Software-Qualitätssicherung (QS-VU)
LV 188.911

Einführung in Software Testen

Block 4 – Praktischer Teil

Wolfgang Gruber
Roland Breiteneder

Vienna University of Technology
Research Group for Industrial Software (INSO)

roland.breiteneder@inso.tuwien.ac.at

Inhalt

- **Einführung in den Software-Test**
- Blackbox Testen
- Test Driven Development und JUnit
- Test Doubles und Mock Object
- Kontrollflussorientierte Testverfahren

Stellenwert von Softwaretests

- Keine Software ist frei von Fehlern
- Testen von Software gehört zu den wichtigsten Aktivitäten des Software-Entwicklungsprozesses
- Systematisches Testen notwendig
- Eine gut geplante Testphase ist die Grundlage für ein erfolgreiches Projekt
- Ziel ist es, Fehler so früh wie möglich zu finden
- Softwaretests sind eine dynamische und produktorientierte Qualitätssicherungstechnik
- Aufgrund wachsender Komplexität und hohe Abhängigkeiten von Softwaresystemen, wird der Softwaretest strategisch immer bedeutender

Ziele des Softwaretests

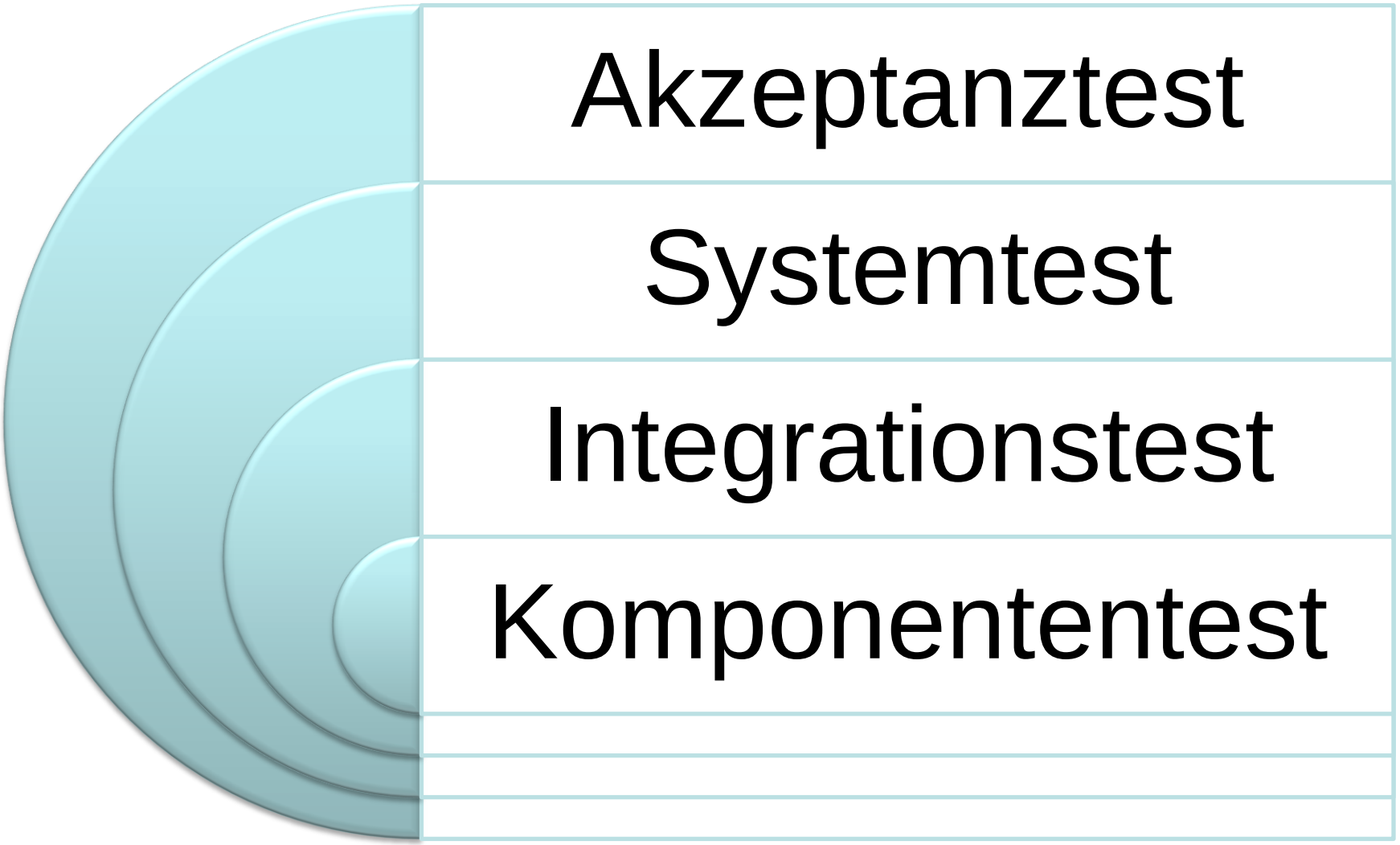
1. Ziel ist es, Testfälle zu identifizieren, mit denen die höchste Wahrscheinlichkeit gegeben ist, festzustellen, ob das Softwaresystem korrekt funktioniert
2. Ein guter Test ist jener, der eine möglichst hohe funktionale oder nichtfunktionale Abdeckung hat
3. Beim Softwaretest sollen Fehler von Programmen identifiziert werden.
4. Wenn ein Test einen Fehler gefunden hat, war er erfolgreich

Test- und Integrationsstufen

- Strategie der Zerlegung der gestellten Aufgabe in beherrschbare Teile bei der Herstellung von Softwaresystemen

