

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

Links und Literatur zu JUnit

- **Unfassende Darstellung:**
M. Tamm: JUnit Profiwissen, 1. Auflage 3013, dpunkt.verlag
- **JUnit Website incl Tutorials:**
<http://junit.org/junit4/>
- **JUnit Artikel**
<http://www.vogella.com/tutorials/JUnit/article.html>
- **JUnit Tutorial**
<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 richtige Version von JUnit!

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

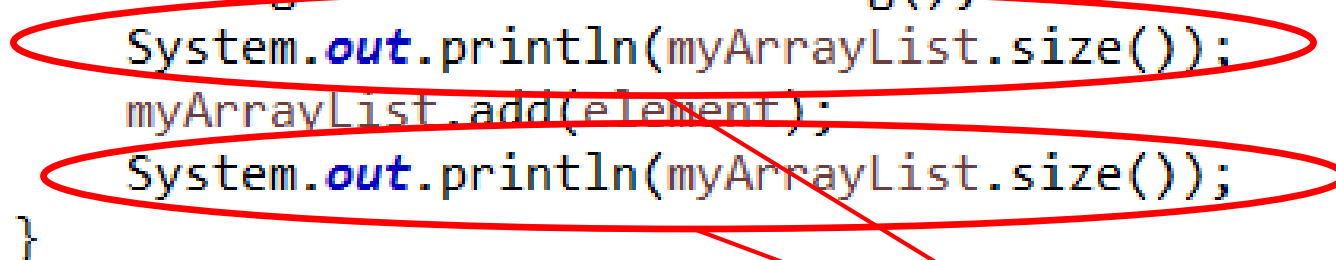
Testautomatisierung

- Motivation / Idee
- Ziele von JUnit
- Features von JUnit
- Einfaches Beispiel
- TestSuites
- TestFixturees
- JUnit Annotations
- JUnit Assertions
- Namenskonventionen
- Parametrisierbare Tests
- JUnit Rules
- JUnit Categories
- JUnit Theories
- JUnit Custom Runners
- JUnit3 vs JUnit4

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 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");  
    }  
}
```

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 Testautomatisierung

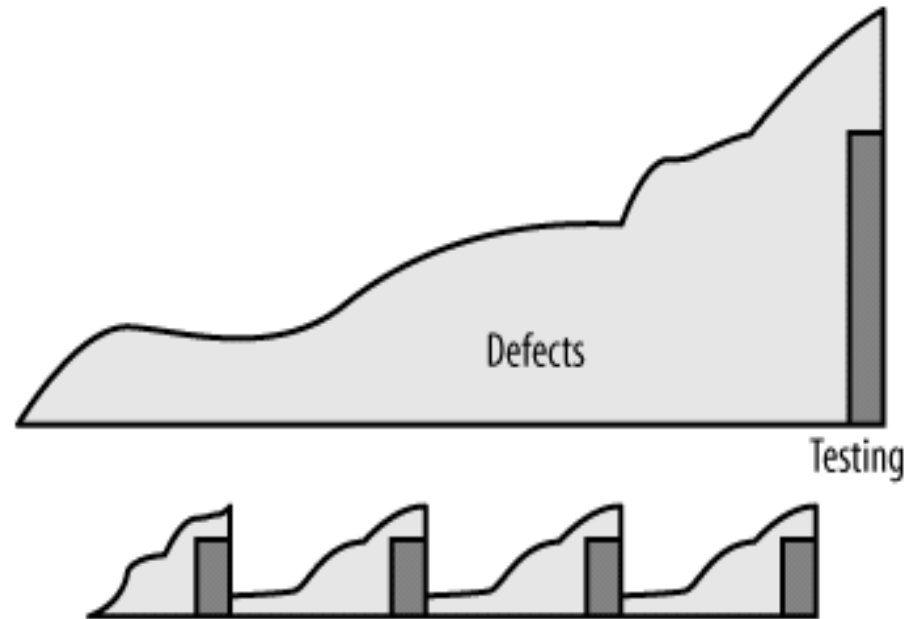


Figure 1-1. Frequent testing leaves fewer defects at the end

Quelle: JUnit Pocket Guide, Kent Beck, Kindle Edition

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

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](#)
- [TestFixtures](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

Features von JUnit

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

Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixturees](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
 - [JUnit Categories](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4](#)

Einfaches Beispiel in Eclipse

Quelle: <https://www.javacodegeeks.com/2014/11/JUnit-tutorial-unit-testing.html>

Java Klasse:

