

13. Programme werden durch Testen erst zu Software

...sonst bleiben sie Bananaware

Testen ist notwendig für biologisches Programmieren

Prof. Dr. rer. nat. Uwe Aßmann
Institut für Software- und
Multimediatechnik
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
Technische Universität Dresden
Version 19-0.8, 13.04.19

- 1) Warum Testen wichtig ist
- 2) Professionelle Softwareentwicklung
- 3) Vertragsüberprüfung
- 4) Testfalltabellen
- 5) Regressionstests mit dem JUnit-Rahmenwerk
- 6) Entwurfsmuster in JUnit

mit Notizenseiten



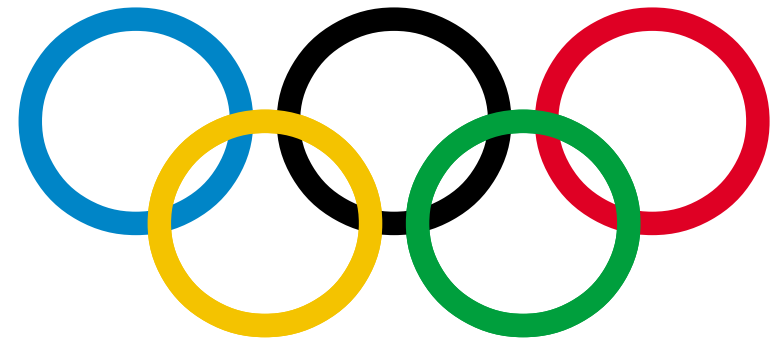
DRESDEN
concept
Exzellenz aus
Wissenschaft
und Kultur

- ▶ Obligatorische Literatur
 - Zuser Kap. 5+12 (ohne White-box tests)
 - ST für Einsteiger Kap. 5+12 in Teil 3
- ▶ Java documentation: <http://docs.oracle.com/javase/8/>
 - Essential Java tutorials on Exceptions and Pattern Matching
<http://docs.oracle.com/javase/tutorial/essential/index.html>
 - www.junit.org
 - Junit 3.x arbeitet noch ohne Metadaten (@Annotationen)
 - <https://sourceforge.net/projects/junit/files/junit/3.8.2/>
 - junit3.8.1/doc/cookstour/cookstour.htm. Schöne Einführung in Junit
 - junit3.8.1/doc/faq/faq.htm Die FAQ (frequently asked questions)
 - Achtung: JUnit 4 versteckt mehr Funktionalität in Metadaten-Attributen
 - <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>
 - <http://junit.sourceforge.net/doc/faq/faq.htm>

Weiterführende Literatur

- Andrew Hunt, David Thomas. The pragmatic programmer. Addison-Wesley. Deutsch: Der Pragmatische Programmierer. Hanser-Verlag. Leseprobe:
 - http://www.beck-shop.de/fachbuch/leseprobe/9783446223097_Excerpt_004.pdf
- Uwe Vigerschow. Objektorientiertes Testen und Testautomatisierung in der Praxis. Konzepte, Techniken und Verfahren. Dpunkt-Verlag, 2005.
- Frank Westphal. Testgetriebene Entwicklung mit jUnit und FIT. dpunkt Verlag.
 - Freies pdf: http://www.frankwestphal.de/ftp/Westphal_Testgetriebene_Entwicklung.pdf

- ▶ Was ist der Unterschied zwischen Programmieren und *sozialem* Programmieren?
 - Welches sind die sozialen Reifegrade von Software
- ▶ Was ist der Unterschied zwischen Software und Programmen?
- ▶ Was ist der Unterschied zwischen Klassen und Komponenten?
- ▶ Was sind die 5 olympischen Ringe der Software? Warum hilft die olympische Dekomposition, den sozialen Reifegrad von Software zu verbessern?
- ▶ Wie sind die Ringe als Komponenten in der Test-Umgebung einzuordnen?
- ▶ Was ist ein Regressionstest?



OUR GOALS

13.1. Testen als stichprobenartige Verifikation

... Testen macht Programme reif und “sozial”...



Beweis durch Probe

- ▶ Wie prüft man, ob ein Array sortiert ist?

```
int myArray[20]
```

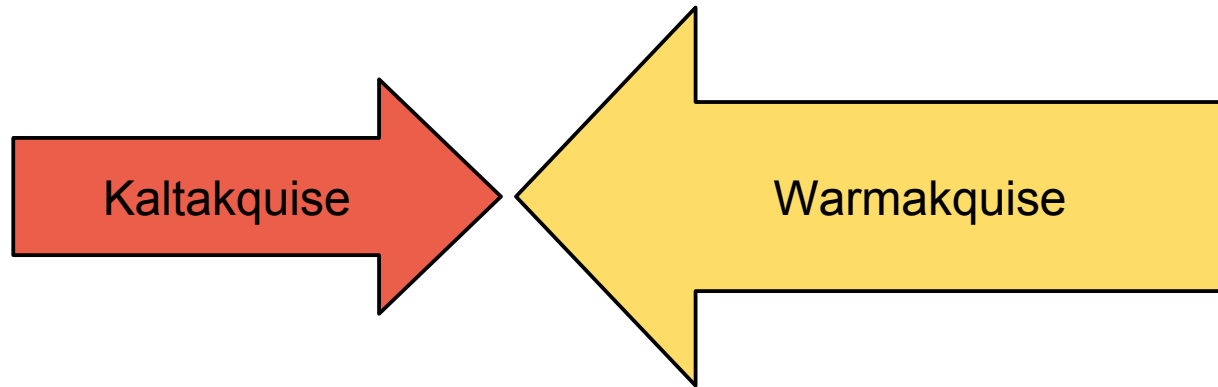
Mit “Beweis durch Probe”

```
boolean testArrayIsSorted(int myArray[20]) {
    for (int counter = 0; counter < 20; counter ++); {
        if (counter == 0) continue;
        if (myArray[counter] < myArray[counter-1]) {
            // A counterexample found: array is not sorted
            System.out.println("Array not sorted at index "
                +counter);
            return false;
        }
        // No counterexample found; array is sorted
    }
    return true;
}
```

Problem

- ▶ Leider können nicht alle Algorithmen und Programme durch “Beweis durch Probe” geprüft werden
- ▶ “Probe”-Verfahren existieren nicht immer (wie in der Schule)
- ▶ Was tut man da?

König Kunde



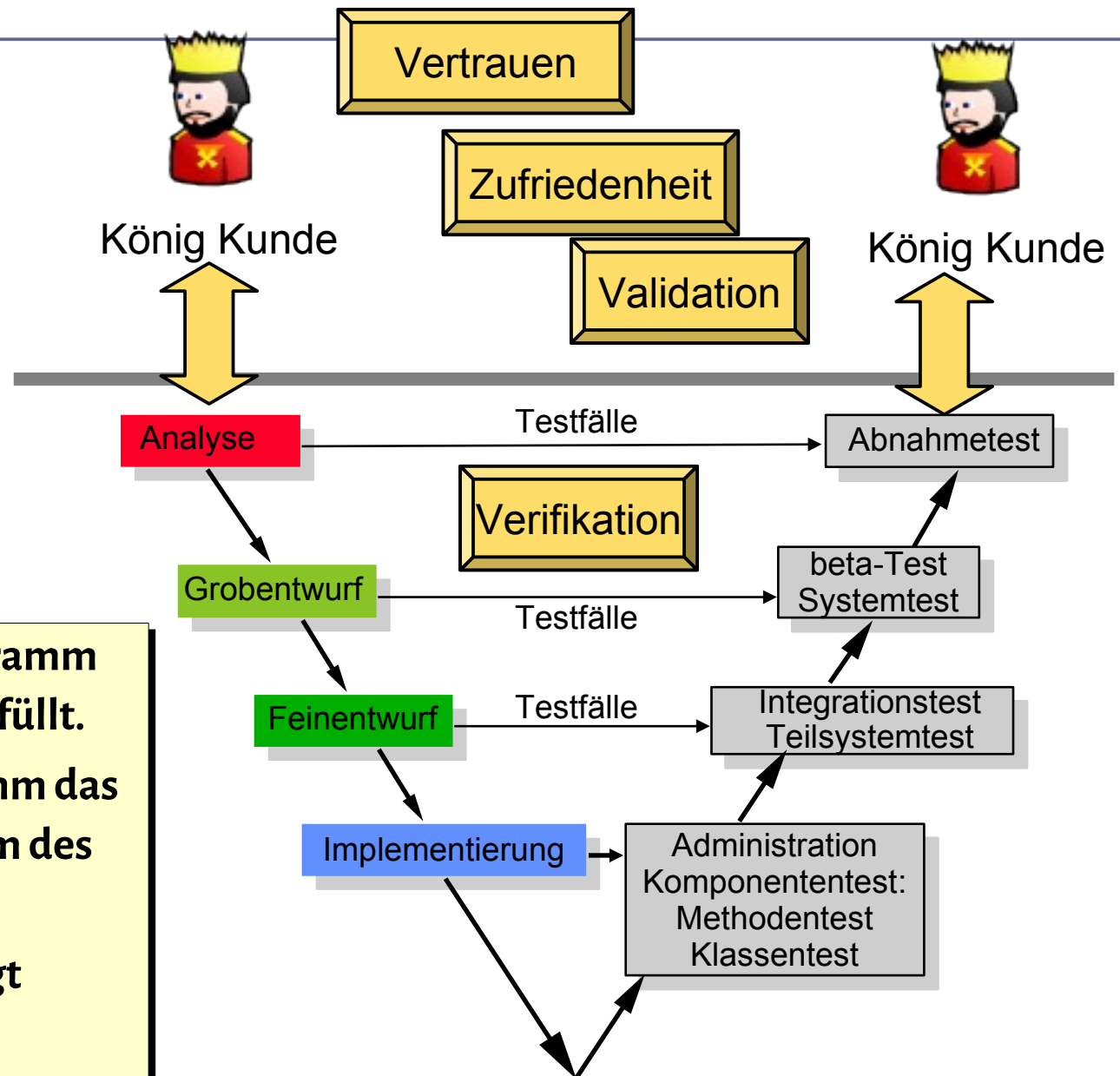
Gesetz 49 (PP): "Bananaware":

Testen Sie Ihre Software, sonst tun es die Anwender!

Gesetz 32 (PP): Ein totes Programm richtet weniger Schaden an als ein schrottreifes.

Verifikation, Validation, Kundenzufriedenheit und -Vertrauen

- ▶ V-Modell (Boehm 79)
- ▶ (Formale) Verifikation
- ▶ Beweis durch Probe
- ▶ Testen (Stichprobe)
 - Test-Abdeckung
 - Testreifegrad
- ▶ Testprozess
- ▶ Validation



Verifikation zeigt, dass das Programm seine Spezifikation richtig erfüllt.
Validation zeigt, dass das Programm das richtige Problem, das Problem des Kunden, löst.
Kundenzufriedenheit erzeugt Kundenvertrauen.
Kundenvertrauen erzeugt Geschäft.



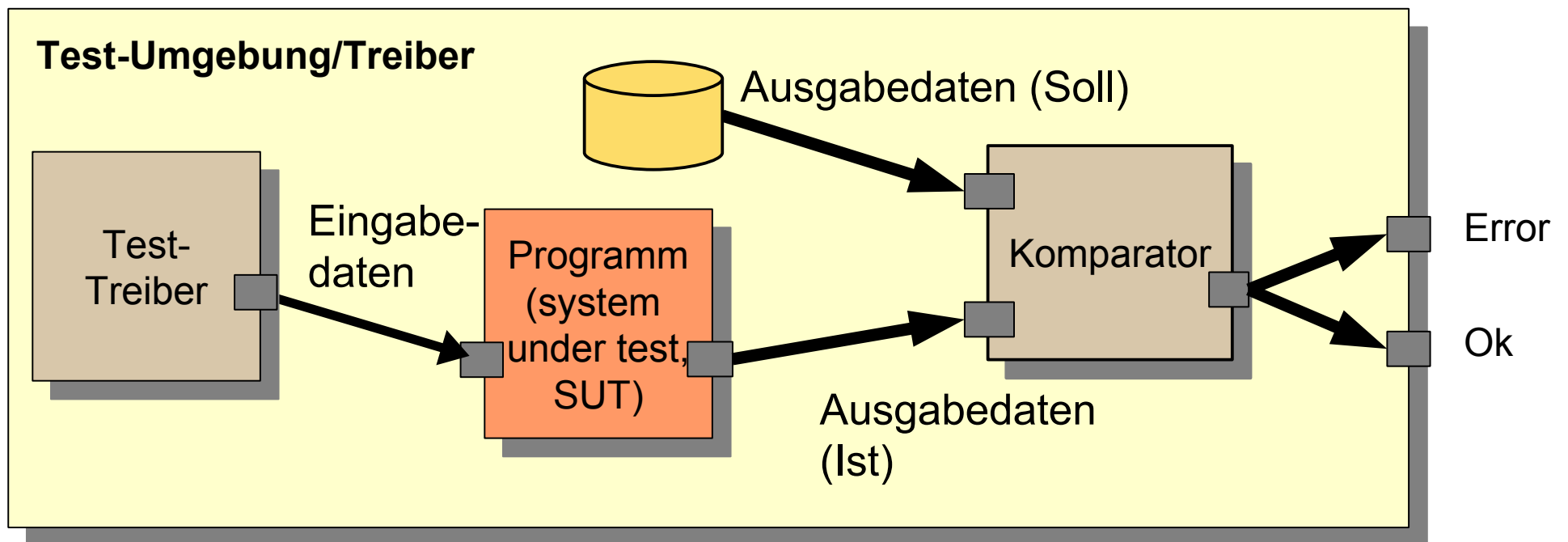
Testen ist Verifikation mittels Stichprobe

Testen besteht aus dem Nehmen von Stichproben:
Testing shows the presence of bugs, but never their absence (Dijkstra)

“Software” hat eine test-getriebene Architektur

- ▶ Solange ein Programm keine test-getriebene Architektur hat, ist es keine Software
- ▶ Andernfalls ist es “bananaware”

Gesetz 63 (PP): Das Programmieren ist nicht getan, bis alle Tests erfolgreich waren



Testgetriebene Entwicklung (Test-Driven Development, TDD, Test-First Development)

Gesetz 62 des Pragmatischen Programmierers: *Testen Sie frühzeitig, häufig und automatisch*

Im TDD spielt die Testsuite die Rolle des Kunden

Bananaware-Gesetz (PP 49b):

Entweder die Software reift beim Kunden (bananaware) oder beim Testen (profiware).

