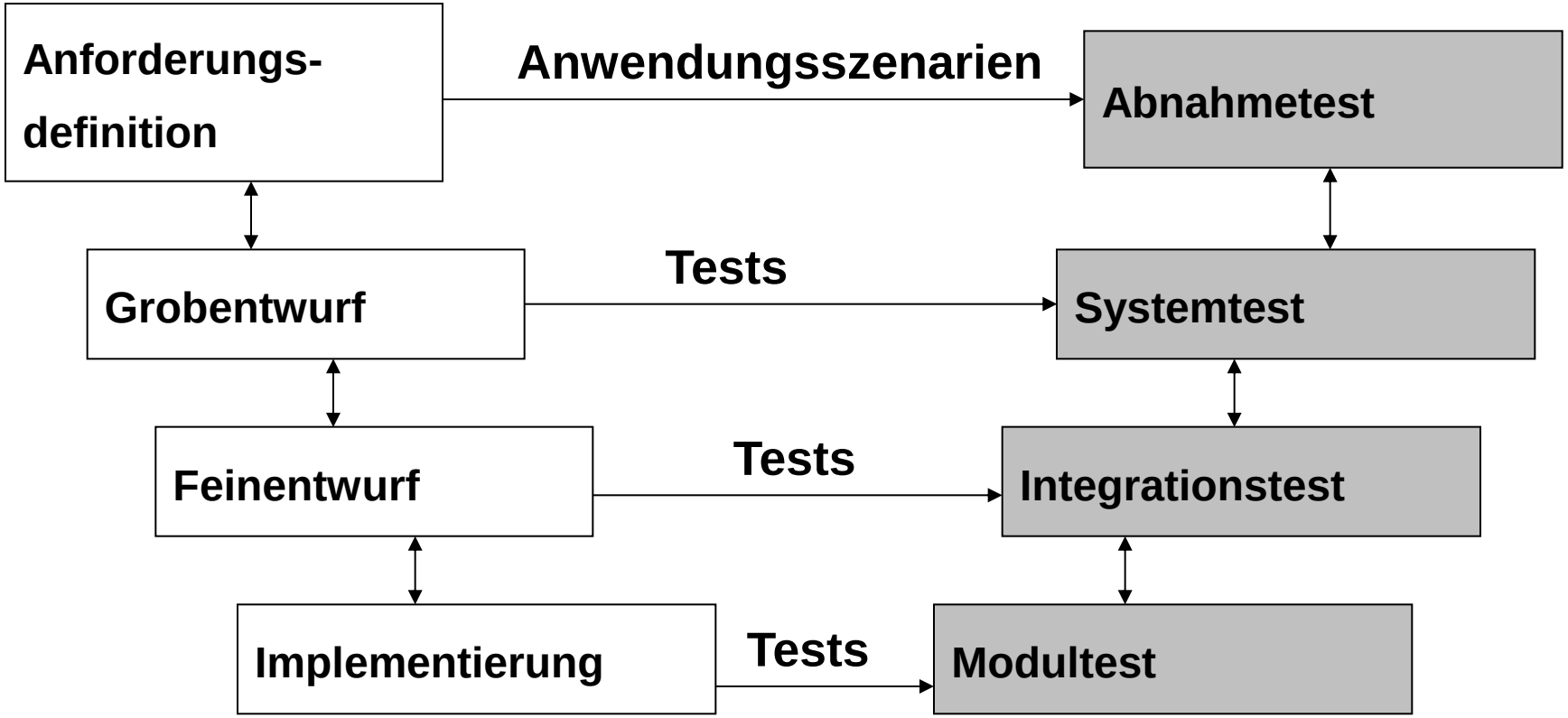
Softwaretests

- verschiedene Testformen
- Testen im V-Modell:



Vorurteile gegen Testen

- **keine Zeit zum Testen:**

  großer Zeitdruck

  -> weniger Tests

  -> instabiler Code

  -> mehr Fehlermeldungen vom Kunden

  -> mehr Debugging-Zeit

  -> noch größerer Zeitdruck

- **Testen langweilig und stupide**
- **Mein Code ist praktisch fehlerfrei, gut genug**
- **Testabteilung testet**

