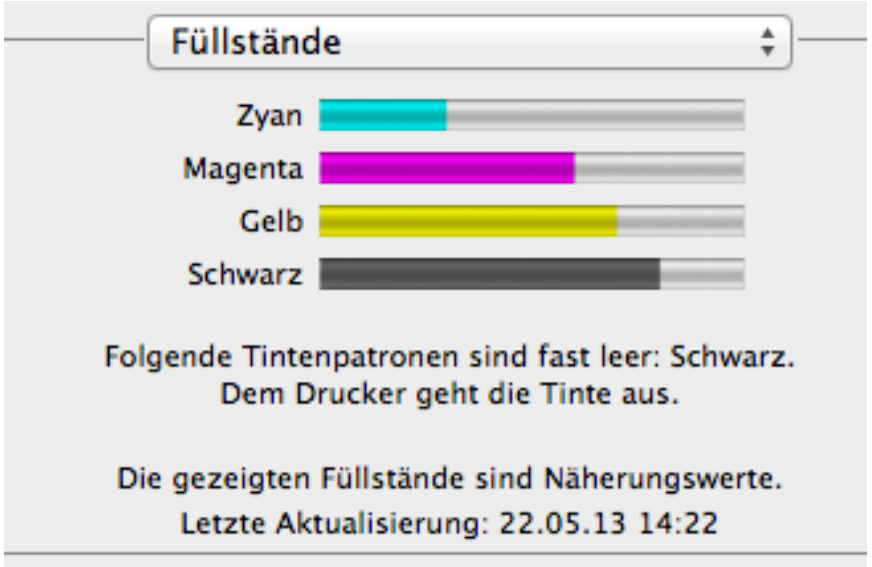
Software Engineering 1

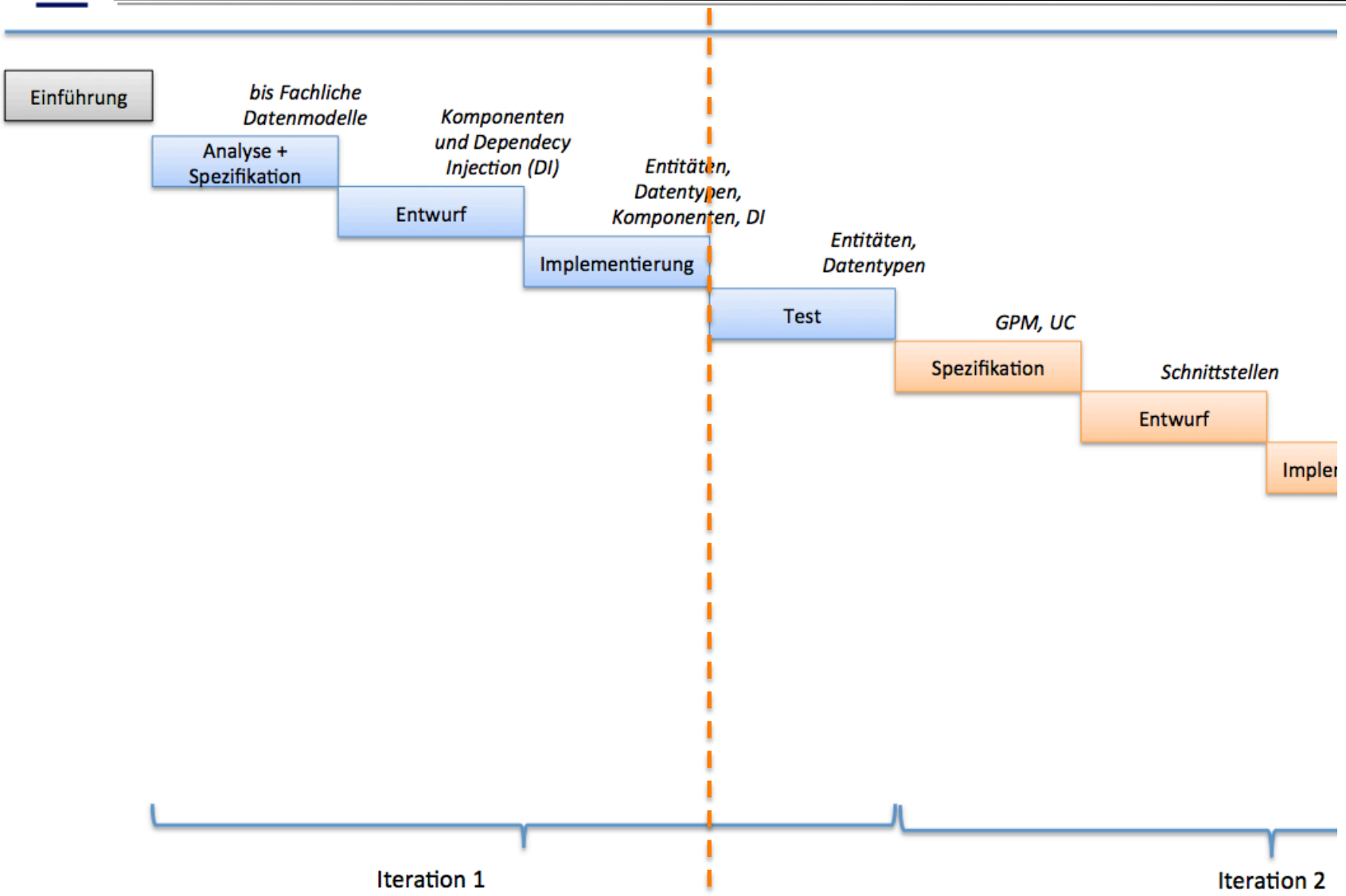
Hochschule für angewandte Wissenschaften Hamburg
Fachbereich Informatik