Methoden des Test-First Development

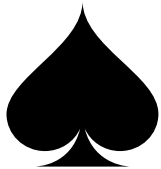
▶ Schreiben der Tests von Hand mit **Test-Framework**



- Test-Treiber
- Test-Tabelle
 - Eingabedaten
 - Ausgabedaten
- Beweis durch Probe finden

▶ **Qualitätsmanagementprozess** einrichten und verbessern (→ ST-2)

- Testsuite kaufen



▶ **Verträge (Administration)** schreiben

- Teile der Tests als Verträge übernehmen
- Innere Checks schreiben, um die Verträge zu überprüfen



▶ **Generierung** mit Werkzeug (→ ST-2)

- Test-Treibergenerierung
- Test-Datengenerierung



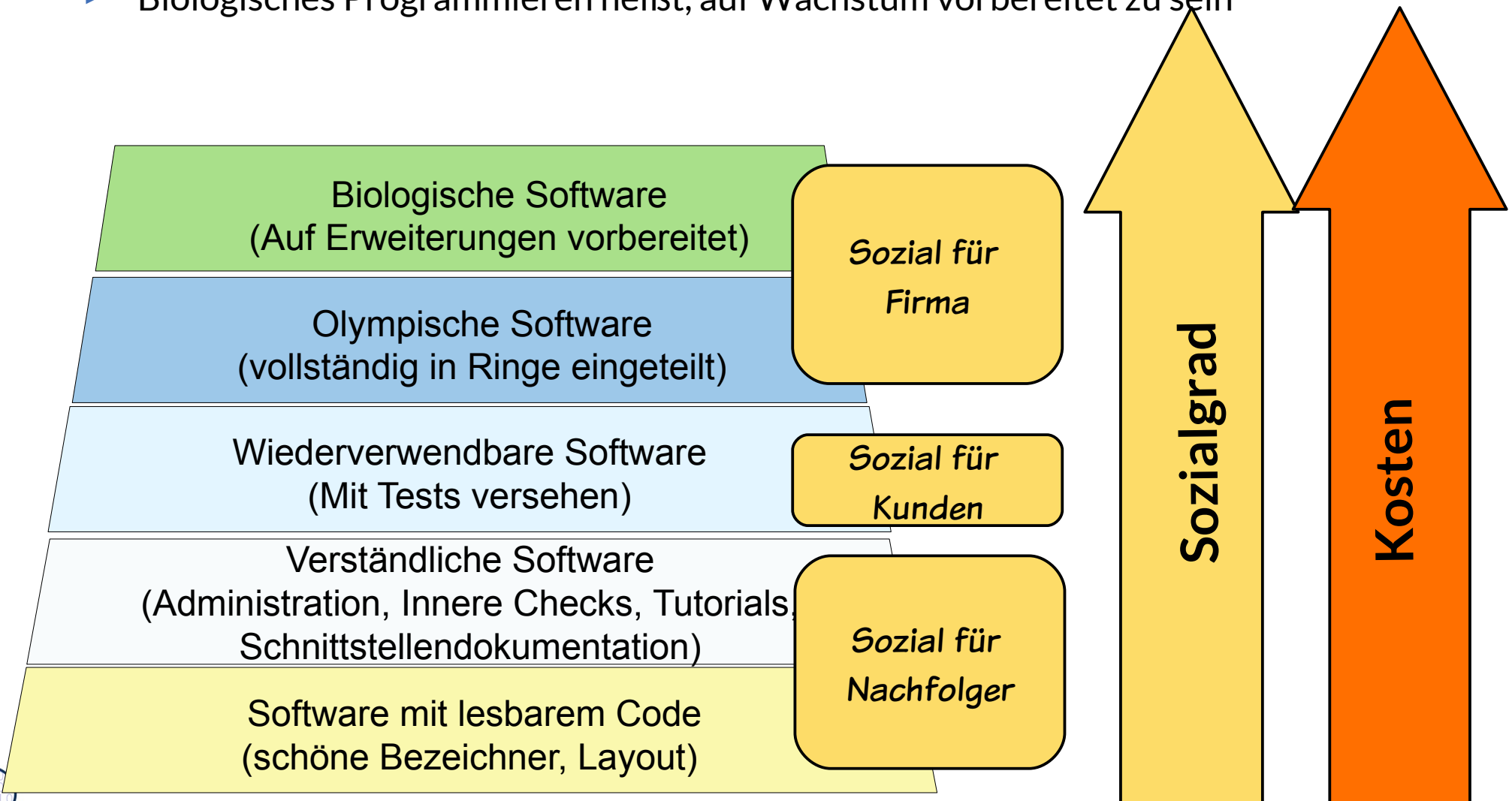
13.2. “Professional Programming is Social Programming”

- ▶ Soziales Programmieren ist *Programmieren für andere*, die die eigene Software wiederverwenden und in ihre Pflege übernehmen
 - Für lange lebende Software
 - Für Software, die sich erweitern lässt
- ▶ Programmieren muss sozial sein, alles andere ist nicht professionell
- ▶ Soziales Programmieren ist “olympisch” und “biologisch”

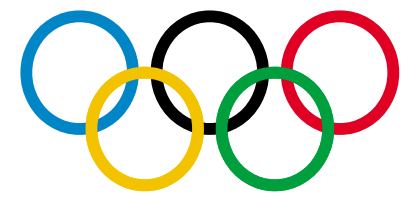


Der soziale Reifegrad von Software

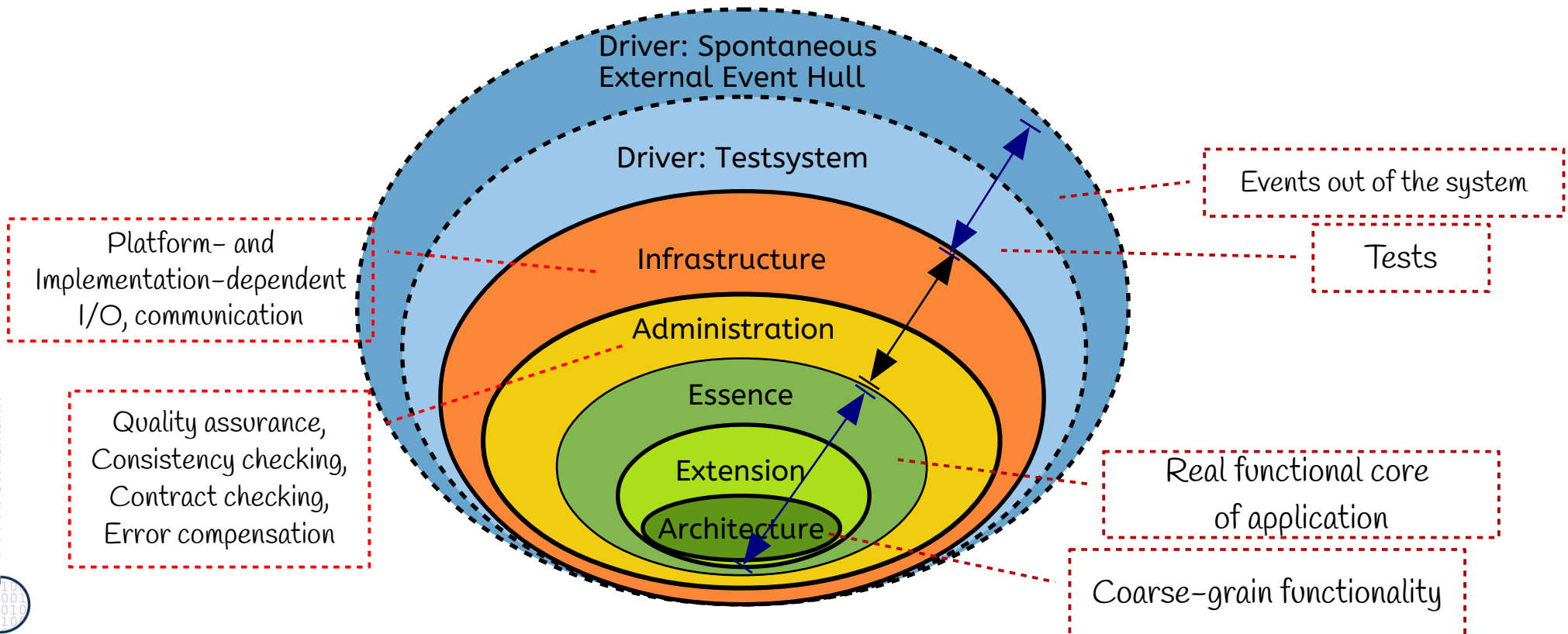
- ▶ Programmieren muss sozial sein, d.h. nutzbar, wiederverwendbar, erweiterbar.
- ▶ Alles andere ist nicht professionell
- ▶ Biologisches Programmieren heißt, auf Wachstum vorbereitet zu sein



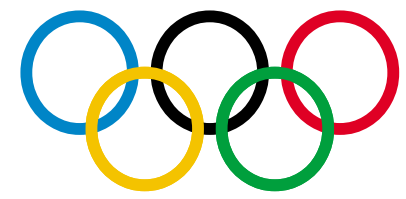
Olympische Dekomposition von Software in Ringe (Essentielle Dekomposition)



- ▶ **Software** hat 5 Ringe (*olympische* oder *essentielle Dekomposition* in 5 Aspekte):
 - **(Funktionale) Essenz** sind Funktionen unabhängig von der unterliegenden Technologie
 - **Architektur** ist ein Unter-Ring der Essenz, der grobkörnige Funktionalität liefert
 - **Administration** sichert die Qualität des Systems
 - **Infrastruktur (Middleware)** bietet die technologieabhängigen Funktionen an
 - **2 Externe Treiber.Ringe** treiben das System: entweder die Umgebung, die spontan Ereignisse und Eingabedaten generiert, oder das **Testsystem**



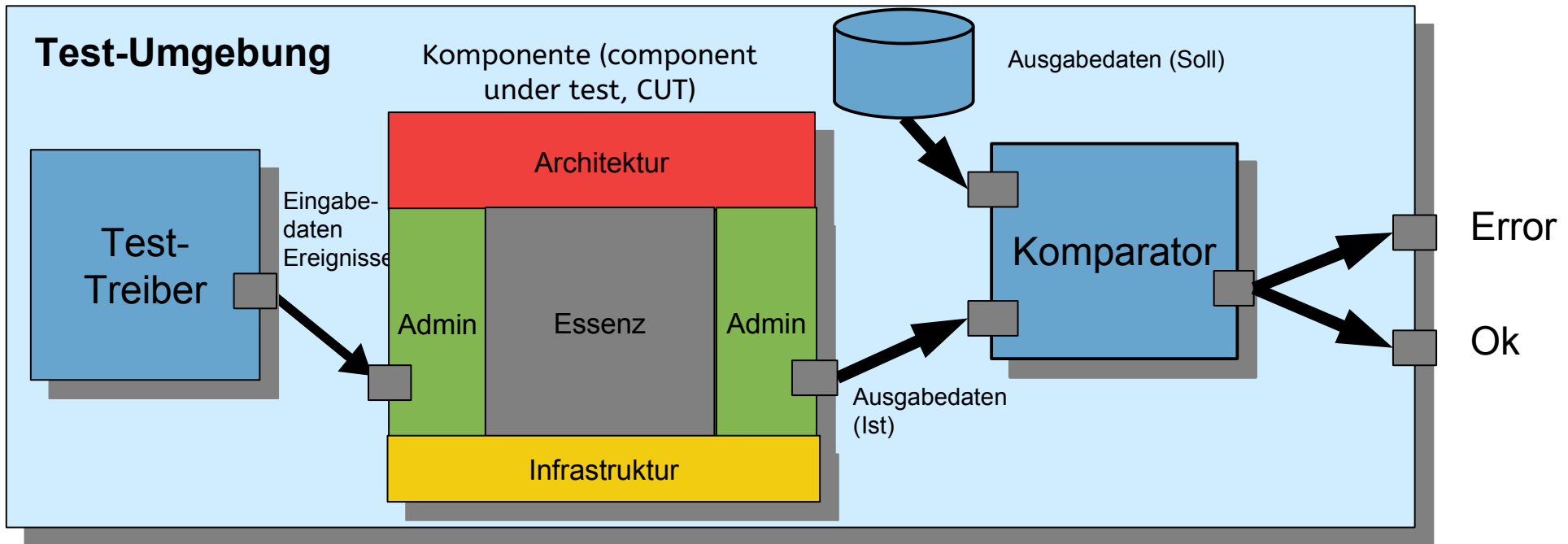
Soziale Softwareentwicklung: Klassen müssen Komponenten werden



Def.: Software-Komponenten sind wiederverwendbare Programmeinheiten. Sie sind Ergebnis „sozialer“ Softwareentwicklung.

Im einfachsten Fall sind sie Klassenpakete mit „olympischen Ringen“:

- klaren Schnittstellen,
- gut abdeckender Testtreiber-Hülle
- gut abdeckendem Administrationsring
- mit Aufrufen an Infrastruktur.



