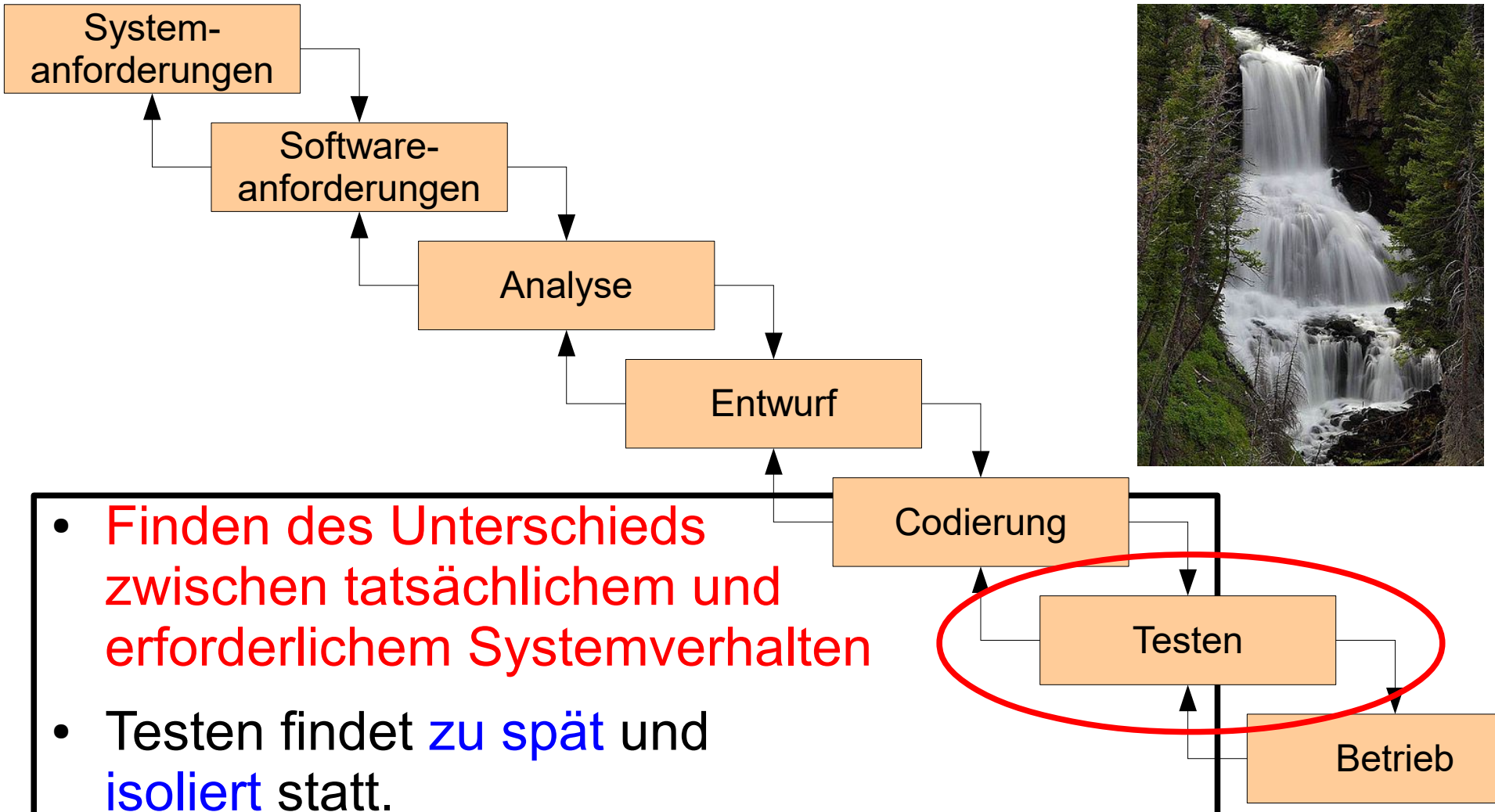
Software Engineering II

4 Testen

SS 2020

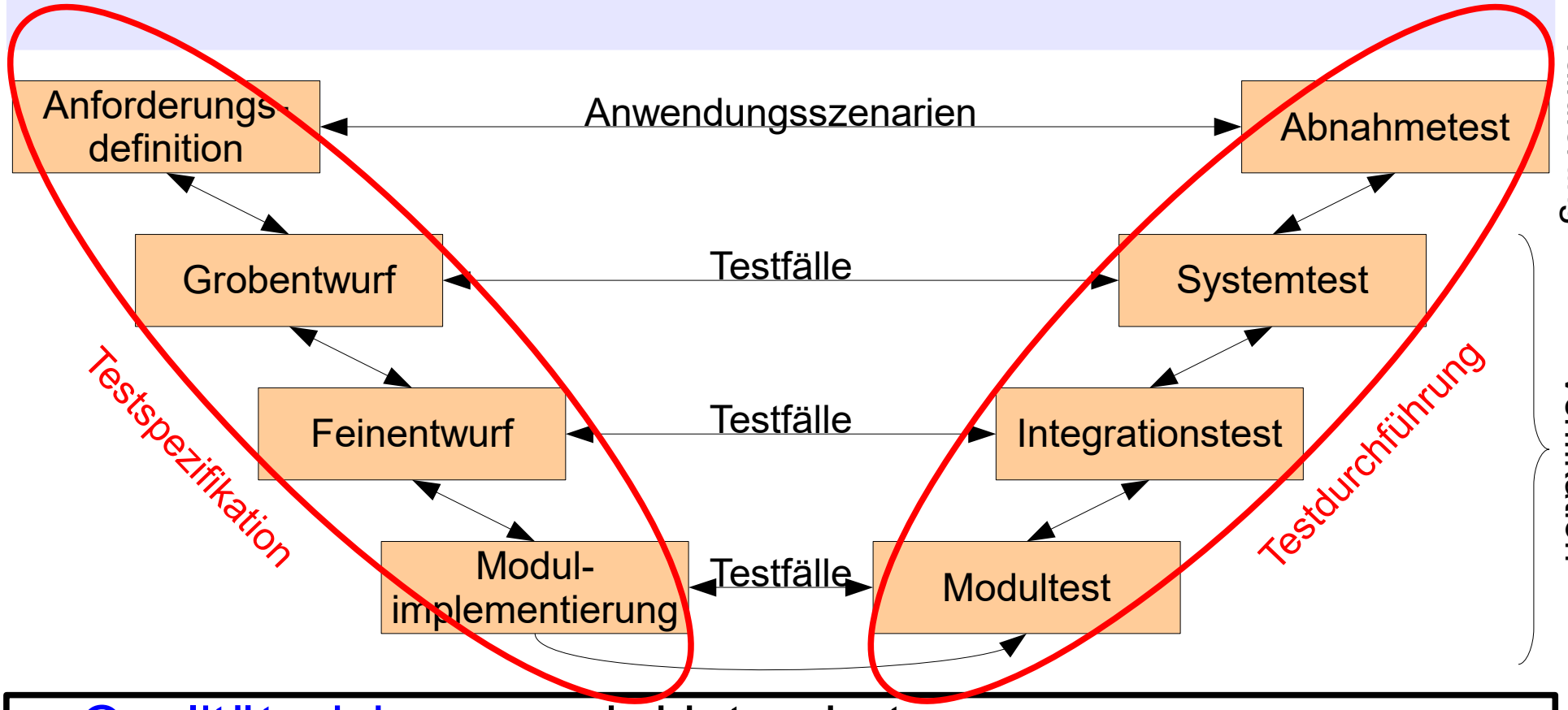
Prof. Dr. Dirk Müller

Testen im Wasserfallmodell



- Finden des Unterschieds zwischen tatsächlichem und erforderlichem Systemverhalten
- Testen findet zu spät und isoliert statt.
