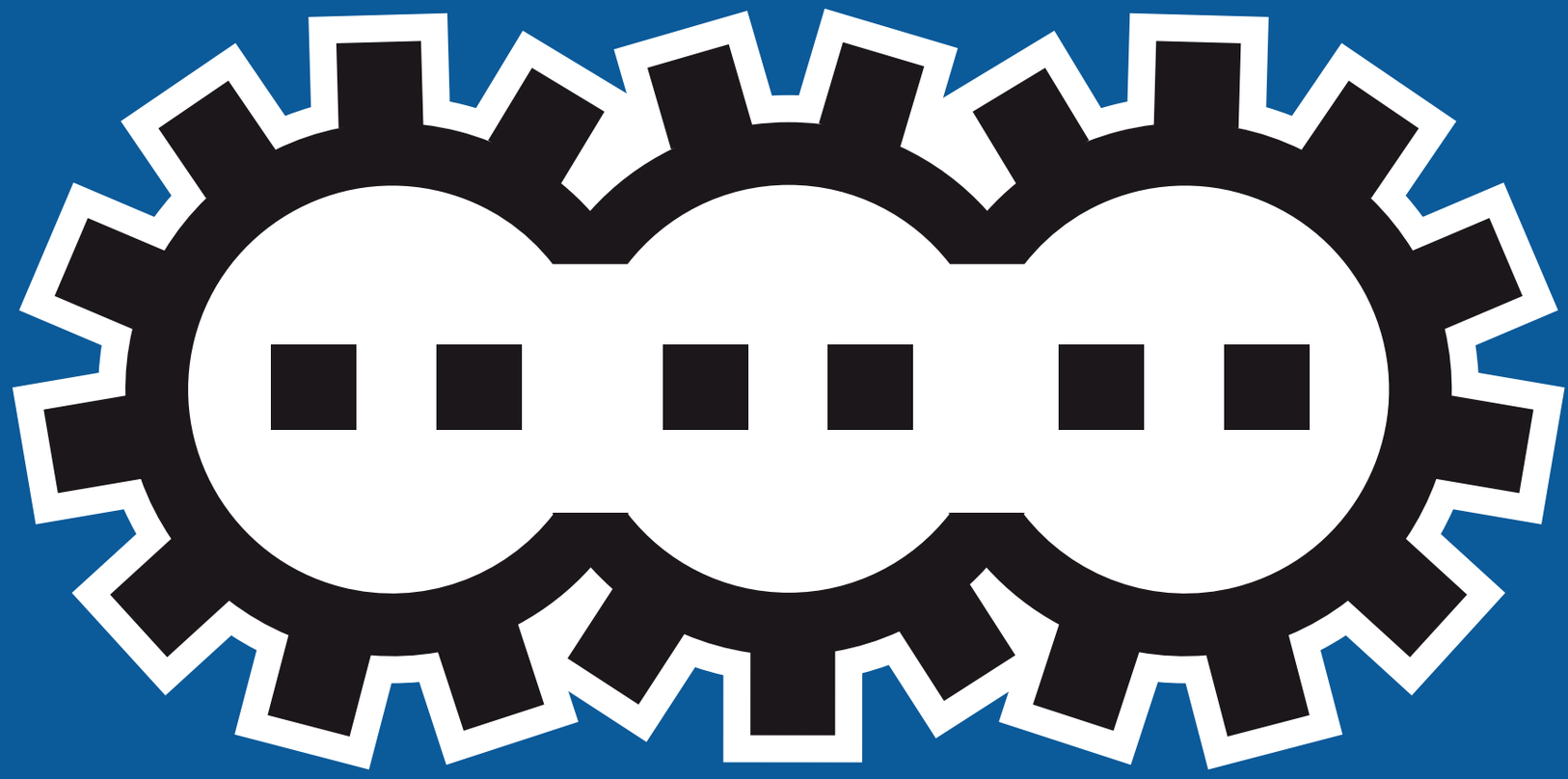
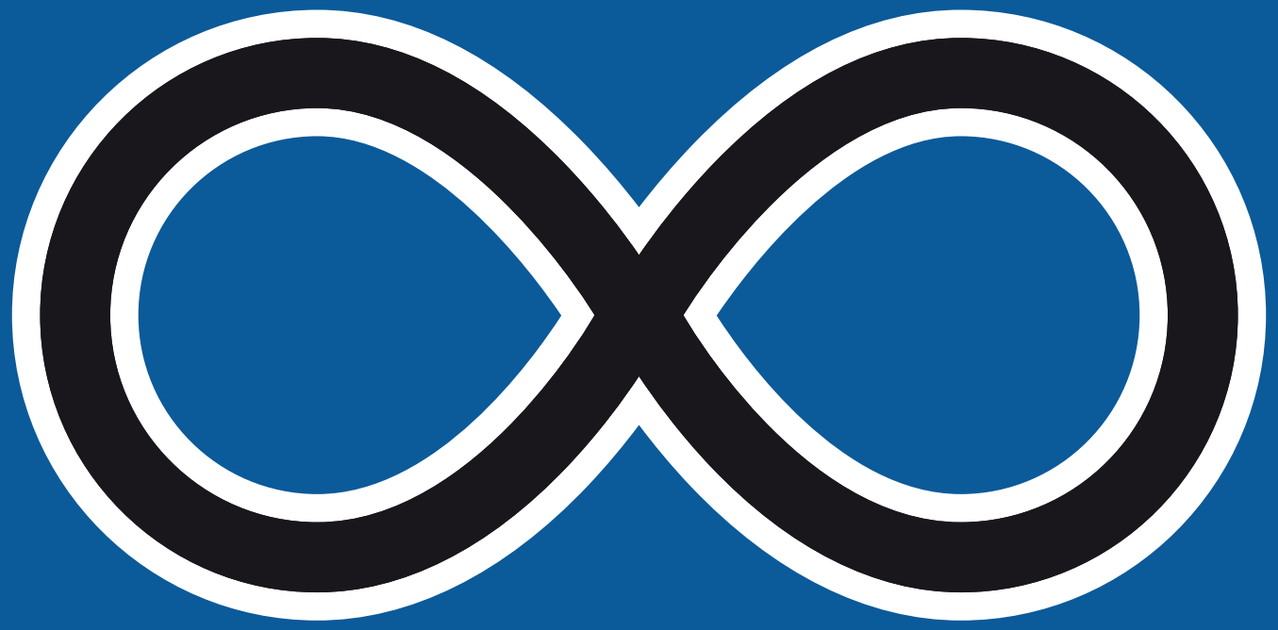


SOFTWAREENTWICKLUNG

IM TEAM MIT OPEN-SOURCE-WERKZEUGEN

07 - Integrationstests

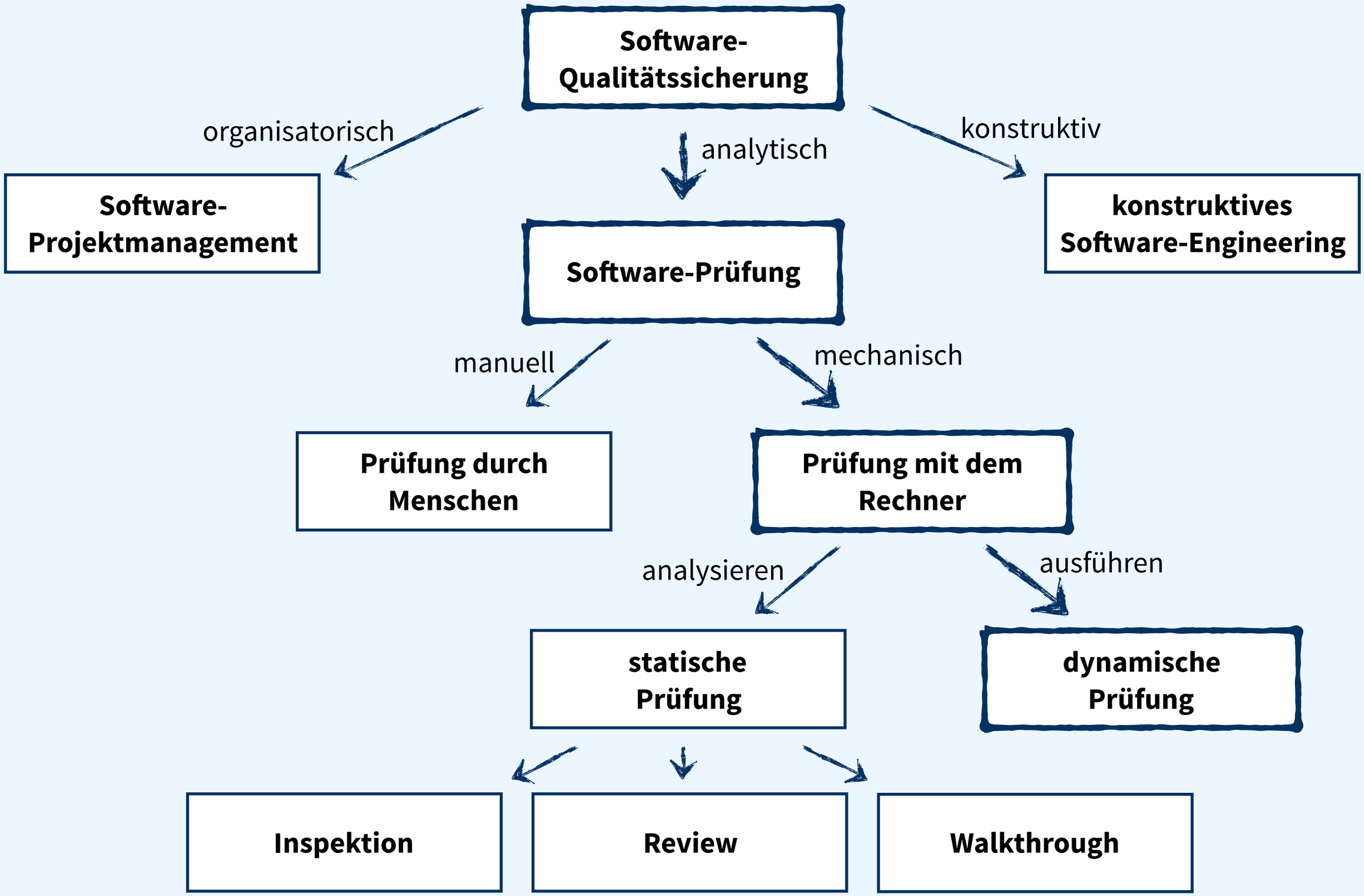
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen



Wiederholung

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Software-Qualitätssicherung



Quelle: J. Ludewig, H. Lichter. Software Engineering, 2010

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

JUnit 4

- JUnit 4 modernisiert und vereinfacht die Testfallerstellung
  - Testklasse muss nicht von der Basisklasse `TestCase` abgeleitet werden (`assert`-Methoden aus globalem Import)
  - verwendet Annotationen, um Testfälle auszuzeichnen
  - Testmethoden benötigen kein Präfix `test` (trotzdem sinnvoll)

