

### **Arbeitsgruppe Software Engineering Prof. Elke Pulvermüller**

Universität Osnabrück  
Institut für Informatik, Fachbereich Mathematik / Informatik  
Raum 31/318, Albrechtstr. 28, D-49069 Osnabrück

[elke.pulvermueller@informatik.uni-osnabrueck.de](mailto:elke.pulvermueller@informatik.uni-osnabrueck.de)

<http://www.inf.uos.de/se>

Sprechstunde: mittwochs 14 – 15 und n.V.



- 1 Software-Krise und Software Engineering**
- 2 Grundlagen des Software Engineering**
- 3 Projektmanagement**
- 4 Konfigurationsmanagement**
- 5 Software-Modelle**
- 6 Software-Entwicklungsphasen, -prozesse, -vorgehensmodelle**
- 7 Qualität**
- 8 ... Fortgeschrittene Techniken**

### **7.1 Einordnung und Begriff**

### **7.2 Qualitätseigenschaften**

### **7.3 Wege zur Qualität**

### **7.4 Qualität und Softwareentwicklung**

### **7.5 Unit-Test**

Literatur:

Ludewig, Lichter: Software Engineering, 2007

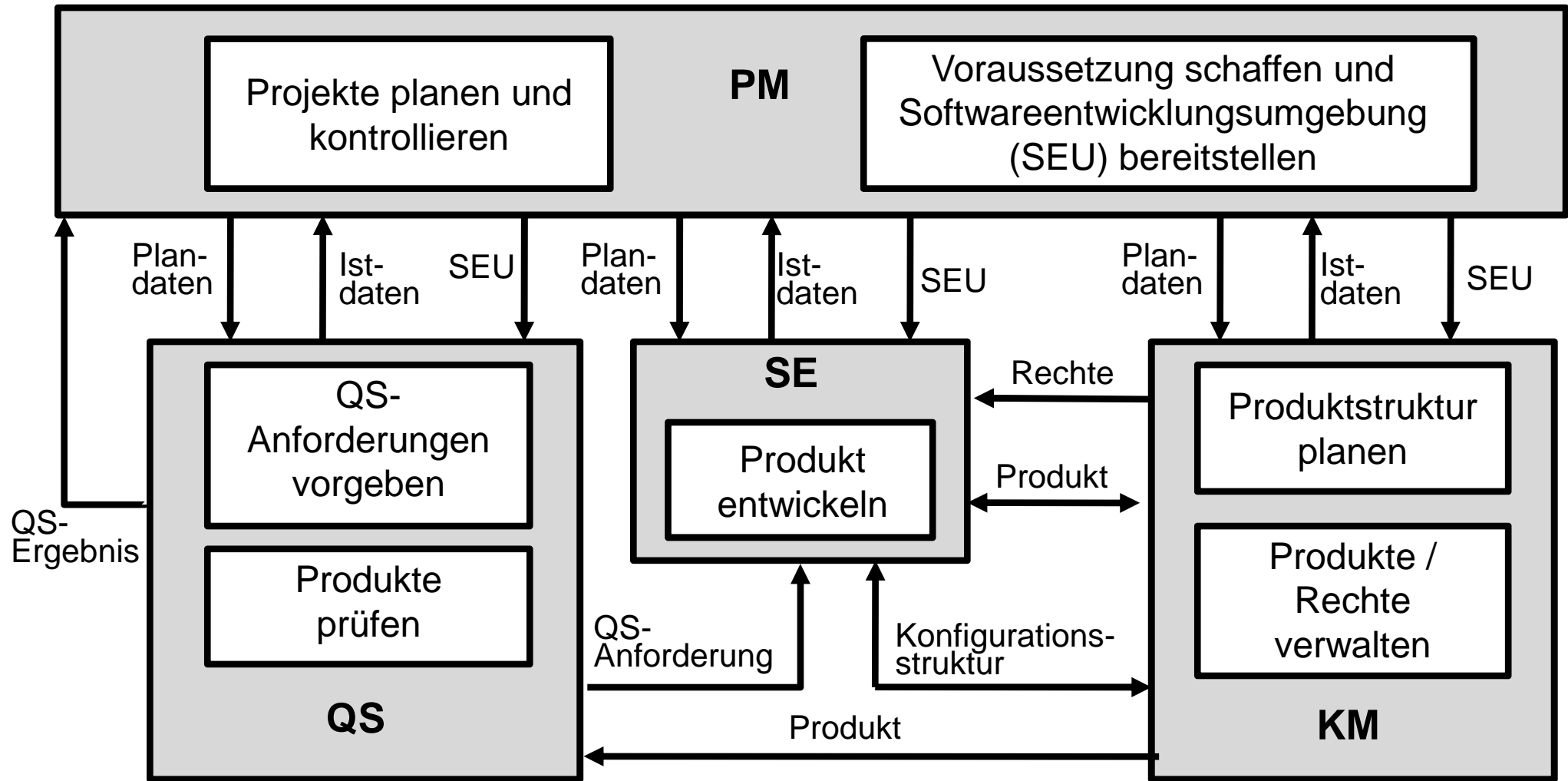
Zuser, Grechenig, Köhle: Software Engineering mit UML und dem Unified Process,  
Pearson

Peter Liggesmeyer: Software-Qualität: Testen, Analysieren und Verifizieren von Software,  
Spektrum Akademischer Verlag, 2009

