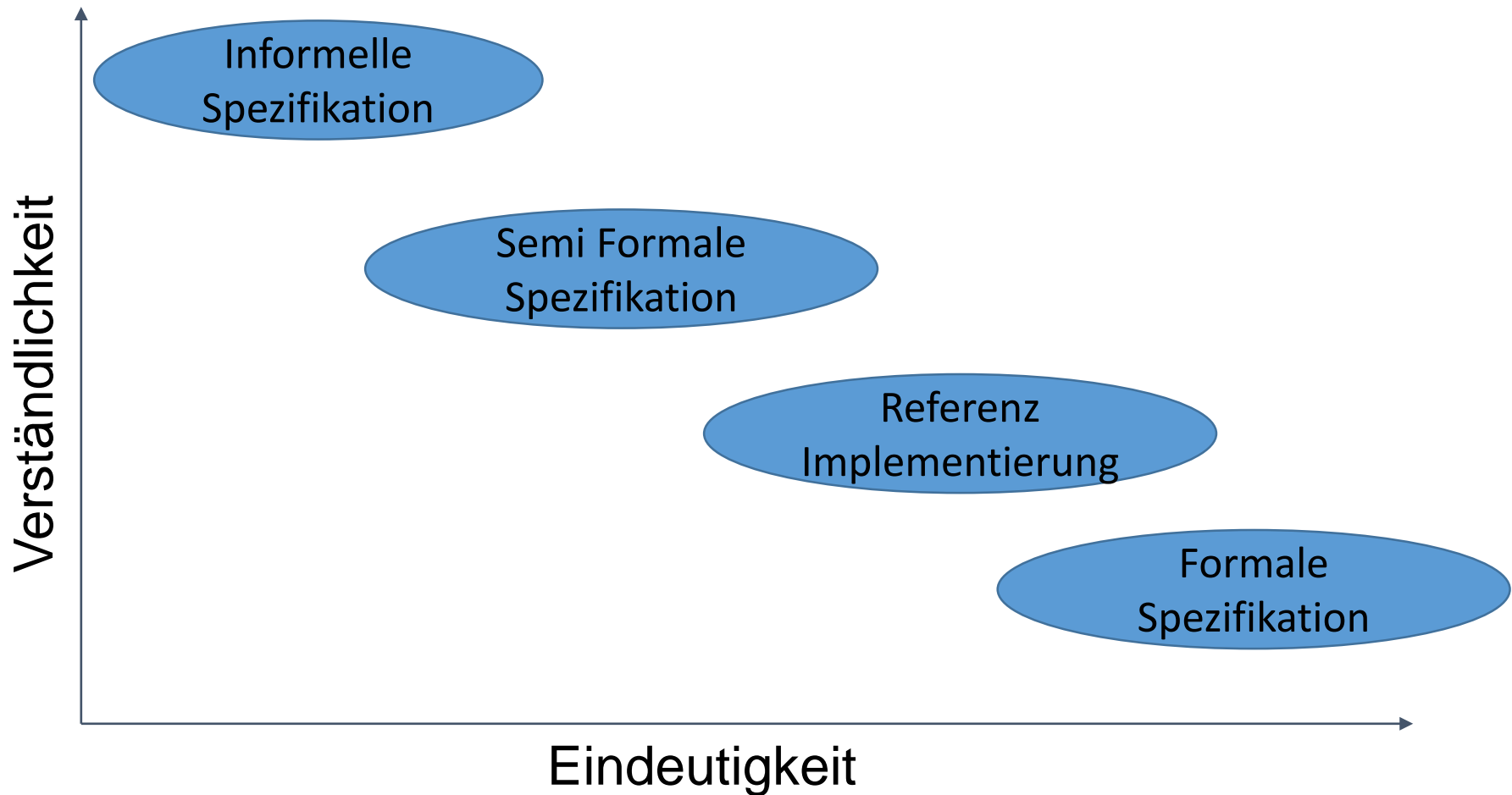
- Spezifikationssprache Z
- Vienna Development Method

Spezifikation wird durch ein Programm ersetzt, dessen Verhalten als richtig definiert wird.