```
package demopackage;

import java.util.Arrays;

public class FirstDayAtSchool {

    public String[] prepareMyBag() {
        String[] schoolbag = { "Books", "Notebooks", "Pens" };
        System.out.println("My school bag contains: "
            + Arrays.toString(schoolbag));
        return schoolbag;
    }

    public String[] addPencils() {
        String[] schoolbag = { "Books", "Notebooks", "Pens", "Pencils" };
        System.out.println("Now my school bag contains: "
            + Arrays.toString(schoolbag));
        return schoolbag;
    }

}
```

Einfaches Beispiel in Eclipse

JUnit Testklasse

```
package demopackage;

import static org.junit.Assert.*;

public class FirstDayAtSchoolTest {

    FirstDayAtSchool school = new FirstDayAtSchool();
    String[] bag1 = { "Books", "Notebooks", "Pens" };
    String[] bag2 = { "Books", "Notebooks", "Pens", "Pencils" };

    @Test
    public void testPrepareMyBag() {
        System.out.println("Inside testPrepareMyBag()");
        assertEquals(bag1, school.prepareMyBag());
    }

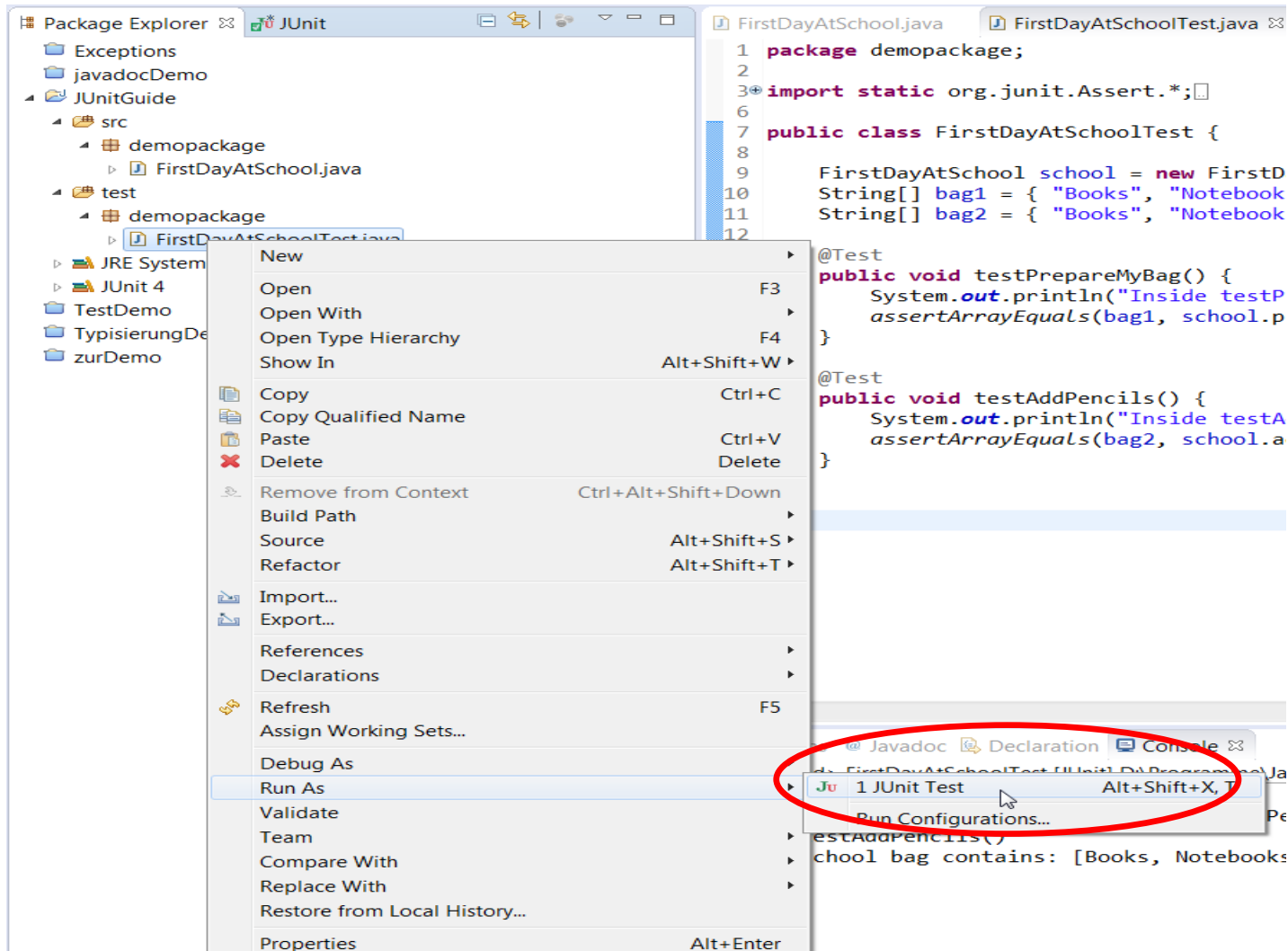
    @Test
    public void testAddPencils() {
        System.out.println("Inside testAddPencils()");
        assertEquals(bag2, school.addPencils());
    }
}
```

Beliebiger Name, keine
spezielle Superklasse (JUnit 4)

Das sind Testmethoden

Assert Methode zur
Überprüfung

Test laufen lassen



Package Explorer

- Exceptions
- javadocDemo
- JUnitGuide
 - src
 - demopackage
 - FirstDayAtSchool.java
 - test
 - demopackage
 - FirstDayAtSchoolTest.java
 - JRE System
 - JUnit 4