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


public class StringTest{
  @Test
  public void testStringtoUpperCase() {
    final String eingabe = "Das ist ein Testfall.";
    final String sollWert = "DAS IST EIN TESTFALL.";
    assertEquals(sollWert, eingabe.toUpperCase());
  }
}
```


HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

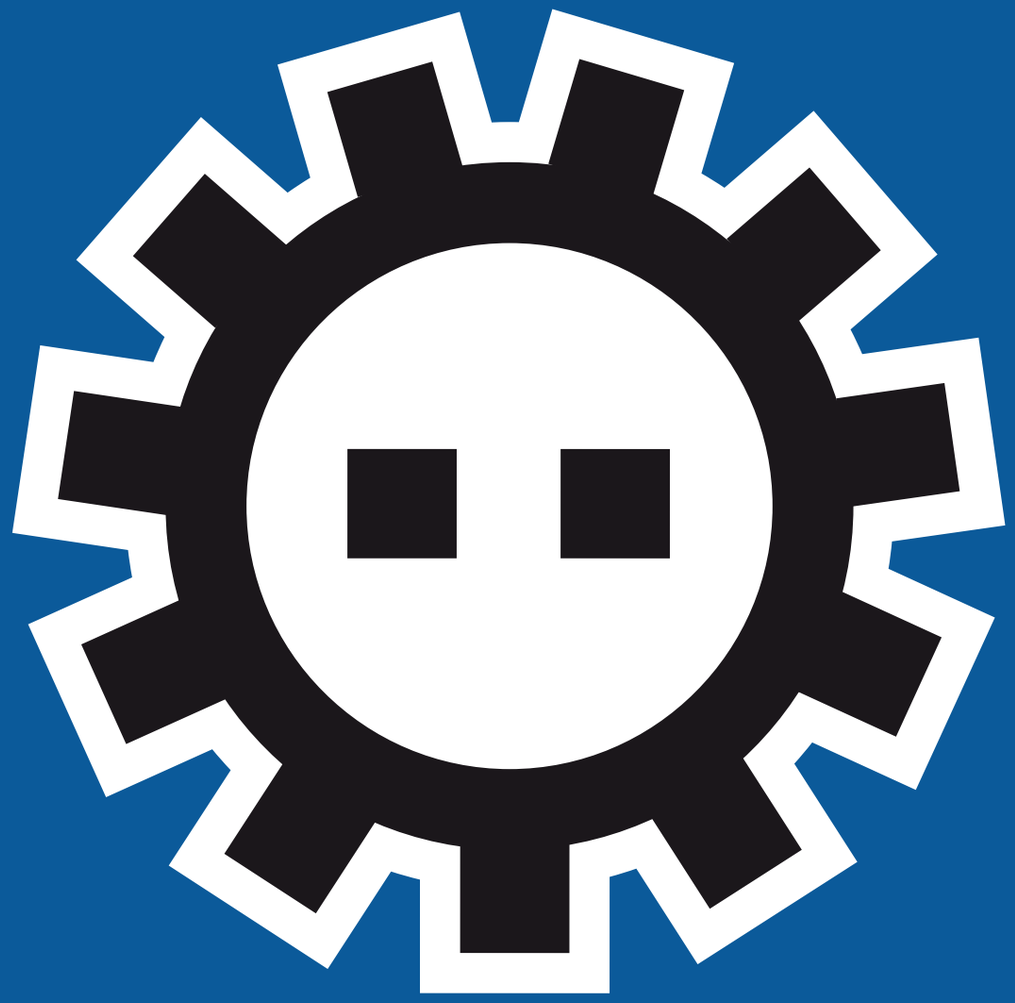
Verzeichnis für Testklassen

- Kompilierte Testklassen (und Testframeworks) sollten später nicht ausgeliefert werden
  - separate Verzeichnisse für Anwendungs- und Testcode
  - Anwendungs- und Testcode sollten aber im gleichen Package liegen

- Maven unterstützt dies durch eine vorgegebene Verzeichnisstruktur:

projekt/	Hauptverzeichnis des Projekts
+-**src/**	Alle Quelldateien
| +-**main/**	Quelldateien für die Anwendung
| | +-**java/**	Java Quellcode
| | | +-Calculator.java	
| +-**test/**	Quelldateien für den Test der Anwendung
| | +-**java/**	Java Quellcode der Testklassen
| | +-CalculatorTest.java	
…	

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen



Motivation

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Integrationstests

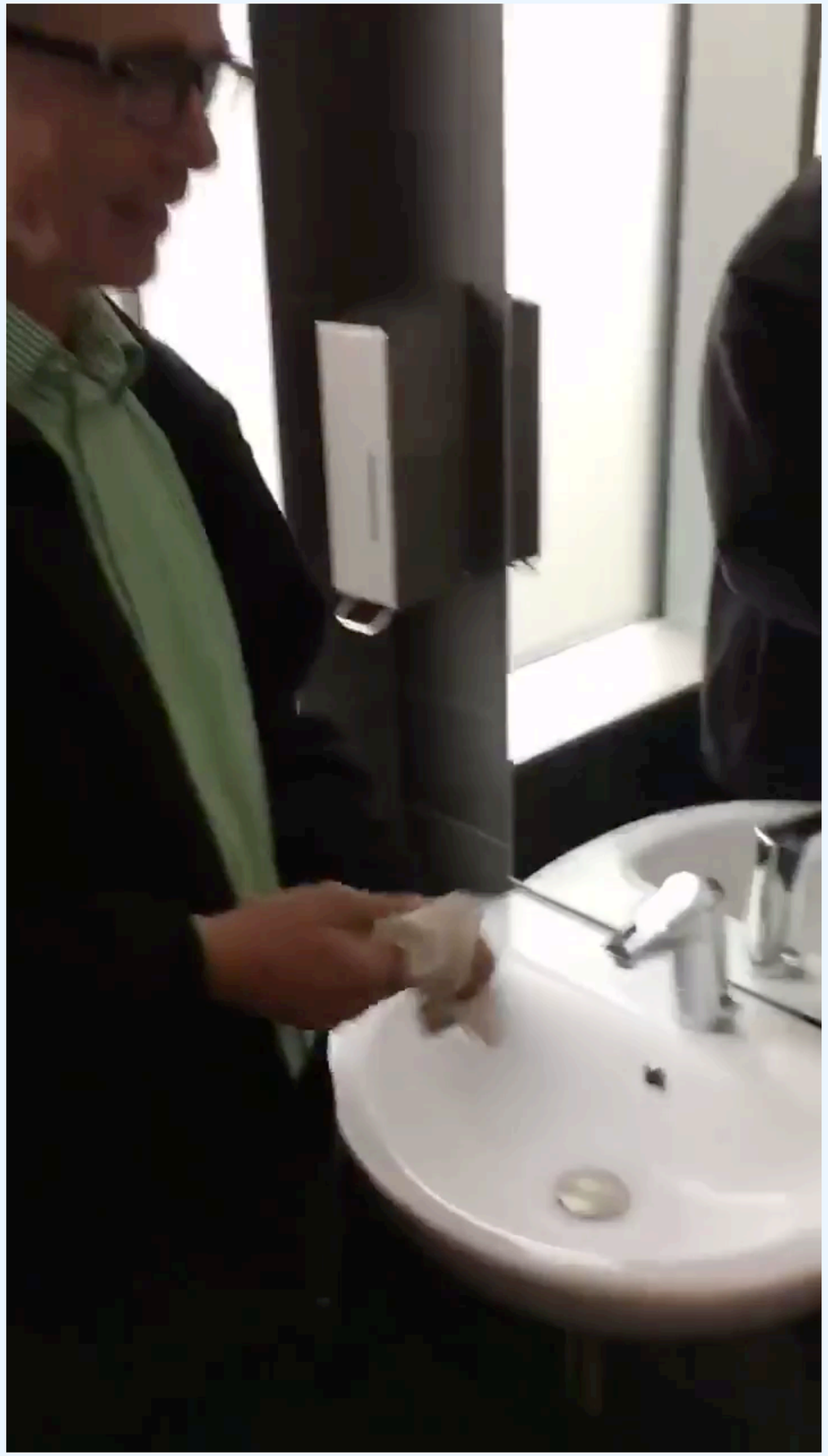
- Wir können mit Unit-Tests einfache, isolierte Methoden testen
- Wie testen wir größere Komponenten?
  - mehrere Klassen
  - mit Abhängigkeiten untereinander
- Wie können wir die Fehlerursache eingrenzen?
  - nicht nur Symptome erkennen → z.B.: Software stürzt ab
  - nicht nur Folgefehler erkennen → z.B.:
    - » Fehlerursache = fehlender Datensatz in Datenbank
    - » Fehlerwirkung = vorheriger Datensatz wird angezeigt

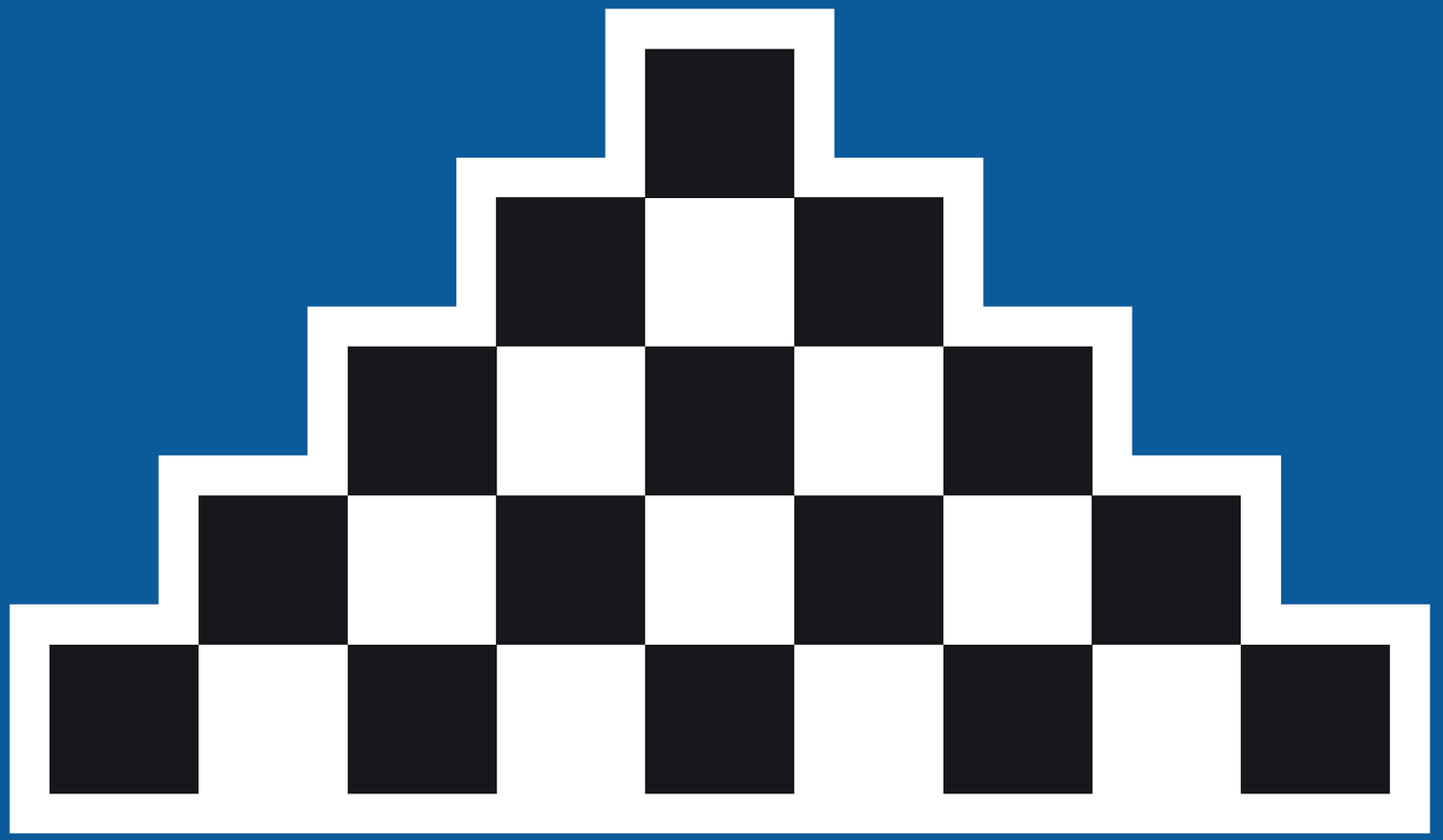
HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Fallbeispiel

Einzelne Komponenten getestet und funktional

Als Ganzes aber fehlerhaft



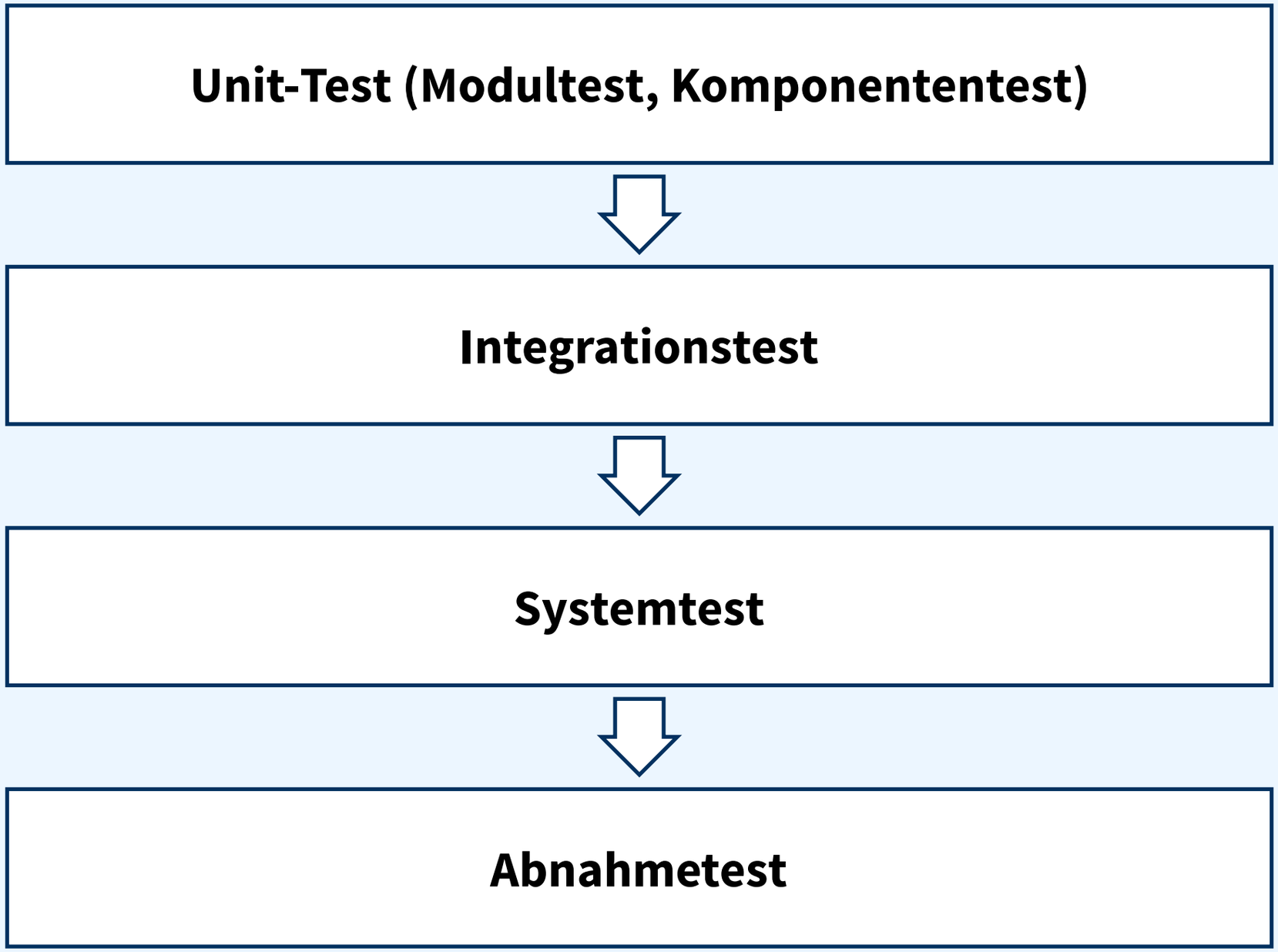


Grundlagen

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Prüfebenen

Tests lassen sich nach der Prüfebene in verschiedene *Teststufen* einteilen:



Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

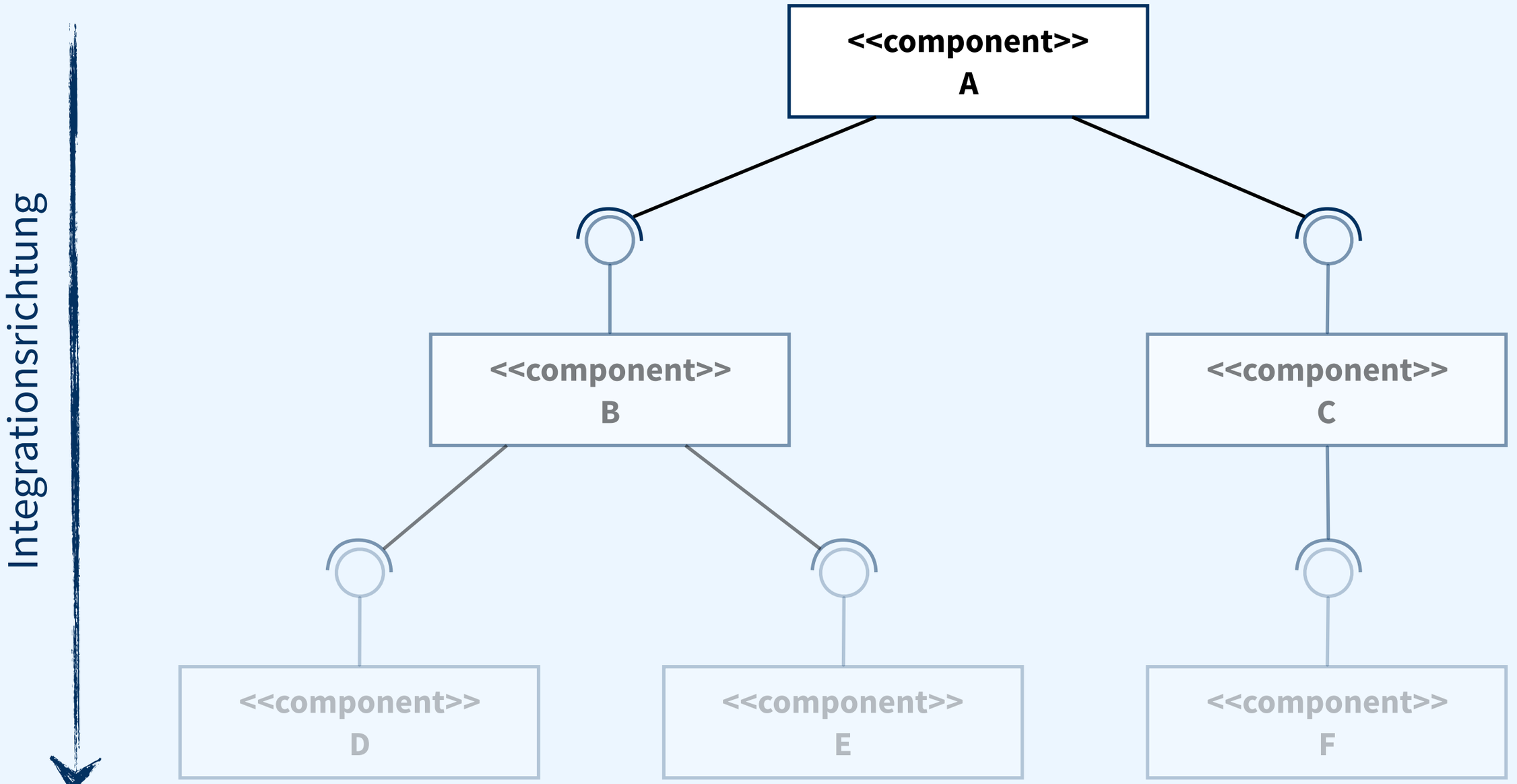
Software-Integration

