UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Software Entwicklung & Programmierung

Modultests

PALUNO
The Ruhr Institute for Software Technology

Disclaimer

Bilder und Texte der Veranstaltungsfolien und -unterlagen sowie das gesprochene Wort innerhalb der Veranstaltung und Lehr-Lern-Videos dienen allein dem Selbst- bzw. Gruppenstudium. Jede weiterführende Nutzung ist den Teilnehmenden der Moodle-Kurse untersagt, z.B. Verbreitung an andere Studierende, in sozialen Netzwerken, dem Internet!

Darüber hinaus ist ein studentischer Mitschnitt von Webkonferenzen im Rahmen der Lehre nicht erlaubt.

Zielsetzung

Am Ende dieser Präsenzstunde könnt Ihr:

- den **Testprozess** in Bezug auf Modultests erläutern
- **Modultestfälle** erstellen
- **Modultests** mithilfe von **JUnit implementieren**

Agenda

1. **Einführung**
2. Modultestentwurf und Durchführung
3. Testauswertung
4. Test-Suits

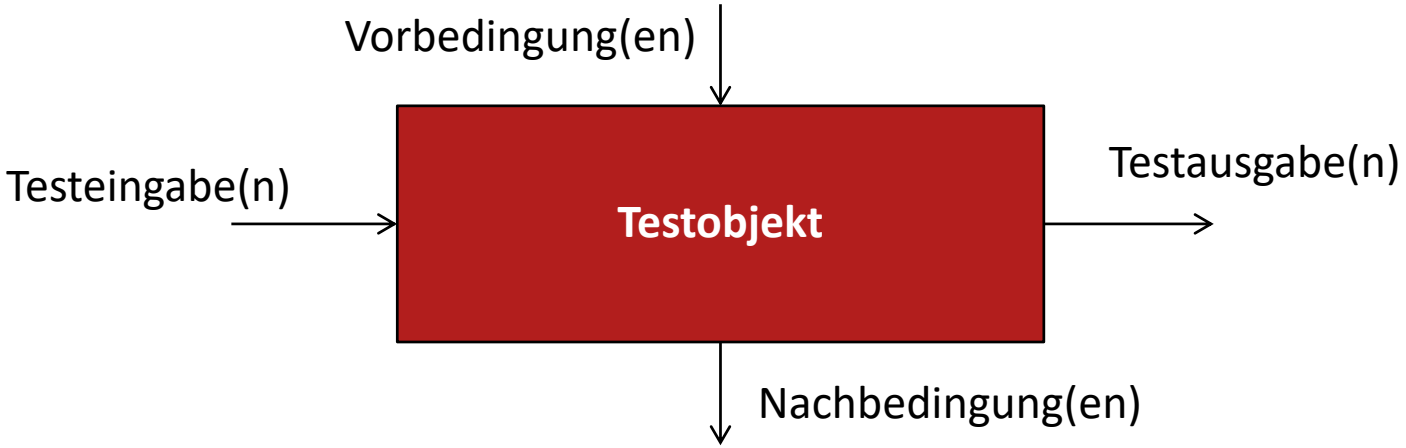
Testen – Definition (1)

- **Testen** bezeichnet im Allgemeinen die **stichprobenartige Ausführung** eines **Testobjekts**
  - unter spezifizierten **Bedingungen** (Eingaben, Vorbedingungen, etc.)
  - zum **Zwecke** des Überprüfens
    - der (beobachteten) Ergebnisse (Ausgaben, Nachbedingungen, etc.)
    - im Hinblick auf gewisse gewünschte (erwartete) Eigenschaften
- Testen umfasst nicht die Fehlersuche (Debugging) und Fehlerkorrektur

Testen – Definition (2)

