

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- ➡ ■ Testautomatisierung

Links und Literatur zu JUnit

- **Bechold et al.: JUnit 5 User Guide:**
<https://junit.org/junit5/docs/current/user-guide/>
- **Gulati, Sharma: Java Unit Testing with JUnit 5**, apress, 2017
- **JUnit5 Tutorial:** <https://howtodoinjava.com/junit-5-tutorial/>
- **Unfassende Darstellung (JUnit 4):**
M. Tamm: JUnit Profiwissen, 1. Auflage 2013, dpunkt.verlag
- **JUnit Website incl Tutorials:**
 - <http://junit.org/junit4/>
 - <https://junit.org/junit5/>
- **JUnit Artikel (JUnit 4)**
<http://www.vogella.com/tutorials/JUnit/article.html>
- **JUnit Tutorial (JUnit 4)**
<http://www.javacodegeeks.com/2014/11/JUnit-tutorial-unit-testing.html>
- **Einführung in JUnit3:** Kent Beck: JUnit Pocket Guide, Kindle Edition, O'Reilly

Achten Sie bei Tutorials und Büchern stets auf die gewünschte Version von JUnit!



Motivation / Idee

- Ziele von JUnit
- Features von JUnit
- Einfaches Beispiel
- TestSuites
- TestFixturees
- JUnit Annotations
- JUnit Assertions
- Namenskonventionen
- Parametrisierbare Tests
- JUnit Tags

Warum überhaupt automatisieren?

➔ Es gibt ganz viele Argumente.

Hauptargument:

Vertrauen in die eigene Arbeit

Zeitliche Vorteile der Testautomatisierung

- Kurzfristig für den Entwickler
 - Zeitersparnis bei Fehlerfinden und Korrigieren
- Langfristig für den Entwickler
 - Sicherheit, den Code langfristig warten zu können, ohne ihn zu brechen.
- Für das Team und den Kunden
 - Einfache Integration von gutgetestetem Code

Defect Entwicklung bei häufigen Tests

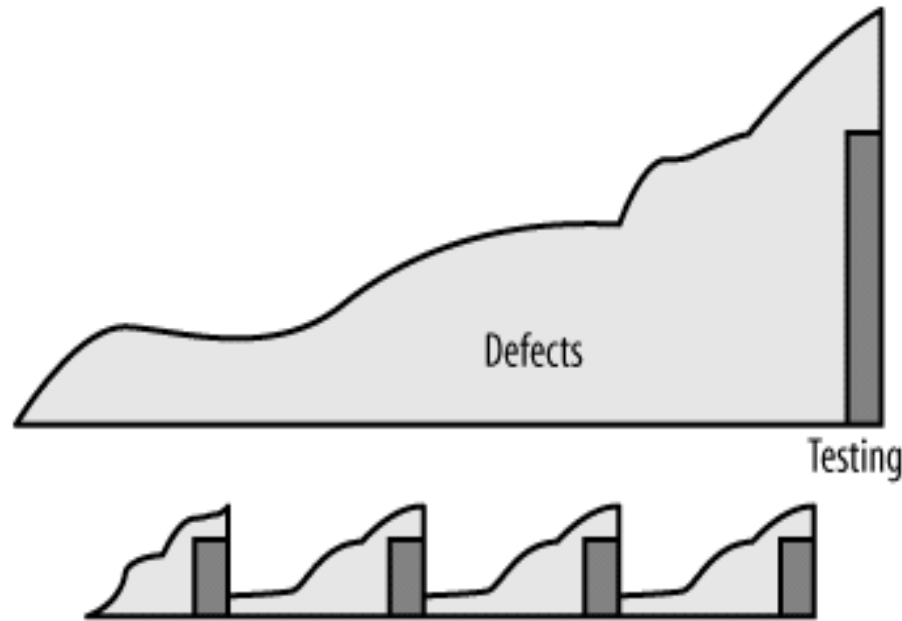


Figure 1-1. Frequent testing leaves fewer defects at the end

Quelle: JUnit Pocket Guide, Kent Beck, Kindle Edition

Kosten für Bugfixes

Software Testing Phase Where Bug Found	Estimate Cost per Bug
System Test	\$5,000
Integration Test	\$500
Full Build	\$50
Unit Test/Test-Driven Development	\$5

Aus: **Gulati, Sharma: Java Unit Testing with JUnit 5**, apress, 2017

TDD:

1. Test schreiben
2. Test fehlschlagen lassen
3. Code schreiben so dass der Test grün ist
4. Sicherstellen, dass alle bisherigen Tests grün sind
5. Refactor
6. 3 bis 5 wiederholen bis man fertig ist

→Regression wird schnell entdeckt

→Design Disziplin

→Sauberes Design möglich (Duch sicheres refactoring)

Bei gemeldeten Bugs wird ein Test geschrieben, um den Fehler zu reproduzieren und dann der Fehler behoben.