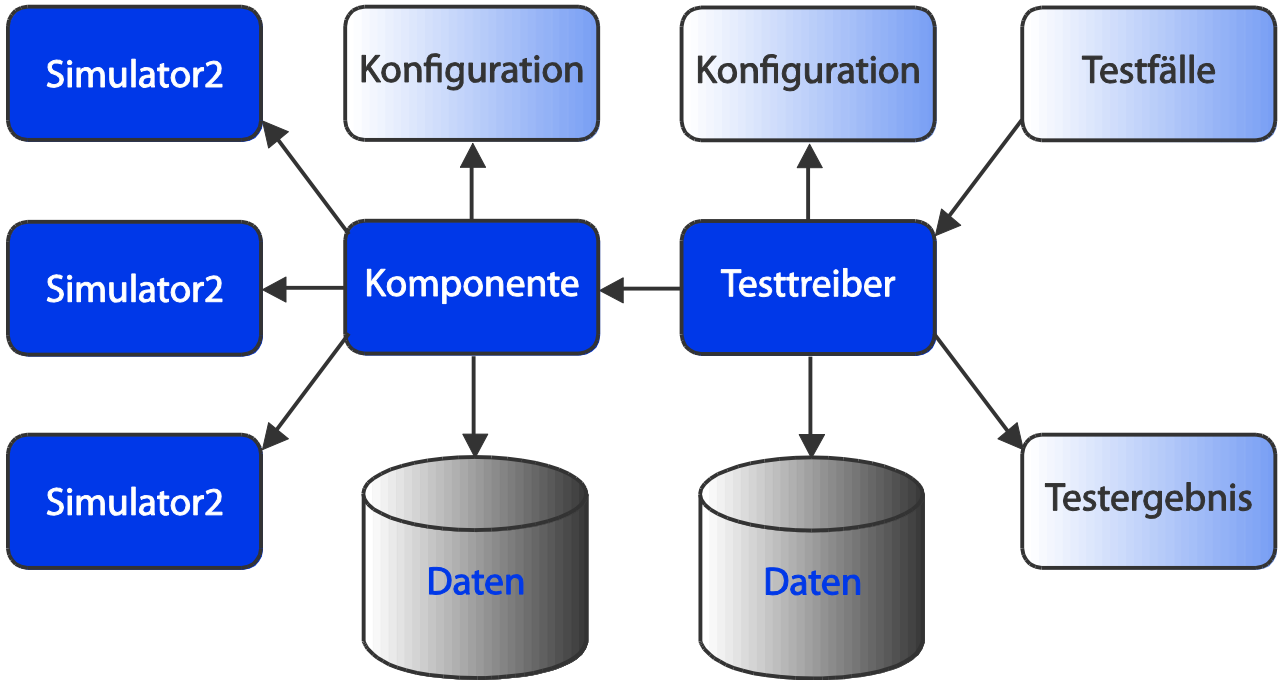
- Zerlegung in Teststufen ermöglicht eine frühe Prüfung von unterschiedliche Teilen des zu entwickelnden Systems

Teststufen


Akzeptanztest
Systemtest
Integrationstest
Komponententest

Komponententest

- Prüfung von separat testbaren Komponenten (z.B. Modulen, Programmen, Objekten, Klassen) auf vorhandene Fehler
- Isolation der einzelnen Komponenten vom Rest des Systems mit Hilfe von Platzhaltern (Stubs) bzw. Simulatoren und Testtreiber



Integrationstest

- Prüft die Schnittstellen zwischen Komponenten
- Es können mehrere Integrationsstufen zum Einsatz gelangen, wobei diese Testobjekte unterschiedlichster Größe betreffen können
- Mit der Größe des Integrationsumfangs wächst auch die Schwierigkeit der Isolation von Fehlerwirkungen in Komponenten oder Systemen
- Inkrementelle Integrationsstrategien vs. Big-Bang-Strategie
- Können auf der Systemarchitektur, auf funktionalen Aufgaben, Transaktionsverarbeitungssequenzen oder weiteren Aspekten des Systems oder seiner Komponenten basieren
- Unterscheidung zwischen horizontaler und vertikaler Integration

Systemtest

- Fokus liegt im spezifizierten Verhalten eines Gesamtsystems oder eines Produktes
- Systemtests sollten funktionale und nichtfunktionale Anforderungen an das System abdecken
- Systemtests basieren auf:
  - Anforderungsspezifikationen
  - Anwendungsfällen oder sonstigen Beschreibungen eines Systems
  - Geschäftsprozessen
  - Risikoanalysen
  - Erfahrungen im Produktionsumfeld
  - Systemressourcen

Akzeptanztest

- Fokus des Abnahmetests liegt darin, die Erbringung der vom Auftraggeber und Auftragnehmer vereinbarten Leistungen nachzuweisen
- Wird meist von Kunden oder Benutzern eines Systems durchgeführt
- Arten von Abnahmetests:
  - Anwender-Abnahmetest
  - Betrieblicher Abnahmetest
  - Regulatorischer und vertraglicher Abnahmetest
  - Alpha- und Beta-Test (oder Feldtest)

Testabdeckung

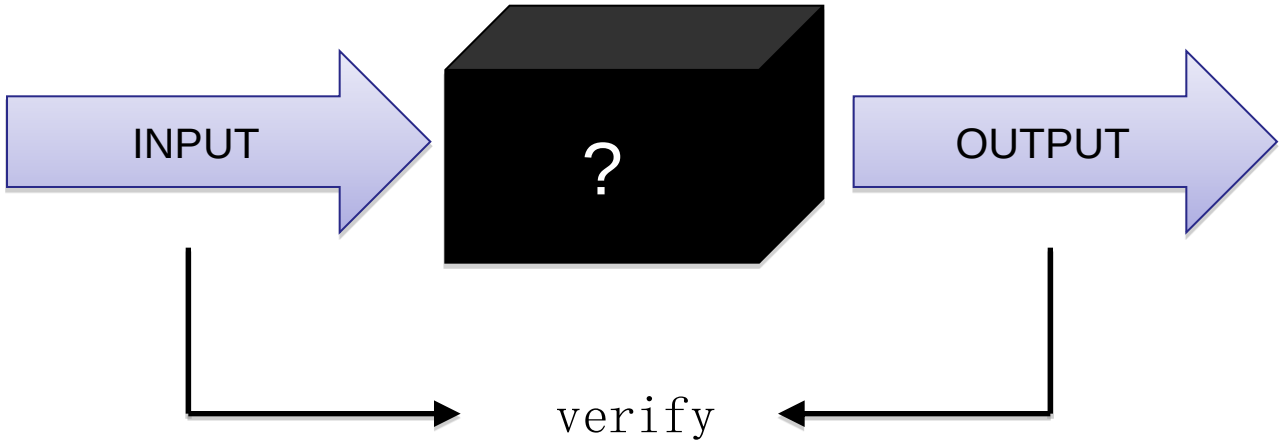
- Ziel der Verifikation von Softwaresystemen ist eine möglichst hohe Abdeckung (Coverage) des Systems mit den entsprechenden Testfällen
- Testfallselektion notwendig, da eine vollständige Aufzählung aller möglichen Systemzustände aufgrund der Komplexität von Systemen unmöglich ist
- Ziel ist es, mittels formeller und deterministischer Verfahren die Menge an Testfällen zu identifizieren, die mit höchster Wahrscheinlichkeit Fehler finden bzw. die größtmögliche Abdeckung erreichen
- Zwei Ansätze von Abdeckung: strukturelle Abdeckung (White-Box-Tests) und funktionale Abdeckung (Black-Box-Tests)

Inhalt

- Einführung in den Software-Test
- **Blackbox Testen**
- Test Driven Development und JUnit
- Test Doubles und Mock Object
- Kontrollflussorientierte Testverfahren

Blackbox Testen

- Testfälle basieren auf Spezifikation
- Funktionales Testen
- Wissen über innere Struktur wird bewusst ignoriert
- Daten-getriebene, Input-Output-getriebene Tests
- Partitionierung der Eingabedaten (Äquivalenzklassen, Grenzwerte)