- Nachdem einzelne Module fertiggestellt und getestet wurden, müssen sie zu einem lauffähigen System zusammengebaut werden
- Der Aufwand für die Integration darf nicht unterschätzt werden, da Fehler an den Schnittstellen auftreten können
- Der Grund für diese Fehler ist häufig eine unvollständige oder inkonsistente Spezifikation
- Bezüglich des *Ablaufs* unterscheidet man:
  - die Integration in einem Schritt
    (Big-Bang-Integration → hohes Risiko!)
  - die Inkrementelle Integration
    (Top-Down, Bottom-Up, Outside-In, kontinuierlich)

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Top-Down-Integration

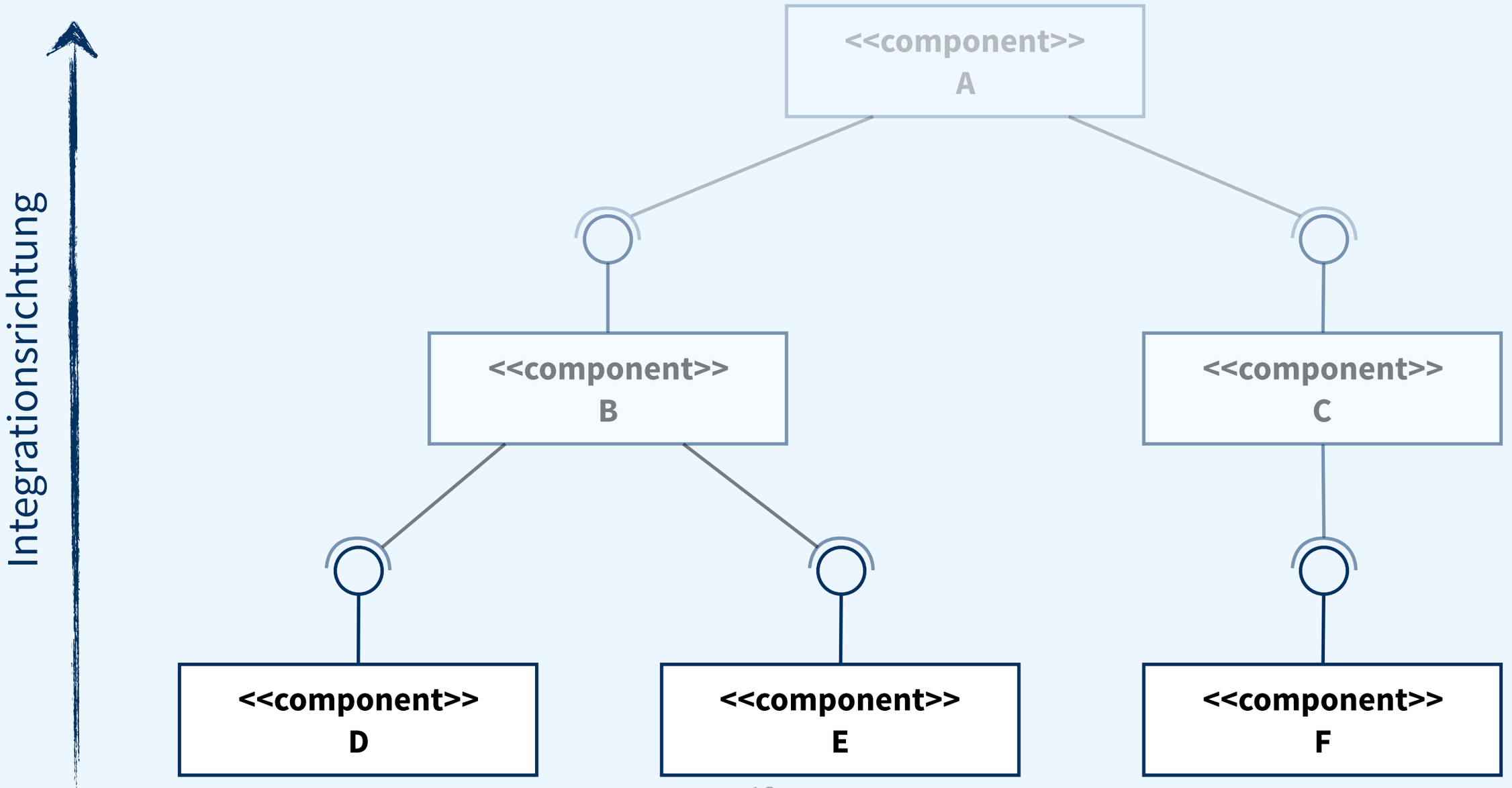
Die Integration beginnt auf der höchsten Hierarchieebene und arbeitet sich nach unten vor:



Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Bottom-Up-Integration

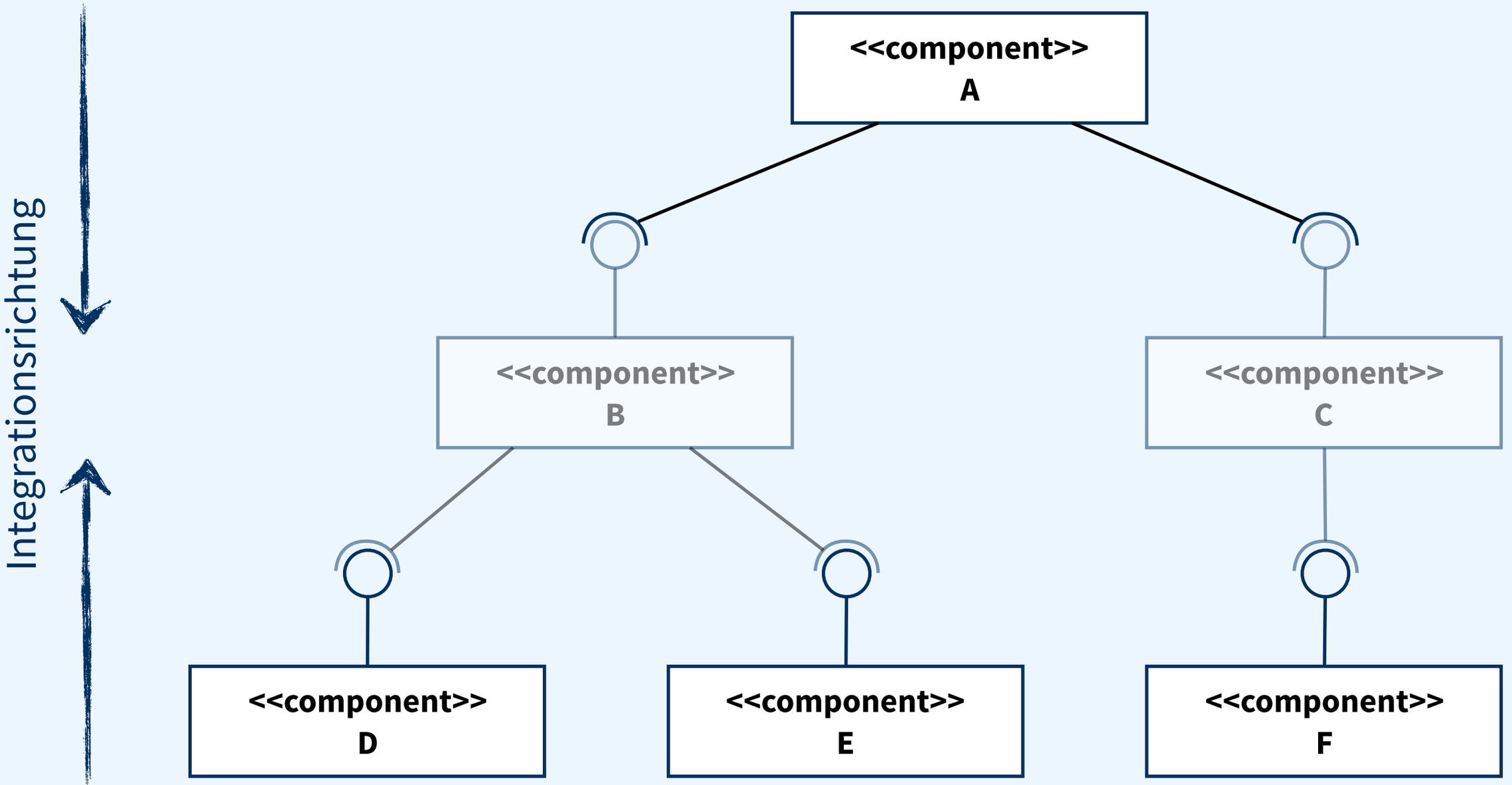
Es wird mit den Komponenten begonnen, die keine Dienstleistungen anderer Komponenten benötigen (bis auf Betriebssystem-Dienste):



HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Outside-In-Integration

Es werden die Komponenten der obersten und der untersten Schicht zuerst integriert (Tunnelbau-Prinzip)



HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

kontinuierliche Integration

- Jede Komponente wird sofort integriert (siehe XP)
- Es soll immer eine lauffähige (Teil-)Version in der Integrationsumgebung zur Verfügung stehen
- Auch die Integration muss geplant werden (Aufwand, Ablauf)
  - termingetriebene Integration
    (Deadline muss erreicht werden)
  - risikogetriebene Integration
    (Budget muss eingehalten werden)
  - testgetriebene Integration
    (Spezifikationen müssen eingehalten werden)

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

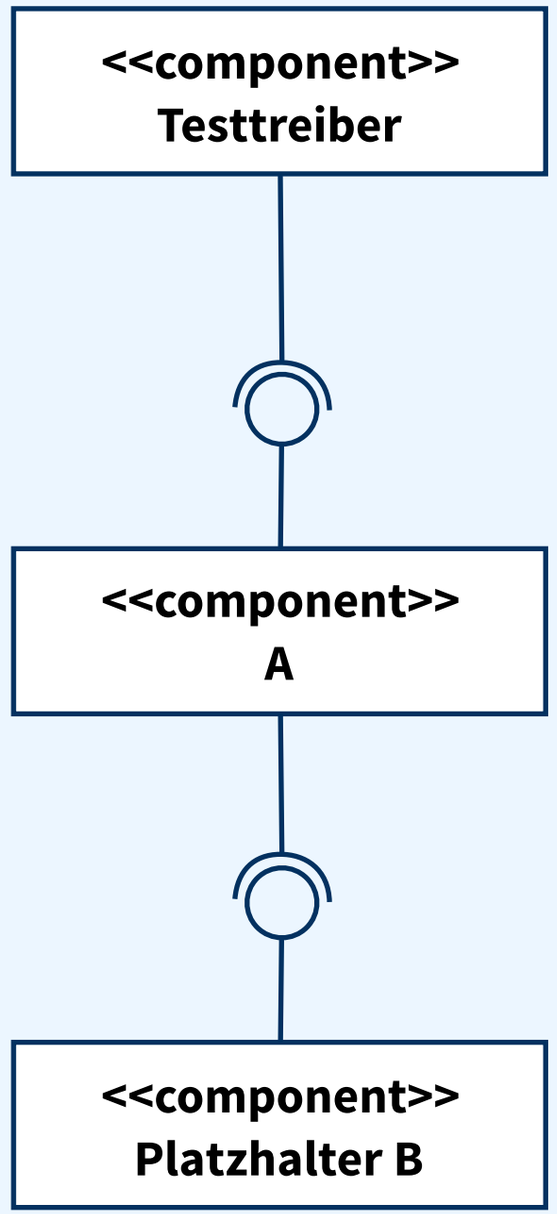
Integrationstest

- Es wird getestet, ob das Zusammenspiel der integrierten Module / Komponenten funktioniert
- An den Schnittstellen können verschiedene Fehler auftreten:
  - inkompatible Schnittstellenformate
  - Protokollfehler
  - semantische Fehler (unterschiedliche Einheiten)
  - Timing-Probleme
