



DevOps Unit Tests



Building Competence. Crossing Borders.

Testing

Testing im Software Engineering

Testpyramide

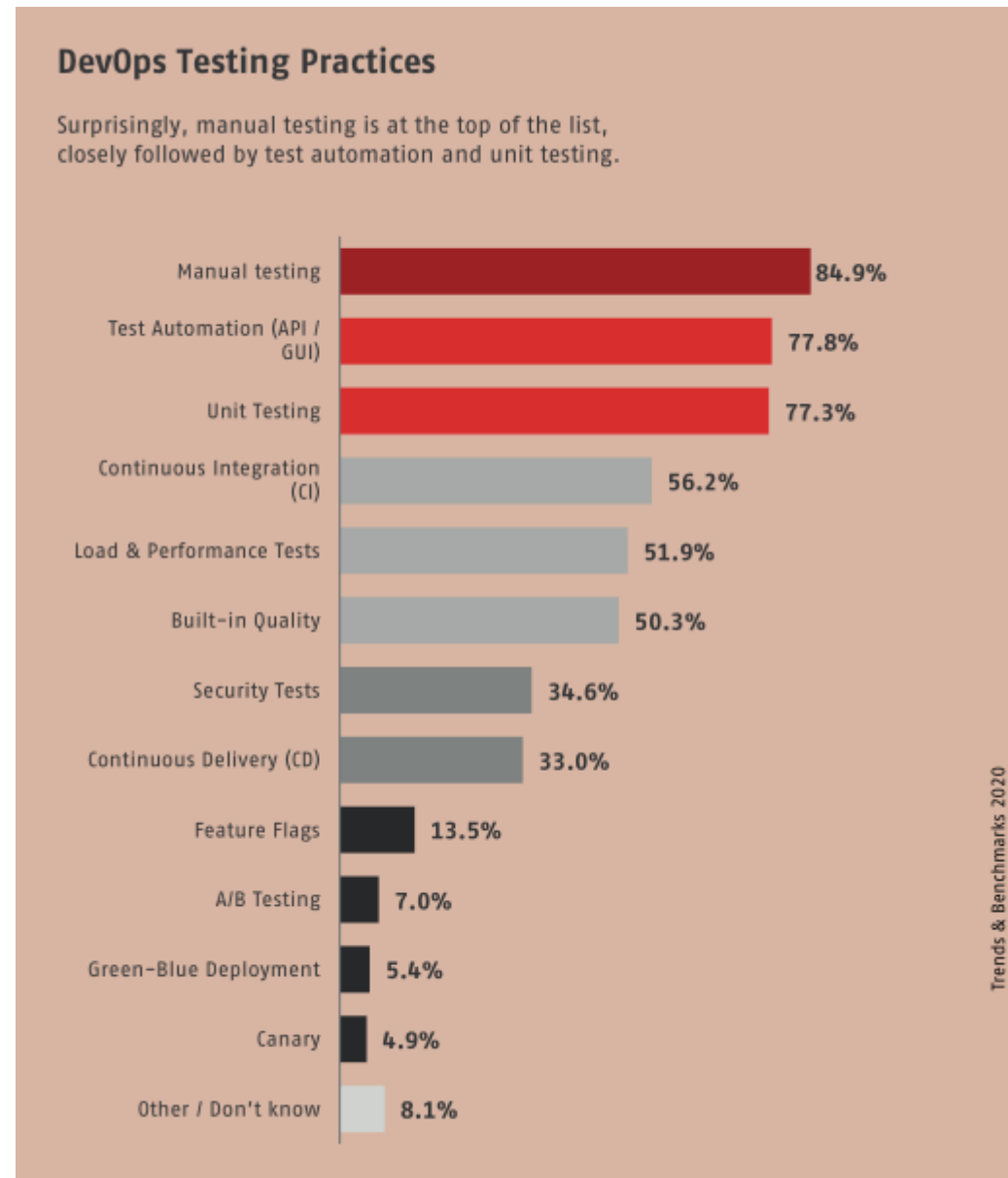
Testbegriffe (Testgrösse, Zeitpunkt, Ziele)