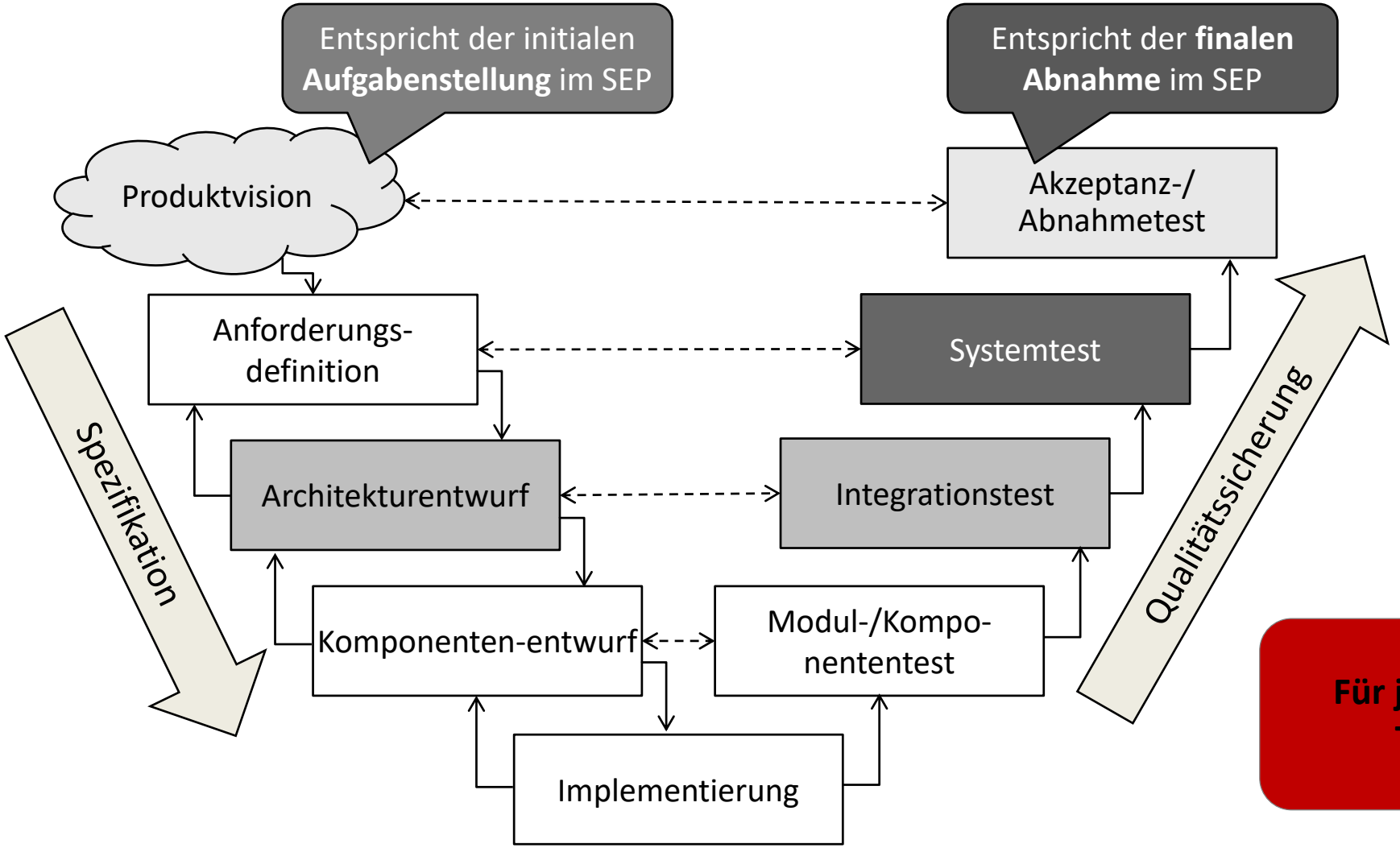
- Ablauf eines **Testvorgangs**:
  - Auswahl einer zu testenden **Software-Einheit** als **Testobjekt** (z.B. eine Methode)
  - Auswahl von **Eingabewerten** erfolgt stichprobenartig, da in der Regel ein Programm nicht vollständig (d.h. mit allen Eingabewerten) getestet werden kann
  - Bestimmung der Soll-Ergebnisse (**erwartete Ergebnisse**)
  - **Ausführung** des Testgegenstands mit den definierten, konkreten Eingabewerten
  - **Überprüfung**, ob die beobachteten Ausgabewerte den erwarteten Ausgabewerten entsprechen

