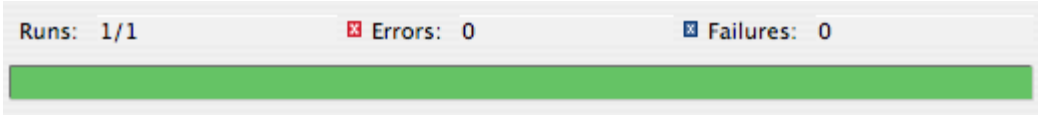



# Unit Tests mit JUnit

Softwaretests und Unit Tests

Grundlagen von JUnit

Fortgeschrittene Techniken

Testbetonte Entwicklungsprozesse

DHBW

# Software und Fehler

- Menschen schreiben Software
- Menschen machen Fehler

→ Software enthält Fehler

DHBW

# Fehler binden Resourcen

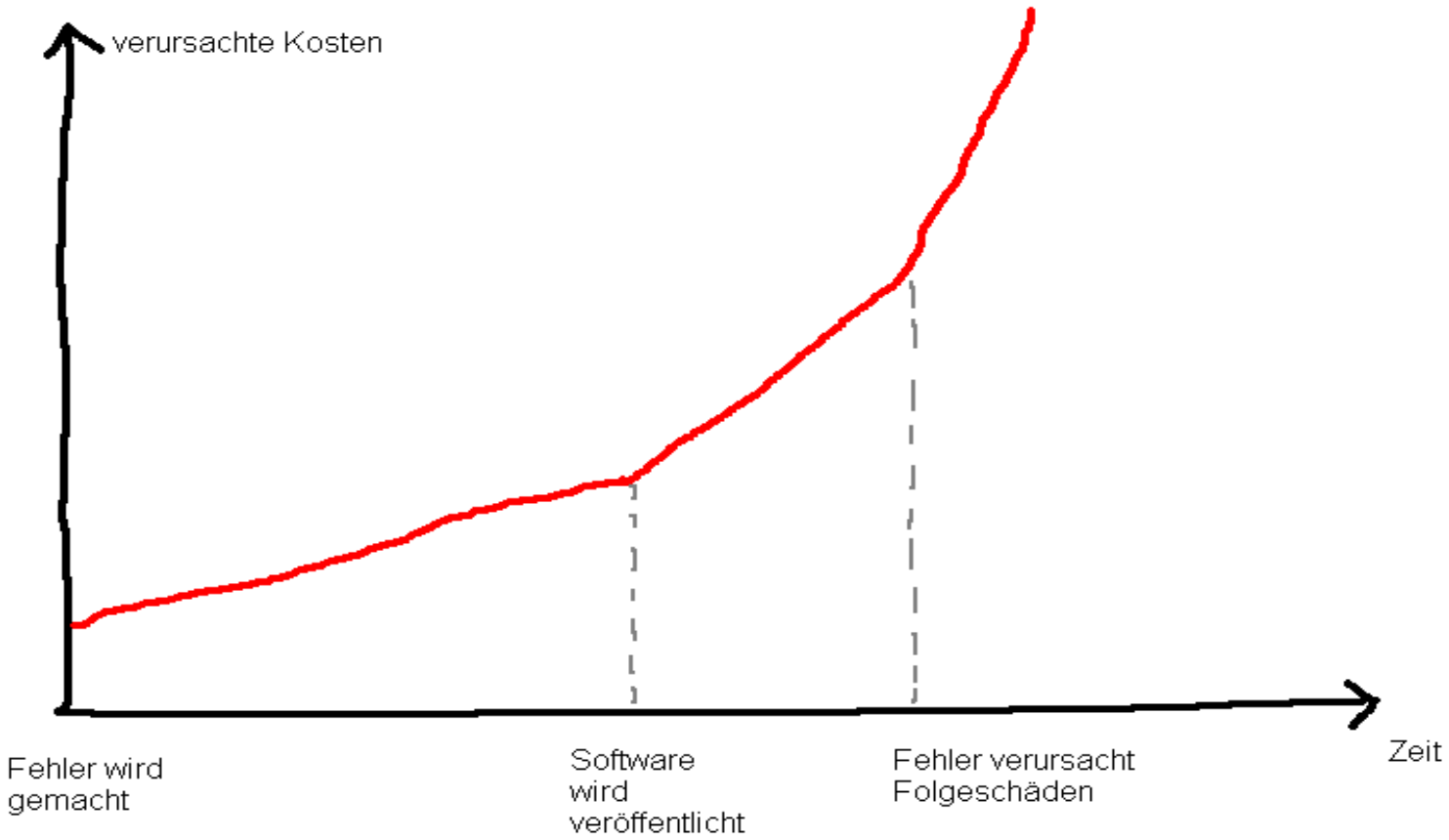
- Jeder Fehler bindet Resourcen
  - Geld, Zeit
  - Aufmerksamkeit, Nerven, Vertrauen
- Die Kosten eines Fehlers steigen mit der Zeit seiner Existenz
  - Kosten selbst, wenn man ihn sofort bemerkt und beseitigt

Dieser Theorie fehlt noch der Beleg, siehe auch:

http://blog.securemacprogramming.com/2012/09/an-apology-to-readers-of-test-driven-ios-development/

DHBW

# Kosten eines Fehlers




- Es gibt Zeitpunkte, ab denen ein Fehler sprunghaft teurer wird

Die Grafiken dieser Folie wurden mit Microsoft Paint erstellt.

# Softwarefehler sind teuer

- Jeder veröffentlichte (releaste) Fehler kostet deutlich mehr als ein unveröffentlichter Fehler
  - Schätzung: knapp 60 Milliarden US-Dollar Schaden für die US-Wirtschaft pro Jahr durch Softwarefehler (Quelle: RTI International)
- Jede fehlerhafte Installation verursacht Behebungskosten (Multiplikator)

DHBW

# Softwaretests sind Pflicht

- Testen von Software ist gesetzlich vorgeschrieben
  - Nicht testen ist „grob fahrlässig“
- Tests beschützen programmierte Funktionalität
  - Versicherung gegen versehentliche Änderung
- Tests können als Orientierungshilfe eingesetzt werden

DHBW

# Tests als Hilfsmittel

- Einige moderne Entwicklungsmethoden basieren auf (Unit) Tests
  - Test First
  - Test Driven Development
- Tests werden dabei als „Leitmotiv“ für die Programmierung verwendet
- Ein vorhandener Test unterscheidet gewollte Funktionalität von zufälligen „Features“