Funktionale Methoden

- Äquivalenzklassenanalyse
  - Einteilung möglicher Ein- und Ausgabewerte in Klassen gleichem Systemverhalten
  - Es werden Werte von Klassen gewählt, bei denen angenommen wird, dass Fehler auftreten, die auch bei allen anderen möglichen Werten der Klasse auftreten können
  - Es gibt gültige und ungültige Äquivalenzklassen
- Grenzwertanalyse
  - Spezialfall der Äquivalenzklassenanalyse
  - Tatsache, dass Fehler besonders oft an den Grenzen der Äquivalenzklassen auftreten
  - Identifikation der Werte, die die Grenze zwischen den Äquivalenzklasse bilden

Beispiel Äquivalenzklassen- und Grenzwertanalyse

- Ein Eingabefeld erlaubt die Eingabe von Integer-Werten. Gültige Werte sollen zwischen 15 und 25 liegen

- Wie lauten die ungültigen und die gültige Äquivalenzklasse?
- Welcher der nachfolgenden Werte fällt in eine ungültige Äquivalenzklasse?

  a) 21
  b) 13
  c) 18
- Wie lauten die Grenzwerte der Äquivalenzklassen?

Beispiel Äquivalenzklassen- und Grenzwertanalyse - Lösung

- Klasse I: Werte < 15 => ungültige Klasse
  Klasse II: 15 bis 25 => gültige Klasse
  Klasse III: Werte > 25 => ungültige Klasse

- a) 21 => Klasse II => gültige Klasse
  b) 13 => Klasse I => ungültige Klasse
  c) 18 => Klasse II => gültige Klasse

- Klasse I: 14, 15, 16
- Klasse II: 14, 15, 16 und 24, 25, 26
- Klasse III: 24, 25, 26

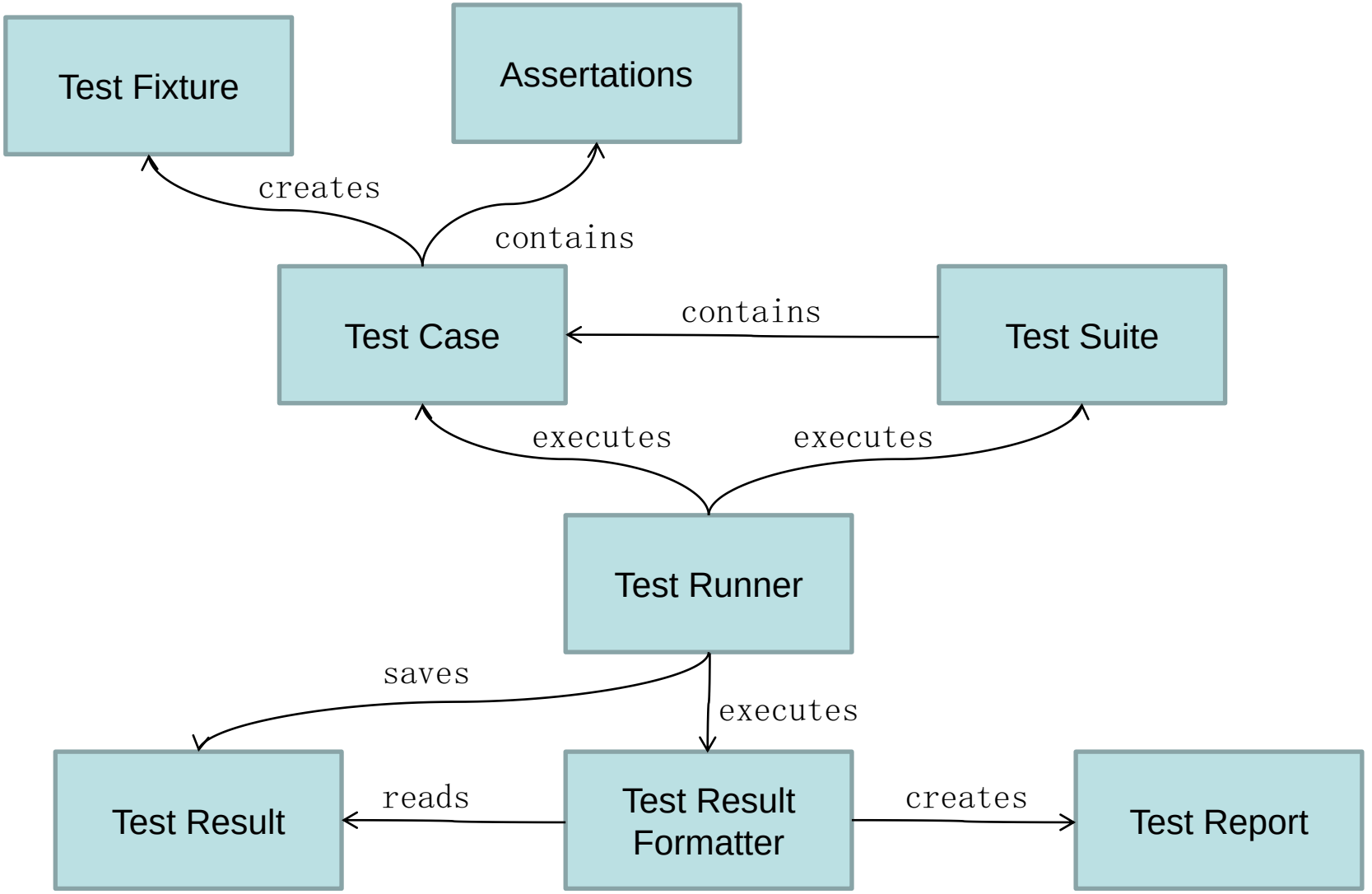
Inhalt

- Einführung in den Software-Test
- Blackbox Testen
- **Test Driven Development und JUnit**
- Test Doubles und Mock Object
- Kontrollflussorientierte Testverfahren

xUnit Frameworks

- Überbegriff für Unit-Test-Frameworks in den verschiedenen Programmiersprachen mit ähnlichem Design
- Urahn ist SUnit von Kent Beck
- Verbreitung durch JUnit von Kent Beck & Erich Gamma
- Test phases
  - Set up
  - Exercise
  - Verify
  - Tear down

xUnit Frameworks - Bestandteile



JUnit 3.X

- Verwendung von Namenskonventionen und Reflection

```
public class XXXTest extends junit.framework.TestCase {
    public void setUp() { }
    public void testXXX() {          }
    public void tearDown() { }

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(XXXTest.class);
        return suite;
    }
}
```

JUnit 4.X I

- Verwendung von Annotationen (ab Java SE 5)

```
public class XXXTest {
    @Before public void init() { }
    @After public void cleanup() { }
    @Test public void testXXX() { }
}

@RunWith(Suite.class)
@SuiteClasses({ XXXTest.class })
public class XXXSuite() { }
```

JUnit 4.X II

```
@RunWith(JUnit4.class)
public class XXXTest {

    @Ignore("Can work due to bug #52893")
    @Test
    public void testXXX() {
    }
}
```