  - Verbesserung z. B. durch V-Modell, Prototypen und agile Methoden

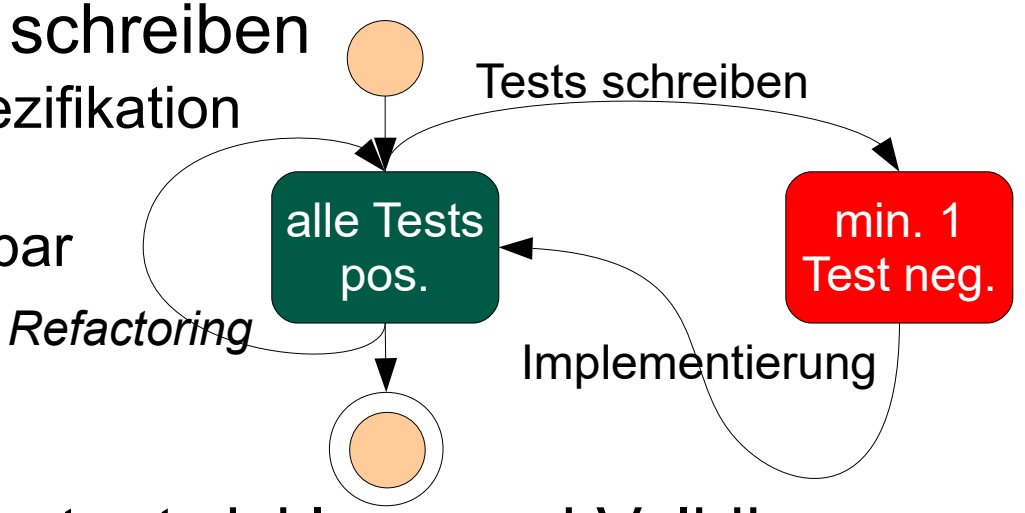
Testen im V-Modell



- Qualitätssicherung wird integriert
- Verifikation: Korrektes Produkt? (bzgl. Spezifikation)
  - „*Are we building the product right?*“
- Validierung: Richtiges Produkt? (bzgl. Nutzer)
  - „*Are we building the right product?*“