  - Kapazitäts- und Lastprobleme

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Integrationstest - Aufbau

- Wenn Komponente A getestet werden soll, wird ggf. ein *Testtreiber* benötigt
  - ▸ versorgt die Schnittstelle der Komponente A mit Testdaten
- Falls die Komponente A Dienste einer Komponente B nutzt, die noch nicht integriert ist, wird ein *Platzhalter* (*stub*) für B benötigt
  - ▸ vertritt die fehlende Komponente und liefert entweder
    - » konstante Werte, oder
    - » simuliert das Verhalten der späteren Komponente in Ausschnitten



HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

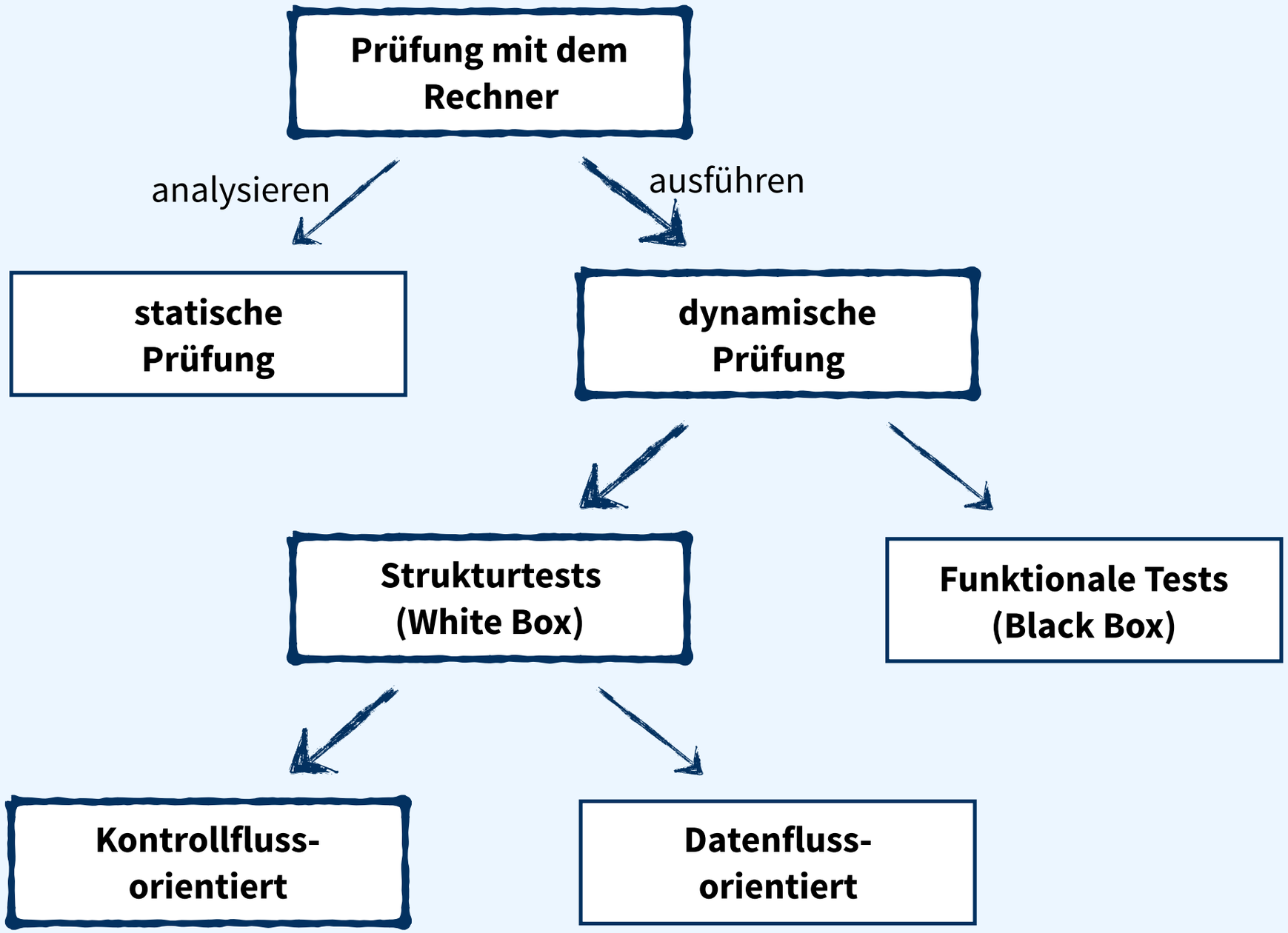
Testfälle

- Wir müssen also mit einer kleinen Teilmenge von Eingaben auskommen
- Um mit den wenigen Eingaben trotzdem viele Fehler finden zu können, müssen wir die Eingaben geeignet wählen
- Die Festlegung der Eingabedaten reicht für einen systematischen Test nicht aus
- Ein Testfall beschreibt:
  - die Ausgangssituation (Vorbedingungen, Randbedingungen)
  - Testdaten, die die vollständige Ausführung des Testobjekts bewirken
  - Sollwerte (das erwartete Ergebnis bzw. Verhalten)

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

White-Box-Tests

Klassifizierung der Testverfahren



Quelle: Balzert, Lehrbuch der Softwaretechnik

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

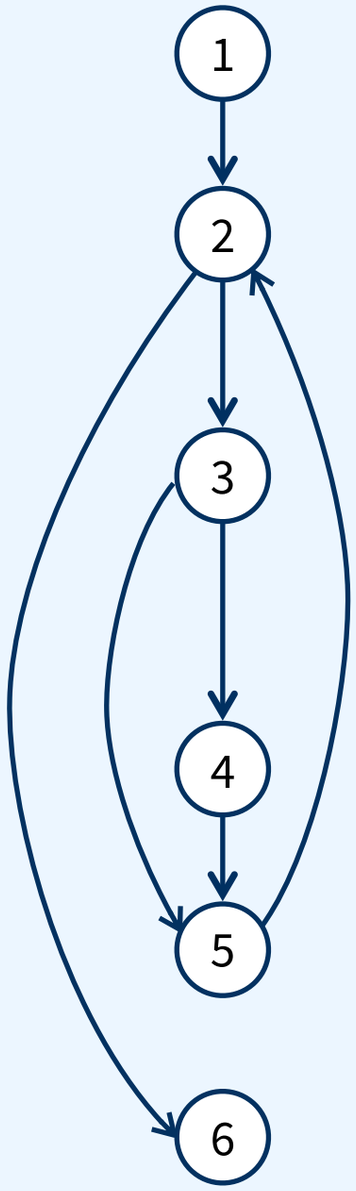
White-Box-Tests

Der Programmcode ist sichtbar. Mit einem Strukturtest wird die *Überdeckung* des Codes durch die Testfälle analysiert. Das Ziel ist es, eine vorgegebene *Überdeckungsrate* zu erreichen.

- Um die Überdeckung bestimmen zu können, muss das Programm mit zusätzlichen Anweisungen für die Messung instrumentiert werden
- Es werden verschiedene Überdeckungskriterien unterschieden:
  - Anweisungsüberdeckung (Statement Coverage, C_0-Test)
  - Zweigüberdeckung (Branch Coverage, C_1-Test)
  - Pfadüberdeckung (Path Coverage, C_2-Test)
  - Bedingungsüberdeckungstest (Condition Coverage, C_3-Test)

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Kontrollflussgraph



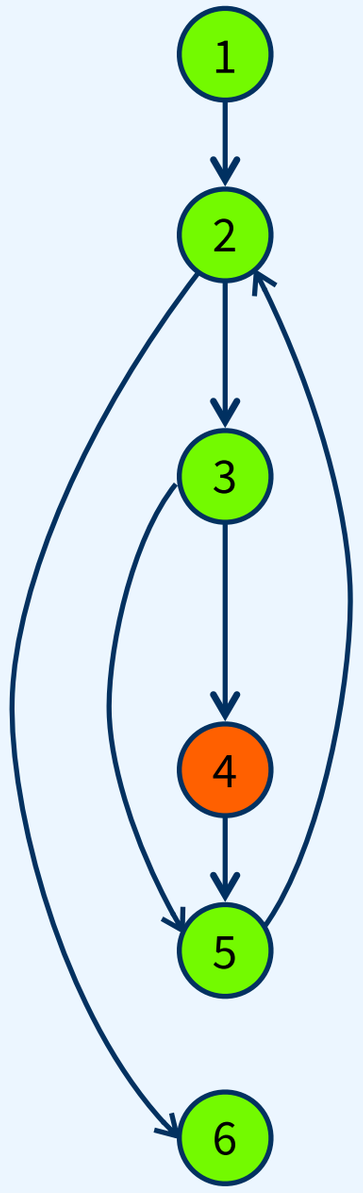