Benennungsschemata für Testmethoden

- Einheitliches Benennungsschemata für Testmethoden bewirken lesbare Logfiles!
- Beispiel nach „xUnit Test Patterns, Gerard Meszaros, 2007“:

```
@Test
public void <<UseCase/Method>>_should<<ExpectedPostState>>

@Test
public void ascendingOrder_shouldReturnElementsInOrder()
```

Lifecycle-Methoden

- Lifecycle-Methoden
  - @BeforeClass
  - @AfterClass
  - @Before
  - @After
- @BeforeClass & @AfterClass auf statischen Methoden!

```
@BeforeClass
public static void init() { }

@After
public void cleanup() { }
```

JUnit Test Lifecycle

@BeforeClass
    // Erste Testmethode
    Constructor
    @Before
        @Test
    @After
    // Zweite Testmethode
    Constructor
    @Before
        @Test
    @After
@AfterClass

Assertations I

- Zusicherungen zur Überprüfung, ob das zu testende Objekt den korrekten Zustand besitzt
- Statische Methoden der Klasse org.junit.Assert
- Können mittels statischem Import importiert werden
- Achtung: assert() ist ein Keyword von Java und hat eine andere Bedeutung
- fail() lässt Test fehlschlagen

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

assertXXX(errorMessage, expectedValue, actualValue);
assertEquals(„First name does not match“, „John“,
    person.getFirstName());
```

Assertations II

- Hamcrest stellt spezielle Matcher mit Fluent Interface bereit
- Neuere JUnit-Versionen nutzen Hamcrest-Kern-Bibliothek

Beispiele:

```
assertThat(„John“, is(EqualTo(person.getFirstName())));
assertThat("Hello", is(allOf(notNullValue(), instanceOf(String.class),
    equalTo("Hello"))));
assertThat("Hello", is(anyOf(nullValue(), instanceOf(String.class),
    equalTo(„World"))));
assertThat("Hello", is(not(instanceOf(Integer.class))));
assertThat("Hello", is(notNullValue()));
```

Testen von Exceptions

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(5);
}

@Test public void expectException() {
    try {
        executeCodeThatthrowsException();
        fail(message);
    } catch (Exception e) {
        assertXXX(e);
}
```

Parameterized Tests I

- Tests werden mehrmals mit unterschiedlichen Testdaten aufgerufen
- Injection von Werten in public Fields oder in den Konstruktor
- Klasse muss mit @RunWith(Parameterized.class) annotiert werden

Definition der Testdaten:

```
@Parameters
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
            { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 },{ 6, 8 }
    });
}
```

Parameterized Tests II

```
@Parameter
public int input;

@Parameter(value = 1)
public int expected;

@Test
public void test() {
    assertEquals(expected, testObject.compute(input));
}
```

Categories I

- Einteilung von Tests in Kategorien mit @Category auf Klassen oder Methoden
- Testlauf kann auf einzelne Kategorie eingeschränkt werden

```
public class A {
  @Test
  public void a() {
    fail();
  }

  @Category(SlowTests.class)
  @Test
  public void b() {
  }
}
```

Categories II

```
public interface FastTests { }
public interface SlowTests { }

@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@ExcludeCategory(FastTests.class)
@SuiteClasses( { A.class })
public class SlowTestSuite {
    // nur A.b wird ausgeführt
}
```

Rules I

- Mächtiger Interceptor-Mechanismus
- Modifikation von Test Cases vor deren Ausführung

```
public class XXXTest {
    @Rule MyRule rule = new MyRule();
}

public class MyRule implements MethodRule {
    @Override
    public Statement apply(Statement stmt, FrameworkMethod method,
    Object obj) {
        return new MyStatement(stmt, method, obj);
    }
```

Rules II

```
public class MyStatement extends Statement {
    public MyStatement(Statement stmt, FrameworkMethod method,
    Object testObject) {
        // setting values
    }

    @Override
    public void evaluate() throws Throwable {
        // modify testObject
        this.stmt.evaluate();
    }
}
```

Testreihenfolge

- Tests sollten unabhängig voneinander laufen => Reihenfolge prinzipiell beliebig
- Ab JUnit 4.11 spezielle Annotation @FixMethodOrder auf Test-Klasse

@FixMethodOrder(MethodSorters.DEFAULT)

@FixMethodOrder(MethodSorters.JVM)

@FixMethodOrder(MethodSorters.NAME_ASCENDING)