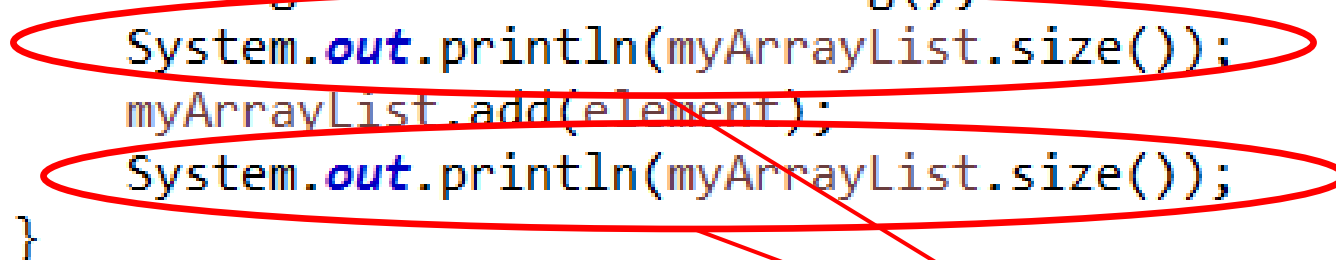
Dieses Vorgehen ist nur mit Automatisierung möglich.

Idee: Wie automatisieren?

Beispiel: Testen der java.util.ArrayList

1. Versuch:

```
public static void firstArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    System.out.println(myArrayList.size());  
    myArrayList.add(element);  
    System.out.println(myArrayList.size());  
}
```



Prüfen und interpretieren!

Beispiel: Testen der java.util.ArrayList

2. Versuch:

```
public static void secondArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    System.out.println(myArrayList.size() == 0);  
    myArrayList.add(element);  
    System.out.println(myArrayList.size()==1);  
}
```

Prüfen!

Beispiel: Testen der java.util.ArrayList

3. Versuch:



Man kriegt mit, wenn der Test fehlschlägt.
➔ Keine manuelle Prüfung, sondern automatisierte Tests

```
public static void automatedArrayListTest(){  
    List <String> myArrayList= new ArrayList<String>();  
    String element = new String();  
    assertTrue(myArrayList.size()==0);  
    myArrayList.add(element);  
    assertTrue(myArrayList.size()==1);  
}
```

```
public static void assertTrue(boolean condition) {  
    if(!condition){  
        throw new RuntimeException("Assertion failed");  
    }  
}
```

Testautomatisierung

- [Motivation / Idee](#)
- ➡ ■ [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

JUnit bietet eine Infrastruktur, um viele Tests automatisiert laufen zu lassen und das Ergebnis wiederzugeben.

JUnit

- Lässt Tests automatisiert laufen.
- Lässt viele Tests gemeinsam laufen und fasst das Ergebnis zusammen.
- Vergleicht Ergebnisse mit Erwartungen und teilt Unterschiede mit.

Ziele von JUnit

- Tests sollen einfach zu schreiben sein.
- Es soll einfach sein, das Schreiben von Tests zu lernen.
- Schnelle Testausführung.
- Einfache Testausführung (per Knopfdruck, einfache Darstellung der Ergebnisse).
- Isolierte Ausführung, keine Beeinflussung von Tests untereinander.
- Tests sollen zusammensetzbar sein.

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
-  [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Features von JUnit

- Infrastruktur für automatisierte Tests
 - Test schreiben
 - Test durchführen
 - Test auswerten
- Test vorbereiten
- Test nachbereiten
- Tests organisieren
- Parametrisierbare Tests
- ...

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Bsp. Calculator

Zu testende Klasse

```
package demopackage;  
  
public class Calculator {  
  
    public int add(int n, int m) {  
        return m + n;  
    }  
  
}
```

Bsp. Calculator

Testklasse

```
package demopackage;
```

```
import static org.junit.jupiter.api.Assertions.*;
```

Import der Assertions

```
import org.junit.jupiter.api.Test;
```

Import der Annotation

```
class MyFirstJUnitJupiterTests {
```

Beliebiger Name der Testklasse

```
    private final Calculator calculator = new Calculator();
```

```
    @Test
```