 - TestDemo
 - TypisierungDe
 - zurDemo

FirstDayAtSchoolTest.java

```

1 package demopackage;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class FirstDayAtSchoolTest {
8
9     FirstDayAtSchool school = new FirstD
10    String[] bag1 = { "Books", "Notebook
11    String[] bag2 = { "Books", "Notebook
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

JUnit 4

Run As

1 JUnit Test

Run Configurations...

Console

FirstDayAtSchoolTest [JUnit4] (D:\Programme\Ja...)

testAddPencils()

school bag contains: [Books, Notebooks]

Ergebnis

The screenshot shows the JUnit test runner interface. At the top, there are tabs for 'Package Explorer' and 'JUnit'. Below the tabs is a toolbar with various icons. The main area displays the test results for 'demopackage.FirstDayAtSchoolTest'. The status is 'Finished after 0,016 seconds'. The summary shows 'Runs: 2/2', 'Errors: 0', and 'Failures: 0'. A green progress bar indicates that all tests passed. Below the progress bar, the test name 'demopackage.FirstDayAtSchoolTest' is listed with the runner 'JUnit 4' and the duration '(0,000 s)'.

Package Explorer JUnit

Finished after 0,016 seconds

Runs: 2/2 Errors: 0 Failures: 0

demopackage.FirstDayAtSchoolTest [Runner: JUnit 4] (0,000 s)

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixturees](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
 - [JUnit Categories](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4](#)

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

Ausführen von mehreren Tests aus verschiedenen Testklassen.