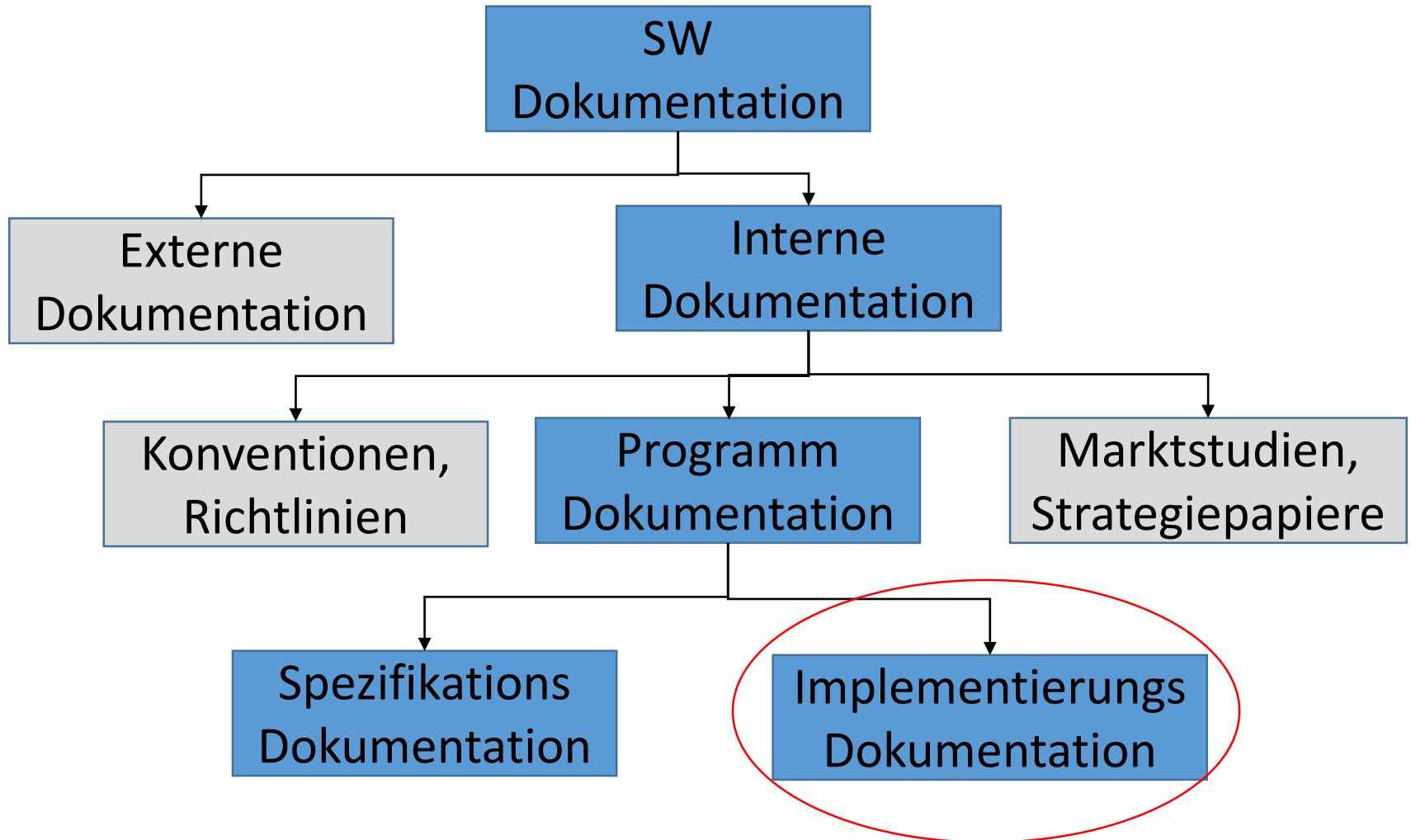
➔ Wenn sich das neue entwickelte Programm genauso verhält wie die Referenzimplementierung, dann entspricht sie der Spezifikation.

- Anwendung der Referenzimplementierung
z.B. beim Refactoring: Neues Verhalten ist
gleich altem Verhalten.
- Referenzimplementierung oftmals nicht
ausreichend, da zwar das Verhalten
nachprüfbar ist, aber nicht, warum sich ein
System so verhalten soll.

Spezifikationstechniken im Vergleich



Dichotomie der SW Doku



Beschreiben, **wie** die Spezifikation umgesetzt wurde.

Oftmals: „Our code is our documentation“

➔ Nicht befriedigend, Nicht ausreichend

Hier behandelt: Integrierte Dokumentation, d.h. Code Dokumentation.

Teil des Agilen Manifests:

...

„Funktionierende Software mehr als
umfassende Dokumentation“

...

➔ In agilen Projekten führt das oft dazu, dass Dokumentation nicht Teil der Iteration und der DoD ist.

Code Doku - Bsp

```
int ack(int n, int m)
{
    while (n!=0) {
        if (m==0) {
            m=1;
        } else {
            m=ack(m, n-1);
        }
        n--;
    }
    return m+1;
}
```

Wo ist der Fehler?

Code Doku – Bsp Doku als Selbstzweck

```

/*
Funktion int ack(int n, int m)
Autor: Irgendwer
Datum: 15.3.2013
Revision History:
    12/15/2006: While Iteration eingefügt
    13/01/2008: Funktionsname geändert
*/
int ack(int n, int m)
{
    /*solange die erste Variable nicht null ist ...*/
    while (n!=0)
    {
        /*teste zweite Variable auf 0 */
        if (m==0)
        {
            m=1;
        }
        else
        {
            /*hier erfolgt der rekursive Aufruf*/
            m=ack(m,n-1);
        }
        n--;
    }
    /*Liefere Ergebnis zurueck*/
    return m+1;
}

```

Diese Information ist im
Source Verwaltungstool
enthalten.

Trivial und überflüssig

Wo ist der Fehler?
Immer noch schwierig
zu finden

Sinnvolle Doku - Bsp

```
/*
Funktion int ack(int n, int m)
Autor: Irgendwer
Beschreibung: Berechnet die Ackermann Funktion
Definition der Ackermann Funktion:
(1) ack(0,m) = m+1;           m>=0
(2) ack(n,0) = ack(n-1,1);   n>0
(3) ack(n,m) = ack(n-1,ack(n,m-1)); m,n>0

Hinweis: Die Ackermann wächst stärker als die Exponentialfunktion
Bsp:
ack(3,1)=13
ack(4,1)=65533
ack(5,1) = ca 20stellige Zahl
*/
```

```
*/
int ack(int n, int m)
{
    while (n!=0)
    {
        if (m==0)
        {
            /*Fall (2)*/
            m=1;
        }
        else
        {
            /*Fall (3)*/
            m=ack(m,n-1);
        }
        n--;
    }
    /*Fall (1)*/
    return m+1;
}
```

Spezifikation, was die Funktion tut

Finden Sie den Fehler jetzt?



Rekursiver Aufruf ist falsch:
statt $\text{ack}(m, n-1)$ müsste es heißen: $\text{ack}(n, m-1)$

Regeln für die Kommentierung von Quellcode

1. Make the code as clear as possible to reduce the need for comments!
2. Never repeat information in a comment that is already available in the code!
3. Where a comment is required, make it concise and complete!
4. Use proper grammar and spelling in comments!
5. Make comments visually distinct from the code!
6. Structure comments in headers so that information can be automatically extracted by a tool!

ADA (1995): A DA 95 Quality and Style: Guidelines for Professional Programmers. Software Productivity Consortium, SPC-94093-CMC, Version 01.00.10

- Vermeiden Sie Dokumentation als Selbstzweck.
- Dokumentation zielt auf einen Adressaten – denken Sie daran, welche Information dieser Adressat benötigt.
- Es kann sinnvoll sein, zu dokumentieren, warum etwas nicht implementiert ist.