13.3. Vertragsprüfung (Ring der Administration)



Jede Code-Einheit sollte mit einem *Administrationscode* ausgestattet sein, der die Gültigkeit der

- Eingabedaten
- Ausgabedaten
- Internen Daten

prüft, und bei Fehlern Ausnahmen auslöst und behandelt.
(Administration)



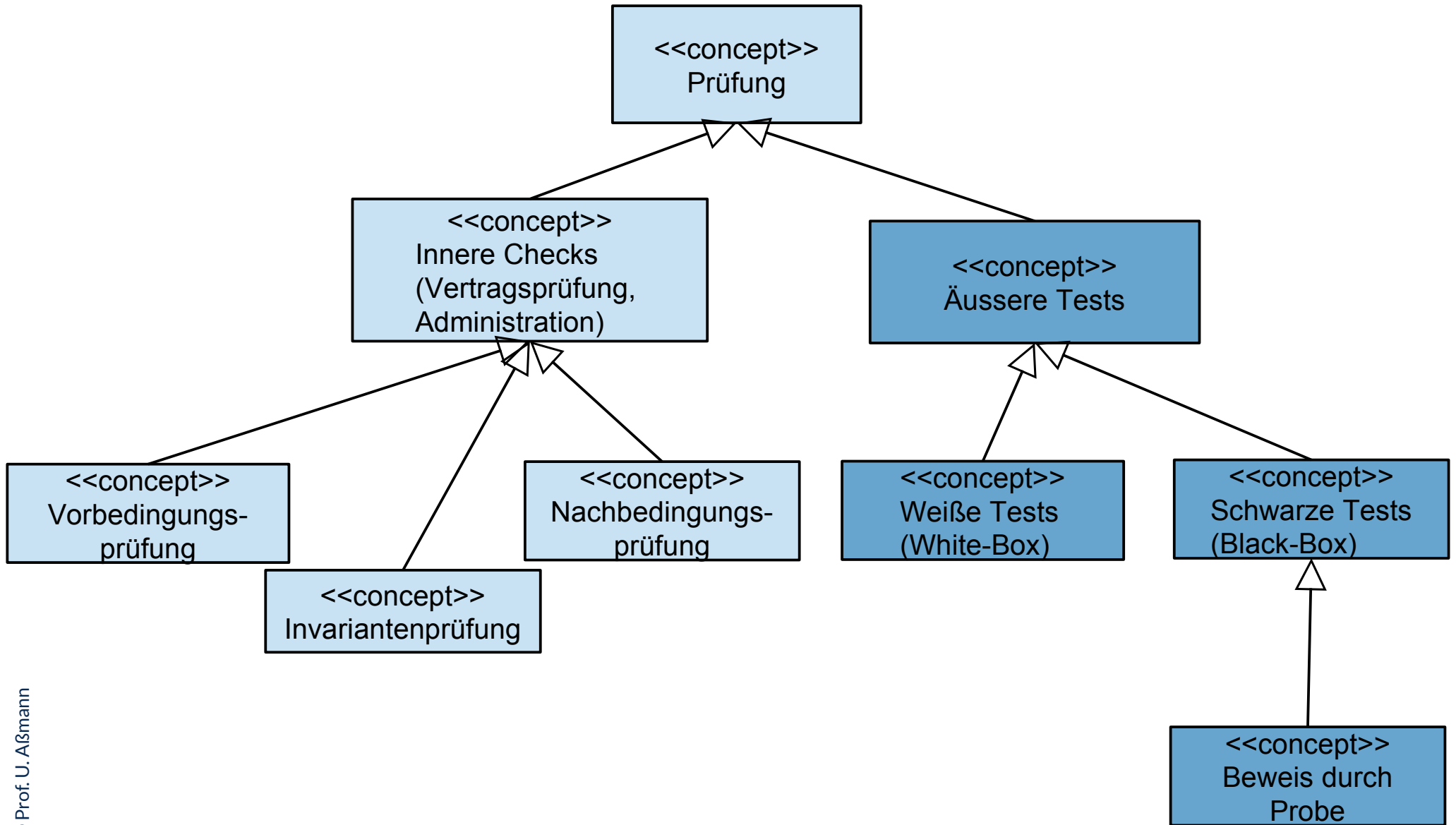
Beispiel: Wie schreibt man einen Test für eine Methode?

DateSimple.java

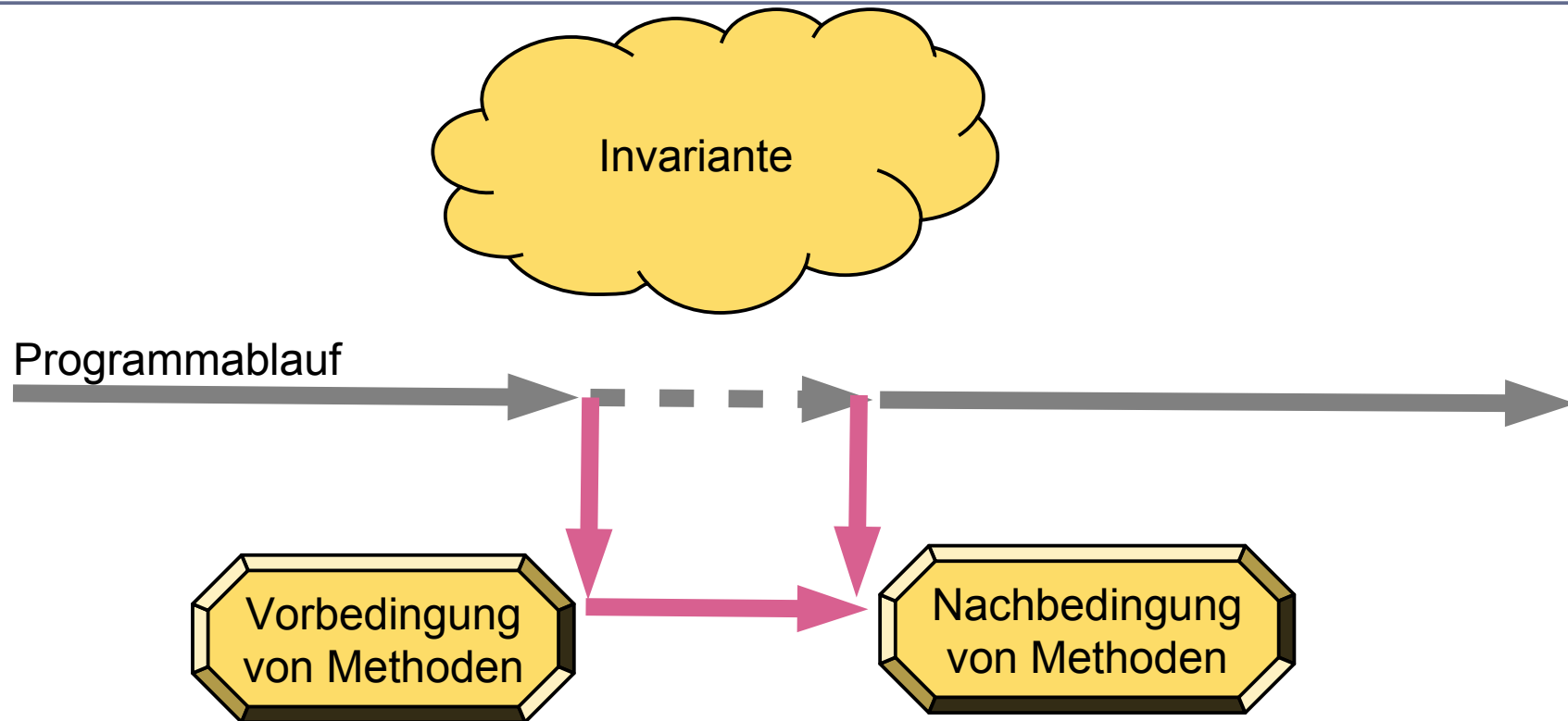
- ▶ Wie testet man `parseDay(String d)`?

```
// A class for standard representation of dates.
public class DateSimple {
    private String myDate;
    public int day; public int month; public int year;
    public DateSimple(String date) { myDate = date;
        parseDate(); }
    public int parseDate() {
        day = parseDay(myDate);
        month = parseMonth(myDate);
        year = parseYear(myDate);
    }
    public int parseDay(String d) {
        if (d.matches("\\d\\d\\.\\d\\d\\.\\d\\d\\d\\d")) {
            // German numeric format day.month.year
            return Integer.parseInt(d.substring(0,2));
        } else {
            .. other formats...
        }
    }
}
```

Antwort: Innere Checks und äussere Tests (Begriffshierarchie)



13.2.1 Innere Checks: Vertragsprüfung für eine Methode (“Design by Contract”)



Gesetz 31 (PP): **Verwenden Sie Design by Contract (Vertragsprüfung), damit der Quelltext nicht mehr und nicht weniger tut, als er vorgibt.**

Gesetz 33 (PP): **Verhindern Sie das Unmögliche mit Zusicherungen.**

Vertragsprüfung für eine Methode mit Rückgabe eines Fehlercodes

DateSimple.java

23 Softwaretechnologie (ST)

- ▶ Preconditions werden im Prolog, Postconditions im Epilog einer Methode geprüft; Invarianten überall (Ring der Administration)
- ▶ Abbruch bei Fehlschlag der Prüfung

Vorbedingung (precondition):
d ist ein String - d ist nicht leer

Invarianten (invariants):
d ist mindestens 10 Zeichen
lang (Datum plus Trenner)

```
public int parseDay(String d) {  
    // Check parameter 1  
    if (d.equals("")) { System.err.println("empty"); return 0; }  
    if (d.size() < 10) { System.err.println("size too small"); return 0; }  
    if (d.matches("\\d\\d\\.\\d\\d\\.\\d\\d\\.\\d\\d\\.\\d\\d\\.\\d\\d")) {  
        if (d.size() < 10) System.err.println("size too small"); return 0; }  
        // German numeric format day. month. Year  
        int day = Integer.parseInt(d.substring(0,2));  
        // Check return  
        if (day < 1 || day > 31) System.out.println("wrong");  
    } else {  
        .. other formats...  
    }  
    if (d.size() < 10) { System.err.println("size too small"); return 0; }  
    // check return  
    if (day < 1 || day > 31) { System.err.println("illegal day"); return 0; }  
    return day;  
}
```

Nachbedingung (postcondition):
Ein int wird zurückgegeben
Zwischen 1 und 31

Metrik für die Güte der Administrationsrings einer Methode

- ▶ Zähle die Checks auf Parameter und vergleiche mit der Zahl der Parameter
 - Abdeckungsmetrik (Coverage-Metrics): Wie viele der Parameter haben eine zugeordnete Prüfung?
- ▶ Zähle die Invarianten-Checks
- ▶ Zähle die Return-Checks
 - Abdeckungsmetrik (Coverage-Metrics): Wie viele der Returnpunkte haben eine zugeordnete Prüfung?
- ▶ Zähle die Kommentare

Solange ein Programm keinen Administrationsring hat, ist es nicht verständlich, noch nutzbar!

Das Programmieren von Administration kostet Zeit!

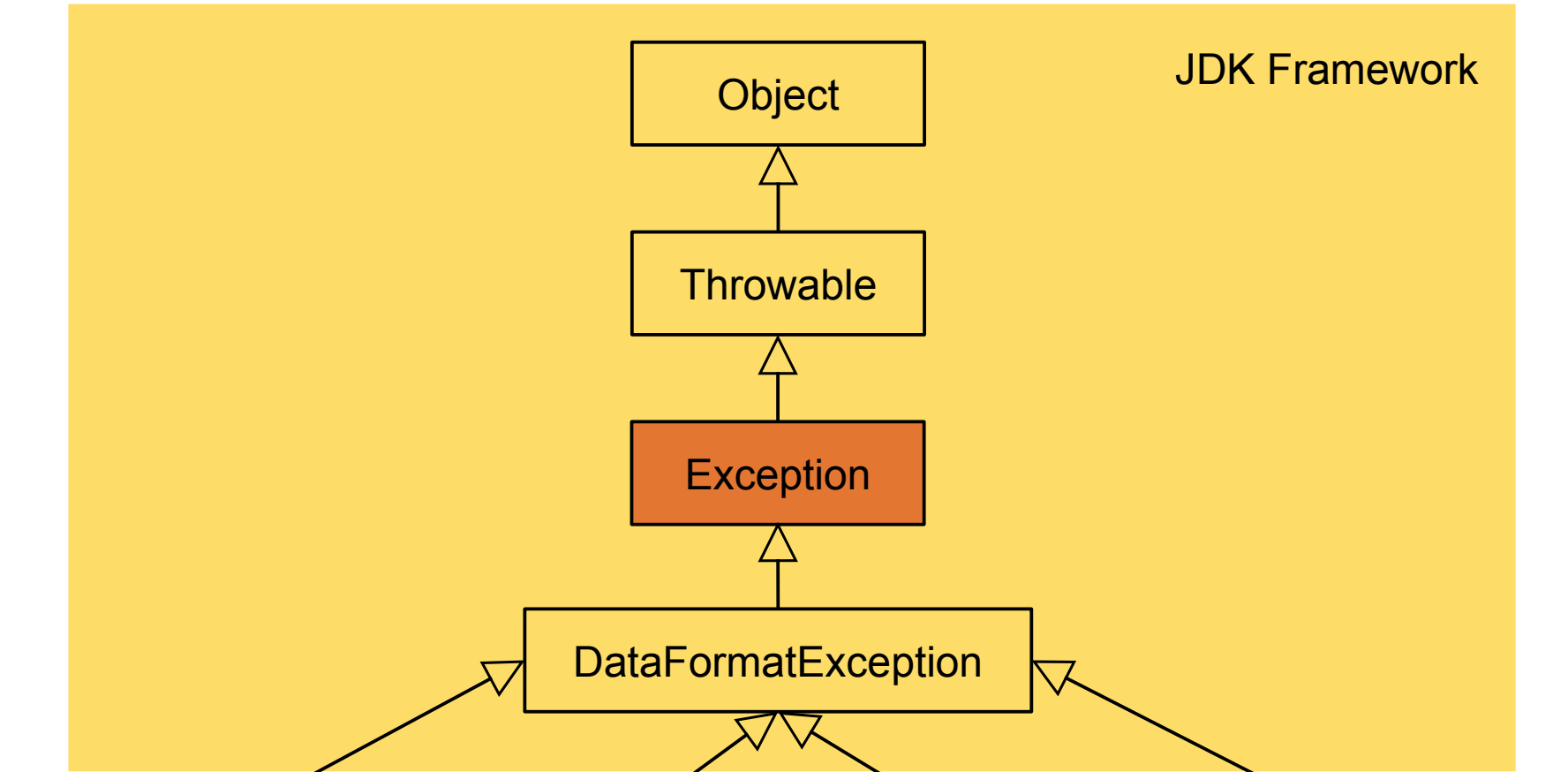
13.2.2 Vertrag einer Methode – Prüfen durch assert