Bsp:

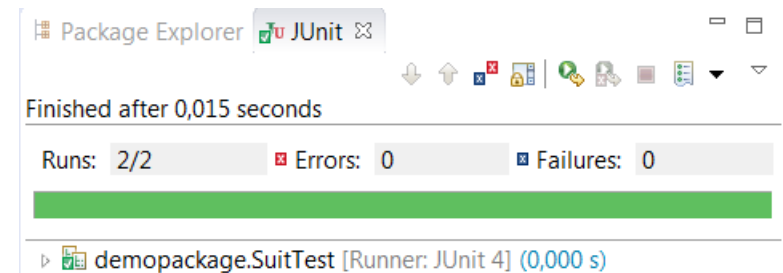
Anderer **Runner** → Gemeinsam laufen lassen!

```
package demopackage;
```

```
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)  
@Suite.SuiteClasses({ PrepareMyBagTest.class, AddPencilsTest.class })  
public class SuitTest {  
}
```

Was soll
gemeinsam laufen?



Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixturees](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
 - [JUnit Categories](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4](#)

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 in JUnit 4 durch Annotationen:

@Before
public void method()

This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).

@After
public void method()

This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.

Testautomatisierung

Realisierung von Testfixtures in JUnit 4 durch Annotationen:

@BeforeClass

public static void method()

This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.

@AfterClass

public static void method()

This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.

Wichtig: JUnit4 garantiert,

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

Testautomatisierung

Beispiel:
Klasse zu
testen:
Calculator.java

```
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 squareRoot(int n) {
        for (; ; ) ;                 //Bug : Endlosschleife
    }

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

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

    public void switchOff() { }        // Ausschalten

    public int getResult() {
        return result;
    }
}
```


Testklasse CalculatorTest

```
package demopackage;

import static org.junit.Assert.*;

import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class CalculatorTest {

    private static Calculator calculator;

    @BeforeClass
    public static void switchOnCalculator() {
        System.out.println("\tSwitch on calculator");
        calculator = new Calculator();
        calculator.switchOn();
    }

    @AfterClass
    public static void switchOffCalculator() {
        System.out.println("\tSwitch off calculator");
        calculator.switchOff();
        calculator = null;
    }

    @Before
    public void clearCalculator() {
        System.out.println("zu Beginn jeden Tests wird der Calculator zurückgesetzt");
        calculator.clear();
    }

    @Test
    public void test_add() {
        calculator.add(8);
        calculator.divide(2);
        assertEquals(calculator.getResult(), 4);
    }

    @Test(expected = ArithmeticException.class)
    public void test_divideByZero() {
        calculator.divide(0);
    }

    @Test
    public void test_multiply() {
        calculator.multiply(10);
        assertEquals(calculator.getResult(), 100);
    }
}
```

Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixturees](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
 - [JUnit Categories](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4](#)

JUnit annotations

Annotation	Description
@Test public void method()	The @Test annotation identifies a method as a test method.
@Test (expected = Exception.class)	Fails if the method does not throw the named exception.
@Test(timeout=100)	Fails if the method takes longer than 100 milliseconds.
@Before public void method()	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After public void method()	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.
@AfterClass public static void method()	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.
@Ignore or @Ignore("Why disabled")	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit3 vs JUnit4](#)

Gängige Assert Methoden

Statement	Description
fail(message)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

Komplett: <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

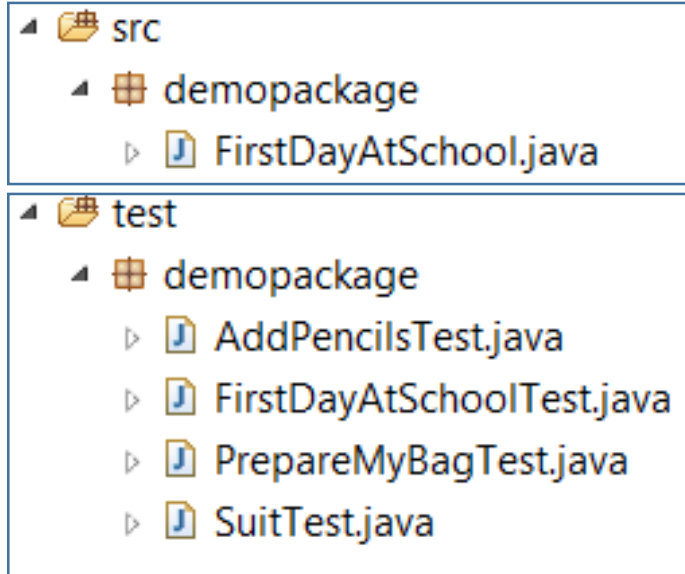
Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixtures](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
 - [JUnit Categories](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4](#)