```
public void zaehlen(String eingabe) {
    int c = 0, v = 0;


    while ((c < eingabe.length())
            && (eingabe.charAt(c) >= 'A')
            && (eingabe.charAt(c) <= 'Z')) {

        if ((eingabe.charAt(c) == 'A')
                || (eingabe.charAt(c) == 'E')
                || (eingabe.charAt(c) == 'I')
                || (eingabe.charAt(c) == 'O')
                || (eingabe.charAt(c) == 'U')) {
            v++;
        }

        c++;
    }

    System.out.println("Der Text enthält " + c +
            " Grossbuchstaben. Davon sind " + v + " Vokale.");
}
```


HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Anweisungsüberdeckung

- Jede Anweisung im Programm muss im Test mindestens einmal ausgeführt werden (100% Anweisungsüberdeckung)
- Kennzahl C_0 = Anzahl der überdeckten Anweisungen / Gesamtzahl der Anweisungen
- Eine Anweisungsüberdeckung $C_0 > 80$ % sollte angestrebt werden
  - $C_0 = 100\%$ ist unrealistisch, da der Aufwand zur Erstellung von Testfällen für die letzten Prozentpunkte stark ansteigt

Beispiel:
eingabe ="XYZ"



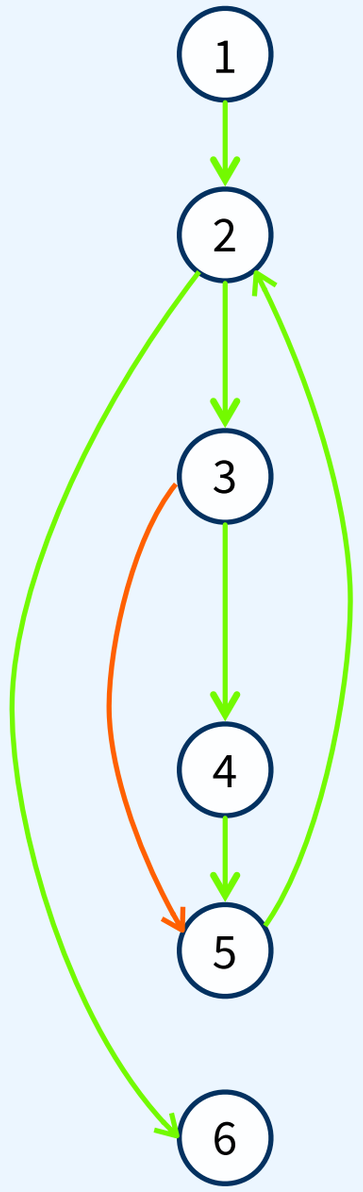
$C_0 = 5 / 6 =$ **83%**

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Zweigüberdeckung

- Jeder Zweig im Programm wird im Test mindestens einmal ausgeführt (100% Zweigüberdeckung)
- Kennzahl C_1 = Anzahl der überdeckten Zweige / Gesamtzahl der Zweige
- Die Zweigüberdeckung ist strenger als die Anweisungsüberdeckung, wenn das Programm leere Zweige enthält (z.B. if-then ohne else- Zweig)

Beispiel:
eingabe ="AA"



C_1 = 6 / 7 = **86%**
(C_0 = 100%)

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Pfad- u. Bedingungsüberdeckung

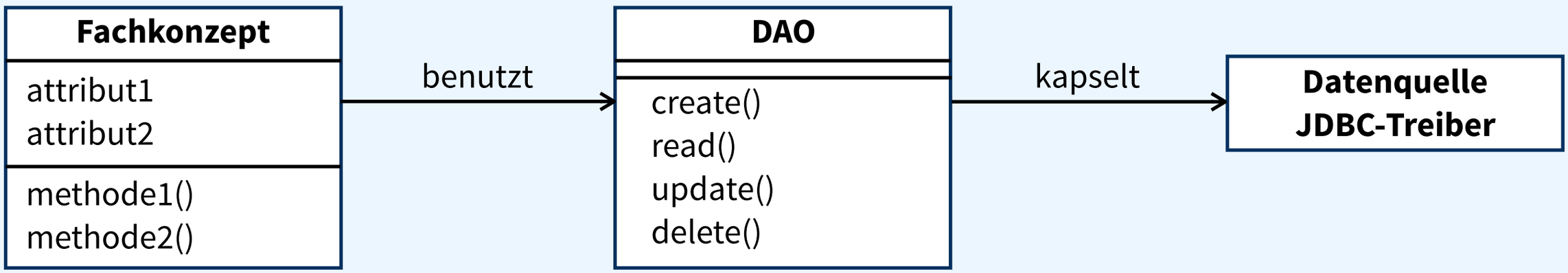
- Jeder Pfad im Programm wird im Test mindestens einmal ausgeführt (100% Pfadüberdeckung, C_2-Test)
  - alle Kombinationen aller Programmzweige bei maximalem Durchlauf aller Schleifen
  - unmöglich bei Endlosschleifen