- ▶ `assert()`, eine Standardmethode, bricht das Programm bei Verletzung einer Vertragsbedingung ab
- ▶ Achtung: Bedingungen müssen dual zu den Bedingungen der vorgenannten Ausnahmen formuliert sein!
- ▶ Achtung: rufe java mit `-ea` auf: `$ java -ea C.class`

```
public int parseDay(String d) {
    assert(!d.equals(""));
    assert(d.size() >= 10);
    if (d.matches("\\d\\d\\.\\d\\d\\.\\d\\d\\.\\d\\d\\.\\d\\d")) {
        assert(d.size() >= 10);
        // German numeric format day. month. Year
        int day = Integer.parseInt(d.substring(0,2));
        assert(day >= 1 and day <= 31);
    } else {
        .. other formats...
    }
    assert(d.size() >= 10);
    assert(day >= 1 and day <= 31);
    return day;
}
```

13.2.3 Auslösen von Ausnahmen (Exception Objects) bei Vertragverletzung

JDK Framework



Application



Java™ Platform Standard Ed. 6

All Classes

Packages

[java.applet](#)
[java.awt](#)
[java.awt.color](#)
[java.awt.datatransfer](#)
[java.awt.dnd](#)
[java.awt.event](#)
[java.awt.font](#)
[java.awt.geom](#)

Event

[EventContext](#)
[EventDirContext](#)
[EventException](#)
[EventFilter](#)
[EventHandler](#)
[EventListener](#)
[EventListener](#)
[EventListenerList](#)
[EventListenerProxy](#)
[EventObject](#)
[EventQueue](#)
[EventReaderDelegate](#)
[EventSetDescriptor](#)
[EventTarget](#)
[ExcC14NParameterSpec](#)
[Exception](#)
[ExceptionDetailMessage](#)
[ExceptionInInitializerError](#)
[ExceptionList](#)
[ExceptionListener](#)
[Exchanger](#)
[ExecutableElement](#)
[ExecutableType](#)
[ExecutionException](#)
[Executor](#)
[ExecutorCompletionService](#)
[Executors](#)
[ExecutorService](#)
[ExemptionMechanism](#)
[ExemptionMechanismException](#)
[ExemptionMechanismSpi](#)
[ExpandVetoException](#)
[ExportException](#)
[Expression](#)
[ExtendedRequest](#)
[ExtendedResponse](#)
[Externalizable](#)

Overview Package **Class** Use Tree Deprecated Index Help

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

java.lang

Class Exception

[java.lang.Object](#)

└─ [java.lang.Throwable](#)

└─ [java.lang.Exception](#)

All Implemented Interfaces:

[Serializable](#)

Direct Known Subclasses:

[AclNotFoundException](#), [ActivationException](#), [AlreadyBoundException](#), [ApplicationException](#), [AWTException](#),
[BackingStoreException](#), [BadAttributeValueTypeException](#), [BadBinaryOpValueTypeException](#), [BadLocationException](#),
[BadStringOperationException](#), [BrokenBarrierException](#), [CertificateException](#), [ClassNotFoundException](#),
[CloneNotSupportedException](#), [DataFormatException](#), [DatatypeConfigurationException](#), [DestroyFailedException](#),
[ExecutionException](#), [ExpandVetoException](#), [FontFormatException](#), [GeneralSecurityException](#), [GSSEException](#),
[IllegalAccessException](#), [IllegalClassFormatException](#), [InstantiationException](#), [InterruptedException](#), [IntrospectionException](#),
[InvalidApplicationException](#), [InvalidMidiDataException](#), [InvalidPreferencesFormatException](#),
[InvalidTargetObjectTypeException](#), [InvocationTargetException](#), [IOException](#), [JAXBException](#), [JMEException](#),
[KeySelectorException](#), [LastOwnerException](#), [LineUnavailableException](#), [MarshalException](#), [MidiUnavailableException](#),
[MimeTypeParseException](#), [MimeTypeParseException](#), [NamingException](#), [NoninvertibleTransformException](#),
[NoSuchFieldException](#), [NoSuchMethodException](#), [NotBoundException](#), [NotOwnerException](#), [ParseException](#),
[ParserConfigurationException](#), [PrinterException](#), [PrintException](#), [PrivilegedActionException](#), [PropertyVetoException](#),
[RefreshFailedException](#), [RemarshalException](#), [RuntimeException](#), [SAXException](#), [ScriptException](#),
[ServerNotActiveException](#), [SOAPException](#), [SQLException](#), [TimeoutException](#), [TooManyListenersException](#),
[TransformerException](#), [TransformException](#), [UnmodifiableClassException](#), [UnsupportedAudioFileException](#),
[UnsupportedCallbackException](#), [UnsupportedFlavorException](#), [UnsupportedLookAndFeelException](#),
[URISyntaxException](#), [URISyntaxException](#), [UserException](#), [XAException](#), [XMLParseException](#),
[XMLSignatureException](#), [XMLStreamException](#), [XPathException](#)

```
public class Exception
extends Throwable
```

The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.

Since: JDK1.0

See Also: [Error](#), [Serialized Form](#)

file:///Users/ua1/Courses/ST1/Material/jdk-8-docs/api/index.html

Vertragsprüfung für eine Methode mit Exceptions

- ▶ Eine fehlgeschlagene Vertragsprüfung kann eine Ausnahme (exception) auslösen, mittels `throw`-Anweisung
 - Dazu muss ein Exception-Objekt angelegt werden
- ▶ Vorteil: Ursache des Fehlers kann in einem großen System weit transportiert werden, gespeichert werden, oder in eine Testumgebung zurückgegeben werden

DateWithExceptions.java

```
public int parseDay(String d) {
    if (d.equals("")) throw new DateInvalid();
    if (d.size() < 10) throw new DateTooShort();
    if (d.matches("\\d\\d\\.\\d\\d\\.\\d\\d\\d\\d")) {
        if (d.size() < 10) throw new DateTooShort();
        // German numeric format day.month.year
        int day = Integer.parseInt(d.substring(0,2));
        if (day < 1 || day > 31) throw new DayTooLarge();
    } else {
        .. other formats...
    }
    if (d.size() < 10) throw new DateTooShort();
    if (day < 1 || day > 31) throw new DateWrong();
    return day;
}
```

Java-Syntax für Ausnahmebehandlung im Aufrufer: Wie man aus dem Schlammassel wieder entkommt

```
class TotalDiv {  
    public static int tDiv (int x, int y)  
        throws ArithmeticException {  
        try {  
            return (x / y);  
        }  
        catch (ArithmeticException e) {  
            System.err.println(„Division by zero“);  
            throw new ArithmeticException();  
        }  
    }  
}
```

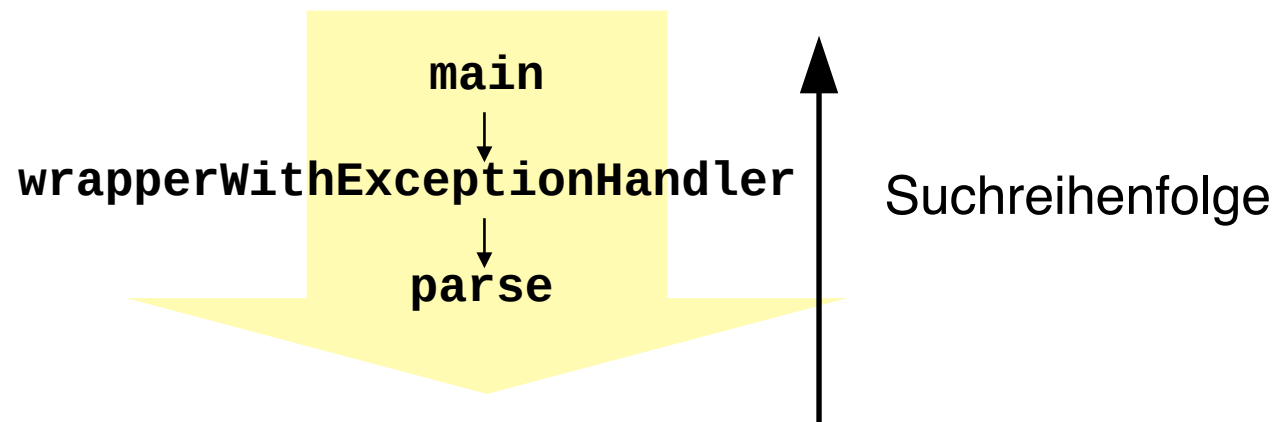
Schnittstellen-
Deklaration

Gekapselter
Block

Ausnahme-
Fänger

Dynamische Suche nach Ausnahmebehandlung

- ▶ Suche nach Abfangklausel (catch block) entlang der (dynamischen) Aufrufhierarchie nach außen:



- ▶ Bei mehreren Abfangklauseln an der gleichen Stelle der Hierarchie gilt die zuerst definierte Klausel:

```
try {  
  catch (DateInvalid e)  
  catch (DayTooLarge e)  
  catch (DateWrong e)
```

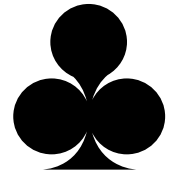
↓ Suchreihenfolge

Regeln zum Umgang mit Ausnahmen in der Administration



- ▶ Gesetz 33: **Verhindern Sie das Unmögliche mit Zusicherungen**
 - Vertragsüberprüfungen generieren Ausnahmen
- ▶ Gesetz des pragmatischen Programmierers 58: **Bauen Sie die Dokumentation ein**
 - Administration und Essenz trennen!
 - Ausnahmebehandlung niemals zur Behandlung normaler (d.h. häufig auftretender) Programmsituationen einsetzen
 - Ausnahmen sind Ausnahmen, regulärer Code behandelt die regulären Fälle!
- ▶ Gesetz 34: **Verwenden Sie Ausnahmen nur ausnahmsweise**
 - Nur die richtige Dosierung des Einsatzes von Ausnahmen ist gut lesbar
- ▶ Gesetz 35: **Führen Sie zu Ende, was Sie begonnen haben**
 - Auf keinen Fall Ausnahmen “abwürgen”, z.B. durch triviale Ausnahmebehandlung
 - Ausnahmen zu propagieren ist keine Schande, sondern erhöht die Flexibilität des entwickelten Codes.

13.4. Ring der Tests: Testfallspezifikation mit Testfalltabellen



13.1.4 Aufschreiben von Testfällen in Testfalltabellen

- ▶ Eine test-getriebene Architektur benötigt eine Spezifikation aller Testfälle
- ▶ Testfalltabellen enthalten Testfälle (**Gut-**, **Fehler-**, **Ausnahmefälle**) mit **Testdaten** und **sätzen**

Nr	Klasse	Eingabedaten	Ausgabedaten			Erwarteter Status
			day	month	year	
1	Gutfall	1. Januar 2006	1	1	2006	Ok
2	Gutfall	05/12/2008	5	12	2008	Ok
3	Gutfall	January 23, 2017	23	1	2017	Ok
4	Fehlerfall	44, 2007				Failure
5	Fehlerfall	Aup 23, 2005				Failure
6	Ausnahme	March 44, 2007	31	03	2007	Exception

Wdh.: Wie schreibt man einen Test für eine Methode?