Prof. Dr. Stefan Sarstedt
(stefan.sarstedt@haw-hamburg.de)
Raum: 10.85

Testen

Iteration 1





aktueller Zeitpunkt

Testen

„Testen ist die Ausführung eines Programms mit dem Ziel, Fehler zu entdecken.“

Meyers (1979)

Testen im eigentlichen Sinn

„Testen ist die Vorführung eines Programms oder Systems mit dem Ziel zu zeigen, dass es tut, was es tun sollte.“

Hetzel (1984)

Abnahmetest

- **Kein Test** in diesem Sinne ist
  - irgendeine Inspektion eines Programms
  - die Vorführung eines Programms
  - die Analyse eines Programms durch Software-Werkzeuge (z. B. die Erhebung von Metriken)
  - die Untersuchung eines Programms mit Hilfe eines Debuggers

Welche Software-Einheiten werden getestet?

Modultest	Testen eines einzigen Moduls bzw. einer Klasse.
Komponententest	Testen einer Komponente.
Integrationstest	Testen des Zusammenspiels mehrerer Komponenten.
Systemtest	Testen des Gesamtsystems ohne reale Nachbarsysteme.
Verbundtest	Testen der Software im Zusammenspiel mit den Nachbarsystemen.
Abnahmetest	Testen durch den Kunden mit dem Ziel, das Projekt als „abgeschlossen“ kennzeichnen zu können.

Testen

- Ein Test muss systematisch sein.
- **Systematischer Test**: Ein Test, bei dem
  - die Randbedingungen definiert oder präzise erfasst sind
    - Welches Programm, Übersetzer, Betriebssystem, andere Software, Wer, Speicher, weitere Geräte, Zustand, …
  - die Eingaben systematisch ausgewählt wurden
    - Tastatur, Dateien, Datenbank, Zustände der Geräte
  - die Ergebnisse dokumentiert und nach Kriterien beurteilt werden, die vor dem Test festgelegt wurden
    - Resultate dokumentieren und ablegen; bei interaktiven Tests häufig manuell

Arten von Tests – Black-Box/Glas-Box

Black-Box-Test (auch: Funktionstest)	Betrachtung des Systems/des Moduls/der Komponente als „Black Box“, d. h., ohne die Interna zu kennen. Testfälle auf Basis der in der Spezifikation geforderten Eigenschaften.
Glass-Box-Test (auch: Strukturtest)	Betrachtung des Systems/des Moduls/der Komponente als „White Box“ mit Wissen über die Interna.

in SE-2

Black-Box-Test (1/3)

- Auswahl der Testfälle richtet sich nach Eingaben, Ausgaben und ihrer funktionellen Verknüpfung
- Ziel: Herausfinden, ob vorgegebene Eingaben erwartete Resultate erzielen
- **Kriterien** für die Auswahl von Testfällen
  - Funktionsüberdeckung
    Jede Funktion wird in mindestens einem Testfall ausgeführt
  - Eingabeüberdeckung
    Jedes Eingabedatum wird in mindestens einem Testfall verwendet
  - Ausgabeüberdeckung
    Jedes Ausgabedatum wird in mindestens einem Testfall erzeugt

