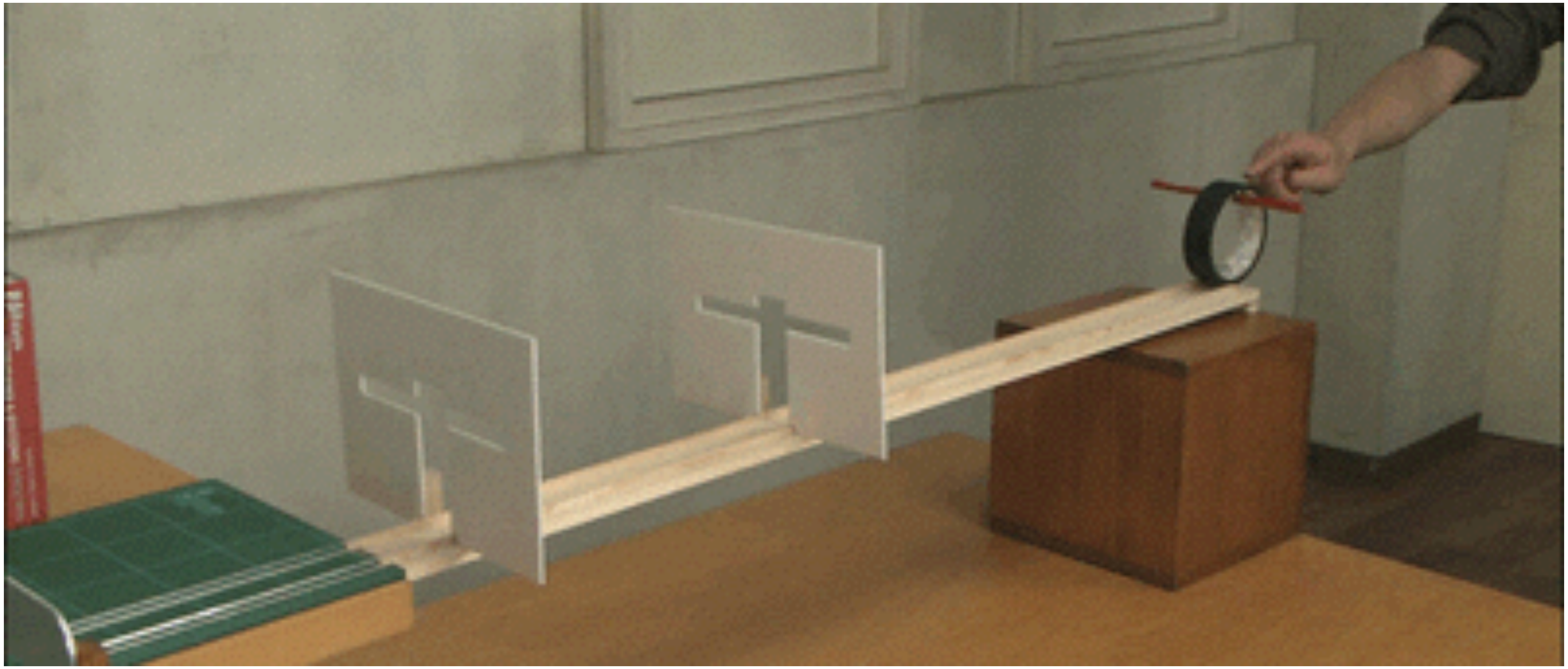
# Testen mit Mockito

http://www.vogella.com/tutorials/Mockito/article.html

# Warum testen?



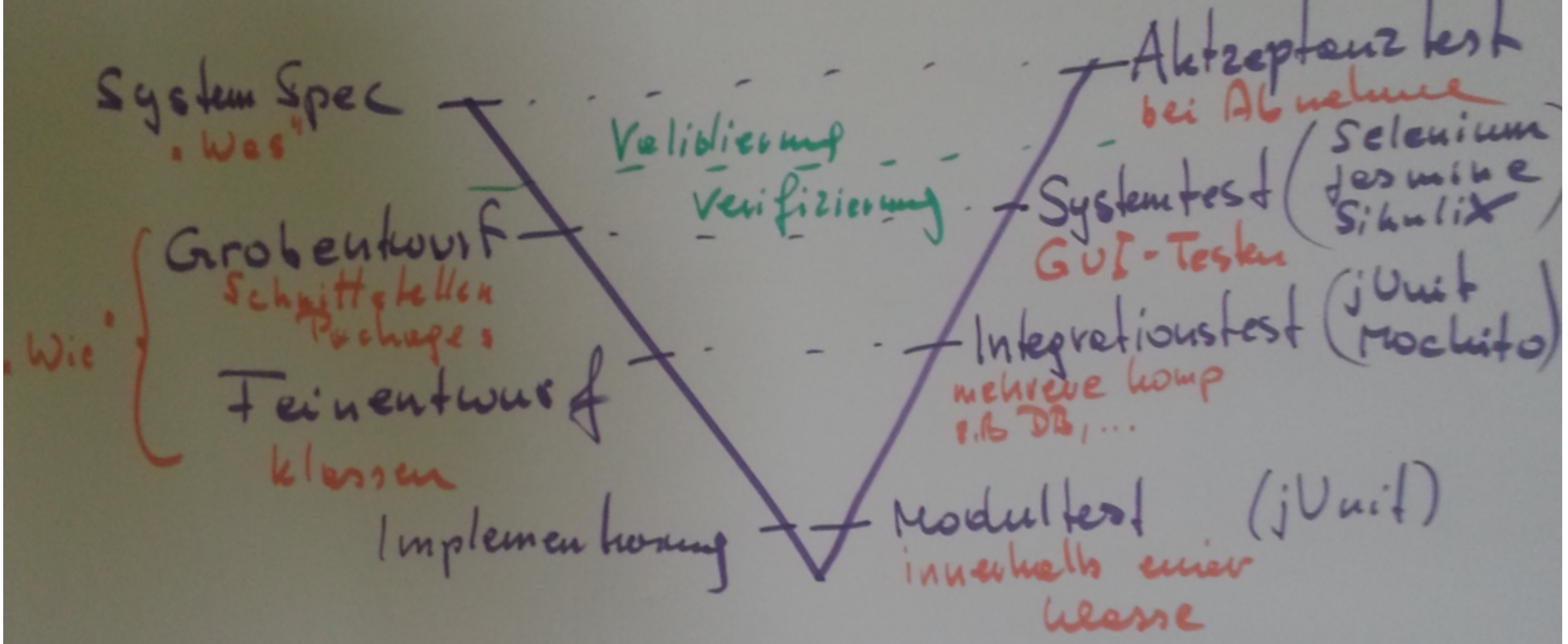
Würde dies funktionieren, ohne es vorab zu testen?

# Definition von Tests

- Ein Softwaretest prüft und bewertet Software auf Erfüllung der für ihren Einsatz definierten Anforderungen und misst ihre Qualität. Die gewonnenen Erkenntnisse werden zur Erkennung und Behebung von Softwarefehlern genutzt. Tests während der Softwareentwicklung dienen dazu, die Software möglichst fehlerfrei in Betrieb zu nehmen.

# Arten von Tests

- Unit-Test: Test der einzelnen Methoden einer Klasse
- Integrationstest: Tests mehrere Klassen / Module
- Systemtest: Test des Gesamtsystems (meist GUI-Test)
- Akzeptanztest / Abnahmetest: Test durch den Kunden, ob Produkt verwendbar
- Regressionstest: Nachweis, dass eine Änderung des zu testenden Systems früher durchgeführte Tests erneut besteht
- Feldtest: Test während des Einsatzes
- Lasttest
- Stresstest
- Smoke-Tests: (Zufallstest) nicht systematisch; nur Funktion wird getestet