- ▶ Wie testet man `parseDay(String d)`?

```
// A class for standard representation of dates.
public class Date {
    private String myDate;
    public int day; public int month; public int year;
    public Date(String date) { myDate = date; }
    public int parseDate() {
        day = parseDay(myDate);
        month = parseMonth(myDate);
        year = parseYear(myDate);
    }
    public int parseDay(String d) {
        if (d.matches("\\d\\d\\.\\d\\d\\.\\d\\d\\d\\d")) {
            // German numeric format day.month.year
            return Integer.parseInt(d.substring(0,2));
        } else {
            .. other formats...
        }
    }
}
/Users/ua1/Courses/ST1/Slides/JavaExamples/TestDate/DateSimple.java
```

Ein neuer Testfall wird aus Testfalltabelle konstruiert

- ▶ Testfälle (Testmethoden) werden in eine **Testfallklasse** geschrieben
 - Die Testdaten befinden sich in einer *Halterung (fixture)*
 - Eine Testfallklasse kann mehrere Testfälle aus der Testfalltabelle enthalten

```
public class DateTestCase {  
    Date d1;  
    Date d2;  
    Date d3;                                     Halterung (fixture)  
  
    public int testDate() {  
        // Init fixture (set up)  
        d1 = new Date(„1. Januar 2006“);  
        d2 = new Date(„05/12/2008“);  
        d3 = new Date(„January 23rd, 2009“);  
        // Processing  
        d1.parseDate(); d2.parseDate(); d3.parseDate();  
        // Checking results  
        assert(d1.day == 1); assert(d1.month == 1); assert(d1.year == 2006);  
        assert(d2.day == 5); assert(d2.month == 12); assert(d2.year == 2008);  
        assert(d3.day == 23); assert(d3.month == 1); assert(d3.year == 2009);  
    }  
}
```



13.5. Ring des Tests von Komponenten: Regressionstests mit dem JUnit-Rahmenwerk

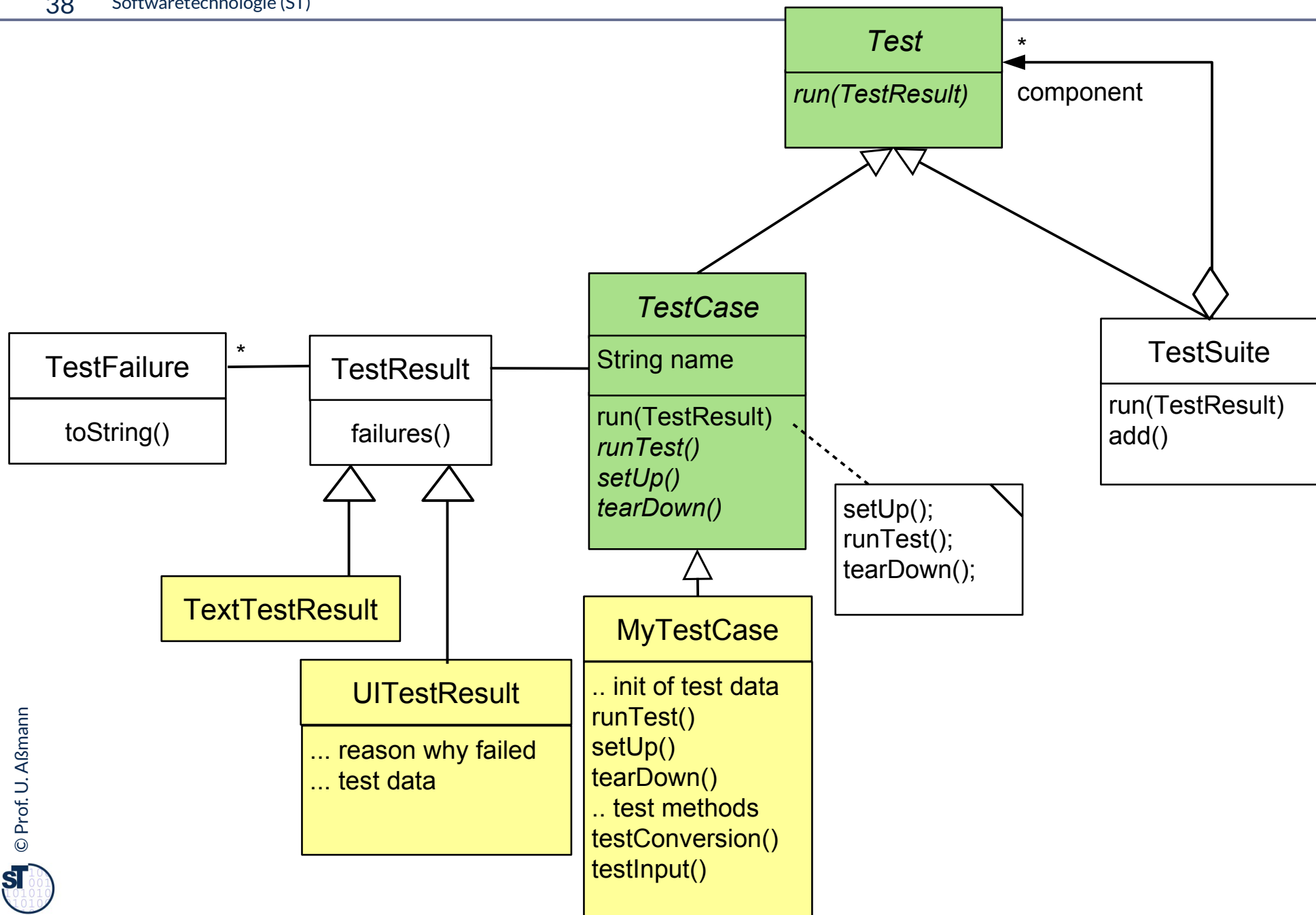
- ▶ **Regressionstest:** Automatisierter Vergleich von Ausgabedaten (gleicher Testfälle) unterschiedlicher Versionen des Programms.
 - Da zu großen Systemen mehrere 10000 Testdatensätze gehören, ist ein automatischer Vergleich unerlässlich.
 - Beispiel: Validierungssuiten von Übersetzern werden zusammen mit Regressionstest-Werkzeugen verkauft. Diese Werkzeuge wenden den Übersetzer systematisch auf alle Testdaten in der Validierungssuite an
- ▶ <https://en.wikipedia.org/wiki/JUnit>



Das JUnit Regressionstest-Framework

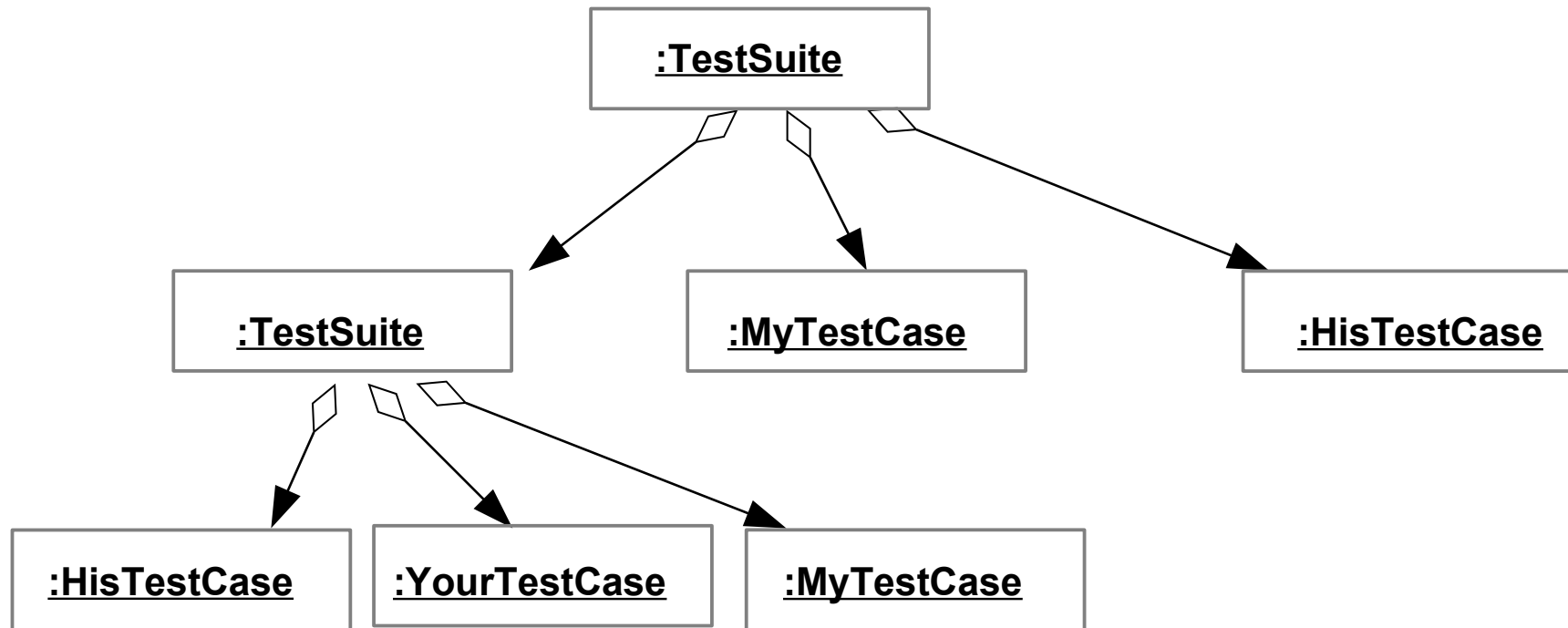
- ▶ Eine **Testsuite (Test-Ring)** werden heute aus Test-Frameworks gemacht
- ▶ **JUnit** www.junit.org ist ein technisches Java-Framework für Regressionstests, sowohl für einzelne Klassen (*unit test*), als auch für Systeme
 - Durchführung von Testläufen mit Testsuiten automatisiert
 - Eclipse-Plugin erhältlich
 - Mittlerweile für viele Programmiersprachen nachgebaut
- ▶ Junit 3.8.1: Good old plain Java
 - 88 Klassen mit 7227 Zeilen
 - im Kern des Rahmenwerks: 10 Klassen (1101 Zeilen)
- ▶ Testresultate:
 - Failure (Zusicherung wird zur Laufzeit verletzt)
 - Error (Unvorhergesehenes Ereignis, z.B. Absturz)
 - Ok
- ▶ JUnit-4 versteckt mehr Funktionalität mit Metadaten (@Annotationen) und ist wesentlich komplexer. Empfehlung: Lernen Sie zuerst 3.8.1!
- ▶ Junit-5 ist zu komplex, erst für den Fortgeschrittenen geeignet!

Kern von JUnit 3.8.1



Laufzeit-Snapshot von TestSuite

- JUnit baut zur Laufzeit eine hierarchisch geschachtelte Suite von Testfällen auf



Exkurs: Erkunde JUnit 3.8.x mit Javadoc

- ▶ Aufgabe:
 - laden Sie die API-Dokumentation von JUnit mit einem Brauser Ihrer Wahl
 - finden Sie die Aufgabe der Klassen TestResult, TestCase und TestSuite heraus
 - Welche Aufgabe hat die Klasse Assert?

</home/ua1/Courses/ST1/Material/junit3.8.1/javadoc/index.html>

Gesetz 68 (PP):

Bauen Sie die Dokumentation ein, anstatt sie dranzuschrauben

Testfall der Datumsklasse in JUnit 3.8.x

- ▶ TestCases sind Methoden, beginnend mit der Markierung `test`
- ▶ Initialisierung der Halterung mit `setUp`, Abbau mit `tearDown`
- ▶ Testfallklassen sind also "Kundenklassen" von zu testenden Klassen
- ▶ Test mit `assertTrue`, geerbt von `TestCase`

```
public class DateTestCase extends TestCase {
    Date d1;
    Date d2;
    Date d3;
    Halterung (fixture)
protected void setUp() {
    d1 = new Date("1. Januar 2006");
    d2 = new Date("01/01/2006");
    d3 = new Date("January 1st, 2006");
}
public void testDate1() {
    // Processing
    d1.parseDate(); d2.parseDate(); d3.parseDate();
    // Checking
    assertTrue(d1.equals(d2)); assertTrue(d2.equals(d3));
    assertTrue(d3.equals(d1));
    .... more to say here ....
}
public void testDate2() { .. more to say here .... }
protected void tearDown() { .. .. }
}
```

Benutzung von TestCase

- ▶ Von einem Java-Programm startet man die Testfälle wie folgt:
 - Ein Testfall wird nun erzeugt durch einen Konstruktor der Testfallklasse
 - Der Konstruktor sucht die Methode des gegebenen Namens ("testDate1") und bereitet sie zum Start vor
 - mit *Reflektion*, d.h. Suche nach dem Methode in dem Klassenprototyp
 - Die *run()* Methode startet den Testfall gegen die Halterung und gibt ein *TestResult* zurück

```
public class TestApplication {  
    ...  
    TestCase tc = new DateTestCase(„testDate1“);  
    TestResult tr = tc.run();  
}
```

Testsuiten

- ▶ Eine Testsuite ist eine Kollektion von Testfällen
- ▶ TestSuites sind komposit

```
public class TestApplication {  
    ...  
    TestCase tc = new DateTestCase(„testDate1“);  
    TestCase tc2 = new DateTestCase(„testDate2“);  
    TestSuite suite = new TestSuite();  
    suite.addTest(tc);  
    suite.addTest(tc2);  
    TestResult tr = suite.run();  
    // Nested test suites  
    TestSuite subsuite = new TestSuite();  
    ... fill subsuite ...  
    suite.addTest(subsuite);  
    TestResult tr = suite.run();  
}
```

Anforderungen, Pflichtenheft und Testsuiten

- ▶ Im Idealfall führt jede Anforderung des Pflichtenhefts zu einem oder mehreren Testfällen, die in Testsuiten gesammelt werden
 - Und automatisiert überprüft werden können (nächtlicher Test).