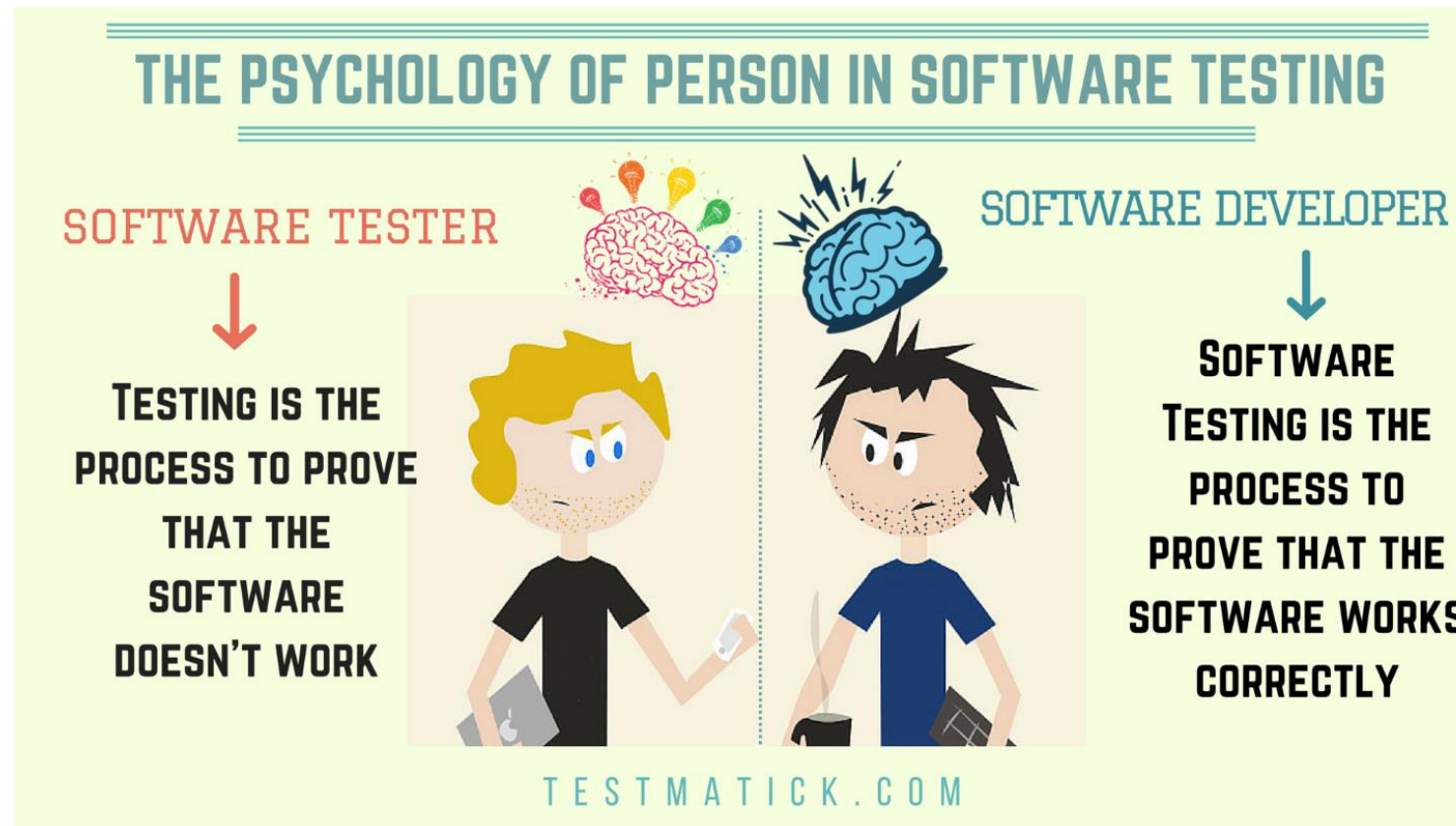
Testen mit JUnit

Code Qualität, Testabdeckung (nächste Woche)

Blackbox Tests (nächste Woche)

DevOps Testing





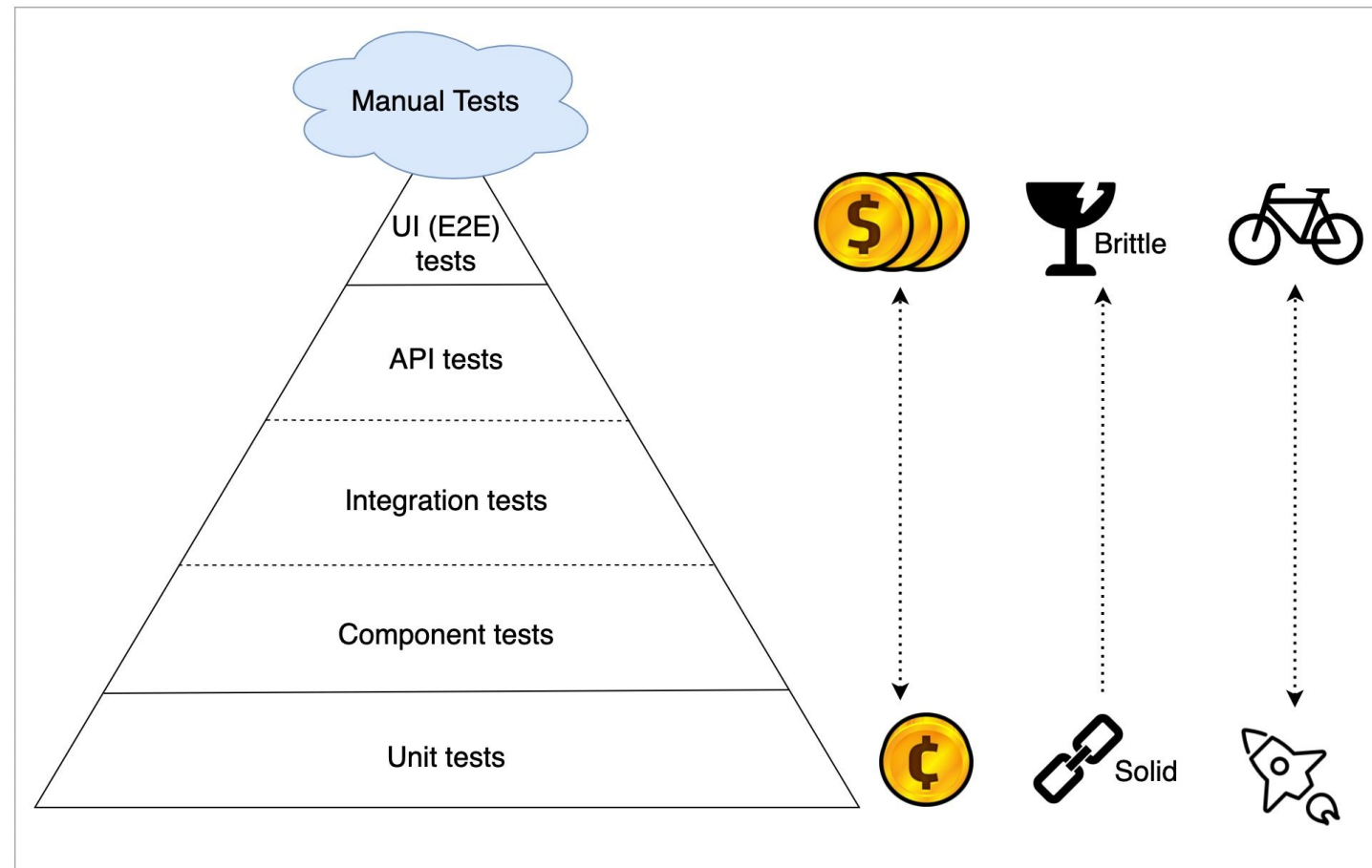
Ziele des Testens

Ziele:

- Dem Entwickler und dem Kunden zeigen, dass die Software die **Anforderungen erfüllt**. Dies bedeutet, dass es mindestens einen Test für jede Anforderung geben sollte.
- Situationen aufspüren, in denen sich die Software **falsch** oder **unerwünscht** verhält. Diese Situation sind typischerweise nicht in den Anforderungen enthalten, weil an die jeweilige Situation nicht gedacht wurde. Viele Angriffe auf Systeme nutzen solches Fehlverhalten aus.

Testpyramide

Software Engineering: Automatisierte Tests als **Versicherung**



Quelle: <https://getmason.io/blog/post/test-pyramid>

Testpyramide

Eigenschaften der verschiedenen Tests:

- Kosten?
- Wartung?
- Versicherung?

