Hochschule
für Wirtschaft und Gesellschaft
Ludwigshafen

Aktuelle Themen der IT

Vorbereitung

Installation Entwicklungsumgebung

- Java IDE (Eclipse)
- JUnit Support
- Checkout von GitHub
- Test der Entwicklungsumgebung

Agenda

- Vorstellung & Kursinhalte
- Abfrage Vorwissen: Radar Chart
- Check/Setup Umgebung + GitHub Crash Kurs
- AntiPattern
- Scrum, Stage Gate, Quality Attributes
- Übung 1: Kontrollflussgraphen & Strukturtestverfahren
- Unit Testing
- Übung 2: Unit Testing

Kontakt & Termine

Britta Stengl	britta.stengl@sap.com
Thomas Hammer	tm.hammer@web.de

20.03.2019	10:00 - 15:45 Uhr
27.03.2019	10:00 - 15:45 Uhr
03.04.2019	10:00 - 15:45 Uhr
10.05.2019	10:00 - 15:45 Uhr
24.05.2019	10:00 - 13:15 Uhr

Fehlerfreiheit eines Produktes zeichnet dessen Qualität aus.

Qualität von Software

Die Gesamtheit der Merkmale, die eine Ware oder Dienstleistung zur Erfüllung vorgegebener Forderungen geeignet macht.

Deutsche Gesellschaft für Qualität

Motivation

Aus wirtschaftlicher Perspektive:

- Reduktion von Kosten generiert durch Fehlerbehebungsaufwand
- Maximierung des Gewinns durch Softwarevertrieb

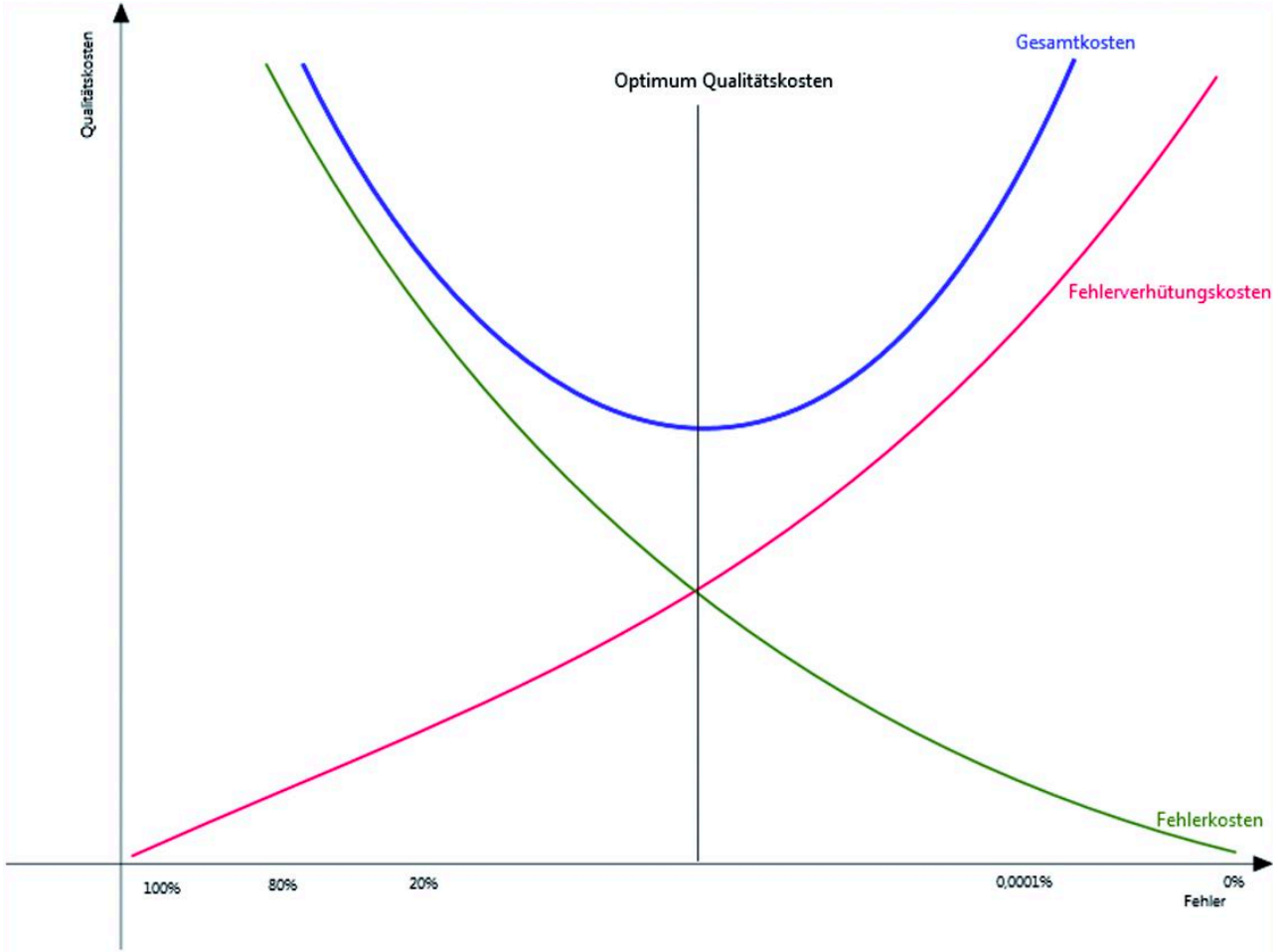
Aus Entwickler-Perspektive:

- Erhöhung der Wartbarkeit
- Flexibilität in der Weiterentwicklung bestehender Programme, Austausch vorhandener Programmschichten
- Einfache Nachvollziehbarkeit und Reduktion des Einarbeitungsaufwandes

Aus Nutzerperspektive:

- Erhöhung der Zufriedenheit
- Steigerung der Zuverlässigkeit
- Störungsreduktion im Betriebsablauf
- Erhöhung der Zuverlässigkeit und des Vertrauens

Zwischen Nutzen und Kosten



Setup Umgebung

1.Installation Eclipse IDE
   Eclipse (https://www.eclipse.org/downloads/packages/installer)
   Java SDK (https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html)

2.Installation Git Client
   https://git-scm.com/downloads

3.Git User Setup
   https://github.com/join

4.Checkout Test-Projekt
   https://github.com/atdit-hslu-ibait18/DevEnvSetupTest

5.Test Entwicklungsumgebung via Eclipse
   https://github.com/atdit-hslu-ibait18/DevEnvSetupTest.git

GitHub

GitHub 

- Basiert auf dem Open Source Quellcode-Verwaltungssystem Git
- Git dient zur Versionsverwaltung von Quelltext und unterstützt den Softwareentwicklungs-Prozess
- GitHub ermöglicht die Kollaboration von Entwicklern an Projekten durch die darunter liegende Plattform
- Verteilte Software-Repositories ermöglichen die gemeinschaftliche Arbeit an Projekten
- Viele weitere nützliche Funktionen (Pages, Doku, Issue Tracking, Release-Mgmt, etc.)

Git und GitHub sind aktuell die meistgenutzten Quellcode-Verwaltungssysteme in der Softwareentwicklung

GitHub

GitHub

Basic Commands:

- $ git init // Initialisieren eines lokalen Git Repositories
- $ git add <file> // Dateien zum Index hinzufügen
- $ git status // Status des Working Tree
- $ git commit // Änderungen auf den Index anwenden
- $ git push // auf remote Repository pushen
- $ git pull // aktuellen Stand vom remote Repository holen
- $ git clone // Repository klonen

GitHub

Git und GitHub Crash Courses:

- https://try.github.io/ (interaktiver GitHub Kurs)
- https://guides.github.com/activities/hello-world/ (Tutorial)
- https://www.youtube.com/watch?v=EUvmCuPjHD4 (GUI + Command Line)
- https://www.youtube.com/watch?v=SWYqp7iY_Tc (ausführliche Erklärung)

GitHub

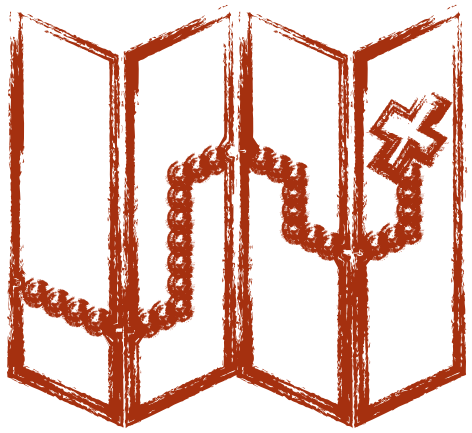
git clone git@github.com:atdit-hslu-ibait18/Material.git

AntiPattern

AntiPattern

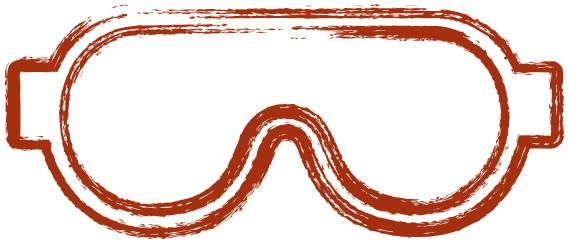
- Negativbeispiele zu wiederkehrenden Problemen in der Softwareentwicklung mit Lösungsanleitungen
- Gegensatz zu Pattern (Lösungen zu bekannten Problemen)
- Häufig lehrreicher und eingängiger als Pattern
- Zeigen einen detaillierten Plan zur Ursachenforschung und Problemlösung

AntiPattern



Software
Development
Antipattern

Software
Architecture
AntiPattern



Software
Project
Management
AntiPattern

AntiPattern

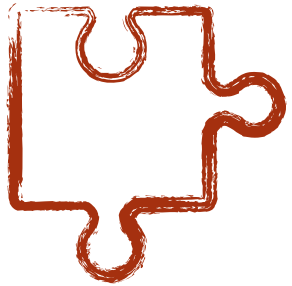
Software Architecture AntiPattern	Software Development AntiPattern	Software Project Management Antipattern
Architecture by Implication	Functional Decomposition	Death by Planning
Reinvent the Wheel	The Blob	Analysis Paralysis
Vendor Lock-In	Golden Hammer	Corncob

Testen von Software

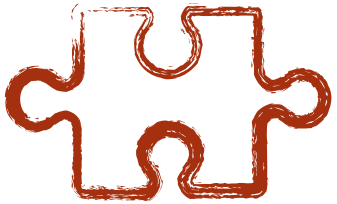
Software-Tests

Ziel:

- Inkonsistenzen zwischen Spezifikation und Implementierung aufdecken
- Abweichungen von Qualitätsmerkmalen vom vorgesehen Soll-Zustand erkennen



Spezifikation



Implementierung

Software-Tests

- Test-Ebene
  - Module, Komponenten, Systeme, Akzeptanz
- Test-Art
  - funktional, strukturell, Last/Leistung, Grenzwerte
- End-Kriterien
- Einbezug von Dokumentation
- Einbezug von Mitarbeiten/Kunden (Know-How, Akzeptanz)

Zweigüberdeckungsverfahren

Ziel:

- Struktureller Test
- Überprüfung auf Abdeckung aller Programmzweige

Mit Hilfe von Kontrollflussgraphen (KFG)