  - strukturierter Pfadüberdeckungstest ist sehr aufwändig
- Jede atomare Bedingung wird einmal mit `true` und `false` getestet (100% Bedingungsüberdeckung, C_3-Test)
  - unmöglich bei Entweder-Oder-Bedingung (Vokale)
- Beides werden wir hier nicht weiter vertiefen…

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Testbarkeit

Testbarkeit

- Die Architektur der Software bestimmt auch die Testbarkeit
  - Klassen können häufig nicht isoliert getestet werden
- Beispiel: Es soll eine Fachkonzeptklasse getestet werden, aber …
  - … die verwendete DAO-Klasse ist aber noch nicht fertiggestellt.
  - … die DAO-Klasse soll eine andere Testdatenquelle verwenden.



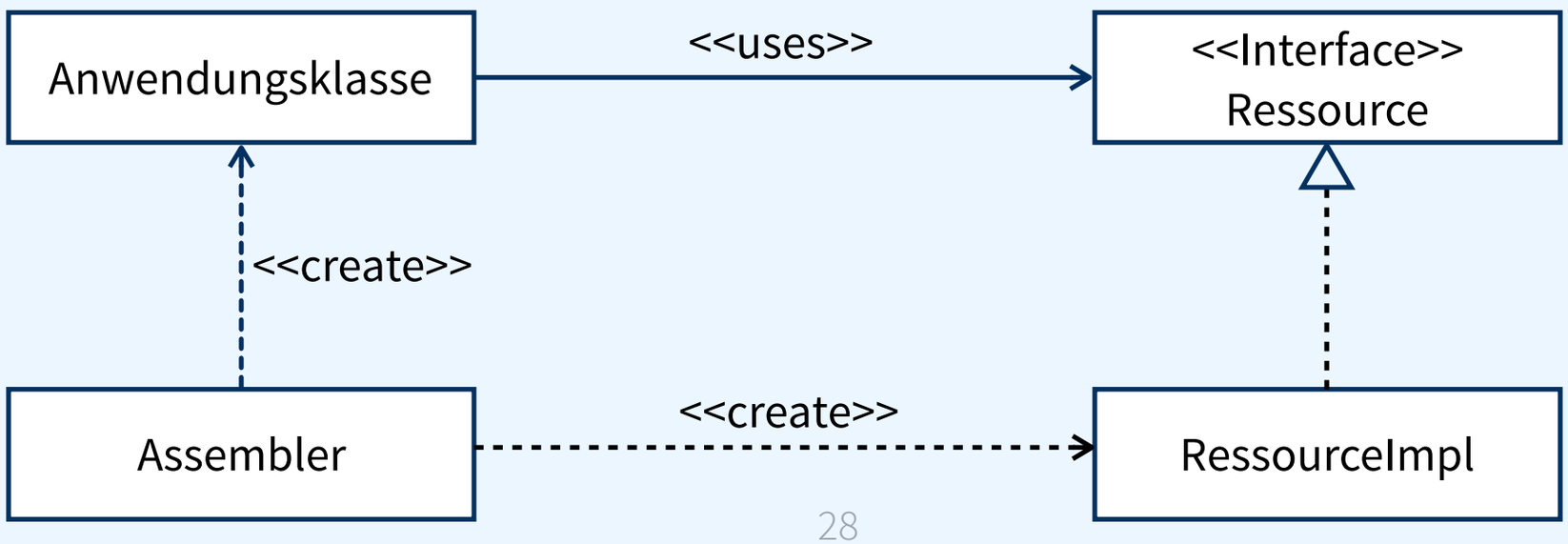
Wie erhalten Klassen Zugriff auf die benötigten Ressourcen?

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Testbarkeit

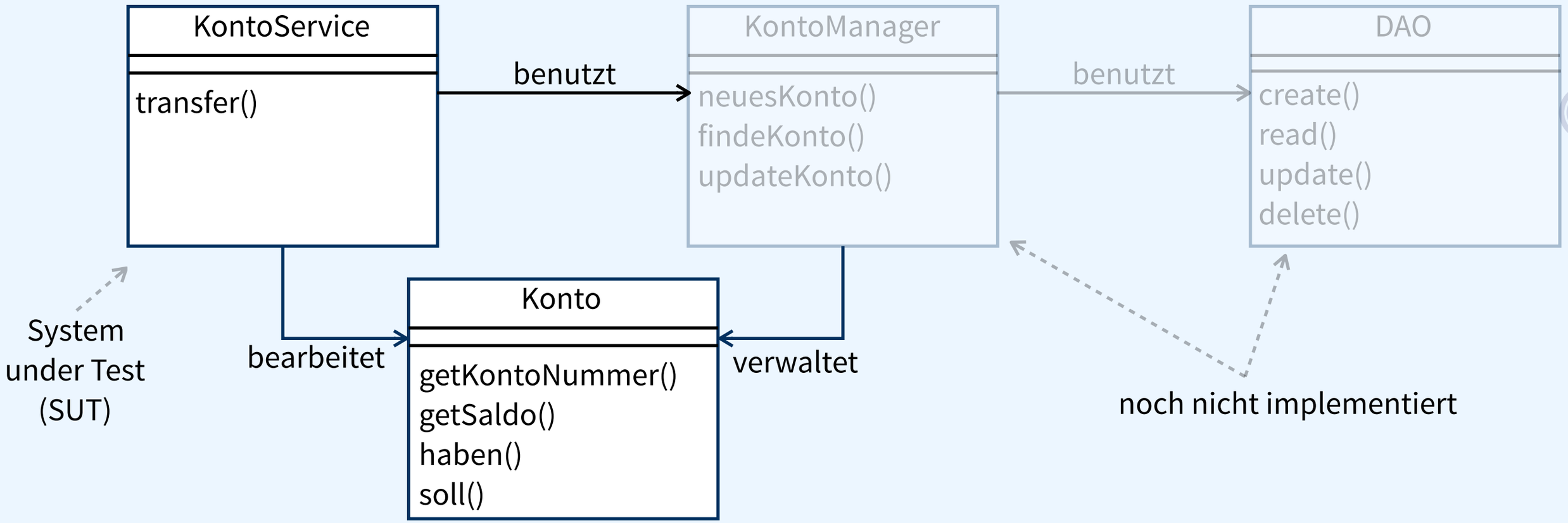
- Es gibt drei Möglichkeiten, wie ein Objekt A eine Referenz auf ein benötigtes Objekt B erhält (in der Reihenfolge der Testbarkeit):
  - A erzeugt selber eine Instanz der Klasse B
  - Verwendung eines Namensdienstes (Service Locator)
  - Dependency Injection

- Dependency Injection = Referenzen zu anderen Objekten werden von außen injiziert (Hollywood Prinzip → *Don't call us, we call you!*)

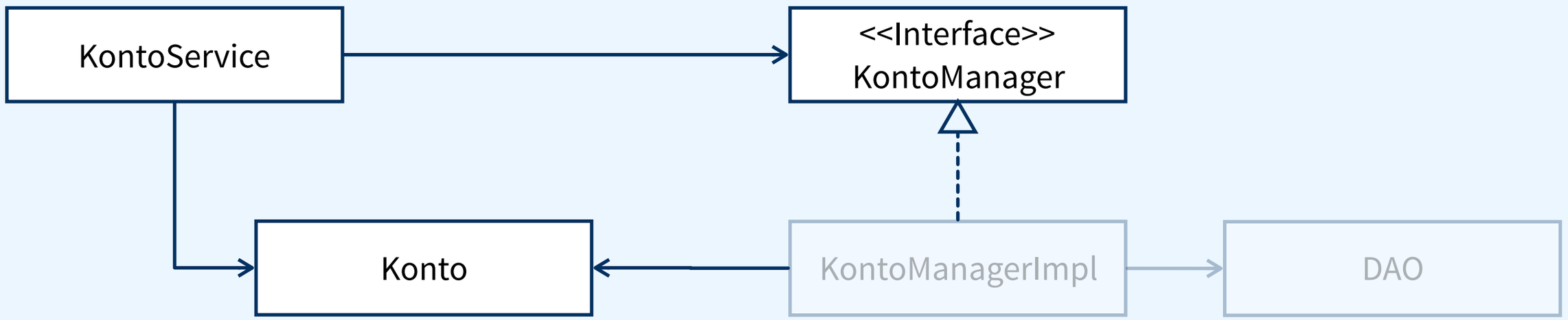


HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Fallbeispiel: Kontenverwaltung



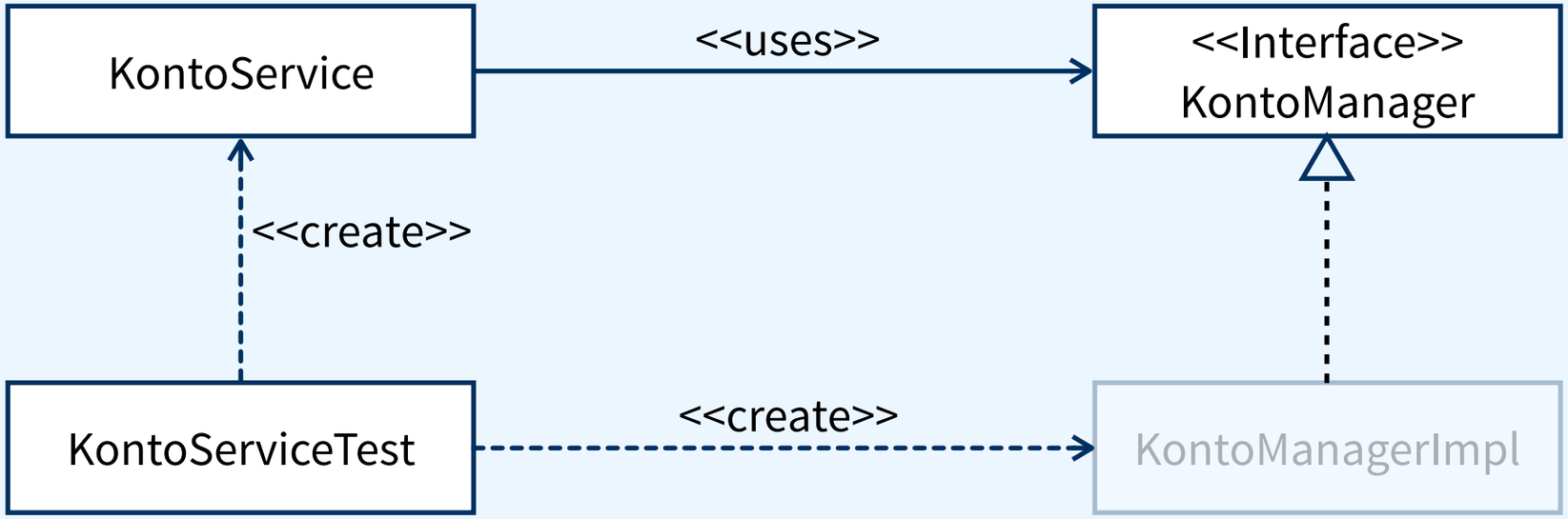
bessere Testbarkeit:



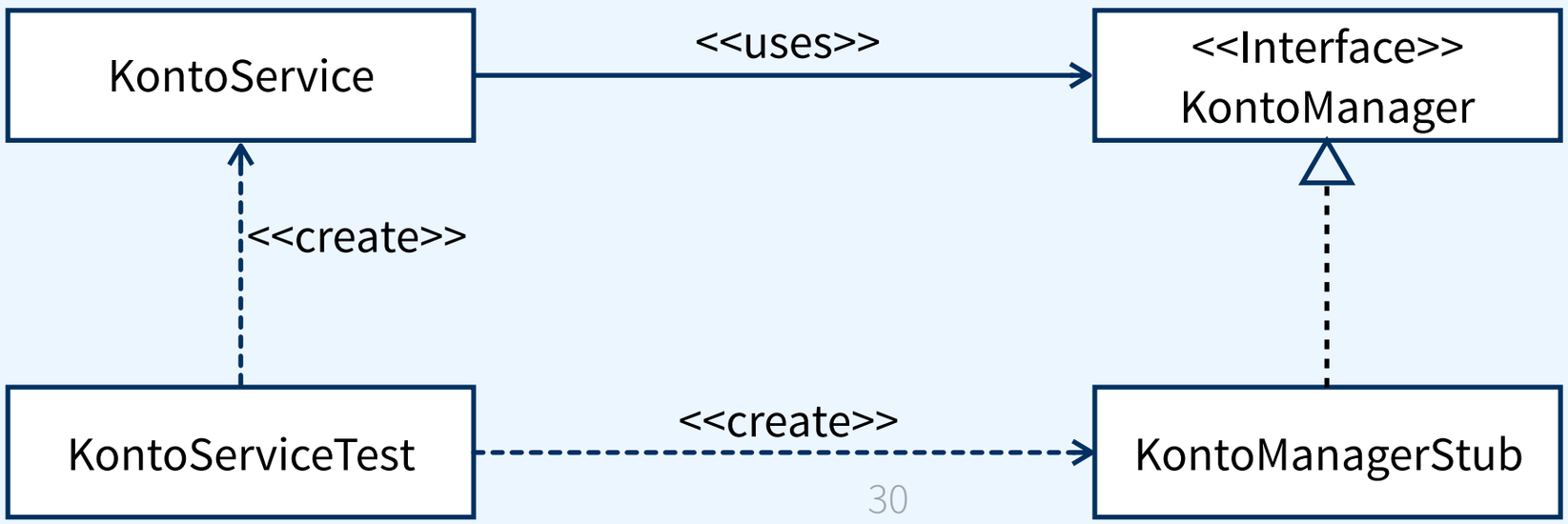
HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

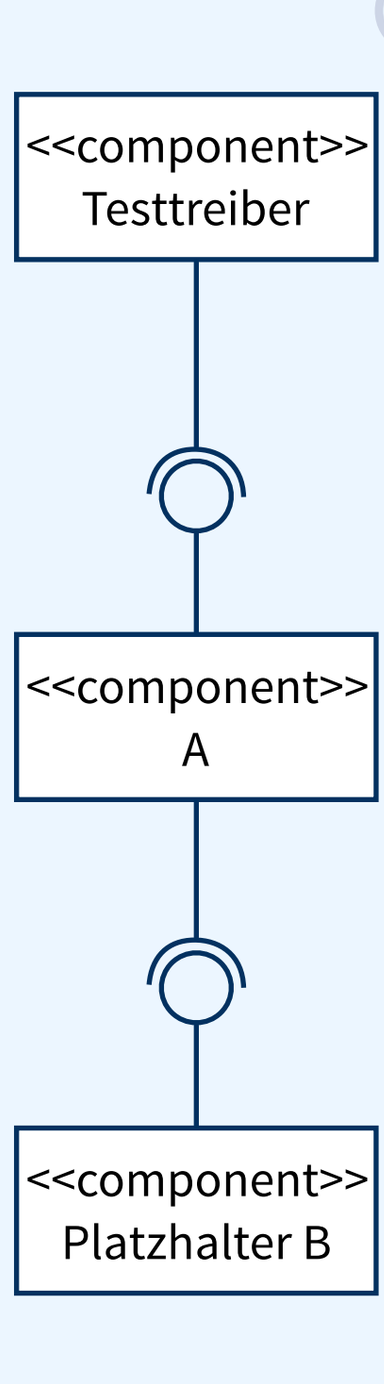
Stub-Objekte

Unser Testfall für den KontoService könnte folgendermaßen aussehen:



Um den KontoService testen zu können, implementieren wir einen Stub (Platzhalter), der das KontoManager-Interface implementiert:





HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

(Hilfs-)Klasse Konto