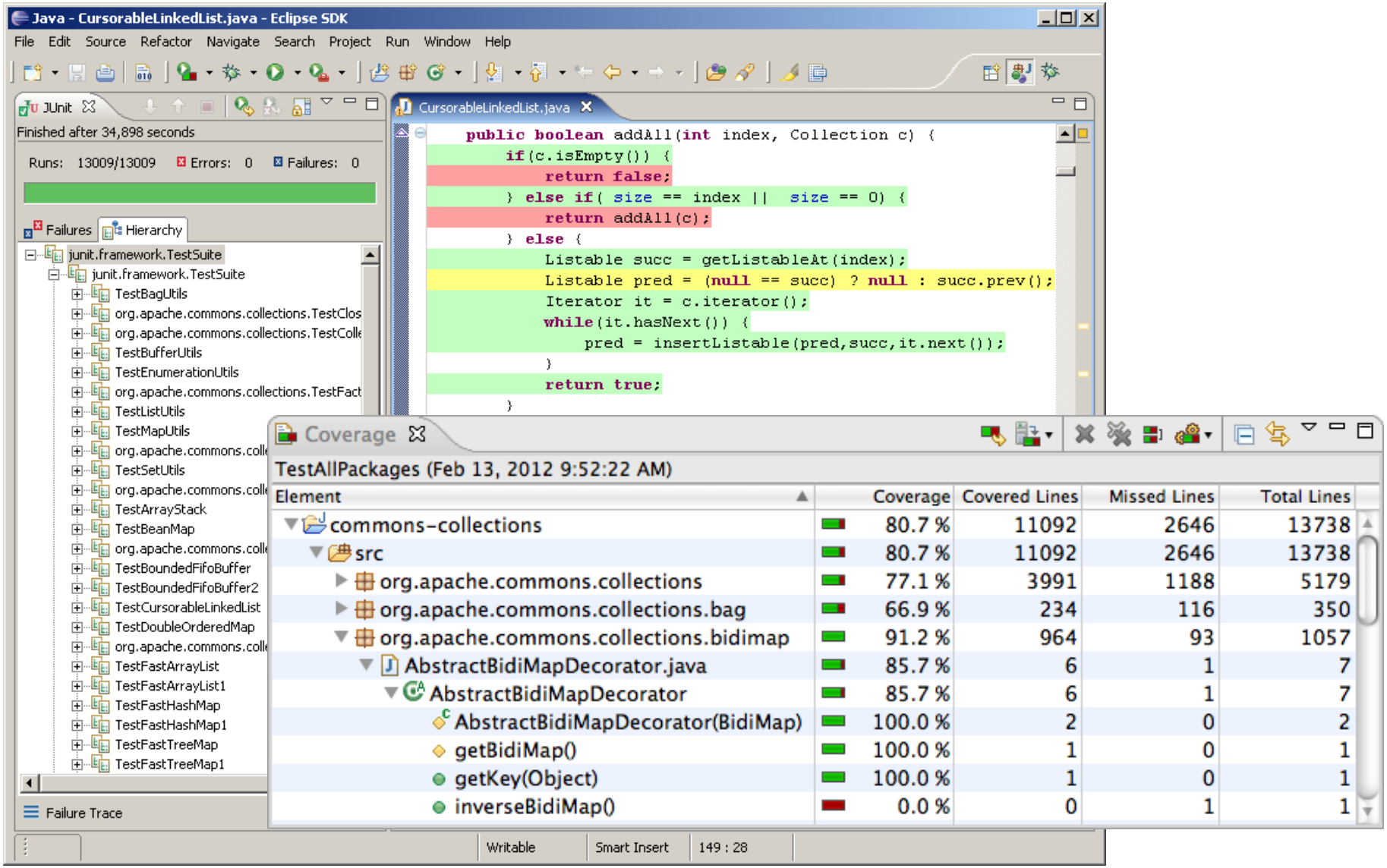
Best Practices

- Jeder Testfall testet nur einen Aspekt bzw. Zustand
- Klassen und Testklassen liegen in unterschiedlichen Verzeichnisstrukturen, aber im selben Package
- Definition & Einhaltung von Namenskonventionen
- Klare Fehlermeldung, wenn Vergleich innerhalb von Assertations nicht stimmen
- Gutes Design ist gut testbar (vgl. Dependency Injection) => Refactoring für Tests
- Keine Abhängigkeiten zwischen Testfällen
- Jeder Testfall initialisiert Umgebung und räumt sie wieder auf
- Keine Logik in Test-Doubles
- Eigene Logik testen, nicht fremde Klassen (Bibliotheken)

Bad Practices

- Zu lang, zu viele Asserts in einem Test.
  - Mehrere Tests in einer Testmethode.
- Zu viele gleiche Asserts in verschiedenen Tests
  - Status eines Objekts wird in verschiedenen Testmethoden genau gleich abgeprüft.
- Bedingte Ausführung von Tests
  - Verzweigungen (if/else, switch, Schleifen) sind von Vorbedingungen abhängig
  - Vorbedingungen des SUT ändert die Ausführung des Tests, manche Testfälle werden vielleicht gar nicht ausgeführt!
- Random Logic
  - Statt Äquivalenzklassen zu bilden werden einfach Zufallsdaten verwendet
  - unvorhersehbare Testergebnisse!
- Überprüfung der Vorbedingungen in jedem Test
  - Z.B. Überprüft jeder Test die Aktionen welche in der setUp bzw. @Before annotierten Methode gesetzt wurden.

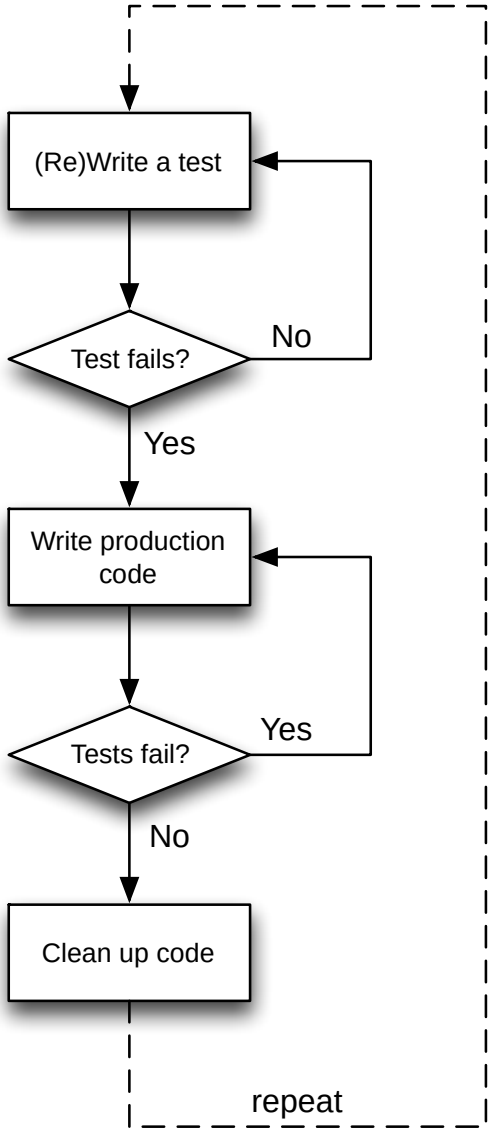
Test Coverage – Beispiel EclEmma



Beispiel: Testen eines Wörterbuchs mit TDD

- Dictionary
  - Enthält Wörter ohne Duplikate
  - Verwaltet Elemente in aufsteigender Reihenfolge
  - Hinzufügen von Wörter: `add`
  - Operationen auf das erste bzw. das letzte Element: `first`, `last`
  - Abrufen eines Elements: `get`
  - Anzahl der Wörter: `size`

Test Driven Development (TDD)



0. Dictionary Klasse vorbereiten

- Erstellen der Dictionaryklasse mit Methodenköpfe
  - Code muss kompilierbar sein
- Eclipse:
  - neues Javaprojekt anlegen
  - Dictionary Klasse erstellen

```
public class Dictionary {

    private SortedSet<String> wordlist;

    public Dictionary(){
        wordlist = new TreeSet<String>();
    }

    public void add(String w){
    }

    public String first(){
        return null;
    }

    public String last(){
        return null;
    }

    public String get(Integer pos){
        return null;
    }

    public Integer size(){
        return null;
    }
}
```

1. Write a Test

- Erstellen einer Testklasse mit einem Testfall
- Eclipse:
  - New „JUnit Test Case“
  - Ausführen mit „Run as Junit Test“
- Test muss mangels Implementierung fehlschlagen!

```
public class DictionaryTest {

    private Dictionary dictionary;

    private String wort1 = "Hallo";
    private String wort4 = "Welt";
    private String wort2 = "INSO";
    private String wort3 = "QSE";


    @Before
    public void setUp(){
        dictionary = new Dictionary();
    }

    @After
    public void tearDown(){
        dictionary = null;
    }

    @Test
    public void add_shouldContainAddedWord(){
        dictionary.add(wort1);
        assertEquals(wort1, dictionary.get(0));
    }
}
```

2. Write code

- Programmierung der Methoden
  - add
  - get
- Test sollte jetzt erfolgreich durchlaufen!

```
public class Dictionary {

    private SortedSet<String> wordlist;

    public Dictionary(){
        wordlist = new TreeSet<String>();
    }

    public void add(String w){
        wordlist.add(w);
    }

    public String get(Integer pos){
        int counter = 0;
        for(String word: wordlist){
            if(counter == pos)
                return word;
            counter ++;
        }
        return null;
    }

    public String first(){
        return null;
    }

    public String last(){
        return null;
    }

    public Integer size(){
        return null;
    }
}
```

3. Cleanup

- Checkin/Commit in lokales GIT Repository
- Optionales Refactoring
- Tests müssen weitherin erfolgreich durchlaufen!
- Wiederhole Schritte 1 bis 3...

