

# Programmierung 2

Kapitel 2

Testen: Unit Tests mit JUnit

# Verständnisfragen



- Muss in jeder Klasse ein Konstruktor definiert werden?
- Welchen Rückgabebetyp kann ein Konstruktor haben?
- Können Datenfelder (Attribute) einer Klasse vom Typ einer anderen Klasse sein?
- Was ist der Unterschied zwischen einer Instanz- und Klassenmethode?
- Kann in Java eine Klassenmethode vom Objekt, d.h. mit `objekt.methode()` aufgerufen werden?
- Wozu ist die Methode `toString()` gut?
- Wenn `toString()` in einer Klasse A (sinnvoll) überschrieben worden ist, wie kann ein Objekt der Klasse A auf dem Bildschirm ausgegeben werden?
- Muss jede Methode robust sein?

# Verständnisfragen



- Wie viele Dimensionen kann ein Array besitzen?
- Von welchem Datentyp sind bei einem mehrdimensionalen Array die Elemente der 1. Dimension?
- Ist die Anzahl der Dimensionen bei der Definition eines Arrays (einer Referenz auf ein Array) immer festgelegt?
- Was sind offene Arrays?
- Können mehrdimensionale Arrays Elemente verschiedener elementarer DT enthalten?

# Überblick Programmierung 2



## Software-Engineering

- Testen: Unit-Tests mit JUnit
- Strukturieren: Pakete
- Kommentieren: Javadoc, Annotationen

## OO- Programmierung

- Vererbung
- super-Operator
- Polymorphie von Objekten
- Finale Klassen
- Abstrakte Klassen und Schnittstellen
- Wrapper- Klassen
- generische Klassen

## Dynamische Datenstrukturen

- Verkettete Listen
- Stack, Queue, Binärbäume

## GUI

- GUI-Programmierung mit Swing

# Softwarequalität

# Was ist Software-Qualität?

- **Funktionalität** (*functional suitability*): SW macht das, was sie sollte, ist vollständig, angemessen, **sicher** und **korrekt** (d.h. **fehlerfrei** ..!)
- **Zuverlässigkeit** (*reliability*): SW ist **ausgereift**, **fehlertolerant** und **wiederherstellbar**, wenn sie mal abgestürzt ist
- **Effizienz** (*performance efficiency*): SW antwortet schnell und verbraucht nicht zu viele Ressourcen
- **Benutzbarkeit** (*usability*): SW ist benutzerfreundlich; verständlich, leicht erlernbar und bedienbar, attraktiv und akzeptiert
- **Änderbarkeit** (*modifiability*) ( $\approx$  **Wartbarkeit**, *maintainability*): SW ist leicht analysierbar (**modular** aufgebaut  $\rightarrow$  **wiederverwendbar**) und modifizierbar (insbes. erweiterbar und korrigierbar)
- **Übertragbarkeit** (*portability*): SW ist installierbar, austauschbar und auf verschiedenen Systemen nutzbar und anpassungsfähig

# Wie erreicht man Qualität? (1)



## Software-Entwurfsregeln:

- Anwendungs- und Testklasse **getrennt**
  - Vorteil: klare Programmstruktur, modularer Aufbau
- **Kapselung**: Instanz- und Klassenvariablen **private**, Schnittstellenmethoden **public**
  - (ausser es gibt einen sehr guten Grund, es anders zu tun)
  - Vorteil: Änderungen an Datenstrukturen (Instanzvariablen) haben *keinen Einfluss* auf die Zugriffsmethoden, flexibel
- **Do One Thing**: *eine* Methode erledigt nur *eine* Aufgabe
  - Vorteil: verständlicher, einfacher zu ändern (klar, an welcher Stelle Änderungen durchzuführen), einfacher Fehler zu finden
- **Don't Repeat Yourself** → eine Aufgabe an *einer* Stelle erledigt
  - Vorteil: Änderungen (kommen sicher!) müssen nur an *einer* Stelle durchgeführt werden

# Wie erreicht man Qualität? (2)



## Software-Entwurfsregeln:

- Numerische Literale (42, 10, 5f) nicht direkt codieren (**magic numbers!**), sondern **Konstanten verwenden**:

```
private static int NUMBER_OF_STUDENTS = 44;
```

(Ausnahme: Zählvariablen in Schleifen)

→ Vorteil: Änderungen der Werte müssen nur an *einer* Stelle durchgeführt werden

- **defensive Implementierung**: mögliche Fehler antizipieren, ankündigen, darauf reagieren (*Exceptions* ankündigen, werfen und fangen oder Rückgabe eines *Fehlercodes* per return)
  - **Methodenzugriff**:
    - `objekt.instanzmethode()`
    - `Klasse.klassenmethode()`
- Vorteil: verständlicher



# Wie erreicht man Qualität? (3)



## Software-Entwurfsregeln:

→ Vorteil: Verständlichkeit, Lesbarkeit

- **Klammern verwenden** statt sich auf Prioritäten zu verlassen (Lesbarkeit):  
statt: `if (a == b && c == d)`  
besser: `if ((a == b) && (c == d))`
- **Namenskonventionen** beachten (**aussagekräftige Namen**)
  - Klassen: `class StudentenVerwaltung`
  - Methoden (**ein Verb**): `kursAuswaehlen()`, `abmelden()`
  - Variablen: `int matrikelNr`, `String vorname`
  - Konstanten: `int ANZAHL_ZUEGE`, `int ZUGSTAERKE`
- **Formatierungsrichtlinien** beachten (Einrückungen ...)
- Quelltext **kommentieren** (sinnvoll und ausreichend)
- Quelltext **dokumentieren** (Headers)