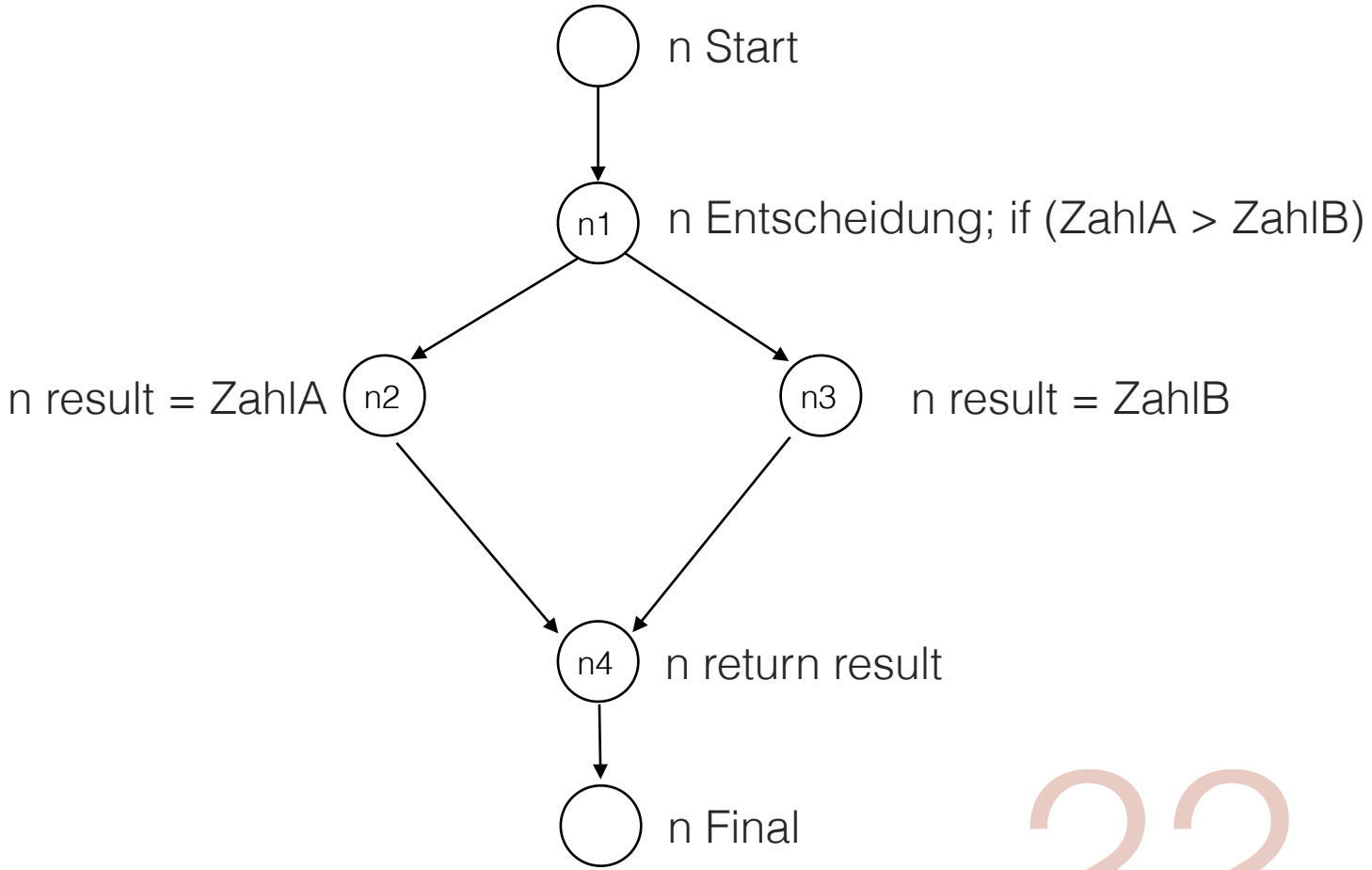
Kontrollflussgraph (KFG)

Aufbau KFG:

- Endliche Menge an Knoten
- Jeder KFG hat einen Start- und Endknoten
- Verbindungen der Knoten durch gerichtete Kanten
- Knoten : ausführbare Anweisung
- Gerichtete Kante : möglicher Kontrollfluss

Beispiel Zweigüberdeckungsverfahren

```
int bestimmeMax(int ZahlA, int ZahlB)
{
    int result = 0;
    if (ZahlA > ZahlB)
    {
        result = ZahlA;
    }else
    {
        result = ZahlB;
    }
    return result;
}
```



Aufgabe Zweigüberdeckungsverfahren

Programm zur Qualitätsauswertung

Spezifikation:

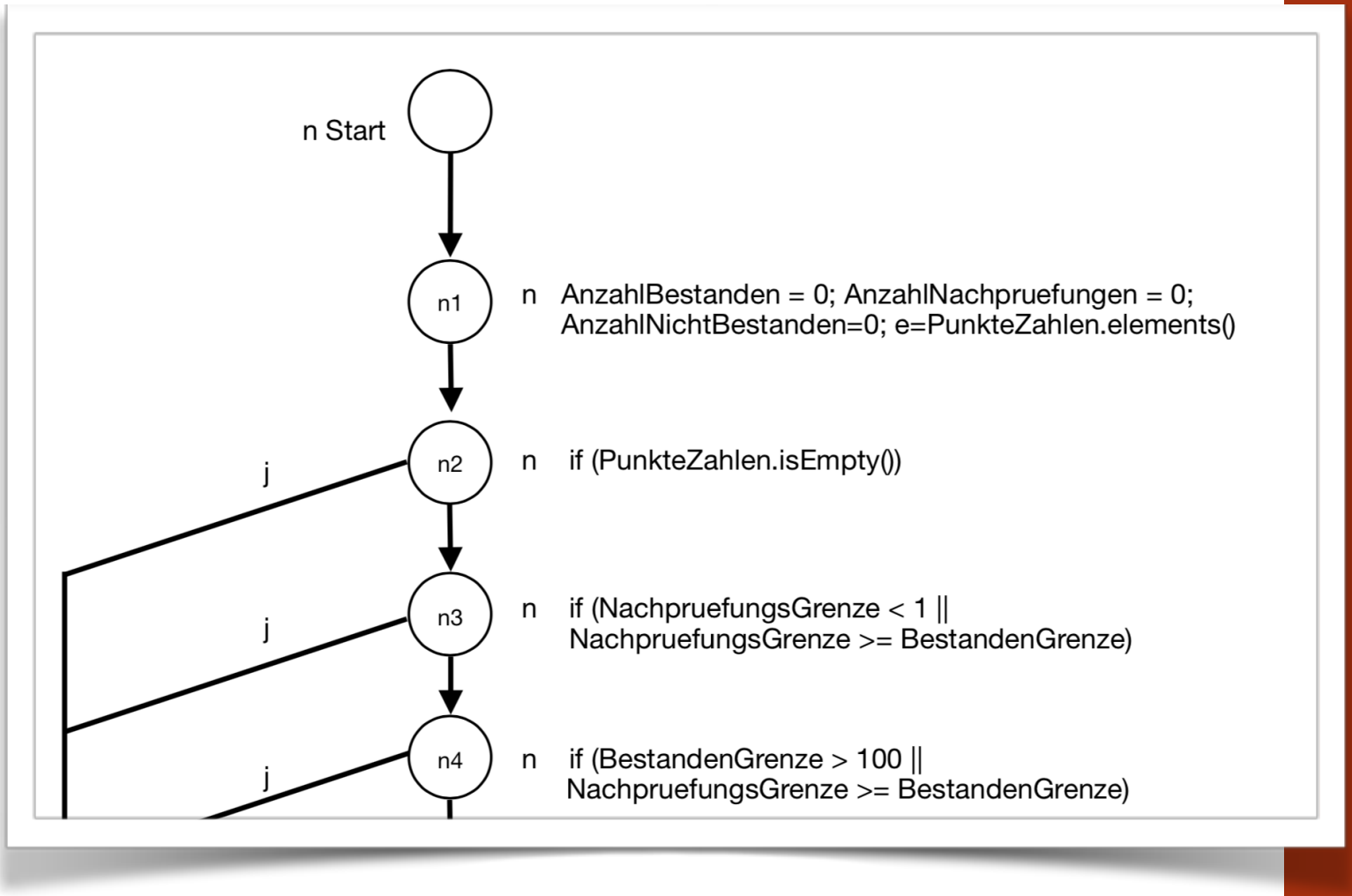
1. Jedes Exemplar eines Produktes durchläuft einen Qualitätstest. Das Ergebnis des Qualitätstests ist pro Exemplar eine ganze Zahl zwischen Null und Hundert.
2. Exemplare, die unterhalb einer unteren Grenzschwelle (<=Nachprüfungsgrenze) liegen, werden sofort als Ausschuss ausgesondert.
3. Exemplare, die oberhalb einer oberen Grenzschwelle (>=Bestandengrenze) liegen, werden an die Kunden ausgeliefert.
4. Exemplare, die zwischen der Nachprüfungsgrenze und der Bestandengrenze liegen, werden einer manuellen Nachbearbeitung mit anschließender Nachprüfung unterzogen.
5. Eine Klassifizierungsoperation soll eine Statistik erstellen, die angibt, wie sich die Messwerte auf die Kategorien „Ausschuss“, „Nachbearbeitung“ und „Bestanden“ verteilen.
6. Für die Nachprüfungsgrenze soll eine Voreinstellung von 50 Punkten, für die Bestandengrenze von 80 Punkten angezeigt werden.

```
public class Qualitaetsauswertung
{
    protected int AnzahlNichtBestanden;
    protected int AnzahlNachpruefungen;
    protected int AnzahlBestanden;
    protected Vector<Integer> PunkteZahlen = new Vector<Integer>();

    public void hinzufuegenPunktzahl (int neuePunktzahl) throws Exception
    {
        if ((neuePunktzahl < 0) || (neuePunktzahl > 100))
            throw new Exception("Punktzahl nicht zwischen 1 und 100");
            PunkteZahlen.addElement(new Integer(neuePunktzahl));
    }

    public void berechneKategorien (int NachpruefGrenze, int BestandenGrenze) throws Exception
    {
        Enumeration<Integer> e = PunkteZahlen.elements();
        AnzahlBestanden = 0;
        AnzahlNachpruefungen = 0;
        AnzahlNichtBestanden = 0;

        if (PunkteZahlen.isEmpty())
            throw new Exception("Es liegen keine PUnktezahlen vor!");

        if ((NachpruefGrenze < 1 ) || (NachpruefGrenze >= BestandenGrenze))
```