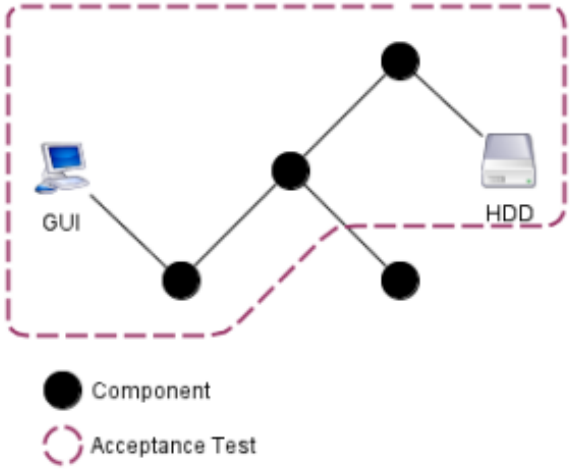
DHBW

# Eine Klassifikation von Tests

- Begriffsvielfalt in Bezug auf Testarten
- Eine mögliche Bezeichnung:
  - Akzeptanztests
  - Integrationstests
  - Komponententests (Unit Tests)
  - Leistungstests

# Akzeptanztests

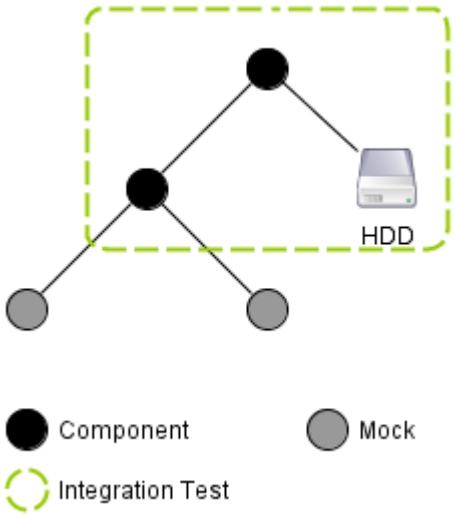
- Vollständiges System wird gestartet
- Möglichst passende Laufzeitumgebung
  - Echte Datenbank, Hardware
- Durchführung mit Mitteln des Benutzers
  - Interaktion mit Bedienoberfläche
- Ziel: Durchspielen echter Bedienszenarien




DHBW

# Integrationstests

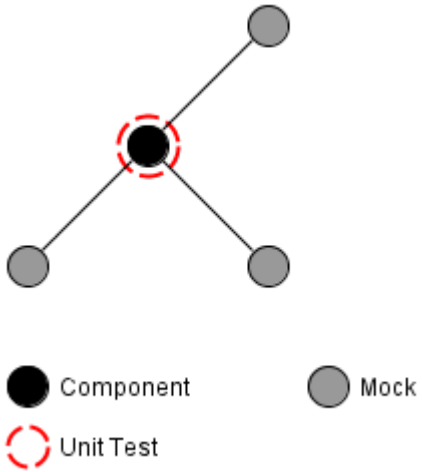
- Nur relevante Teile des Systems werden gestartet
- Nicht zu testende Systemteile durch Stellvertreter ersetzt
- Durchführung mittels Test-framework (Methodenaufrufe)
  - Interaktion der Systemteile untereinander
- Ziel: Zusammenspiel der Komponenten sicherstellen




DHBW

# Komponententests aka Unit Tests

- Nur relevanter Teil des Systems wird gestartet
- Alle anderen Systemteile durch Stellvertreter ersetzt
- Durchführung mittels Test-framework (Methodenaufrufe)
  - Prüfen der Rückgabewerte
- Ziel: Korrekte Implementierung der Komponente (Unit) sicherstellen




DHBW

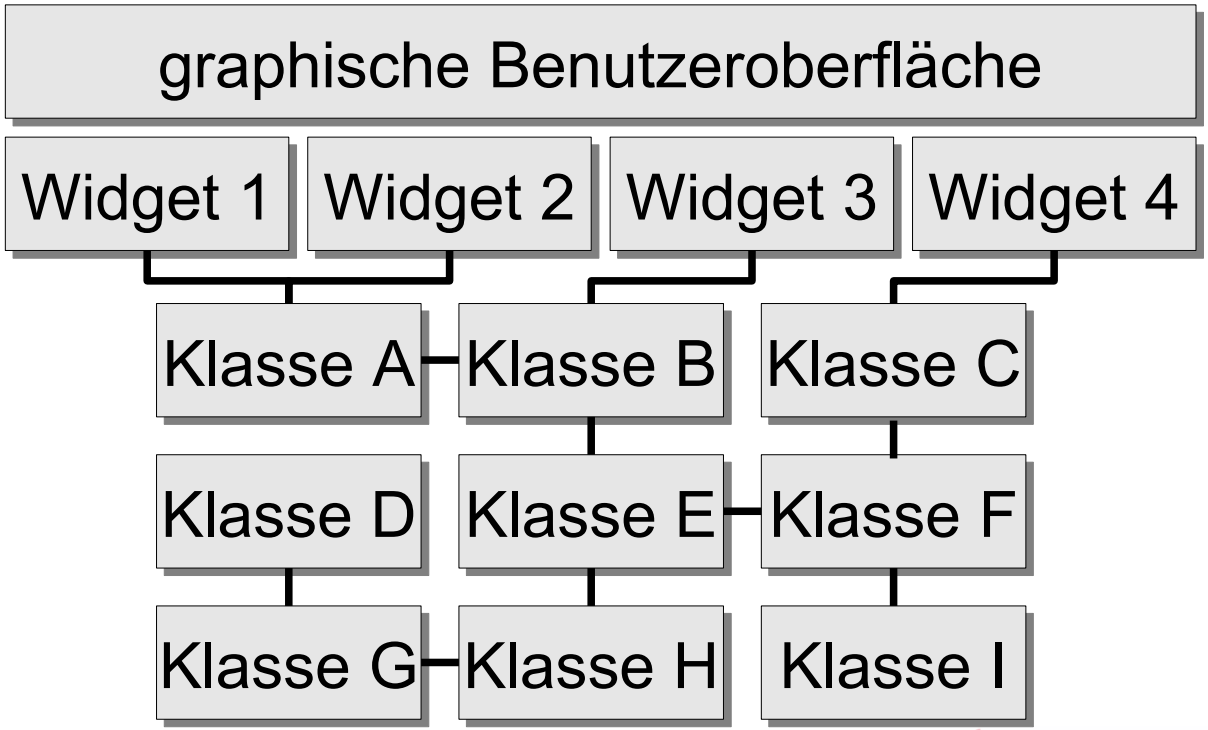
# Was sind Unit Tests?