Bis **jetzt**: Behandlung dessen, welche Codeteile wie dokumentiert werden.

Jetzt: Wer hat in welcher Form Zugriff auf die Dokumentation.

Wer benötigt Code Dokumentation?

- Jeder, der den Code **benutzt**
 - ➔ Entwickler, die Ihre Komponenten/Klassen etc benutzen.
 - ➔ Dokumentation ist relevant für den Nutzer, ohne dass er direkt in den Code schauen muss (z.B. weil er ein jar File bei sich einbindet).
- Jeder, der den Code **weiterentwickelt**
 - ➔ Entwickler, die auf Ihrem Code aufbauen.

Bsp für magere API Dokumentation

All Classes

Packages

- org.apache.lucene
- org.apache.lucene.analysis
- org.apache.lucene.analysis.token
- org.apache.lucene.codecs
- org.apache.lucene.codecs.com
- org.apache.lucene.codecs.lucene
- org.apache.lucene.codecs.lucene

All Classes

- AfterEffect
- AfterEffect.NoAfterEffect
- AfterEffectB
- AfterEffectL
- AlreadyClosedException
- Analyzer
- Analyzer.GlobalReuseStrategy
- Analyzer.PerFieldReuseStrategy
- Analyzer.ReuseStrategy
- Analyzer.TokenStreamComponent
- AnalyzerWrapper
- ArrayUtil
- AtomicReader
- AtomicReaderContext
- Attribute
- AttributeImpl
- AttributeReflector
- AttributeSource
- AttributeSource.AttributeFactory
- AttributeSource.State
- Automaton
- AutomatonProvider
- AutomatonQuery
- AveragePayloadFunction
- BaseCompositeReader
- BasicAutomata

grow

```
public static byte[] grow(byte[] array)
```

shrink

```
public static byte[] shrink(byte[] array,
    int targetSize)
```

grow

```
public static boolean[] grow(boolean[] array,
    int minSize)
```

grow

```
public static boolean[] grow(boolean[] array)
```

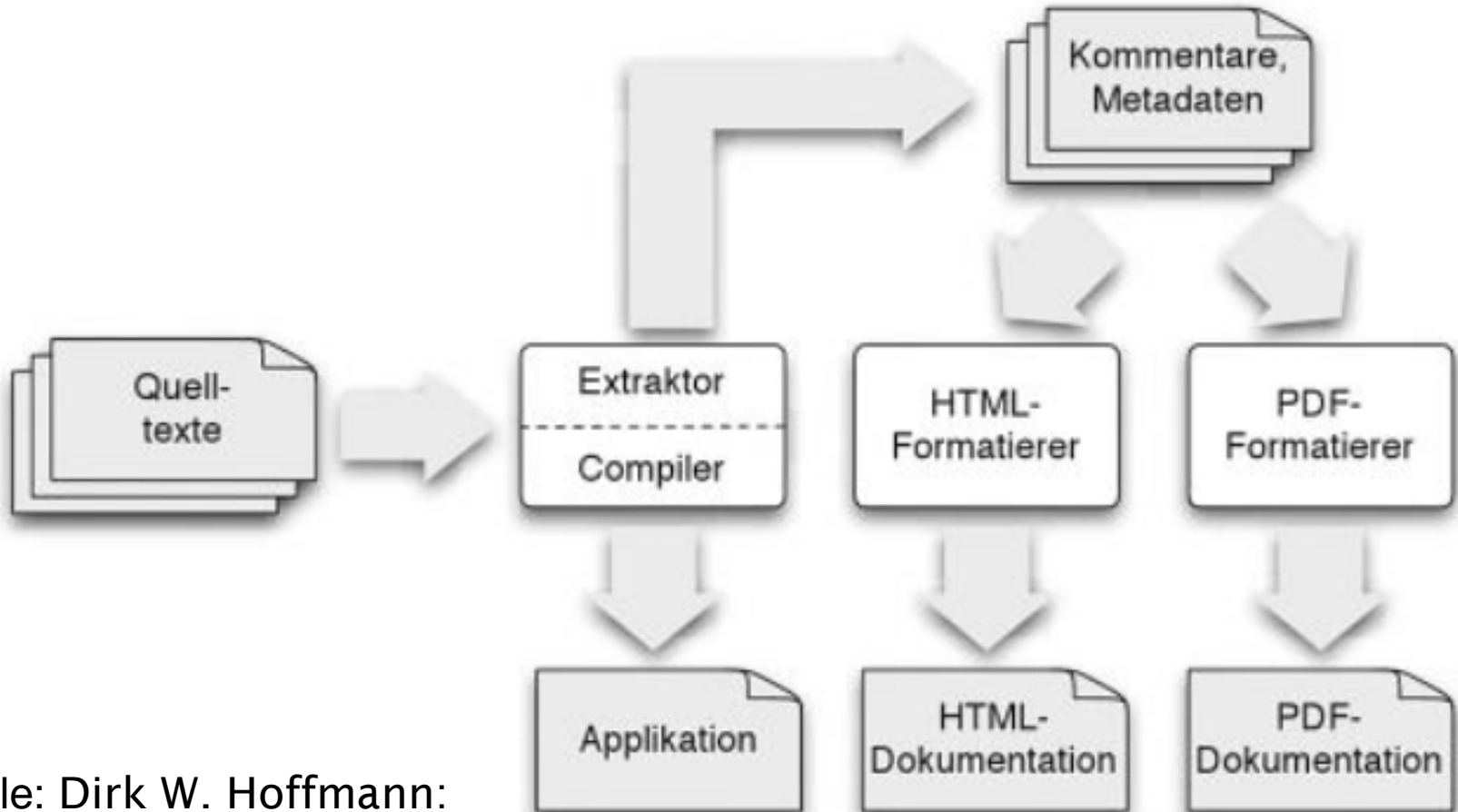
shrink

```
public static boolean[] shrink(boolean[] array,
    int targetSize)
```

grow

```
public static char[] grow(char[] array,
    int minSize)
```

Extraktion von Dokumentation



Quelle: Dirk W. Hoffmann:
Software-Qualität,
2. Auflage, Springer Vieweg

Dokumentation Extraktion

- ➔ Dokumentation ist auch von Personen lesbar, die keinen Zugriff auf den Code haben.
- ➔ Dokumentation ist ohne Code lesbar.
- ➔ Dokumentation ist in aufbereiteter Form lesbar (HTML oder Pdf).
- ➔ Querverweise sind im Kommentar leichter nachzuverfolgen.

