



Programmierungsmethodik 1

Programmiertechnik

Testen

- 4.5.15
 - Typos korrigiert
 - Übungsaufgabe "Objektzähler" zugefügt

- Schreiben Sie eine Klasse Element.
- In der Klasse soll protokolliert werden, wenn ein neues Objekt erzeugt wurde.
- Geben Sie dann die Anzahl von bisher erzeugten Objekten auf der Konsole aus.

- Statische Objektvariablen
- Statische Methoden
- Aufzählungstypen

Ausblick für heute

- Wir haben Code geschrieben. Wie stellen wir nun sicher, dass er auch das macht, was er soll?

- Einführung: Fehler + Testen
- Testen mit JUnit
- Fehlertypen
- Platzhalterobjekte

Einführung: Fehler + Testen

- selbstverständliche Erwartung
 - Software muss fehlerfrei arbeiten
 - hohe Qualität aufweisen
- Hohe Qualität? \Leftrightarrow Anforderungen werden erfüllt!

- Validierung
 - Überprüfen, ob die Anforderungen erfüllt werden
 - Test: dynamische Analyse
 - Formale Verifikation: statische Analyse
- Beseitigen von Abweichungen ("Fehlern"), falls Anforderungen nicht erfüllt sind
 - z.B. Debugging

- Ziel: Auffinden von Fehlern durch systematisches Ausprobieren (Stichproben)
- Ausführen des Programms in ausgewählten Situationen = **Testfälle**
 - Erwartete Ergebnisse für Testfälle bekannt
 - Vergleich von tatsächlichen („Ist“) mit erwarteten Ergebnissen („Soll“)
 - Entscheidend: „gute“ Auswahl von Testfällen
- Erhöht das Vertrauen in die Korrektheit, liefert aber keinen endgültigen Nachweis
- Praxis: oft einzig gangbarer Weg

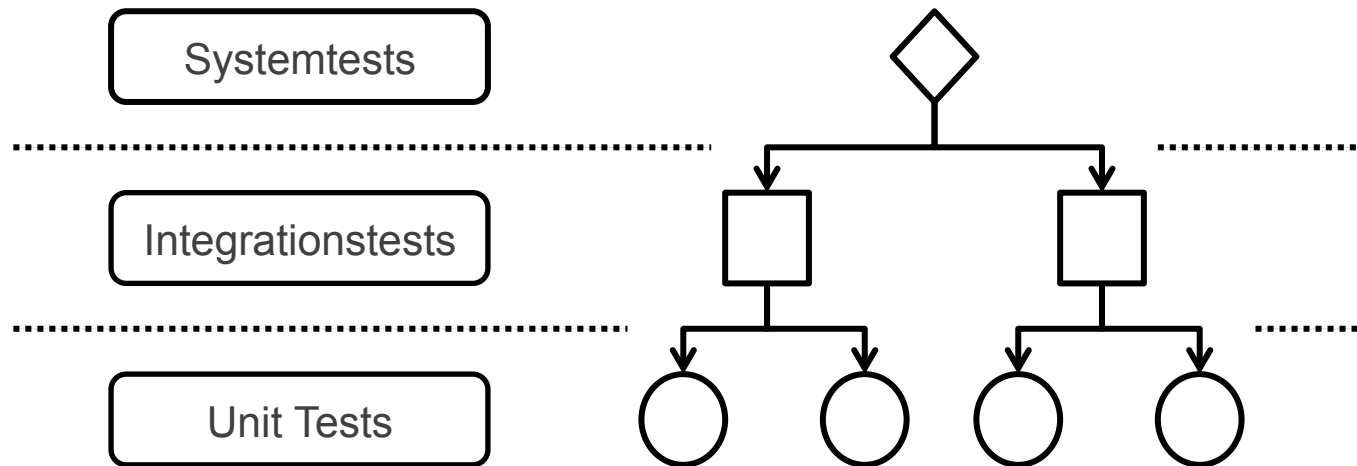
- funktionale Anforderungen
 - Korrektheit – rechnet die SW richtig
 - Angemessenheit - löst die SW das gegebene Problem
 - Interoperabilität
 - Datensicherheit
 - Ordnungsmäßigkeit (Standardkonformität)
 -
- nicht-funktionale Anforderungen – viele Kategorien
 - z.B. Performance (Speicher, Laufzeit)
 - z.B. Ausfallsicherheit
 - z.B. Bedienbarkeit
 - ...