- Tests für eine Komponente (Unit)
  - Unabhängig von anderen Komponenten
  - Pro Test ein Aspekt der Komponente
- Ausführbare und selbst-überprüfende Spezifikation von Komponenten
  - “Komponente” ist meist eine Klasse
- Aktive Dokumentation für eine Klasse
  - Zeigt Verwendung und Verhalten in Ausnahmefällen

DHBW

# Isolation von Komponenten

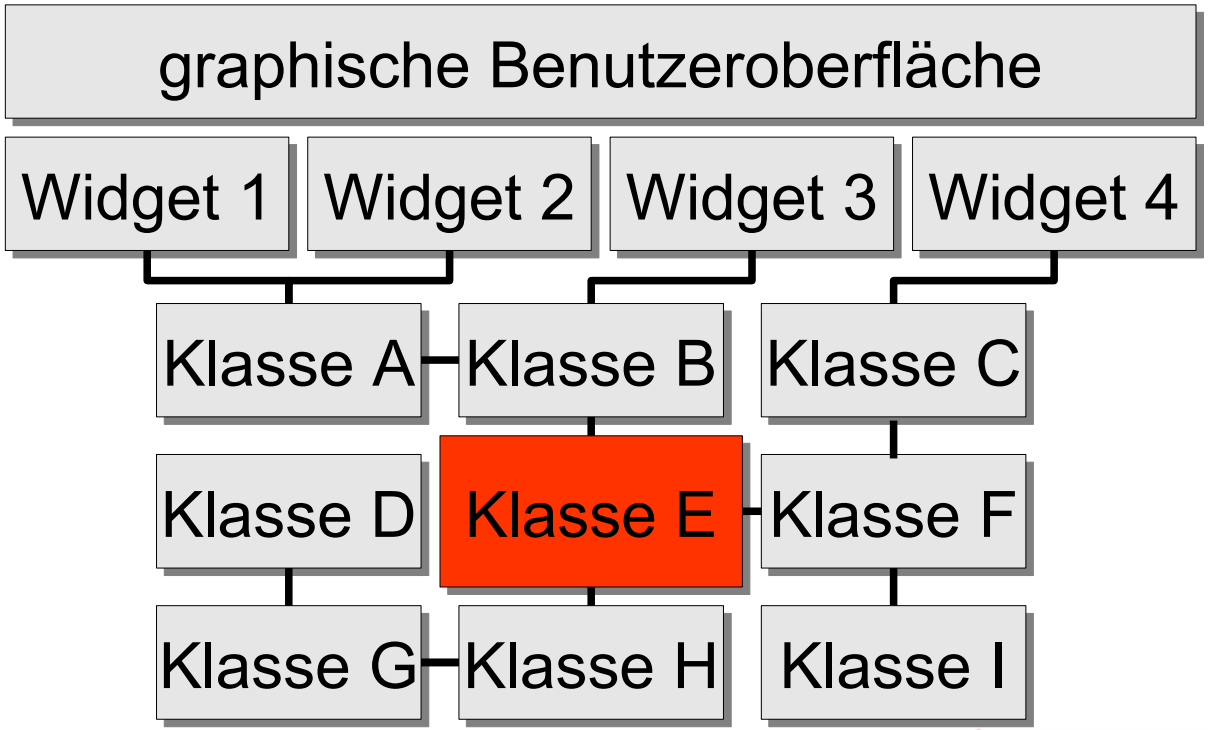
Ein Software-system




DHBW

# Isolation von Komponenten

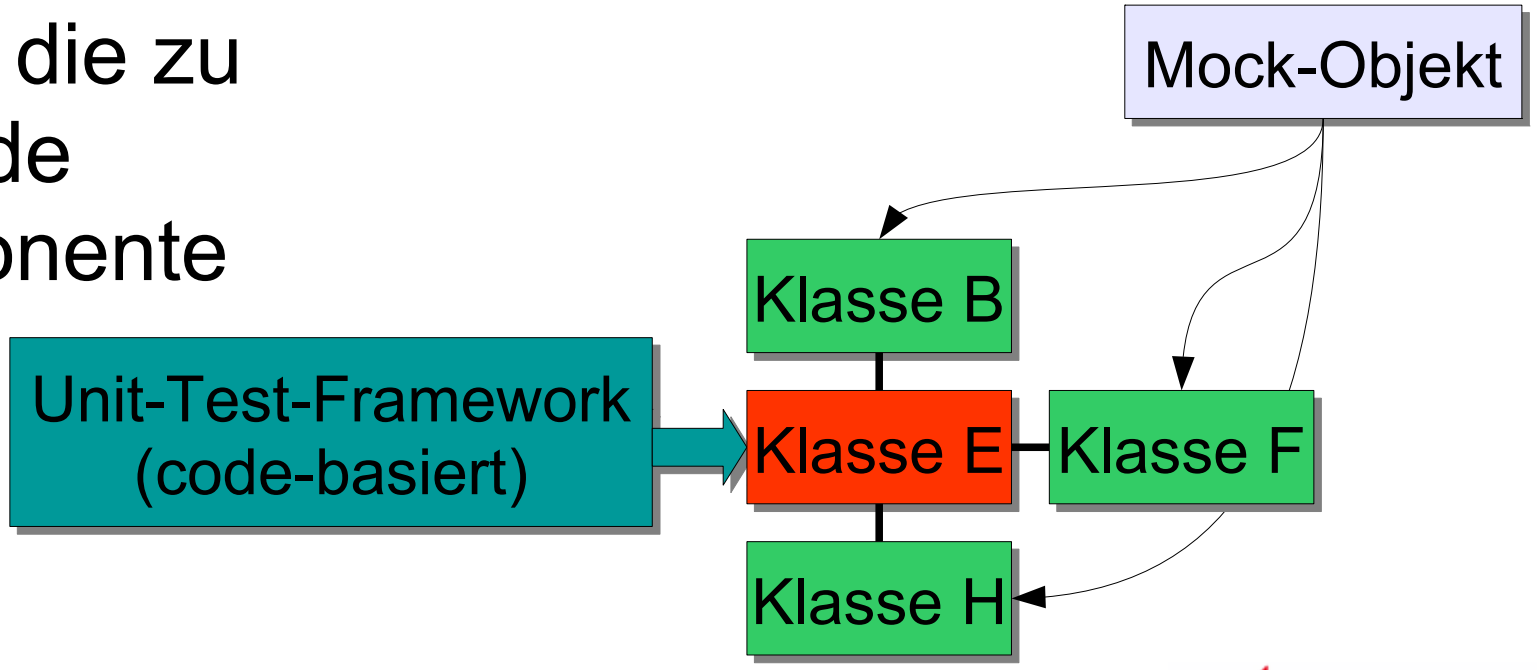
Eine Funktionalität muss getestet werden