Inhalt

- Einführung in den Software-Test
- Blackbox Testen
- Test Driven Development und JUnit
- **Test Doubles und Mock Object**
- Kontrollflussorientierte Testverfahren

Test Doubles

- Dummy Object
  - Ohne Funktionalität, dienen als “leere” Methodenparameter
- Fake Object
  - Ausführbare Implementierung, liefern keine Echtdaten,
    z.B.: Emulation einer Datenquelle zur rascheren Testausführung
- Stub
  - Liefert vordefinierte Werte an den Aufrufer.
- Mock
  - Liefert vordefinierte Werte an den Aufrufer und überprüft im Gegensatz zum Stub die bekommenen Übergabeparameter und wird somit Teil des Tests.

Mock Objekt

- Nachahmung von Funktionalitäten realer Objekte, wenn das reale Objekt z.B.:
  - zum Testzeitpunkt nicht verfügbar ist
  - nicht deterministische Ergebnisse liefert (z.B. Datum/Uhrzeit)
  - langsam oder noch nicht vorhanden ist
  - schwer in einen bestimmten Status zu setzen ist

- Mocking Frameworks
  - Jmock (Java)
  - Mockito (Java)
  - EasyMock (Java)
  - PowerMock (Java)
  - NMock3 (.Net)
  - EasyMock.NET (.Net)

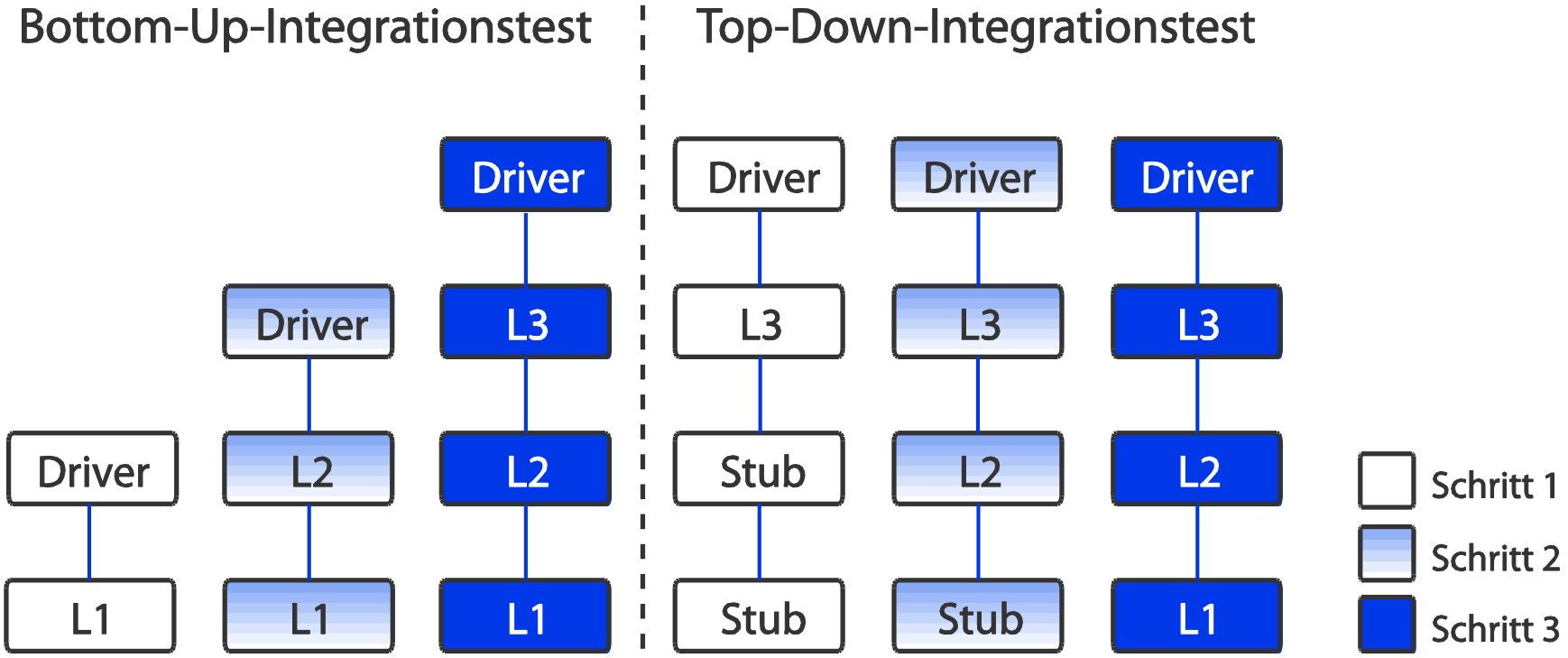
Beispiel Mockito

```
// Direkte Initialisierung
MyClass c = Mockito.mock(MyClass.class);

// Verwendung MockitoJUnit4Runner
@Mock MyClass c

c.computeSomething().willReturn(expectedValue);

c.computeSomething().times(2);
c.computeSomething().doSelfCheck();
```

Achtung bei Einschränkungen von Mockito:
finale Klassen, finale Methoden, Enums, statische Methoden, usw

Integrationstest cont.