Bsp: Java SE 6, API Specification:

<http://docs.oracle.com/javase/6/docs/api/>

Bsp: Java SE 6, API Specification

<http://docs.oracle.com/javase/6/docs/api/>

**Java™ Platform
Standard Ed. 6**

[All Classes](#)

Packages

- [java.applet](#)
- [java.awt](#)
- [java.awt.color](#)
- [java.awt.datatransfer](#)

All Classes

- [AbstractAction](#)
- [AbstractAnnotationValueV](#)
- [AbstractBorder](#)
- [AbstractButton](#)
- [AbstractCellEditor](#)
- [AbstractCollection](#)
- [AbstractColorChooserPan](#)
- [AbstractDocument](#)
- [AbstractDocument.Elemen](#)
- [AbstractElementVisitor6](#)
- [AbstractExecutorService](#)
- [AbstractInterruptibleChann](#)
- [AbstractLayoutCache](#)
- [AbstractLayoutCache.Nod](#)
- [AbstractList](#)
- [AbstractListModel](#)
- [AbstractMap](#)
- [AbstractMap.SimpleEntry](#)
- [AbstractMap.SimpleImmute](#)
- [AbstractMarshallerImpl](#)
- [AbstractMethodError](#)

Standard Edition 8 Documentation
acle.com/javase/8/docs/

Overview Package Class Use **Tree** [Deprecated](#) [Index](#) [Help](#)

PREV NEXT [FRAMES](#) [NO FRAMES](#)

**Java™ Platform, Standard Edition 6
API Specification**

This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

See:
[Description](#)

Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometr

Java unterstützt 3 Typen von Kommentaren:

- `/* text */`: Der Compiler ignoriert alles zwischen `/*` und `*/`
- `//text`: Der Compiler ignoriert alles von `//` bis Zeilenende
- `/**`
documentation
`*/`
sogenannter „doc comment“: wird von javadoc genutzt.

Bsp javadoc

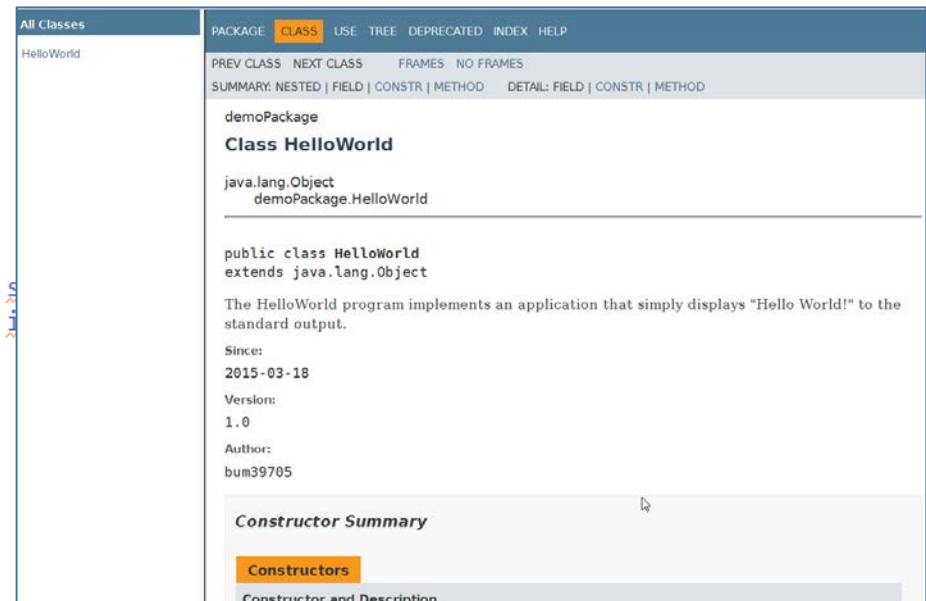
```
package demopackage;
```

```
/**
 * The HelloWorld program implements an application that simply displays
 * "Hello World!" to the standard output.
 */
@author bum39705
@version 1.0
@since 2015-03-18

public class HelloWorld {

    /**
     * Ein kurzer Satz, der im Abschnitt "Method Summary"
     * Es folgt die ausführliche Beschreibung, die später
     * Abschnitt "Method Detail" erscheint.
     */
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Javadoc aus eclipse generiert

The screenshot shows the Eclipse IDE with the Javadoc for the `demoPackage.HelloWorld` class. The left pane shows the class hierarchy, and the right pane displays the Javadoc content. The Javadoc includes the package name, class name, inheritance, a summary, and a constructor summary.

```
demoPackage
Class HelloWorld
java.lang.Object
demoPackage.HelloWorld

public class HelloWorld
extends java.lang.Object

The HelloWorld program implements an application that simply displays "Hello World!" to the
standard output.