Literatur

- Uwe Vigenschow: *Objektorientiertes Testen und Testautomatisierung in der Praxis*; dpunkt-Verlag.
- Johannes Link: *Softwaretests mit JUnit –Techniken der testgetriebenen Entwicklung –* ; dpunkt-Verlag.
- Andreas Spillner & Tilo Linz: *Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester (Foundation Level)*; dpunkt-Verlag.
- Robert V. Binder: *Testing Object-Oriented Systems – Models, Patterns, and Tools*; Addison-Wesley.
- Erich Gamma & Kent Beck: *JUnit Framework*; www.junit.org.
- Frank Westphal: *Testgetriebene Entwicklung mit JUnit & FIT*; dpunkt-Verlag

manuelle Tests

- **Testen von** BenutzerVerwaltungAdmin **durch main-Progr.:**

```
BenutzerVerwaltungAdmin adm =
                        new BenutzerVerwaltungAdmin();
 adm.dbInitialisieren();                    //mit und ohne
 Benutzer ben = new Benutzer("Heinz","Pwd");
 System.out.println(adm.benutzerOk(ben));
 adm.benutzerEintragen(ben) ;
 System.out.println(adm.benutzerOk(ben));
 adm.benutzerEintragen(ben);
 adm.benutzerLöschen(ben);
 System.out.println(adm.benutzerOk(ben));
 adm.benutzerLöschen(ben);
```

- **Auswertung durch scharfes Hinsehen:**
  - Vergleich: erwartetes und erhaltenes Ergebnis

Probleme bei manuellen Tests

- **häufige Ausführung der Tests**
  - zeitaufwändig
  - ermüdend
    -> Fehler übersehen
  - Tests vergessen

- **häufig am Ende keine Zeit mehr, noch Tests**
  - zu überlegen und
  - Durchzuführen

->Testautomatisierung

Testautomatisierung

- **Tests & Überprüfung**
  - programmierbar
  - automatisch ausführbar

- **lohnt sich ziemlich schnell, wenn mehrmals ausgeführt**

- **möglichst Tests vor oder während der Entwicklung programmieren (test-getriebene Entwicklung) ->**
  - sichere Programme
  - einfaches Design
  - effektive Schnittstellen
  - bessere Spezifikation

- **zusätzlicher Programmieraufwand der Tests führt meist nicht zu zusätzlichem Gesamtaufwand**

Was automatisiert getestet?

- **Erstellung von Test-Treibern**
- **Testen einzelner Methoden**
  - Vor- und Nachbedingungen der Methoden
  - welche Eingaben, welche Ausgaben
  - Design by Contract
- **Testen des Protokolls einer Klasse (Kettentest)**
  - typische Verwendungsszenarien von Instanzen der Klasse
- **Testen von Interaktionen**
  - zwischen mehreren Objekten