Grundlegende Begriffe



- **Testfall:** Eingabedaten, erwartete Ausgaben des Testobjekts, Vor- und Nachbedingungen der Testdurchführung
- **Testziel**: Zweck des Testentwurfs und der Testdurchführung
- **Testobjekt:** zu testende Funktion/Methode, Komponente oder System
- **Testkritierium:** Abbruchbedingung für ausreichendes Testen
  (Erinnerung: vollständiges Testen ist i. d. R. **unmöglich**!)
- **Testabdeckung:** Grad der Abdeckung einer Überdeckungseinheit (z. B. Anforderungen, Codezeilen) durch Tests

Teststufen im V-Modell



Für jede Teststufe werden Testfälle benötigt!

Angelehnt an V-Modell (s. Vorlesung „Software Engineering“)

Teststufen

- **Modul-/Komponententest**
  - Test eines **isolierten Moduls** (z. B. einer Java-Klasse) **unabhängig** von **anderen Modulen**
  - **Eingaben und Sollverhalten** werden aus dem Feinentwurf der Module abgeleitet
  - I. d. R. Einsicht in Quellcode möglich („White-Box“-Testen)
- **Integrationstest**
  - Test des **Zusammenspiels** von zwei oder **mehreren Modulen**, die den Modultest bestanden haben
  - Eingaben und Sollverhalten werden aus der Architektur abgeleitet

Im Rahmen des SEP nicht durchgeführt

- **Systemtest**
  - Test des **Gesamtsystems** gegen die **Anforderungen** nach erfolgreicher Integration
  - Eingaben und Sollverhalten werden aus der **Anforderungs-spezifikation** abgeleitet

Siehe Systemtest-Foliensatz

- **Akzeptanz-/Abnahmetest**
  - Test des Gesamtsystems unter **realen Einsatzbedingungen** gegen die vertraglich festgelegten **Abnahmekriterien** sowie die Kundenwünsche

Testfälle – Vergleich Modeltests und Systemtests

Modul-/ Komponententestfall

- Referenz zur Spezifikation der Testfälle: Schnittstellen-Spezifikation des Moduls und Quellcode
- *Im SEP:* ***UML-Klassendiagramm*** *und* ***Quellcode*** *(→ White-Box-Testen)*

Systemtestfall

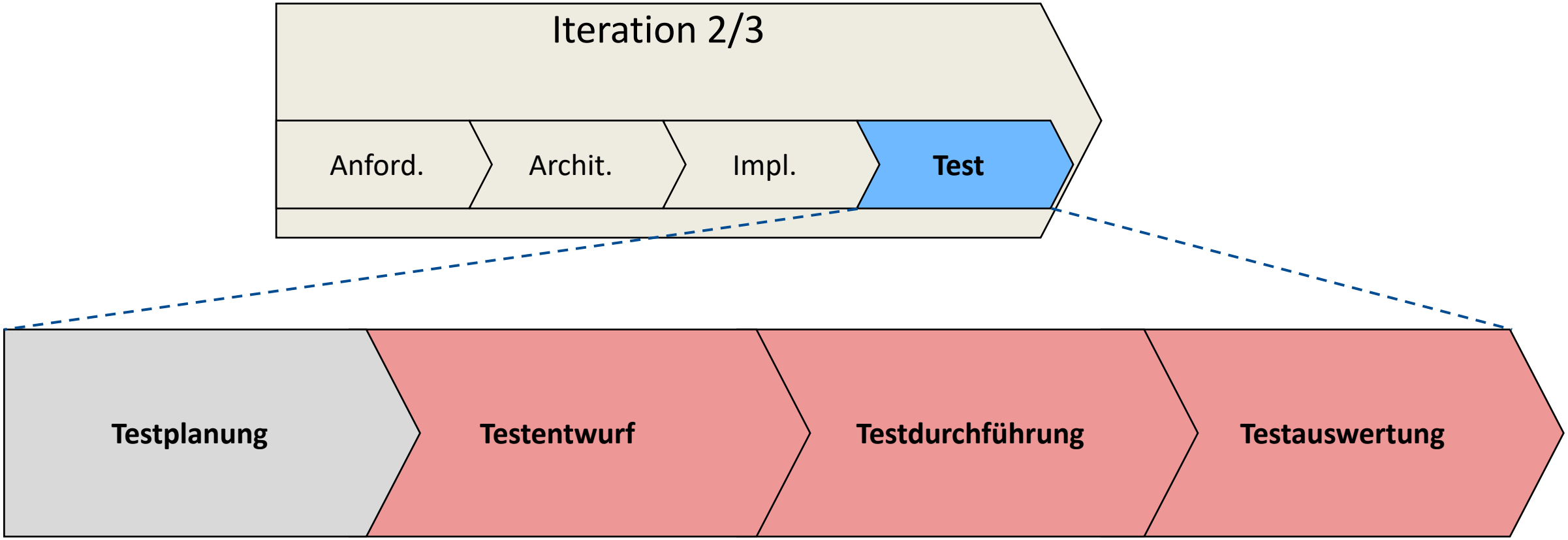
- Referenz zur Spezifikation der Testfälle: **Anforderungen** aus der Anforderungsspezifikation
- *Im SEP:* ***Szenarien + Aufgabenstellung***