# Wie erreicht man Qualität? (4)

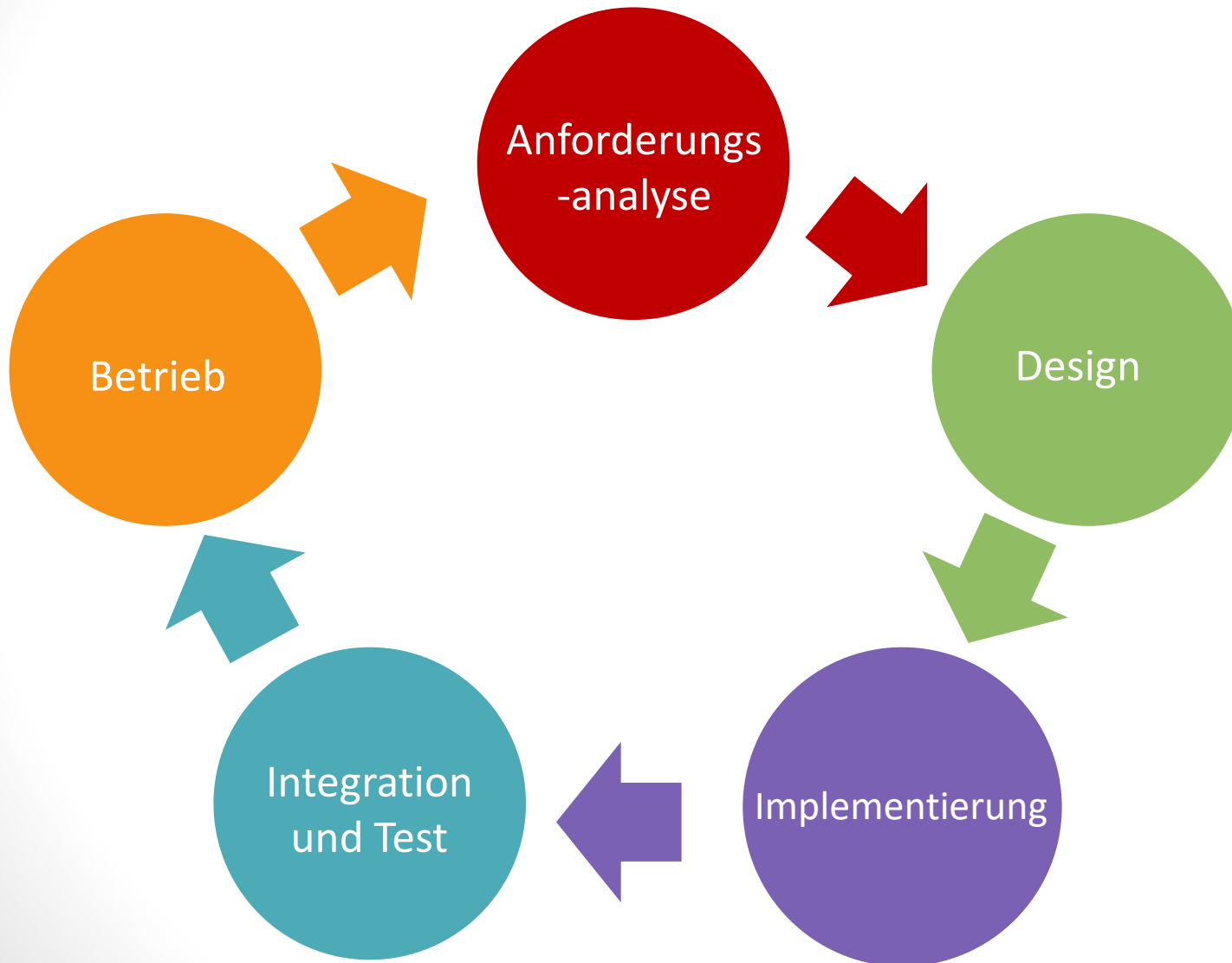


## Software-Entwurfsregeln:

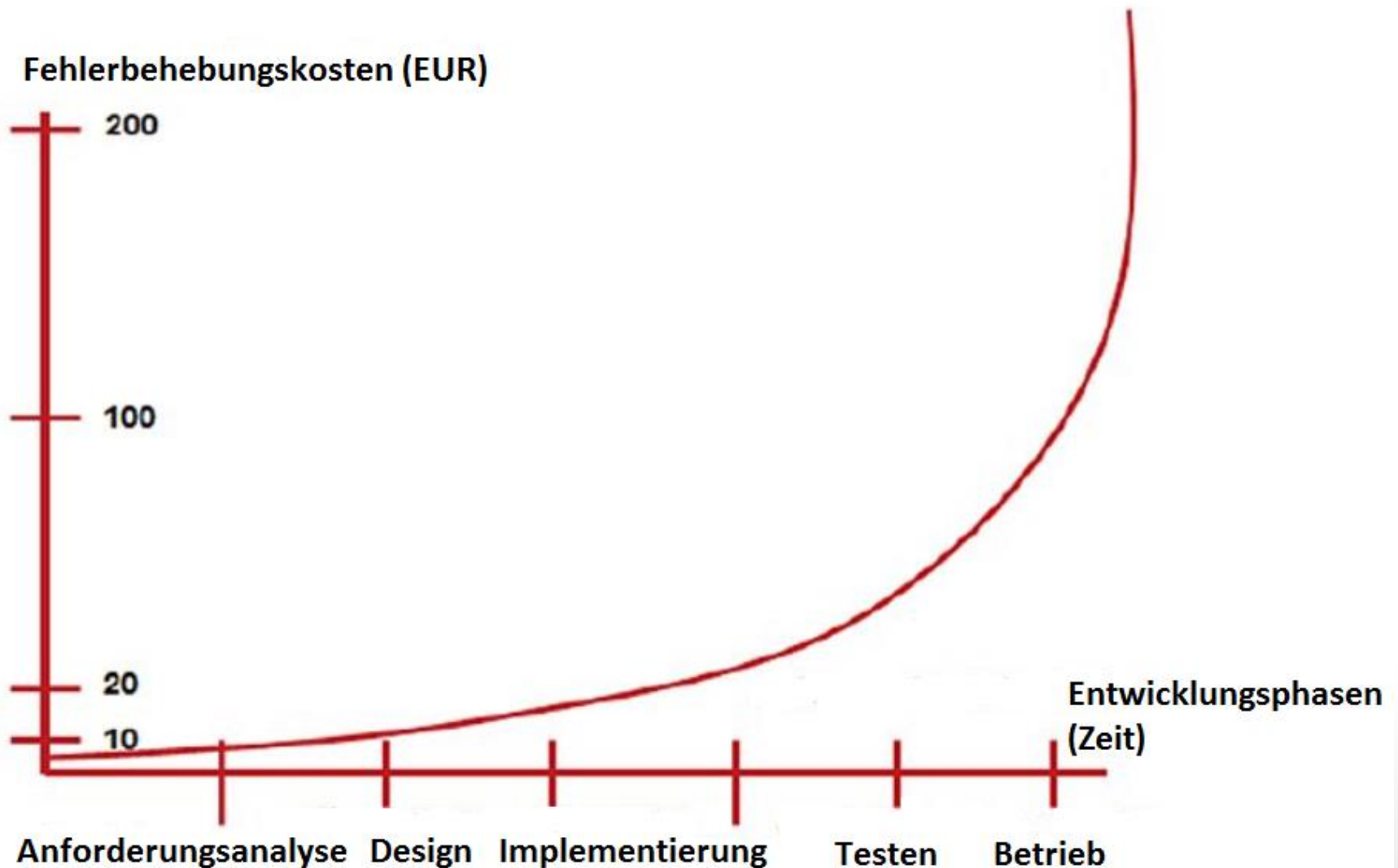
- **Testen (Unit Tests)**
  - Vorteil: Funktionalität gesichert, weniger Fehler

# Softwaretest

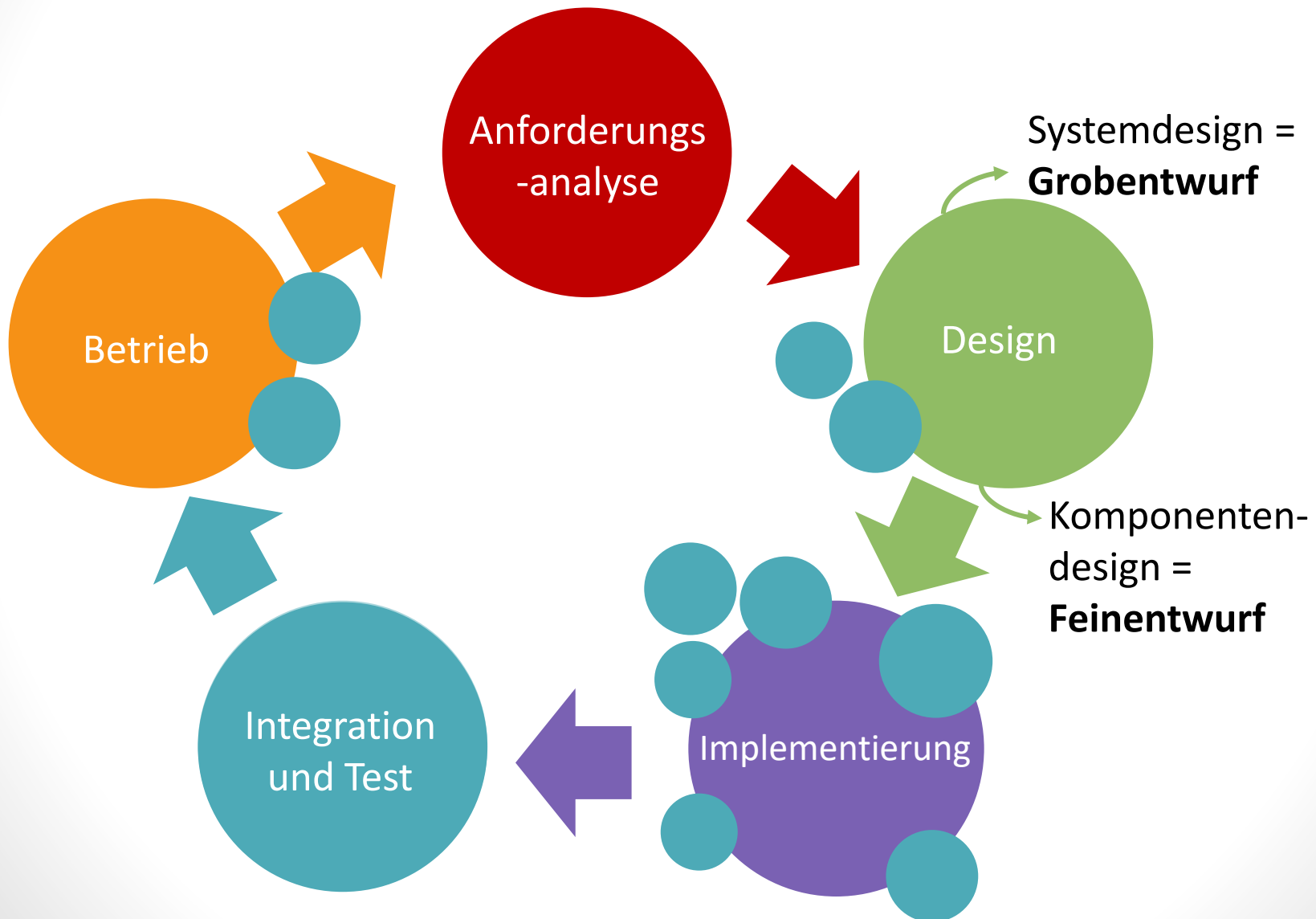
# Software-Entwicklungsprozess



# Kosten der Fehlerbehebung



# Software-Entwicklungsprozess



# Grenzen der Prüfbarkeit

„Program testing can be used to show the presence of bugs, but never show their absence!“