- Bottom-Up-Integrationstests vs. Top-Down-Integrationstests



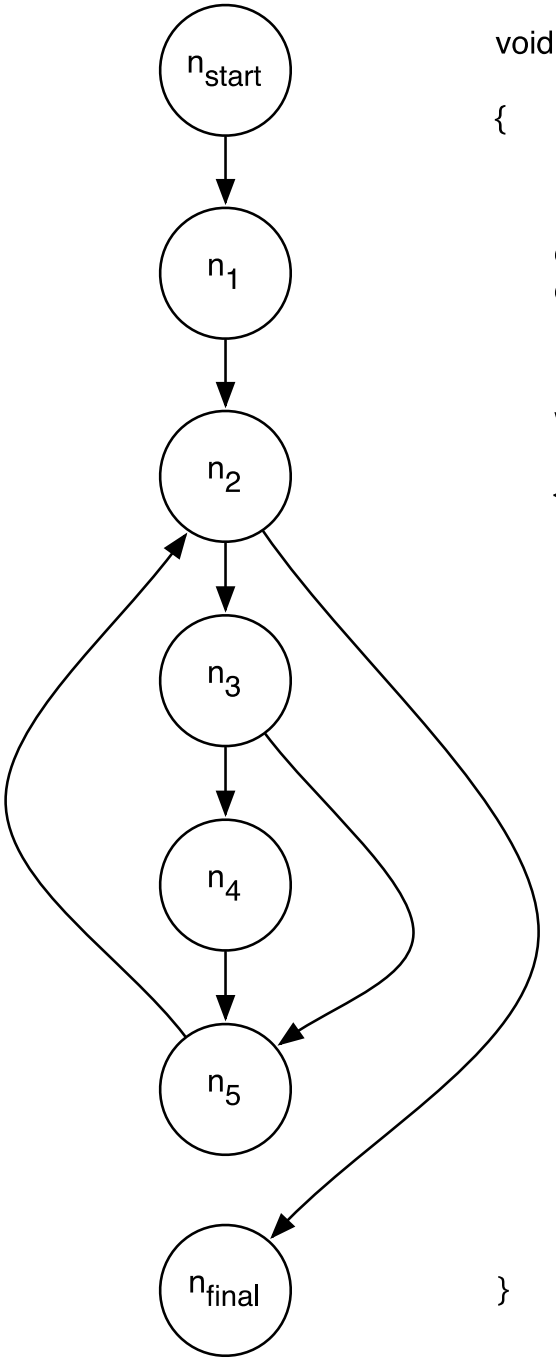
Inhalt

- Einführung in den Software-Test
- Blackbox Testen
- Test Driven Development und JUnit
- Test Doubles und Mock Object
- **Kontrollflussorientierte Testverfahren**

Kontrollflussorientierte Testverfahren (Überdeckungstests)

- Strukturorientierte Testmethoden
  - basieren auf Quellcode (White Box Test)
- Orientieren sich am Kontrollflussgraphen des Programms
- Relevant auf Unit Test Ebene
- Definieren keine Regeln für die Erzeugung von Testfällen
- Testarten:
  - Anweisungsüberdeckungstest (C_0)
  - Zweigüberdeckungstest (C_1)
  - Bedingungsüberdeckungstest (C_2)
  - Pfadüberdeckungstest (C_3)

Kontrollflussgraph



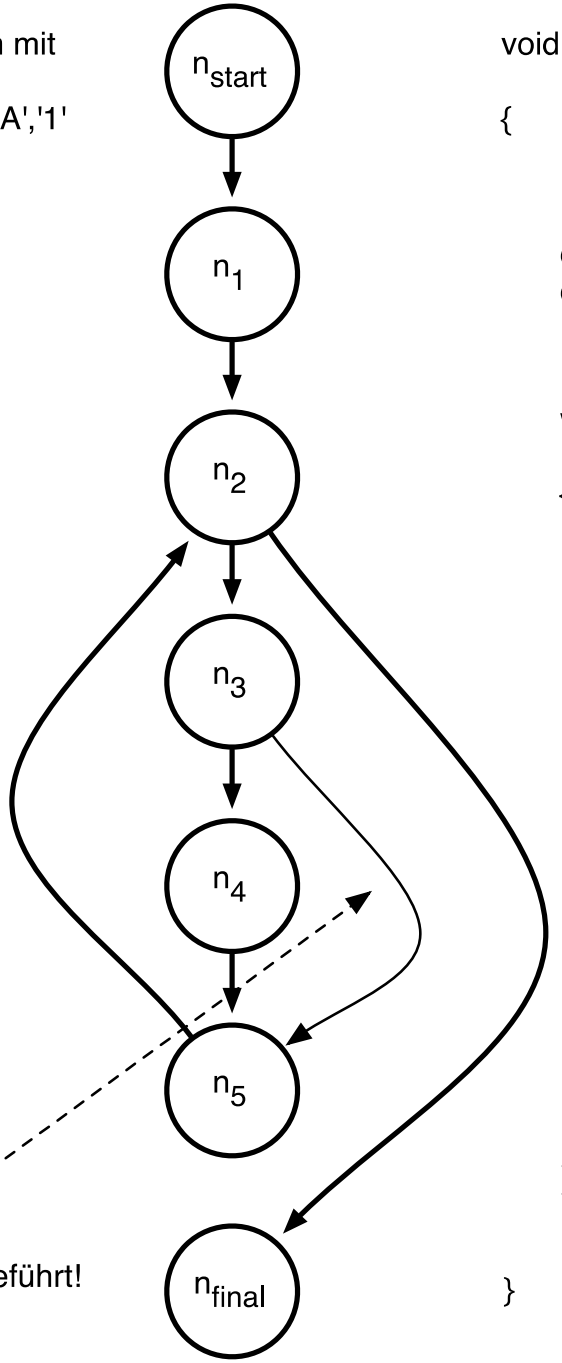

```
void ZaehleZchn (int& vokalAnzahl,
                 int& gesamtZahl)
{


    char zchn;
    cin >> zchn;


    while((zchn>='A') && (zchn<='Z')
          && (gesamtZahl<INT_MAX))
    {
          gesamtZahl++;


          if((zchn=='A')||(zchn=='E')
            ||(zchn=='I')||(zchn=='O')
            ||(zchn=='U'))
          {



                vokalAnzahl++;


          }


          cin >> zchn;


    }

}
```

Anweisungsüberdeckungstest (C_0)

- Ziel: Abdeckung aller Knoten des Kontrollflussgraphen
- 100% Anweisungsüberdeckung erreicht, wenn sämtliche Anweisungen mindestens einmal durchlaufen werden
- Bietet die Möglichkeit nicht ausführbare Anweisungen (toten Code) aufzuspüren
- Zu schwaches Kriterium für sinnvolle Testdurchführung
  - Auftreten der Fehlerwirkung kann an die Ausführung mit bestimmten Testdaten gekoppelt sein

Kontrollflussgraph – Anweisungsüberdeckungstest (C_0)

Aufruf von ZaehleZchn mit
gesamtZahl=0
eingelesene Zeichen: 'A','1'



Zweig (n_3,n_5) wird nicht notwendigerweise ausgeführt!