Unabhängigkeit der Qualitätsprüfung

- Komponententests werden oftmals **vom Entwickler selbst** spezifiziert (und ausgeführt)
  - Beim **Test-Driven Development** bereits vor der Implementierung
  - Vorteil: Entwickler kennt das implementierte Produkt sehr genau
  - Dem stehen allerdings **wesentliche Nachteile** gegenüber:
    - Entwickler erkennen eigene Fehler meist nicht
    - Entwickler haben wenig Anreize, Fehler zu finden (bereitet ihnen mehr Arbeit zum Beheben)

- Daher wird häufig eine **personelle Trennung** zwischen Entwicklern und Testern vollzogen, um eine möglichst **objektive Prüfung** zu gewährleisten

Testprozess im SEP



Agenda

1. Einführung
2. **Modultestentwurf und Durchführung**
3. Testauswertung
4. Test-Suits

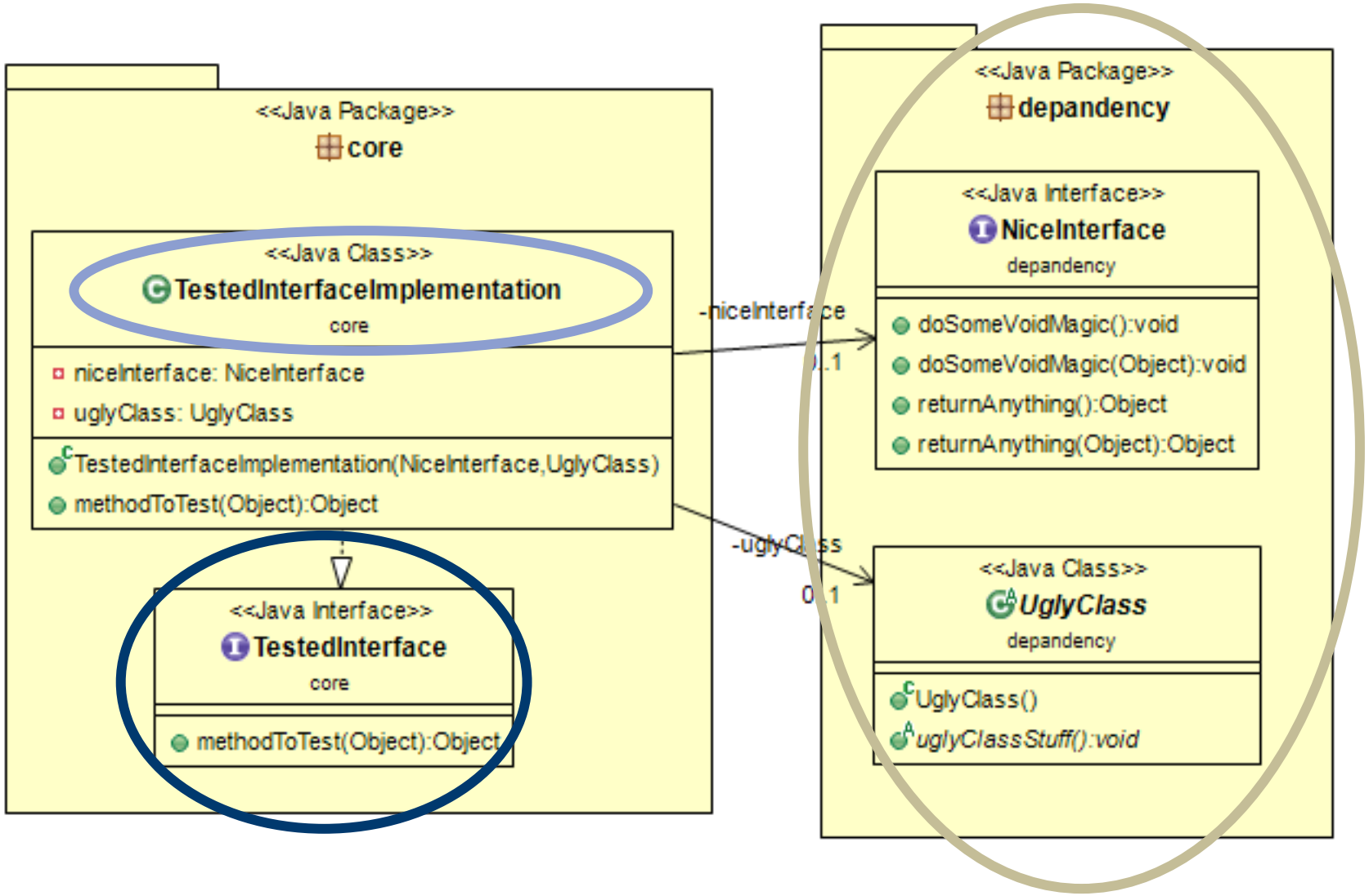
Modultestentwurf

- Erstellung von Modul**testfällen**
  - **Testreferenz**: Klassendiagramm und -dokumentation (User Stories und Szenarien), Quellcode

- Vorgehen:
  - Entscheidend ist das **Klassendiagramm**!
  - **Dokumentation** und **Anforderungen** sind als Präzisierung bzw. Begründungen des Klassendiagramms aufzufassen

  - Weichen diese in SEP von der Aufgabenstellung ab, so gilt es, **gegen die Aufgabenstellung zu testen**!

Ableitung von Modultestfällen



- Orientierung an implementierten **Interfaces**
- Identifikation der zu **testenden Klassen**
- Auflösen von **Abhängigkeiten** durch Verwendung von **Mockobjects**

Testdurchführung mithilfe von JUnit

- JUnit ist ein **Test-Framework** für Java-Code
- JUnit **automatisiert** Modultests (Unit-Tests)
- JUnit eignet sich für **Test-Driven-Development**
- Implementierung von Testfällen in Testklassen mit Hilfe von:
  - **Annotations** (@Before, @Test …):
    - Geben vor, wann und wie oft eine Methode aufgerufen wird
  - **Assertions** (void assertTrue(boolean condition)):
    - Implementieren Bestehenskriterien