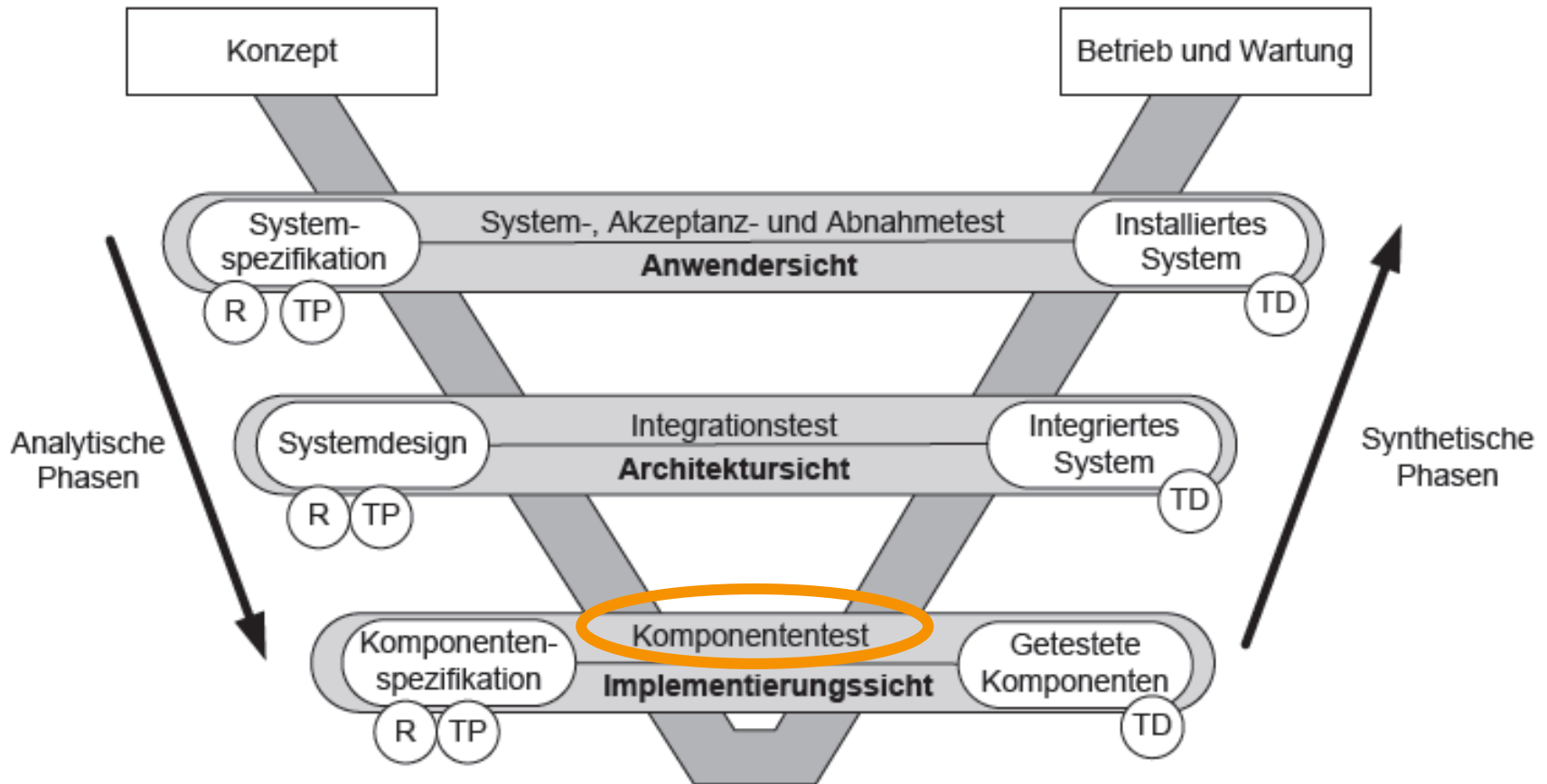
(Das Testen von Programmen kann die Existenz von Fehlern zeigen, aber niemals deren Nichtvorhandensein.)



Edsger W. Dijkstra

- Gründe: alle möglichen Eingabewerte müssten in allen möglichen Kombinationen getestet werden → praktisch nicht möglich, da begrenzt viele Ressourcen (Zeit, Speicher) verfügbar
- Durch Testen **keine Fehlerfreiheit der Software** nachweisbar!
- wenn ein Test fehlgeschlagen → Fehler in der Software
- wenn alle Tests korrekt durchgelaufen → ? (keine Aussage)

# V-Modell



Legende: R ... Reviews, TP ... Testplanung- und Definition, TD ... Testdurchführung



# Testebene: Komponententest

**Komponententest** (auch: **Unit Test**):

- ein Stück Quelltext geschrieben *nur zum Testen* einer **kleinen**, klar umrissenen **Funktionalität** (meist eine Methode oder sogar nur Teil einer Methode)
- vom Entwickler implementiert → auch: **Entwicklertest**
- auch: *Durchführung* eines Komponententests

# Komponententest: Beispiele

- a) Methode `void append(myArray, 2)`: hängt die Zahl 2 ans Ende des Arrays `myArray`

→ Unit Test von `append()`: Array `myArray` ausgeben und überprüfen, ob 2 ans Ende angehängt worden ist

```
int[] myArray = [1, 2, 3];  
printArray(myArray);  
append(myArray, 2);  
printArray(myArray);
```

- b) Methode `void delete(myString, 'a')`: löscht alle a's aus der Zeichenkette `myString`


→ Unit Test von `delete()`: Zeichenkette `myString` ausgeben und überprüfen, ob alle a's gelöscht worden sind

# Wie schreibe ich ein Unit Test?

- Gibt es **Vorbedingungen**, die erfüllt sein müssen?
  - Was sind die **Eingabewerte**?
  - Welche **Schritte** führe ich im Test aus (= **Testszenario**)?
  - Was sind die **erwarteten Ausgabewerte**?
  - zusätzliche Infos: *wer* (Autor) und *wann* (Datum) führt den Test aus, welche Programmversion wird getestet, *was* wird getestet → bei Entwicklertests oft weggelassen
- Test auswerten (lassen) = **Testergebnis**

# Manuell vs. automatisiert

Ausführen und Auswerten von implementierten Tests → zwei Möglichkeiten:

- manuell
  - print-Methoden (`System.out.print()`, ...), Ausführen mit , Auswerten durch Programmierer (Überprüfung der Ausgabe)
- automatisiert
  - Ausführen und Auswerten durch ein Testwerkzeuges

# Verständnisfragen



- Was ist Software-Qualität?
- Was ist das wichtigste Merkmal der Software-Qualität?
- Wie erreicht man Software-Qualität?
- Was ist der Software-Lebenszyklus?
- Wann entdeckte Software-Fehler kosten in der Behebung am meisten?
- Kann man durch ausreichend viele Tests garantieren/nachweisen, dass die Software *fehlerfrei* ist?
- Was ist ein Unit-Test?
- Kann man Unit Tests manuell implementieren? Wie?

# Einführung in JUnit

# Testframework

Anforderungen an Automatisierungs(test)frameworks

- **Sprache der Testspezifikation** (d.h. Sprache, in der Testfälle beschrieben wurden) = **Programmiersprache** selbst
- **Trennung des Anwendungscodes und Testcodes:**  
Anwendungsklasse und eigene Testklasse
- Ausführung einzelner Testfälle *unabhängig* voneinander, d.h. Ausführung eines Testfalls ohne Auswirkungen auf nachfolgende Testfälle → Reihenfolge der Tests beliebig
- zusammengehörige Testfälle gemeinsam behandelt (z.B. automatisch ausgeführt) → Sammlung von Tests = sog. **Testsuite**
- **Report:** Erfolg oder Misserfolg einer Testausführung auf einen Blick erkennbar → nicht lange durch Testreports blättern

# JUnit

**JUnit** = **Testframework** für Java zum Schreiben und automatischen Ausführen von Unit Tests

- [www.junit.org](http://www.junit.org)
- Autoren: Erich Gamma, Kent Beck, 1998
- Open-Source-Software
- sehr populär, Quasistandard für Java-Entwicklertests
- in Entwicklungsumgebungen Eclipse, NetBeans und IntelliJ bereits integriert