DHBW

# Isolation von Komponenten

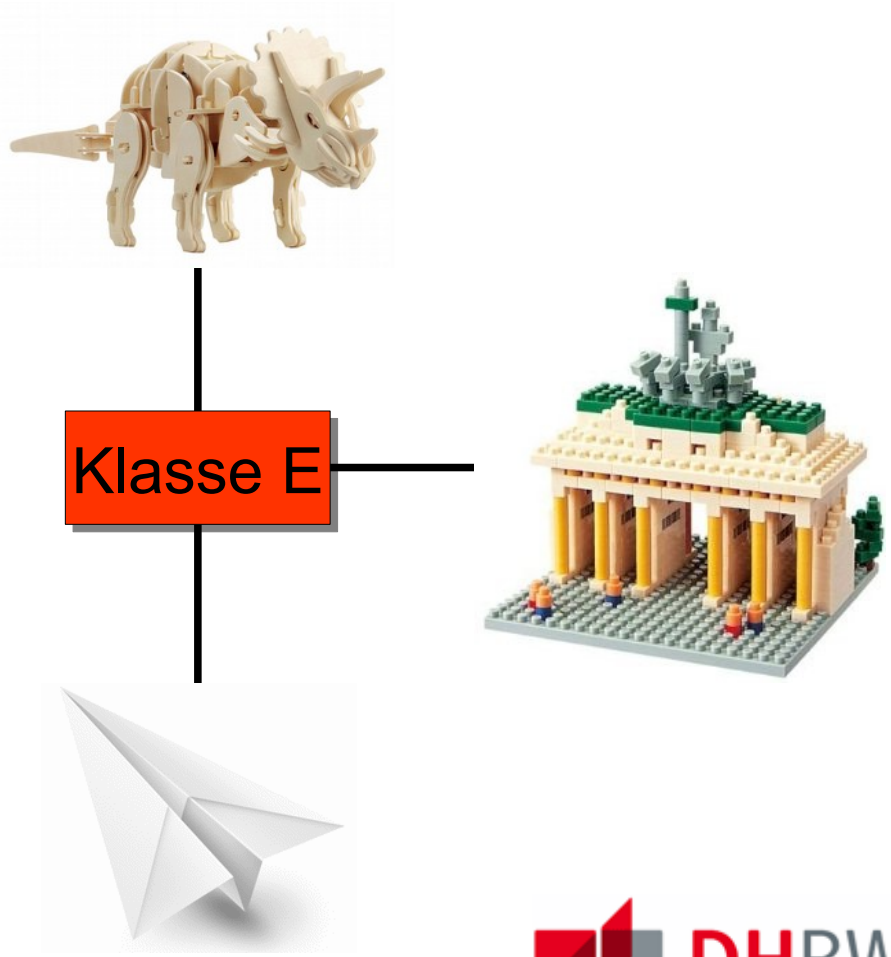
Ein Unit-Test isoliert die zu testende Komponente




DHBW

# Isolation durch Mocks

- Um eine Klasse isoliert testen zu können, müssen die Abhängig-keiten ersetzt werden
- Durch „Mock-Objekte“
- Minimal-Umsetzung der notwendigen Funktionalität (Fakes)
- „Gut genug“ für Test



# Unit Test Frameworks

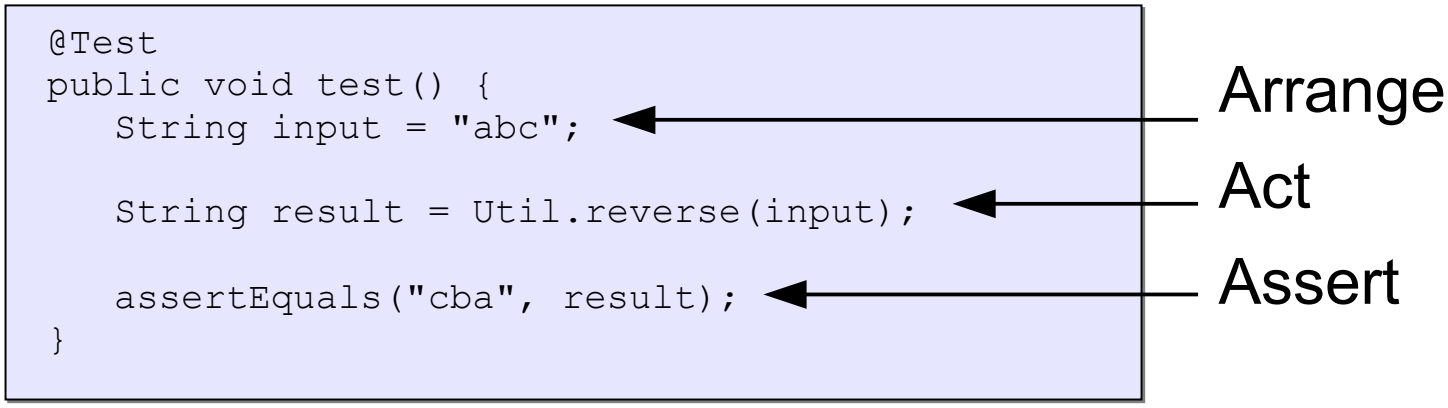
- Für jede Sprache existiert ein Unit Test Framework:

http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

- Sogenannte xUnit-Frameworks haben vergleichbaren Aufbau
  - Assertion-basierte Testformulierung
- Oft starke Trennung zwischen Produktiv- und Testcode