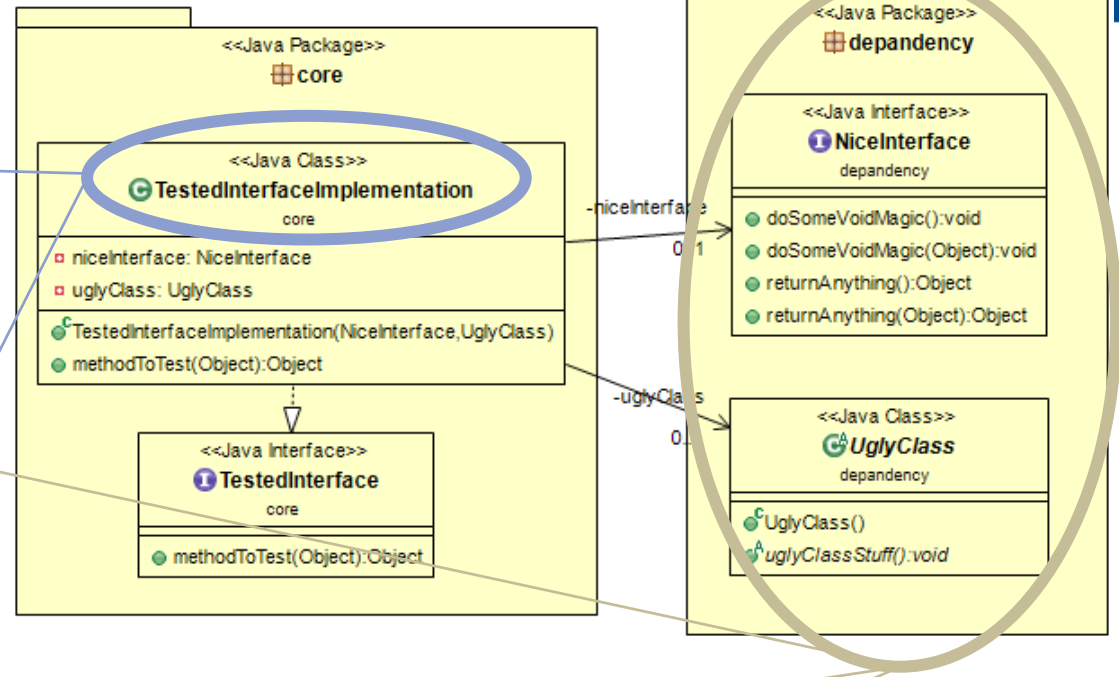
JUnit Testfall (1)


```
public class ManuallyMockedTest {

    NiceInterface niceInterface_Mock;
    UglyClass uglyClass_Mock;
    int[] methodsWereCalled;

    @Before
    public void setUp() throws Exception {
        niceInterface_Mock = new NiceInterface() {
        methodsWereCalled = new int[]{ 0, 0, 0, 0, 0 };
    }

    @After
    public void tearDown() throws Exception {
        for (int i = 0; i < methodsWereCalled.length; i++) {
            assertTrue(methodsWereCalled[i] == 1);
        }
    }

    @Test
    public void valid_UglyClass_Test() throws Exception {
        uglyClass_Mock = new UglyClass() {

        TestedInterface testedInstance = new TestedInterfaceImplementation(niceInterface_Mock, uglyClass_Mock);
        Object returnedObject = testedInstance.methodToTest(new Object());

        assertTrue(returnedObject.equals("404"));
    }
```

- **@Before/ @BeforeEach**: Wird vor jedem Testfall aufgerufen
- **@After/ @AfterEach**: Wird nach jedem Testfall aufgerufen
- **@Test**: Bezeichnet einen konkreten Testfall

JUnit Testfall (2)

```
public class ManuallyMockedTest {

    NiceInterface niceInterface_Mock;
    UglyClass uglyClass_Mock;
    int[] methodsWereCalled;

    @Before
    public void setUp() throws Exception {
        niceInterface_Mock = new NiceInterface() {
        methodsWereCalled = new int[]{ 0, 0, 0, 0, 0 };
    }

    @After
    public void tearDown() throws Exception {
        for (int i = 0; i < methodsWereCalled.length; i++) {
            assertTrue(methodsWereCalled[i] == 1);
        }
    }

    @Test
    public void valid_UglyClass_Test() throws Exception {
        uglyClass_Mock = new UglyClass() {

        TestedInterface testedInstance = new TestedInterfaceImplementation(niceInterface_Mock, uglyClass_Mock);
        Object returnedObject = testedInstance.methodToTest(new Object());

        assertTrue(returnedObject.equals("404"));
    }
```