Kennzeichnung als Test

```
    void addition() {
```

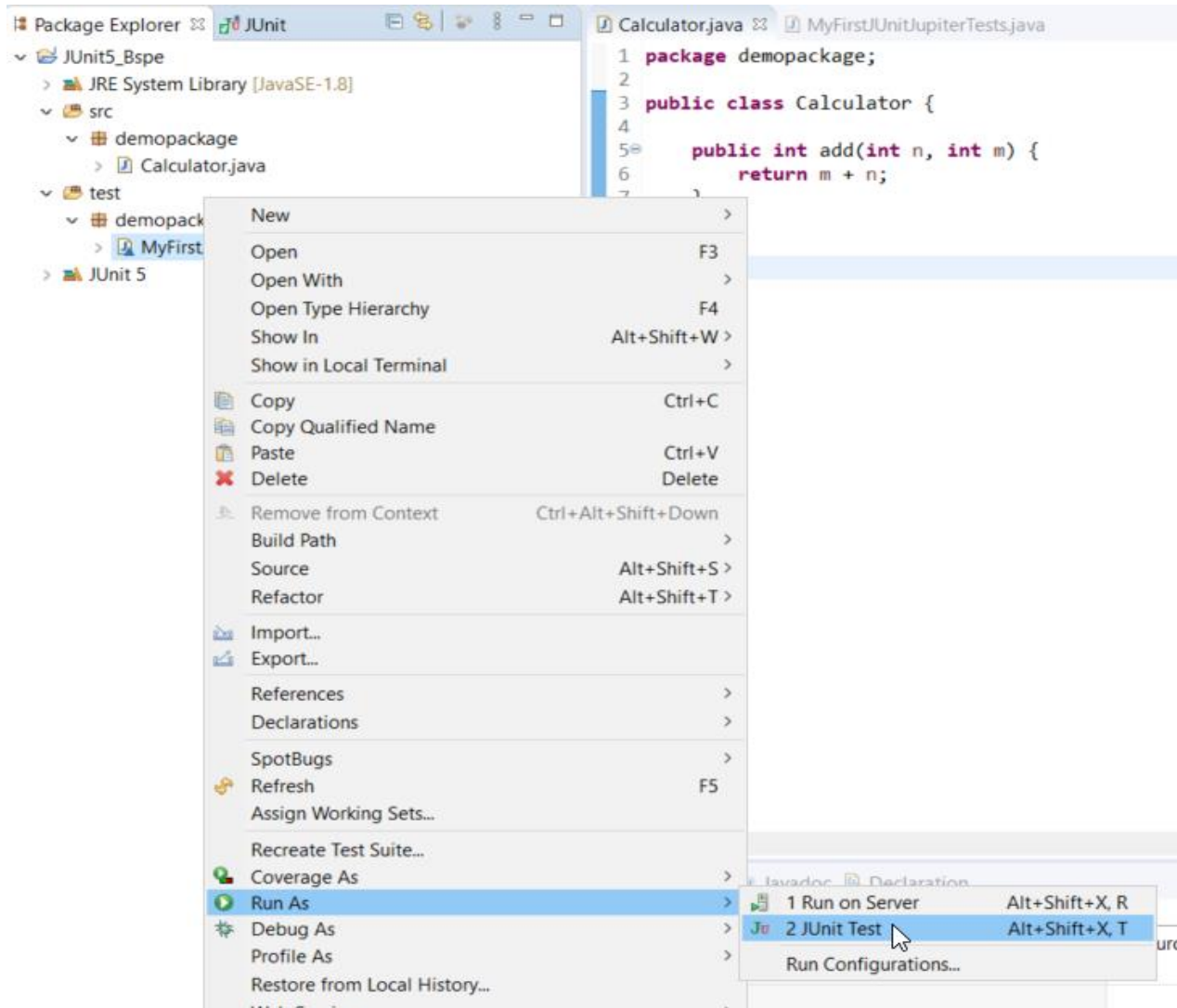
```
        assertEquals(2, calculator.add(1, 1)) ;
```

Assert Methode zur Überprüfung

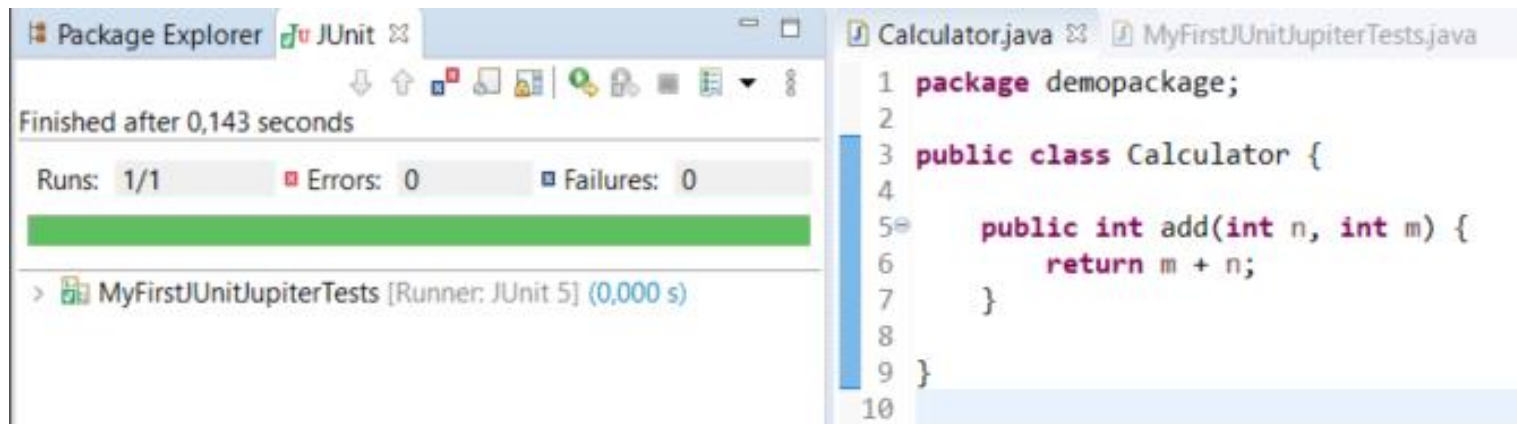
```
    }
```

```
}
```

Test laufen lassen (eclipse)



Ergebnis



The screenshot displays an IDE interface with two main panels. The left panel, titled 'JUnit', shows the results of a test run. It indicates that the test 'MyFirstJUnitJupiterTests' was completed after 0.143 seconds, with 1/1 runs, 0 errors, and 0 failures. A green progress bar is visible. The right panel shows the source code of the 'Calculator.java' file, which defines a 'Calculator' class with an 'add' method. The code is as follows:

```
1 package demopackage;
2
3 public class Calculator {
4
5     public int add(int n, int m) {
6         return m + n;
7     }
8
9 }
10
```

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- ➔ ▪ [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Es kann sinnvoll sein, nur einen Teil der Tests laufen zu lassen (z.B. alle Smoketests, alle Tests zu einer Komponente etc).

→ Unterstützung durch IDE Konfigurationen
Oder

→ Erstellung von **JUnit TestSuites**

→ Standard Weg, unabhängig von einer IDE.

→ Test Suites können einfacher in einer Sourceverwaltung verwaltet werden.

Test Suites

JUnit 5: Möglichkeit Tests gemeinsam laufen zu lassen, die in verschiedenen Testklassen oder verschiedenen Packages liegen.

```
package demopackage;
```

```
import org.junit.platform.runner.JUnitPlatform;  
import org.junit.platform.suite.api.SelectClasses;  
import org.junit.runner.RunWith;
```

Runner für Suites

```
@RunWith(JUnitPlatform.class)  
@SelectClasses({MyFirstJUnitJupiterTests.class, SecondClassUnderTest.class})  
class TestSuiteExample {  
}
```

Annotation für Klassen
der Suite

TestSuites

```
import org.junit.platform.suite.api.SelectClasses;  
import org.junit.platform.suite.api.Suite;
```

```
@Suite  
@SelectClasses({AppTest.class, TagUse.class})  
public class AllTests{}
```

Suite

Auszuführende Testklassen

Annotationen für Testsuites:

- `@SelectPackages()` : Führt die Tests in den angegebenen Packages und den Unterpackages aus.
 - `@IncludePackages()`: führt nur die angegebenen Packages aus.
 - `@ExcludePackages()`: führt alle außer den angegeben Packages aus
- `@SelectClasses()`: Führt die Tests der übergebenen Klassen aus.
- `@IncludeClassNamePatterns()`
- `@ExcludeClassNamePatterns`
- `@IncludeTags()`
- `@ExcludeTags()`

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
-  [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Bsp: Test einer Client-Server-Kommunikation

```
public void testPing(){  
    Server server = new Server();  
    server.start();  
    Client client = new Client();  
    client.start();  
    client.send("ping");  
    assertEquals("ack", client.receive());  
    client.stop();  
    server.stop();  
}
```

Fixtures - Motivation

Bsp: Test einer Client-Server-Kommunikation, sauber

```
public void testPingsauber() {  
    Server server = new Server();  
    server.start();  
    try {  
        Client client = new Client();  
        client.start();  
        try {  
            client.send("ping");  
            assertEquals("ack", client.receive());  
        } finally {  
            client.stop();  
        }  
    } finally {  
        server.stop();  
    }  
}
```

Test vorbereiten

Eigentlicher Test

aufräumen

Fixtures

Deklaration

```
Server server;  
Client client;
```

Test setup

```
protected void bereiteTestVor(){  
    Server server = new Server();  
    server.start();  
    Client client = new Client();  
    client.start();  
}
```

Test Durchführung

```
public void testPing() {  
    client.send("ping");  
    assertEquals("ack", client.receive());  
}
```

Aufräumen

```
protected void raeumeTestauf(){  
    try{  
        client.stop();  
    }finally{  
        server.stop();  
    }  
}
```

Fixtures: Vorbereitungscode und Nachbereitungscode für Testmethoden.