- Auswahl der Testfälle alleine aufgrund der Schnittstellen-Spezifikation
- anwendbar ohne Quellcode
 - ohne Kenntnis der Implementierungssprache
- Konstruktion von Tests vor der Implementierung sind möglich
- anwendbar für unterschiedliche Implementierungen (z.B. konkurrierende Produkte)
- unabhängig von Änderungen im Quellcode (z.B. Programmversionen und -varianten)

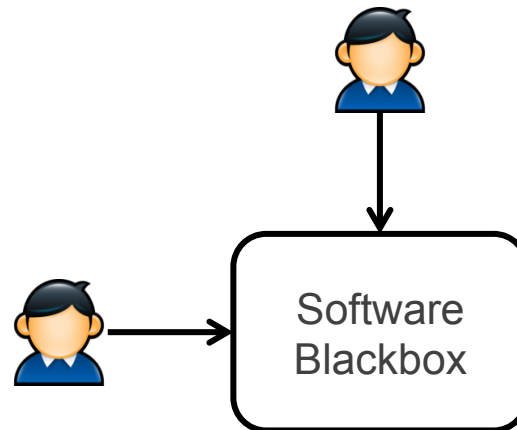
- Konstruktion von Testfällen an Hand des Quellcodes
- systematische Erstellung von Testfällen aus der (Programm-) Struktur
- Überprüfung aller Codeabschnitte
- Abdeckung des Prozentsatz des Quellcodes
 - z. B. 100% „Anweisungsüberdeckung“
- beide Vorgehensweisen auf allen Stufen möglich – aber unterschiedlich sinnvoll

- Gegeben ist folgendes Codefragment. Wie könnte
 - ein Blackbox-Test aussehen
 - ein Whitebox-Test aussehen

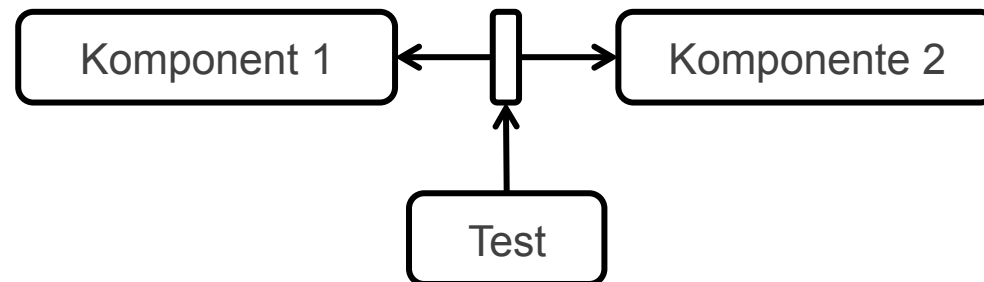
```
int berechneGgt(int zahl1, int zahl2) {  
    // Sicherstellung, dass beide Zahlen positiv sind  
    zahl1 = Math.abs(zahl1);  
    zahl2 = Math.abs(zahl2);  
  
    // Algorithmus von Euklid  
    int ergebnis = 0;  
    if (zahl1 == 0) {  
        ergebnis = zahl2;  
    } else {  
        while (zahl2 != 0) {  
            if (zahl1 > zahl2) {  
                zahl1 = zahl1 - zahl2;  
            } else {  
                zahl2 = zahl2 - zahl1;  
            }  
        }  
        ergebnis = zahl1;  
    }  
    return ergebnis;  
}
```



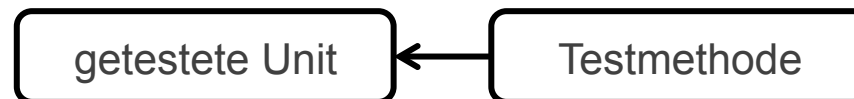
- Test der finalen Software
- nicht mehr am Code orientiert (Blackbox)
- Basis für den erfolgreichen Projektabschluss



- Kollaboration zwischen unabhängigen Komponenten
- Fokus auf der Schnittstelle
- Überprüfung ganzer Abläufe
- kann manuelle Anteile beinhalten

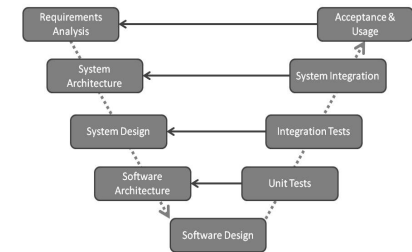


- ist ein Block Quellcode
- ruft anderen Block Quellcode auf
- prüft die Korrektheit von Annahmen
- eine Unit ist normalerweise eine Methode oder Klasse
- bei uns
 - Unit = Klasse
 - je ein Test pro (testbare) Methode