Anforderungen an Testframework

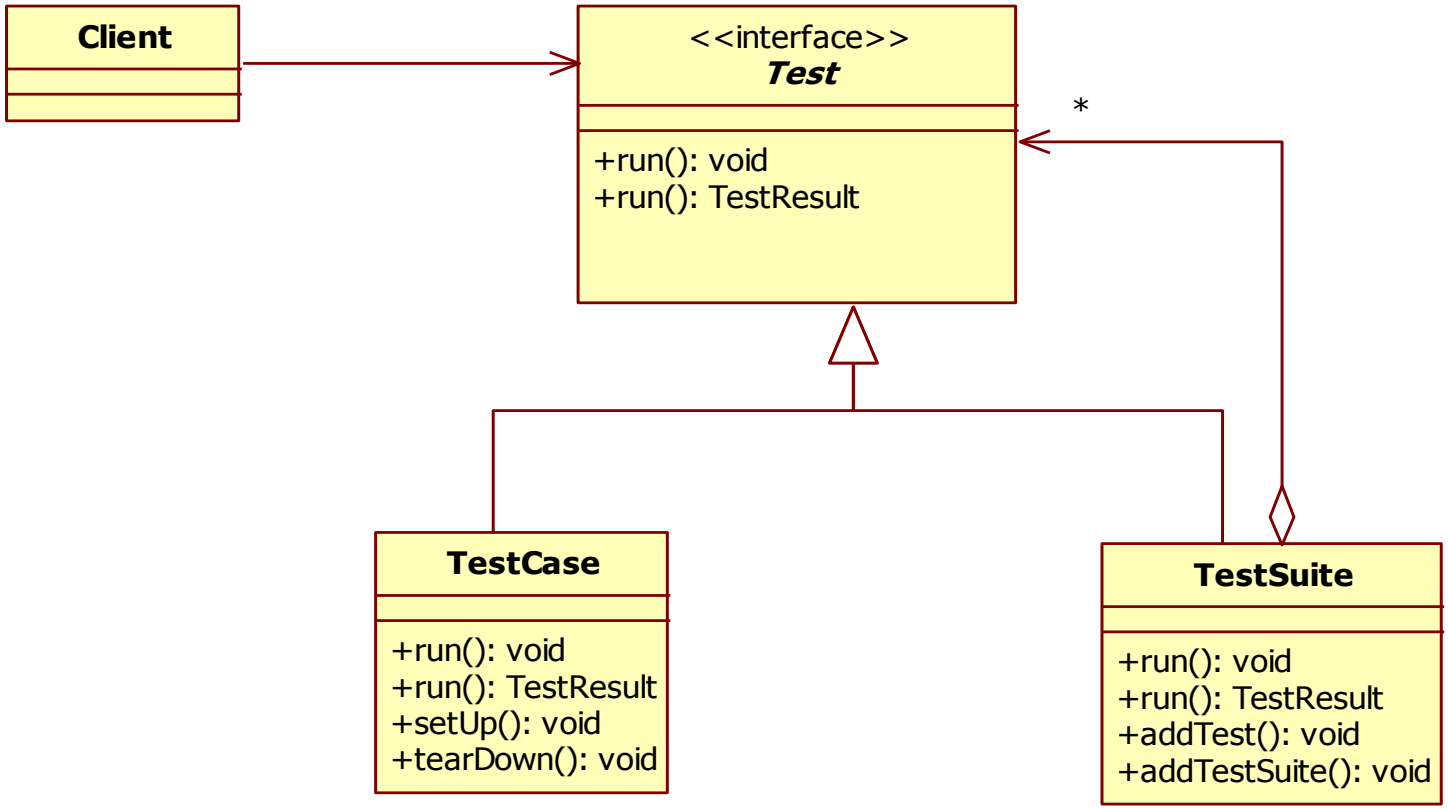
- **Testsprache == Programmiersprache**
  - gibt auch Skriptsprachen zur Erstellung von Testtreibern
  - Verwendung zweier Sprachen: geistiges Umschalten
- **Anwendungscode und Testcode trennbar**
  - wichtig für Auslieferung des Produkts
  - unabhängige Strukturierung des Testcodes
- **Ausführung der Testfälle voneinander unabhängig**
  - sonst Fehlalarm an anderer Stelle
- **Zusammenfassung in Testsuiten**
  - oft viele Tests zusammengehörig
- **Protokollierung der Tests**
  - auf einen Blick, Erfolg oder Misserfolg erkennbar

JUnit

- **Open-Source-Projekt**
- **von Kent Beck und Erich Gamma entwickelt**
- **Schreiben von Tests und Code getrennt**
- **Tests in selber Sprache geschrieben wie Code**
- **Tests laufen automatisch ab**
- **Tests nehmen Verifikation der Testergebnisse vor**
- **automatische Protokollierung der Tests**
- **Tests verschiedener Autoren einfach kombinierbar**
- **gemäß Composite-Pattern strukturiert:**
  - Basisfall: **TestCase**
  - Induktion: **TestSuite**