Dirk Hoffmann: Software-Qualität, Reihe: eXamen.press, 2008

## 7.1 Einordnung und Begriff

### Schnittstellen des Qualitätsmanagements:



SEU: Software-Entwicklungsumgebung

QS: Qualitätssicherung

SE: Software-Entwicklung (beinhaltet auch Wartung und Evolution)

PM: Projektmanagement

KM: Software-/Konfigurationsmanagement

aus [Zuser, SW Engineering, 2004]

## 7.1 Einordnung und Begriff

### ■ Qualität [DIN-ISO-Norm 9126]

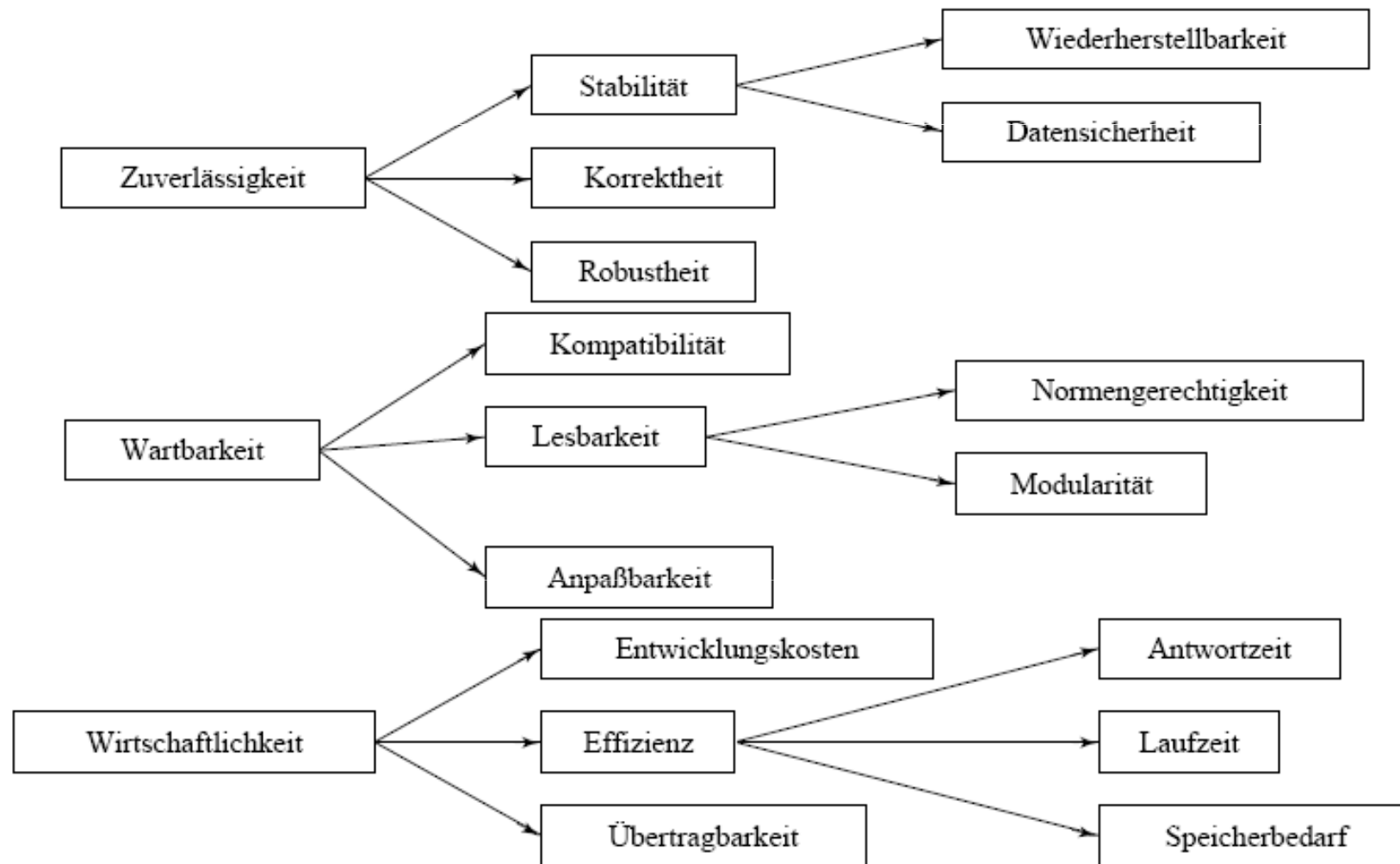
**Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen** [Hoffmann08]

### ■ Qualität [DIN 55350-11 1995]

**Beschaffenheit einer Einheit bezüglich ihrer Eignung, festgelegte und abgeleitete Erfordernisse (Qualitätsanforderungen) zu erfüllen.**

⇒ es gibt nicht das eine Kriterium, das direkt (und quantitativ) mit Software-Qualität verbunden ist