➔ Initialisierung

Nutzen:

- Isolation des eigentliche Testcodes
- Vermeidung redundanten Vor- und Nachbereitungscode

Realisierung von Testfixtures durch Annotationen:

@BeforeEach

The annotated method will be run before each test method in the test class.

@AfterEach

The annotated method will be run after each test method in the test class.

@BeforeAll

The annotated method will be run before all test methods in the test class. This method must be static.

@AfterAll

The annotated method will be run after all test methods in the test class. This method must be static.

Wichtig: JUnit garantiert,

- alle `@After` Methoden werden immer aufgerufen, auch wenn eine davon eine Exception wirft.
- alle `@After` Methoden werden immer aufgerufen, auch wenn eine der `@Before` Methoden eine Exception wirft.

Test Life Cycle

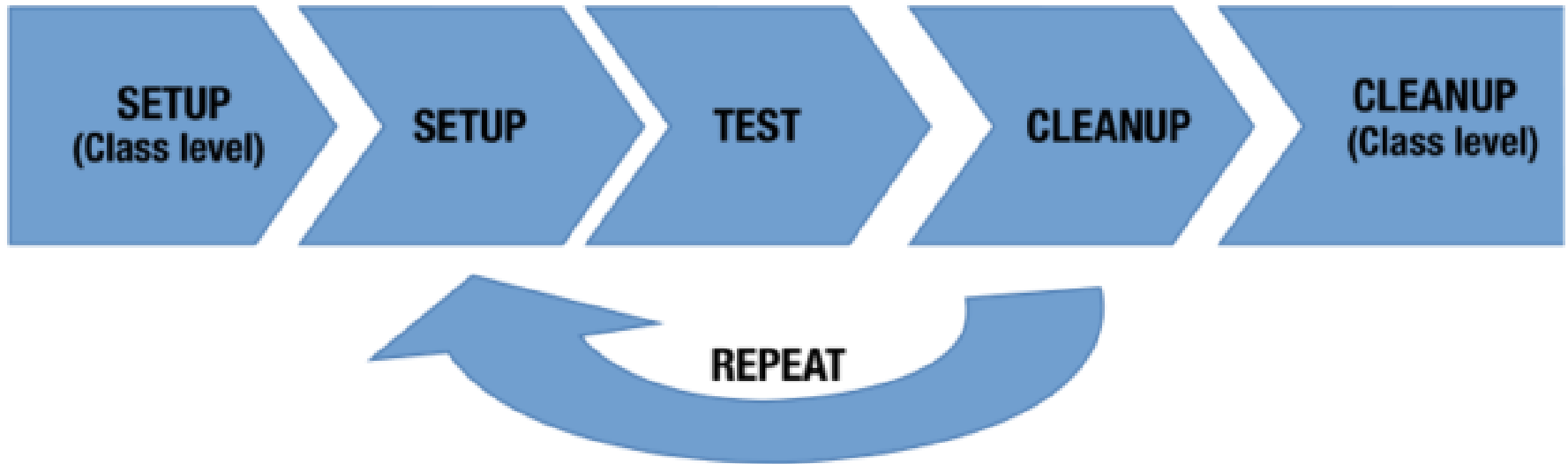


Abbildung aus: **Gulati, Sharma: Java Unit Testing with JUnit 5**, apress, 2017

Bsp

Zu testende Klasse

```
package demopackage;

public class Calculator {

    private static int result;

    public void add(int n) {
        result = result + n;
    }

    public void subtract(int n) {
        result = result - 1;           //Bug : result = result - n
    }

    public void multiply(int n) {}     //Not implemented yet

    public void divide(int n) {
        result = result / n;
    }

    public void square(int n) {
        result = n * n;
    }

    public void clear() {               // Ergebnis löschen
        result = 0;
    }

    public void switchOn() {           // Bildschirm einschalten, Piepsen, oder was
        result = 0;                   // Taschenrechner halt so tun
    }
}
```