Since:
2015-03-18
Version:
1.0
Author:
bum39705

Constructor Summary
Constructors
Constructor and Description
```

Docu: <http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html>
<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>
 Siehe auch http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_05_014.htm#mj1b008761e7e60dd49fcab7bf8de0d2cd

Was ist Javadoc?

Javadoc ist ein Werkzeug, das eine standardisierte Dokumentation für Java unterstützt.

Javadoc ist integraler Bestandteil des JDK.
Es ist nach der Installation im gleichen Verzeichnis wie der Java Compiler javac.

Wie funktioniert javadoc?

Javadoc liest Deklarationen und Doc Comments aus den Quelldateien.

Es können auch weitere Dateien eingebunden werden.

Aus diesen Informationen wird eine HTML basierte Dokumentation erzeugt.

Doc Comments:

```
/**
```

```
*Das ist ein Doc Comment
```

```
*/
```

Welche Dateien werden von javadoc berücksichtigt?

Javadoc erstellt die Dokumentation aus folgenden Dateien:

- Quellcodedateien
- Package Bemerkungen
- Overview Bemerkungen
- diversen weiteren Dateien

Doc Comments in Quellcodedateien können vor

- Klassen
- Konstruktoren
- Datentypen
- Methoden stehen.

Doc Comments werden aber nur beachtet, wenn sie genau vor einer Deklaration stehen!

Ausgelesen als HTML → es kann HTML Syntax enthalten sein.

Package Bemerkungen

Für jedes Package kann eine Beschreibung geschrieben werden.

Es muss eine Datei erstellt werden die den Namen `package.html` trägt. Diese Datei muss in das Verzeichnis des Packages abgelegt werden.

Die Datei `package.html` benötigt keine Doc Comments, sie enthält nur HTML.

Javadoc bindet diese Datei automatisch ein.

Overview Bemerkungen

In diese HTML Datei kommen Beschreibungen die für die ganze Anwendung gelten.

Sie ist wie die Datei package.html typischerweise ohne Doc Comments.

Diese Datei braucht keinen speziellen Namen. Wenn Javadoc ausgeführt wird, muss dieser Dateiname als Parameter angegeben werden.

Einbindung weiterer Dateien

Diese Dateien werden von Javadoc in das Ausgabeverzeichnis der Dokumentation kopiert.

Dies können weitere HTML Dateien sein, Applets, Beispielcode aber auch Grafiken die in die Dokumentation eingebunden werden.

Diese Dateien müssen in einem Verzeichnis unterhalb des Packages abgelegt sein, welches ../doc-files/ heißen muss.

Zugriff kann innerhalb der Doc Comments so erfolgen:

```
/**  
 * This button looks like this:  
 *   
 */
```

Javadoc Tags

Es gibt spezielle Schlüsselwörter welche als Tags bezeichnet werden. Tags werden mit einem beginnenden @ gekennzeichnet.

author	deprecated	docRoot
exception	inheritDoc	link
linkplain	param	return
see	serial	serialData
serialField	since	throws
value	version	

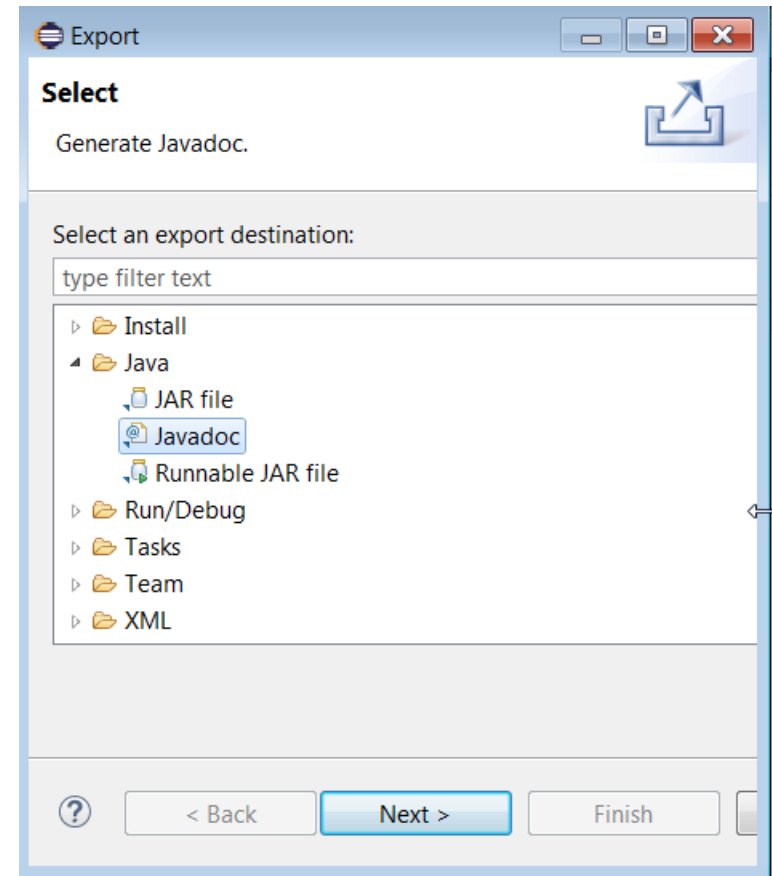
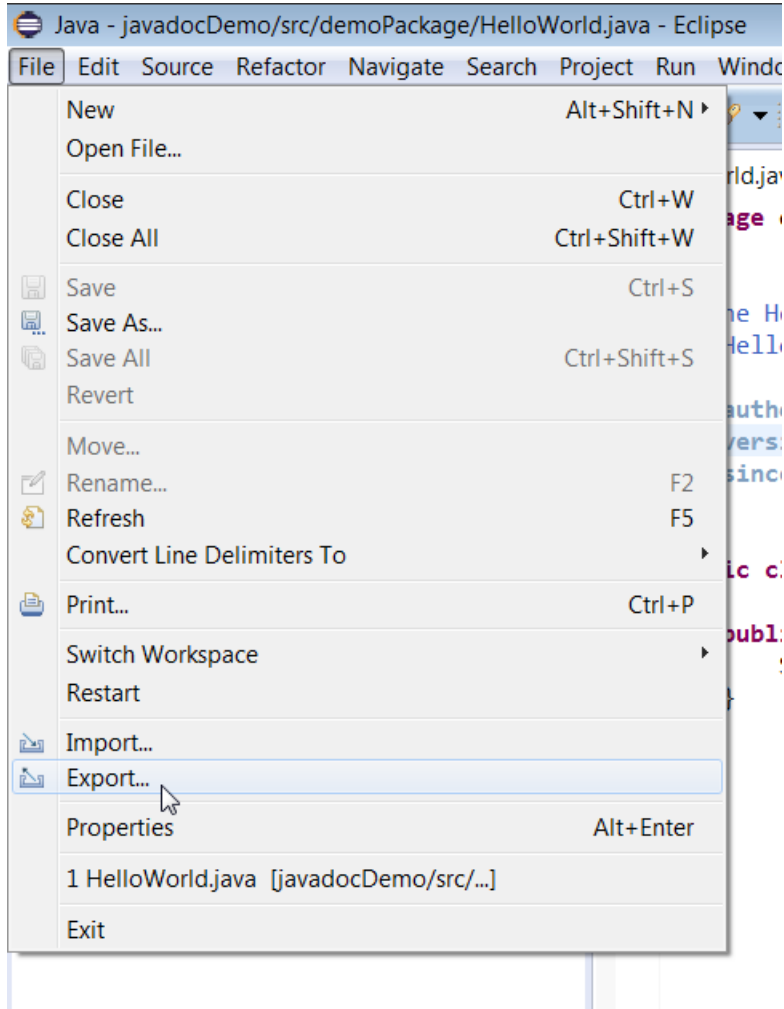
Spec unter

<http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadoctags>

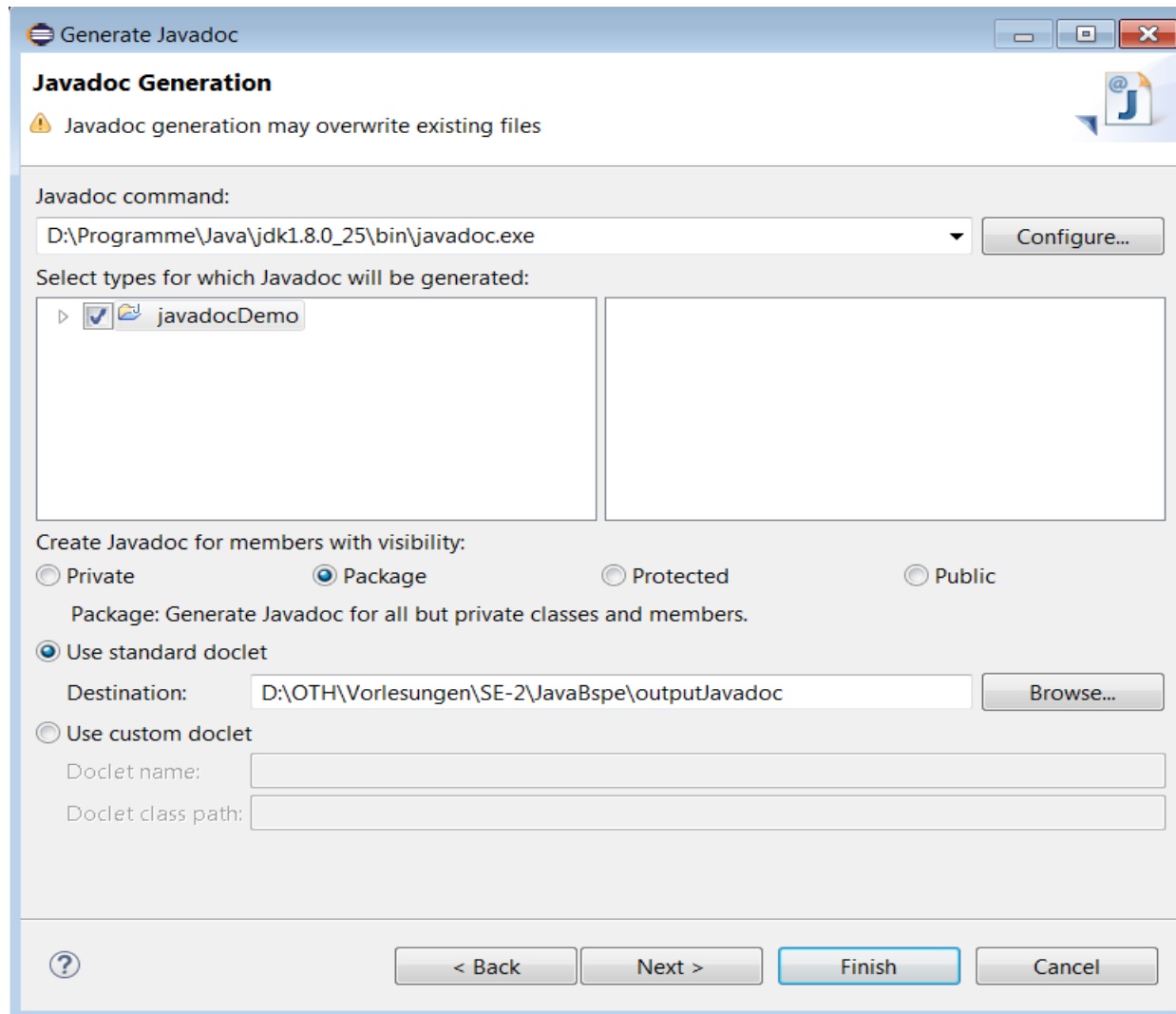
Die wichtigsten Javadoc Tags erklärt

Kommentar	Beschreibung	Beispiel
<code>@param</code>	Beschreibung der Parameter	<code>@param a A Value.</code>
<code>@see</code>	Verweis auf ein anderes Paket, einen anderen Typ, eine andere Methode oder Eigenschaft	<code>@see java.util.Date @see java.lang.String#length()</code>
<code>@version</code>	Version	<code>@version 1.12</code>
<code>@author</code>	Schöpfer	<code>@author Christian Ullenboom</code>
<code>@return</code>	Rückgabewert einer Methode	<code>@return Number of elements.</code>
<code>@exception/@throws</code>	Ausnahmen, die ausgelöst werden können	<code>@exception NumberFormatException</code>
<code>{@link Verweis}</code>	Ein eingebauter Verweis im Text im Code-Font. Parameter wie bei <code>@see</code>	<code>{@link java.io.File}</code>