Type	No. of tests	Execution speed	Stability	Written by
Unit	70%	Fast	Solid	Developers
Integration	20%	Medium	Good	QA/Developers
UI (E2E)	10%	Slow	Fragile	QA

Quelle: <https://getmason.io/blog/post/test-pyramid>

Test	Beschreibung
Code-Analyse	Analyse des Codes durch Menschen (manuell) oder Software (automatisch).
Code-Review	Überprüfen von Code-Anpassungen oder neuem Code durch weitere Programmierer.
Continuous Testing	Tests im Rahmen von Continuous Integration/Delivery automatisch ausführen ohne manuelle Eingriffe.

Test	Beschreibung
Unittest	Kleinste Testeinheit, z.B. Test einer Methode. Häufig ist ein vollständiger Test möglich.
Integrationstest	Mehrere aufeinander abgestimmte Einzeltests welche das Zusammenspiel mehrerer Komponenten testen.
Systemtest	Gesamtes System wird getestet.

Modultests (Unit Tests) werden in der «untersten Ebene» eingesetzt, um das korrekte Verhalten einzelner Methoden und Klassen zu überprüfen. Modultests sind ausführbar, werden meist von Entwicklern geschrieben und verringern die Anzahl notwendiger manueller Tests.

Komponententests: Mehrere Klassen werden verbunden, um zusammengesetzte Komponenten zu erzeugen. Der Test konzentriert sich auf das Testen der Komponentenschnittstellen.

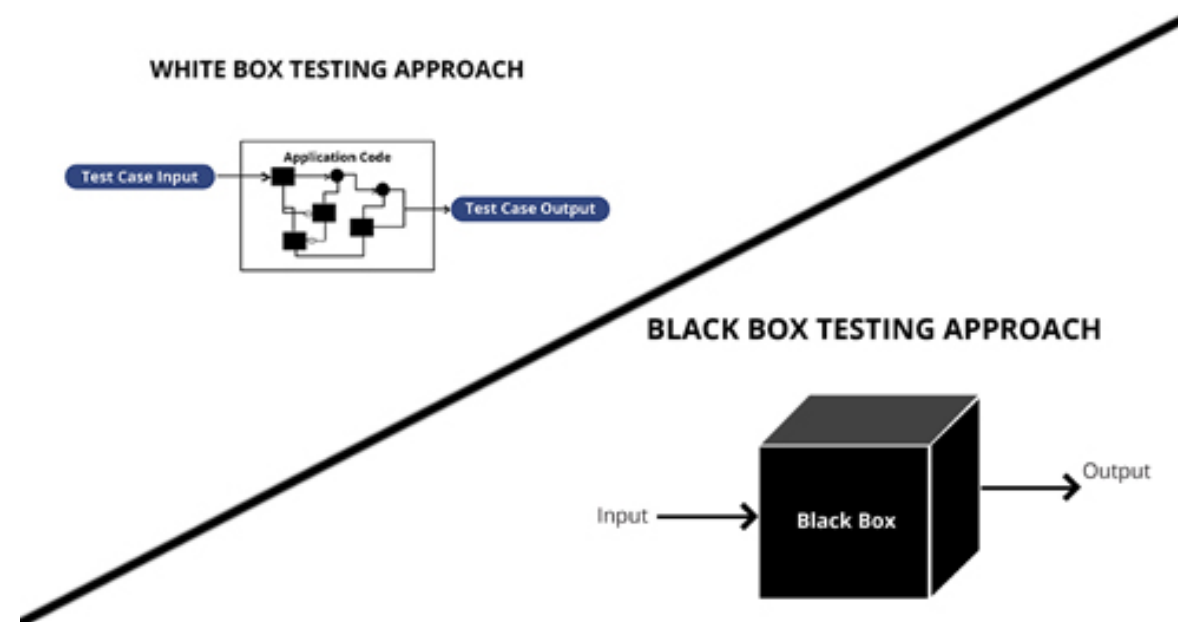
Systemtest: Einige oder alle Komponenten des Systems werden integriert und das System wird als Ganzes getestet. Der Test konzentriert sich auf das Testen der Interaktionen zwischen den Komponenten und beteiligten Akteuren (Benutzer, Drittsysteme).

Test	Beschreibung
Smoke Testing	Erster, oberflächlicher Test einer neuen Software oder einer neuen Installation.
Alpha Testing	Testen eines frühen, noch unfertigen Standes der Software, häufig durch Personen des Projektteams selbst und mit genügend Wissen über den Zustand.
Beta Testing	Testen einer funktional meist vollständigen Software, häufig durch ausgewählte Endbenutzer.
Regression Testing	Tests von vorhergehenden Softwareversionen werden wiederholt, um sicherzustellen dass die bestehenden Funktionen auch weiterhin korrekt funktionieren.
Acceptance Testing	Test ob die Software den Anforderungen entspricht.

Test	Beschreibung
Destructive Testing	Testen mit Fokus auf Fehlersuche und Abweichungen vom «Happy Path».
Performance Testing	Testen mit hoher Last (z.B. viele parallele Benutzer) mit dem Ziel die Belastungsgrenze zu finden.
Usability Testing	Benutzerfreundlichkeit sicherstellen.
Accessibility Testing	Testen ob die Software für Menschen mit eingeschränkten Fähigkeiten (Hörbeeinträchtigung, Farbblindheit, etc.) ausgelegt ist.
Security Testing	Testen ob Daten und Funktionen der Software vor unerlaubtem Zugriff geschützt sind.

Unterscheidungen

- Functional vs. Non-functional Testing
- Statischer Test vs. Dynamischer Test
- Blackbox Testing vs. Whitebox Testing



Quelle: invensis.net

Statische und dynamische Codeanalyse

Die Prüfung kann auf zwei Arten erfolgen:

- **Statische Analyse:** Der Programmcode wird geprüft, ohne dass er ausgeführt wird. Die Analyse erfolgt vor oder während dem Kompilieren. Die Analyse kann manuell (Code-Review), oder automatisch mit entsprechenden Werkzeugen durchgeführt werden (z.B. PMD bei Java).
- **Dynamische Analyse:** Das Programm wird ausgeführt und dabei analysiert. In der Regel wird das Verhalten des Programms bei festgelegten Inputs (Testcases) untersucht. Für die Auswertung wird der Output des Programms, sowie das Verhalten auf Systemebene (z.B. Latenz, CPU-Auslastung) betrachtet.

