JUNIT & MOCKITO

# TESTING

# TESTGETRIEBENE ENTWICKLUNG

- Test vor den eigentlichen Program
- Test müsse nicht vom selben Entwickler sein
- Es dürfen Tests fehlschlagen

# ANNOTATIONS

- @Test
- @Before/@After
- @BeforeClass/@AfterClass
- @Ignor / @Ignor(„Why disabled“)
- @Test (expected = Exception.class)
- @Test(timeout =100)

# ASSERT STATEMENTS

- fail(message)
- assertTrue(message, boolean)
- assertFalse(message, boolean)
- assertEquals(message, expected, actual, tolerance)
- assertNull(message, object)
- assertNotNull(message, object)
- assertSame(message, expected, actual)
- assertNotSame(message, expected, actual)

# PARAMETERIZED TEST

## Test mit mehreren verschiedenen Daten ausführen

```
@RunWith(Parameterized.class)
public class ParameterizedTestFields {

    // fields used together with @Parameter must be public
    @Parameter(0)
    public int m1;
    @Parameter(1)
    public int m2;
    @Parameter(2)
    public int result;

    // creates the test data
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.asList(data);
    }

    @Test
    public void testMultiplyException() {
        MyClass tester = new MyClass();
        assertEquals("Result", result, tester.multiply(m1, m2));
    }
```

# RULES

```
@Rule
public ExpectedException exception = ExpectedException.none();
```

EigeneRegeln erstellen

```
public class MyCustomRule implements TestRule

@Override
public Statement apply(Statement base, Description description)
```

# TEST SUITES

```
@SuiteClasses({ MyClassTest.class, MySecondClassTest.class })
```

Mehrere TestKlassen miteinander ausführen

# JUNIT5 ANNOTATIONS

- @RepeatedTest(<Number>)
- @BeforeEach/@AfterEach & @BeforeAll/@AfterAll
- @Tag(„<TagName>")
- @Disabled
- @DisplayName(„<Name>")

# TEST SUITS

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.junit5.examples")
public class AllTests {}
```

```
@SelectClasses({AssertionTest.class, AssumptionTest.class,
ExceptionTest.class}
```

# GROUPED ASSERTIONS

```
assertAll("address name",
        () -> assertEquals("John", address.getFirstName()),
        () -> assertEquals("User", address.getLastName())
    );
```

# TIMEOUT

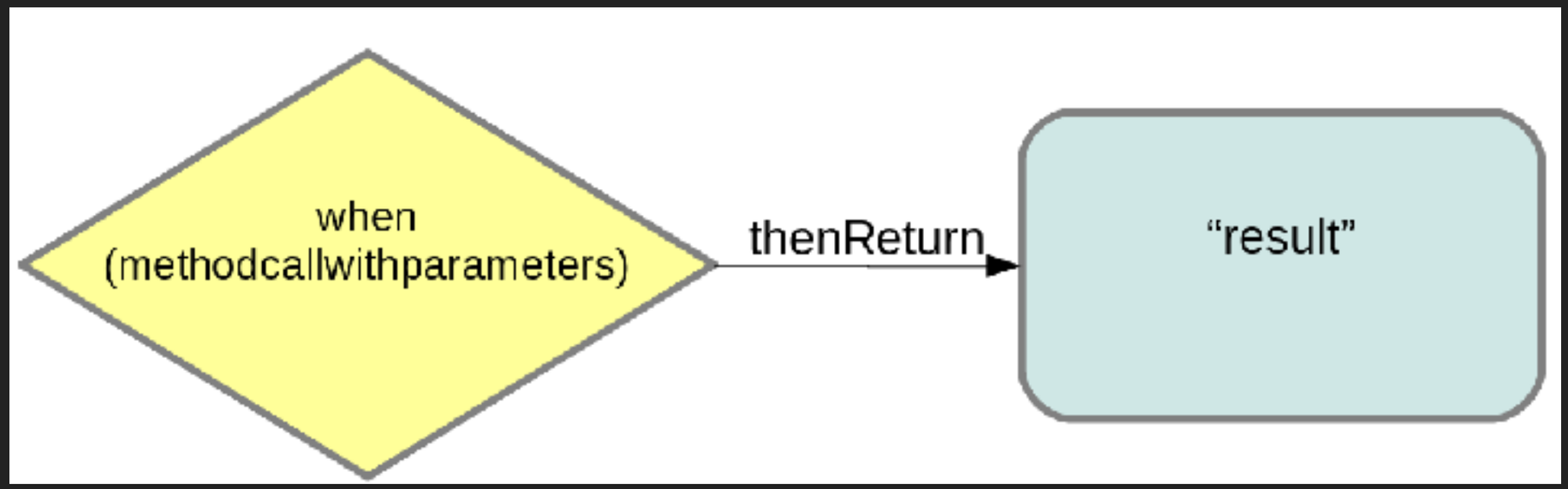
```
assertTimeout(ofMinutes(1), () -> service.doBackup());
```

# DYNAMICTEST

```
@TestFactory
    public Stream<DynamicTest> testMultiplyException() {
        MyClass tester = new MyClass();
        int[][] data = new int[][] { { 1, 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.stream(data).map(entry -> {
            int m1 = entry[0];
            int m2 = entry[1];
            int expected = entry[2];
            return dynamicTest(m1 + " * " + m2 + " = " + expected, () -> {
                assertEquals(expected, tester.multiply(m1, m2));
            });
        });
    }
```

# MOCKITO

- einzelne Objekte sollen isoliert getestet werden
- Schnittstellen müssen mit Platzhalter ersetzt werden



# THENRETURN

```
MyClass test = mock(MyClass.class);
when(test.getUniqueId()).thenReturn(42);
```

bestimmten Rückgabe Wert festlegen

# DORETURN

```
doReturn(“42”).when(spyProperties).getUniqueId()
```

Ist das selbe, nur das wenn in diesem Fall wenn getUniqueId eine Exception wirft trotzdem 42 geturnt wird

# SPY

```
List<String> list = new LinkedList<>();
List<String> spy = spy(list);
when(spy.get(0)).thenReturn("foo");
```

Damit kann man reale Objekte als mock verwenden

# VERIFY

```
verify(test).testing(ArgumentMatchers.eq(12));
```
Zum Überprüfen ob die Methode mit diesem Parameter aufgerufen wurde
```
verify(test, times(2)).getUniqueId();
```
Zum Überprüfen ob die Methode zweimal aufgerufen wurde