Testen in XP: Testgetriebene Entwicklung

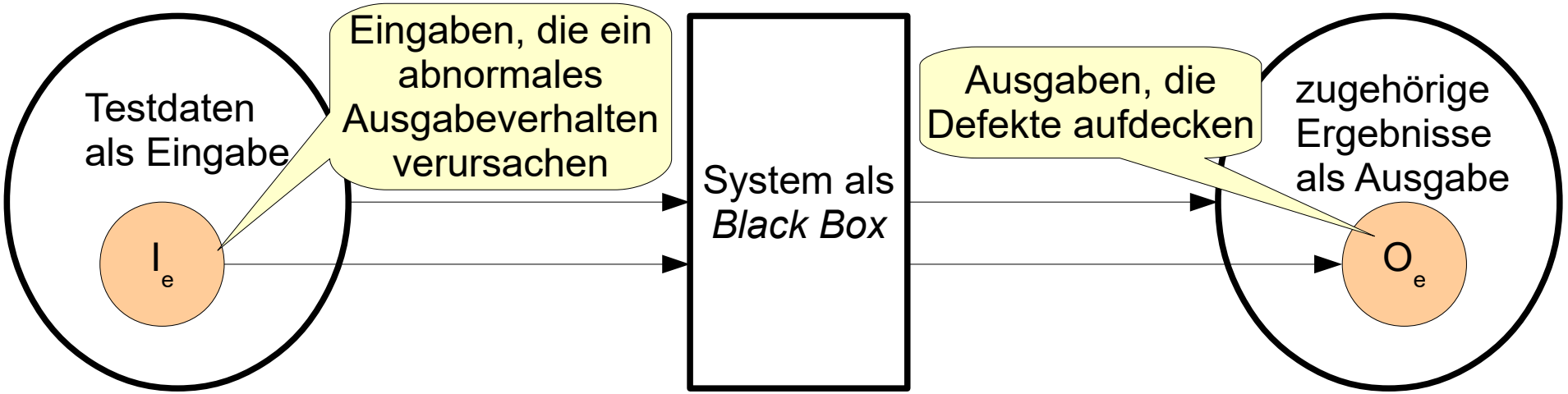
- Tests vor Implementierung schreiben
  - implizit Schnittstelle und Spezifikation
  - Mehrdeutigkeiten/Lücken in Spezifikation besser erkennbar
- schrittweise Entwicklung von Tests aus Storys



- Kunden-Einbeziehung in Testentwicklung und Validierung
- Testautomatisierung mit Test-Harnischen: Regressionstests
  - Testausführungs-*Engine*
  - Testdaten-*Repository* (Testskripte, Testprogramme)
- unterstützt hervorragend *Refactoring*
- Debugging stark erleichtert, da Fehler bereits lokalisiert
- Paradigmenwechsel, schwer zu erlernen
- Kompensation für fehlende Entwurfsphase + Dokumentation

Quelle: [Som10], S. 221 ff.