Unit Tests und JUnit

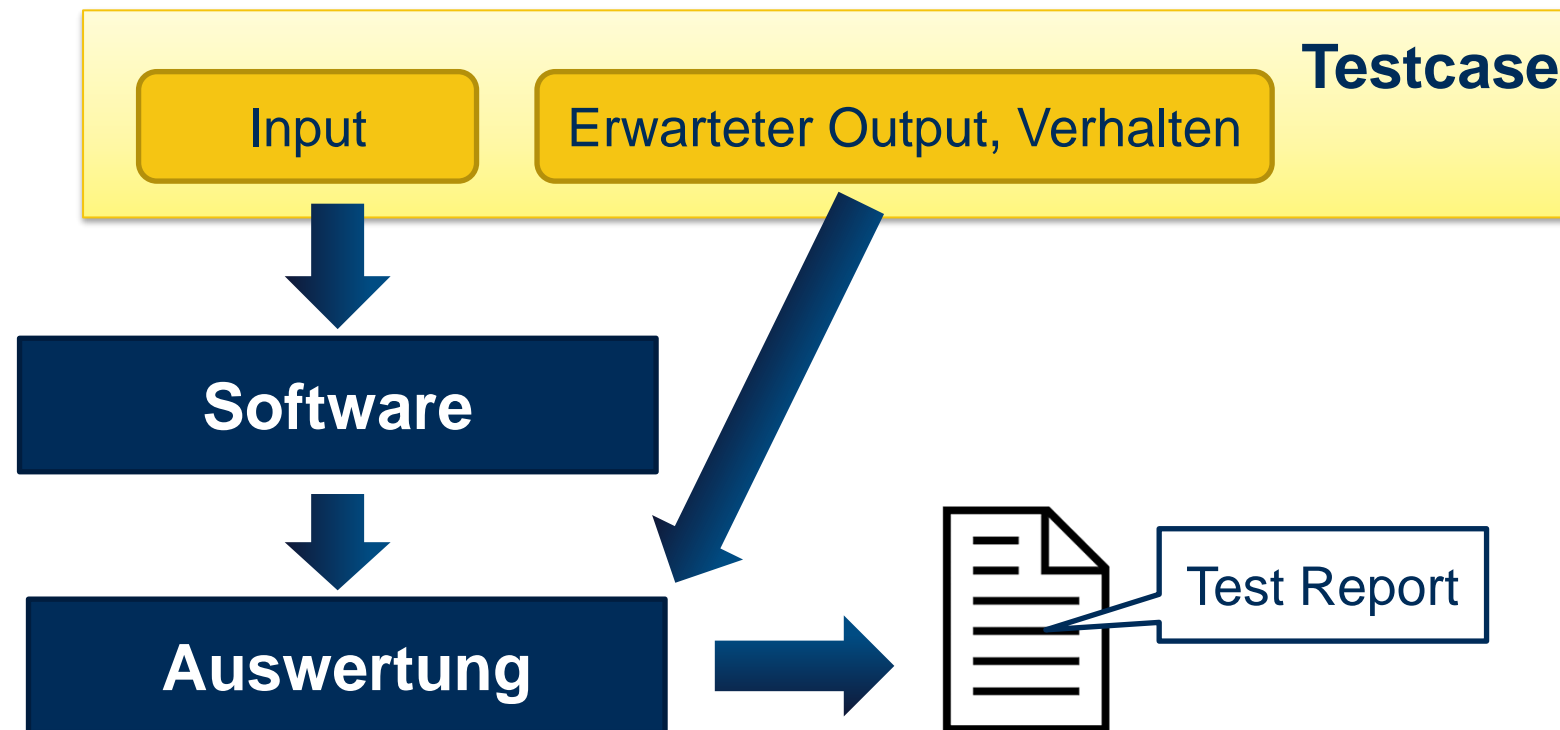
Themen:

- Testcases, automatisierte Tests
- Testgetriebene Softwareentwicklung
- Testen mit JUnit



Testcases für die dynamische Analyse

Für die dynamische Analyse werden Testcases bestehend aus Input-Daten und den zu erwartenden Output-Daten sowie dem Verhalten festgelegt. Bei den Daten kann es sich um «echte» Daten handeln, oder auch um Benutzerinteraktionen wie Maus-Clicks, Tastatureingaben oder den ausgegebenen Bildschirminhalt.



JUnit Beispiel

JUnit wird vorwiegend für Modultest eingesetzt, teilweise auch für Komponententests.

```
public class Monster {  
    private int gesundheit = 100;  
  
    public void angriff(int staerke) {  
        if (gesundheit > staerke) {  
            gesundheit = gesundheit - staerke;  
        } else {  
            System.out.println("besiegt");  
            gesundheit = 0;  
        }  
    }  
  
    public int getGesundheit() {  
        return gesundheit;  
    }  
}
```

```
import static org.junit.Assert.*;  
import org.junit.Before;  
import org.junit.Test;  
  
public class MonsterTest {  
    private Monster m;  
  
    @BeforeEach  
    public void setUp() throws Exception {  
        m = new Monster();  
    }  
  
    @Test  
    public void testAngriff() {  
        m.angriff(30);  
        assertTrue(m.getGesundheit() == 70);  
    }  
  
    @Test  
    public void testGetGesundheit() {  
        assertTrue(m.getGesundheit() == 100);  
    }  
}
```

JUnit Syntax

Mit **Annotations** werden Code-Bereiche markiert. Beispiele:

Ausdruck	Bedeutung
@BeforeAll, @BeforeEach	Markiert eine Methode, die vor jedem Test ausgeführt werden muss.
@Test	Markiert eine Methode als Test.

Mit **Assert-Ausdrücken** wird das Ergebnis geprüft. Beispiele:

Ausdruck	Bedeutung
assertTrue(boolean condition)	Der Test scheitert, wenn die Bedingung nicht <true> ist.
assertEquals(expected, actual)	Der Test scheitert, wenn die beiden Werte nicht übereinstimmen.