DHBW

# Normalform eines xUnit-Tests



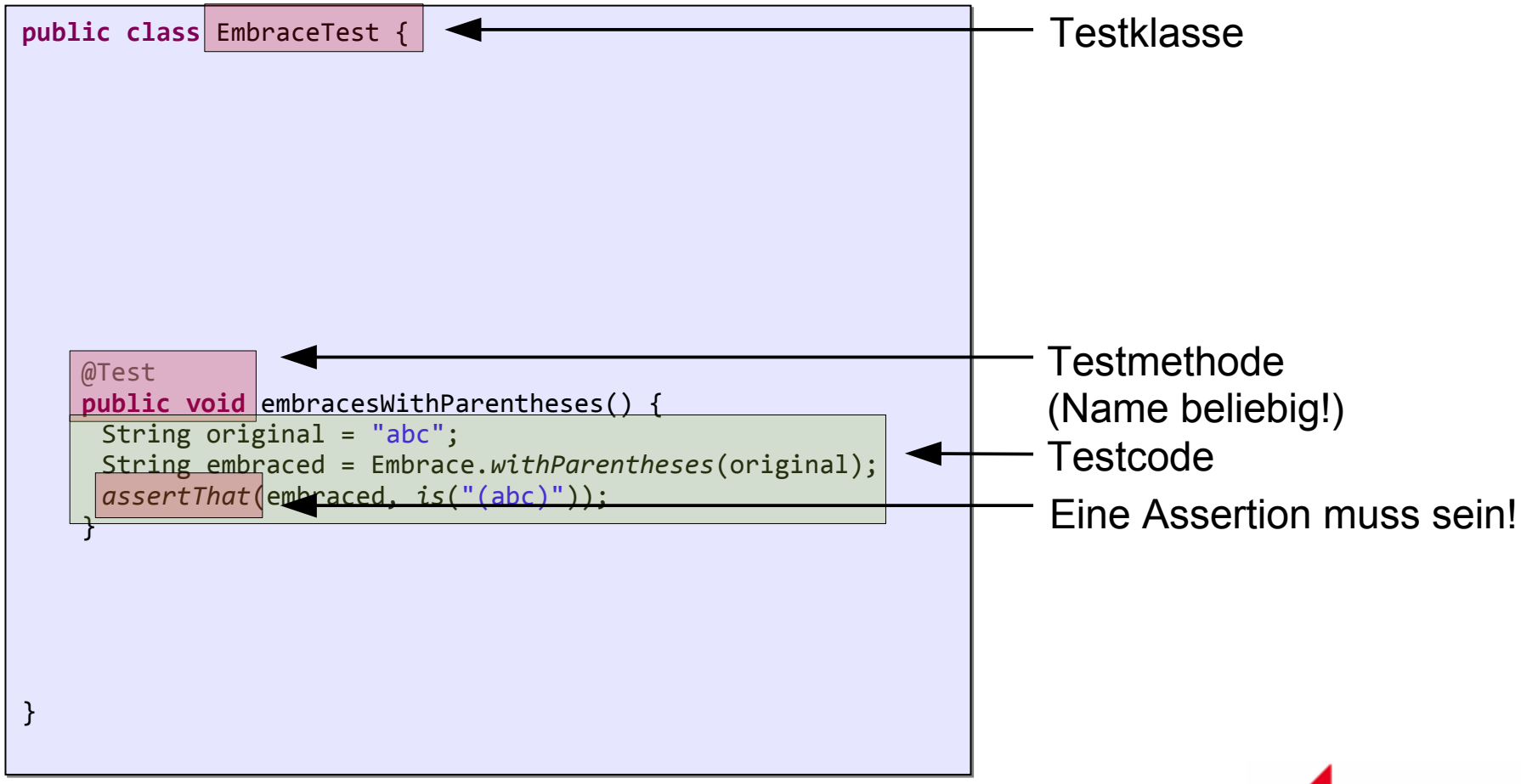

- Arrange: Initialisieren der Test-“Welt“
- Act: Ausführen der zu testenden Aktion
- Assert: Prüfen der Test-Zusicherungen

DHBW

# JUnit

- Entwickelt von Kent Beck und Erich Gamma
- Aktuell in Version 4.11
  - Große Unterschiede zur Version 3
- Sehr gute Integration in viele IDEs
- Java-Standard für Unit Tests

# Struktur eines JUnit-Tests




DHBW

# Struktur eines JUnit-Tests


Testklasse

Konstruktor sollte leer sein

Vorbereitungen für jede Testmethoden

Testmethode (Name beliebig!)

Testcode

Eine Assertion muss sein!

Aufräumarbeiten nach jeder Testmethode

DHBW

# Mehrere Testmethoden pro Testklasse

```
public class EmbraceTest {

    @Test
    public void embracesWithParentheses() {
      assertThat(Embrace.withParentheses("abc"), is("(abc)"));
    }

    @Test
    public void embracesWithBraces() {
      assertThat(Embrace.withBraces("xyz"), is("{xyz}"));
    }

    @Test
    public void embracesWithDoubleQuotes() {
      assertThat(Embrace.withDoubleQuotes("def"),
                                        is("\"def\""));
    }

    @Test
    public void embracesWithXMLTag() {
      assertThat(Embrace.withXMLTag("text", "tag"),
                                  is("<tag>text</tag>"));
    }
}
```

- Alle Testmethoden einer Klasse werden ausgeführt
- Reihenfolge der Testausführung nicht definiert
- Ergebnisse werden zusammengefasst

DHBW

# Mögliche Testergebnisse

- Success: Bestanden
  - Die Testmethode läuft durch
  - Alle Assertions sind erfüllt
    - Eine leere Testmethode besteht immer
- Failure: Wegen Assertion nicht bestanden
  - Eine Assertion in der Testmethode schlägt fehl
- Error: Wegen Fehler nicht bestanden
  - In der Testmethode tritt eine Ausnahme auf
    - Gewollte Exceptions müssen deklariert werden

# Eine übliche Testmethode

```
@Test
public void takesSkontoIntoAccount() {
    Customer testCustomer = new Customer();
    testCustomer.setSkonto(new Percent(3.0d));
    Euro price = new Euro(100.0d);

    Euro netto = price.getNettoFor(testCustomer);

    assertThat(netto.getAmount(), is(closeTo(97.0d, withMargin(1E-2))));
}
```

Spezielle Matchers (Hamcrest-Bibliothek)

```
private static double withMargin(double margin) {
    return margin;
}
```

Eigens geschriebener Code Squiggle

DHBW

# Eigenschaften guter Unit Tests

- Zeitnahe Erstellung zum Produktivcode
  - Eventuell sogar vor dem Produktivcode
- Die „A-TRIP“ Eigenschaften:
  - Automatic
  - Thorough
  - Repeatable
  - Independent
  - Professional

DHBW

# Automatic

- Die Tests müssen selbständig ablaufen
  - Keine manuellen Eingriffe notwendig
    - Kein Dialog-Wegklicken, keine Werteeingaben
- Die Tests müssen ihre Ergebnisse selbst überprüfen
  - Für jeden Test nur das Ergebnis „bestanden“ oder „nicht bestanden“ zulässig
  - „Nicht bestanden“ wird als „gebrochen“ bezeichnet
- Dadurch keinerlei Ansprüche bezüglich der Testdurchführung an den Durchführenden

# Thorough

- Gute Tests testen alles Notwendige
  - „notwendig“ bestimmt sich aus den Rahmenbedingungen
- Minimale, iterative Regel
  - Jede missionskritische Funktionalität ist getestet
  - Für jeden aufgetretenen Fehler existiert ein Testfall, der ein erneutes Auftreten verhindert
- Softwarefehler sind nicht gleichmäßig über ein System verteilt
  - Fehler „klumpen“ zusammen
  - Zusätzliche Tests im „Umfeld“ eines Fehlers

# Repeatable

- Jeder Test sollte jederzeit (automatisch) durchführbar sein und immer das gleiche Ergebnis liefern
  - Ergebnis (bestanden/nicht bestanden) unabhängig von der Umgebung
    - Beliebte Problemquellen: Datum und Zufall
    - Dateisystemzugriffe sind plattformabhängig
- Ein Test, der spontan bricht, obwohl nichts verändert wurde, ist selbst fehlerhaft
  - Ein fehlerhafter Test ist schlechter als keiner

# Independent

- Tests müssen jederzeit in beliebiger Zusammenstellung und Reihenfolge funktionsfähig sein
  - Keine implizite Abhängigkeiten zwischen den Tests
    - z.B. persistente Datenstrukturen wie Datenbanken oder Dateisysteme
- Jeder Test testet genau einen Aspekt der Komponente
  - Bei Testbruch sollte die Ursache möglichst direkt ableitbar sein

DHBW

# Professional

- Testcode ist auch produktionsrelevanter Code
  - Allerdings wird Testcode selbst nicht automatisch getestet
  - Fehler in Tests sind ebenfalls teuer in ihrer Behebung
- Testcode sollte so leicht verständlich wie möglich sein
- Testcode als Selbstzweck (Anzahl Tests erhöhen) ist nicht sinnvoll
  - Tests für unrelevante Aspekte sind ebenfalls nicht sinnvoll