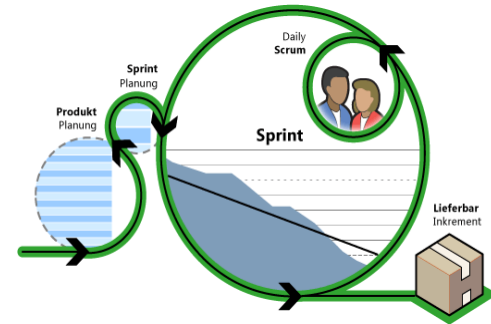


Zu welchem Zeitpunkt teste ich meine Software?

- traditioneller Ansatz
 - am Ende des Entwicklungsprozesses
 - nachdem alle Funktionalität implementiert ist
 - unmittelbar bevor die Software ausgeliefert wird
 - Durchführung durch Tester (nicht Programmierer)
- hat häufig zum Ergebnis:
 - Testphase wird als Projektpuffer verwendet
 - Tests entfallen weil andere Aufgaben "wichtiger" sind



- Moderner Ansatz
 - Tests parallel zum Entwicklungsprozess
 - oft in Form agiler Softwareentwicklung
- Vorgehen
 - kleine Änderung am Quellcode durchführen
 - alle Tests laufen lassen
 - alle Tests müssen positiv ausfallen
 - Sicherstellen, dass die Änderungen keine unvorhergesehen Probleme verursachen
- Ultimative Umsetzung
 - Testgetriebene Softwareentwicklung (Test-Driven-Development, TDD)



Testen mit JUnit

- Java-Framework zum Testen (Unit-Tests)
- nicht Teil des JDK aber meist mit installiert
 - ansonsten: *www.junit.org*
 - aktuelle Version 4.x
- unterstützt sehr gut testgetriebene Softwareentwicklung
- Verwendung
 - Paket *junit.jar* muss im CLASSPATH der Anwendung zu finden sein
 - unter Eclipse: JUnit muss im Buildpath eingetragen sein
 - *Projekt – Eigenschaften – Java Build Path*

- Anlegen einer Testklasse
 - üblicherweise eine Testklasse pro Klasse, die getestet wird
 - Beispiel: BruchTest für die Klasse Bruch
- Testklasse beinhaltet Testmethoden
 - erforderlicher Import:

```
import org.junit.Test;
```
 - Syntax von Testmethoden

```
@Test  
public void <Bezeichner>(){ ... }
```
 - Bezeichner der Methoden sollte mit test beginnen
 - Beispiel:

```
public void testIrgendeineFunktionalitaet()
```


- `@Test` ist eine Annotation
- Annotationen sind
 - Mittel zur Strukturierung von Quellcode
 - zur automatisierten Erzeugung von Programmcode
- werden beispielsweise eingesetzt in Java, C# und VB.NET
- Syntax in Java
 - `@`-Zeichen gefolgt vom Namen der Annotation
- Die `@Test`-Annotation wird von der JUnit-Bibliothek verarbeitet

- an der Annotation `@Test` wird erkannt, dass es sich um einen Test handelt
- die Annotation muss vor jeder Testmethode stehen
- in der Testmethode wird ein Testergebnis mit einem erwarteten Ergebnis verglichen
- Vergleich findet mit einem Assert statt

- Aussage oder Behauptung über den Zustand eines Programms
- Ziel: Erkennen von
 - logischen Fehlern im Programm
 - Defekten in der umgebenden Hard- oder Software
- im Fehlerfall: kontrolliertes Beenden des Programms
- beim Unit-Test: Abbruch des Testfalls (Fehlschlag)

- Importieren der Assert-Funktionalität

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

- Erinnerung: statischer Import = Zugriff auf statische Member ohne Angabe des Klassennamens

- einfacher Testfall mit Assertion:

```
assertEquals(String nachricht, Object erwartet, Object tatsaechlich)
```

- damit assertEquals funktionieren kann, muss equals() für erwartet und tatsaechlich implementiert sein (kommt später)

- es gibt sehr viele verschiedene Assertions in JUnit

- assertEquals(String nachricht, double erwartet, double tatsaechlich, double delta)
- assertTrue(String nachricht, boolean bedingung)
- ...