Black-Box-Test (2/3)

- Im Allgemeinen **unmöglich**, **alle Werte** einer Eingabegröße zu testen

Beispiel:

```
int f(int x, int y) { … }
```

32 bit int $\Rightarrow 2^{32} \cdot 2^{32} = 2^{64} \approx 10^{19}$ Eingaben

→ Wenn für einen Aufruf 10^{-9} sec zur Verfügung stehen, ergibt sich für den Test eine Laufzeit von

$10^{-9} \cdot 10^{19}$ sec ≈ 300 Jahre

Black-Box-Test (2/3)

- Deshalb: Festlegung von Äquivalenzklassen (= Teilbereiche des Wertebereichs, die sich bzgl. Fehler gleich verhalten)
  - Auswahl eines Repräsentanten jeder Äquivalenzklasse
  - Überprüfung (der Teilbereiche) auf Vollständigkeit

Beispiel	**Äquivalenzklasse „Gültige Werte“**	**Äquivalenzklasse „Ungültige Werte“**
Ganzzahliger Bereich $a \leq W \leq n$	$a \leq W \leq n$	$W < a$ $W > n$
Enumeration {A, B, C}	W in {A, B, C}	W nicht in {A, B, C}
Wort W: 1.Zeichen = Buchstabe?	W.1 = Buchstabe	W.1 ≠ Buchstabe

Black-Box-Test (3/3)

- Jede Äquivalenzklasse der Eingabedaten wird in mindestens einem Testfall berücksichtigt
- Unterschiedliche Repräsentanten, wenn Klasse in mehreren Testfällen angesprochen (zum Ausschluss dummer Zufälle)
- Bevorzugung von Grenzwerten

- Beispiele
  - „Reelle“ Zahlen -1.0..1.0 (3 Stellen) -1.001, -1.000, 1.000, 1.001
  - Natürliche Zahlen 1..255 0, 1, 255, 256
  - Zeichenkette mit Länge < 10 Ketten der Länge 0, 1, 10, 11
  - Liste mit max. 1000 Elementen Listen der Länge 0, 1, 1000, 1001

Unittest

- Unittests eignen sich gut für Black-Box-Tests und prüfen,
  - ob Methoden die erwarteten Ergebnisse liefern und ob Invarianten erfüllt sind.
  - meist bei allen Test-Arten anwendbar (aber nicht für alle Tests)
  - Mehrere Tests werden zu einer Test-Suite zusammengefasst
- Frühzeitig, häufig und automatisiert Unittests durchführen (lassen)!
  - denn: 1. Regel für manuelle Tests: Manuelle Tests werden nicht ausgeführt.
- Fehlerbehebung umso teurer, je später ein Fehler gefunden wird.
- Unittests entstehen eventuell sogar vor der Implementierung = „Test-Driven-Development“ (TDD) oder „Test-First-Development“ (TFD)
- Ohne Unittests zu arbeiten, ist nicht zeitgemäß und unprofessionell!

Unittest

- Beispiel: Test eines Erfolgsszenarios
- In JUnit annotiert mit `@Test`, in C# mit `[TestMethod]`

```
[Test]
public void TesteBuchungenDurchführenSuccess()
{
  Assert.AreEqual(0, konto.Kontostand);
  konto.Buche(100000, "Budget für 2014");
  Assert.AreEqual(100000, konto.Kontostand);
  Assert.AreEqual(1, konto.Buchungen.Count);

  konto.Buche(-100001, ”Pizza ist teuer");
  Assert.AreEqual(-1, konto.Kontostand);
}
```

Wählen Sie gute(!!) Methodennamen; machen Sie Erfolgs-/ Misserfolgsszenarien im Namen kenntlich („...Success“, „...Fail“)

Unittest

- Methoden, zur Initialisierung von Testdaten und bauen der Komponenten
  - In Java: `@Before`, `@BeforeClass`, etc.
  - in C#: `[TestInitialize]`, `[ClassInitialize]`

```
[TestInitialize]
public void InitializeTest()
{
  konto = new Konto(new KontoNrTyp(100, 123)) { DispoKredit = 500 };
}
```