# Ein üblicher Test

```
@Test
public void takesSkontoIntoAccount() {
    Customer testCustomer = new Customer();
    testCustomer.setSkonto(new Percent(3.0d));
    Euro price = new Euro(100.0d);

    Euro netto = price.getNettoFor(testCustomer);

    assertThat(netto.getAmount(), is(closeTo(97.0d, withMargin(1E-2))));
}
```

DHBW

# Testen auf Exceptions

```
@Test
public void needsMeaningfulSkonto() {
    Customer testCustomer = new Customer();
    testCustomer.setSkonto(null); // This means trouble!
    Euro price = new Euro(100.0d);
    try {
        Euro netto = price.getNettoFor(testCustomer);
    } catch (NullPointerException e) {
    }
}
```

Geht auch ohne Exception grün durch

```
}
```

# Testen auf Exceptions

```
@Test
public void needsMeaningfulSkonto() {
   Customer testCustomer = new Customer();
   testCustomer.setSkonto(null); // This means trouble!
   Euro price = new Euro(100.0d);
   try {
      Euro netto = price.getNettoFor(testCustomer);
      fail(„Need to throw a NullPointerException here.“)
   } catch (NullPointerException e) {
      assertThat(e.getMessage(), is(„No skonto set“));
   }
}
```

DHBW

# Testen auf Exceptions

```
@Test(expected=NullPointerException.class)
public void needsMeaningfulSkonto() {
    Customer testCustomer = new Customer();
    testCustomer.setSkonto(null); // This means trouble!
    Euro price = new Euro(100.0d);
    Euro netto = price.getNettoFor(testCustomer);
}
```

DHBW

# Besondere Code-Konstrukte

- Berechnungen und Konditionalstrukturen sind besonders anfällig für menschliche Fehler
  - Jede Berechnung testen
  - Jede Konditionalstruktur testen
- Konditionalstrukturen in eigene, testbare Methode auslagern
  - `protected boolean isSomething() {...}`
  - Entscheidungsraum nach Möglichkeit im Test abdecken

DHBW

# Genauigkeit von Vergleichen

- Zeichenketten- und Ganzzahl-Vergleiche sind diskret (ohne Toleranzbereich)
- Fließkommazahl-Vergleiche sind aufgrund ihres Speicherformats mit einer Toleranz behaftet:

```
assertThat(doubleValue(), is(closeTo(1.0d, 1E-2)));
```

- Diese Toleranz kann im Format 1E-n mit n = Anzahl Nachkommastellen Genauigkeit angegeben werden

DHBW

# Arithmetische Kuriositäten

- In Java gibt es keinen DivisionByZeroError o.ä.:
  - 1.0d / 0.0d => Double.POSITIVE_INFINITY
  - 1.0d / -0.0d => Double.NEGATIVE_INFINITY
- Es gibt einen fiesen Unterschied in der Semantik von MIN_VALUE:
  - Integer.MIN_VALUE = größter negativer Wert
  - Double.MIN_VALUE = kleinster positiver Wert
- Java kennt keinen IntegerOverflowError o.ä.:
  - (Integer.MAX_VALUE + 1) == Integer.MIN_VALUE

# Wie teste ich meine Tests?

- Testabdeckung (Code Coverage) bestimmen
- Bewusst (und temporär intern!) Probleme für den Test in den Produktivcode einbauen
  - Tests müssen auf diese Probleme reagieren
- Ein Werkzeug einsetzen, das den Produktivcode im Test willkürlich verändert und beschädigt
  - Die Tests sollten solche Veränderungen registrieren

# Testabdeckung

- Code Coverage hat verschiedene Bezugspunkte
  - Line Coverage
  - Branch Coverage
- Wichtig: Immer Bezugspunkt angeben, die Werte können stark unterschiedlich sein

DHBW

# Branch Coverage

- Jede Entscheidung kann auf verschiedenen Pfaden durchlaufen bzw. verlassen werden

```
public class SomeTest {
    @Test
    public void hasAnswerOnEverything() {
        assertThat(new Some().thing(true), is(42));
    }
}
```

```
public class Some {
    public int thing(boolean mode) {
        if (mode) {
            return 42;
        }
        return 17;
    }
}
```

Branch Coverage = 50%

DHBW

# Line Coverage

- Jede Zeile Code wird durch die Tests entweder durchlaufen oder nicht

```
public class SomeTest {
    @Test
    public void hasAnswerOnEverything() {
        assertThat(new Some().thing(true), is(42));
    }
}
```

```
public class Some {
    public int thing(boolean mode) {
        if (mode) {
            return 42;
        }
        return 17;
    }
}
```

Line Coverage = 66,6%

DHBW

# Hohe Testabdeckungen

- Hohe Testabdeckung ist hilfreich
  - Größer 80% (LC/BC) ist sehr aufwendig
- Eine Testabdeckung von 100% (LC) bedeutet noch nicht, dass alles getestet wurde
  - In den Tests wurden nur alle Zeilen durchlaufen
- Im Testabdeckungsbericht sind letztlich nur die roten Zeilen aussagekräftig

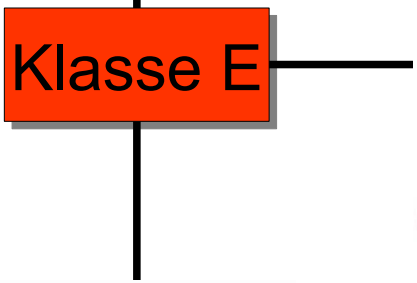
# Mock-Objekte

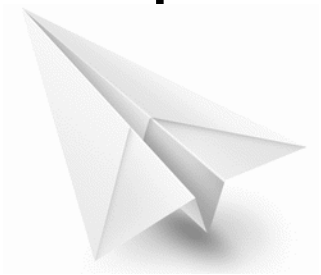
- Mock-Objekte, kurz „Mocks“ sind einfache Stellvertreter für „echte“ Objekte
- Ersetzen Abhängigkeiten während eines Tests
- Vergleichbar dem Licht-Double in der Film-Industrie
  - Hat nur die relevanten Eigenschaften mit dem echte Star gemeinsam

DHBW

# Wiederholung: Isolation durch Mocks

- Um eine Klasse isoliert testen zu können, müssen die Abhängig-keiten ersetzt werden
- Durch „Mock-Objekte“
- Minimal-Umsetzung der notwendigen Funktionalität (Fakes)
- „Gut genug“ für Test



DHBW

# Einsatz von Mocks

- Selbst programmierte Mocks sind aufwendig
  - Der Einsatz eines Mock-Tool lohnt sich
- Mocks müssen für den jeweiligen Einsatzzweck „trainiert“ werden
- Jedes Mock durchläuft drei Phasen:
  - Einlernen (Training-Phase)
  - Abspielen (Einsatz-Phase)
  - Überprüfen (Verifikation-Phase)

# Mocks für Java/JUnit

- Hier: Mocks am Beispiel von EasyMock
- Es gibt zahlreiche Mock-Frameworks für Java, z.B.
  - JMock2 (akademisch, sehr penibel)
  - Mockito (modern, sehr mächtig)
  - PowerMock (Aufsatz auf Mockito)
  - JMockit und JMockit2
  - Spock (u.a. ein Mock-Framework)