Testen: Prinzip und 2 Ziele

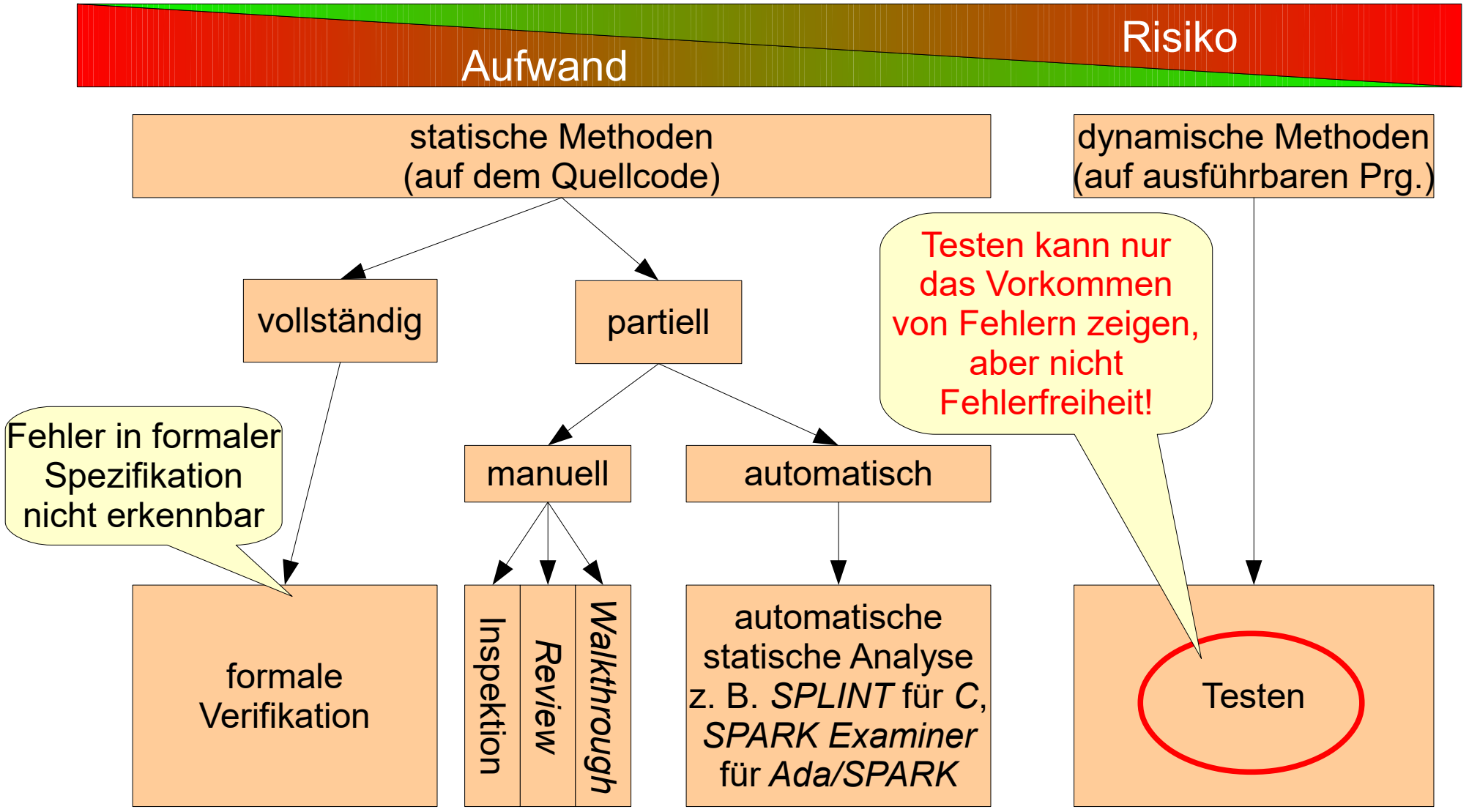


- Programm wird mit künstlichen Eingabedaten ausgeführt.
- Ausgaben des Testlaufs werden auf Fehler, Anomalien und Erfüllung nicht-funktionaler Anforderungen geprüft.
- Validierungstest: Zeigen, dass Anforderungen erfüllt werden (min. 1 Testfall pro Anforderung), Vertrauen in SW
- Defekt-Test (destruktiver Test): Ziel ist der negative Testausgang als Beleg für Fehler.

Quelle: [Som10], S. 206 f.

Debugging nötig

Testen zur Qualitätssicherung

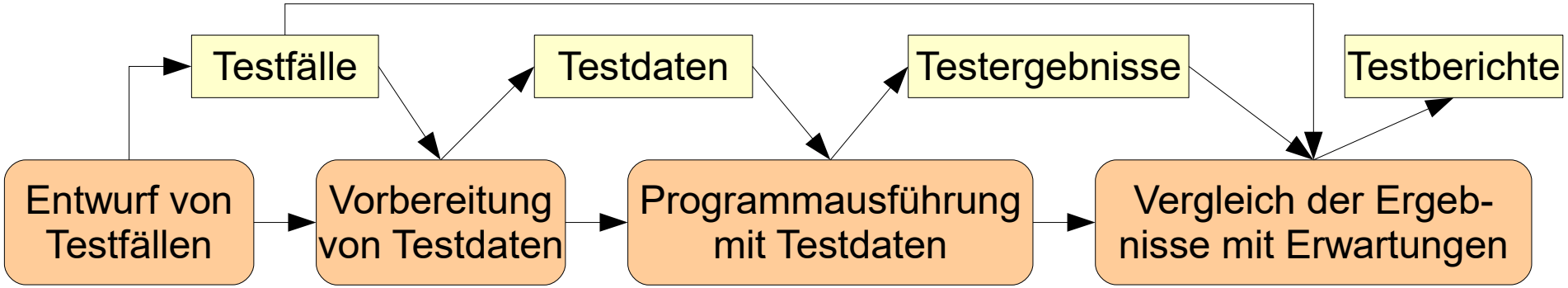


Inspektion vs. Testen

- Vorteile der Software-Inspektion gegenüber dem Testen
  - Wechselwirkungen nicht relevant für statische Inspektion; Fehler können andere Fehler maskieren, Seiteneffekte, aufwendige Regressionstests als Abhilfe
  - unvollständige Versionen ohne Zusatzkosten überprüfbar; Testharnische müssen extra entwickelt werden
  - Qualitätssicherung in einem weiteren Sinne abgedeckt: z. B. Einhaltung von Standards, Portabilität, Änderbarkeit
- Defektarten, die nur durch Testen entdeckt werden können
  - unerwartete Wechselwirkungen zwischen Programmteilen
  - Fehler im Zeitverhalten
- Inspektion nicht als Ersatz fürs Testen, am besten in Kombination
  - Inspektion von Testfällen

Quelle: [Som10], S. 208 f.

Software-Testprozess und Teststufen

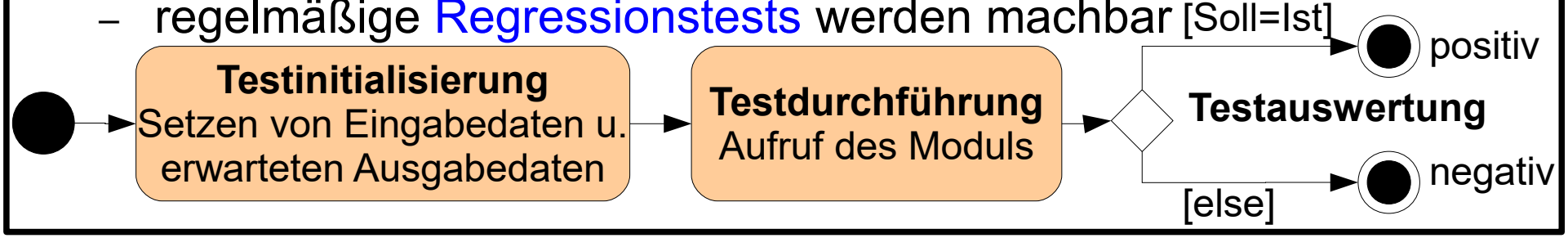


- Testfall: Eingabedaten + erwartete Ausgabedaten + Testgegenstand
- Entwicklungstests (meist Defekt-Tests mit Debugging)
  - während der gesamten Entwicklung, vor der Freigabe
  - durch Systemdesigner und Programmierer
- Freigabetests (Validierungstests, *Black-Box*-Tests)
  - kurz vor Freigabe (*Release*) gesamtes System getestet
  - durch spezielles Test-Team
- Nutzertests
  - Marketing- und Verkaufsaspekte mit einbezogen
  - durch echte oder potenzielle Nutzer

Quelle: [Som10], S. 209 f.