### Qualitätseigenschaften (Anforderungen an industriell verwendbare Software)



### Wege zur Qualität:

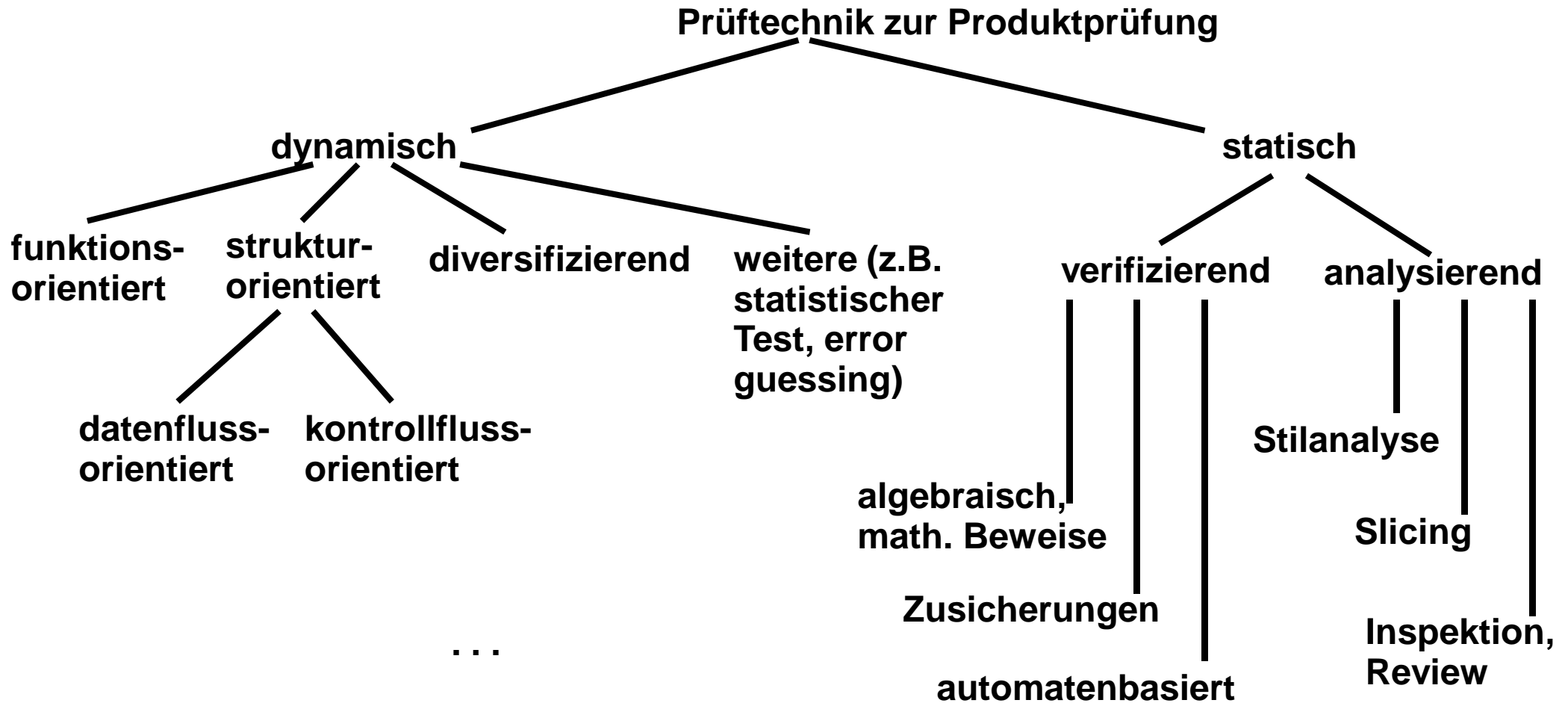
- **Qualitätssicherung (QS):**

Summe aller Maßnahmen, die die Qualität des entstehenden Produktes gewährleisten soll (konstruktiv oder analytisch).