Teststammdaten in Testfall verwendet

```
public void TesteBuchungenDurchführenSuccess()
{
  Assert.AreEqual(0, konto.Kontostand);
  konto.Buche(100000, "Budget für 2014");
  Assert.AreEqual(100000, konto.Kontostand);
```

Unittest

- Beispiel: Test eines Misserfolgsszenarios

```
[TestMethod]
[ExpectedException(typeof(KontoNotGedecktException))]
public void TesteBuchungenDurchführenFailWegenÜberzogenemDispo()
{
  konto.Buche(-501, "Sehr teuer die Pizza heutzutage.");
}
```

Misserfolgsszenario

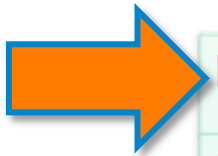
- hier praktisch durch die Annotation `ExpectedException`, ansonsten

```
public void TesteBuchungenDurchführenFailWegenÜberzogenemDispo()
{
    bool exceptionWasThrown = false;
    try
    {
        konto.Buche(-501, "Teuer die Pizza heutzutage.");
    }
    catch (KontoNotGedecktException ex)
    {
        Assert.AreEqual(konto.KontoNummer, ex.Konto.KontoNummer);
        exceptionWasThrown = true;
    }
    Assert.IsTrue(exceptionWasThrown, "KontoNotGedecktException erwartet.");
}
```

Unittest

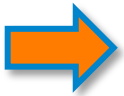
- **Keine Abhängigkeiten zwischen den Testfällen zulassen!!!!**
  - z.B. Testfall A erzeugt Testdaten, die in Testfall B benutzt werden
  - die Reihenfolge der Testausführung ist allerdings „zufällig“! Bei einem Update/Wechsel des Testframeworks (und auch zwischendurch) kann sich dies ändern...dann hat man ein Problem
- Also: Testfälle **unabhängig** voneinander gestalten!
  - Gemeinsame Stammdaten mit `@Before`, `@BeforeClass`, etc. entsprechend erstellen
  - Gut über Teststammdaten nachdenken!

Tests der Klasse Konto



Modultest	Testen eines einzigen Moduls bzw. einer Klasse.
Komponententest	Testen einer Komponente.
Integrationstest	Testen des Zusammenspiels mehrerer Komponenten.
Systemtest	Testen des Gesamtsystems ohne reale Nachbarsysteme.
Verbundtest	Testen der Software im Zusammenspiel mit den Nachbarsystemen.
Abnahmetest	Testen durch den Kunden mit dem Ziel, das Projekt als „abgeschlossen“ kennzeichnen zu können.

Fallbeispiel Banking – Modultests



```
[TestInitialize]
0 Verweise
public void InitializeTest()
{
    konto = new Konto(new KontoNrTyp(100, 123)) { DispoKredit = 500 };
}

[TestMethod]
0 Verweise
public void TesteBuchungenDurchführenSuccess()
{
    Assert.AreEqual(0, konto.Kontostand);

    konto.Buche(100000, "Budget für 2014");
    Assert.AreEqual(100000, konto.Kontostand);
    Assert.AreEqual(1, konto.Buchungen.Count);

    konto.Buche(100, "zweite Buchung");
    konto.Buche(50, "dritte Buchung");
    konto.Buche(-70, "vierte Buchung");
    Assert.AreEqual(4, konto.Buchungen.Count);
    Assert.AreEqual(100080, konto.Kontostand);
    Assert.IsTrue(konto.Buchungen[3].Beschreibung.Contains("vierte Buchung"));

    konto.Buche(-100081, "fünfte Buchung");
    Assert.AreEqual(-1, konto.Kontostand);
}
```

Fallbeispiel Banking – Modultests

```
[TestMethod]
0 Verweise
public void TesteBuchungenDurchführenDispoAusgenutztSuccess()
{
    konto.Buche(-500, "Teuer die Currywurst heutzutage.");
}
```

```
[TestMethod]
[ExpectedException(typeof(KontoNotGedecktException))]
0 Verweise
public void TesteBuchungenDurchführenFailWegenÜberzogenemDispo_Verbessert()
{
    konto.Buche(-501, "Sehr teuer die Pizza heutzutage.");
}
```

Weitere Beispiele

- http://www.sqlite.org/testing.html