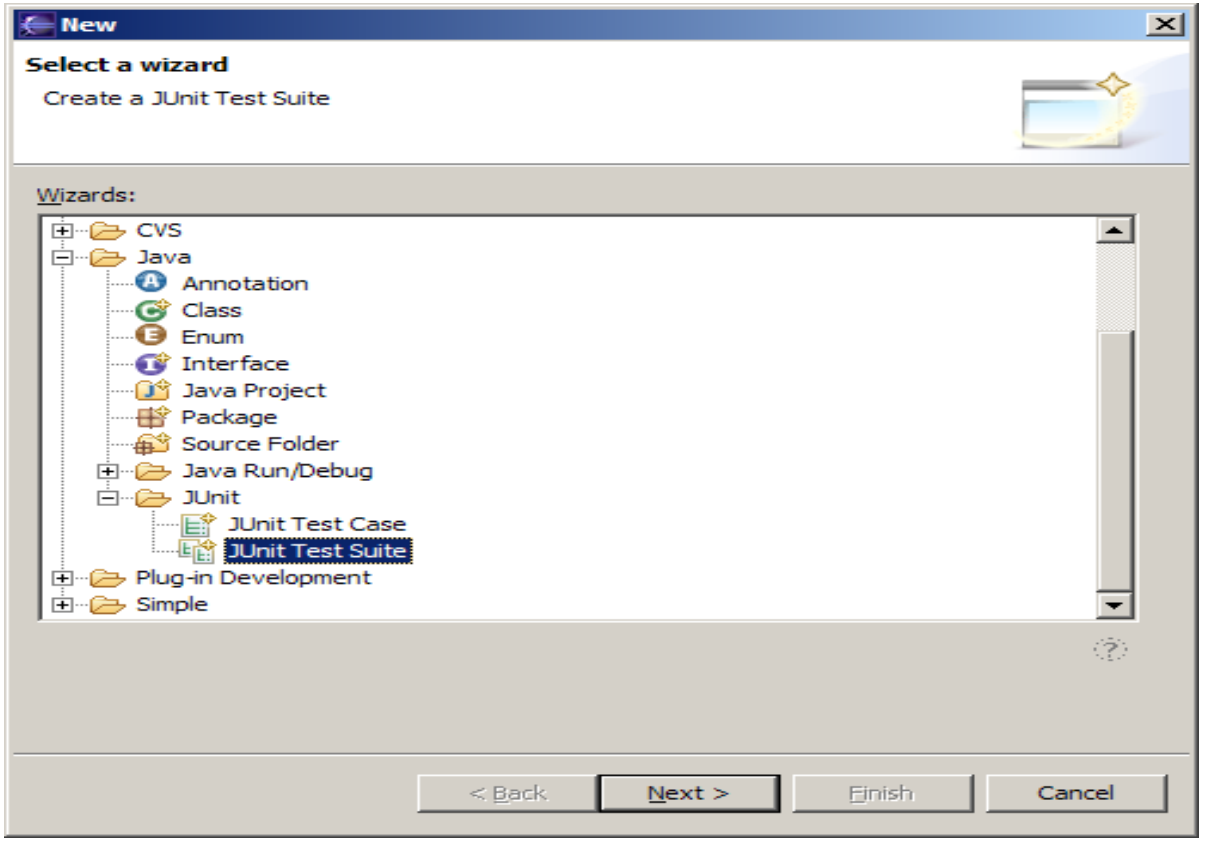
JUnit-Framework

- **in TestCase: einzelner Test**
- **in TestSuite: Sammlung von Tests**



Anwendung von JUnit

- **läuft innerhalb von TestRunner**
- **über GUI:** junit.swingui.TestRunner
- **über Konsole**: junit.text.TestRunner
- **über Eclipse:**



Testklasse zu `Benutzer` erstellen

```
import junit.framework.TestCase;
public class BenutzerTest extends TestCase {
  private Benutzer ben1;
  protected void setUp() throws Exception {
    super.setUp();
  }
  protected void tearDown() throws Exception {
    super.tearDown();
  }
// hier Testmethoden hin, siehe nächste Folie
  public static void main(String[] args) {
    junit.textui.TestRunner.run(BenutzerTest.class);
  }
}
```

Testmethoden zu BenutzerTest

```
public void testBenutzerStringString() {     //Konstruktortest
   Benutzer ben1 = new Benutzer("uid","pwd");//-> setUp()
   assertEquals("uid",ben1.userId);
   assertEquals("pwd", String.valueOf(ben1.passWort));
}

public void testEqualsBenutzer() {                  //Equalstest
   Benutzer ben1 = new Benutzer("uid","pwd");    //-> setUp()
   Benutzer ben2 = new Benutzer("uid","pwd");
   assertEquals(ben1,ben2);
   assertNotSame(ben1,ben2);
   ben1 = ben2;
   assertSame(ben1,ben2);
   Benutzer ben3 = new Benutzer("ui","pw");
   assertFalse(ben1.equals(ben3));
}

public void testToStringBenutzer() {          // noch machen
```

`assert`-Methoden

- **Klasse Assert stellt Methoden zur Verfügung:**
  - `assertEquals(expected: Typ, actual: Typ)`, für
    - alle primitiven Typen
    - für Object: ruft equals auf
  - `assertEquals(expected, actual, delta: double)`
  - `assertNotNull(obj: Object)`
  - `assertNull(obj: Object)`
  - `assertSame(expected: Object, actual: Object)`
    - Test auf gleiche Referenz
  - `assertTrue(condition: boolean)`
  - `assertFalse(condition: boolean)`
  - `fail()`
  - ....