- **Qualitätsmanagement (QM):**

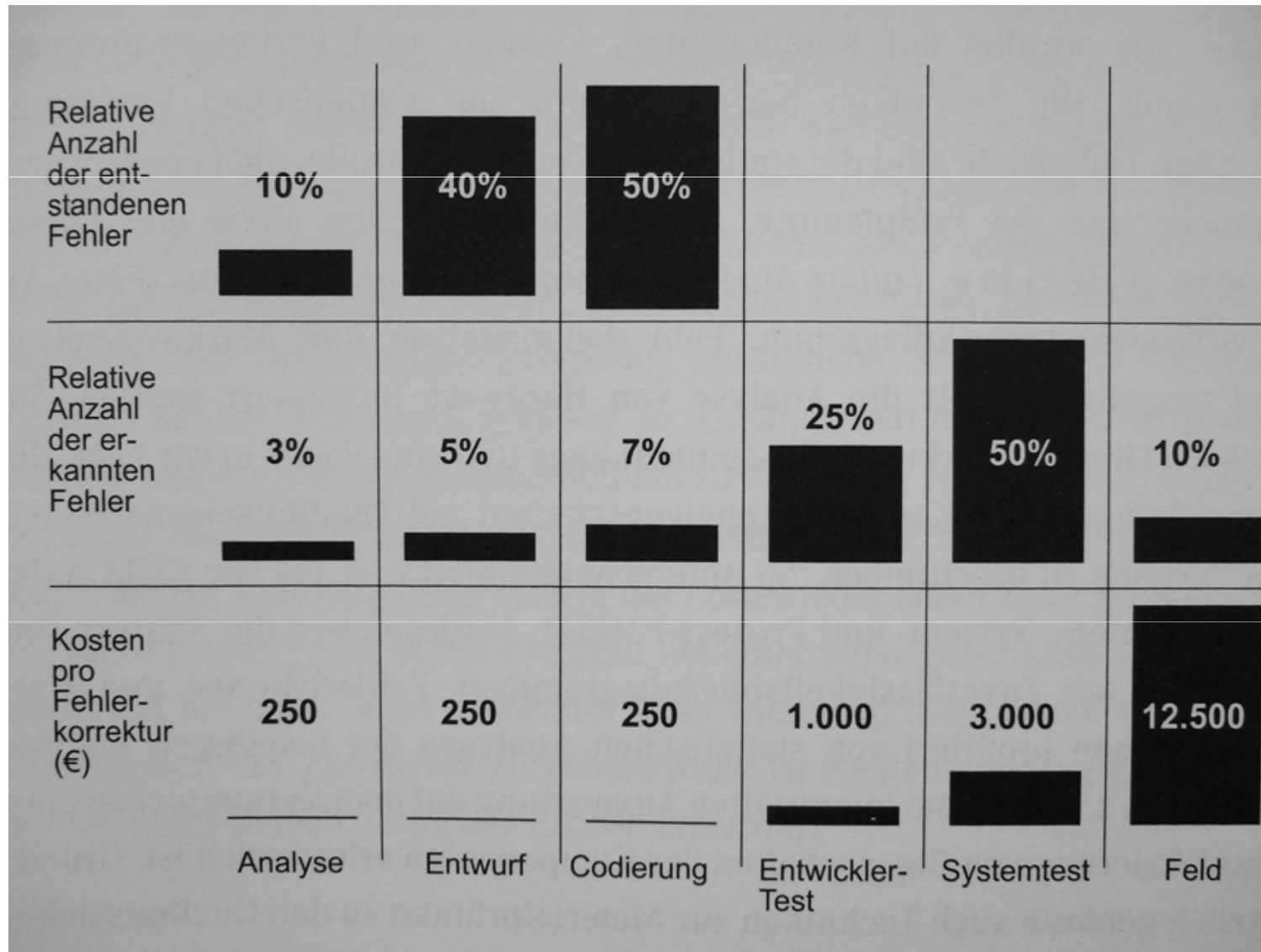
alle organisatorischen Maßnahmen zur Erreichung und für den Nachweis einer hohen bzw. einer bestimmten Qualität