JUnit Testfall (3)

```java
class DatabaseTest {

    static Database db;

    @BeforeAll
    public static void before()  {
        db = new Database();
        if (!db.doesUserExist( name: "JUnitTest"))
            db.addUser( username: "JUnitTest",  email: "JUnitTest@web.de",  pw: "JUnitTest");
    }

    @Test
    public void testSEP()  {
        db.updateSEP( username: "JUnitTest",  SEP: 1234);
        int test = db.loadSEP( username: "JUnitTest");
        assertEquals( message: "Der SEP-Wert wurde falsch abgespeichert.",  expected: 1234, test);
    }
```

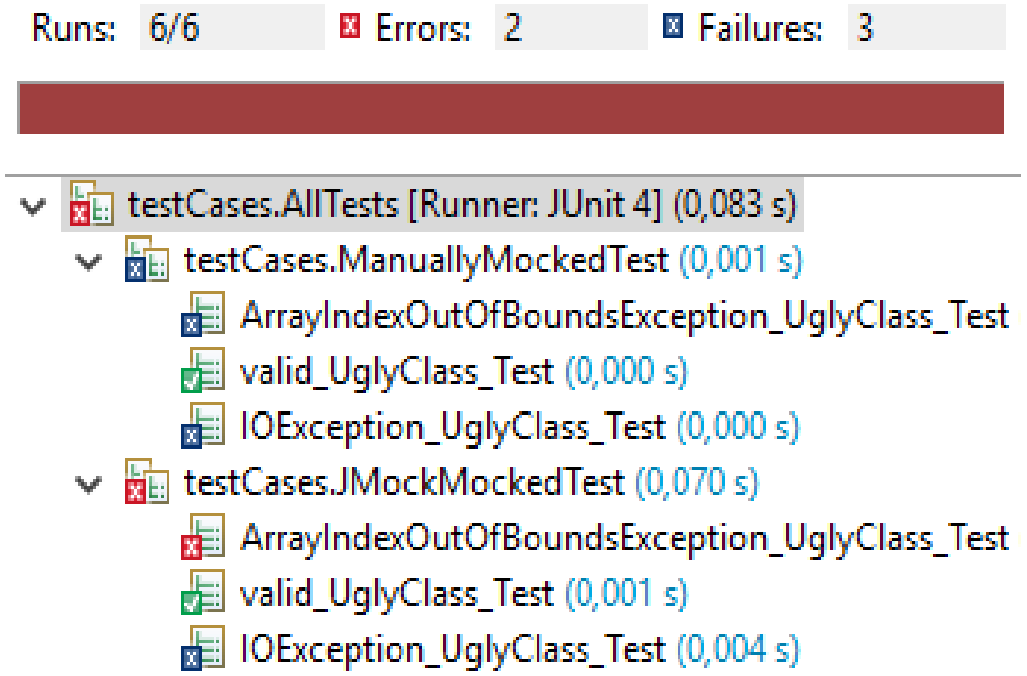
- **@BeforeAll:** Wird vor allen Testfällen einmalig ausgeführt (gut für zeitintensive Vorbedingungen)
- Beispiel: Testet zwei Datenbankmethoden zum schreiben und lesen in der Datenbank

- assertEquals(<Fehlermeldung>, <erwarteter Wert>, <tatsächlicher Wert>);
- Testet, ob zwei Werte identisch sind
- Bei Nicht-Erfüllung der Bedingung wird die Fehlermeldung ausgegeben und der Testfall scheitert

Agenda

1. Einführung
2. Modultestentwurf und Durchführung
3. **Testauswertung**
4. Test-Suits

JUnit Testergebnisse



Successes:

Der Test war erfolgreich. Alle Assertions und Expectations (JMock) **wurden erfüllt**.

Failures:

Der Test ist fehlgeschlagen. Einige Assertions oder Expectations wurden **nicht erfüllt**.

Errors:

Der Test ist fehlgeschlagen. Es wurde eine **Exception geworfen** die nicht abgefangen wurde (try-catch).