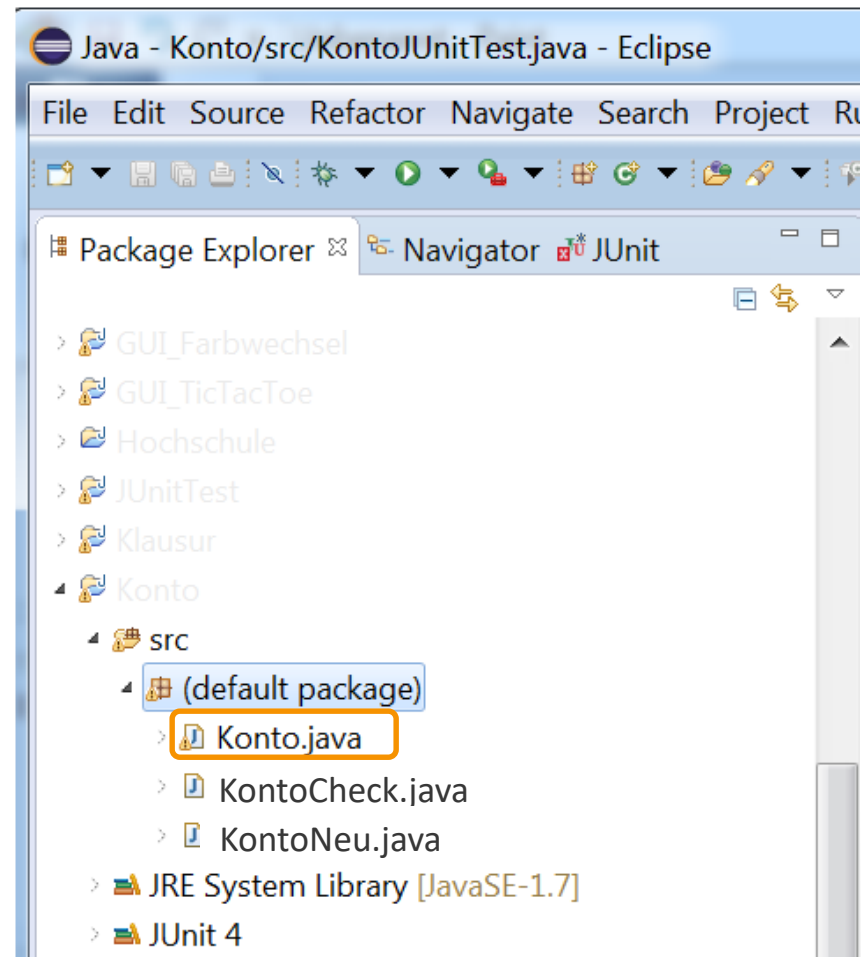


# Class under test

zu testende Klasse (oft: Class Under Test = CUT): Konto

- Konto anlegen
- Kontostand erfragen
- Geld einzahlen
- Geld abheben

→ Kontostand als float

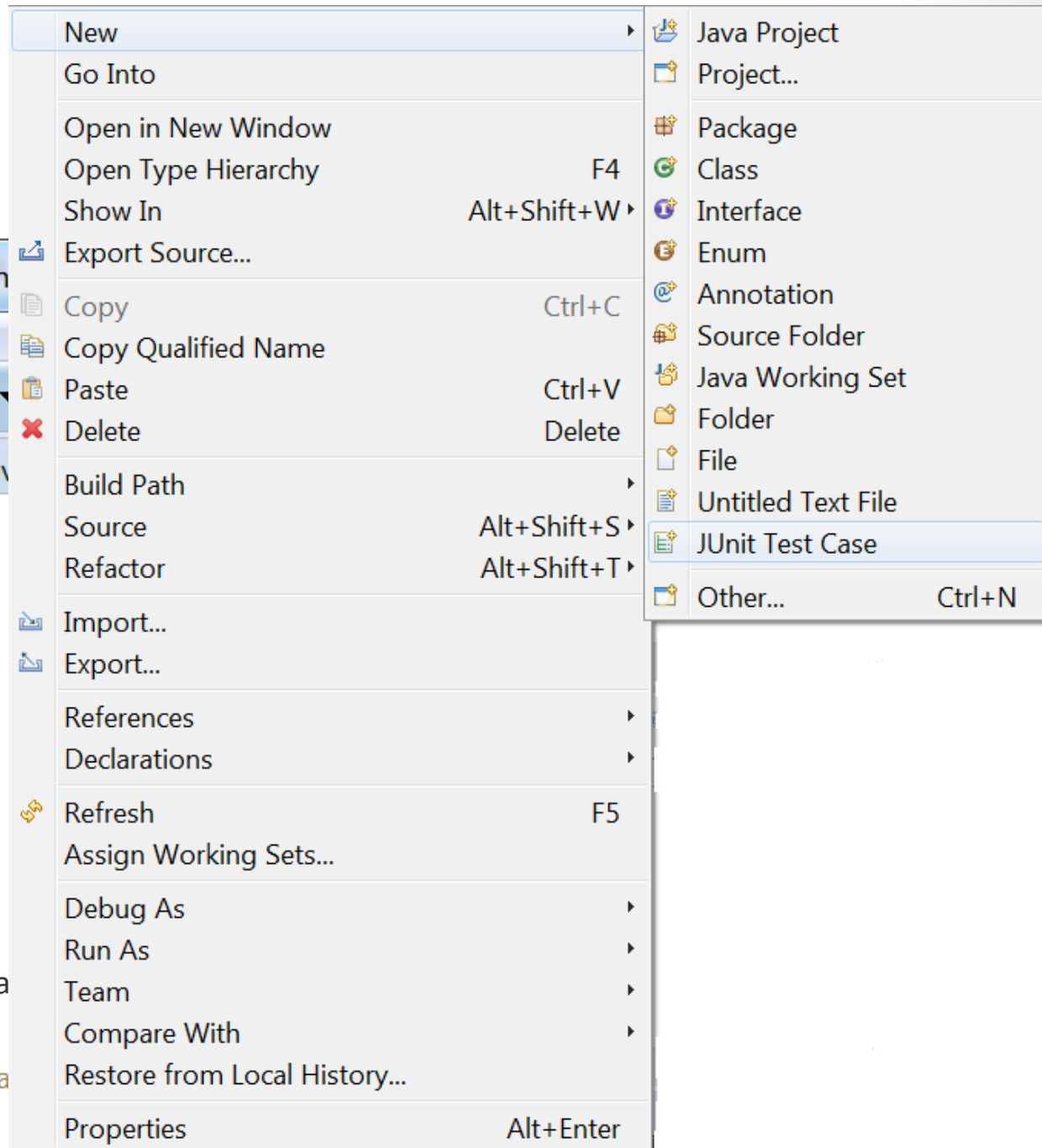
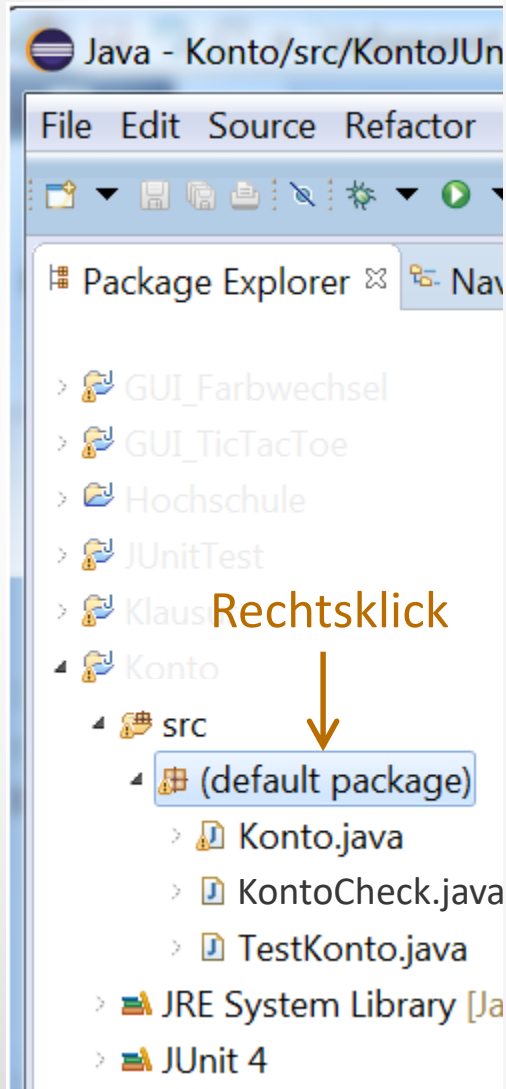


# Klasse Konto

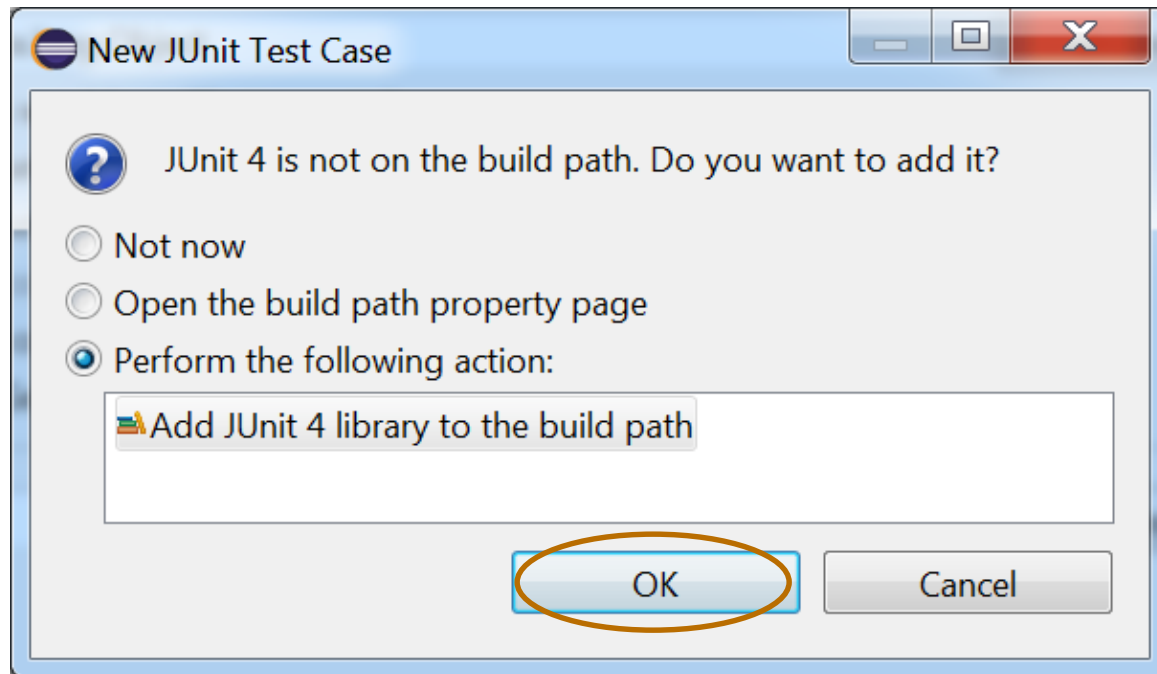
zu testende Klasse (= CUT): Konto

```
public class Konto {  
    private float kontostand;  
    public float getKontostand() {  
        ...  
    }  
    public void abheben(float summe) {  
        ...  
    }  
    public void einzahlen(float summe) {  
        ...  
    }  
}
```

# JUnit-Testklasse innerhalb eines Projektes anlegen



# JUnit-Libs zum build path hinzufügen



# Testklasse benennen

**New JUnit Test Case**

**JUnit Test Case**

⚠ The use of the default package is discouraged.