Entwicklungstests: Modultests

- Testgegenstand
  - OOP: Methode oder Klasse
  - strukturierte Programmierung: Funktion oder Prozedur
- Schritte beim Testen einer Klasse
  - Testen aller Operationen
  - Setzen und Prüfen aller Attribute
  - Simulation aller möglicher Lebensläufe eines Objekts (UML-Zustandsdiagramm); alle Ereignisse mit Zustandsänderung
- *Mock*-Objekte: passende Daten simuliert, z. B. akt. Uhrzeit
- Testautomatisierungswerkzeuge, z. B. *JUnit*, *ECU-TEST*
  - regelmäßige Regressionstests werden machbar



Quelle: [Som10], S. 211 f.

Auswahl von Testfällen für Modultests

- Validierungs- (normaler Betrieb) und Defekttests (ungewöhnliche Eingaben)
- Äquivalenzklassentests mit Randwertanalyse
  - gültige Äquivalenzklassen mit gültigen Eingabedaten: 2 Randwerte und 1 mittlerer Wert
  - ungültige Äquivalenzklassen mit ungültigen Eingabedaten: 1 Randwert

Off-by-one-Errors finden

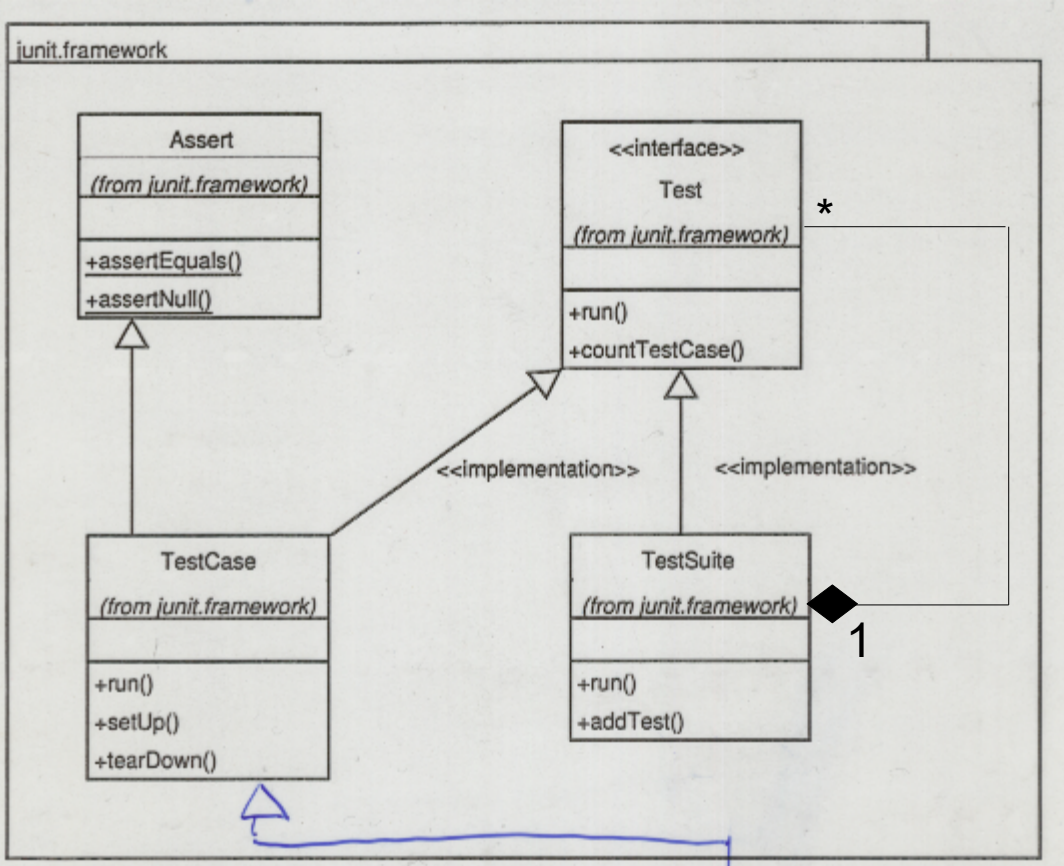
gilt so nur für eindimensionale Daten

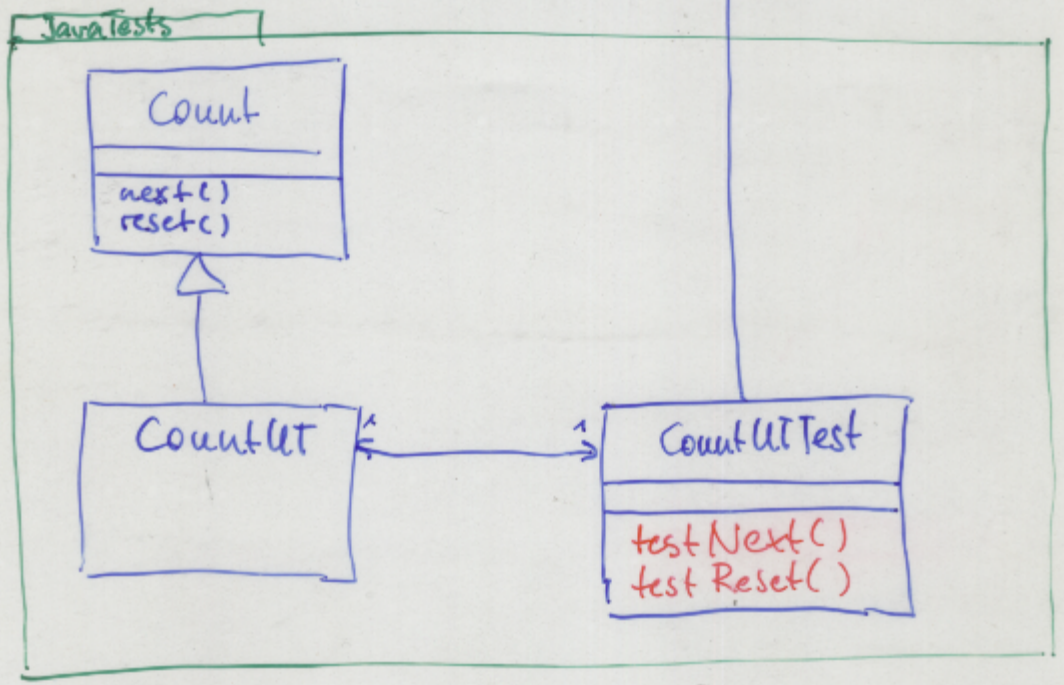
- richtlinienbasiertes Testen
  - Erfahrung liefert Richtlinien für Testfälle für Defekttests
  - für jede mögliche Fehlermeldung 1 Testfall (Zweigabdeckung)
  - Überlauf von Eingabepuffern testen
  - Wiederholung einer Eingabe oder einer Eingabefolge
  - numerischer Unter-/Überlauf

Quelle: [Som10], S. 213 ff.

JUnit

- von *Erich Gamma* und *Kent Beck* initiiert
- aktuelle Version 5.4.0 (Februar 2019)
- Paket `junit.framework`
  - Methodenpräfix `test`
- Testausgang
  - positiv
  - negativ
    - Fehler (*error*) — unerwartet
    - falsches Ergebnis (*failure*), technisch als `AssertionFailedError` — erwartet
- Portierung in viele Sprachen: *xUnit*





JUnit-3-Beispiel

```
public class DummyPojo {
    public int addValues(final int firstInt, final int secondInt) {
        return firstInt + secondInt;
    }
}
```

```
public class DummyPojoTest extends TestCase { /* (1) */
    DummyPojo pojo;

    public void setUp() { /* (2) */
        pojo = new DummyPojo();
    }

    public void testAdd() { /* (3) */
        int result = pojo.addValues(5, -5); /* (4) */
        assertEquals(0, result); /* (5) */
        assertNotEquals("Result is 10!?", 10, result); /* (6) */