# Ein einfacher Test mit Mocks

- Klasse zum Laden von Person-Objekten aus Datenbank
- Wir wollen die Datenbank nicht mittesten
- Datenbank implementiert das folgende Interface:

```
public interface Database {

      public List<Long> getHandlesOfType(Class<? extends Object> type);

      public Properties loadDataOf(Long handle) throws IOException;

      public int getLastErrorNumber();
}
```

# Ein einfacher Test mit Mocks

```
public class PersonLoaderTest {
  @Test
  public void loadsPerson() throws IOException {
    Database database = EasyMock.createMock(Database.class);
    List<Long> personHandles = Arrays.asList(42L);
    EasyMock.expect(database.getHandlesOfType(Person.class)).andReturn(personHandles);
    EasyMock.expect(database.loadDataOf(42L)).andReturn(propertiesOfPerson0());
    EasyMock.replay(database);

    PersonLoader loader = new PersonLoader(database);

    List<Person> allPersons = loader.allPersons();

    assertThat(allPersons.size(), is(1));
    assertThat(allPersons.get(0).name(), equalTo("Max Mustermann"));

    EasyMock.verify(database);
  }

  private Properties propertiesOfPerson0() {
    Properties result = new Properties();
    result.setProperty("forename", "Max");
    result.setProperty("surname", "Mustermann");
    return result;
  }
}
```

static imports

# Ein einfacher Test mit Mocks

```
public class PersonLoaderTest {
  @Test
  public void loadsPerson() throws IOException {
    Database database = createMock(Database.class);                    // database ist ein Mock
    List<Long> personHandles = Arrays.asList(42L);
    expect(database.getHandlesOfType(Person.class)).andReturn(personHandles);          // Einlernen
    expect(database.loadDataOf(42L)).andReturn(propertiesOfPerson0());                 // Einlernen
    replay(database);                                                  // Umschalten auf Wiedergabe

    PersonLoader loader = new PersonLoader(database);

    List<Person> allPersons = loader.allPersons();

    assertThat(allPersons.size(), is(1));
    assertThat(allPersons.get(0).name(), equalTo("Max Mustermann"));

    verify(database);                                                  // Überprüfen der Annahmen
  }

  private Properties propertiesOfPerson0() {
    Properties result = new Properties();
    result.setProperty("forename", "Max");
    result.setProperty("surname", "Mustermann");
    return result;
  }
}
```

# Analyse des Tests

- Ein Mock muss explizit erstellt und trainiert werden
- Ohne die beiden expect()-Aufrufe würde das Mock-Objekt in der Wiedergabe-Phase keine Werte liefern
- Durch replay() wird das Mock von der Lern- (oder Aufnahme-) Phase in die Wiedergabe-Phase geschaltet
- verify() überprüft die Erwartungen

# Normalform von Tests mit Mocks

```
public class PersonLoaderTest {
  @Test
  public void loadsPerson() throws IOException {
    Database database = createMock(Database.class);
    List<Long> personHandles = Arrays.asList(42L);
    expect(database.getHandlesOfType(Person.class)).andReturn(personHandles);
    expect(database.loadDataOf(42L)).andReturn(propertiesOfPerson0());
    replay(database);

    PersonLoader loader = new PersonLoader(database);

    List<Person> allPersons = loader.allPersons();

    assertThat(allPersons.size(), is(1));
    assertThat(allPersons.get(0).name(), equalTo("Max Mustermann"));

    verify(database);
  }
}
```

Capture

Arrange

Act

Assert

Replay

Verify

- Ein mock-basierter Test in seiner Normalform hat fünf Phasen

DHBW

# Mocks mit Erwartungen

```
public class FileDeleterTest {
  @Test
  public void deletesOnlyEmptyFiles() throws IOException {
    File fileMock = createStrictMock(File.class);
    expect(fileMock.exists()).andReturn(true);
    expect(fileMock.isFile()).andReturn(true);
    expect(fileMock.length()).andReturn(0L);
    expect(fileMock.delete()).andReturn(true);
    expect(fileMock.exists()).andReturn(false);
    replay(fileMock);

    DeleteOnly.ifFile(fileMock).isEmpty();

    verify(fileMock);
  }
}
```

„strict“-Mocks beachten die genaue Reihenfolge der Trainingsaufrufe

Statt expliziter Assertions verlangt dieser Test nur, dass die Erwartungen des Mock erfüllt sind

- Die Erwartungen eines Mock-Objekts spezifizieren ein Aufruf-Protokoll, dem das Objekt unter Test genügen muss

# Voraussetzungen für Mocks

- Statische Methoden und vergleichbare Konstrukte sind problematisch
- Sinnvoller Einsatz nur, wenn Dependency Injection möglich
- Tiefe Abhängigkeits-Strukturen sind nur aufwendig nachzubilden
  - Einhalten des Law of Demeter
  - Lose Kopplung verringert Wissen, das ein Mock haben muss

DHBW

# Zusammenfassung Mocks

- Mächtige, werkzeuggestützte Technik, um Objekte in Tests zu isolieren
  - Und Aufruf-Protokolle zu testen
- Mock-Objekte werden direkt vor Benutzung im Test definiert
  - Sind nicht Teil des Produktivcodes
- Aufpassen: Man testet relativ leicht nur noch, dass das Mock-Framework funktioniert (Zirkelschluss-Test)

# Test First

Klassisches Vorgehen:

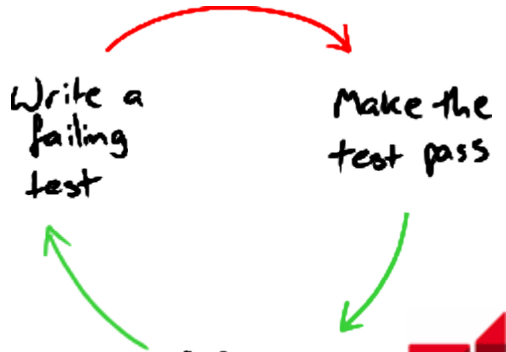
- Funktionalität planen (im Kopf)
- Funktionalität programmieren
- Funktionalität umschreiben (Refactoring)
- Tests schreiben
- Fehler beheben

DHBW

# Test First

Vorgehensweise bei Test First:

- Funktionalität planen (im Kopf)
- Test schreiben
- Funktionalität programmieren
- Refactoring




DHBW

# Test First

- Test First und Test Driven Development stellen Tests über Produktivcode
- Produktivcode wird nur geschrieben, um einen Test zu erfüllen
  - Minimale Implementierung ausreichend
- Tests weisen den Weg
- Nach jedem Schritt kann die Implementierung verbessert werden