```java
public class Konto {

  private String kontoNummer;
  private double saldo;

  public Konto(String kontoNummer, double saldo) {
    this.kontoNummer = kontoNummer;
    this.saldo = saldo;
  }

  public void soll(double umsatz) {
    saldo -= umsatz;
  }
  public void haben(double umsatz) {
    saldo += umsatz;
  }
  public String getKontoNummer() {
    return kontoNummer;
  }
  public double getSaldo() {
    return saldo;
  }
}
```


HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Klasse KontoService

```java
public class KontoService {
  private KontoManager kontoManager;

  public void setKontoManager(KontoManager kontoManager) {
    this.kontoManager = kontoManager;
  }

  public void transfer(String quelle, String ziel, double umsatz) {
    Konto quelleKonto = kontoManager.findeKonto(quelle);
    Konto zielKonto = kontoManager.findeKonto(ziel);
    quelleKonto.soll(umsatz);
    zielKonto.haben(umsatz);
    kontoManager.updateKonto(quelleKonto);
    kontoManager.updateKonto(zielKonto);
  }
}
```

Programmierung gegen Interface

Instanz von KontoManager nicht selbst erzeugt

zu testende Implementierung

```java
public interface KontoManager {
  Konto findeKonto(String kontoNummer);
  void updateKonto(Konto konto);
  void neuesKonto(Konto konto);
}
```


HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Klasse KontoManagerStub

Rudimentäre Implementierung für das Testen von `KontoService`

```
public class KontoManagerStub implements KontoManager {

  private Konto konto1;
  private Konto konto2;

 public void neuesKonto(Konto konto) {
    if (konto1 == null) { konto1 = konto; }
    else { konto2 = konto; }
  }

  public void updateKonto(Konto konto) {
  }

  public Konto findeKonto(String kontoNummer) {
    if (konto1.getKontoNummer().equals(kontoNummer))
      return konto1;
    if (konto2.getKontoNummer().equals(kontoNummer))
      return konto2;
    throw new IllegalArgumentException();
  }
}
```

Es können nur zwei Konten verwaltet werden

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Testtreiber KontoServiceTest

```
public class KontoServiceTest {
  @Test
  public void testTransfer() {
    Konto quelle = new Konto("1", 1000.0);
    Konto ziel = new Konto("2", 100.0);
    KontoManager stub = new KontoManagerStub();   <-------- Erzeugung und Konfiguration des Platzhalters
    stub.neuesKonto(quelle);
    stub.neuesKonto(ziel);

    KontoService service = new KontoService();
    service.setKontoManager(stub);   <---------- DI des Platzhalters

    service.transfer("1", "2", 500.0);  <---------- Aufruf der zu testenden Methode

    assertEquals(500.0, quelle.getSaldo(), 0.0);  <---------- Zustand überprüfen
    assertEquals(600.0, ziel.getSaldo(), 0.0);
  }
}
```


HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Werkzeuge

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Mock-Objekte

Mocks

- Wir haben Platzhalter-Objekte (Stubs) kennengelernt
  - die zu testenden Klassen werden von der Umgebung entkoppelt
  - Stub enthält erforderliche Fachlogik
  - Zustandstest (Test der Zustandsänderung)
  - Stubs sind aber aufwändig zu programmieren!

- Testen mit Mock-Objekten
  - Mocks sorgen ebenfalls für eine Entkopplung
  - Mocks unterstützen feingranulare Tests (auf Methodenebene)
  - Mocks enthalten keine Fachlogik
  - Mocks kontrollieren die Anzahl der Methodenaufrufe (Verhaltenstest)

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Mocks

- Da Mocks keine Fachlogik enthalten, können sie aus einer Schnittstellendefinition automatisch generiert werden
- Dies erfordert die Unterstützung durch entsprechende Werkzeuge oder Frameworks
  - wir werden EasyMock vorstellen
- Der Aufwand für die Testimplementierung wird reduziert
- Easymock besteht aus einer JAR-Bibliothek, die in das Projekt mit aufgenommen werden muss:

```
<dependency>
  <groupId>org.easymock</groupId>
  <artifactId>easymock</artifactId>
  <version>3.1</version>
  <scope>test</scope>
</dependency>
```


HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

EasyMock

- Aus einem Interface wird im Testtreiber mit der Methode `createMock` zur Testlaufzeit ein Mock-Objekt generiert
- Das Mock-Objekt befindet sich dann automatisch in einer *Aufzeichnungsphase*
- Es werden alle, während des eigentlichen Tests erwarteten, Methodenaufrufe durchgespielt und vom Mock-Objekt aufgezeichnet
- Danach wird das Mock-Objekt in einen *Replay*-Modus versetzt
- Der eigentliche Testfall wird gestartet
- In einem *Verify*-Modus vergleicht das Mock-Objekt dann die aufgezeichneten Methodenaufrufe mit den tatsächlich im Test erfolgten Methodenaufrufen
- Abweichungen führen zum Scheitern des JUnit-Tests

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

EasyMock

```java
import static org.easymock.EasyMock.*;

public class KontoServiceEasyMockTest {
  @Test
  public void test() {
    Konto quelle = new Konto("1", 1000.0);
    Konto ziel = new Konto("2", 100.0);

    KontoManager kontoManagerMock = createMock(KontoManager.class);
    expect(kontoManagerMock.findeKonto("1")).andReturn(quelle);
    expect(kontoManagerMock.findeKonto("2")).andReturn(ziel);
    kontoManagerMock.updateKonto(quelle);
    kontoManagerMock.updateKonto(ziel);

    replay(kontoManagerMock);
    KontoService service = new KontoService();
    service.setKontoManager(kontoManagerMock);

    service.transfer("1", "2", 500.0);

    verify(kontoManagerMock);
  }
}
```

Erzeugung und Konfiguration des Mocks

DI des Mocks

Aufruf der zu testenden Methode

Verhalten überprüfen

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

assert-Matcher

assert-Historie

- Einzelne Methode: `assert(boolean test)`
  - universell nutzbar, z.B.: `assert(a > b)`
  - wenn das assert aber fehlschlägt, kann keine hilfreiche Fehlermeldung generiert werden (kein Zugriff mehr auf a oder b)
- Sammlung von Methoden: assertEquals, assertGreater, etc…
  - für spezielle Zwecke gut nutzbar, z.B.: `assertGreater(a,b)`
  - eine hilfreiche Fehlermeldung kann erzeugt werden
  - Sammlung wird schnell sehr groß
- Neueste Variante: assert mittels Matcher-Objekten
  - `assertThat(a, greaterThan(b));`
  - kann auch komplex verschachtelt werden

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Hamcrest

- Die hamcrest-Bibliothek bietet vordefinierte Matcher an, die mit JUnit verwendet werden können
  - not(), equalTo, greaterThan(), equalToIgnoringWhiteSpace() …
- Beispiel mit assertTrue() und assertThat():
  - `assertTrue("nuts", biscuit.nutCount() > 3 && biscuit.nutCount() % 2 == 0);`
    Fehlermeldung: »nuts: got true, wanted false«
  - `assertThat("nuts", biscuit.nutCount(), allOf(greaterThan(3), evenNumber()));`
    Fehlermeldung: »nuts: expected a number greater than 3 and even got 5«
- Hamcrest besteht aus einer JAR-Bibliothek, die in das Projekt mit aufgenommen werden muss:

```
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
  <version>1.3</version>
</dependency>
```


HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Hamcrest