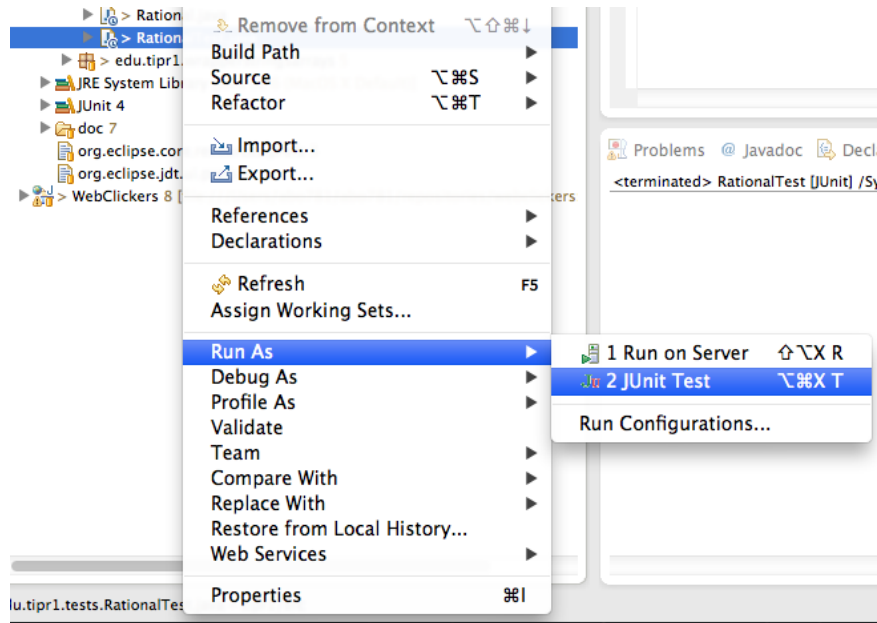
- Beispiel
 - zu testender Code in der Klasse **Bruch**

```
public Bruch mult(double faktor) {  
    return new Bruch(zaehler * faktor,  
        nenner * faktor);  
}
```

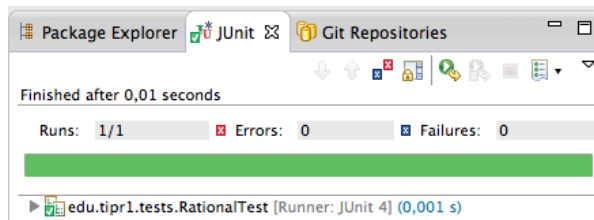
- Testmethode in der Klasse **BruchTest**

```
@Test  
public void testMult() {  
    Bruch bruch1 = new Bruch(1, 3);  
    Bruch ergebnis = bruch1.mult(2);  
    int zaehlerErwartet = 2;  
    assertEquals("Fehler bei mult()", ergebnis.getZaehler(),  
        zaehlerErwartet);  
}
```

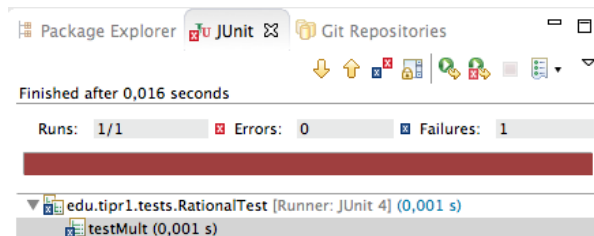
– Ausführen des Tests



- Auswertung
 - neues Fenster neben dem Package Explorer
 - Erfolg



- Fehlschlag



- Schreiben Sie eine JUnit-Testklasse BruchTest in der Sie die Bruch-Methode
`public static int berechneGgt(int zahl1, int zahl2)`
testen.

Fehlertypen

- Vorkommen verschiedener Fehlertypen
 - Syntaxfehler, statische Semantikfehler
 - dynamische Semantikfehler
 - Logik-Fehler

- ... werden vom Compiler bei der Analyse des Sourcecodes entdeckt
- Ort des Fehlers wird i.d.R. recht genau ausgewiesen
- Korrektur kann im Rahmen der programmiersprachlichen Ausdrucksmittel leicht erfolgen

- Beispiel Syntaxfehler (unzulässige Konstrukte):

```
public class {} // <identifizier> expected
```

- Beispiel statische Semantik (unzulässige Kombination isoliert korrekter Konstrukte):

```
public class Test {  
    int x;  
    x = true; // incompatible types; found: boolean required: int  
}
```