    }
}
```

Quelle: [1]

Von JUnit3 zu JUnit4

- seit 2006
- org.junit.*
- Vererbung von Klasse TestCase nicht mehr nötig
- Steuerung über Annotationen
- Unterstützung von Timeouts
- bessere Fehlermeldungen durch Nutzung des *Hamcrest*-Frameworks u. a. mit assertThat

JUnit 3.x	JUnit 4.x
Base-Class *TestCase*	(no longer necessary)
Method "*setUp*"	Annotation *@Before*
Method "*tearDown*"	Annotation *@After*
Test-Method "testXYZ"	Annotation *@Test*
(no equivalent existing)	Annotation *@BeforeClass* (Once-and-only setUp)
(no equivalent existing)	Annotation *@AfterClass* (Once-and-only tearDown)
try { ... fail() } catch (Exception e) { assertTrue(true) }	Annotation *@Test(expected == Exception.class)*
Custom implementation for testing long-running operations and timeouts	Annotation *@Test(timeout = 1000)* Timeout setting in milliseconds.
Custom implementation to ignore test using *TestSuite*	Annotation *@Ignore*

Quelle: [1]

JUnit-4-Beispiel

```
public final class MyTest {
    private DummyPojo pojo;
    @BeforeClass public void initialize() {
        System.out.println("This is printed only once");
    }
    @AfterClass public void destroy() {
        System.out.println("The last thing printed");
    }
    @Before public void setUpMyTest() {
        pojo = new DummyPojo();
    }
    @Test public void adder() {
        final int result = pojo.addValues(5, -5);
        assertEquals(0, result);
    }
    @Ignore @Test public void multiplier() {
        // This test seems broken
        final int result = pojo.addValues(-5, -5);
        assertEquals(25, result);
    }
}
```

Quelle: [1]

Meldungen eines *Failure* unter `JUnit4`

früher 0,001 s

```
assertTrue(responseString.contains("color") ||
           responseString.contains("colour"));
// ==> failure message:
// java.lang.AssertionError:
```

jetzt bis zu 0,021 s (eine Größenordnung länger), aber viel leichter zu debuggen

```
assertThat(responseString, anyOf(   containsString("color"),
                                    containsString("colour")));
// ==> failure message:
// java.lang.AssertionError:
// Expected: (a string containing "color" or a string containing "colour")
//      but: was "Please choose a font"
```