### Klassifikation: Welche Arten von Maßnahmen?



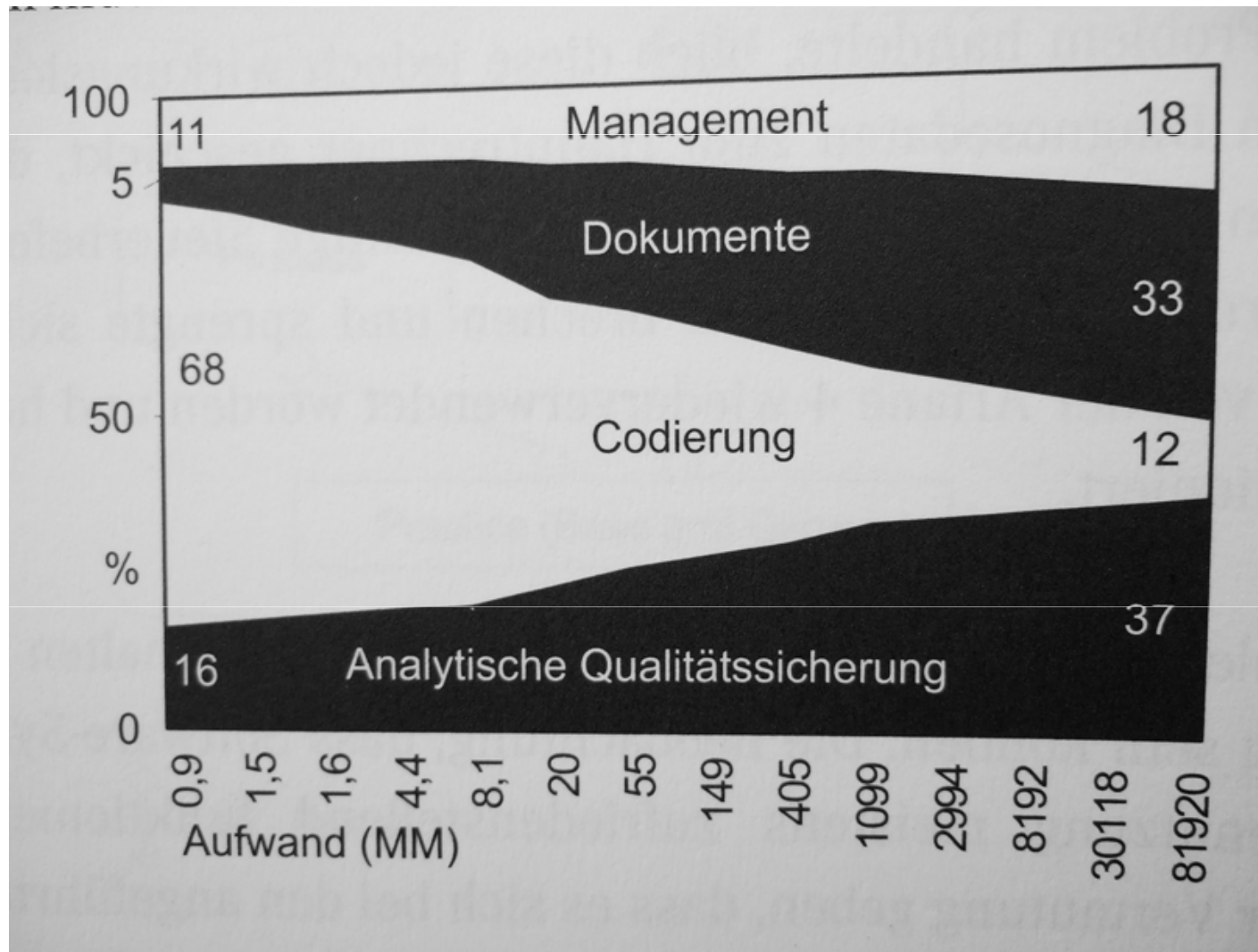


### Fehlerentstehung, Fehlererkennung, Fehlerkorrekturkosten:



[nach Möller 96]

Anteil von QS am Gesamtentwicklungsaufwand in einem Projekt:



[nach C. Jones, 91]

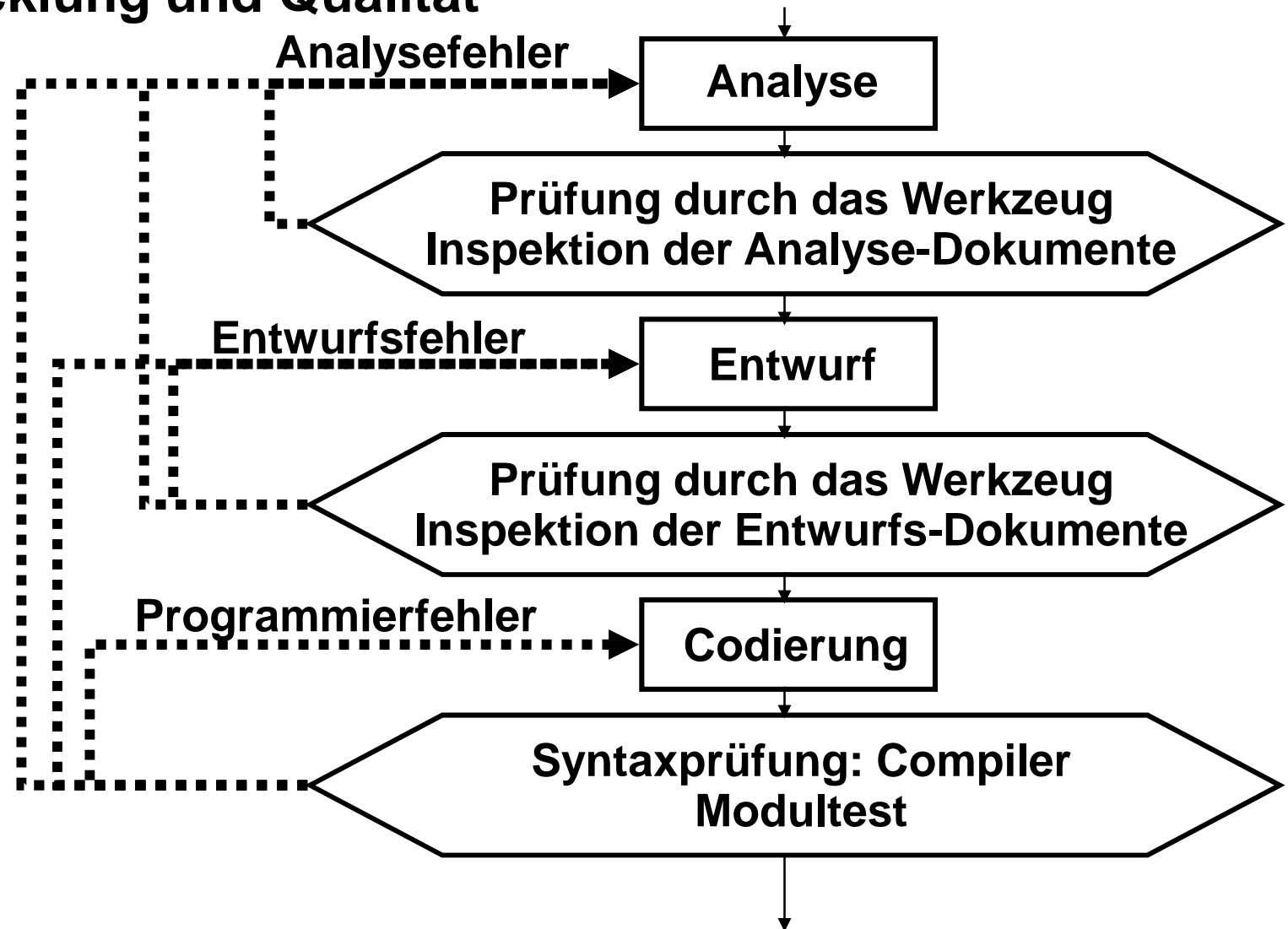
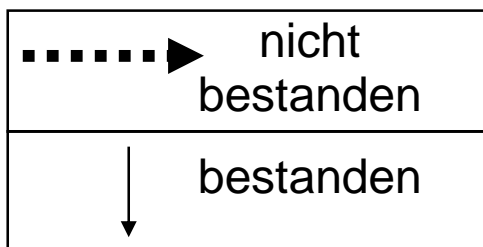
### Software-Entwicklung und Qualität

konstruktive QS



analytische QS

Legende:



- **Unit testing: Does a single object work?**
- **Integration testing: Do multiple objects work together?**
- **Functional testing: Does my application work?**
- **Performance testing: Does my application work well?**
- **Acceptance testing: Does the customer like my application?**

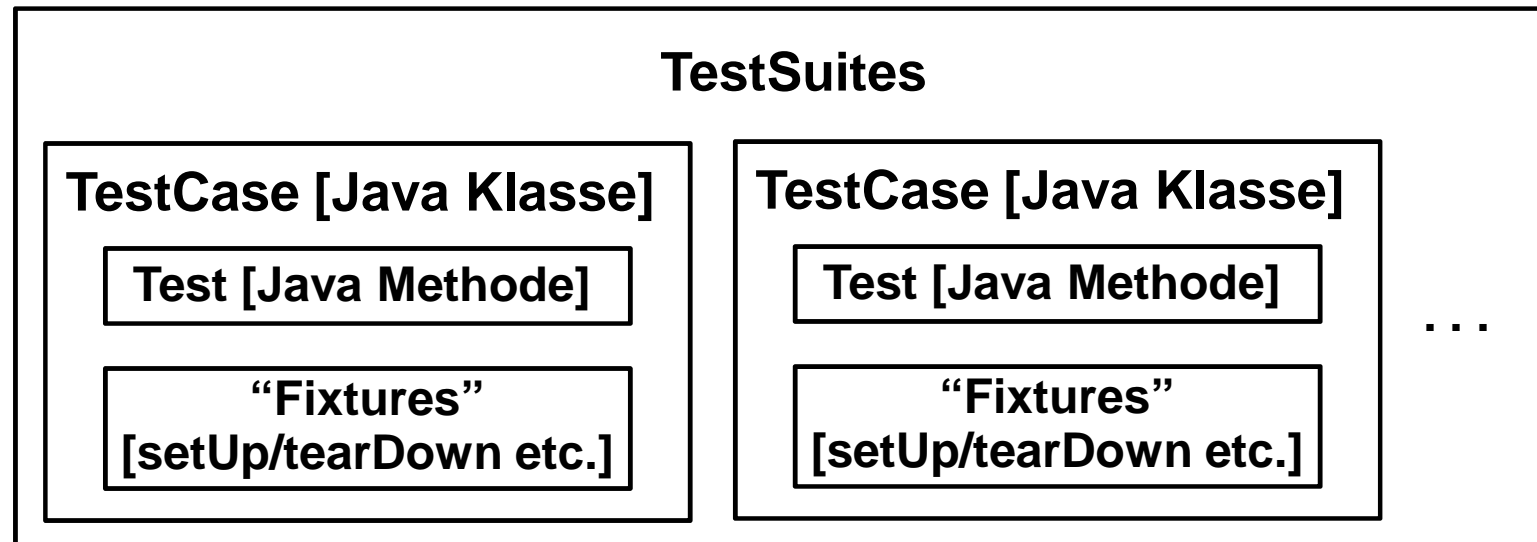
## 7.5 Unit-Test: JUnit Framework

- **Unit-Test-Umgebung zum Testen von Java Code**
- **Entwickler: Erich Gamma, Kent Beck**
- **Ziel: Einfaches Schreiben von Unit-Tests, Unterstützung der Entwickler**
- **Realisiert als Open-Source Framework, das alle Werkzeuge für das Testen bereitstellt**  
Framework: Menge von Klassen und Konventionen für ihre Nutzung
- **JUnit 3.x und JUnit 4.x**
- **xUnit: Analoge Frameworks auch für andere Sprachen (andere z.B. CppUnit, HTTPUnit)**

## 7.5 Unit-Test: JUnit Framework

- **Testfall (Test Case) =**  
**Sequenz von Operationen + Inputs + erwartete Werte**
- **Unterstützung:**
  - (1) Schreiben von Testfällen
  - (2) Ausführen von Testfällen
  - (3) Pass/Fail? (erwartetes Ergebnis = erreichtes Ergebnis?)

- **Struktur:**



## 7.5 Unit-Test: JUnit 3.x (Testklasse)

### Schritt 1: Testklasse als Subklasse zu TestCase

- TestCase steht im JUnit-Framework zur Verfügung
- Beispiel:

```
import junit.framework.TestCase;

public class StackTester extends TestCase {
    public StackTester(String name) {
        super(name);
    }
    ...
}
```

```
class Stack {
    ...
}
```

- Eigene Testklasse enthält die selbst definierten Tests

## 7.5 Unit-Test: JUnit 3.x (Testmethode)

### Schritt 2: Testmethoden

- Wichtig: Namen der Testmethoden beginnen mit „test“
- Beispiel:

```
import junit.framework.TestCase;

public class StackTester extends TestCase {
    public StackTester(String name) {
        super(name);
    }

    public void testStack() {
        Stack aStack = new Stack();
        if (!aStack.isEmpty()) {
            System.out.println("Stack should be empty!");
            aStack.push(10);
            aStack.push(4);
            System.out.println("Last element: " + aStack.pop());
            System.out.println("First element: " + aStack.pop());
        }
        ...
    }
}
```

```
class Stack {
    public boolean isEmpty { ... }
    public void push(int i) { ... }
    public int pop() { ... }
    ...
}
```