```
public class KontoServiceEasyMockTest {
  @Test
  public void test() {
    Konto quelle = new Konto("1", 1000.0);
    Konto ziel = new Konto("2", 100.0);

    KontoManager kontoManagerMock = createMock(KontoManager.class);
    expect(kontoManagerMock.findeKonto("1")).andReturn(quelle);
    expect(kontoManagerMock.findeKonto("2")).andReturn(ziel);
    kontoManagerMock.updateKonto(quelle);
    kontoManagerMock.updateKonto(ziel);

    replay(kontoManagerMock);
    KontoService service = new KontoService();
    service.setKontoManager(kontoManagerMock);

    service.transfer("1", "2", 500.0);

    verify(kontoManagerMock);

    assertThat(quelle.getSaldo(), equalTo(500.0));
    assertThat(ziel.getSaldo(), equalTo(600.0));
  }
}
```

Überprüfung des Zustands mittels Matcher

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Test-Metriken

Metriken

- Test-Metriken können beim Ausführen der Tests gemessen werden
  - C_0 Anweisungsüberdeckung
  - C_1 Zweigüberdeckung
- Die Java-Klassen müssen beim Kompilieren instrumentiert werden
  - Ausführung einzelner Codezeilen wird protokolliert
  - manuell eine komplexe Aufgabe
- Die Bibliothek »JaCoCo« hilft bei der Umsetzung:

```
<reporting>
   <plugins>
     <plugin>
       <groupId>org.jacoco</groupId>
       <artifactId>jacoco-maven-plugin</artifactId>
       <version>0.8.2</version>
     </plugin>
…
```


HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

JaCoCo

Testmetriken werden automatisch von »`mvn site`« erstellt:

beispiel > de.betermieux > ZeichenZaehler — Sessions

ZeichenZaehler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
zaehlen(String)		100%		62%	6	9	0	13	0	1
ZeichenZaehler()		100%		n/a	0	1	0	1	0	1
Total	0 of 66	100%	6 of 16	62%	6	10	0	14	0	2

Created with JaCoCo 0.8.1.201803210924

McCabes Cyclomatic Complexity (sollte <10 sein)

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Test der Benutzeroberfläche

Test der Benutzerschnittstelle

- Wie soll man grafische Oberflächen testen?
  - JUnit bietet dafür keine Hilfsmittel
- Häufig sind Benutzerschnittstellen programmiert
  - Sourcecode bis »dicht« unter der Oberfläche
  - das lässt sich mit JUnit testen
- Spezielle Werkzeuge testen die Benutzeroberfläche
  - z.B. Open-Source-Tool Selenium für den Test von Web-Anwendungen
  - realisiert als Browser-Plugin

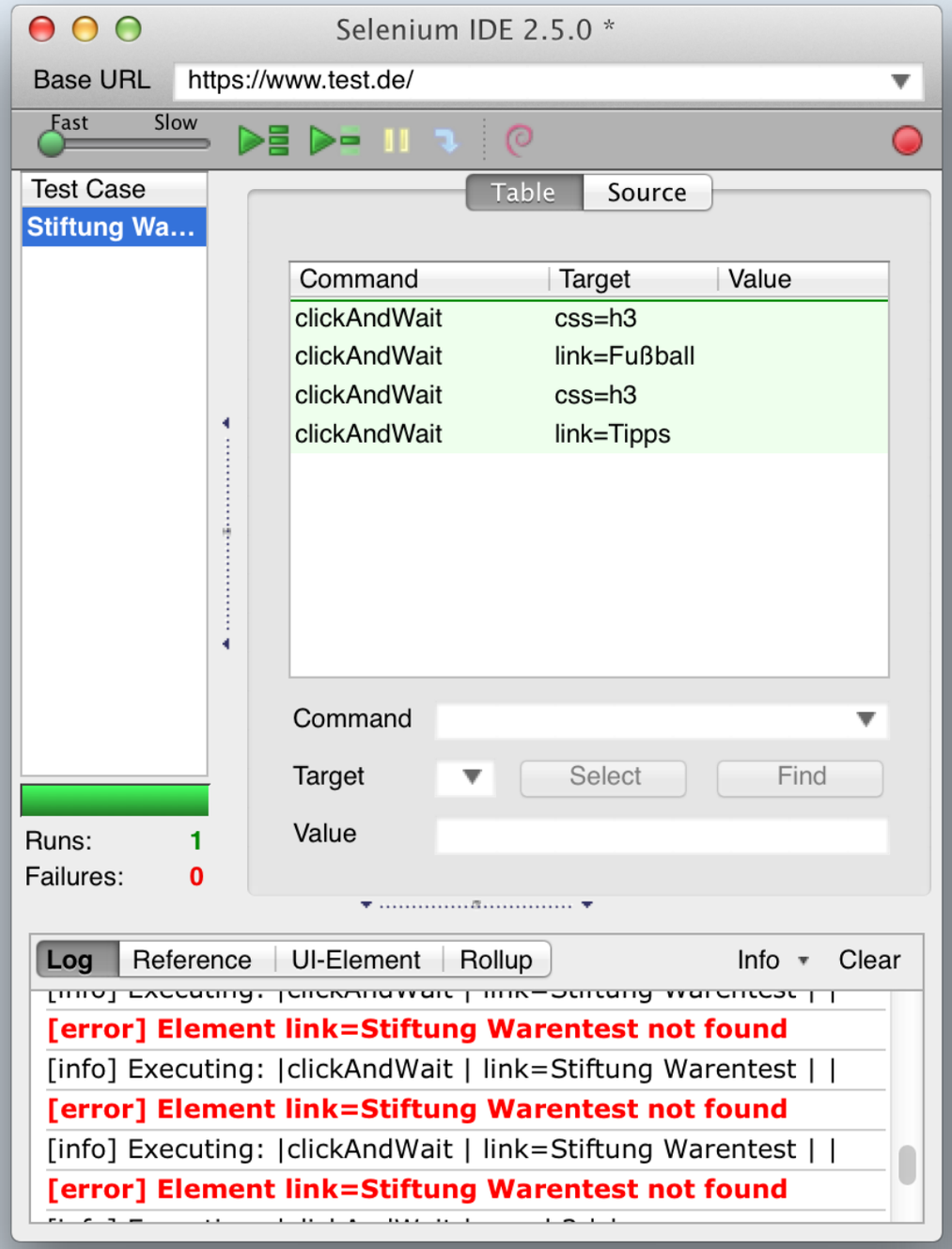
HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

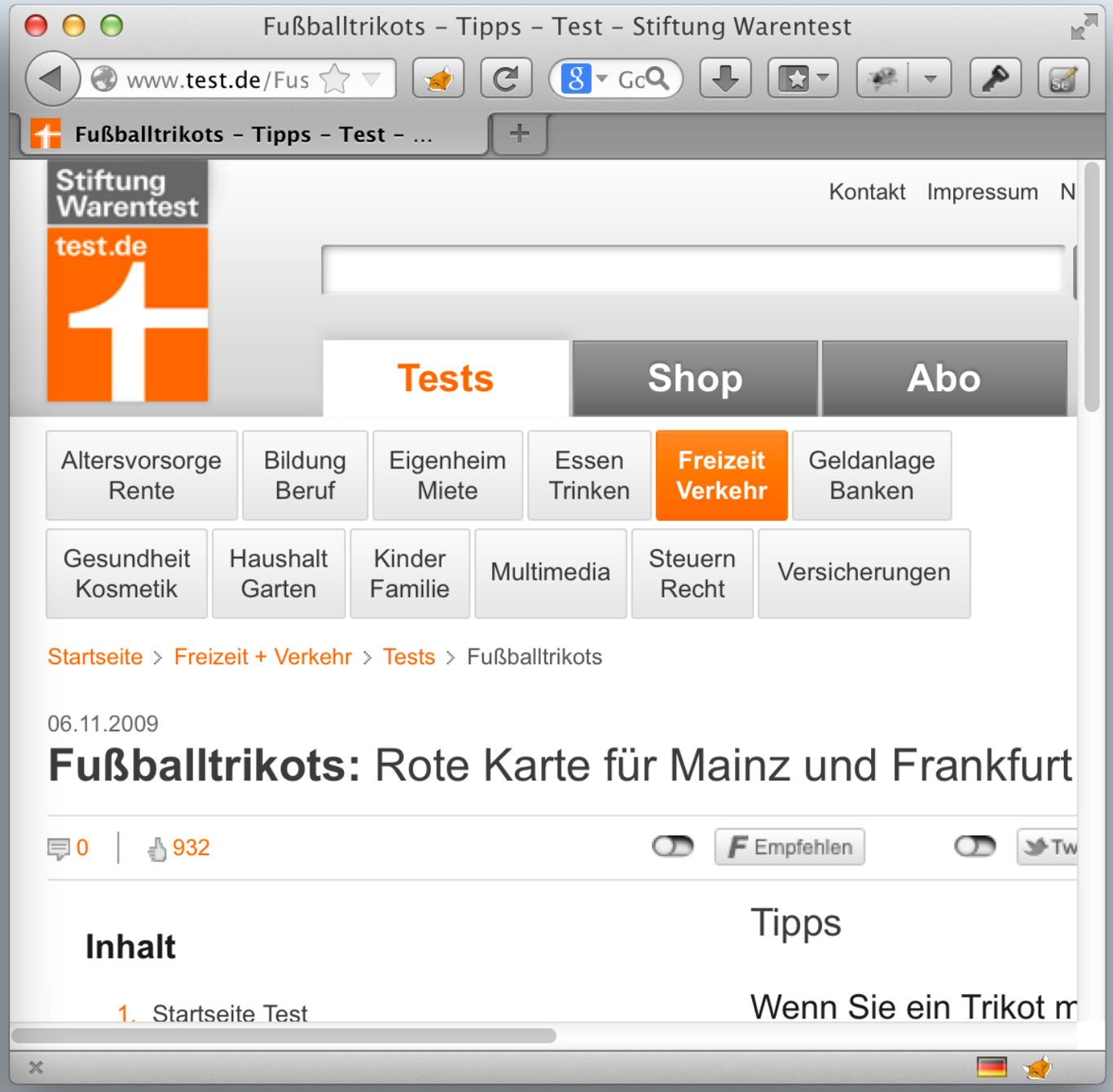
Selenium Funktionsweise

- Testet Aktion und Reaktion der Webseite
- Record-And-Play
  - Browser-Plugin wechselt in den Aufnahmemodus
  - Benutzer agiert auf der Web-Anwendung, erzeugt damit Testfall
    - » z.B. Einloggen, Artikel schreiben, Ausloggen
  - Aufnahmemodus des Plugins beenden, aufgezeichneten Testfall abspeichern
  - Testfall kann beliebig abgespielt werden
    - » manuell aus dem Plugin heraus
    - » automatisch, z.B. über ein Maven-Plugin
- Testfälle sollten am Ende Ursprungszustand wiederherstellen

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Selenium





HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Zusammenfassung

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Zusammenfassung

- Integrationstests unterscheiden sich von Unit-Tests, weil die zu testenden Anwendungsklassen von weiteren Klassen abhängen
  - Stub- und Mock-Objekte reduzieren die Abhängigkeiten
  - Dependency-Injection erlaubt einfacheres Testen
- Testmetriken bewerten die Testabdeckung
  - Metriken können automatisch erstellt werden
- Testen der Benutzerschnittstelle erfordert eigene Werkzeuge

HFU
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Danke

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen