

# **Automatisierte Tests**

Stützpfeiler der Langlebigkeit

# Software und Fehler

- Menschen schreiben Software
- Menschen machen Fehler

→ Software enthält Fehler

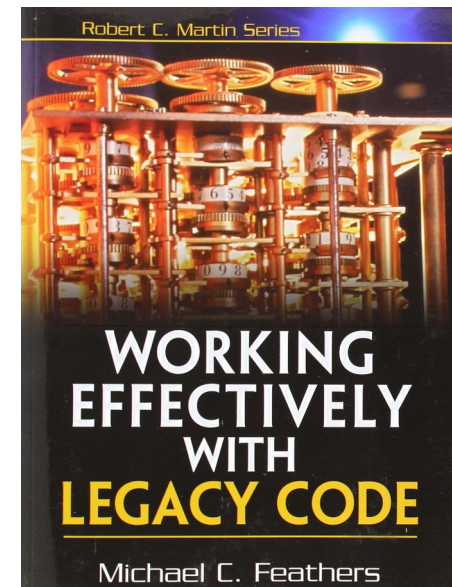
# Fehler binden Ressourcen

- Jeder Fehler bindet Ressourcen
  - Geld, Zeit
  - Aufmerksamkeit, Nerven, Vertrauen
- Die Kosten eines Fehlers steigen mit der Zeit seiner Existenz
  - Das ist zumindest eine häufig geäußerte Theorie
  - Kosten selbst, wenn man ihn sofort bemerkt und beseitigt
- Multiplikator bei Veröffentlichung des Fehlers

• <http://www.sicpers.info/2012/09/an-apology-to-readers-of-test-driven-ios-development>

# Legacy Code

- In Deutschland sind Softwaretests gesetzlich vorgeschrieben
  - Nicht testen ist „grob fahrlässig“
- Ein vorhandener Test unterscheidet gewollte Funktionen von zufälligen Features
- Passende Definition
  - Legacy Code is Code without Test
- Einige Entwickler verwenden Tests als Hilfsmittel für ihre Arbeit
  - Testgetriebene Entwicklung



# Eine Klassifikation von Tests

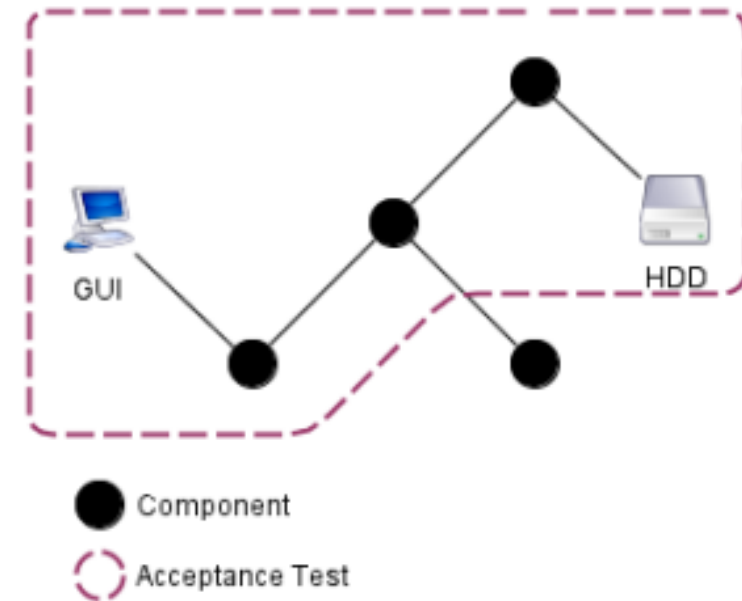
- Begriffsvielfalt in Bezug auf Testarten
- Eine mögliche Bezeichnung:
  - Leistungstests
  - Akzeptanztests
  - Integrationstests
  - Komponententests (Unit Tests)

# Leistungstests

- Vollständiges System wird gestartet
  - Eventuell mit zusätzlicher Sensorik für Kenngrößen
- Echte bzw. Referenzumgebung
  - Echte Datenbank, Hardware
  - Zahlreiche simulierte Klienten/Benutzer
- Parallele Bedienungsdurchführung mit Mitteln des Benutzers
- Ziel: Echte Lastbedingungen erzeugen und Kenngrößen ermitteln
- Kein Ziel: Überprüfen der Korrektheit

# Akzeptanztests

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



# Beispiel für Akzeptanztest

```
@Test
public void maintenanceStatusIsExportedToKTF() throws Exception {
    center().startWithArchiveDirectory(
        new LocalVirtualFile(temporaryFolder().newFolder("archive")));
    center().stationArchive().assertIsEmpty();

    StationManagementDialogDriver stationManagementDialog =
        StationManagementDialogDriver.openOn(center());
    stationManagementDialog.clearStationTable();
    stationManagementDialog.createStation().withName(stationName).create();
    stationManagementDialog.close();
    setupExportConfigurationFor(stationName, exportName);

    final StationMock station = new StationMock(stationName,
        new LocalVirtualFile(temporaryFolder().getRoot()));

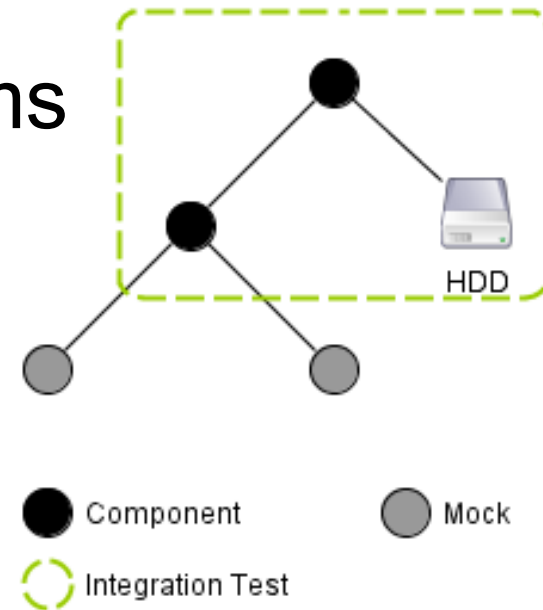
    delphinValuesWithMaintenanceFlagsIn(station.currentPackage()).send();
    WaitUntilPackage.from(stationName).isProcessedWithin(Wait.LONGER)
        .asShownOn(center().statusbar());

    Wait.SHORT.perform();
    assertThatMaintenanceFlagsAreRepresentedInExportedKTF();
}
```



# Integrationstests

- Nur die relevanten Teile des Systems werden gestartet
- Nicht zu testende Systemteile sind durch Stellvertreter ersetzt
- Durchführung mittels Testframework (Methodenaufrufe)
  - Interaktion der Systemteile untereinander
- Ziel: Zusammenspiel der Komponenten sicherstellen
- Kein Ziel: Einzelheiten der Komponenten testen



# Beispiel für Integrationstest

```
public class FtpVirtualFileIntegrationTest {

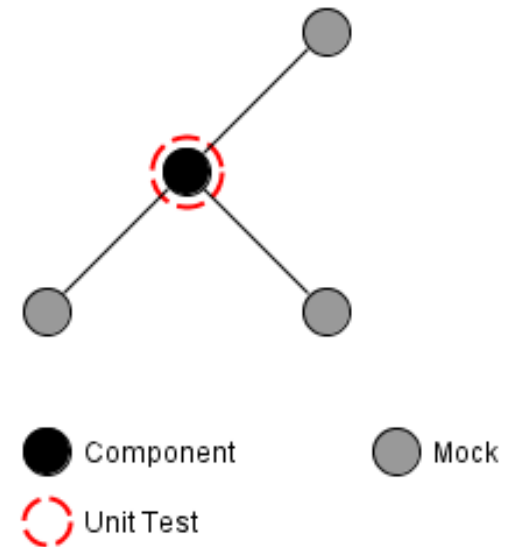
    private FTPClient client;
    private FTPClient fakeClient;

    @Before
    public void setUp() throws Exception {
        this.client = new FTPClient();
        this.fakeClient = mock(FTPClient.class);
    }

    @SuppressWarnings("boxing")
    @Test
    public void pathResolutionProblemsResultInSuppliedPath() throws Exception {
        when(this.fakeClient.isConnected()).thenReturn(true);
        when(this.fakeClient.changeWorkingDirectory(anyString())).thenThrow(
            new FtpVirtualFileException(EXISTING_DIRECTORY_NAME));
        FtpVirtualFile target = mockedTarget(NON_EXISTING_FILE_PATH);
        assertThat(target.getCanonicalName(), equalTo(NON_EXISTING_FILE_PATH));
    }
}
```

# Komponententests (Unit Tests)

- Nur der relevante Teil des Systems wird gestartet
- Alle anderen Systemteile sind durch Stellvertreter ersetzt
- Durchführung mittels Testframework (Methodenaufrufe)
  - Prüfen der Rückgabewerte
- Ziel: Korrekte Implementierung der Komponente (Unit) sicherstellen
- Kein Ziel: Zusammenspiel oder Performanz



# Beispiel für Unit Test

```
public class KeepOnlyTest {

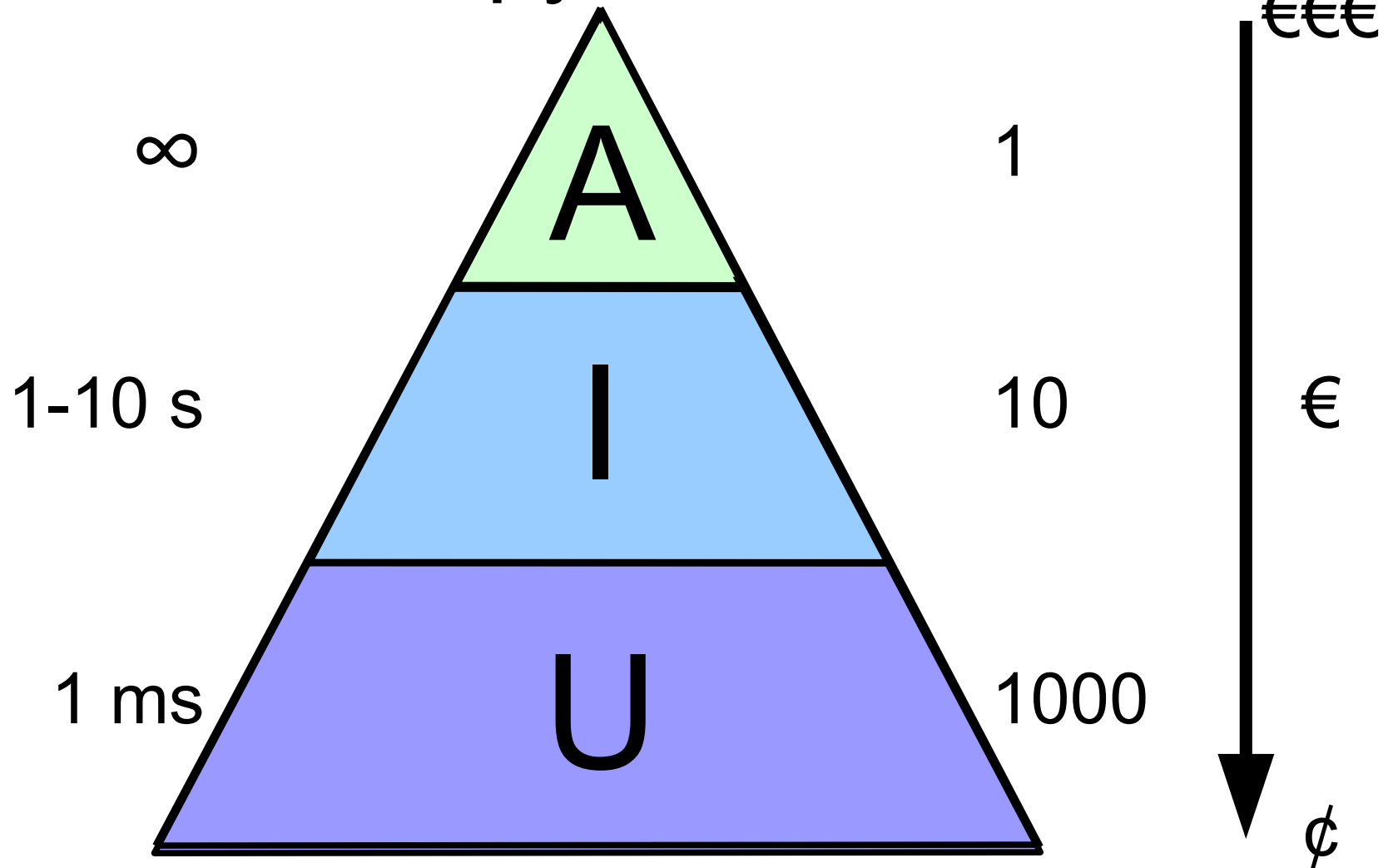
    @Test
    public void omitsSpecialCharacters() {
        assertEquals(
            "FASSDATEN",
            KeepOnly.letters().andDigits().from("++ FASSDATEN ++"));
        assertEquals(
            "123abcDEF",
            KeepOnly.letters().andDigits().from("+1,2.3-a*b#c_D?E/F$"));
    }

    @Test
    public void omitsSpecialCharactersButNoWhitespace() {
        assertEquals(
            " FASSDATEN ",
            KeepOnly.letters().andDigits().andWhitespaces().from("++ FASSDATEN ++"));
        assertEquals(
            "123  abc  DEF",
            KeepOnly.letters().andDigits().andWhitespaces().from("+1,23  ab#c  DE/F$"));
    }
}
```

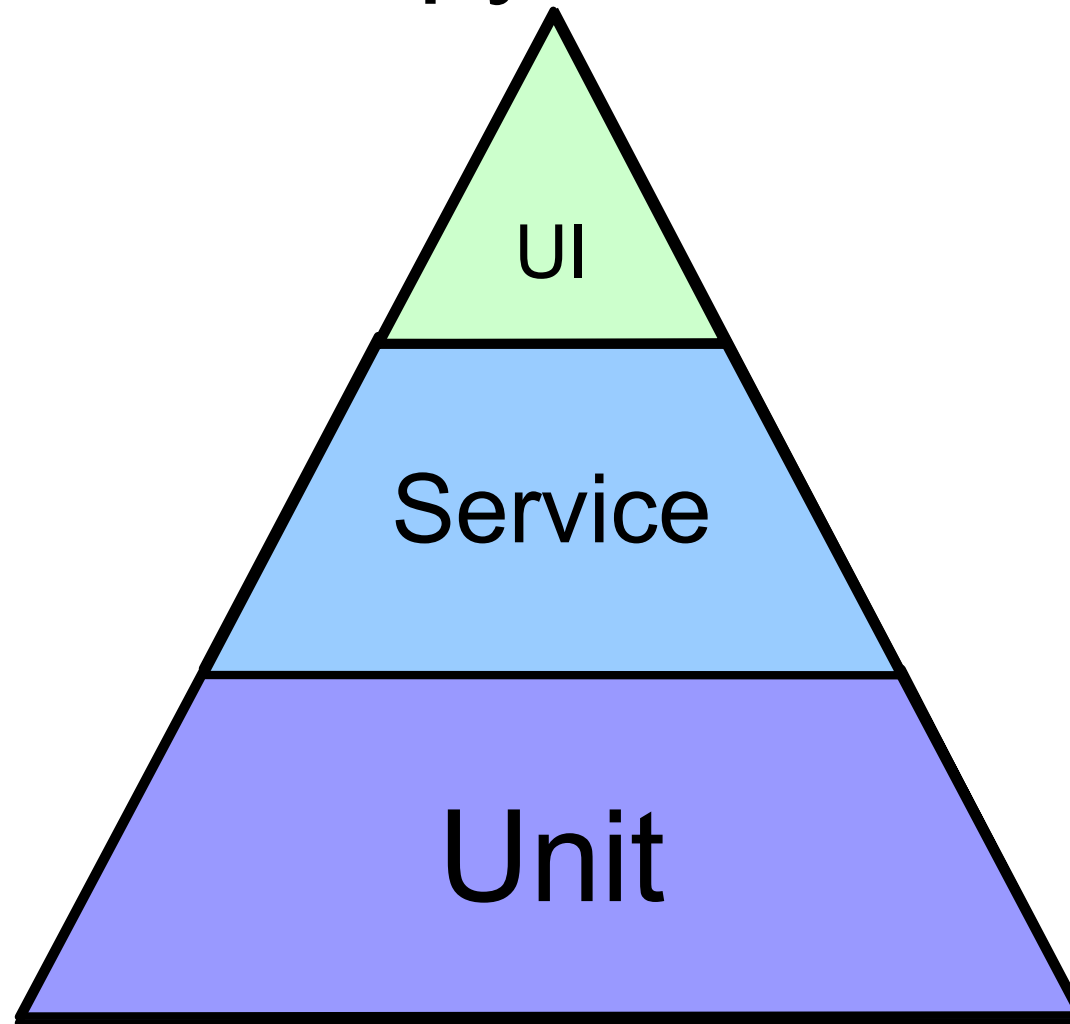
# Ernährungspyramide



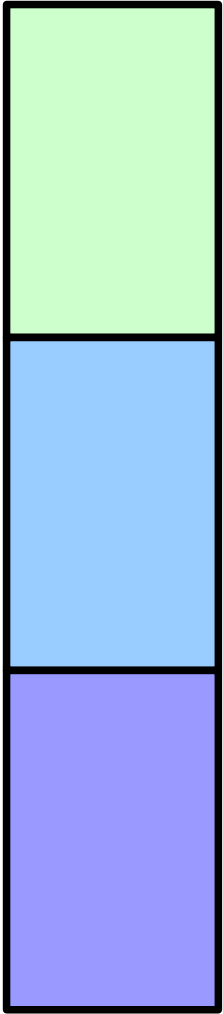
# Testpyramide



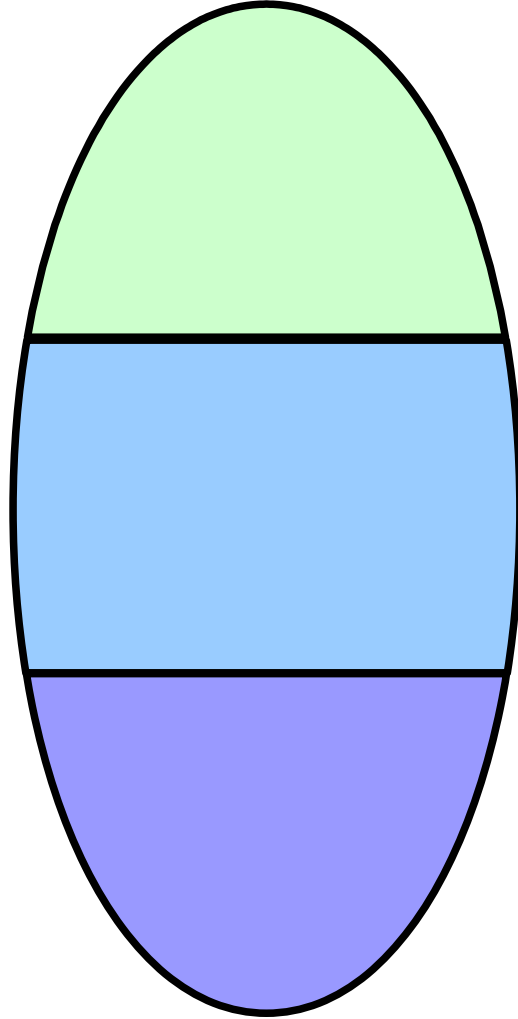
# Testpyramide



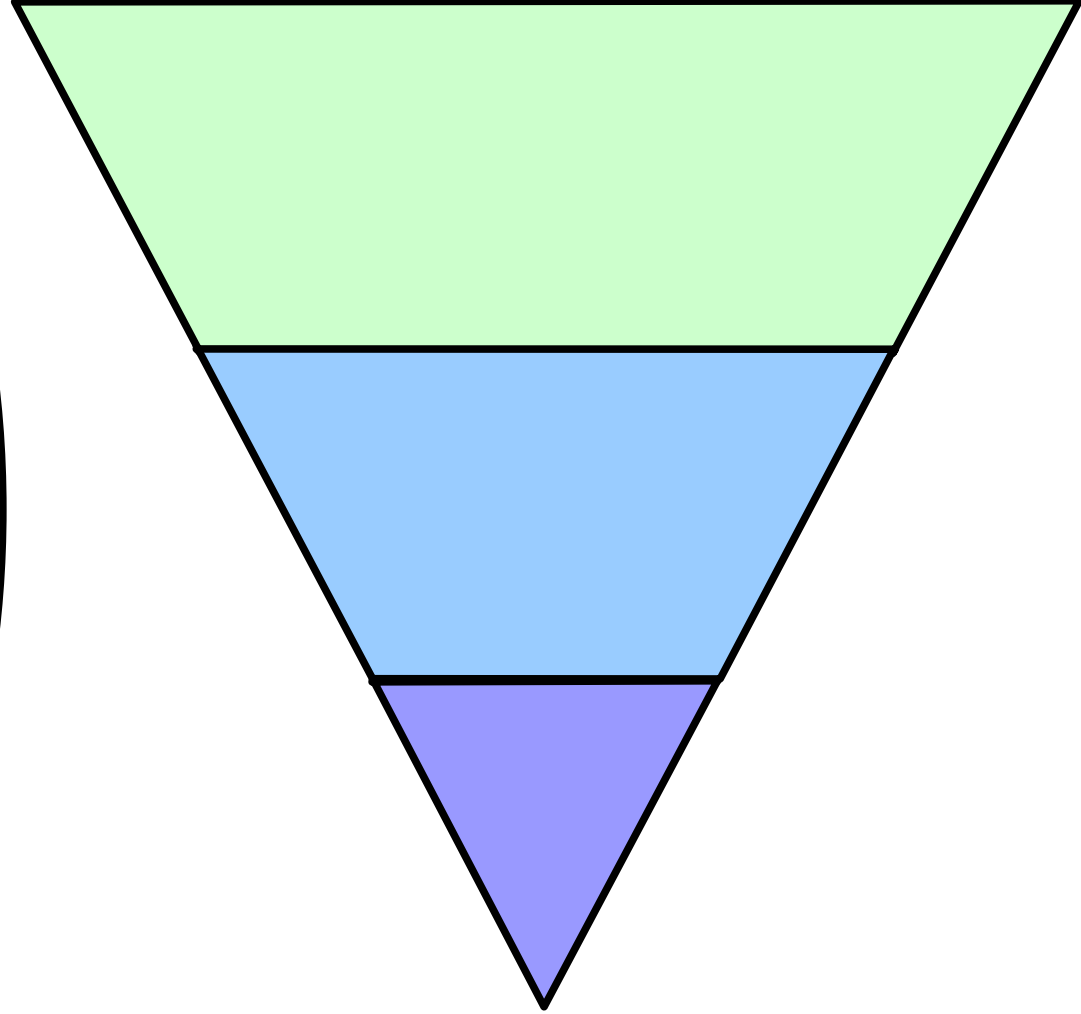
# Ungünstige Testausstattungen



Der Turm



Das Ei



Die Eistüte

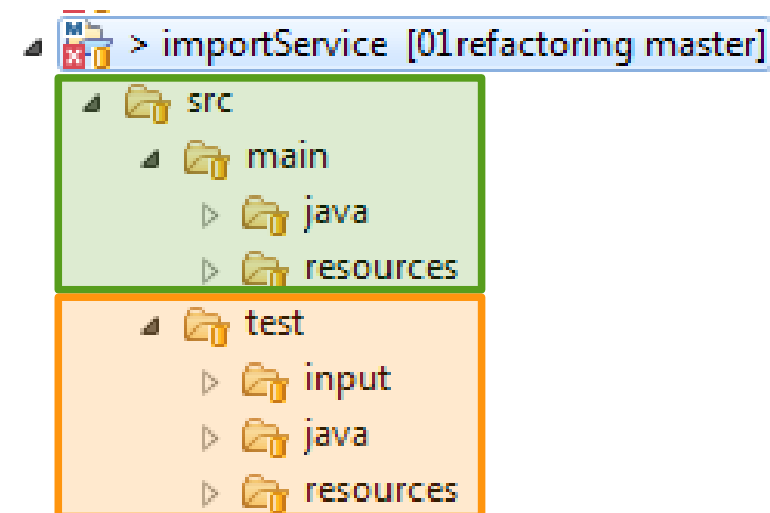


# Frameworks für Testarten (Java)

- Akzeptanztests
  - Arquillian, FitNesse, Cucumber, Conordion, Easyb
  - Selenium, Sahi
- Integrationstests
  - Arquillian
  - Sehr wenige Alternativen
- Unit Tests
  - JUnit, TestNG, Spock
  - Sehr viele weniger bekannte Alternativen

# xUnit-Frameworks

- XUnit-Testframeworks haben einen vergleichbaren Aufbau
  - Gleiches Konzept für Testdurchführung
- Assertion-basierte Überprüfungen
  - `assertThat(calculatedValue, is(expectedValue))`
- Oft starke Trennung zwischen Produktiv- und Testcode



# Struktur eines JUnit-Tests

```
public class EmbraceTest {
```

← Testklasse

```
@Test
```

← Testmethode  
(Name beliebig!)

```
public void embracesWithParentheses() {
```

```
    String original = "abc";
```

```
    String embraced = Embrace.withParentheses(original);
```

← Testcode

```
    assertThat(embraced, is("(abc)"));
```

← Eine **Assertion**  
sollte sein!

```
}
```

```
}
```

# Vollständige Struktur von JUnit-Tests

```
public class EmbraceTest {  
    public EmbraceTest() { /* empty */ }  
  
    @Before  
    public void anythingBeforeEachTest() {  
        // nothing in this example  
    }  
  
    @Test  
    public void embracesWithParentheses() {  
        String original = "abc";  
        String embraced = Embrace.withParentheses(original);  
        assertThat(embraced, is("(abc)"));  
    }  
  
    @After  
    public void anythingAfterEachTestGoesHere() {  
        // nothing, too  
    }  
}
```

**Testklasse**

**Konstruktor**  
(sollte leer sein)

**Vorbereitung** für jede Testmethode  
(möglichst leer)

**Testmethode**  
(Name beliebig!)

**Testcode**

Eine **Assertion** sollte sein!

**Aufräumarbeiten** nach jeder Testmethode  
(möglichst leer)

# Mehrere Tests 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\""));  
    }  
}
```

- Alle Testmethoden einer Testklasse werden ausgeführt
- Die Reihenfolge der Testausführung ist nicht definiert!
- Alle Ergebnisse werden zu einem Gesamtergebn zusammengefasst

# AAA-Normalform eines Unit Tests

```
@Test
public void embracesWithParentheses() {
    String original = "abc";
    String embraced =
        Embrace.withParentheses(original);
    assertThat(embraced, is("(abc)"));
}
```

← Arrange

← Act

← Assert

- Drei Phasen
  - **Arrange**: Initialisieren der Test-“Welt“
  - **Act**: Ausführen der zu testenden Aktion
  - **Assert**: Prüfen der Test-Zusicherungen
- Die AAA-Normalform funktioniert für alle Unit Tests

# Einzelergebnis eines Unit Tests

- **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

# Eigenschaften guter Unit Tests

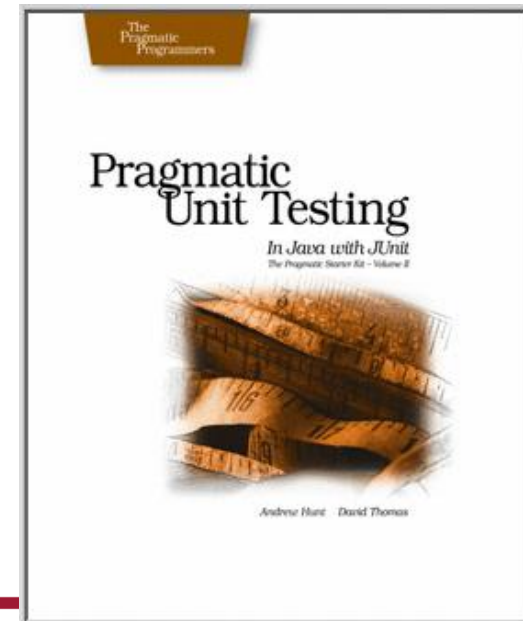
- Good Unit Tests = GUTs
- Wichtigste Eigenschaft:
  - Ein guter Unit Test ist tatsächlich vorhanden
- Gute Unit Tests werden zeitnah zum Produktivcode geschrieben
  - Test First → (Unit) Test zuerst schreiben
- Gute Unit Tests werden vom Entwickler als wertvoll und hilfreich angesehen
  - Tests schreiben darf/sollte keine lästige Pflicht sein

<https://www.youtube.com/watch?v=KtHQGs3zFAM>



# Fünf Regeln für gute Unit Tests

- ATRIP-Regeln
  - **Automatic** - Eigenständig
  - **Thorough** - Gründlich (genug)
  - **Repeatable** - Wiederholbar
  - **Independent** - Unabhängig voneinander
  - **Professional** - Mit Sorgfalt hergestellt
- Beschrieben in „Pragmatic Unit Testing“
  - Weitere Regeln dort: „Right BICEP“



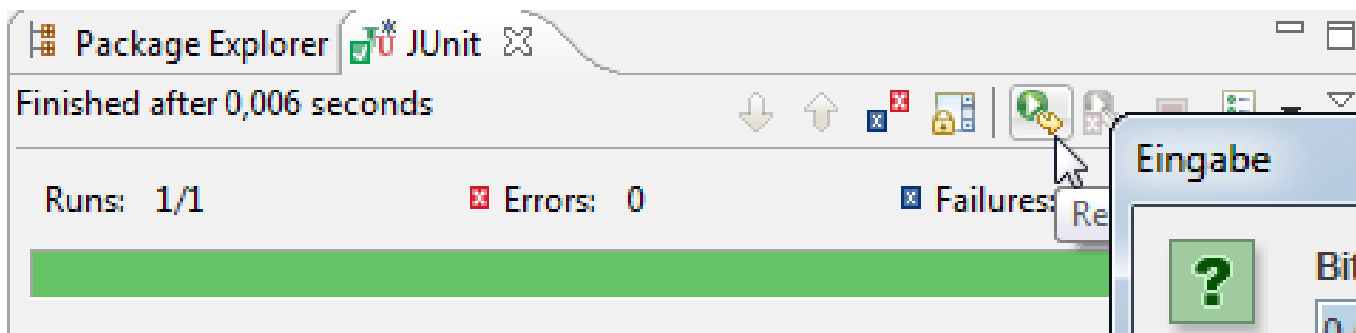
# Automatic



- Die Tests müssen eigenständig ablaufen
  - Keine manuellen Eingriffe notwendig
    - Kein Dialog-Wegklicken, keine manuellen 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 Testdurchführung ohne zusätzliches Wissen möglich → Automatisierbar

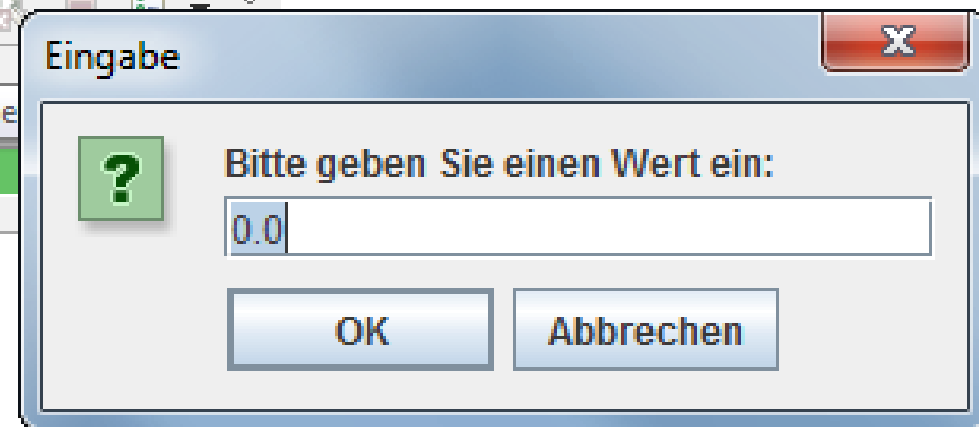
# Anti-Beispiel für Automatic

```
@Test
public void calculatesSquareRoot() {
    final double value = askForValue();
    final double squareRoot = SquareRoot.of(value);
    System.out.println("The square root of "
        + value
        + " is "
        + squareRoot);
}
```



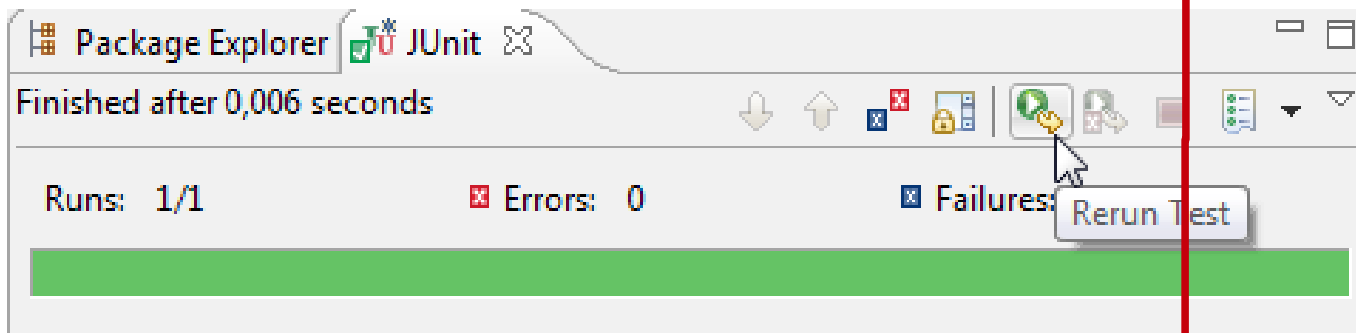
Console

```
<terminated> SquareRootTest [JUnit] D:\Program Files (x
The square root of 625.0 is 25.0
```



# Beispiel für Automatic

```
@Test
public void calculatesSquareRootOf625() {
    final double value = 625.0D;
    final double squareRoot = SquareRoot.of(value);
    assertThat(squareRoot, closeTo(25.0D, 1E-6));
}
```



Fließkommazahlenvergleiche sind ungenau!

**Immer eine Genauigkeit angeben!**

Idealerweise in E-Notation: 1E-x

x = Anzahl übereinstimmender Nachkommastellen

# 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, sondern bilden „Klumpen“
  - Zusätzliche Tests im „Umfeld“ eines Fehlers

# Folgen von Thorough

- Alle initialen Unit Tests können einer Anforderung zugeordnet werden
  - Traceability: Link zum Issue
- Für jedes Bug-Issue existiert mindestens ein neuer(?) Test

```
/**
 * #Bugfix http://issuetracker:8080/browse/FNE-214
 */
@Test
public void reportsSmallAmountsAccurately() {
    assertThat(
        new KadabraActivity(inBecquerel(1.37E-03)))
        .isCloseTo(1.37E-03, offset(1E-6));
}
```

**JIRA** Dashboards ▾ Projects ▾ Issues ▾

**FAME** **Fame Neutron / FNE-214**  
Aktivitätswerte kleiner 1 werden

**Details**

Type: ☒ Bug

Priority: ↑ Major

**Activity**

All **Comments** Work Log History Activity C

✓ **Daniel Lindner** added a comment - 22.07.2016 14:45  
Mit Test behoben. Die doubles wurden intern als long g... interpretiert wurden.

✓ **Schneide Bot** added a comment - 22.07.2016 14:48

SUCCESS: Integrated in Jenkins build fame-neutron #30... Aktivitäten können Werte kleiner 1 enthalten. Fixes FNE-214...  
[/c9119e32a9b3b0621ed5e0603eeede2886898a44j](#))

\* (add) Fame Neutron Module Kadabra/test/unit/com/schneide/fame/  
[/KadabraActivityTest.java](#)

\* (edit) Fame Neutron Module Kadabra/src/common/com/schneide/fame/  
[/KadabraActivity.java](#)

\* (edit) Fame Neutron Framework/src/com/schneide/fame/

# 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 oft schlechter als keiner
  - Tests müssen zuverlässig sein, sonst keine Absicherung

# Anti-Beispiel für Repeatable

- Beispiel aus einer medizinischen Anwendung
- Referenzmessungen sollen maximal 14 Tage gültig bleiben

```
@Test
public void isValidFor14Days() {
    DateTime now = new DateTime();
    ReferenceMeasurement target = new ReferenceMeasurement(now);
    DateTime inTwoWeeks = now.plusHours(14 * 24);
    assertThat(target.validAt(inTwoWeeks), is(true));
}
```

- Zwei Probleme:
  - Test ist abhängig von aktueller Systemzeit
  - Unnötiger Wechsel der Präzision (Stunden statt Tage)



# Beispiel für Repeatable

- Info vom Kunden: Wir verwenden keine Sommerzeit!
- Wir müssen also die Zeitzone fixieren!

```
@Test
public void ignoresSummertimeChange() {
    DateTimeZone noSummertime = DateTimeZone.forTimeZone(
        TimeZone.getTimeZone("UTC+01:00"));
    DateTime reference = new DateTimeFor().string("25.03.2017")
        .withZone(noSummertime);
    ReferenceMeasurement target = new ReferenceMeasurement(reference);
    DateTime inTwoWeeks = reference.plusHours(14 * 24);
    assertThat(target.validAt(inTwoWeeks), is(true));
}
```

# Beispiel für Repeatable

- Jetzt noch mit Tagen statt Stunden arbeiten
- Das Wissen um die Zeitzone muss unbedingt auch in den Produktivcode!

```
@Test
public void ignoresSummertimeChange() {
    DateTimeZone noSummertime = DateTimeZone.forTimeZone(
        TimeZone.getTimeZone("UTC+01:00"));
    DateTime reference = new DateTimeFor().string("25.03.2017")
        .withZone(noSummertime);
    ReferenceMeasurement target = new ReferenceMeasurement(reference);
    DateTime inTwoWeeks = reference.plusDays(14);
    assertThat(target.validAt(inTwoWeeks), is(true));
}
```

<https://schneide.wordpress.com/2009/12/28/a-blind-spot-of-continuous-integration>

# 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
- Ideal: Jeder Test testet genau einen Aspekt der Komponente
  - Im Fehlerfall bricht ein Test, nicht hunderte
  - Bei Testbruch sollte die Ursache möglichst direkt ableitbar sein

# Anti-Beispiel für Independent

```
@Test
public void canWriteToStorage() {
    RamsesStation station = new PersistedFullRamsesStation(
        new TestStationIdentifier("test01"),
        "First Teststation", null);
    StationStorage storage = new StationStorage(new File("C:/temp"));
    assertThat(storage.contains(station), is(false));
    storage.store(station); // writes to disk
    assertThat(storage.contains(station), is(true));
}
```

```
@Test
public void canRemoveFromStorage() {
    RamsesStation station = new PersistedFullRamsesStation(
        new TestStationIdentifier("test01"),
        "First Teststation", null);
    StationStorage storage = new StationStorage(new File("C:/temp"));
    assertThat(storage.contains(station), is(true));
    storage.remove(station); // changes disk state
    assertThat(storage.contains(station), is(false));
}
```

# Beispiel für Independent

```
@Rule  
public TemporaryFolder temporary = new TemporaryFolder(); // empty at first
```

```
@Test  
public void canWriteToStorage() {  
    RamsesStation station = testStation();  
    StationStorage storage = new StationStorage(this.temporary.getRoot());  
    storage.store(station); // writes temporarily to disk  
    assertThat(storage.contains(station), is(true));  
    // temporary files are now deleted automatically  
}
```

```
@Test  
public void canRemoveFromStorage() {  
    RamsesStation station = testStation();  
    StationStorage storage = new StationStorage(this.temporary.getRoot());  
    storage.store(station); // writes temporarily to disk  
    storage.remove(station); // changes disk state  
    assertThat(storage.contains(station), is(false));  
    // temporary files are now deleted automatically  
}
```

# Professional



- Testcode ist auch produktionsrelevanter Code
  - Allerdings wird Testcode selbst nicht automatisch getestet
  - Fehler in Tests sind folgenreich und 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

# Anti-Beispiel für Professional

```
public class SPEChannelValuesTest {
    private static final TestResult ZERO_COUNT = new TestResult(
        new PulseCount(0),
        new byte[] {0x0, 0x0, 0x0, 0x0});
    private static final TestResult VERY_SMALL_COUNT = new TestResult(
        new PulseCount(34),
        new byte[] {0x0, 0x0, 0x22, 0x0});
    private static final TestResult MEDIUM_COUNT_BORDER = new TestResult(
        new PulseCount(65536),
        new byte[] {0x1, 0x0, 0x0, 0x0});

    @Test
    public void serializeSingleChannelValues() throws Exception {
        SPEChannelValuesSerializer scv = new SPEChannelValuesSerializer();
        Assertion.assertArrayEquals(ZERO_COUNT.getBytes(),
            scv.serializeCounts(ZERO_COUNT.getCounts()));
        Assertion.assertArrayEquals(VERY_SMALL_COUNT.getBytes(),
            scv.serializeCounts(VERY_SMALL_COUNT.getCounts()));
        Assertion.assertArrayEquals(MEDIUM_COUNT_BORDER.getBytes(),
            scv.serializeCounts(MEDIUM_COUNT_BORDER.getCounts()));
    }
}
```

# Anti-Beispiel für Professional

```
@Test
public void serializeMultipleChannelValues() {
    SPEChannelValuesSerializer scv = new SPEChannelValuesSerializer();
    PulseCount[] counts = new PulseCount[] {
        ZERO_COUNT.getCounts()[0],
        VERY_SMALL_COUNT.getCounts()[0],
        MEDIUM_COUNT_BORDER.getCounts()[0]};
    byte[] resultBytes = new byte[3 * ZERO_COUNT.getBytes().length];
    ByteUtil.copyIntoArray(
        ZERO_COUNT.getBytes(),
        resultBytes, 0);
    ByteUtil.copyIntoArray(
        VERY_SMALL_COUNT.getBytes(),
        resultBytes,
        ZERO_COUNT.getBytes().length);
    ByteUtil.copyIntoArray(
        MEDIUM_COUNT_BORDER.getBytes(),
        resultBytes,
        2 * ZERO_COUNT.getBytes().length);
    Assertion.assertArrayEquals(resultBytes, scv.serializeCounts(counts));
}
```



# Beispiel für Professional

- Lesbare Tests sind sehr wichtig
  - Gerade Testcode wird einmal geschrieben, aber 100 mal gelesen
- Beispiel aus einer Bankkonto-Verwaltungssoftware

```
@Test
public void canWithdrawOnCredit() {
    performWithdrawalTestWith(
        true,
        new Euro(30),
        new Euro(30),
        new Euro(-30));
}
```

# Beispiel für Professional

- Gleicher Test mit Fokus auf Lesbarkeit
- Hier: Erklärende Variablen eingeführt

```
@Test
public void canWithdrawOnCredit() {
    boolean aCustomerThatCanOverdraw = true;
    Euro heWithdraws30Euro = new Euro(30);
    Euro receivesTheFullAmount = new Euro(30);
    Euro andIsNow30EuroInTheRed = new Euro(-30);

    performWithdrawalTestWith(
        aCustomerThatCanOverdraw,
        heWithdraws30Euro,
        receivesTheFullAmount,
        andIsNow30EuroInTheRed);
}
```

<https://schneide.wordpress.com/2013/12/30/from-ugly-to-pretty-three-steps-is-all-it-takes>

# Wie teste ich meine Tests?

- Testabdeckung (Code Coverage) bestimmen
  - Code Coverage Analyse in der IDE nutzen
- 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 eventuell beschädigt
  - Die Tests sollten solche Veränderungen registrieren
  - Mutation Testing

# Testabdeckung

- Code Coverage (Testabdeckung) hat verschiedene Bezugspunkte
- Am gebräuchlichsten und nützlichsten sind
  - Line Coverage
  - Branch Coverage
- Wichtig: Immer Bezugspunkt mit angeben, die Werte können stark unterschiedlich sein

# Line Coverage

- Jede Zeile Code wird durch die Tests entweder durchlaufen oder nicht
  - Durchlaufen heißt noch nicht abgesichert!

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

Line Coverage = 67%

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

# Branch Coverage

- Jede Entscheidung im Code (if-statement) kann auf verschiedenen „Abzweigungen“ durchlaufen bzw. verlassen werden

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

Branch Coverage = 50%

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

# Hohe Testabdeckungen

- Hohe Testabdeckung ist hilfreich
  - Größer 80% (LC/BC) ist sehr aufwendig
- Wirtschaftliches Optimum ist wichtig!
  - Aufwand versus Kosten
- Eine Testabdeckung von 100% (LC) bedeutet noch nicht, dass alles getestet wurde
  - In den Tests wurden nur alle Zeilen durchlaufen
- Nur die „roten Zeilen“ sind aussagekräftig
  - „grüne Zeilen“ = könnte getestet sein

# Mutation Testing

- Lässt alle Tests mehrfach durchlaufen
- Bei jedem Durchlauf wird eine Änderung (Mutation) in den Produktivcode eingebracht
  - If-statement negiert
  - Berechnung geändert
  - Methodenaufruf entfernt
- Tests müssen die Mutation entdecken
  - Tests gehen rot = Mutation „gekillt“
  - Tests bleiben grün = Mutation „hat überlebt“

<http://pitest.org/quickstart/mutators>



<http://pitest.org>



# Mutation wird gekillt

- Der Produktivcode wird an einer Stelle geändert, die durch einen Test abgesichert ist
  - Hier: Negate Conditionals Mutator

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

Mutation killed

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

# Mutation überlebt

- Der Produktivcode wird an einer Stelle geändert, die nicht durch Tests abgesichert ist
  - Hier: Return Values Mutator

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

← Mutation survives

```
public class Some {  
    public int thing(boolean mode) {  
        if (mode) {  
            return 42;  
        }  
        return 0;  
    }  
}
```

# Mutation Testing

- Sinnvolle und unaufwendige Zusatzprüfung
- Leider üblicherweise mit hoher Anzahl an False-Positive-Meldungen
  - Nicht den dauerhaft überlebenden Mutationen hinterherjagen
  - Nur neue Meldungen untersuchen
- Mutation Testing ist eine langfristige Investition
  - Einmal-Einsatz scheitert an den Falschmeldungen
  - Gibt nach und nach Best Practices für Tests und Produktivcode

# Tests auszeichnen

- Projekte haben zahlreiche Tests
- Welche sind wichtig? Welche sind essentiell?

```
/**
 * #Requirement https://jira.dnb.de/browse/URN-231
 */
@Test
public void namespaceExists() {
    final String urnServerAddress = "http://localhost:8080";
    startURNServerAt(urnServerAddress);

    final WebConversation conversation = new WebConversation();
    final WebRequest request = new HeadMethodWebRequest(
        urnServerAddress + "/v1/namespaces/name/urn%3Anbn%3Ade%3A1111");
    final WebResponse response = conversation.getResponse(request);

    assertThat(response.getResponseCode(), equalTo(200));
    assertThat(response.getHeaderFieldNames(), emptyArray());
}
```

# Tests auszeichnen

- „Herkunft“ bzw. Motivation des Tests dokumentieren
  - Details gehören ins zugehörige Issue

```
/**
 * #Bugfix http://issuetracker.intranet:8080/browse/RAM-2681
 */
@Test
public void doesntIncludeLocalhostIfSecondaryIsEmpty() {
    AdvancedStationConfiguration target = new AdvancedStationConfiguration(
        configurationWith(
            "main.center.address=localhost:55554",
            "secondary.center.addresses=",
            "default.center.port=55555",
            "secondary.addresses.first=true"));
    assertThat(target.getSecondaryCenterAddressesCable(), emptyIterable());
}
```

# Tests auszeichnen

- Die Geschichte und Verdienste eines Tests als „Orden“ anheften
- Unterscheidet wertvolle Tests sofort von (noch) nicht so wertvollen
- Gibt dekorierten, gebrochenen Tests Glaubwürdigkeit



# Tests auszeichnen

- Auszeichnung der Motivation:
  - **#Requirement** – Sichert eine Anforderung ab
  - **#Bugfix** – Verhindert die Anwesenheit eines Fehlers
- Auszeichnung des bisherigen Nutzens:
  - **#Regression** – Hat einen neuen Fehler verhindert
  - **#Lifesaver** – Hat vor einer Katastrophe gewarnt

<https://schneide.wordpress.com/2013/08/25/award-your-tests>

# Tests auszeichnen

- Auszeichnungen gerne mehrfach vergeben
  - Für jeden Nutzen „belohnen“

```
/**
 * #Bugfix http://issuetracker.intranet:8080/browse/RAM-2622
 * #Regression #Regression #Regression
 */
@Test
public void noNegativeDurations() {
    FesaOperationInterval target = new FesaOperationInterval(
                                                someFesa(),
                                                this.clock.before(Hours.ONE));
    target.setEnd(this.clock.before(Hours.THREE));
    assertThat(target.getDuration(), is(Duration.ZERO));
}
```



# Tests auszeichnen

- Auszeichnung der bisherigen Kosten:
  - **#Adjusted** – Am Test waren Anpassungen notwendig
  - **#Erratic** – Brach ohne erkennbaren Zusammenhang
  - **#Fragile** – Schlug wegen Nebensächlichkeiten fehl

```
/**
 * #Bugfix http://issuetracker.intranet:8080/browse/RAM-2620
 * #Erratic #Erratic #Fragile #Adjusted
 */
@Test
public void updatesAssociatedStationsOnFesaRemoval() {
    FullRamsesStation associated = fullStation("associated", true);
    AbstractRamsesStationManager target = stationManagerWith(associated);
    target.fesaRemoved(someFesa());
    verify(target, times(1)).updateStation(any(RamsesStation.class));
}
```

# Tests auszeichnen

- Auszeichnungen verdeutlichen für jeden Test
  - (Bisherige) Glaubwürdigkeit
  - **Investitionsnutzen**
- Fundierte Entscheidungen möglich(er):
  - „Warum muss ich diesen Test dauernd anpassen?“
  - **„Lohnt sich dieser Test überhaupt?“**
  - „Der Fehler liegt wohl in meiner Änderung und nicht im Test“
- Tests sind nicht alle gleichwertig und gleich gut

# Tests lesbarer gestalten

- Tests müssen möglichst „zugänglich“ sein:
  - Leicht lesbar
  - Direkt verständlich
  - Einfach strukturiert
- Meine Meinung:
  - **Self-Contained Tests:** Lieber Duplikation als durch gemeinsam genutzte Strukturen verflochten
  - **Pragmatic, not Dogmatic:** Lieber ein paar (lokale) seltsame Konstrukte als eine beliebige Idealvorstellung erfüllt

# Tests lesbarer gestalten

- Zusätzliche Test-Bibliotheken nutzen
  - Direkt einbinden, auch wenn man sie „adoptiert“
- Beispiel: AssertJ
  - Bietet Fluent Interfaces für Assertions

```
@Test
public void lordOfTheRingCharactersAreReady() {
    assertThat(frodo.getName()).isEqualTo("Frodo");
    assertThat(frodo).isNotEqualTo(sauron);

    List<TolkienCharacter> fellowship = assembleFellowshipOfTheRing();
    assertThat(fellowship).hasSize(9)
        .contains(frodo, sam)
        .doesNotContain(sauron);

    assertThat(fellowship).extracting(TolkienCharacter::getName)
        .doesNotContain("Sauron", "Elrond");
}
```

# Tests lesbarer gestalten

- Neue Version von JUnit steht vor Fertigstellung
- JUnit 5
  - Release geplant gegen Ende 2017
  - Macht wieder vieles/alles anders

## 1.1. What is JUnit 5?

Unlike previous versions of JUnit, JUnit 5 is composed of several different modules from three different sub-projects.

*JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage*

- Einige sinnvolle Neuerungen
  - Trennung von Anzeigename und Implementierungsname

```
@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("J°□°J")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

    @Test
    @DisplayName("😄")
    void testWithDisplayNameContainingEmoji() {
    }

}
```

# Isolation von Komponenten

Ein Software-  
system

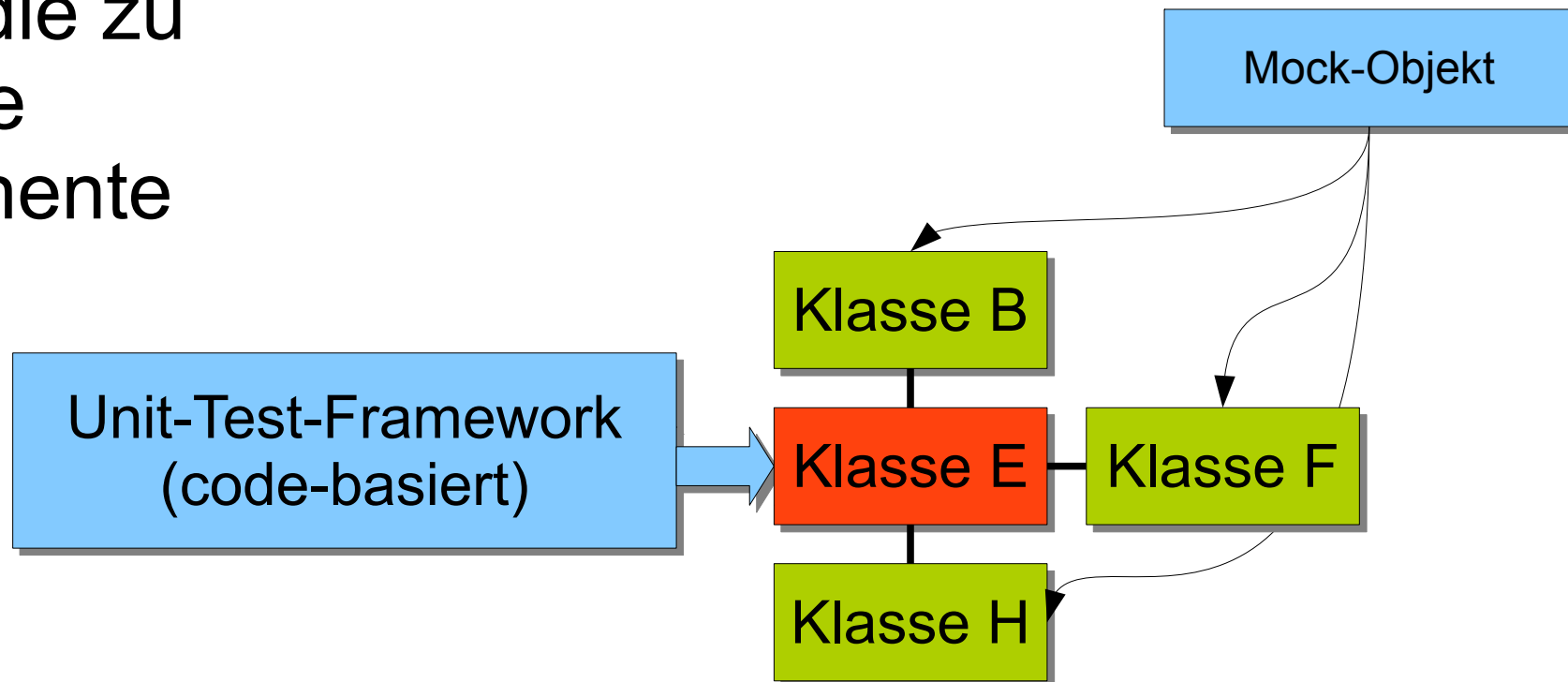


# Eine Funktionalität muss getestet werden



# Isolation von Komponenten

Ein Unit-Test  
isoliert die zu  
testende  
Komponente



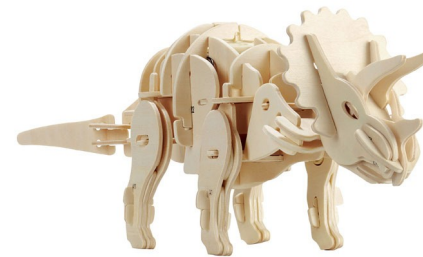


# Mock-Objekte

- 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 beleuchtungsrelevanten Eigenschaften mit dem echten Star gemeinsam

# Isolation durch Mocks

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



Klasse E



# 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 dokumentenbasierter Datenbank
- Wir wollen die Datenbank nicht mittesten
  - Die Datenbank muss gemockt werden
- 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;
```

```
}
```

Mock erstellen

Mock einlernen

Lernphase vorbei.  
Umschalten auf  
Abspielen

Mock überprüfen

# Analyse des Tests mit Mock

- 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
- Ein Mock-basierter Test hat fünf Phasen:

# Normalform von Tests 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;  
    }  
}
```

Capture

Arrange

Act

Assert

Verify

Replay



# Tests mit Mock-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

# Zusammenfassung Mocks

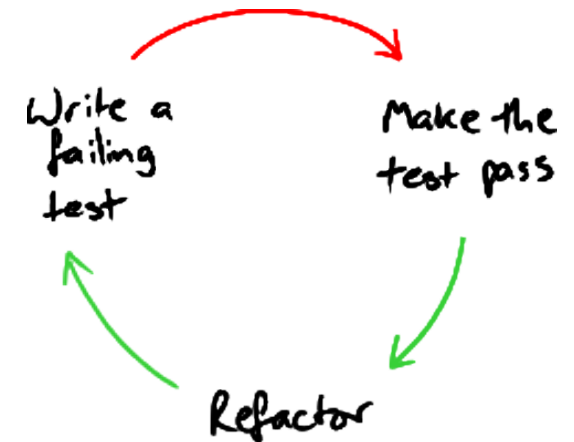
- Mächtige, werkzeuggestützte Technik, um Objekte in Tests zu isolieren
  - Und Aufruf-Protokolle zu testen
- Mock-Objekte werden direkt vor Benutzung im Test definiert
  - Sind nicht Teil des Produktivcodes
- Aufpassen: Man testet relativ leicht nur noch, dass das Mock-Framework funktioniert (Zirkelschluss-Test)

# Test First

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

# Test First Entwicklungszyklus

- Vorgehensweise bei Test First:
  - Funktionalität planen (im Kopf)
  - Test schreiben
  - Funktionalität programmieren
  - Refactoring



# 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

# 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	ↀ	ↁ
Wert	1	5	10	50	100	500	1.000	5.000	10.000

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

$$4 = IV$$

$$9 = IX$$



# Praxisbeispiel Schritt 1a

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

# Praxisbeispiel Schritt 1b (+1c)

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

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

# Praxisbeispiel Schritt 2a

```
public class RomanNumeralTest {
```

```
@Test
```

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

```
@Test
```

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

```
}
```

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

# Praxisbeispiel Schritt 2b (+2c)

```
public class RomanNumeralTest {  
    @Test  
    public void convertsOne() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void convertsTwo() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
}
```

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

# Praxisbeispiel Schritt 3a

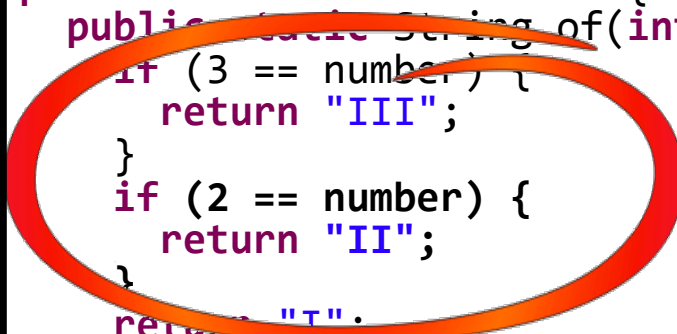
```
public class RomanNumeralTest {  
    @Test  
    public void convertsOne() {  
        assertThat(RomanNumeral.of(1), is("I"));  
    }  
  
    @Test  
    public void convertsTwo() {  
        assertThat(RomanNumeral.of(2), is("II"));  
    }  
  
    @Test  
    public void convertsThree() {  
        assertThat(RomanNumeral.of(3), is("III"));  
    }  
}
```

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

# Praxisbeispiel Schritt 3b

```
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"));  
    }  
}
```

```
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

```
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"));  
    }  
}
```

```
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();  
    }  
}
```

# Praxisbeispiel Schritt 4a

```
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"));  
    }  
}
```

```
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();  
    }  
}
```



# Abschluss Praxisbeispiel

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

<https://remonsinnema.com/2011/12/05/practicing-tdd-using-the-roman-numerals-kata>

- Viele Screencasts im Netz
- Entscheidungshilfe für Implementierung:

<https://8thlight.com/blog/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html>

- Achtung! TF/TDD ist anfangs sehr ungewohnt und schwierig
  - 3 Monate Geduld und Übung sind sinnvoll

# Härtefälle für (Unit) Tests

- Abtesten der visuellen Eigenschaften von Widgets oder Webseiten
- Asynchrone Ereignisfolgen
  - Threading-Verhalten nicht deterministisch
  - Aspekt der Asynchronität auslagern
- Komplexe Berechnungen
  - Test-Triangulation erforderlich
- Zufalls- und datumsabhängige Funktionalität
  - Quellen von Abweichungen während des Testens fixieren