Bsp Testklasse

```
class LifecycleDemoTest {  
  
    private static Calculator calculator;  
  
    @BeforeAll  
    public static void switchOnCalculator() {  
        System.out.println("\tSwitch on calculator");  
        calculator = new Calculator();  
        calculator.switchOn();  
    }  
  
    @BeforeEach  
    public void clearCalculator() {  
        System.out.println("zu Beginn jeden Tests wird der Calculator zurueckgesetzt");  
        calculator.clear();  
    }  
}
```

```
    @AfterEach  
    void tearThis(){  
        System.out.println("@AfterEach executed");  
    }  
  
    @AfterAll  
    public static void switchOffCalculator() {  
        System.out.println("\tSwitch off calculator");  
        calculator.switchOff();  
        calculator = null;  
    }  
}
```

```
@Disabled("not ready yet")  
@Test  
public void test_multiply() {  
    System.out.println("test_multiply()");  
    calculator.add(10);  
    calculator.multiply(10);  
    assertEquals(calculator.getResult(), 100);  
}
```

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
-  [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Test Annotations

ANNOTATION	DESCRIPTION
<code>@BeforeEach</code>	The annotated method will be run before each test method in the test class.
<code>@AfterEach</code>	The annotated method will be run after each test method in the test class.
<code>@BeforeAll</code>	The annotated method will be run before all test methods in the test class. This method must be static.
<code>@AfterAll</code>	The annotated method will be run after all test methods in the test class. This method must be static.
<code>@Test</code>	It is used to mark a method as junit test
<code>@DisplayName</code>	Used to provide any custom display name for a test class or test method
<code>@Disable</code>	It is used to disable or ignore a test class or method from test suite.
<code>@Nested</code>	Used to create nested test classes
<code>@Tag</code>	Mark test methods or test classes with tags for test discovering and filtering
<code>@TestFactory</code>	Mark a method is a test factory for dynamic tests

Aus <https://howtodoinjava.com/junit-5-tutorial/#annotations>

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- ➔ ▪ [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Assertions

- assertEquals()
- assertNotEquals()
- assertEqualsArrayEquals()
- assertEqualsIterableEquals()
- assertEqualsLinesMatch()
- assertNotNull()
- assertNull()
- assertNotSame()
- assertSame()
- assertEqualsTimeout()
- assertEqualsTimeoutPreemptively()
- assertTrue()
- assertFalse()
- assertEqualsThrows()
- fail()

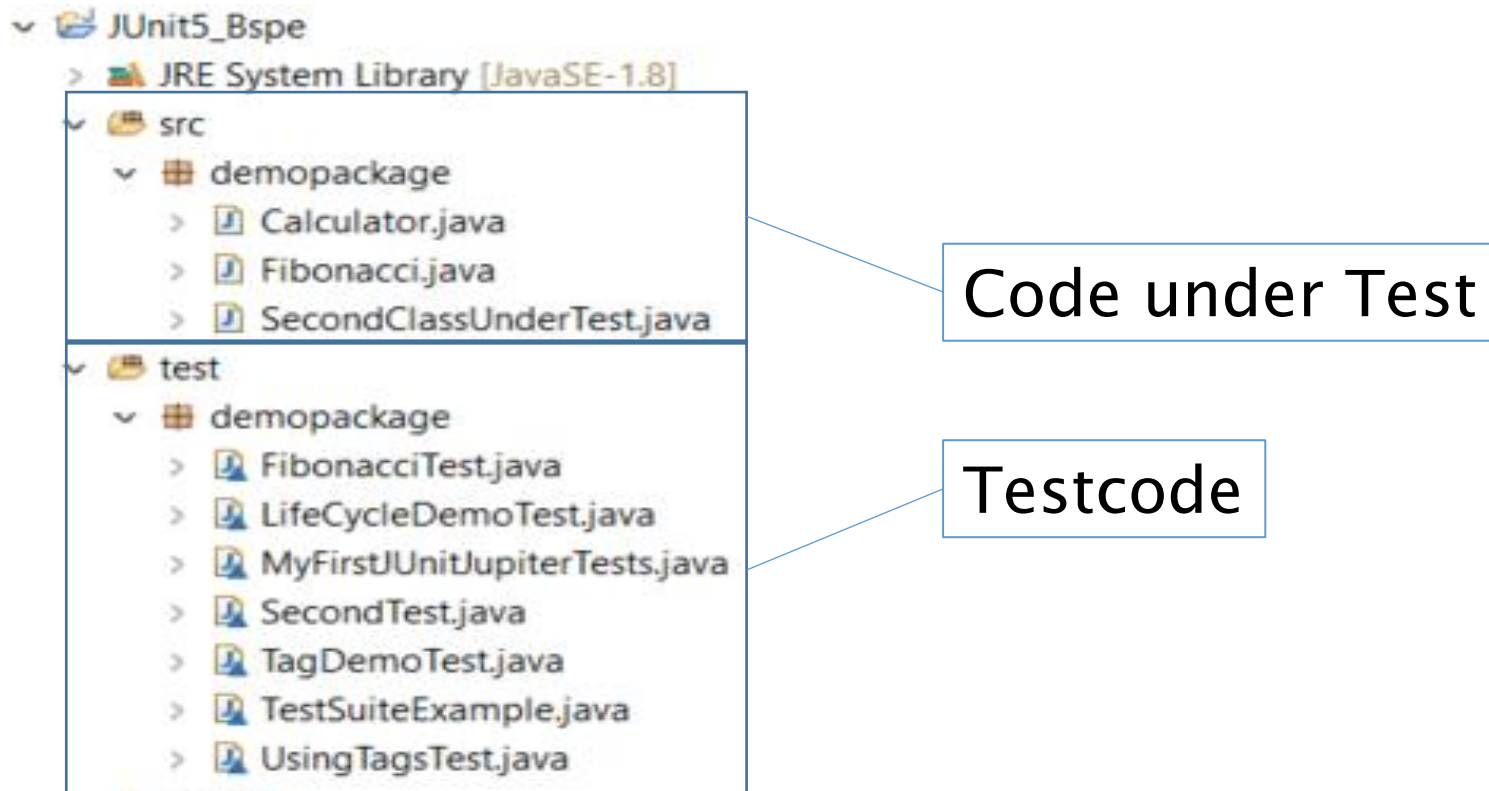
Beispiele unter <https://howtodoinjava.com/junit5/junit-5-assertions-examples/>