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()  
☐ setUp() ☐ tearDown()  
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))  
☐ Generate comments

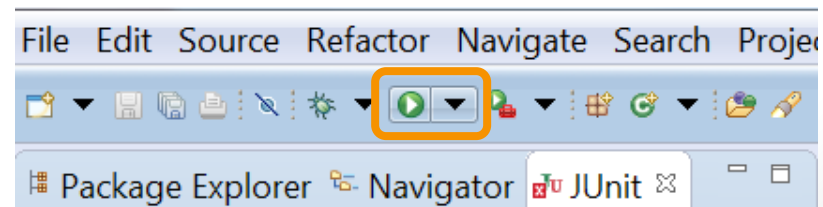
Class under test:

# Testklasse erzeugt

```
1 import static org.junit.Assert.*;
2 import org.junit.Test;
3
4
5 public class KontoTest {
6
7     @Test
8     public void test() {
9         fail("Not yet implemented");
10    }
11
12 }
13
```

lässt einen Test explizit fehlschlagen

- Testlauf (erstmalig) starten:  
im **Package Explorer**: Rechtsklick auf Testklasse  
KontoTest.java → Run as → 1 JUnit Test
- für jeden weiteren Testlauf:  
einfach Play-Button drücken



# Testergebnis

roter Balken = Test fehlgeschlagen

grüner Balken = Test bestanden

Package Explorer Navigator JUnit

Finished after 0,021 seconds

Runs: 1/1 Errors: 0 Failures: 1

KontoTest [Runner: JUnit 4] (0,000 s)

test (0,000 s)

Failure Trace

java.lang.AssertionError: Not yet implemented

at KontoTest.test(KontoTest.java:9)

- wie testen → Unit Tests mit JUnit
- welche Fälle testen
- wann testen → TDD

## Unit Test für Klasse Konto



# getKontostand() testen

```
public class KontoTest {  
    @Test  
    public void testGetKontostand() {  
        Konto ko = new Konto();  
        assertTrue(ko.getKontostand() == 0f);  
    }  
}
```

Annotation: ein Testfall  
oder: Testnamen mit *test* beginnen

↑  
überprüfe, ob Bedingung wahr

*to assert (engl.)* = behaupten  
`assertTrue(boolean condition)`

```
public float getKontostand() {  
    ...  
}
```

# einzahlen() testen

```
public class KontoTest {  
  
    @Test  
    public void testGetKontostand() {...}  
  
    @Test  
    public void testEinzahlenNormallfall() {  
        Konto ko = new Konto();  
        ko.einzahlen(10.5f);  
        assertTrue(ko.getKontostand() == 10.5f);  
    }  
}  
  
public float einzahlen() {  
    ...  
}
```

# einzahlen() weiter testen

mehrmals Methoden aufrufen → Testszenario

```
public class KontoTest {  
  
    @Test  
    public void testGetKontostand() {...}  
  
    @Test  
    public void testEinzahlenNormalfall() {...}  
  
    @Test  
    public void testEinzahlenZweimal() {  
        Konto ko = new Konto();  
        ko.einzahlen(25.0f);  
        ko.einzahlen(4.5f);  
        assertTrue(ko.getKontostand() == 29.5f);  
    }  
}
```

# Welche Fälle testen?

- Normalfall (*Happy Path*)
- Grenzfall (Grenzwerte der zulässigen Eingaben)
- bester (günstigster) Fall (*best case*)  
*Beispiel:* Arrays sortieren → Test-Eingabe: bereits sortiertes Array sortieren
- schlechtester (ungünstigster) Fall (*worst case*)  
*Beispiel:* Arrays sortieren → Test-Eingabe: umgekehrt sortiertes Array sortieren
- unzulässige Eingaben → Fehler provozieren  
*Beispiel:* Methode abheben() → negative Zahl abheben, Arrays sortieren → leeres Array sortieren
- Ressourcen-Fresser → Leistungsfähigkeit des Programms  
*Beispiel:* Arrays sortieren → Array der Länge 1000 sortieren
- ...

# Wann testen?



Erst implementieren, dann testen?  
Oder vielleicht umgekehrt?

# Test-Driven Development

*Test-Driven Development* (TDD, auch *Test-First Development*):  
Ansatz aus dem Bereich der agilen Software-Entwicklung



## Grundidee:

- ZUERST Testfälle (zur Beschreibung der Funktionalität) erstellen
- DANN die eigentliche Funktionalität implementieren und testen

?... geht nicht, wenn man nicht zumindest ein paar Infos über die zu implementierende Methode hat ...

# TDD: 1. Schritt

Vorgehensweise:

Gegeben: Beschreibung ( $\approx$  Anforderungen) der Methode

## 1. **Kompilierbaren** Quellcode für Methode anlegen

- Methodenkopf implementieren
- Methodenkopf kommentieren (was tut Methode, welche Parameter hat sie, was liefert sie zurück)
- **keine Logik** (Funktionalität) der Methode implementieren!  
→ leeren Methodenrumpf oder Attrappe

*Beispiele:*

```
public void abheben(float summe) { ; }
```

```
public float getKontostand() {return -1;}
```

leere Anweisung



# TDD: weitere Schritte

Vorgehensweise (Forts.)

2. Unit Tests in einer Testklasse implementieren
  - 2.1 Unit Tests durchführen → alle **fehlgeschlagen**
3. Logik der Methode implementieren (ggf. Unit Tests ergänzen)
4. Tests ausführen
  - alle Tests **bestanden** → gehe zu 5
  - ein Test **fehlgeschlagen** → gehe zu 3→ solange wiederholen, bis alle Tests **bestanden**
5. Code- und Testqualität verbessern → *Refactoring*



# Klasse Konto

zu testende Klasse (= CUT): Konto

```
public class Konto {  
    private float kontostand;  
    public float getKontostand() {  
        return -1;  
    }  
    public void abheben(float summe) {  
        ;  
    }  
    public void einzahlen(float summe) {  
        ;  
    }  
}
```

# getKontostand() testen

2.

```
public class KontoTest {  
  
    @Test  
    public void testGetKontostand() {  
        Konto ko = new Konto();  
        assertTrue(ko.getKontostand() == 0f);  
    }  
}  
  
    public float getKontostand() {  
        return -1;  
    }
```

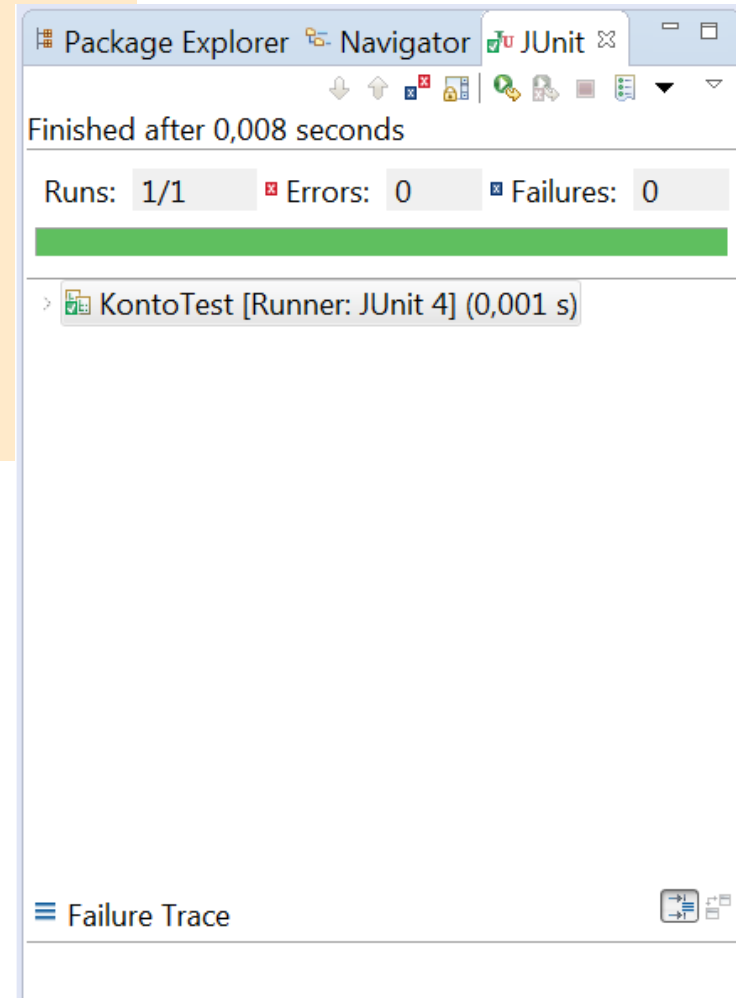
Testlauf → wie erwartet: Test fehlgeschlagen (**roter Balken**)

# getKontostand() korrigieren

3.

```
public class Konto {  
    private float kontostand;  
    public float getKontostand()  
        return kontostand;  
}  
...  
}
```

4. → Test bestanden (grüner Balken)



# einzahlen() testen

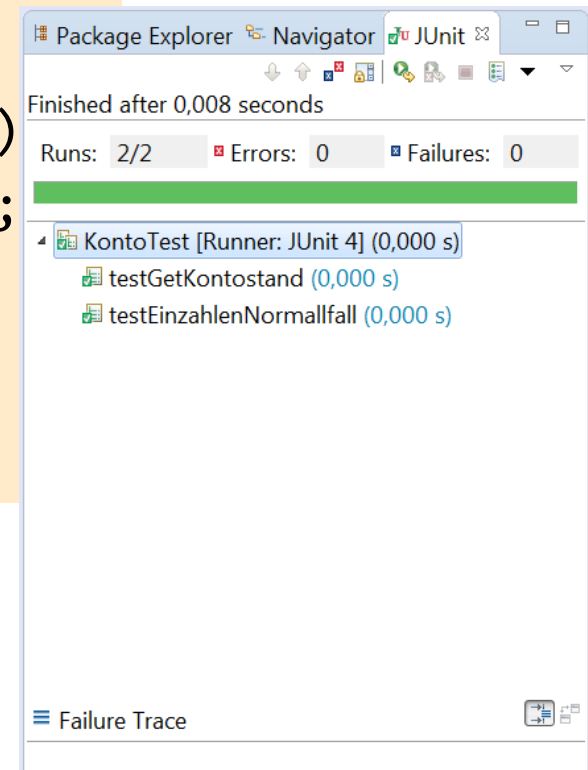
```
public class KontoTest {  
  
    @Test  
    public void testGetKontostand() {...}  
  
    @Test  
    public void testEinzahlenNormalfall() {  
        Konto ko = new Konto();  
        ko.einzahlen(10.5f);  
        assertTrue(ko.getKontostand() == 10.5f);  
    }  
}  
  
    public float einzahlen() {  
        ;  
    }
```