<code>{@linkplain Verweis}</code>	Wie <code>{@link}</code> , nur im normalen Font	<code>{@linkplain java.io.File}</code>
<code>{@code Code}</code>	Quellcode im Code-Zeichensatz – auch mit HTML-Sonderzeichen	<code>{@code 1 ist < 2}</code>
<code>{@literal Literale}</code>	Maskiert HTML-Sonderzeichen. Kein Code-Zeichensatz	<code>{@literal 1 < 2 && 2 > 1}</code>
<code>@category</code>	Für Java 7 oder 8 geplant: Vergabe einer Kategorie	<code>@category Setter</code>

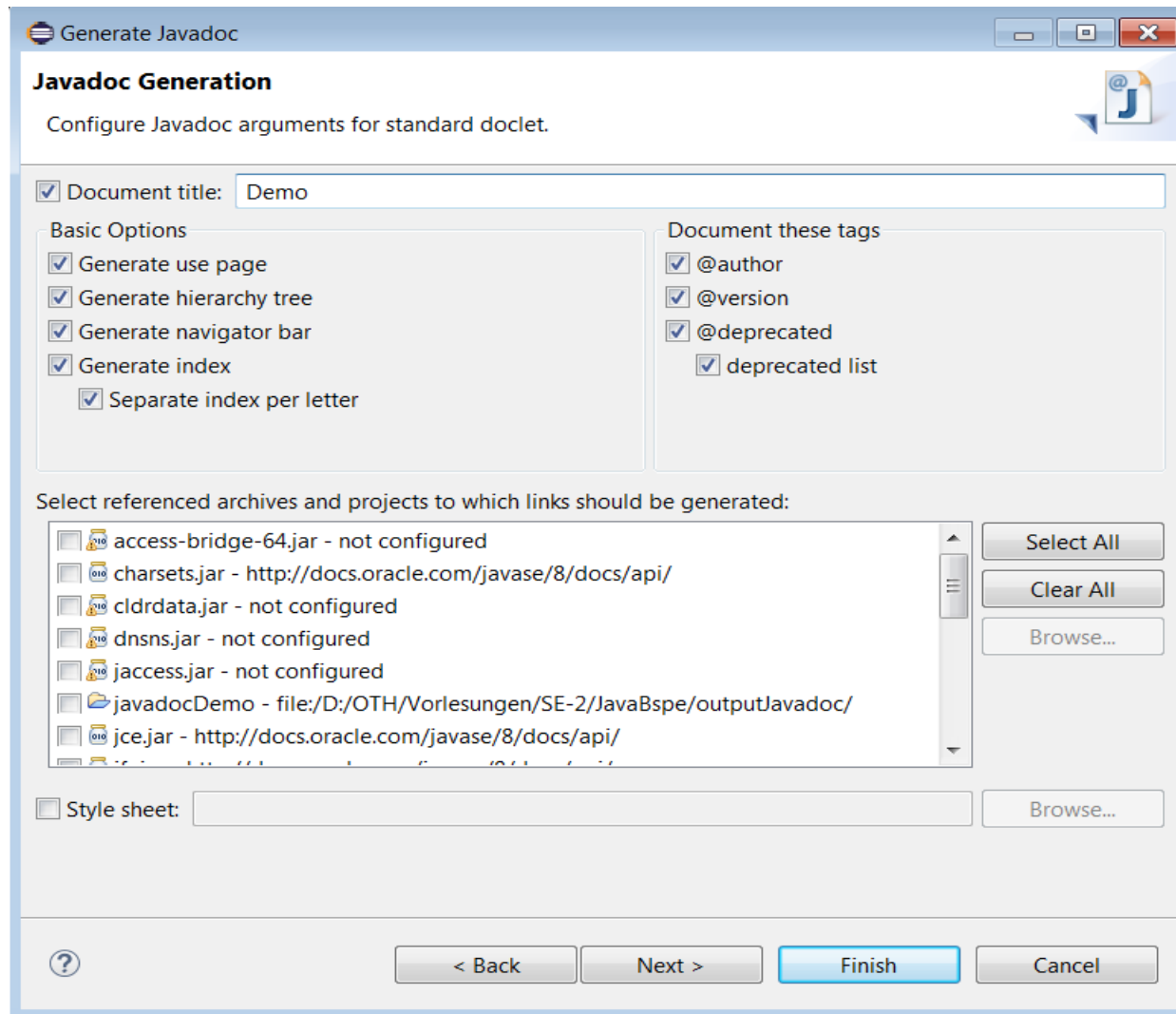
Aufruf von javadoc in Eclipse



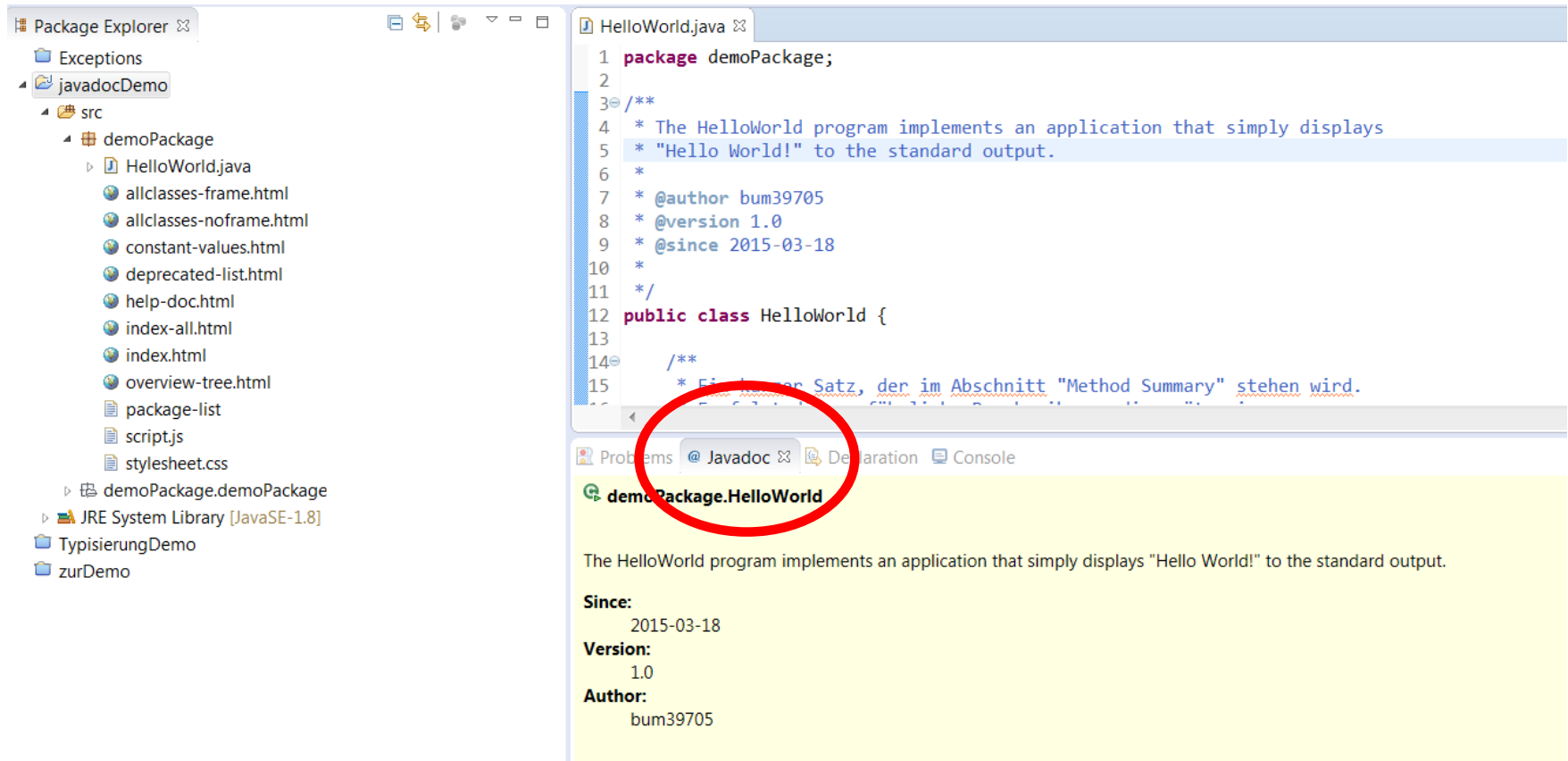
Aufruf von javadoc in Eclipse



Aufruf von javadoc in Eclipse



Vorschau von javadoc in Eclipse



The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer displays the project structure, including the 'javadocDemo' package and its 'src' directory. The 'HelloWorld.java' file is selected. The main editor shows the source code of 'HelloWorld.java', which includes a package declaration, a class comment, and a public class definition. A red circle highlights the 'Javadoc' tab in the bottom right corner, which displays the generated javadoc for 'demoPackage.HelloWorld'. The javadoc content includes the class comment and the 'Since', 'Version', and 'Author' tags.