- nicht behandelte Laufzeitfehler, äußern sich in Exceptions
→ das Programm „stürzt ab“
- Beispiel: Euklid-Algorithmus des GGT mit $n = 0$
Exception in thread "main" java.lang.ArithmeticException: / by zero
- Keine Werkzeuge zum automatischen, erschöpfenden Erkennen der Fehlersituationen "von außen" vorhanden
- Lokalisieren des Fehlers über
→ Stacktrace
- Behandlung des Fehlers ggf. durch das Programm selbst
→ Umgang mit Exceptions

- Programm stürzt nicht ab, liefert aber falsche Ergebnisse
- Können vom Compiler nicht entdeckt werden
- Beispiel:
 - falscher relationaler Operator $<$ statt \leq bei **for**-Schleife
- Lokalisieren des Fehlers oft schwierig
- heikel:
 - Endlosschleifen (Programm „hängt“, liefert überhaupt kein Ergebnis),
 - unregelmäßig auftretende Fehler (race conditions in parallel laufenden Programmteilen)
 - nicht wiederholbare Fehler (Datei, Datenbank, Festplatte gelöscht)

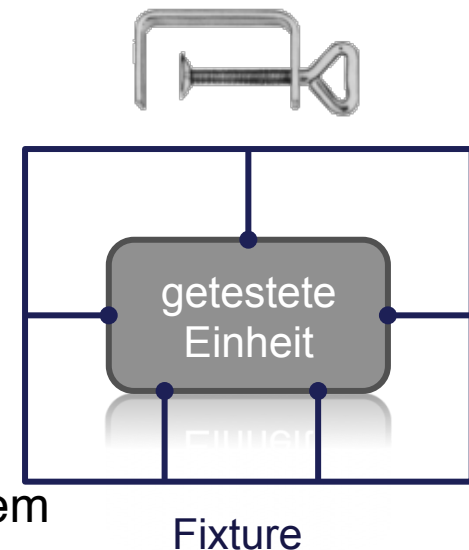
- Syntaxfehler, statische Semantikfehler
 - Compiler hilft
 - auch hilfreich: Code-Konventionen
 - dynamische Semantikfehler
 - Logik-Fehler
- }
- Testen
 - guter Softwareentwurf
 - übersichtlicher Quellcode → Code-Konventionen

- In jeder der folgenden Anweisungen steckt ein Fehler. Nennen Sie den zugehörigen Fehlertyp

```
int y = 23.42;  
for ( int i = 0; i < 5; i--){ ... }  
int z = 3 / ( 4 - 4);  
int x = 23
```

Platzhalterobjekte

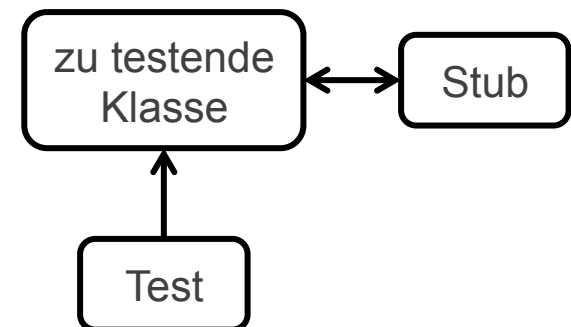
- Fixture
 - konsistenter Test von Software
 - fester, unveränderlicher Zustand
 - bekannte Umgebung
 - macht den Test wiederholbar
 - auch Kontext genannt
- Beispiele
 - Datenbank mit bekannten Daten laden
 - Löschen der Festplatte, frisches Betriebssystem
 - Kopieren von bekannten Dateien
 - Vorbereitung der Eingabedaten
 - Mocks und Stubs (kommt später)



- Auflösen von Abhängigkeiten
- Lösung: Platzhalterobjekte, die Abhängigkeiten ersetzen
 - > "dumme" Objekte
 - > werden nicht getestet
 - > testen nicht selber
 - > machen einfache, nachvollziehbare Dinge
 - > beinhalten keine Logik