*Testsuiten werden im **Akzeptanztest** dem Kunden vorgeführt, um zu zeigen, dass alle seine Anforderungen erfüllt worden sind.*

junit 3.8.1 TestRunner GUI

- ▶ Die Klassen `junit.awtui.TestRunner`, `junit.swingui.TestRunner` bilden einfach GUIs, die Testresultate anzeigen
 - `$ junit.awtui.TestRunner DateTestCase¶`
- ▶ Von Java-Anwendung aus:
 - Gibt man einem Konstruktor eines Testfalls eine Klasse mit, findet er die “test*“-Methoden (die Testfallmethoden) selbständig
 - Dies geschieht mittels *Reflektion*, d.h. Absuchen der Methodentabellen im Klassenprototypen und Methodenspeicher

```
public class TestApplication {  
    public static Test doSuite() {  
        // Abbreviation to create all TestCase objects  
        // in a suite  
        TestSuite suite = new TestSuite(DateTestCase.getClass());  
    }  
    // Starte the GUI with the doSuite suite  
    public static main () {  
        junit.awtui.TestRunner.run(doSuite());  
    }  
}
```

13.5.2) Testläufe in Junit 4.X

- ▶ <https://www.admfactory.com/junit-3-vs-junit-4-comparison/>

</Users/uweassmann/Courses/ST1/Material/junit4.4/javadoc/index.html>
<http://docs.oracle.com/javase/tutorial/java/annotations/index.html>



Neuer Testfall in Junit 4.X mit Metadaten-Annotationen

- ▶ TestCase-Methoden werden mit **Metadaten-Annotationen** gekennzeichnet, Annotationen an Attribute und Methoden, die mit @ beginnen
- ▶ Vorteil: Testmethoden und ihre Testees können gemeinsam auftreten

```
import org.junit.*;           // Import of the annotations
public class DateTestCase /* no superclass is necessary */ {
    Date d1;
    Date d2;
    Date d3;
    Halterung (fixture)
    @Before protected int setUp() {
        d1 = new Date („1. Januar 2006“);
        d2 = new Date („01/01/2006“);
        d3 = new Date („January 1st, 2006“);
    }
    @Test public int compareDate1() {
        // Processing
        d1.parseDate(); d2.parseDate(); d3.parseDate();
        // Checking
        assertTrue(d1.equals(d2)); assertTrue(d2.equals(d3));
        assertTrue(d3.equals(d1));
        .... more to say here ....
    }
    @Test public int compareDate2() {
        .... more to say here ....
    }
}
```

Benutzung von Testfall-Klasse in 4.x

- ▶ Von der Kommandozeile:
 - `$ java org.junit.runner.JUnitCore DateTestCase¶`
- ▶ Von Eclipse aus: In einer IDE wie Eclipse werden die Testfall-Prozeduren automatisch inspiziert und gestartet
- ▶ Von einem Java-Programm aus:
 - Ein Testfall wird erzeugt durch einen Konstruktor der Testfallklasse
 - Suche den Klassenprototyp der Testfallklasse
 - Die `run()` Methode von `JUnitCore` startet alle enthaltenen Testfälle über den Klassenprototypen
 - Starten aller annotierten Initialisierungen, Testfallmethoden, Abräumer
 - und gibt ein "Result"-Objekt zurück

```
public class TestApplication {  
    ...  
    DateTestCase tc = new DateTestCase();  
    // getClass() holt den Klassenprototypen  
    Result tr = JUnitCore.run(tc.getClass());  
}
```


Junit 4.X mit vielen weiteren Metadaten-Annotationen

- ▶ Viele weitere Test-Annotationstypen sind definiert

```
public class DateTestCase {
    Date d1;
    @BeforeClass protected int setUpAll() {
        // done before ALL tests in a class
    }
    @AfterClass protected int tearDownAll() {
        // done before ALL tests in a class
    }
    @Test(timeout=100, expected=IndexOutOfBoundsException.class)
    public int compareDate2() {
        // test fails if takes longer than 50 msec
        // test fails if IndexOutOfBoundsException is NOT thrown
        .... more to say here ....
    }
}
```

Für Interessierte: Was die Annotationen tun

- ▶ Junit 4 ist ein *Codegenerator-Framework* (*Metaprogramm-Framework*, siehe Vorlesung CBSE im SoSe), das sich im Java-Comiler registriert.
- ▶ Liest der Compiler die Annotationen, ruft er eine Prozedur aus dem junit-Codegenerator auf und übergibt die zu übersetzende Methode als Parameter:
`@test` → `call Junit4.evaluateTestAnnotation(Method orig);`
- ▶ Wobei der Codegenerator derart definiert ist:

```
// Reflective, metaprogramming code
public class Junit4 {
    Method evaluateTestAnnotation(Method orig) {
        Method testOrig = orig.clone();
        // somehow modify „testOrig“ to be a new
        // Test-Case method and put it into a TestCase
        // class
        return testOrig;
    }
}
```

13.6. Entwurfsmuster in JUnit



Was ist ein Entwurfsmuster?

Def.: Ein **Entwurfsmuster** beschreibt eine Standardlösung für

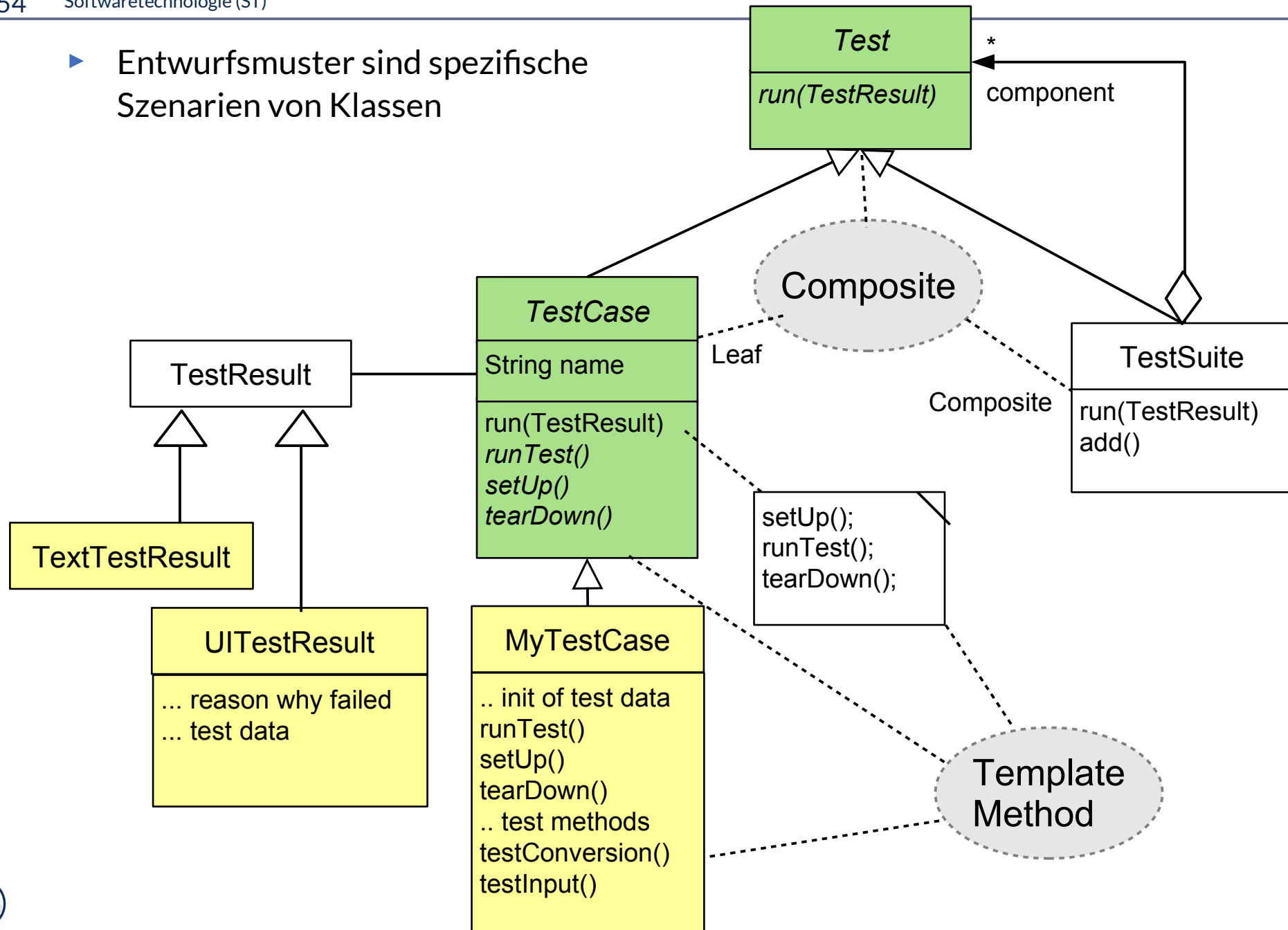
- ein Standardentwurfsproblem
- in einem gewissen Kontext



- ▶ Ein Entwurfsmuster wiederverwendet bewährte Entwurfsinformation
 - Ein Entwurfsmuster darf nicht *neu*, sondern muss wohlbewährt sein
- ▶ Ein Entwurfsmuster enthält mindestens:
 - Klassendiagramm der beteiligten Klassen
 - Objektdiagramm der beteiligten Objekte
 - Interaktionsdiagramm (Sequenzdiagramm, Kommunikationsdiagramm)
- ▶ Entwurfsmuster sind ein wesentliches Entwurfshilfsmittel aller Ingenieure
 - Maschinenbau – Elektrotechnik - Architektur
- ▶ Entwurfsmuster treten auch in Frameworks wie JUnit auf

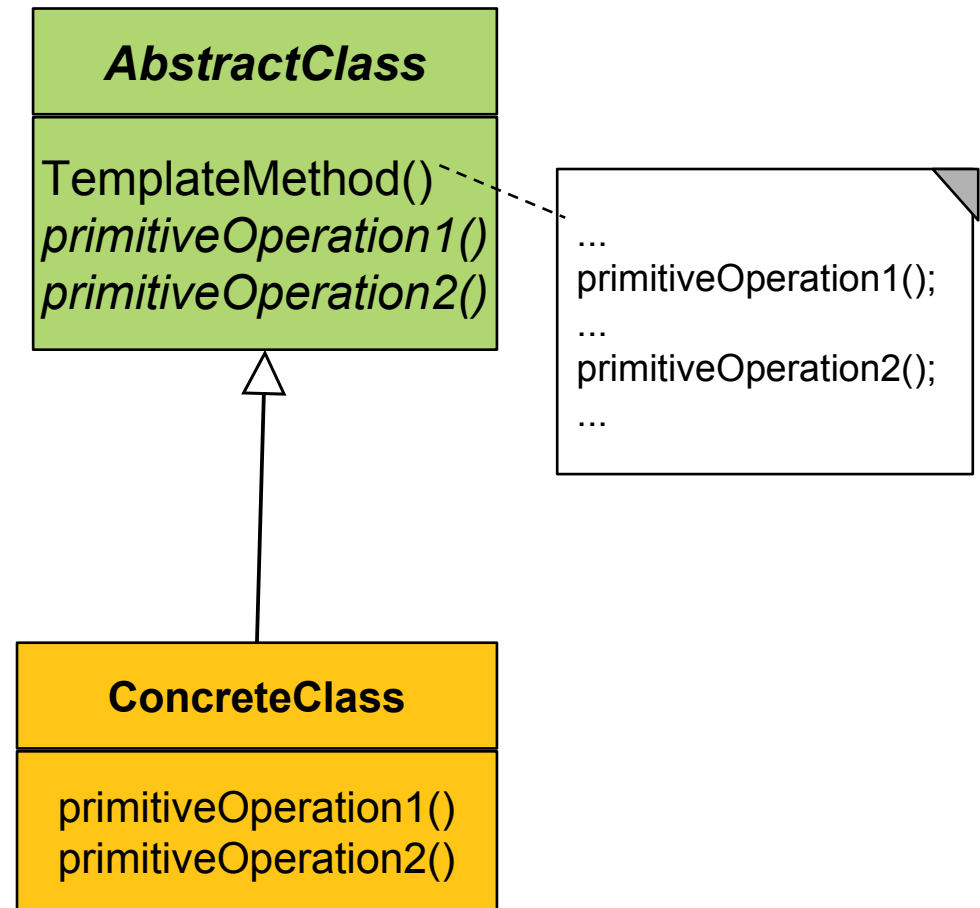
Beispiel: Entwurfsmuster in Junit 3.x

- Entwurfsmuster sind spezifische Szenarien von Klassen



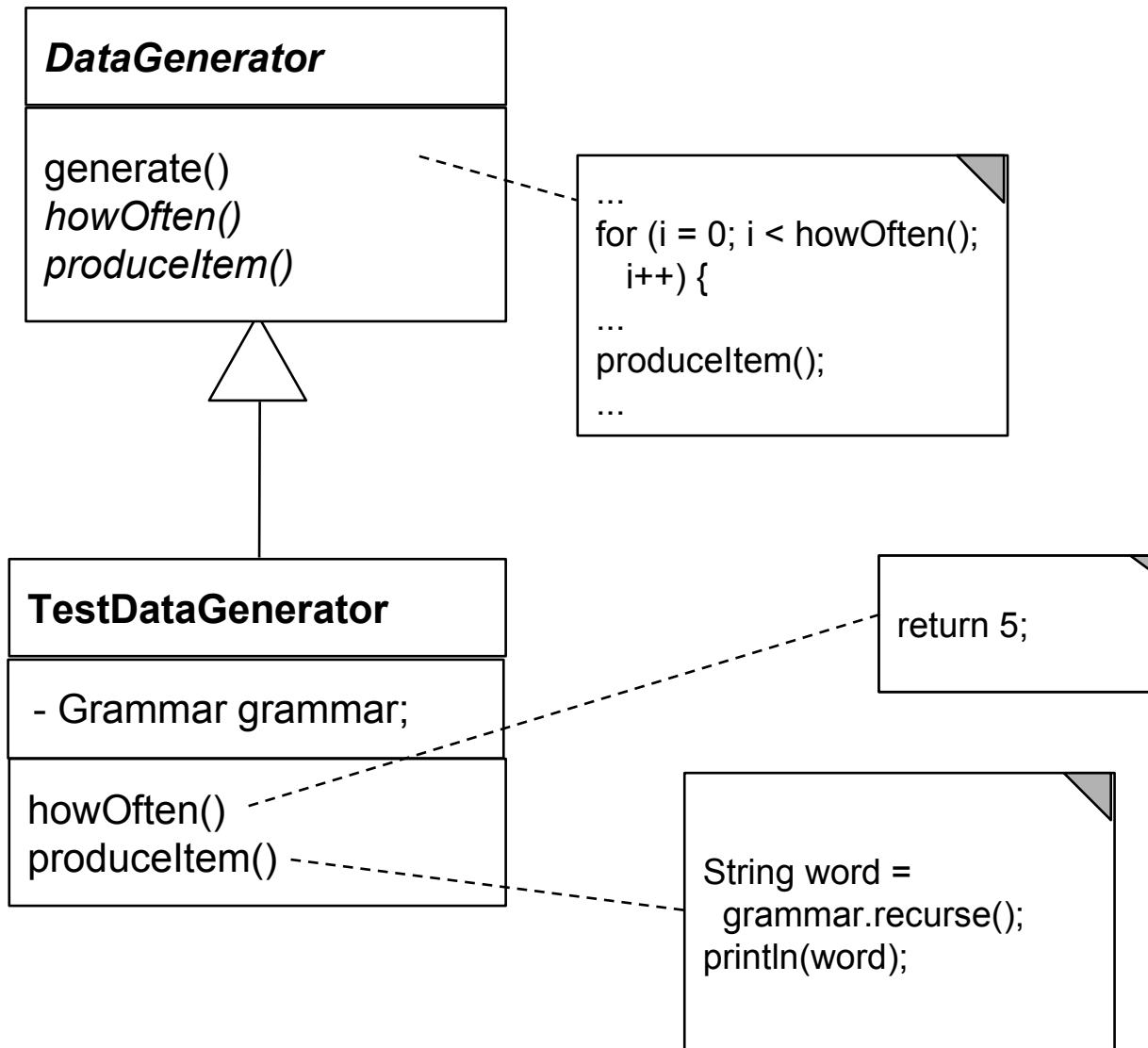
Entwurfsmuster TemplateMethod

- ▶ Definiert das Skelett eines Algorithmusses in einer *Schablonenmethode (template method)*
 - Die Schablonenmethode ist konkret
- ▶ Delegiere Teile zu abstrakten *Hakenmethoden (hook methods)*
 - die von Unterklassen konkretisiert werden müssen
- ▶ Variiere Verhalten der abstrakten Klasse durch verschiedene Unterklassen
 - Separation des “fixen” vom “variablen” Teil eines Algorithmus