JUnit Konventionen

- Trennung Code under Test von Testcode

JUnitGuide



Code under Test

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 Rules](#)
 - [JUnit Categories](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4](#)

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 demo;

import static org.junit.Assert.*;

public class AlterFibonacciTest {

    @Test
    public 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

Testklasse: FibonacciTest

Jede Instanz des Tests
wird mit dem
Konstruktor und den
Werten aus der
`@Parameters` Methode
konstruiert.

```
import static org.junit.Assert.assertEquals;

import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class FibonacciTest {

    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
        });
    }

    private int fInput;
    private int fExpected;

    public FibonacciTest(int input, int expected) {
        fInput = input;
        fExpected = expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

Parametrisierte Tests howto:

- Testklasse mit `@RunWith(Parameterized.class)` annotieren.
- Implementieren einer public static Methode, annotiert mit `@Parameters`, die eine Collection von Objekt Arrays als Test Datensätze zurückliefert.
- Public Konstruktor, der als Argument die Elemente eines Object Arrays aus der statischen `@Parameters` Methode annimmt.
- Instanzvariablen für Elemente der Testdaten.
- Testfälle, die diese Instanzvariablen als Quelle der Testdaten verwenden.

Möglichkeit, auf den Konstruktor zu verzichten:
Annotation der Membervariablen mit
`@Parameter(...)`

```
@Parameters
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] { { 0, 0 }, { 1, 1 }, { 2, 1 },
        { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 } });
}

@Parameter(0)
public int fInput;

@Parameter(1)
public int fExpected;
```

Testautomatisierung

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


- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

Motivation:

- @Before und @After wird verwendet, um Tests vorzubereiten und nachher aufzuräumen.
- Bsp: temporäres Verzeichnis anlegen, nachher löschen.
 - ➔ das braucht man öfter als nur in einer Testklasse

➔ Was tun?

Lösungsmöglichkeiten:

1. Redundante @Before und @After Methoden in verschiedenen Klassen 
2. Basisklasse für alle Testklassen mit gemeinsam zu nutzenden @Before und @After Methoden 
3. Hierarchie von Testklassen mit Kombinationen von @Before und @After Methoden 

oder

JUnit Rules



JUnit Rules

- JUnit Rules erlauben es, querschnittliche Vorbereitungs- und Aufräumarbeiten nur einmal zu implementieren und zu pflegen.
- Es gibt bereits vorgefertigte JUnit Rules, die mit JUnit ausgeliefert werden.

JUnit Rules Beispiel

```
public class AssetManagerTest {  
  
    @Rule  
    public TemporaryFolder tempFolder = new TemporaryFolder();  
  
    @Test  
    public void countsAssets() throws IOException {  
        AssetManager am = new AssetManager();  
        File assets = tempFolder.newFolder("assets");  
        am.createAssets(assets, 3);  
        assertEquals(3, am.countAssets());  
    }  
}
```

Zu TemporaryFolder siehe

<http://junit.org/junit4/javadoc/latest/org/junit/rules/TemporaryFolder.html>

Um eine Testklasse mit einem als MethodRule Klasse programmierten Testaspekt auszustatten:

- Öffentliche Membervariable zur Testklasse.
- Diese mit `@Rule` annotieren
- Der Variablen eine neue Instanz der gewünschten MethodRule Klasse zuweisen.

Diese Rules gibt es bereits:

- TemporaryFolder Rule
- ExternalResource Rules
- ErrorCollector Rule
- Verifier Rule
- TestWatchman/TestWatcher Rules
- TestName Rule
- Timeout Rule
- ExpectedException Rules

*Teilweise als
Superklassen für selbst
zu implementierende
Klassen*

Siehe <https://github.com/junit-team/junit4/wiki/Rules>

Entsprechungen

@Before	@Rule
@BeforeClass	@ClassRule

RuleChain

RuleChain erlaubt es, Rules hintereinander auszuführen.

```
public static class UseRuleChain {  
    @Rule  
    public TestRule chain = RuleChain  
        .outerRule(new LoggingRule("outer rule"))  
        .around(new LoggingRule("middle rule"))  
        .around(new LoggingRule("inner rule"));  
  
    @Test  
    public void example() {  
        assertTrue(true);  
    }  
}
```



```
starting outer rule  
starting middle rule  
starting inner rule  
finished inner rule  
finished middle rule  
finished outer rule
```


Eigene Rules schreiben:

- Klasse anlegen, die das Interface TestRule implementiert:

<http://junit.org/junit4/javadoc/latest/org/junit/rules/TestRule.html>

```
public interface TestRule {  
    /**  
     * Modifies the method-running {@link Statement} to implement this  
     * test-running rule.  
     *  
     * @param base The {@link Statement} to be modified  
     * @param description A {@link Description} of the test implemented in {@code base}  
     * @return a new statement, which may be the same as {@code base},  
     *         a wrapper around {@code base}, or a completely new Statement.  
     */  
    Statement apply(Statement base, Description description);  
}
```

Custom Rules

```
public interface TestRule {  
    /**  
     * Modifies the method-running {@link Statement} to implement this  
     * test-running rule.  
     *  
     * @param base The {@link Statement} to be modified  
     * @param description A {@link Description} of the test implemented in {@code base}  
     * @return a new statement, which may be the same as {@code base},  
     *         a wrapper around {@code base}, or a completely new Statement.  
     */  
    Statement apply(Statement base, Description description);  
}
```

Repräsentiert den Aufruf
der eigentlichen @Test
Methode (incl aller before
und after Methoden)

Weitere Beschreibung des Tests

```
public abstract class Statement {  
    /**  
     * Run the action, throwing a {@code Throwable} if anything goes wrong.  
     */  
    public abstract void evaluate() throws Throwable;  
}
```

Hier wird der eigentliche Test ausgeführt

Custom Rules – Bsp Verifier

```
public abstract class Verifier implements TestRule {
    public Statement apply(final Statement base, Description description) {
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
                base.evaluate();
                verify();
            }
        };
    }

    /**
     * Override this to add verification logic. Overrides should throw an
     * exception to indicate that verification failed.
     */
    protected void verify() throws Throwable {
    }
}
```

Muster: Erzeugen und Rückgabe eines anonymen inneren Statements, das in der evaluate Methode denjenigen Code enthält, der den Test ausmacht und an passender Stelle base.evaluate aufruft.

Testautomatisierung

- [Motivation / Idee](#)
 - [Ziele von JUnit](#)
 - [Features von JUnit](#)
 - [Einfaches Beispiel](#)
 - [TestSuites](#)
 - [TestFixturees](#)
 - [JUnit Annotations](#)
 - [JUnit Assertions](#)
 - [Namenskonventionen](#)
 - [Parametrisierbare Tests](#)
- [JUnit Rules](#)
 - [JUnit Categories](#)
 - [JUnit Theories](#)
 - [JUnit Custom Runners](#)
 - [JUnit3 vs JUnit4](#)

JUnit Categories

JUnit Categories dienen dazu, Testmethoden zu gruppieren und nur Tests einer bestimmten Kategorie laufen zu lassen. Dies geschieht zusätzlich zu den TestSuites.

siehe

- ➔ M. Tamm: JUnit Profiwissen, 1. Auflage 2013, dpunkt.verlag
- ➔ <https://github.com/junit-team/junit4/wiki/Categories>
- ➔ <https://community.oracle.com/blogs/johnsmart/2010/04/25/grouping-tests-using-junit-categories-0>
- ➔ <https://examples.javacodegeeks.com/core-java/junit/junit-categories-example/>

JUnit Categories - Bsp

1. Interface als Marker definieren

```
package demopackage;  
  
public interface MySpecialTests {  
  
}
```

2. Testmethoden annotieren

```
@Test  
@Category(MySpecialTests.class)  
public void test_add() {  
    calculator.add(1);  
    calculator.add(1);  
    assertEquals(calculator.getResult(), 2);  
}
```

3. Laufen lassen

```
@RunWith(Categories.class)  
@SuiteClasses({CalculatorTest.class})  
@IncludeCategory(MySpecialTests.class)  
public class KategorieSuite {  
  
}
```

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

Idee:

Allgemeingültige Aussage als Test formulieren
und durch eine Reihe von Testwerten
überprüfen.

Paper dazu:

<http://web.archive.org/web/20110608210825/http://shareandenjoy.saff.net/tdd-specifications.pdf>

Titel:

The Practice of Theories: Adding “For-all” Statements to “There-Exists” Tests

Bsp zu JUnit Theories

```
@RunWith(Theories.class)
public class BigIntegerTest {

    @Theory
    public void multiply_follows_commutative(BigInteger i1, BigInteger i2) {
        assertEquals(i1.multiply(i2), i2.multiply(i1));
    }

    @Theory
    public void divide_is_inverse_of_multiply(BigInteger i1, BigInteger i2) {
        assumeThat(i2.toString(), is(not("0")));
        assertEquals(i1.multiply(i2).divide(i2), i1);
    }