## 7.5 Unit-Test: JUnit 3.x (Testmethode)

### Ausführungsregeln für Testfälle

- JUnit führt für jede **TestCase**-Klasse alle ihre **public-test-Methoden** aus (d.h. diejenigen, die mit „test“ beginnen)
- alles übrige wird ignoriert, z.B. auch
- „Helper-Methoden“: können in Test-Klassen enthalten sein
  - a) Methoden, die nicht **public** deklariert sind
  - b) Methoden, deren **Methodenname** nicht mit „test“ beginnt

## 7.5 Unit-Test: JUnit 3.x (Test Suite)

### Schritt 3: Gruppieren von Testfällen zu einer Test Suite

- Beispiel:

```
import junit.framework.TestSuite;

public class AllTests extends TestSuite {
    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(StackTester.class);
        suite.addTestSuite(AnotherTester.class);
        return suite;
    }
}
```

```
import junit.framework.*;
public class StackTester
    extends TestCase {
    ...
}
```

```
import junit.framework.*;
public class AnotherTester
    extends TestCase {
    ...
}
```

## 7.5 Unit-Test: JUnit 3.x (assert\*)

### Gestaltung der Testmethoden: assert\*()-Methodenfamilie

- Beispiel:

```
import junit.framework.TestCase;

public class StackTester extends TestCase {
    public StackTester(String name) {
        super(name);
    }

    public void testStack() {
        Stack aStack = new Stack();
        assertTrue("stack should be empty", aStack.empty());
        aStack.push(10);
        aStack.push(4);
        ...
    }
}
```

```
class Stack {
    public boolean isEmpty { ... }
    public void push(int i) { ... }
    public int pop() { ... }
    ...
}
```

## 7.5 Unit-Test: JUnit 3.x (assert\*)

### Methoden in der assert\*()-Methodenfamilie

- Methodennamen beginnen mit „assert“
- public-Methoden, die in der Superklasse `TestCase` definiert sind
- Verwendet der Testentwickler in den Testmethoden
- Wenn die Bedingung zu `FALSE` ausgewertet wird, dann
  - a) war der Test nicht erfolgreich (test fails)
  - b) die Testausführung überspringt den Rest der Testmethode
  - c) die Nachricht (falls vorhanden) wird ausgegeben`AssertionFailedError` wird geworfen
- Wenn die Bedingung zu `TRUE` ausgewertet wird, dann wird die Auswertung normal fortgesetzt

## 7.5 Unit-Test: JUnit 3.x (assert\*)

### Methoden der assert\*()-Methodenfamilie

- `assertTrue(„message for fail“, condition);`
- `assertFalse(„message“, condition);`
- `fail(„message“);`
  
- `assertEquals(expected_value, expression);`
- `assertEquals(expected_value, actual_value, delta);` // max. delta
- `assertArrayEquals(expected_array, actual_array);`
- `assertThat(actual_value, matcher);` // Matcher-Object spezifiziert die Bedingung
  
- Für Objekt-Referenzen
  - `assertNull(reference);`
  - `assertNotNull(reference);`
  - `assertSame(reference_expected, reference_actual);`
  - `assertNotSame(reference_expected, reference_actual);`
  
- ... <http://junit.org/apidocs/org/junit3/Assert.html>
  
- Pro Methoden mit und ohne „message“

## 7.5 Unit-Test: JUnit 3.x (assert\*)

- Beispiel mit assert\*():

```
import junit.framework.TestCase;

public class StackTester extends TestCase {
    public StackTester(String name) {
        super(name);
    }

    public void testStack() {
        Stack aStack = new Stack();
        assertTrue("Stack should be empty!", aStack.empty());
        aStack.push(10);
        assertTrue("Stack should not be empty!", !aStack.empty());
        aStack.push(4);
        assertEquals(4, aStack.pop());
        assertEquals(10, aStack.pop());
    }
    ...
}
```

```
class Stack {
    public boolean isEmpty { ... }
    public void push(int i) { ... }
    public int pop() { ... }
    ...
}
```

## 7.5 Unit-Test: JUnit 3.x (assert\*)

- Beispiel mit assert\*(): Jeweils nur ein Konzept (Modularisierung!)

```
import junit.framework.TestCase;
public class StackTester extends TestCase {
    public StackTester(String name) {
        super(name);
    }

    public void testStackEmpty() {
        Stack aStack = new Stack();
        assertTrue("Stack should be empty!", aStack.empty());
        aStack.push(10);
        assertTrue("Stack should not be empty!", !aStack.empty());
    }

    public void testStackOperations() {
        Stack aStack = new Stack();
        aStack.push(4); aStack.push(4);
        assertEquals(4, aStack.pop());
        assertEquals(10, aStack.pop());
    }
}
```

```
class Stack {
    public boolean isEmpty { ... }
    public void push(int i) { ... }
    public int pop() { ... }
    ...
}
```