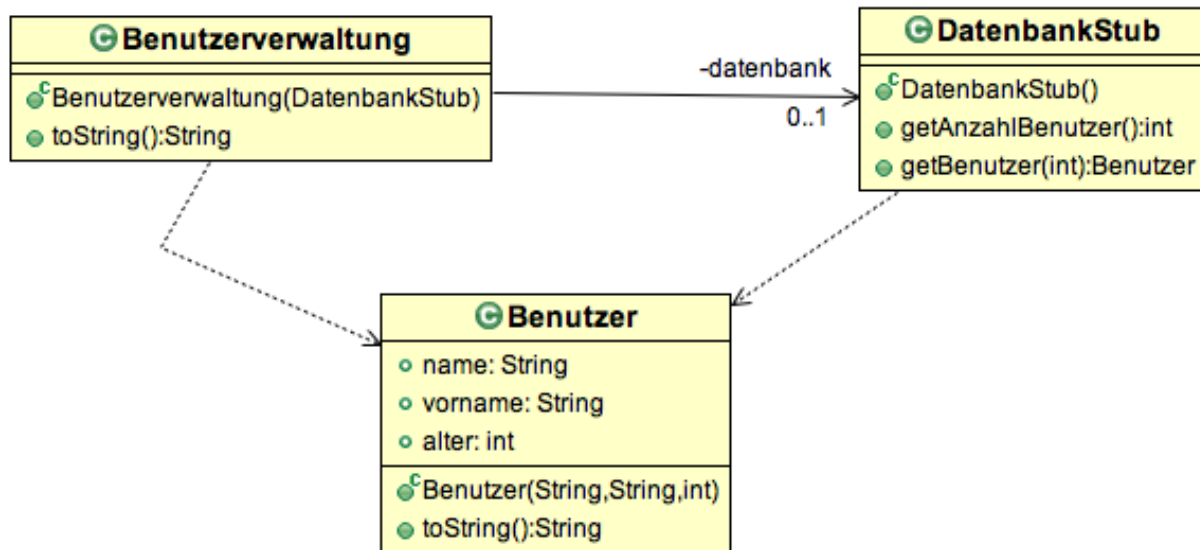


- Definition
 - > kontrollierbarer Ersatz für eine Abhängigkeit
 - > Testen des Quellcodes, ohne direkt mit der Abhängigkeit zu interagieren
- Zustandsbasiertes Testen
 - > auch: state-based testing, state verification
 - > testet, ob eine Methode korrekt arbeitet
 - > testet den Zustand des zu testenden Systems
 - > Überprüfung, nachdem Methode ausgeführt wurde



- eine einfach Benutzerverwaltung
- Benutzerinformationen (Name, Vorname, Alter) liegen in einer Datenbank
- Benutzerverwaltung erzeugt Text aus allen Benutzern
 - gibt Text auf Konsole aus (Methode toString)
- Testen: Abhängigkeit von Datenbank
 - Lösung: Datenbankzugriff durch Platzhalterobjekt ersetzen

– Klassendiagramm



- DatenbankStub
 - beinhaltet einige Dummy-Datensätze
 - hier 3 Testbenutzer
 - Vorteil
 - keine Änderung der Fixture
 - Inhalt immer bekannt

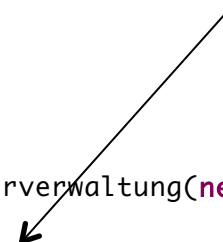
– Testklasse

```
import org.junit.Test;

/**
 * Testklasse für Benutzerverwaltung.
 */
public class BenutzerverwaltungTest {

    @Test
    public void testGetAnzahl() {
        Benutzerverwaltung verwaltung = new Benutzerverwaltung(new
        DatenbankStub());
        Benutzer scholl = new Benutzer("Scholl", "Mehmet", 42);
        Benutzer haessler = new Benutzer("Häßler", "Ikke", 51);
        Benutzer walter = new Benutzer("Walter", "Fritz", 88);
        String erwarteteBeschreibung =
            "Anzahl Benutzer: 3\n" + scholl.toString() +
            haessler.toString()
            + walter.toString();
        assertEquals("Beschreibung passt nicht", erwarteteBeschreibung,
            verwaltung.toString());
    }
}
```

Test "weiß", dass genau
diese Benutzer im
Platzhalter sind



- je nach Literatur:
 - verschiedene Typen von Platzhalterobjekten: Stubs, Mocks, ...

- Gegeben ist eine Klasse WebAusgabe, die einen Text auf einer Webseite ausgibt.
- Entwickeln Sie eine geeignete Platzhalterklasse und einen Test

```
/**
 * Gibt einen Text auf einer Webseite aus.
 */
public class WebAusgabe {

    /**
     * Referenz auf die Webseite.
     */
    private Webseite webseite;

    /**
     * Konstruktor.
     */
    public WebAusgabe(Webseite webseite) {
        this.webseite = webseite;
    }

    /**
     * Gibt den Text an die Webseite weiter.
     */
    public void gibAus(String text) {
        webseite.stelleDar(text);
    }
}
```

- Einführung: Fehler + Testen
- Testen mit JUnit
- Fehlertypen
- Platzhalterobjekte