JUnit-5-Neuerungen

- Neuentwicklung, benötigt *Java 8*, kompatibel zu *Java 9*
  - Version 5.6.1 am 22.03.2020 erschienen
- modularisiert, Integration in *Gradle*, *Surefire*
- Lambda-Ausdrücke unterstützt
- gruppierte Zusicherungen
- *Assumptions*, *Tags*, *Disabled Tests*

```
import org.junit.gen5.api.Assertions;
import org.junit.gen5.api.Test;
public class Test1 {
    @Test
    public void test()  {
        Assertions.assertEquals(3 * 6, 18);
        Assertions.assertTrue(5 > 4);
}}
```

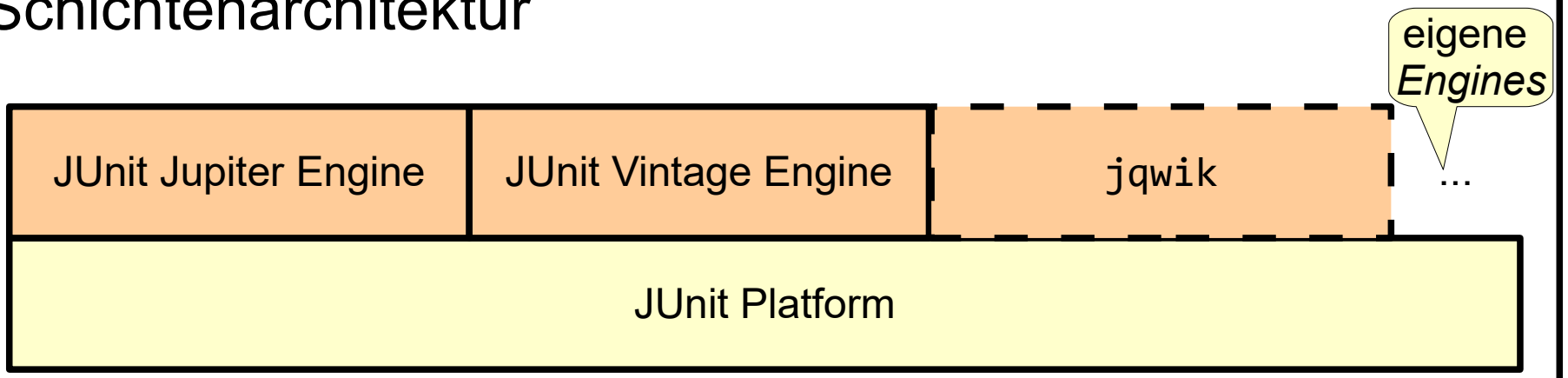
```
@Test
public void groupedAssertions() {
    Dimension dim = new Dimension(800, 600);
    assertAll("dimension",
    () -> assertTrue(dim.getWidth() == 800, "width"),
    () -> assertTrue(dim.getHeight() == 600, "height"));
}
```

- *Open Test Alliance for JVM*
  - Versuch, einen Standard bei *Exception-APIs* zu etablieren

Quelle: [2]

Architektur von `JUnit5`

- Schichtenarchitektur



- offene, generische Ablaufumgebung zur Ausführung und Protokollierung von Tests durch sogenannte *Engines*
- *JUnit Jupiter Engine*: neue API, Kern von *JUnit 5*
- *JUnit Vintage Engine*: Kompatibilitätsschicht zur Ausführung bestehender *Junit-4*-Tests
- `jqwik`: eigenschaftsbasiertes Testen, höhere Abstraktion

Entwicklungstests: Integrationstests

- Schnittstellen zwischen Komponenten
  - Parameterschnittstelle: Methoden, Prozeduren, Funktionen
  - prozedurale Schnittstelle: Weiterreichung einer Menge von Prozeduren durch Objekte/wiederverwendbare Komponenten
  - *Shared-Memory*-Schnittstelle: eingebettete Systeme
  - *Message Passing Interface (MPI)*: Nachrichtenaustausch z. B. bei Parallelrechnern oder Client-Server-Systemen
- Schnittstellenfehler
  - Schnittstellenmissbrauch: Parameter falschen Typs, in falscher Reihenfolge oder falsche Anzahl
  - Schnittstellen-Unverständnis: falsche implizite Annahmen, die zur Verletzung von Vorbedingungen führen,
    z. B. Binärsuche auf einem unsortierten Feld
  - Fehler im Zeitverhalten: Echtzeitsysteme, Konsument liest veraltete Produzentendaten durch verschiedene Geschwindigkeit

Quelle: [Som10], S. 216 ff.

Schnittstellenfehler

- schwer zu finden
  - sichtbares Fehlverhalten nur unter ungewöhnlichen Bedingungen, z. B. fehlerhafte Implementierung einer Warteschlange als Datenstruktur endlicher Länge, Testfall > endliche Länge nötig
  - Interaktion mehrerer Module kann nötig sein, um sichtbares Fehlverhalten zu erreichen, z. B. gültiger, aber fehlerhafter Wert
- Richtlinien zum Testen, um Schnittstellenfehler zu finden
  - Randwertanalyse bei Aufrufen externer Module
  - Zeiger, die übergeben werden, mit `null` belegen
  - gezielte Defekttests prozeduraler Schnittstellen
  - Stresstests für MPIs: viel mehr Nachrichten als in der Praxis zu erwarten sind, um Fehler im Zeitverhalten aufzudecken
  - *Shared-Memory*-Schnittstellen: verschiedene Aktivierungsreihenfolgen, um Fehler im Zeitverhalten aufzudecken
- starke Typisierung, Inspektion + autom. statische Analyse

Quelle: [Som10], S. 218

Testreihenfolge

Frühauf betrachtet zwei Arten von Beziehungen als entscheidend:

1. Vererbungsbeziehung
2. "benutzt"-Beziehung

Grundregeln:

- teste erst dann eine Klasse, wenn deren Superklassen getestet sind.
- teste erst dann eine Klasse, wenn die Klassen getestet sind, die von der zu testenden Klasse benutzt werden.