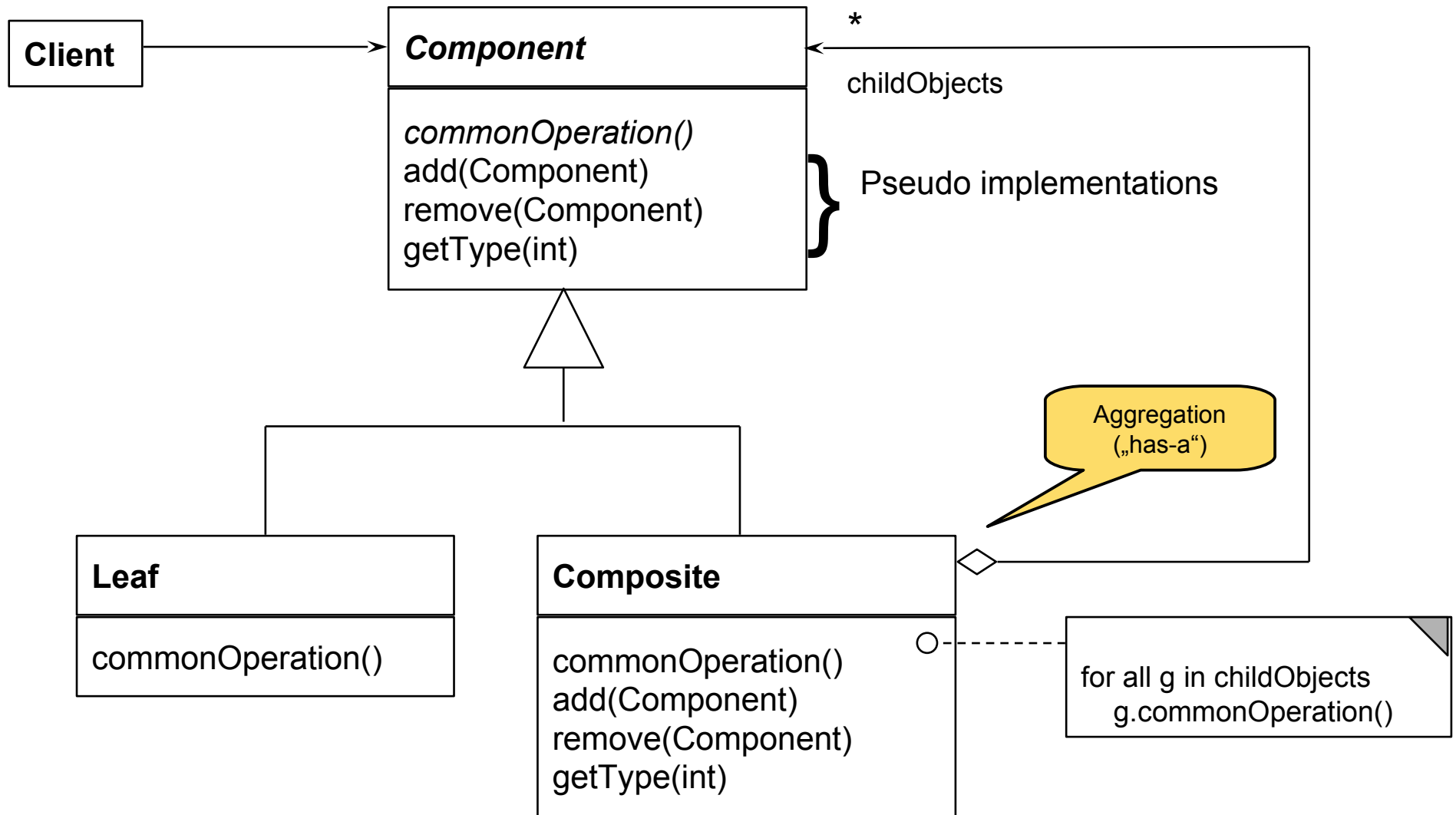
Beispiel TemplateMethod: Ein Datengenerator

- ▶ Parameterisierung eines Generators mit Anzahl und Produktion
 - (Vergleiche mit TestCase aus JUnit)



Entwurfsmuster Composite

- ▶ Composite besitzt eine rekursive n-Aggregation zur Oberklasse

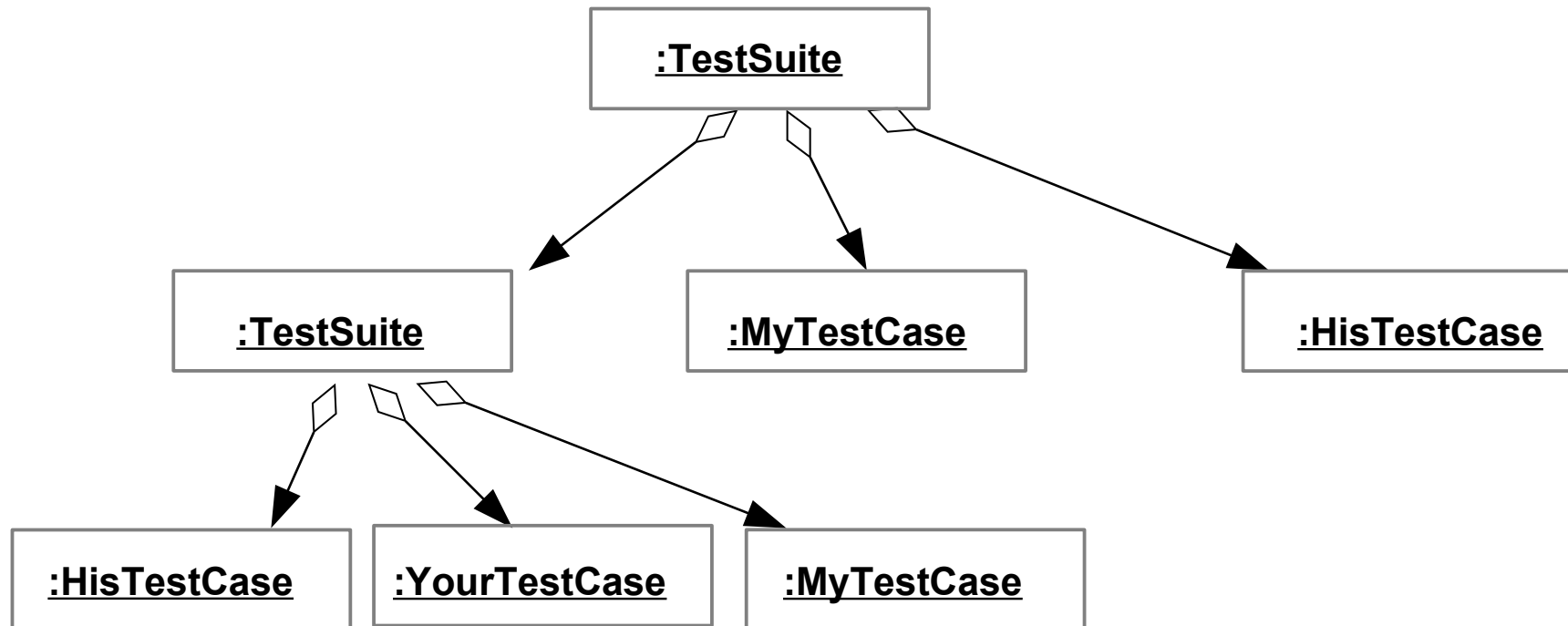


Composite in Junit 3.x

- ▶ Mehrere Methoden von *Test* sind komposit strukturiert
 - `run()`
 - `countTestCases()`
 - `tests()`
 - `toString()`

Laufzeit-Snapshot von Composite

- ▶ Composite beschreibt Ganz/Teile-Hierarchien von Laufzeit-Objekten, z.B. geschachtelte Testsuiten und -fälle



Bsp.: Zählen von Testfällen in JUnit

```
abstract class Test {
    abstract int countTestCases();
}
class TestSuite extends Test {
    Test [20] children; // here is the n-recursion
    int countTestCases() { // common operation
        for (i = 0; i <= children.length; i++) {
            curNr += children[i].countTestCases();
        }
        return curNr;
    }
    void add(Test c) {
        children[children.length++] = c;
    }
}
```

```
class TestCase extends Test {
    private int myTestCaseCount = 10;
    int countTestCases() { // common operation
        return myTestCaseCount;
    }
    void add(Test c) {
        /// impossible, dont do anything
    }
}

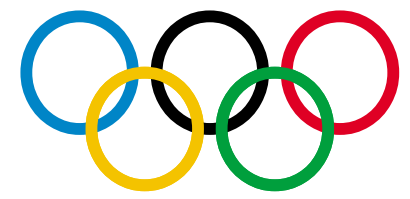
// application
main () { int nr = test.countTestCases(); }
```

Funktionales Programmieren:

- Iteratoralgorithmen (map)
- Faltungsalgorithmen (folding)

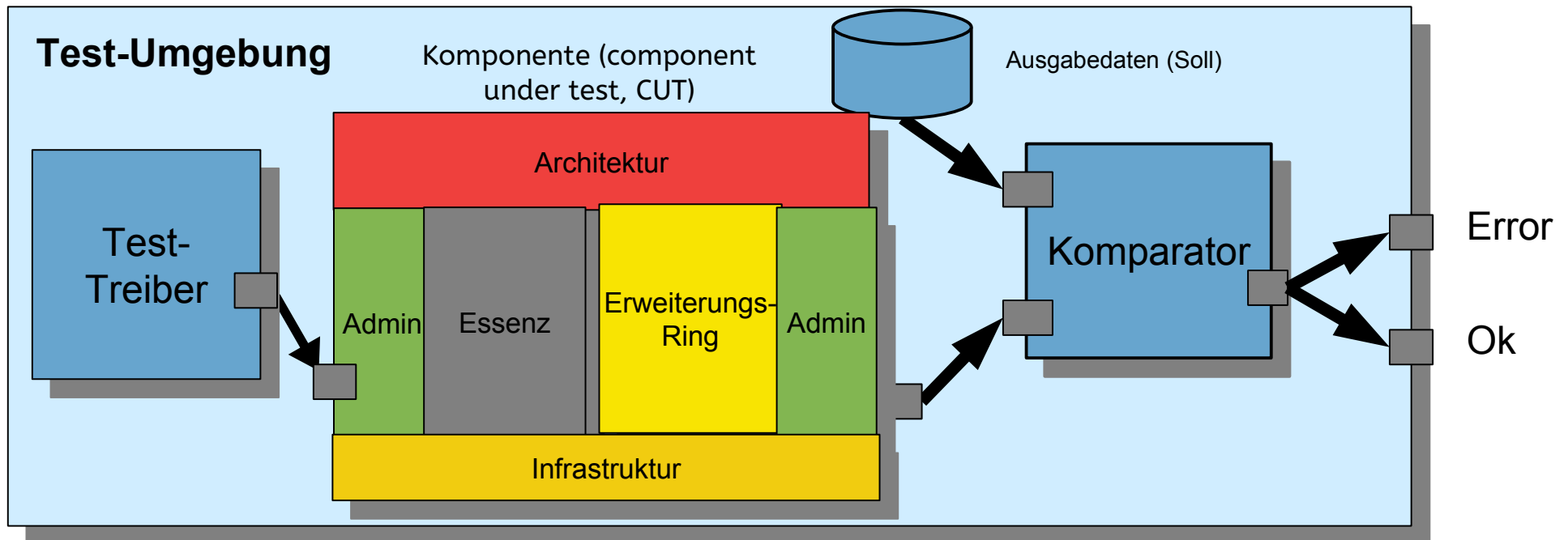
- ▶ Erstellung einer Akzeptanztestbeschreibung im Vertrag (Pflichtenheft)
 - Ohne Erfüllung kein Bestehen des Praktikums!
 - Eine Iteration: Kunde stellt einen Zusatzwunsch: Wie reagiert man auf die Veränderung?
- ▶ **Tip:** Erstellen Sie sich von Anfang an einen Regressionstest!
 - Und lassen Sie diesen bei jeder Veränderung laufen, um zu überprüfen, ob Sie wesentliche Eigenschaften des Systems verändert haben

Biologische Software: Erweiterbare Komponenten (Frameworks)



Def.: Software-Frameworks (-Rahmenwerke) sind *erweiterbare* Komponenten, d.h. erweiterbare und wiederverwendbare Programmeinheiten. Sie sind Ergebnis „biologischer“ Softwareentwicklung. Im einfachsten Fall sind sie Klassenpakete mit „olympischen Ringen“:

- klaren Schnittstellen, Administrationsring und Infrastruktur,
- gut abdeckender Testtreiber-Hülle und
- **Erweiterungsring**, an den Erweiterungen eingebaut werden können.



Was haben wir gelernt?