## 7.5 Unit-Test: JUnit 3.x (setUp/tearDown)

### Fixtures: setUp() und tearDown()

- Methoden, die jenseits der mit „test“-beginnenden Methoden automatisch aufgerufen werden
- setUp(): Methode, die vor allen Testmethoden gerufen wird  
Aufgabe: Initialisierung von Objekten
- tearDown(): Methode, die nach allen Testmethoden gerufen wird  
Aufgabe: Freigabe von Objekten



### JUnit 4 ...

- benötigt mindestens Java5
- kann auch JUnit3 Tests ausführen (abwärtskompatibel)
- unterstützt die assert\*()-Methoden wie in JUnit3
- arbeitet mit Annotationen @ (Metadaten statt Vererbung)

- Import

```
import org.junit.*
```

```
import static org.junit.Assert.*
```

(statt Import von `junit.framework.*`)

- Der Import ersetzt die Vererbung von Superklasse `TestCase`
- Spezielle Annotationen vor den Methodennamen markieren die Methoden (statt spezieller Methodennamen)

**@Test** statt Methodennamen, die mit “test” beginnen

**@Before** statt `setUp`

**@After** statt `tearDown`

## 7.5 Unit-Test: JUnit 4.x (setUp/tearDown)

- **@Before:** Falls benötigt, eine oder mehrere Methoden, die vor jedem Test ausgeführt werden (z.B. zur Initialisierung)

**@Before**

```
public void setUp() {  
    // initialization code  
}
```

- **@After:** Falls benötigt, eine oder mehrere Methoden, die nach jedem Test ausgeführt werden (z.B. zur Freigabe von Ressourcen)

**@After**

```
public void tearDown() {  
    // cleanup code  
}
```

## 7.5 Unit-Test: JUnit 4.x (Testmethode)

- Testmethode durch Annotation mit `@Test`
- Die Testmethode hat keine Parameter und liefert kein Ergebnis
- Beispiel:

```
@Test  
public void sum() {  
    ...  
}
```

## 7.5 Unit-Test: JUnit 4.x (Testmethode, Nutzung von assert\*)

- Assert\*()-Methoden können wie in JUnit 3.x verwendet werden
- Beispiel:

@Test

```
public void sum() {  
    assertEquals(15, program.sum(array));  
    assertTrue(program.min(array) > 0);  
}
```

- Gescheiterte Tests werfen **AssertionError** (statt **AssertionFailedError**)
- Kleinere Unterschiede in den assert\*-Methoden (z.B. zusätzlich **assertEquals**-Methoden für Object-Arrays)

## 7.5 Unit-Test: JUnit 4.x (Testmethode)

- **Spezifikation eines Ausführungszeitlimits in Millisekunden**  
Der Test scheitert, wenn die Methode das Zeitlimit überschreitet.  
Ziel: z.B. Vermeiden von Endlosschleifen

```
@Test (timeout=10)
public void greatBig() {
    assertTrue(program.ackermann(5,5) > 10e12);
}
```

- **Test von Exceptions**  
Der Test ist erfolgreich, wenn die erwartete Exception geworfen wird.  
Anderenfalls scheitert der Test.

```
@Test (expected=IllegalArgumentException.class)
public void factorial() {
    program.factorial(-5);
}
```

## 7.5 Unit-Test: JUnit 4.x (Test Suite)

- Aufbau einer Menge von Tests (Test Suite)

```
import org.junit.runners.Suite;  
import org.junit.runners.Suite.SuiteClasses;
```

```
@RunWith(value=Suite.class)
```

```
@SuiteClasses(value={  
    value=test1.class,  
    value=test2.class  
})
```

```
public class AllTests { ... }
```

Eigene Testklassen, die in  
der Test Suite ausgeführt  
werden sollen



Kann auch leer sein



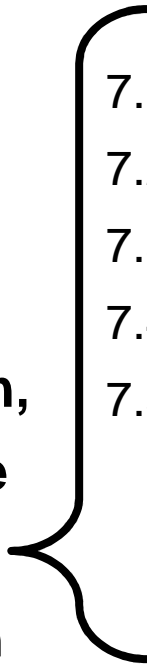
### Best Practices

- If it's hard to write a test, the code you are testing should probably be changed.  
Example: To test a private method, make it protected (or package-protected)
- Test anything that can reasonably break
- Always verify that tests fail when they should (this is particularly important if you're fixing a bug)
- In Test-Driven Development always write a failing test before writing any new code
- Test one object/issue at a time
- Test things which could break
- Tests should succeed quietly:  
Don't print „Doing foo ... done with foo!“
- What shouldn't I test:  
Don't test set/get methods  
Don't test the compiler



# Zusammenfassung und Ausblick

- 1 **Software-Krise und Software Engineering**
- 2 **Grundlagen des Software Engineering**
- 3 **Projektmanagement**
- 4 **Konfigurationsmanagement**
- 5 **Software-Modelle**
- 6 **Software-Entwicklungsphasen, -prozesse, -vorgehensmodelle**
- 7 **Qualität**
- 8 **... Fortgeschrittene Techniken**

- 
- 7.1 Einordnung und Begriff
  - 7.2 Qualitätseigenschaften
  - 7.3 Wege zur Qualität
  - 7.4 Qualität und Softwareentwicklung
  - 7.5 Unit-Test

→ **Wege im Umgang mit der Software-Krise und Umsetzung der Grundlagen und Prinzipien:  
Fähige SoftwareentwicklerInnen /  
Software IngenieurInnen**