Beispiel:

- A benutzt A1
- A benutzt A2
- B erbt von A
- E benutzt E1
- E benutzt E2
- F erbt von E
- B benutzt F
- C erbt von B
- D erbt von B
- C benutzt C1

Eine mögliche Test-Reihenfolge wäre: A1 - A2 - A - E1 - E2 - E - F - B - D - C1 - C

Entwicklungstests: Systemtests

- alle Komponenten werden durch alle Schnittstellen in ein System integriert
  - verschiedene Teams
  - Standardsoftware und -bibliotheken
- fallbasiertes Testen
  - jeder Anwendungsfall mittels Sequenzdiagramm durchgespielt
- vollständige Testüberdeckung unmöglich
- Richtlinien zur Auswahl von Testfällen
  - jede Anweisung einmal ausgeführt
  - alle Funktionen in Menüs testen
  - Kombinationen von Zuständen (z. B. Textformatierung, Fußnoten im Mehrspaltensatz) testen
  - bei freier Eingabe durch Nutzer: korrekte und inkorrekte Strings

Quelle: [Som10], S. 219 ff.

Freigabetests

- Validierungstest kurz vor Freigabe (*Release*) des Systems für Kunden und Nutzer durch extra Test-Team
  - evtl. auch für andere Teams bei großen Projekten
- anforderungsbasiertes Testen
  - gute Anforderungen testbar (Erfüllung durch Tests überprüfbar)
  - systematischer Ansatz, um Testfälle zu entwerfen
  - Validierungstest
  - häufig mehrere Tests pro Anforderung, um Test der Anforderung ausreichend abzudecken
- szenariobasiertes Testen
  - realistische Kontextszenarien als Feldtest (nahe an echten Nutzern)
  - Szenario: *Story* beschreibt, wie ein System genutzt werden könnte
  - Szenariotest: glaubwürdige, relevante, komplexe *Story*
  - häufig mehrere Anforderungen incl. Kombinationen dieser getestet

Quelle: [Som10], S. 224 ff.

Performance-Tests

- Testen nichtfunktionale Anforderungen wie Antwortzeiten und Durchsätze unter Last: Skalierbarkeit
- als Validierungs- und Defekttest
- Lasttest: Last wird künstlich (z. B. Lastserver) schrittweise erhöht.
  - Stresstest: gezielt über spezifizierte Maximallast
  - Dauerlasttest: über mehrere Tage, z. B. Finden von Speicherlecks
  - Ausfalltest: Verhalten bei Ausfall von Systemkomponenten
- besonders wichtig für verteilte Systeme: Netzwerk muss mehr und mehr Koordinationsarbeit leisten.
  - tolerierbaren Schwellwert experimentell (per Lasttest) bestimmen
  - neue Transaktionen ab diesem Schwellwert ablehnen

Quelle: [Som10], S. 227

Nutzertests

- realistischste Art des Testens, alle anderen künstlich
- Alphatests
  - Nutzer und Entwickler testen zusammen
  - Reduzierung von Risiko
  - für agile Methoden Nutzerbeteiligung beim Testen zentral
- Betatests
  - frühes *Release* ausgewählten/allen Nutzern zur Verfügung gestellt
  - Wechselwirkungen der Software mit verschiedenen Umgebungen (Hardware, Betriebssystem, Treiber, etc.)
- Abnahmetests
  - Abnahmekriterien müssen ggf. an geänderte Anforderungen angepasst werden.
  - Kunde entscheidet bei Individualsoftware, ob er sie kauft (bezahlt)
  - häufig bedingte Akzeptanz: Nachverhandlungen

Zusammenfassung

- Testen: Finden des Unterschieds zwischen tatsächlichem und erforderlichem Systemverhalten
  - Validierungstests und Defekttests
- Testen kann nur das Vorkommen von Fehlern zeigen, aber nicht Fehlerfreiheit
- Entwicklungstests: Modul-, Integrations- und Systemtests
- Äquivalenzklassentest mit Randwertanalyse: gute Testfälle
- automatisiertes Testen: Regressionstests ermöglicht
  - *JUnit* als De-facto-Standard für *Java*
- testgetriebene Entwicklung: erst Test, dann Implementierung, in XP in Kombination mit *Refactoring*
- szenariobasiertes Testen: nahe an Realität
- Abnahmetest: anhand von Abnahmekriterien Entscheidung, ob SW bezahlt wird, häufig Nachverhandlung

Literatur

[1] Axel Irriger, cirquent GmbH: „JUnit“, Download am 4.4.2016, http://www.methodsandtools.com/tools/tools.php?junit

[2] Stefan Bley, Saxonia Systems: „What’s new in JUnit 5?“, 8.2.2016, Download am 4.4.2016, http://blog.so-geht-software.de/2016/02/whats-new-in-junit-5/