```

1 package demoPackage;
2
3 /**
4  * The HelloWorld program implements an application that simply displays
5  * "Hello World!" to the standard output.
6  *
7  * @author bum39705
8  * @version 1.0
9  * @since 2015-03-18
10 *
11 */
12 public class HelloWorld {
13
14     /**
15      * Ein kurzer Satz, der im Abschnitt "Method Summary" stehen wird.
16      */
17 }

```

demoPackage.HelloWorld

The HelloWorld program implements an application that simply displays "Hello World!" to the standard output.

Since:
2015-03-18

Version:
1.0

Author:
bum39705

Ausgabe anpassen

Javadoc verwendet standardmässig das Standard Doclet um die Ausgabe zu erzeugen.

Wenn man eine andere Ausgabe möchte, kann man ein anderes Doclet verwenden (bestehende oder selber eins schreiben)

➔ Unter <http://www.doclet.com/> finden Sie eine Liste mit Doclets

Requirements for Writing Java API Specifications (Oracle)

- ➔ <http://www.oracle.com/technetwork/java/javase/documentation/index-142372.html>
- ➔ Vorgaben und Beispiele für
 - ➔ Top Level Specification
 - ➔ Package Specification
 - ➔ Class/Interface Specification
 - ➔ Field Specification
 - ➔ Method Specification

Styleguide für javadoc

Siehe Unter

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#styleguide>

Beispiele unter

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#examples>

Nützliche Links

How to Write Doc Comments for the Javadoc Tool:

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

Javadoc reference pages:

<http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html#javadocdocuments>

Javadoc reference Guide:

<http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html>

Doxygen

- <http://www.doxygen.org>
- erstellt Dokumentation für C++,C,IDL, Java und C#

Doc++

- <http://docpp.sourceforge.net>
- erstellt Dokumentation für C++,C,IDL und Java

Doxygen:

- Klassendiagramme
- dependency graphs, inheritance diagrams, and collaboration diagrams
- Optionales Source Code Browsing
- Zusätzlicher tag Support
- Output in Tex Format

Javadoc

- Speziell für Java designed
- Integraler Bestandteil der Sprache → keine weitere Installation nötig.

Maven Integration mit beiden, Javadoc ignoriert unbekannte tags
→ gleichzeitig verwendbar.