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

JUnit Konventionen

- Trennung Code unter Test von Testcode



JUnit Namenskonventionen

Klasse, die eine andere Klasse testet, hat den Namen der zu testenden Klasse + „Test“

Bsp:

- zu testen: *Car.java*
- Testklasse: *CarTest.java*

JUnit Namenskonventionen

Test Methoden:

Konvention (nicht zwingend seit JUnit 4): Beginne den Namen mit „*test*“

Benennung (Konvention nach *M. Tamm: JUnit Profiwissen*):

- *test()*
- *test_<Name der getesteten Methode>()*
- *test_that_<erwartetes Verhalten>()*
- *test_that_<erwartetes Verhalten>_when_<Vorbedingung>()*

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Parametrisierter Test → Instanzen für das Kreuzprodukt aus Test Daten Elementen und Testmethoden.

Bsp: Klasse zu testen: Fibonacci.java

```
package demo;

public class Fibonacci {
    public static int compute(int n) {
        int result = 0;

        if (n <= 1) {
            result = n;
        } else {
            result = compute(n - 1) + compute(n - 2);
        }

        return result;
    }
}
```

Testen verschiedener Kombinationen

Bisher:

```
package demopackage;

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

class FibonacciTest {

    @Test
    void testCompute() {
        assertEquals(0, Fibonacci.compute(0));
        assertEquals(1, Fibonacci.compute(1));
        assertEquals(1, Fibonacci.compute(2));
        assertEquals(2, Fibonacci.compute(3));
        assertEquals(3, Fibonacci.compute(4));
        assertEquals(5, Fibonacci.compute(5));
        assertEquals(8, Fibonacci.compute(6));
    }
}
```

Testen verschiedener Kombinationen - besser

```
@ParameterizedTest
@CsvSource({
    "0,0",
    "1,1",
    "1,2",
    "2,3",
    "3,4",
    "5,5",
    "8,6"
})
void testWithCsvSource(int result, int input) {
    System.out.println("Test mit " + result + " , " + input);
    assertEquals(result, Fibonacci
        .compute(input));
}
```

Parametrisierter Test

Daten, hier als csv Values

Verwendung der Daten im
Test

Bsp aus <https://www.infoworld.com/article/3537563/junit-5-tutorial-part-1-unit-testing-with-junit-5-mockito-and-hamcrest.html?page=2>

```
@ParameterizedTest
@MethodSource("generateEvenNumbers")
void testIsEvenRange(int number) {
    Assertions.assertTrue(MathTools.isEven(number));
}

static IntStream generateEvenNumbers() {
    return IntStream.iterate(0, i -> i + 2).limit(500);
}
```

Typen von Sources für Parametrisierte Tests

- **ValueSource:** Specifies a hardcoded list of integers or Strings.
- **MethodSource:** Invokes a static method that generates a stream or collection of items.
- **EnumSource:** Specifies an enum, whose values will be passed to the test method. It allows you to iterate over all enum values or include or exclude specific enum values.
- **CsvSource:** Specifies a comma-separated list of values.
- **CsvFileSource:** Specifies a path to a comma-separated value file with test data.
- **ArgumentSource:** Allows you to specify an argument provider that generates a stream of arguments to be passed to your test method.
- **NullSource:** Passes null to your test method if you are working with Strings, collections, or arrays. You can include this annotation with other annotations, such as the ValueSource, to write code that tests a collection of values and null.
- **EmptySource:** Includes an empty value if you are working with Strings, collections, or arrays.
- **NullAndEmptySource:** Includes both null and an empty value if you are working with Strings, collections, or arrays.