# Aussagekräftige Tests

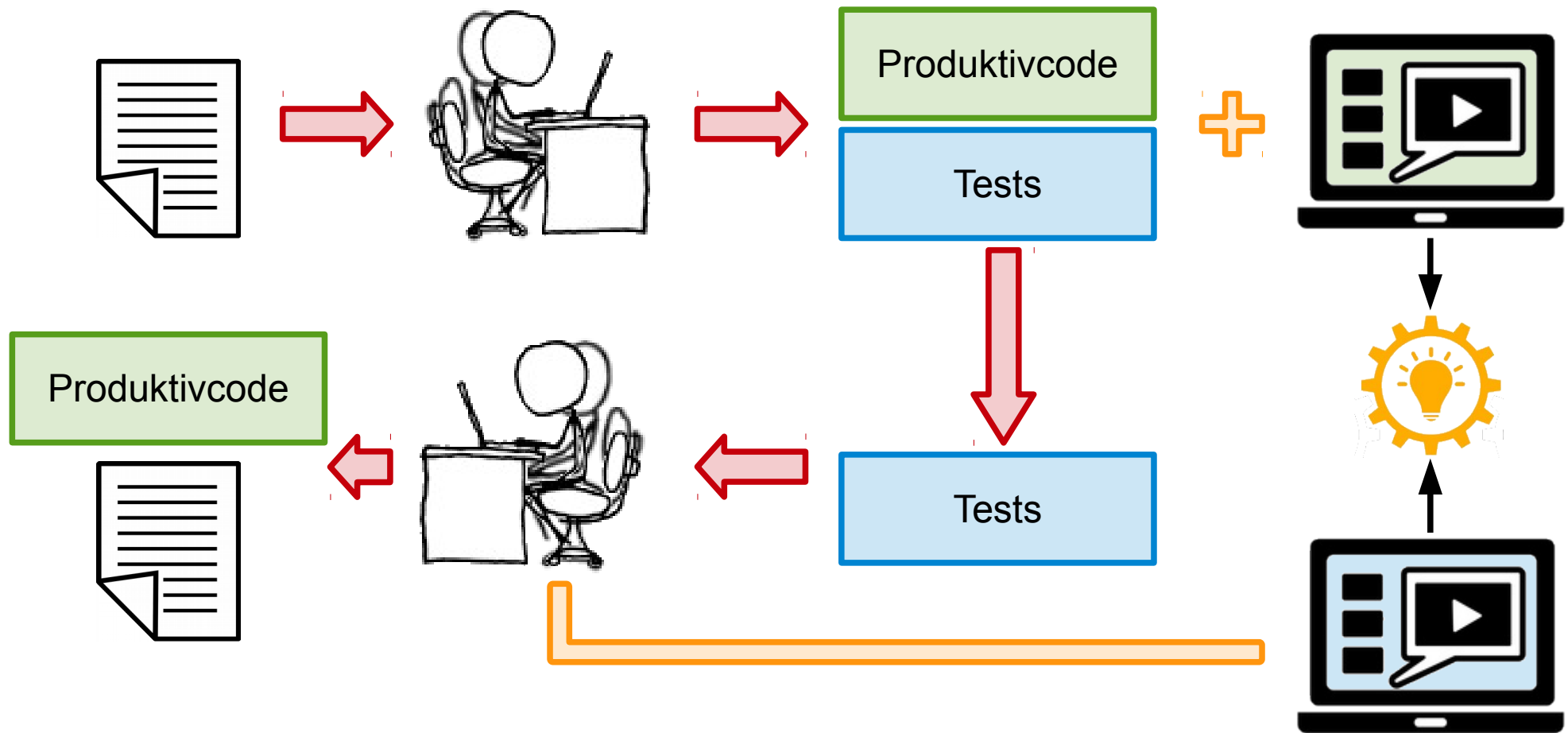
- Interessante Fragestellung:
  - Was würde passieren, wenn der Produktivcode verloren geht und nur die Tests übrig bleiben?
  - Könnte man die Anwendung aus den Tests rekonstruieren?
  - Wie gut transportieren die Tests die Intention der Anwendung?
- Interessantes Experiment
  - Bei uns im Rahmen einer internen Schulungsreihe zu automatisierten Tests durchgeführt

<https://schneide.wordpress.com/2012/12/10/an-experiment-about-communication-through-tests>

# Communication through Tests

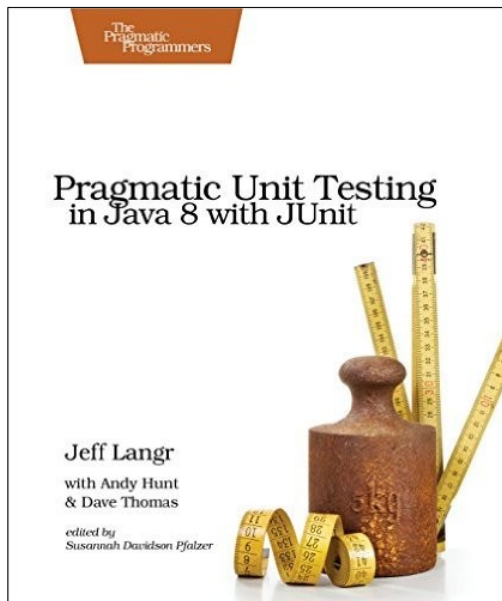
- Experiment „Communication through Tests“:
  - Eine nicht-triviale, unbekannte Aufgabenstellung
    - Conway's Game of Life ist zu klein
    - Minesweeper ist zu bekannt
  - Ein Team (2 Personen), das die Aufgabe mit hoher Testabdeckung (mehr als 90 % Line Coverage) implementiert
  - Ein anderes Team (2 Personen), das nur die Tests erhält und den Produktivcode und damit die Aufgabe rekonstruieren soll
- Idealerweise zeichnen beide Teams ihre Arbeit als Screenshot auf

# Communication through Tests

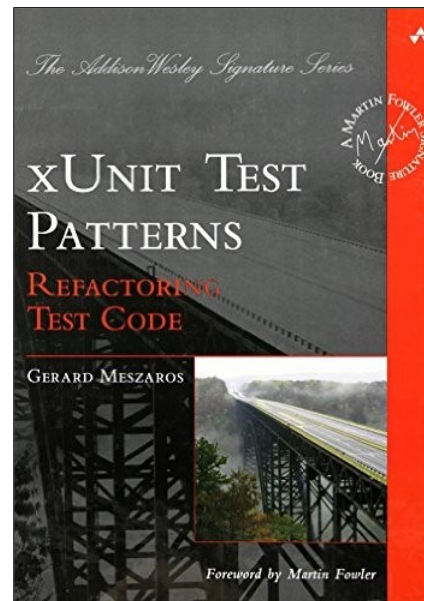


<https://schneide.wordpress.com/2013/03/04/communication-through-tests-a-larger-experiment>

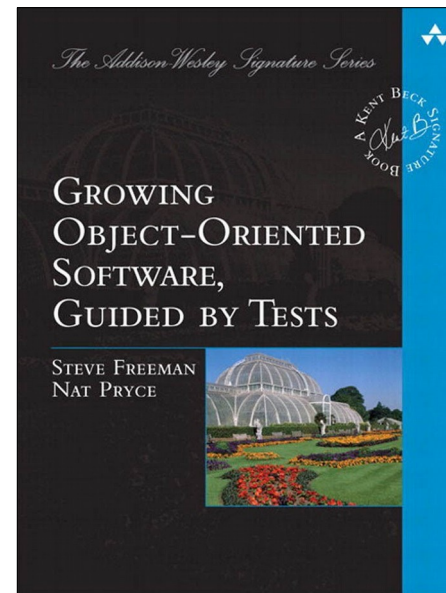
# Automated Testing: Weiterführende Literatur



2015



2007



2009



2005

<http://www.frankwestphal.de/MeinBuchzumDownload.html>

# Weiterführende Web-Literatur

- Automated Testing Patterns and Smells (60 min)

<https://www.youtube.com/watch?v=Pq6LHFM4JvE>

- Testing is Overrated (25 min)

<https://www.infoq.com/presentations/franc1-testing-overrated>

- Integration Tests Are a Scam (90 min)

<https://www.infoq.com/presentations/integration-tests-scam>

- Is TDD Dead? (insgesamt ca. 200 min)

<https://martinfowler.com/articles/is-tdd-dead>

<https://www.infoq.com/presentations/unit-testing-tips-tricks>

# Schneide Blog-Artikel zum Thema

- The difference between Test First and TDD

<https://schneide.wordpress.com/2013/04/29/the-difference-between-test-first-and-test-driven-development>

- Refactoring turns unit test into integration tests

<https://schneide.wordpress.com/2013/04/22/does-refactoring-turn-unit-test-of-tdd-to-integration-tests>

- A small test saves the day

<https://schneide.wordpress.com/2012/11/19/a-small-test-saves-the-day>

- Game of Life: TDD style in Java

<https://schneide.wordpress.com/2012/05/31/game-of-life-tdd-style-in-java>



# Bildnachweise

- Old rusty rugged anvil: Fotolia Datei: #75028506 | Urheber: cronislaw
- Weiche rote Decke: Rhystick.ch
- Military ribbons: Pinterest, <https://s-media-cache-ak0.pinimg.com/736x/d7/3a/97/d73a97488564fda46d868bc7f48d5fa3.jpg>
- Ernährungspyramide: Von Targan - Eigenes Werk, Gemeinfrei, <https://commons.wikimedia.org/w/index.php?curid=511990>
- Tutorial simple icon: Fotolia Datei: #110762446 | Urheber: redlinevector
- Gears Question, Idea & Answer Grey/Yellow Outline: Fotolia Datei: #121137692 | Urheber: Jan Engel