Aufgabe Zweigüberdeckungsverfahren

1. Arbeitet euch in den Quellcode ein und erstellt einen Kontrollflussgraphen für die Funktion *berechneKategorien()*
2. Überlegt euch Testfälle mit den zugehörigen Testdaten, damit alle Kanten und Pfade durchlaufen werden
3. Überlegt euch Grenzen/Schwächen dieses Testverfahrens

Kontrollflussgraph



Testfälle

TF1: Punktezahl = leer
NPG = 30
BG = 70

n1 - n2 - n final

TF2: Punktezahl = 70
NPG = 0
BG = 70

n1 - n2 - n3 - n final

TF3: Punktezahl = 75
NPG = 50
BG = 110

n1 - n2 - n3 - n4 - n final

TF4: Punktezahl = 100, 50, 51
NPG = 50
BG = 100

n1 - n2 - n3 - n4 - n5 - n6 - n7 - n8 -
n5 - n6 - n7 - n9 - n11 -
n5 - n6 - n7 - n9 - n10 - n5 - n final

Grenzen / Schwächen des Verfahrens:

- Unzureichend bei komplexen (zusammengesetzten) Bedingungen
- Testen von Kontrollstrukturen schwer (Schleifen etc.)
- ungeeignet zum Identifizieren von logischen Fehlern (ohne Spezifikation)

Unit Testing

Unit Tests

- Unit Tests dienen der Überprüfung von Komponenten/Modulen und deren Arbeitsweise
- Sie sollen sicherstellen, dass gegebene Eingabeparameter auch das gewünschte Resultat/Verhalten liefern
- In der agilen Entwicklung spielen Unit-Tests eine große Rolle
- Konstante Ausführbarkeit des Quellcode nach Anpassungen soll gewährleistet werden (Continuous Integration)
- Ziel: automatisierte Testbarkeit des existierenden Quellcodes

Vorgehen beim Unit-Testing

1. Unit / Methode wählen
2. Mögliche Eingabeparameter identifizieren
3. Gewünschte Rückgabeparameter / Verhalten für Eingaben identifizieren
4. Testfall formulieren (Kombination Eingabeparameter, Rückgabe/Resultat)

Hilfreich:

- Struktogramme
- Schreibtischtest

JUnit 5 - Assertions

```
assertArrayEquals (expected, actual, message)

assertEquals(expected, actual, message)

assertFalse(condition, message)

assertNull(object, message)

assertThrows(expectedType, executable)
```

JUnit 5 - Annotations

@Test	Kennzeichnen der Methode als Test
@BeforeEach	Annotierte Methode wird vor jedem @Test ausgeführt
@AfterEach	Annotierte Methode wird nach jedem @Test ausgeführt
@BeforeAll	Annotierte Methode wird vor der Ausführung aller mit @Test annotierten Methoden einmalig ausgeführt
@AfterAll	Annotierte Methode wird vor der Ausführung aller mit @Test annotierten Methoden einmalig ausgeführt
@Disabled	Deaktiviert eine Testmethode

JUnit 5

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;


class FirstJUnit5Tests {

    @Test
    void myFirstTest() {
        assertEquals(2, 1 + 1);
    }

}
```

JUnit 5

Website: https://junit.org/junit5/

User Guide: https://junit.org/junit5/docs/current/user-guide/

JavaDoc: https://junit.org/junit5/docs/current/api/overview-summary.html

Unit Testing Hands-On: Kreis

Aufgabe: Taschenrechner

Aufgabe: Taschenrechner

1. Identifiziert zu testende Module
2. Definiert Testfälle für die Module
3. Implementiert die Testfälle mit JUnit
4. Identifiziert zusätzliche Fehlerquellen im Programmcode
5. Definiert sinnvolle Tests für die identifizierten Fehlerquellen oder refactort den Quellcode

Hausaufgabe / Follow Up

Aufgabe Bibliothek:

- Identifizieren der Module
- Definition der Testfälle
- Implementierung der Tests mit JUnit
- Identifizierung weiterer Fehlerquellen (logische/strukturelle Fehler, etc.)
- Refactoring des Quellcodes