    @DataPoints
    public static BigInteger[] TEST_DATA = new BigInteger[]{
        new BigInteger("-1"),
        new BigInteger("0"),
        new BigInteger("42"),
        new BigInteger("12121234343434231213"),
    };
}
```

Was ist an dem Beispiel besonders?

- Die Testmethoden haben mindestens einen Eingabeparameter.
- Es werden allgemeingültige Aussagen als Testmethoden formuliert.
- Der eigentliche Test läuft dann mit einem Set an Parametern durch, die man als `@DataPoints` annotiert.

- Die Testmethode und die Testdaten sind voneinander getrennt.
- Eine mit `@DataPoints` oder mit `@DataPoint` annotierte Variable oder statische Methode liefert ein Array oder einen Wert als Eingabewerte für einen bestimmten Datentyp.
- Der Theories Runner sammelt alle Beispielwerte ein unmittelbar bevor er die Theories Methode ausführt. Also nach allen `@BeforeClass` und `@Before` Methoden.

Testautomatisierung

- [Motivation / Idee](#)
- [Ziele von JUnit](#)
- [Features von JUnit](#)
- [Einfaches Beispiel](#)
- [TestSuites](#)
- [TestFixturees](#)
- [JUnit Annotations](#)
- [JUnit Assertions](#)
- [Namenskonventionen](#)
- [Parametrisierbare Tests](#)
- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

Bereits angesprochene JUnitRunner:

- *BlockJUnit4ClassRunner*: Default Runner
- *Suite*: Standard Runner um TEstSuites laufen zu lassen
- *Parameterized*: Runner für parametrisierte Tests
- *Categories*: Standard Runner für subsets von Tests, die entsprechend getagged sind.
- *Theories*: Runner für Testtheorien

Eigene Test Runner

Test Runner sind für die Ausführung der Tests zuständig.

Um sie zu verwenden, ist die Annotation `@RunWith()` nötig.

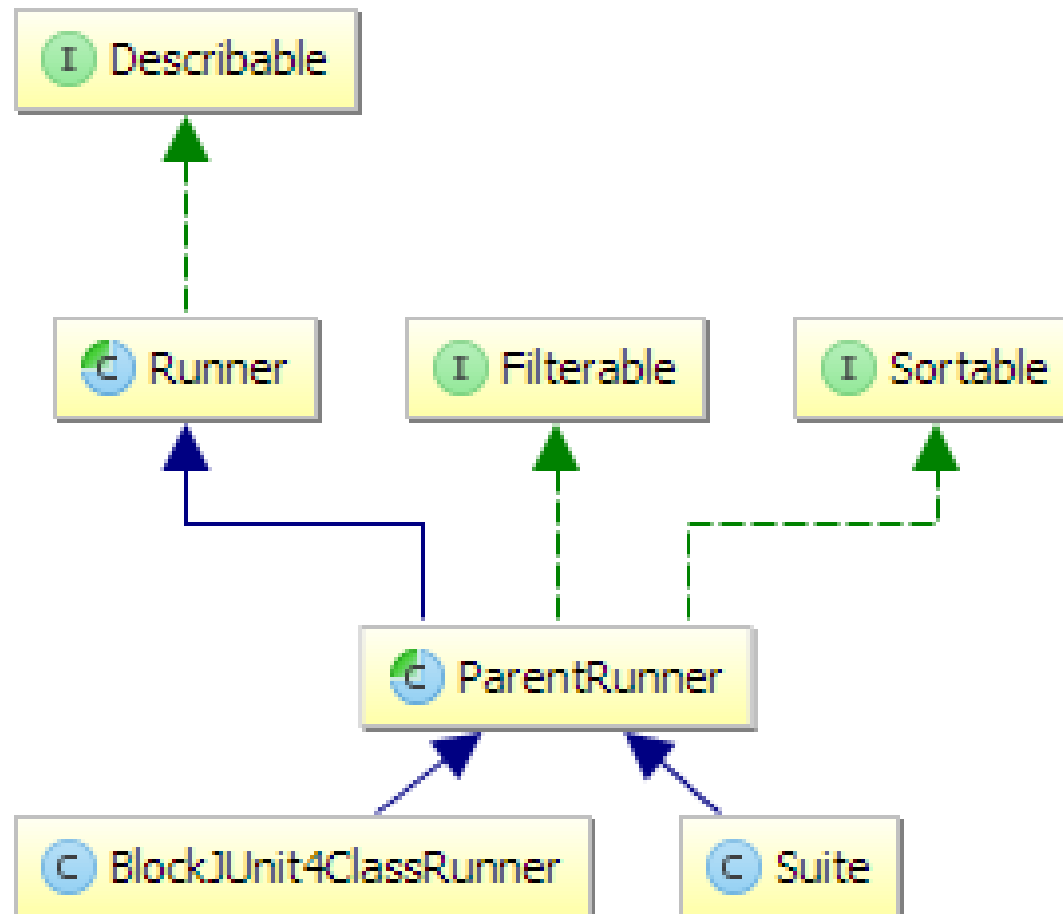
Testrunners Verantwortung:

- Testklassen Instanziierung
- Test Ausführung
- Reporting der Test Resultate

Test Runner

- Ein Test Case kann den Runner selbst angeben:
`@RunWith` Annotation
- Normalerweise genügen die bereits vorhandenen Runner. Wenn man selbst einen schreiben will, siehe
 - Michael Tamm: JUnit Profiwissen, dpunkt.verlag
 - <http://www.mscharhag.com/java/understanding-junits-runner-architecture>

Runners Hierarchie



Testautomatisierung

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

- [JUnit Rules](#)
- [JUnit Categories](#)
- [JUnit Theories](#)
- [JUnit Custom Runners](#)
- [JUnit3 vs JUnit4](#)

JUnit 3 vs JUnit 4

- Aktuell: JUnit4
- In Arbeit: 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;
    }
}
```