Testlauf → wie erwartet: Test fehlgeschlagen (**roter Balken**)

# einzahlen() korrigieren

3.

```
public class Konto {  
    private float kontostand;  
    public float getKontostand() {  
        return kontostand;  
    }  
    public void einzahlen(float summe)  
        kontostand = kontostand + summe;  
    }  
}
```



4. Testlauf → Test bestanden (grüner Balken)

# einzahlen() weiter testen

mehrmals Methoden aufrufen → Testszenario

```
public class KontoTest {  
  
    @Test  
    public void testGetKontostand() {...}  
  
    @Test  
    public void testEinzahlenNormalfall() {...}  
  
    @Test  
    public void testEinzahlenZweimal() {  
        Konto ko = new Konto();  
        ko.einzahlen(25.0f);  
        ko.einzahlen(4.5f);  
        assertTrue(ko.getKontostand() == 29.5f);  
    }  
}
```

Testlauf → Test bestanden (grüner Balken)

# Refactoring

- wenn alle Tests bestanden (d.h. **Balken grün**) → ein Teststand erreicht → Zeit für **Refactoring** ...

Wo sollte man in den Klassen Konto und KontoTest **aufräumen**?

# Refactoring

**Refactoring** (Refaktorisierung) = Verbesserung des **Quellcodes**, ohne dabei das Systemverhalten zu ändern [Fowler, 2000]

→ Ziele: **Wartbarkeit**, **Lesbarkeit** und **Verständlichkeit** des **Quellcodes** verbessern → damit auch: **Wiederverwendung** des Quellcodes zu erhöhen

danach sicherstellen, dass Refactoring das Verhalten nicht geändert hat, d.h. dass durch die Modifikationen keine neuen Fehler (sog. **Regressionen**) verursacht!



WIE?

→ **Unit Tests** nochmals ausführen und schauen, ob der Balken immer noch **grün** ist (→ **Regressionstest**)



# Refactoring: Beispiele

## *typische Aktivitäten (Beispiele):*

- Namen von Klassen und/oder Methoden ändern
- redundanten Code entfernen
- duplizierten Code bereinigen
- zwei ähnliche Klassen aus zwei Subsystemen zu einer Klasse zusammenfassen (ggf. Vererbung einsetzen)
- Klassen ohne Verhalten in Attribute umwandeln
- komplexe Klasse in mehrere einfachere Klassen aufteilen
- Neuordnung von Klassen und Operationen zur Erhöhung des Vererbungs- und Zusammenhangsgrades
- ...

# Duplizierter Code



```
@Test
public void testGetKontostand() {
    Konto ko = new Konto();
    assertTrue(ko.getKontostand() == 0f);
}

@Test
public void testEinzahlenNormalfall() {
    Konto ko = new Konto();
    ko.einzahlen(10.5f);
    assertTrue(ko.getKontostand() == 10.5f);
}

@Test
public void testEinzahlenZweimal() {
    Konto ko = new Konto();
    ko.einzahlen(25.0f);
    ko.einzahlen(4.5f);
    assertTrue(ko.getKontostand() == 29.5f);
}
```

in jedem Testfall ein  
neues Objekt (später:  
mehrere Objekte) erzeugt

→ **Anfangszustand**  
hergestellt

→ in JUnit möglich: in  
eine extra Methode  
auslagern

- annotiert: `@Before`  
(ab JUnit5:  
`@BeforeEach`)
- `setUp()` nennen

# Anfangszustand herstellen

```
public class KontoTest {  
    private Konto ko;  
  
    @Before //oder: @BeforeEach  
    public void kontoErzeugen() {  
        ko = new Konto();  
    }  
  
    @Test  
    public void testGetKontostand() {  
        Konto ko = new Konto();  
        assertTrue(ko.getKontostand() == 0f);  
    }  
  
    @Test  
    public void testEinzahlenNormallfall() {  
        Konto ko = new Konto();  
        ko.einzahlen(10.5f);  
        assertTrue(ko.getKontostand() == 10.5f);  
    }  
}
```

← zusätzlich: **import org.junit.Before;**

← Methode wird **vor** jedem Testfall ausgeführt

# Nach Testfall aufräumen

- analog zur Herstellung des Anfangszustandes – durch Methode `setUp()` oder eine andere als `@Before` annotierte Methode – in JUnit möglich, das **Aufräumen nach einem Testfall** in eine Methode auszulagern:
  - Methode `tearDown()` („niederreißen“) verwenden
  - eine Methode als `@After` annotieren (ab JUnit5: `@AfterEach`)
    - wird **nach** jedem Testfall ausgeführt

Was wird aufgeräumt?

Ressourcen freigeben (z.B. Dateien, Datenbankverbindungen usw.)

# Methodennamen ändern



```
@Test                testEinzahlenEinmal()  
public void testEinzahlenNormalfall() {  
    ko.einzahlen(10.5f);  
    assertTrue(ko.getKontostand() == 10.5f);  
}
```

# Andere assert-Methode



@Test

```
public void testEinzahlenZweimal() {  
    ko.einzahlen(25.0f);  
    ko.einzahlen(4.5f);  
    assertTrue(ko.getKontostand() == 29.5f);  
    assertEquals(29.5f, ko.getKontostand(), 0);  
}
```

erwartet      tatsächlich      zulässige Differenz

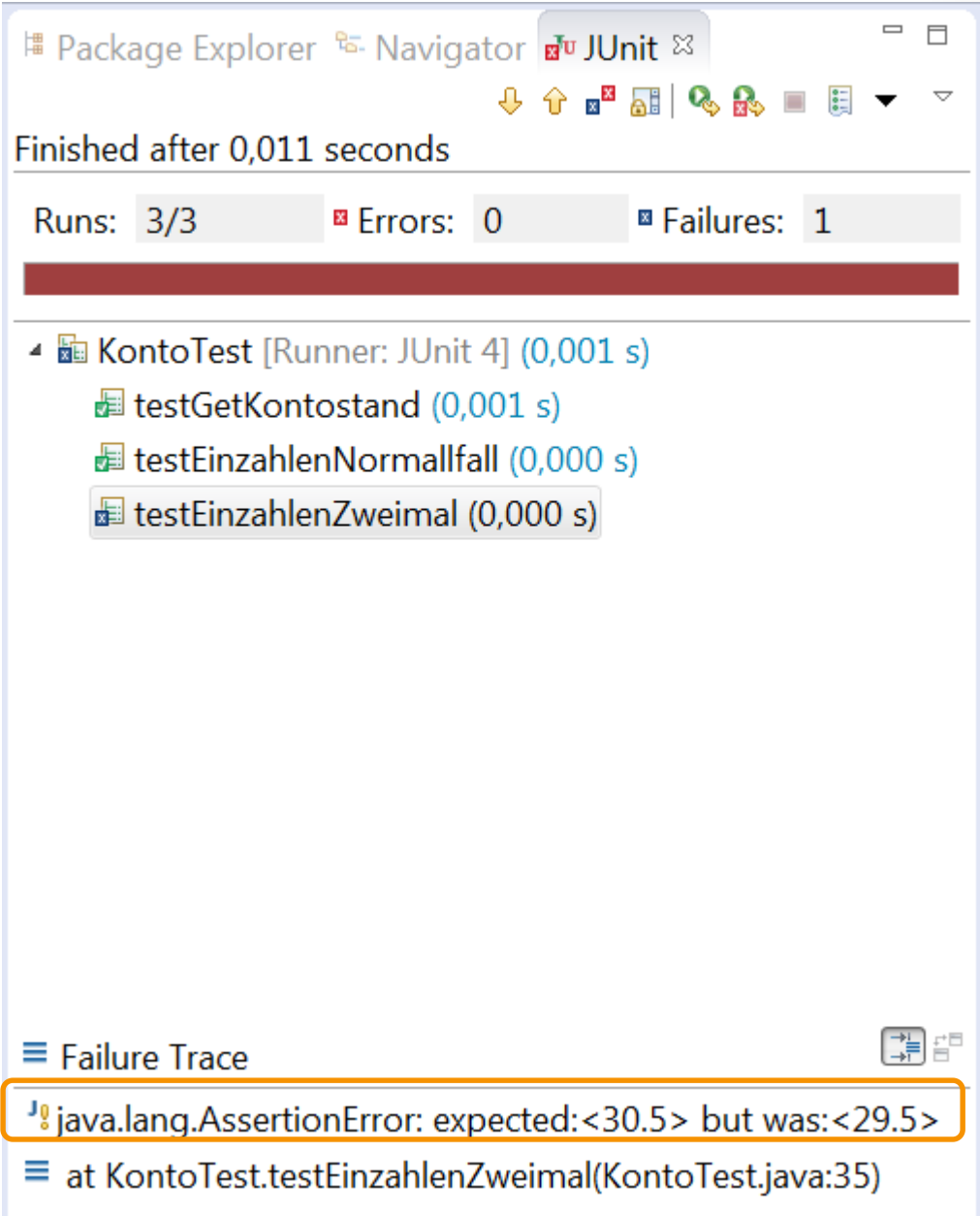
- assertEquals(float expected, float actual, float delta) → überprüft, ob die Differenz zwischen expected und actual  $\leq$  delta ist
- analog für double:  
    assertEquals(double exp, double act, double delta)