DevOpsDemo Unit Test Beispiel

Testet und prüft die Erstellung eines ToDo mit Controller:

```
package ch.zhaw.iwi.devops.demo;

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

import org.junit.jupiter.api.Test;

public class ToDoControllerTest {

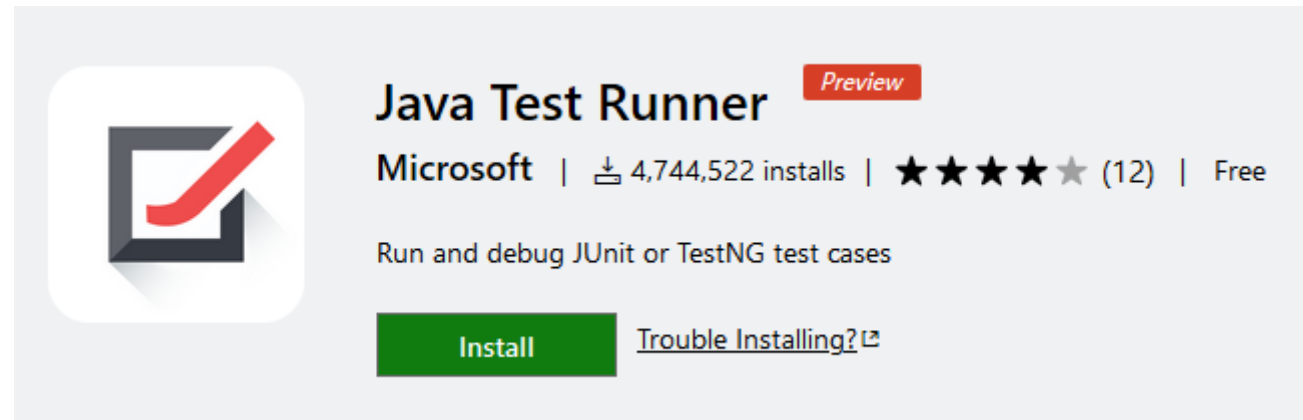
    @Test
    public void testCreate() {
        ToDoController controller = new ToDoController();
        var todo = new ToDo(1, "t", "d");
        controller.createTodo(1, todo);
        assertEquals(1, controller.count());
        assertEquals(1, controller.todo().size());
    }

}
```

JUnit in VisualStudio Code

Tutorial

<https://marketplace.visualstudio.com/items?vscjava.vscode-java-test>



Java Test Runner

Tutorial

The screenshot displays an IDE window titled 'ToDoControllerTest.java - DevOps (Workspace) - Visu...'. The interface is divided into several panels:

- TESTING** (Left): Shows a list of tests. The first test, 'demo', is expanded, showing 'ch.zhaw.iwi.devops.demo' and 'ToDoControllerTest'. The 'testCreate()' method is highlighted with a green checkmark and a '149ms' duration. A yellow callout bubble labeled 'Test-Sicht' points to this section.
- Code Editor** (Center): Displays the source code of 'ToDoControllerTest.java'. The code includes imports for 'org.junit.jupiter.api.Assertions' and 'org.junit.jupiter.api.Test'. The 'testCreate()' method is highlighted with a green checkmark and a '149ms' duration. A yellow callout bubble labeled 'Test starten' points to this section.
- DEBUG CONSOLE** (Bottom): Shows the output of the test run, including the message 'Initializing Servlet 'dispatcherServlet'' and 'Completed initialization in 8 ms'.
- TERMINAL** (Bottom): Shows the command 'node' being executed.

At the bottom of the IDE, the status bar indicates 'Ln 4, Col 1', 'Spaces: 4', 'UTF-8', 'CRLF', and 'Java'.

Ziel:

- Mit **Visual Studio Code**...
- ein **Spring Boot** Projekt erstellen...
- und einen ersten **JUnit** Test ausführen.

Aufgabe:

- JUnit Projekt erstellen und vorgefertigte Klasse **DemoApplicationTests** ausführen

Vorgehen (detaillierte Beschreibung siehe Übung 1 der ersten Woche):

- Visual Studio Code: New Window
- Ctrl-Shift-P: Sprint Initializr: Create Maven Project
- Parameter: Version / Java / com.example.demo / demo / Jar / 17 / 0 Dependencies
- Open / Yes, trust the Authors

Unsere Tests sollen Code testen! Tests gehören immer zu Code, der getestet wird.

Konventionen:

- Name der Testklasse: Klasse + Test
- Gleiches Package, aber in **main** (Code) bzw. **test** (Test)

```
public class MathLib {  
  
    public static boolean isEven(int value) {  
        return value % 2 == 0;  
    }  
  
}
```

```
import static  
org.junit.jupiter.api.Assertions.assertTrue;  
import org.junit.jupiter.api.Test;  
  
public class MathLibTest {  
  
    @Test  
    public void testEven1() {  
        assertTrue(MathLib.isEven(2));  
    }  
  
}
```