- ▶ **Software ohne Tests ist keine Software**
- ▶ **Programme ohne Administration sind nicht nutzbar**
- ▶ **Ringe** sind querschneidende Schichten des Programms, die es für die Zukunft vorbereiten
- ▶ Achten Sie auf das Management Ihres Projekts im Praktikum
 - Planen Sie hinreichend
 - Testen Sie sorgfältig und von Anfang an (*test-driven development, TDD*)
 - Entwerfen Sie eine Testarchitektur, Akzeptanztestsuite, Regressionstest
- ▶ Erste Entwurfsmuster TemplateMethod, Composite
- ▶ Lernen Sie, Java zu programmieren:
 - Ohne ausreichende Java-Kenntnisse weder Bestehen der Klausur noch des Praktikums
 - Nutzen Sie fleissig das Java-INLOOP-System!



SUMMARY

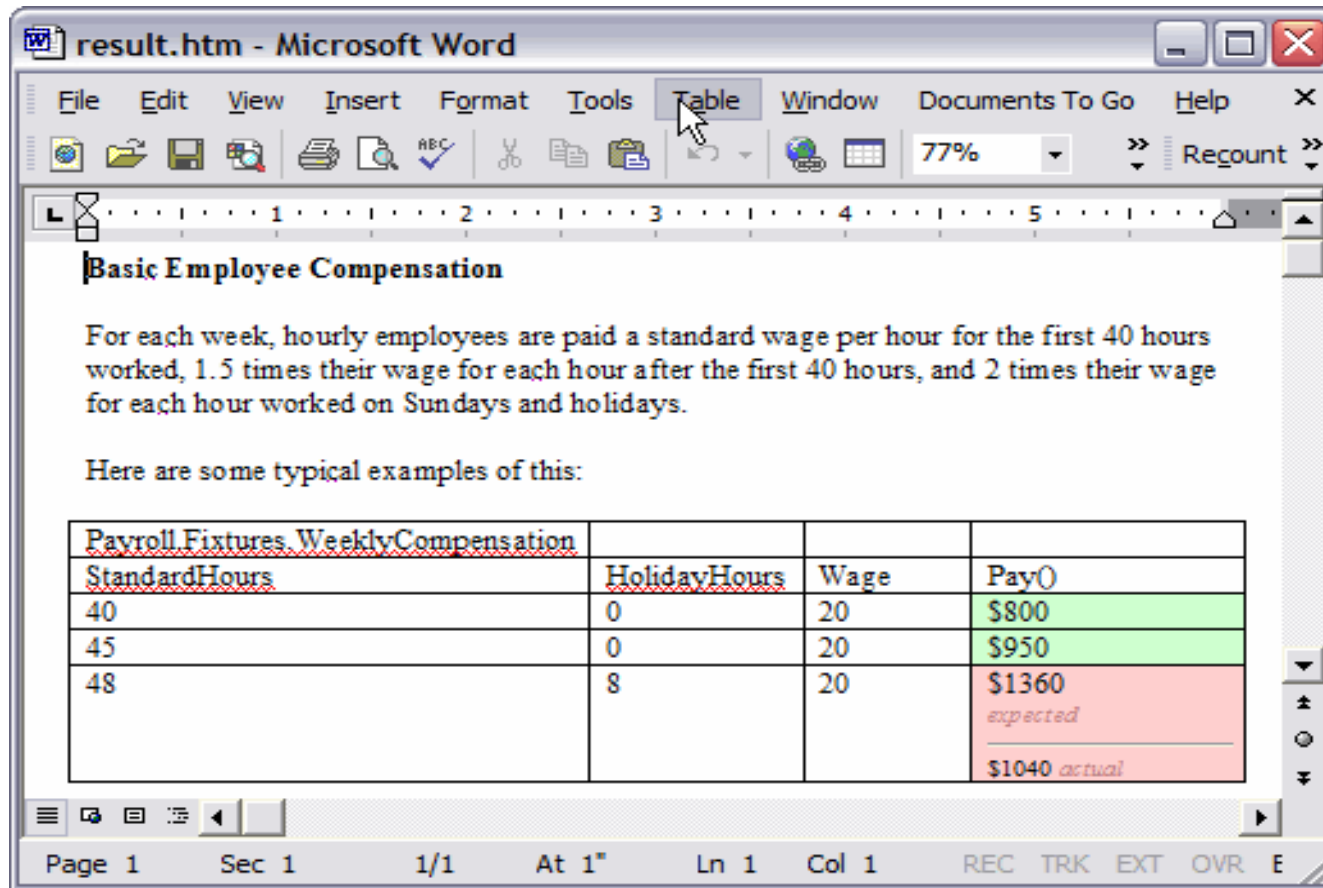
Verständnisfragen

- ▶ Erklären Sie den “Ring der Administration”.
- ▶ Welche Teil-Ringe besitzt der Ring der Administration?
- ▶ Wieso ist der Test-Ring für Wiederverwendung so wichtig?
- ▶ Was ist der Unterschied zwischen einer Klasse und einer Komponente?
- ▶ Was unterscheidet den Ring der funktionalen Essenz von den anderen Ringen?

Anhang

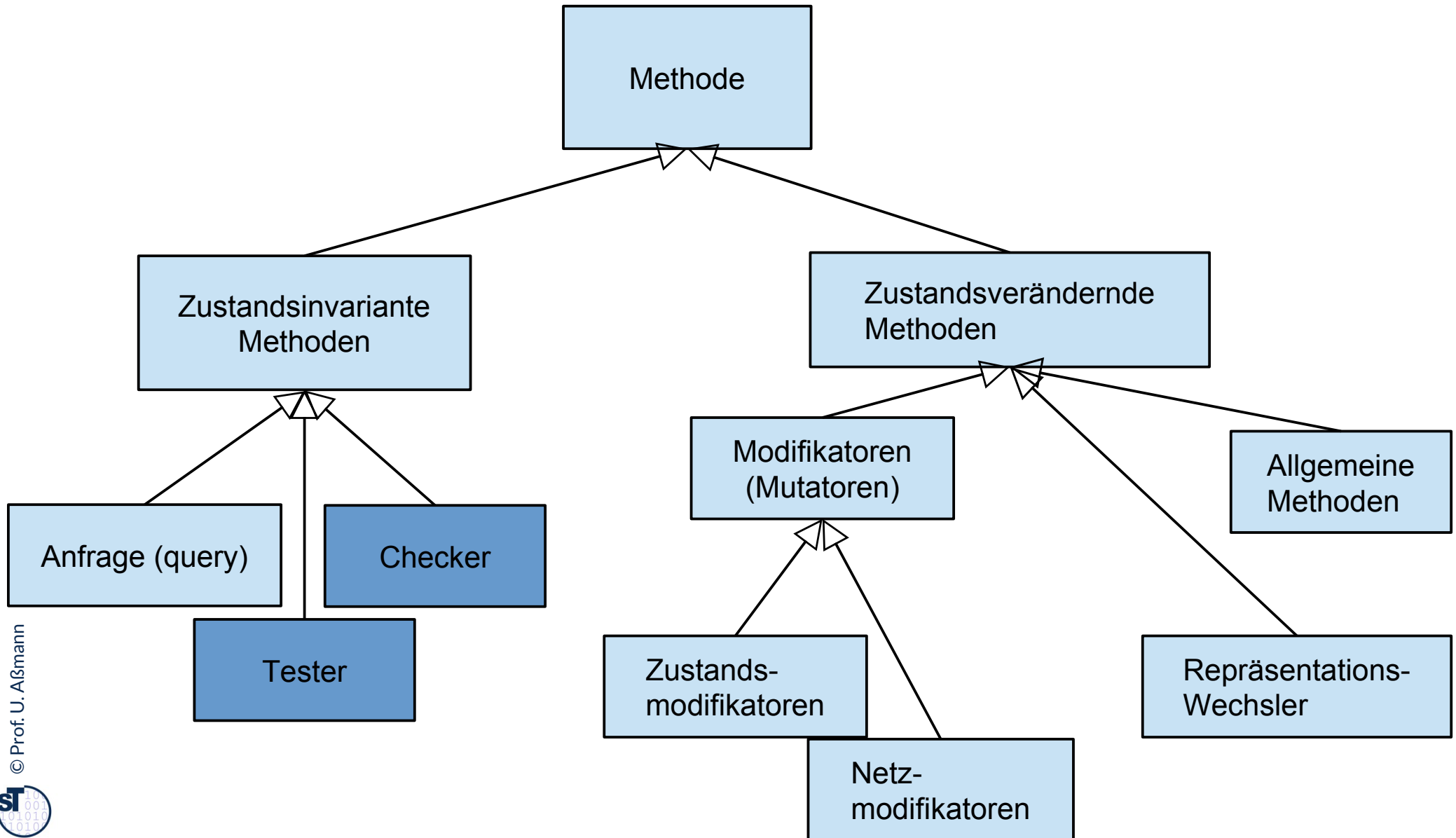
FIT Testfalltabellen Framework <http://fit.c2.com>

- ▶ FIT bietet eine Spezifikation der Testfälle in Word oder Excel
 - Automatische Generierung von Junit-Testfällen
 - Automatischer Feedback
- ▶ siehe Softwaretechnologie-II, WS



Erweiterung: Begriffshierarchie der Methodenarten

- ▶ **Wiederholung:** Welche Arten von Methoden gibt es in einer Klasse?



Wie wähle ich Testdaten für Testfälle aus?

- ▶ Bestimme die **Extremwerte** der Parameter der zu testenden Methode
 - Nullwerte immer testen, z.B. 0 oder null
 - Randwerte, z.B. 1.1., 31.12
- ▶ Bestimme **Bereichseinschränkungen**
 - Werte ausserhalb eines Zahlenbereichs
 - negative Werte, wenn natürliche Zahlen im Spiel sind
- ▶ Bestimme **Zustände**, in denen sich ein Objekt nach einer Anweisung befinden muss
- ▶ Bestimme **Äquivalenzklassen** von Testdaten und teste nur die Repräsentanten
- ▶ Bestimme alle Werte aller **boolschen Bedingungen** in der Methode
 - Raum aller Steuerflußbedingungen

Even Worms are Tested

- ▶ StuxNet Tests in Israel
 - <http://catless.ncl.ac.uk/Risks/26.31.html#subj3.1>
- ▶ LAUSD payroll fiasco
 - <http://catless.ncl.ac.uk/Risks/24.84.html>
- ▶ Surprising reimplementations of systems with good new tests:
 - <http://catless.ncl.ac.uk/Risks/24.85.html#subj6.1>

- ▶ Hamburg-Altona Bahnhof 1995
 - <http://catless.ncl.ac.uk/Risks/16.93.html#subj1.1>
 - <http://catless.ncl.ac.uk/Risks/16.94.html#subj1.1>
 - <http://catless.ncl.ac.uk/Risks/17.02.html#subj3.1>
- ▶ Toll Collect Krise 2004
 - <http://catless.ncl.ac.uk/Risks/23.21.html#subj6.1>
- ▶ Velaro-D-Züge von Siemens
 - <http://www.sueddeutsche.de/wirtschaft/verspaetete-lieferung-von-ice-zuegen-eine-halbe-milliarde-euro-auf-dem-abstellgleis-1.1655927>
 - http://www.nwzonline.de/wirtschaft/bericht-neue-siemens-ice-der-bahn-erhalten-zulassung_a_11,5,196943309.html

Edison, der Erfinder der Glühbirne

"If I find 10,000 ways something won't work, I haven't failed. I am not discouraged, because every wrong attempt discarded is another step forward."

Thomas A. Edison

"Müsste Edison eine Nadel im Heuhaufen finden, würde er einer fleißigen Biene gleich Strohalm um Strohalm untersuchen, bis er das Gesuchte gefunden hat."

- Nikola Tesla, New York Times, 19. Oktober 1931

Aber: Ein Wort der Warnung

[Edison] had no hobby, cared for no sort of amusement of any kind and lived in utter disregard of the most elementary rules of hygiene. [...] His method was inefficient in the extreme, for an immense ground had to be covered to get anything at all unless blind chance intervened and, at first, I was almost a sorry witness of his doings, **knowing that just a little theory and calculation would have saved him 90% of the labour.**

But he had a **veritable contempt for book learning and mathematical knowledge**, trusting himself entirely to his inventor's instinct and practical American sense.

Nikola Tesla

Definition neuer Ausnahmen

Benutzung von benutzerdefinierten Ausnahmen möglich und empfehlenswert !

```
class TestException extends Exception {
    public TestException () {
        super();
    }
}
class SpecialAdd {
    public static int sAdd (int x, int y)
        throws TestException {
        if (y == 0)
            throw new TestException ();
        else
            return x + y;
        }
    }
}
```


Deklaration und Propagation von Ausnahmen

- ▶ Wer eine Methode aufruft, die eine Ausnahme auslösen kann, muß
 - entweder die Ausnahme abfangen
 - oder die Ausnahme weitergeben (*propagieren*)
- ▶ Propagation in Java: Deklarationspflicht mittels **throws** (außer bei Error und RuntimeException)

```
public static void main (String[] argv) {  
    System.out.println (SpecialAdd.sAdd (3, 0)) ;  
}
```

Java-Compiler: Exception TestException must be caught, or it must be declared in the throws clause of this method.

Bruch von Verträgen und Ausnahmen

- ▶ Man kann Verträge auch mit Ausnahmetests prüfen
- ▶ Vorteil: kontrollierte Reaktion auf Vertragsbrüche

```
class ContractViolation extends Exception {..};  
class ParameterContractViolation extends ContractViolation  
{..};  
class FigureEditor{  
    draw (Figure figure) throws ContractViolation {  
        if (figure == null)  
            throw new ParameterContractViolation();  
    }  
}
```

```
▶ im Aufrufer:  
try {  
    editor.draw(fig);  
} catch (ParameterContractViolation) {  
    fig = new Figure();  
    editor.draw(fig);  
}
```