# Warum besser?

```
@Test
public void testEinzahlenZweimal() {
    ko.einzahlen(25.0f);
    ko.einzahlen(4.5f);
assertTrue(ko.getKontostand() == 29.5f);
    assertEquals(30.5f, ko.getKontostand(), 0);
}
```

# Report

Report genauer →  
zeigt nicht nur das  
Fehlschlagen des Tests  
an, sondern auch den  
Grund dafür



Package Explorer Navigator JUnit

Finished after 0,011 seconds

Runs: 3/3 Errors: 0 Failures: 1

KontoTest [Runner: JUnit 4] (0,001 s)

- testGetKontostand (0,001 s)
- testEinzahlenNormallfall (0,000 s)
- testEinzahlenZweimal (0,000 s)

Failure Trace

java.lang.AssertionError: expected: <30.5> but was: <29.5>

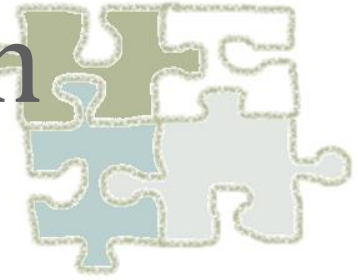
at KontoTest.testEinzahlenZweimal(KontoTest.java:35)



# Weitere Testfälle für `einzahlen()`

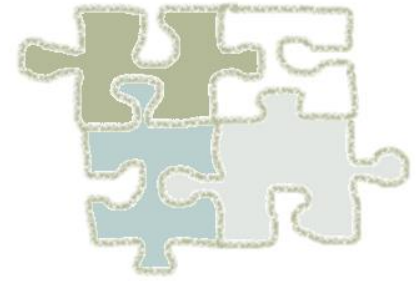
- testen, ob beim negativen Kontostand die Methode `einzahlen()` korrekt funktioniert
  - dazu Methode `abheben()` und ggf. weitere Methoden (wie z.B. `kontoLeer()`) benötigt
- **Fehlerfall:** testen, ob `einzahlen()` bei negativer Eingaben korrekt funktioniert
  - was soll da eigentlich passieren?...

# Testimplementierung nach TDD zusammengefasst



- Methode in der Klasse Konto mit einem dummy-Methodenrumpf anlegen
- Testfälle für diese Methoden in KontoTest implementieren
  - gewünschten Anfangszustand herstellen (@Before oder setUp())
  - eigentliche Tests implementieren (@Test oder test...())
  - hinterher aufräumen (@After oder tearDown())
- Tests ausführen → **fehlgeschlagen**
- Methode in Konto richtig implementieren
- Testfälle nochmals ausführen → bis alle **bestanden**
- **Refactoring**

# TDD: Ablauf



kompilierbaren Quellcode anlegen  
(Methodenkopf + Dummy-Rumpf)

Unit Tests implementieren

ausführen

alle fehlgeschlagen

Methodenrumpf implementieren  
(Logik der Methode)

Unit Tests ausführen

alle bestanden

Refactoring

ein fehlgeschlagen

# Vorteile von TDD

- **Anforderungen** in Form von Testfällen klar und nachvollziehbar dokumentiert (und auch verstanden)
- Quellcode **qualitativ besser** (weniger Fehler)
- **Entwicklungszeit** insgesamt **verkürzt**: Zeit am Anfang (Implementierungsphase) investiert, aber hinterher (Testphase) mehr gespart



Test-First-Ansatz hat den großen Vorteil, dass er überschnelle Entwickler, die ohne groß zu denken zur Tastatur greifen, dann implementieren und nach 20 Minuten wieder alles ändern, zum Nachdenken zwingt.

Ullenboom: *Java ist auch eine Insel*, Bonn 2014

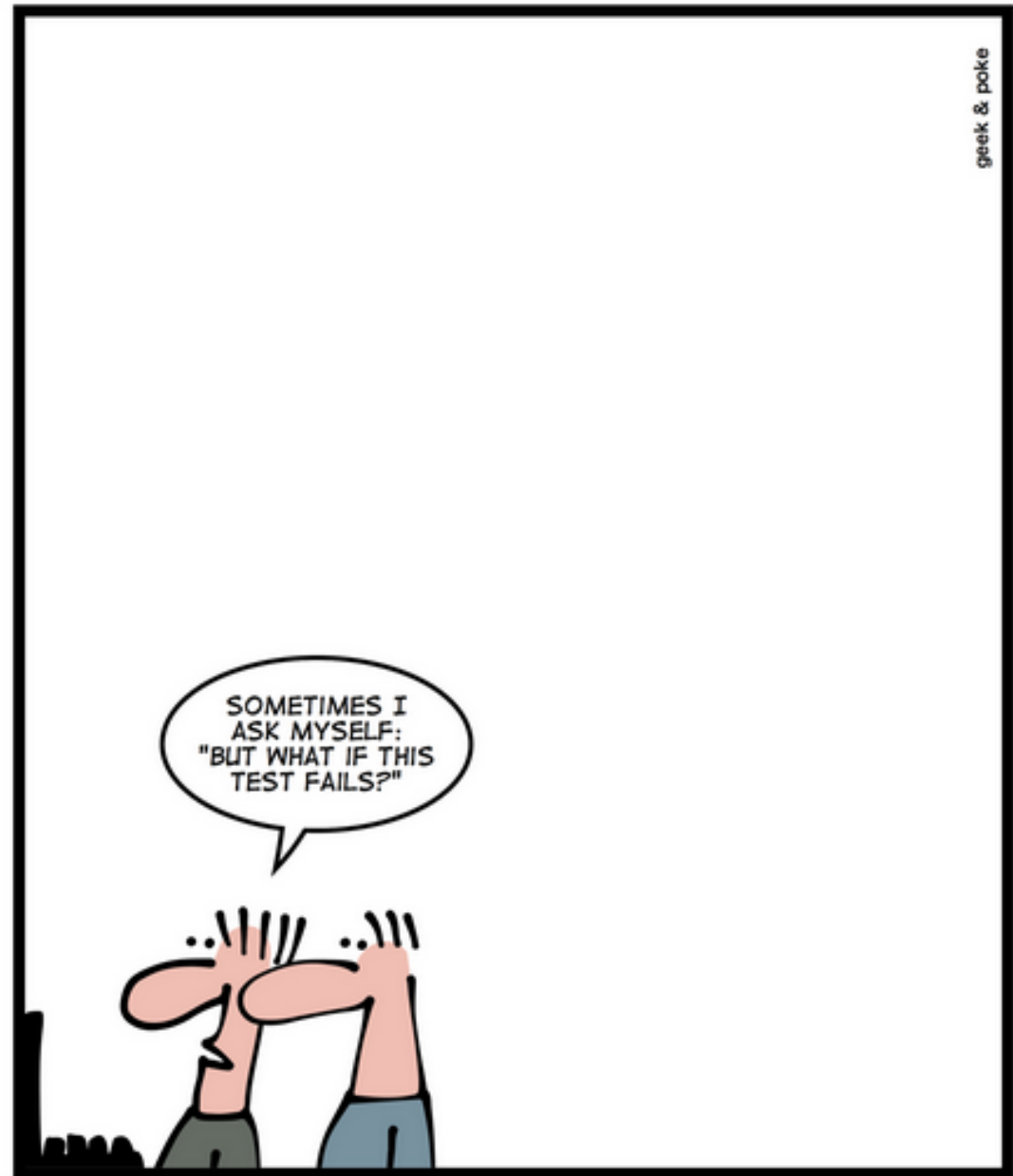
<http://www.tutego.de/blog/javainsel/2010/04/junit-4-tutorial-java-tests-mit-junit/>, 26.3.2015

# Verständnisfragen



- Wenn alle Tests bestanden sind, kann man davon ausgehen, dass die Software fehlerfrei ist?
- Was ist ein Uni-Test?
- Was muss ein Unit-Test z.B. einer Methode enthalten?
- Was sind Ziele eines Unit-Tests?
- Welche Fälle testet man in einem Uni-Test?
- Wann sollte man Unit-Tests schreiben: vor oder nach der Implementierung der Komponente (Unit)?
- Was ist TDD?
- Wie geht man beim TDD vor? Zählen Sie die Testschritte auf und erläutern sie (z.B. beim Testen einer Methode).
- Was sind die Vorteile von TDD?