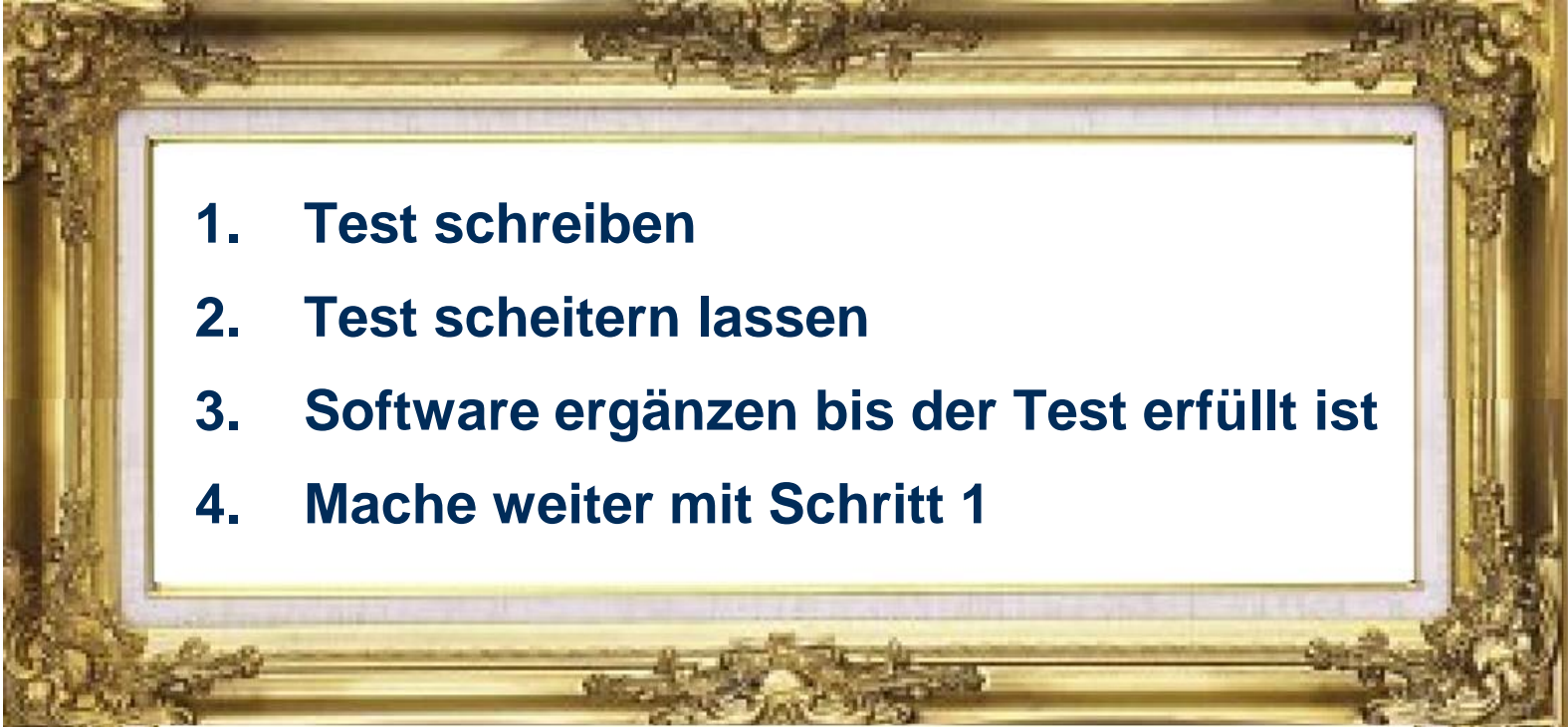
TDD

Test Driven Development

macht DevOps erst so richtig möglich

Testgetriebene Softwareentwicklung

Bei der testgetriebenen Softwareentwicklung wird vor dem Implementieren neuer Funktionen immer zuerst ein **ausführbarer Test** geschrieben, welcher diese neue Funktionalität testet. Erst danach wird implementiert.

- 
1. **Test schreiben**
 2. **Test scheitern lassen**
 3. **Software ergänzen bis der Test erfüllt ist**
 4. **Mache weiter mit Schritt 1**

Anforderungen

Schreiben Sie ein Programm, das die Zahlen von 1 bis 100 ausgibt.

Bei jeder Zahl, die durch 3 teilbar ist, soll «fizz» ausgegeben werden und bei jeder Zahl, die durch 7 teilbar ist, soll «buzz» ausgegeben werden.

Wenn die Zahl sowohl durch 7 als auch durch 3 teilbar ist, soll «fizzbuzz» ausgegeben werden.

Hinweise

Der Modulo-Operator ermittelt den Rest bei Division. Somit ist eine Teilbarkeit einfach dann erreicht, wenn die Modulo-Operation (% , MOD) den Rest 0 liefert.

FizzBuzz Lösungsidee

Idee

Eine neue Klasse (FizzBuzzConverter) mit einer Methode. Diese Methode nimmt eine «gewöhnliche» Zahl entgegen und konvertiert die Zahl (sofern notwendig) in eine FizzBuzz-Zahl

Beispiele

FizzBuzzConverter.convert(1)	→	Rückgabewert 1
FizzBuzzConverter.convert(2)	→	Rückgabewert 2
FizzBuzzConverter.convert(3)	→	Rückgabewert Fizz
FizzBuzzConverter.convert(4)	→	Rückgabewert 4
FizzBuzzConverter.convert(5)	→	Rückgabewert 5
FizzBuzzConverter.convert(6)	→	Rückgabewert Fizz
FizzBuzzConverter.convert(7)	→	Rückgabewert Buzz

FizzBuzz Schritt 1

TDD

Als erster Schritt wird ein Test geschrieben! Achtung: Der Code wird **nicht** kompilieren, da es die Klasse FizzBuzzConverter gar noch nicht gibt.

```
@Test
public void fizzBuzzConverter1() {

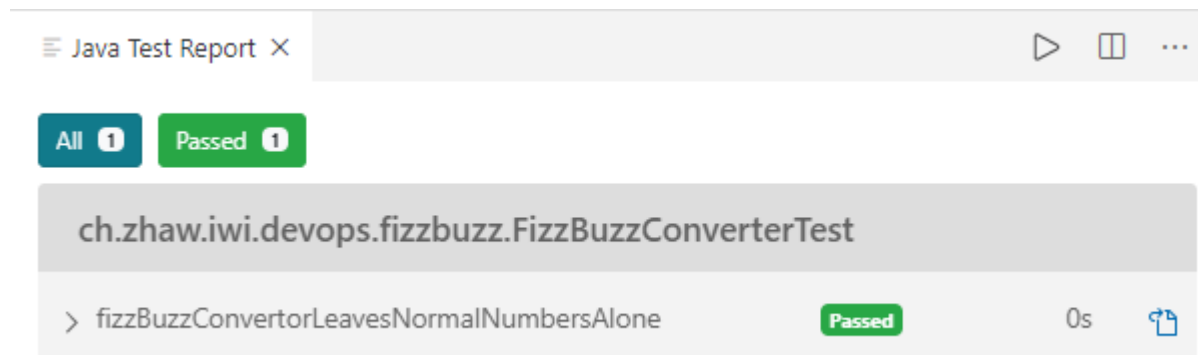
    FizzBuzzConverter fizzBuzz = new FizzBuzzConverter();
    Assert.assertEquals("1", fizzBuzz.convert(1));

}
}
```

FizzBuzz Schritt 2

Nun wird die fehlende Klasse erstellt. Der Code kompiliert und der Test wird grün.

```
public class FizzBuzzConverter {  
  
    public String convert(int i) {  
        return "1";  
    }  
  
}
```



The screenshot shows a Java Test Report interface. At the top, there's a tab labeled 'Java Test Report' with a close button. Below the tab, there are two buttons: 'All 1' and 'Passed 1'. The main content area shows a test class 'ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest'. Underneath, a single test case 'fizzBuzzConvertorLeavesNormalNumbersAlone' is listed with a 'Passed' status, a duration of '0s', and a document icon.

ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest		
> fizzBuzzConvertorLeavesNormalNumbersAlone	Passed	0s

FizzBuzz Schritt 3

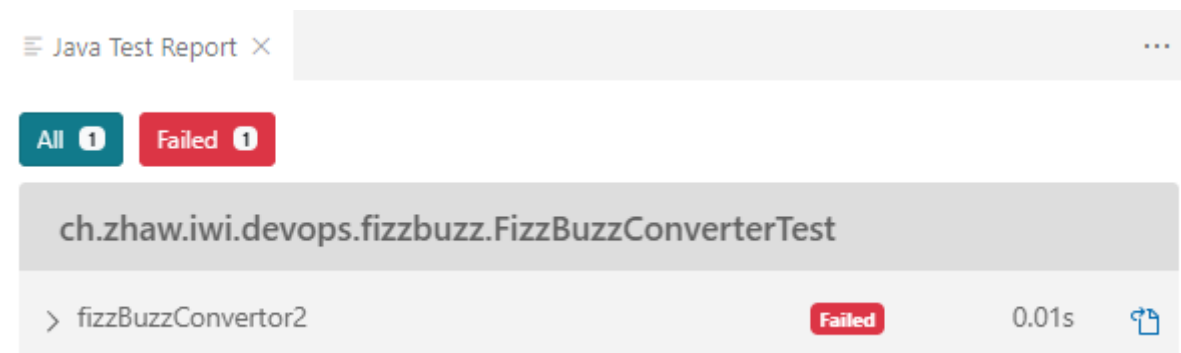
Nicht die Logik weiterprogrammieren, sondern wieder einen **neuen Test** schreiben!

```
@Test
public void fizzBuzzConvertor2() {

    FizzBuzzConverter fizzBuzz = new FizzBuzzConverter();
    Assert.assertEquals("2", fizzBuzz.convert(2));

}
```

Dieser Test ist üblicherweise rot:



FizzBuzz Schritt 4

Nun wird der Code angepasst und es werden **alle bestehenden** (zwei) **Tests** gestartet:



```
public class FizzBuzzConverter {  
  
    public String convert(int i) {  
        return String.valueOf(i);  
    }  
  
}
```

Java Test Report

All 2

Passed 2

ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest

> fizzBuzzConverter1	Passed	0.01s	
> fizzBuzzConvertor2	Passed	0s	

FizzBuzz Schritt 5

Nun müssen weitere Tests erstellt werden. Es dürfen auch mehrere Tests sein:

```
@Test
public void fizzBuzzConvertor3() {
    FizzBuzzConverter fizzBuzz = new FizzBuzzConverter();
    Assert.assertEquals("Fizz", fizzBuzz.convert(3));
}

@Test
public void fizzBuzzConvertorMultiplesOfThree() {
    FizzBuzzConverter fizzBuzz = new FizzBuzzConverter();
    Assert.assertEquals("Fizz", fizzBuzz.convert(6));
}
```

All 4	Failed 2	Passed 2
ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest		
> fizzBuzzConverter1	Passed	0s
> fizzBuzzConvertor2	Passed	0s
> fizzBuzzConvertor3	Failed	0.01s
> fizzBuzzConvertorMultiplesOfThree	Failed	0s

FizzBuzz Schritt 6





Da es fehlgeschlagene Tests gibt, muss der Code in Ordnung gebracht werden:

```
public String convert(int i) {  
    if (i%3 == 0) {  
        return "Fizz";  
    }  
    return String.valueOf(i);  
}
```

All 4

Passed 4

ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest

> fizzBuzzConverter1	Passed	0.01s	
> fizzBuzzConvertor2	Passed	0s	
> fizzBuzzConvertor3	Passed	0s	
> fizzBuzzConvertorMultiplesOfThree	Passed	0s	

FizzBuzz Schritt 7

Nun muss ein weiterer Test geschrieben werden – **Test first!**

```
@Test
public void fizzBuzzConvertorMultiplesOfSeven() {






    FizzBuzzConverter fizzBuzz = new FizzBuzzConverter();
    Assert.assertEquals("Buzz", fizzBuzz.convert(7));
}
```

All	5	Failed	1	Passed	4
ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest					
> fizzBuzzConverter1	Passed	0.01s			
> fizzBuzzConvertor2	Passed	0s			
> fizzBuzzConvertor3	Passed	0s			
> fizzBuzzConvertorMultiplesOfSeven	Failed	0.01s			
> fizzBuzzConvertorMultiplesOfThree	Passed	0s			

FizzBuzz Schritt 8

Der Code wird wieder korrigiert, so dass der Test grün wird:

```
public String convert(int i) {  
    if (i % 3 == 0) {  
        return "Fizz";  
    } else if (i % 7 == 0) {  
        return "Buzz";  
    }  
    return String.valueOf(i);  
}
```

All	5	Passed	5
ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest			
> fizzBuzzConverter1	Passed	0s	
> fizzBuzzConverter2	Passed	0s	
> fizzBuzzConverter3	Passed	0s	
> fizzBuzzConverterMultiplesOfSeven	Passed	0s	
> fizzBuzzConverterMultiplesOfThree	Passed	0s	

FizzBuzz Schritt 9

Die Ausgabe für Vielfache von 3 **und** 7 muss «FizzBuzz» lauten.