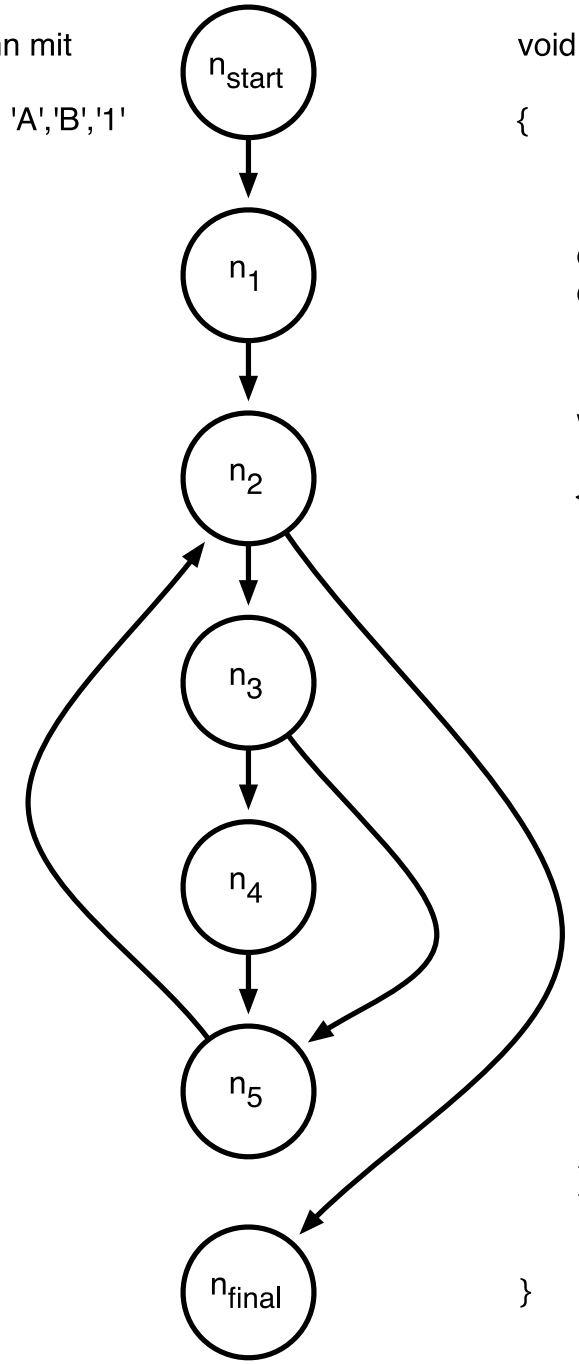
```
void ZaehleZchn (int& vokalAnzahl,
                 int& gesamtZahl)
{


    char zchn;
    cin >> zchn;


    while((zchn>='A') && (zchn<='Z')
          && (gesamtZahl<INT_MAX))
    {
          gesamtZahl++;


          if((zchn=='A')||(zchn=='E')
            ||(zchn=='I')||(zchn=='O')
            ||(zchn=='U'))
          {



                vokalAnzahl++;


          }


          cin >> zchn;


    }


}
```

Zweigüberdeckungstest (C_1)

- Ziel: Abdeckung aller Kanten des Kontrollflussgraphen
- Zweigüberdeckungstest subsumiert den Anweisungsüberdeckungstest
- Gilt als Minimalkriterium im kontrollflussorientierten Testen
- Spürt nicht ausführbare Programmzweige auf
- Besonders oft durchlaufene Programmteile können erkannt und gezielt optimiert werden
- Im Gegensatz zum Anweisungsüberdeckungstest muss jede Entscheidung mindestens einmal true und false annehmen
- Problematik:
  - Unzureichender Test von Schleifen (ein Durchlauf genügt)
  - Kompliziert aufgebaute Entscheidungen werden nicht berücksichtigt

Kontrollflussgraph – Zweigüberdeckungstest (C_1)

Aufruf von ZaehleZchn mit
gesamtZahl=0
eingelesene Zeichen: 'A','B','1'




```
void ZaehleZchn (int& vokalAnzahl,
                 int& gesamtZahl)
{

    char zchn;
    cin >> zchn;

    while((zchn>='A') && (zchn<='Z')
          && (gesamtZahl<INT_MAX))
    {
          gesamtZahl++;

          if((zchn=='A')||(zchn=='E')
            ||(zchn=='I')||(zchn=='O')
            ||(zchn=='U'))
          {

                vokalAnzahl++;

          }

          cin >> zchn;

    }

}
```

Bedingungsüberdeckungstest (C_2)

- Grundidee: Gründliche Überprüfung zusammengesetzter Entscheidungen, Teilentscheidungen
- Einfacher Bedingungsüberdeckungstest
  - Subsumiert nicht C_0 und C_1
  - Fordert den Test aller atomaren Teilentscheidungen gegen true und false
- Mehrfach-Bedingungsüberdeckungstest
  - Subsumiert C_1
  - Fordert den Test aller Wahrheitswertekombinationen der atomaren Teilentscheidungen
- Weitere Ausprägung
  - Minimaler Mehrfach-Bedingungsüberdeckungstest

Beispiel - Einfacher Bedingungsüberdeckungstest (C_2)

- Gegeben sei folgende Entscheidung:
  (((u == 0) || (x > 5)) && (y < 6) || (z == 0)))
- Kann abkürzend wie folgt geschrieben werden
  ((A || B) && (C || D))
- Sind u, x, y und z unabhängig, so können A, B, C und D unabhängig voneinander true oder false werden
- Einfacher Bedingungsüberdeckungstest kann mit zwei Testfällen erreicht werden:
  - A=f, B=f, C=f, D=f
  - A=t, B=t, C=t, D=t

Beispiel – Mehrfach-Bedingungsüberdeckungstest (C_2)

- Gegeben sei folgende Entscheidung:
  (((u == 0) || (x > 5)) && (y < 6) || (z == 0)))
- Kann abkürzend wie folgt geschrieben werden
  ((A || B) && (C || D))
- Sind u, x, y und z unabhängig, so können A, B, C und D unabhängig voneinander true oder false werden
- Ergibt 16 Wahrheitswertekombinationen (2^n => 2^4)

Pfadüberdeckungstest (C_3)

- Fordert die Ausführung aller unterschiedlichen Pfade der Software
- Ein Pfad ist eine Sequenz von Knoten
- Für reale Softwaremodule meist nich durchführbar, da sie eine unendlich hohe Anzahl von Pfaden besitzen können
- Bei endlicher Anzahl von Pfaden ist der Aufwand meist trotzdem zu hoch

Literaturreferenzen

- Biffl Stefan, Winkler Dietmar, Frast Denis: „Qualitätssicherung, Qualitätsmanagement und Testen in der Softwareentwicklung“, Skriptum zur Lehrveranstaltung, 2004. http://qse.ifs.tuwien.ac.at/courses/skriptum/script.htm
- Thomas Grechenig, Mario Bernhart, Roland Breiteneder, Karin Kappel: „Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten“, Pearson Studium, 2009
- Software Engineering Body of Knowledge, http://www.swebok.org, 2004.
- SOMMERVILLE, IAN: Software Engineering, 8th Edition, Addison Wesley, 2007.
- MARTIN FOWLER, KENT BECK, JOHN BRANT, WILLIAM OPDYKE, DON ROBERTS: Refactoring: Improving the Design of Existing Code, Addison-Wesley Object Technology Series, 1999, ISBN 0201485672
- FOWLER, MARTIN: UML Distilled Third Edition A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, 2003.
- MESZAROS, GERARD: xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007.
- LIGGESMEYER, PETER: Software Qualität – Testen, Analysieren und Verifizieren von Software, Spektrum Verlag, 2009

Vielen Dank für Ihre Aufmerksamkeit!