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Tags](#)

Tags

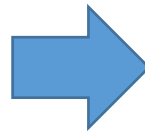
Mit der JUnit5 @Tag Annotation können Sie Filter für Testpläne setzen.

Tags definieren

```
@Test
@Tag("Tagdemo")
void testCompute_1() {
    assertEquals(1, Fibonacci.compute(1));
}

@Test
@Tag("Tagdemo")
void testCompute_2() {
    assertEquals(1, Fibonacci.compute(2));
}

@Test
@Tag("Tagdemo")
void testCompute_3() {
    assertEquals(3, Fibonacci.compute(4));
}
```



Tags nutzen

```
package demopackage;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.IncludeTags;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("demopackage")
@IncludeTags("Tagdemo")
class UsingTagsTest {
}
```

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)



- [JUnit Tags](#)
- [JUnit3 vs JUnit4 vs JUnit5](#)

JUnit 3

- Sehr weit verbreitet: JUnit4
- Aktuellste Version: JUnit5
- In vielen Projekten noch verwendet: JUnit3

```
package demopackage;

import junit.framework.TestCase;

public class CalculatorTest extends TestCase {

    Calculator calculator;

    protected void setUp() throws Exception {
        System.out.println("\tSwitch on calculator");
        calculator = new Calculator();
        calculator.switchOn();
        System.out
            .println("zu beginn jeden Tests wird der Calculator zuruecgesetzt");
        calculator.clear();
    }

    protected void tearDown() throws Exception {
        System.out.println("\tSwitch off calculator");
        calculator.switchOff();
        calculator = null;
    }

    public void testAdd() {
        calculator.add(1);
        calculator.add(1);
        assertEquals(calculator.getResult(), 2);
    }

}
```

JUnit3 Testcase – Unterschiede zu JUnit4

```
package demopackage;
```

```
import junit.framework.TestCase;
```

```
public class CalculatorTest extends TestCase {
```

```
    Calculator calculator;
```

```
    protected void setUp() throws Exception {  
        System.out.println("\tSwitch on calculator");  
        calculator = new Calculator();  
        calculator.switchOn();  
        System.out.  
            .println("zu Beginn jeden Tests wird der Calculator zuruecgesetzt");  
        calculator.clear();  
    }
```

```
    protected void tearDown() throws Exception {  
        System.out.println("\tSwitch off calculator");  
        calculator.switchOff();  
        calculator = null;  
    }
```

```
    public void testAdd() {  
        calculator.add(1);  
        calculator.add(1);  
        assertEquals(calculator.getResult(), 2);  
    }
```

```
}
```

Kein `import static org.junit.Assert.*;`

Ableitung von `TestCase`

Methode `setUp()` und `tearDown()`
Statt Annotierte Methoden

Name muss mit „test“ beginnen

TestSuites in JUnit3

```
package meinpackage;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests extends TestSuite
{
    public static Test suite()
    {
        TestSuite mySuite = new TestSuite( "Meine Test-Suite" );
        mySuite.addTestSuite( meinpackage.MeineKlasseTest.class );
        // ... weitere Testklassen hinzufügen
        return mySuite;
    }
}
```


JUnit4 vs Junit5 - Annotations

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	<code>@Test</code>	<code>@Test</code>
Execute before all test methods in the current class	<code>@BeforeClass</code>	<code>@BeforeAll</code>
Execute after all test methods in the current class	<code>@AfterClass</code>	<code>@AfterAll</code>
Execute before each test method	<code>@Before</code>	<code>@BeforeEach</code>
Execute after each test method	<code>@After</code>	<code>@AfterEach</code>
Disable a test method / class	<code>@Ignore</code>	<code>@Disabled</code>
Test factory for dynamic tests	NA	<code>@TestFactory</code>
Nested tests	NA	<code>@Nested</code>
Tagging and filtering	<code>@Category</code>	<code>@Tag</code>
Register custom extensions	NA	<code>@ExtendWith</code>

Aus <https://howtodoinjava.com/junit5/junit-5-vs-junit-4/>

JUnit4 vs JUnit5

Architektur

JUnit4: einzelnes jar file

JUnit5: Junit Platform, Junit Jupiter, Junit Vintage

JDK

JUnit4: Java 5 oder höher

JUnit5: Java 8 oder höher

Tagging/Filtering

JUnit4: @Category

JUnit5: @Tag

Suites

JUnit4: @RunWith und @Suite

JUnit5: @RunWith, @SelectPackages, @SelectClasses

Erweiterungen

JUnit4: Rules und Runners

JUnit5: Extensions