Hinweise: Es sind auch mehrere Asserts in einem Test möglich. Zudem bietet Assert viele weitere Möglichkeiten, z.B. `assertNotEquals`.

```
@Test
public void fizzBuzzConvertorMultiplesOfThreeAndSeven() {
    FizzBuzzConverter fizzBuzz = new FizzBuzzConverter();
    Assert.assertNotEquals("FizzBuzz", fizzBuzz.convert(14));
    Assert.assertEquals("FizzBuzz", fizzBuzz.convert(21));
    Assert.assertEquals("FizzBuzz", fizzBuzz.convert(42));
    Assert.assertEquals("FizzBuzz", fizzBuzz.convert(63));
}
```

> fizzBuzzConvertorMultiplesOfThreeAndSeven

Failed

0.01s

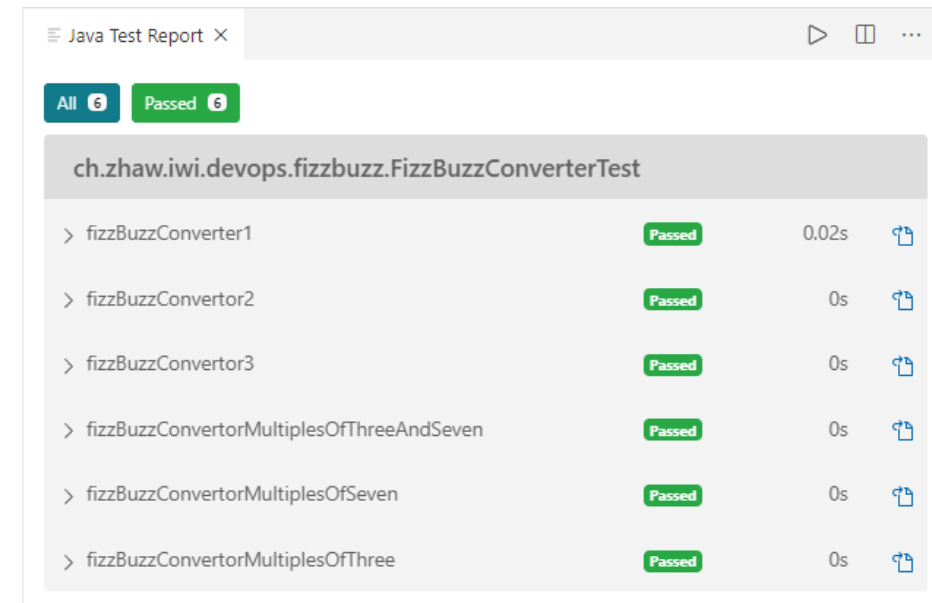


FizzBuzz Schritt 10

Der Code wird wieder korrigiert, so dass nun alle Testfälle grün sind.

Refactoring: Nun kommt der grosse Vorteil von TDD. Der Code kann nun optimiert und angepasst werden, solange alle Testfälle weiterhin grün sind. Dieser Vorgang heisst Refactoring und wird durch TDD erst richtig ermöglicht.

```
public String convert(int i) {  
    if (i % 3 == 0 && i % 7 == 0) {  
        return "FizzBuzz";  
    } else if (i % 3 == 0) {  
        return "Fizz";  
    } else if (i % 7 == 0) {  
        return "Buzz";  
    }  
    return String.valueOf(i);  
}
```



The screenshot shows a 'Java Test Report' window. At the top, there are two buttons: 'All 6' and 'Passed 6'. Below this, the test class 'ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest' is listed. A table follows with seven rows, each representing a test case. All test cases are marked as 'Passed' with a green status icon. The execution times are listed in the third column, and there are icons for expanding/collapsing and saving each test case.

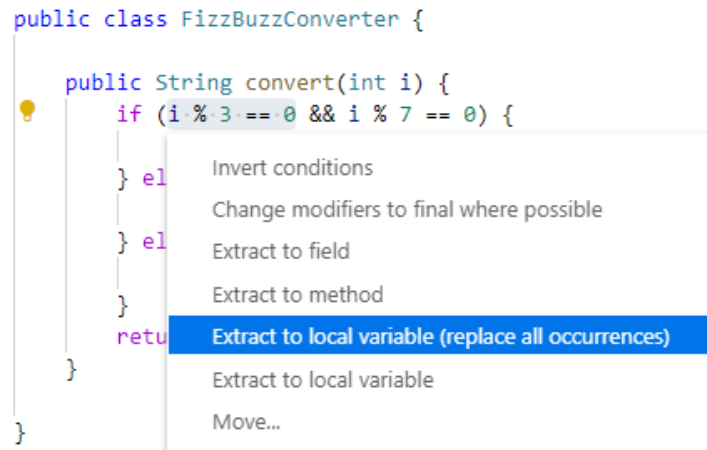
ch.zhaw.iwi.devops.fizzbuzz.FizzBuzzConverterTest		
> fizzBuzzConverter1	Passed	0.02s
> fizzBuzzConvertor2	Passed	0s
> fizzBuzzConvertor3	Passed	0s
> fizzBuzzConvertorMultiplesOfThreeAndSeven	Passed	0s
> fizzBuzzConvertorMultiplesOfSeven	Passed	0s
> fizzBuzzConvertorMultiplesOfThree	Passed	0s

FizzBuzz Refactoring – ohne Fehler einzubauen

Refactoring

- doppelten Code in Variablen auslagern
- doppelten Code in Methoden auslagern
- Variablen sauber benennen
- Formatierung

```
public class FizzBuzzConverter {  
    public String convert(int i) {  
        if (i % 3 == 0 && i % 7 == 0) {  
        } else {  
        } else {  
        }  
        return  
    }  
}
```



```
public String convert(int input) {  
    boolean isMultipleOf3 = input % 3 == 0;  
    boolean isMultipleOf7 = input % 7 == 0;  
  
    if (isMultipleOf3 && isMultipleOf7) {  
        return "FizzBuzz";  
    } else if (isMultipleOf3) {  
        return "Fizz";  
    } else if (isMultipleOf7) {  
        return "Buzz";  
    }  
    return String.valueOf(input);  
}
```

Learnings

Unit Test

Kleine Einheiten des Codes testen (z.B. Methode)

JUnit

Framework zur Testentwicklung in Java. Kann für Unit Tests oder Integrationstests eingesetzt werden.

TDD

Test Driven Development. Tests werden jeweils **vor** dem Code geschrieben. Erhöht die Anzahl und die Qualität der Tests.

Lernjournal

Ziel

- Erstellen mehrerer Unit Tests gemäss TDD

Checkliste

- ✓ Quelle der Idee angegeben
- ✓ Anforderungen sind zusammengefasst
- ✓ Jeder Teilschritt wurde dokumentiert (aktueller Code, Test-Resultate) und einzeln committed
- ✓ TDD ist anhand der Commits erkennbar: Test wurde zuerst comitted, danach passender Code
- ✓ Es sind mindestens 10 Tests (Iterationen) für die Methode vorhanden

Ideen für Methoden

- <https://kata-log.rocks/tdd>