`setUp()`- & `tearDown()`-Methoden

- **mit setUp()**
  - Initialisierung vor jeder Testmethode:

```
protected void setUp() throws Exception {
  super.setUp();                          // am Anfang
  Benutzer ben1 = new Benutzer("uid","pwd");
}
```

  - für Variablen private-Attribut in Testklasse anlegen

- **mit tearDown()**
  - Freigeben von Resourcen nach jeder Testmethode:

```
protected void tearDown() throws Exception {
  super.tearDown();                  // am Ende
}
```

TestSuite erstellen

```
import junit.framework.Test;
import junit.framework.TestSuite;
public class AllTests {
  public static Test suite() {
    TestSuite suite = new TestSuite("Test prak3");
    suite.addTestSuite(BenutzerTest.class);

suite.addTestSuite(BenutzerVerwaltungAdminTest.class);
    // nach Praktikum 3
    return suite;
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(AllTests.suite());
  }
```

Test von Exceptions

- **möchten testen, ob Exceptions richtig geworfen werden:**
- **in equals von Benutzer:**

```
if (ben == null)
  throw new NullPointerException("Par ist null!");
```

- **in BenutzerTest:**

```
public void testNullPointerException() {
  try { Benutzer ben = null;
        this.ben.equals(ben1);
        fail("ben must not be null");
  } catch (NullPointerException e) {
    e.printStackTrace();}
}
```

Änderungen in jUnit 4.0

- Testklasse nicht mehr von `TestCase` abgeleitet
- dafür Package `org.junit` importiert
- Klasse muss öffentlichen Default-Konstruktor haben
- Test-Methoden müssen nicht mehr mit "test" beginnen
- dafür mit der Annotation `@Test` markiert sein
  Bsp.: `@Test public void testadding() {...`
- Methode muss Rückgabewert `void` und keine Parameter haben
- `@Before` und `@After` markieren Setup- bzw. Teardown-Aufgaben, die für jeden Testfall wiederholt werden (müssen öffentlich sein)
- `@BeforeClass` und `@AfterClass` markieren Setup- oder Teardown Aufgaben, nur einmal pro Testklasse ausgeführt werden. Die Methoden müssen statisch sein.
- `@Ignore` kennzeichnet temporär nicht auszuführende Testfälle.

Wie testen?

- **Fehlerfreiheit durch Testen nicht nachweisbar!**
- **kann beliebig viel testen**
- **häufig eher zu wenig getestet**
- **Was ist richtiges Testmaß:**
  - Finden möglichst vieler Bugs mit überschaubarem Aufwand
- **akzeptables Fehlerniveau hängt von Systemart ab**
- **Testaufwand wächst exponentiell zum Nutzen**
- **Regel: "Test everything that could possibly break."**
  - sagt nichts
  - bekommt mit der Zeit heraus, welche Fehler man macht
  - finde eigenes optimales Testniveau

Was testen?

- **für jede nicht triviale Klasse mindestens eine Testklasse**
  - triviale Klassen: Datenklassen, Exceptionklassen
- **setter / getter normalerweise nicht testen**
- **nicht öffentliche Methoden normalerweise nicht testen**
  - werden bei Verwendung der öffentlichen mit getestet
- **Tests sollten normalerweise nicht auf Innereien zugreifen**
- **möglichst keine komplexen Integrationstests**
  - bei Änderungen schwierig nachzuvollziehen
  - finden jedoch oft nicht erwartete Fehler
- **keine Tests für Tests**
  - getestete Klassen dienen automatisch als Tests der Tests

Testabdeckung

- **Wieviel von X wird durch meine Tests abgedeckt?**

- **X steht für**
  - spezifikationsbasierte Abdeckung (Blackbox Test)
    - überprüfe für Eingaben, ob richtige Ausgaben geliefert
    - Problem: Eingabenmenge oft unendlich
  - codebasierte Abdeckung (Whitebox Test)
    - überprüfe, ob alle Pfade durchlaufen werden
    - häufig automatisch durchführbar
    - sagt nichts über korrekte Funktionalität & Fehlerfreiheit aus
    - deutet auf fehlerträchtige Teile hin

spezifikationsbasierte Tests

- **Testen des Ein-/Ausgabe-Verhaltens für alle nichttrivalen Methoden**

- **finde für möglicherweise unendlich viele Eingaben endlich viele gute Repräsentanten**

  Äquivalenzklassenbildung: Eingaben mit gleicher Ausgabe
  Randfälle besonders wichtig:
  leere Liste & volle Liste (Induktionsanfänge)
  fastleere Liste & fastvolle Liste
  Mittlere Fälle zusammenfassen (Induktionsschritte)
  sinnvolle Repräsentanten

- **immer gut- und bösartige Eingaben testen**
  - Löschen für vorhandenen und nicht vorhandenen Benutzer