JUnit Testergebnisse - Detailinformationen

Runs: 6/6 Errors: 2 Failures: 3

- testCases.AllTests [Runner: JUnit 4] (0,083 s)
  - testCases.ManuallyMockedTest (0,001 s)
  - testCases.JMockMockedTest (0,070 s)
    - ArrayIndexOutOfBoundsException_UglyClass_Test (0,065 s)
    - valid_UglyClass_Test (0,001 s)
    - IOException_UglyClass_Test (0,004 s)

Failure Trace

```
java.lang.AssertionError: not all expectations were satisfied
expectations:
  expected once, already invoked 1 time: niceInterface.doSomeVoidMagic(); in sequence callSequence
  expected once, already invoked 1 time: niceInterface.doSomeVoidMagic(<java.lang.Object@5ba23b66>); in sequence callSequence
  ! expected once, never invoked: niceInterface.returnAnything(); in sequence callSequence; returns <404>
  ! expected once, never invoked: niceInterface.returnAnything(<404>); in sequence callSequence; returns "404"
  expected once, already invoked 1 time: uglyClass.uglyClassStuff(); throws <java.io.IOException: DANGER!>
what happened before this:
  niceInterface.doSomeVoidMagic()
  niceInterface.doSomeVoidMagic(<java.lang.Object@5ba23b66>)
  uglyClass.uglyClassStuff()

at org.jmock.lib.AssertionErrorTranslator.translate(AssertionErrorTranslator.java:20)
at org.jmock.Mockery.assertIsSatisfied(Mockery.java:199)
at org.jmock.integration.junit4.JUnitRuleMockery$1.evaluate(JUnitRuleMockery.java:50)
```

Agenda

1. Einführung
2. Modultestentwurf und Durchführung
3. Testauswertung
4. **Test-Suits**

Test-Suites

- **Testklassen** und ihre einzelnen Testfälle können zu **Test-Suites** zusammengefasst werden.
- Diese werden dann bei Teststart **komplett** ausgeführt.
- **Ergebnisse** werden nach Testklassen **geordnet** aufgeführt.

```
@RunWith(Suite.class)
@SuiteClasses({
    ManuallyMockedTest.class,
    JMockMockedTest.class
})

public class AllTests {

}
```

Vorteile von Test-Suites

- Das Gliedern von **Testklassen** und ihren einzelnen Testfällen mittels **Test-Suites** sorgt für:
  - **Strukturierung** der Test-Fälle in sinnvolle Abschnitte
  - zusammengefasstes Ausführen bestimmter Tests erspart Zeit und schafft **Übersicht** über die **Testklassen**
  - **Verwaltung** der Tests aus einer Klasse heraus
- **Fazit:** nützliches Tool, um Tests bei Projekten zu strukturieren und zu verwalten

Literatur

- Lehrbücher/Standards:
  - [Liggesmeyer 2009] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren Und Verifizieren Von Software*. 2nd ed., Spektrum Akademischer Verlag, 2009. Online verfügbar über die UB: https://link.springer.com/book/10.1007%2F978-3-8274-2203-3
  - [ISTQB 2021] ISTQB Glossary: https://glossary.istqb.org/app/de/search
  - [Roßner et al. 2016] C. Brandes, H. Götz, T. Roßner, M. Winter: Basiswissen modellbasierter Test, 2nd Edition, dpunkt, 2016. Online verfügbar über die UB: https://www.oreilly.com/library/view/basiswissen-modellbasierter-test/9781492019275/?ar
  - [Spillner & Linz 2014] A. Spillner, T. Linz: *Praxiswissen Softwaretest – Testmanagement, 4th Edition*. 4th ed. Dpunkt, 2014. Online verfügbar über die UB: https://www.oreilly.com/library/view/praxiswissen-softwaretest/9781492014942/?ar

Tutorials

- JUnit Homepage: http://www.junit.org/
- Tutorials:
  - http://www.tutego.de/blog/javainsel/2010/04/junit-4-tutorial-java-tests-mit-junit/
  - http://www.vogella.com/articles/JUnit/article.html
- JUnit 4 Cookbook: http://junit.sourceforge.net/doc/cookbook/cookbook.htm
- A Cook's Tour of Junit: http://junit.sourceforge.net/doc/cookstour/cookstour.htm
- Programmers Love Writing Tests: http://junit.sourceforge.net/doc/testinfected/testing.htm

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Vielen Dank für Ihre Aufmerksamkeit

PALUNO
The Ruhr Institute for Software Technology