# PHILOSOPHISING GEEKS



Mehr zu JUnit

`assert(true);`

# Wieviel Tests pro Methode?

- *mindestens ein* Testfall für jede Methode
  - oft zu wenig
- besser: ein Testfall für jede *typische Verwendung* eines Objektes
  - meist nicht nur durch *eine* Methode, sondern durch eine Sequenz von Methodenaufrufen

# Assert-Methoden

Assert-Methode	Beschreibung
<code>assertTrue(boolean condition)</code>	Prüft, ob die Bedingung <b>wahr</b> ist
<code>assertFalse(boolean condition)</code>	Prüft, ob die Bedingung <b>falsch</b> ist
<code>assertEquals(int exp, int act)</code>	Prüft, ob zwei <code>int</code> -Werte gleich sind. Auch für <code>byte</code> , <code>short</code> , <code>long</code> und <code>char</code> .
<code>assertEquals(float exp, float act, float delta)</code>	Prüft, ob zwei <code>float</code> -Werte bis auf die Differenz <b>delta</b> gleich sind. Auch für <code>double</code> .
<code>assertArrayEquals(int[] exp, int[] act)</code>	Prüft, ob zwei <code>int</code> -Arrays gleich sind. Auch für <code>byte</code> , <code>short</code> , <code>long</code> und <code>char</code> .
<code>assertArrayEquals(float[] exp, float[] act, float delta)</code>	Prüft, ob zwei <code>float</code> -Arrays bis auf die Differenz <b>delta</b> gleich sind. Auch für <code>double</code> .
<code>assertEquals(Object exp, Object act)</code>	Prüft anhand der <code>equals</code> -Methode des Objektes, ob zwei Objekte gleich sind.
<code>assertNull(Object object)</code> <code>assertNotNull(Object object)</code>	Prüft, ob das Objekt gleich <code>null</code> ist. Prüft, ob das Objekt ungleich <code>null</code> ist.
<code>assertSame(Object exp, Object act)</code> <code>assertNotSame(Object unexp, Object act)</code>	Prüft, ob zwei Referenzen auf das gleiche Objekt zeigen. ...auf unterschiedliche Objekte zeigen
<code>fail(String message)</code>	Lässt Test fehlschlagen, keine Prüfmethode



# Annotationen

Annotation	Beschreibung
<b>@Test</b> <code>public void method()</code>	Die Methode ist eine Testmethode
<b>@Before</b> <code>public void method()</code>	Die Methode wird vor jedem Test ausgeführt
<b>@After</b> <code>public void method()</code>	Die Methode wird nach jedem Test ausgeführt
<b>@BeforeClass</b> <code>public static void method()</code>	Die Methode wird einmalig ausgeführt bevor die Tests starten (Methode muss static sein) -- für aufwendige Vorbereitungen (Objekte erzeugen, Datenbankverbindung herstellen)
<b>@AfterClass</b> <code>public static void method()</code>	Die Methode wird einmalig ausgeführt nachdem die Tests gelaufen sind (Methode muss static sein)
<b>@Ignore("Kommentar")</b> <code>public void method()</code>	Die Methode wird temporär nicht ausgeführt; der Kommentar (unbedingt als Parameter übergeben) wird im Protokoll des Testlaufs ausgegeben
<b>@Test(expected = Exception.class)</b> <code>public void method()</code>	Test ist nur dann erfolgreich, wenn die Methode die geforderte Exception wirft

# Wozu @Ignore?

- Durch Umstrukturierung von Quellcode möglich, dass ein Testcode nicht mehr gültig → Testfall soll **nicht mehr** ausgeführt werden
- **Auskommentieren** ungünstig, da das **Refactoring** z.B. im Zuge einer *Umbenennung von Bezeichnern* sich **nicht** auf auskommentierte Bereiche auswirkt
- **Löschen** noch ungünstiger, da Testfall einen Stand repräsentiert (Dokumentation der Entwicklung) und zukünftig vielleicht wieder gültig sein wird

→ **besser**: Testfall als **@Ignore** annotieren

@Ignore

@Test

```
public void testMeineMethode() {...}
```

# Wie Exception testen?

```
public class KontoTest {  
    ...  
    @Test  
    public void testEinzahlenEinmal() {...}  
  
    @Test  
    public void testEinzahlenZweimal() {...}  
  
    @Test(expected = IllegalArgumentException.class)  
    public void testEinzahlenFehlerfall() {  
        ko.einzahlen(-3.5f)  
    }  
}
```

Test bestanden, wenn `ko.einzahlen(-3.5f)`  
eine `IllegalArgumentException` wirft

# Exception testen ab JUnit 5.0

```
public class KontoTest {  
    ...  
    @Test  
    public void testEinzahlenEinmal() {...}  
  
    @Test  
    public void testEinzahlenZweimal() {...}  
  
    @Test  
    public void testEinzahlenFehlerfall() {  
        assertThrows(IllegalArgumentException.class, ()  
        -> {ko.einzahlen(-3.5f);} );  
    }  
}
```

Test bestanden, wenn `ko.einzahlen(-3.5f)`  
eine `IllegalArgumentException` wirft

# Wie Laufzeit testen?

- Nach großem Refactoring möglich, dass Software (funktional läuft, aber) viel langsamer geworden ist →
- Laufzeitveränderungen mit Angabe eines *Timeouts* getestet:

```
@Test( timeout = 500 )  
public void testGeschwindigkeit() {  
    meineMethode();  
}
```

→ wenn Testmethode **nicht** innerhalb der Schranke von 500 Millisekunden ausgeführt, dann Test **durchgefallen**

# Failure vs. Error

- **Failure** = Versagen, **Fehlschlagen eines Testfalls** → deutet auf einen Fehler in der Implementierung der Anwendungsmethode
- **Error** = **Fehler im Testfall** (kommt zustande, wenn Exception bis in die Testmethode gelangt) → deutet auf einen Fehler in der Implementierung des Testfalls

# Error: Beispiel

```
@Test
public void testEinzahlenEinmal() {
    ko = null; //oder kein Objekt vorher erzeugt
    ko.einzahlen(10.5f);
    assertTrue(ko.getKontostand() == 10.5f);
}
```

# Error: Beispiel

```
@Test  
public void testEinzah  
    ko = null; //oder kein  
    ko.einzahlen(10.5f);  
    assertTrue(ko.getKon  
}
```

The screenshot shows the JUnit runner interface in an IDE. At the top, it says "Finished after 0,013 seconds". Below this, a summary bar displays "Runs: 5/5", "Errors: 1" (highlighted with an orange circle), and "Failures: 0". A red progress bar is shown below the summary. The test list includes:

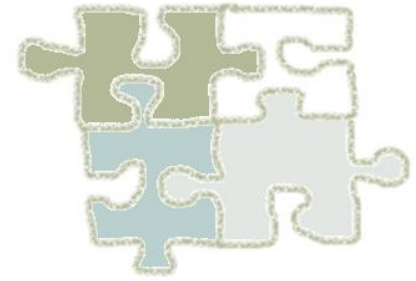
- KontoTest [Runner: JUnit 4] (0,001 s)
  - testNegativeEingabe (0,000 s) [Success]
  - testEinzahlenEinmal (0,000 s) [Failure]
  - testGetKontostand (0,000 s) [Success]
  - testEinzahlenBeimNegKontostand (0,001 s) [Success]
  - testEinzahlenZweimal (0,000 s) [Success]

At the bottom, the "Failure Trace" section shows the following error (highlighted with an orange box):

```
java.lang.NullPointerException  
at KontoTest.testEinzahlenEinmal(KontoTest.java:27)
```



# JUnit zusammengefasst



- Programmiersprache = Testsprache → JUnit – ein reines Java-Framework
- Anwendungscode und Testcode von einander getrennt: Testfälle in einer eigenen Klassenhierarchie erstellt (mit der Basisklasse: `junit.framework.TestCase`)
- Reihenfolge der Tests spielt keine Rolle → Vorarbeit und Nacharbeit bei einem Testfall in eine extra Methode ausgelagert (analog dazu: Vorarbeit und Nacharbeit bei einer Sammlung von Tests – Testsuite)
- Methoden in der Testklasse durch Annotationen oder vorgegebene Namen gekennzeichnet
- Tests aufgerufen, ausgeführt, Versagen gezählt, Ausnahmen abgefangen, Ergebnisse angezeigt

# Literatur

Johannes Link: *Softwaretests mit JUnit, Techniken der testgetriebenen Entwicklung*, 2.Auflage, Heidelberg: dpunkt.verlag, 2005

<http://www.tutego.de/blog/javainsel/2010/04/junit-4-tutorial-java-tests-mit-junit/>

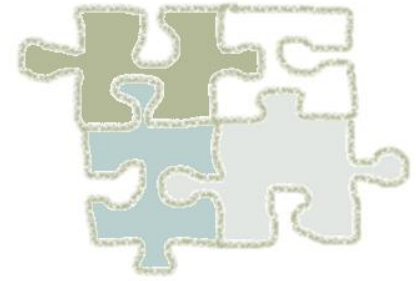
User Guide für JUnit 5:

<https://junit.org/junit5/docs/current/user-guide/>

# Links

- Agile in Practice: Test Driven Development  
<https://www.youtube.com/watch?v=uGaNkTahrIw>
- JUnit 5 Basics 12 - Test driven development with JUnit  
[https://www.youtube.com/watch?v=zFJdQYn9u\\_8](https://www.youtube.com/watch?v=zFJdQYn9u_8)
- **Exceptions testen:** <https://blog.oio.de/2019/12/11/junit-5-behandlung-von-exceptions/>

# Zusammenfassung



- Software-Qualitätsmerkmale: Funktionalität, Zuverlässigkeit, Effizienz, Benutzbarkeit, Änderbarkeit (Wartbarkeit), Übertragbarkeit
- Hohe Software-Qualität u.A. durch Testen erreichbar
- Fehler: je später entdeckt, desto teurer in der Behebung → Fehler so früh wie möglich entdecken → so früh wie möglich testen
- Test auf verschiedenen Ebenen: Komponentenebene (Unit Test), Modulebene (Integrationstest), Systemebene (Systemtest) und Anforderungsebene (Abnahmetest)
- Unit Test: SW-Bausteine testen: Vorbedingungen, Eingaben, Testszenario, Ausgaben, Zusatzinfos → Testergebnis
- Ansatz: Test-Driven Development (TDD): ZUERST Tests, DANN Funktion implementieren (Schritte)