DHBW

# Test First

| Vorteile | Nachteile |
| --- | --- |
| Tests sind nicht optional | Erfordert Einarbeitung |
| Vollständige Testabdeckung | Höherer Aufwand |
| Weniger Fehler | Funktioniert nicht immer |
| Automatische Spezifikation | Tests sind nicht optional |
| Kleine Komponenten | |
| Stabile Implementierung | |

# Test First Praxisbeispiel

- Zahl in römischen Ziffern ausdrücken

| Die heute verwendeten römischen Ziffern | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Zeichen** | I | V | X | L | C | D | M | ↁ | ↀ |
| **Wert** | 1 | 5 | 10 | 50 | 100 | 500 | 1.000 | 5.000 | 10.000 |

- Die Römer kannten keine Null
- Es gibt eine seltsame Schreibweise:

4 = IV

9 = IX

DHBW

# Praxisbeispiel Schritt 1a

```
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }
}
```

# Praxisbeispiel Schritt 1b (+1c)

```java
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }
}
```

```java
public class RomanNumeral {
  public static String of(int number) {
    return "I";
  }
}
```

DHBW

# Praxisbeispiel Schritt 2a

```java
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }

  @Test
  public void convertsTwo() {
    assertThat(RomanNumeral.of(2), is("II"));
  }
}
```

```java
public class RomanNumeral {
  public static String of(int number) {
    return "I";
  }
}
```

DHBW

# Praxisbeispiel Schritt 2b (+2c)

```java
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }

  @Test
  public void convertsTwo() {
    assertThat(RomanNumeral.of(2), is("II"));
  }
}
```

```java
public class RomanNumeral {
  public static String of(int number) {
    if (2 == number) {
      return "II";
    }
    return "I";
  }
}
```

DHBW

# Praxisbeispiel Schritt 3a

```java
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }

  @Test
  public void convertsTwo() {
    assertThat(RomanNumeral.of(2), is("II"));
  }

  @Test
  public void convertsThree() {
    assertThat(RomanNumeral.of(3), is("III"));
  }
}
```

```java
public class RomanNumeral {
  public static String of(int number) {
    if (2 == number) {
      return "II";
    }
    return "I";
  }
}
```

DHBW

# Praxisbeispiel Schritt 3b

```java
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }

  @Test
  public void convertsTwo() {
    assertThat(RomanNumeral.of(2), is("II"));
  }

  @Test
  public void convertsThree() {
    assertThat(RomanNumeral.of(3), is("III"));
  }
}
```

```java
public class RomanNumeral {
  public static String of(int number) {
    if (3 == number) {
      return "III";
    }
    if (2 == number) {
      return "II";
    }
    return "I";
  }
}
```

Duplication!

# Praxisbeispiel Schritt 3c

```java
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }

  @Test
  public void convertsTwo() {
    assertThat(RomanNumeral.of(2), is("II"));
  }

  @Test
  public void convertsThree() {
    assertThat(RomanNumeral.of(3), is("III"));
  }
}
```

```java
public class RomanNumeral {
  public static String of(int number) {
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < number; i++) {
      result.append("I");
    }
    return result.toString();
  }
}
```

DHBW

# Praxisbeispiel Schritt 4a

```java
public class RomanNumeralTest {
  @Test
  public void convertsOne() {
    assertThat(RomanNumeral.of(1), is("I"));
  }

  @Test
  public void convertsTwo() {
    assertThat(RomanNumeral.of(2), is("II"));
  }

  @Test
  public void convertsThree() {
    assertThat(RomanNumeral.of(3), is("III"));
  }

  @Test
  public void convertsFour() {
    assertThat(RomanNumeral.of(4), is("IV"));
  }
}
```

```java
public class RomanNumeral {
  public static String of(int number) {
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < number; i++) {
      result.append("I");
    }
    return result.toString();
  }
}
```

DHBW

# Abschluss Praxisbeispiel

- Fortsetzen als Fingerübung („Kata“)
- Siehe auch

http://securesoftwaredev.com/2011/12/05/practicing-tdd-using-the-roman-numerals-kata/

- Viele Screencasts im Netz
- Entscheidungshilfe für Implementierung:

http://cleancoder.posterous.com/the-transformation-priority-premise

- Achtung! TF/TDD ist anfangs sehr ungewohnt und schwierig

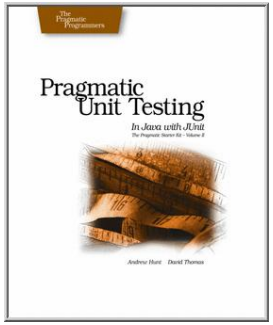
DHBW

# Härtefälle für (Unit) Tests

- Abtesten der visuellen Eigenschaften von Widgets oder Webseiten
- Asynchrone Ereignisfolgen
  - Threading-Verhalten
- Komplexe Berechnungen
  - Test-Triangulation erforderlich
- Zufalls- und datumsabhängige Funktionalität

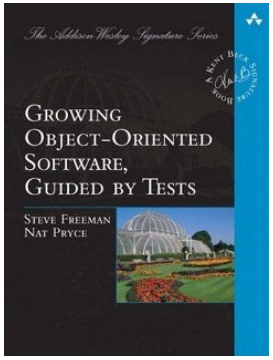
DHBW

# Literatur zu Unit Tests



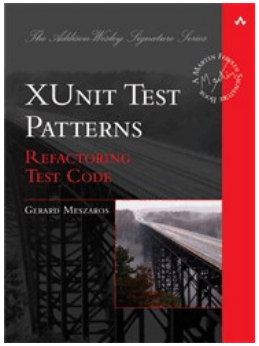

## Pragmatic Unit Testing
Andrew Hunt, Dave Thomas

Leicht verständliches und kompaktes Einsteiger-Buch. Für Java/JUnit und C#/NUnit erhältlich.




## Growing Object-Oriented Software, Guided by Tests
Steve Freeman, Nat Pryce

Klare Leseempfehlung. Ein gutes Buch über moderne Softwareentwicklung, mit Beispielprojekt in TDD-Manier

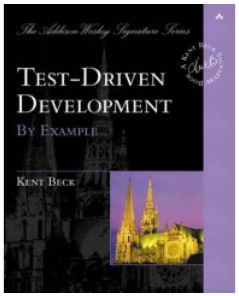



## xUnit Test Patterns
Gerard Meszaros

Erklärt die Motivation, Struktur und Implementierung von Tests. Gibt klare Schemata für gute Tests.

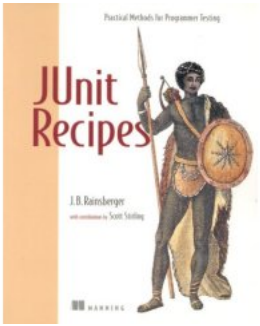
DHBW

# Literatur zu Unit Tests




Test Driven Development, by Example
Kent Beck

Unaufgeregte, sehr gründliche Einführung in Test Driven Development




JUnit Recipes
J. B. Rainsberger

Für jedes Test-Problem ist hier die Lösung enthalten.
Nachschlagewerk, leicht veraltet

DHBW