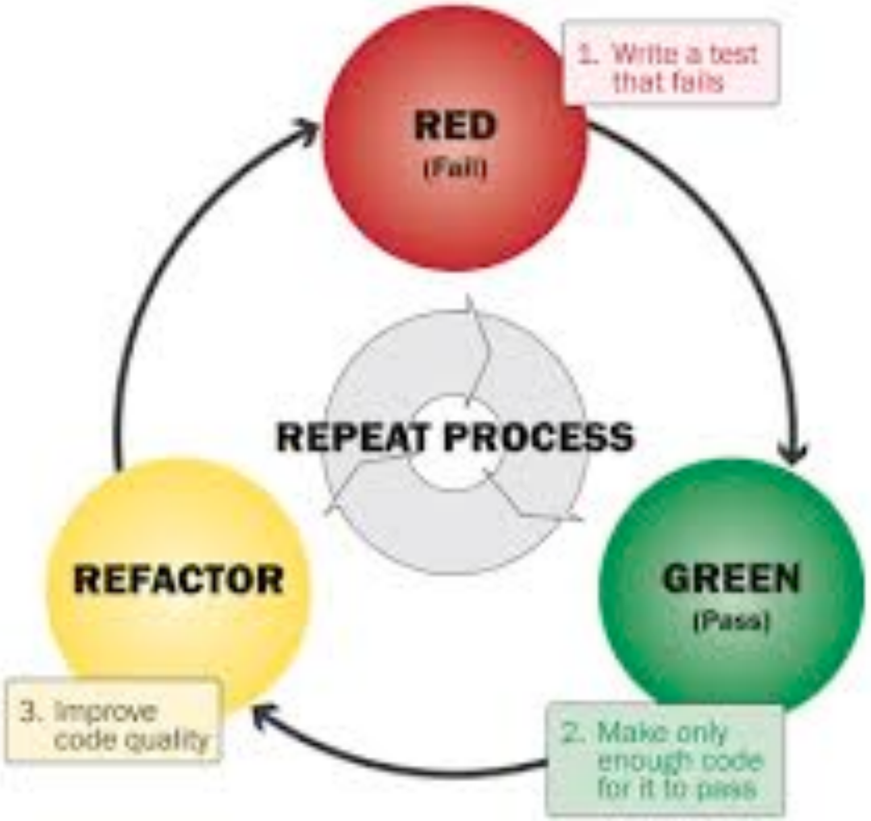
Validierung: Mache ich die richtigen Dinge?
WAS

Verifizierung: Mache ich die Dinge richtig
WIE

# Definitionen

- SUT … system under test
- CUT … class under test

# TDD - Test Driven Development



RED (Fail)

1. Write a test that fails

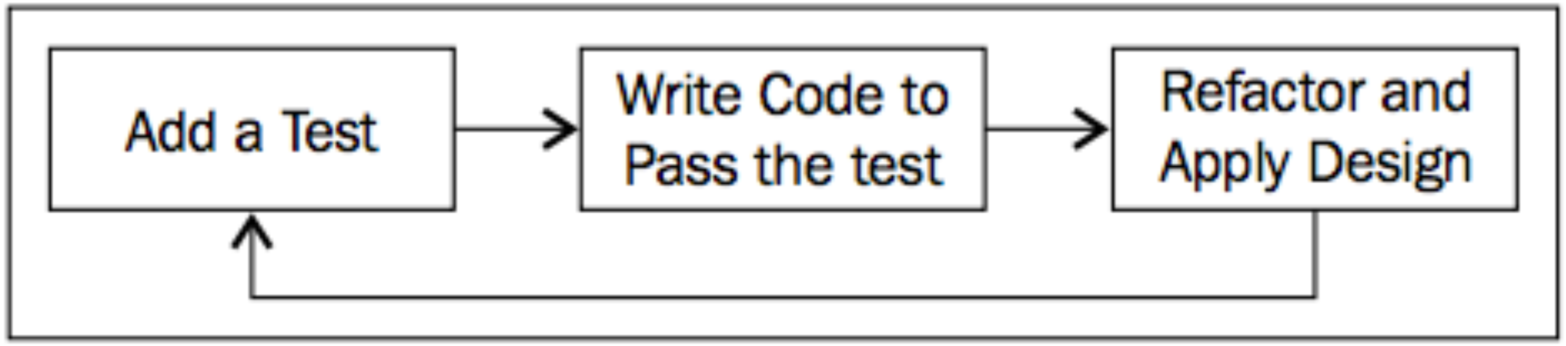
GREEN (Pass)

2. Make only enough code for it to pass

REFACTOR

3. Improve code quality

REPEAT PROCESS



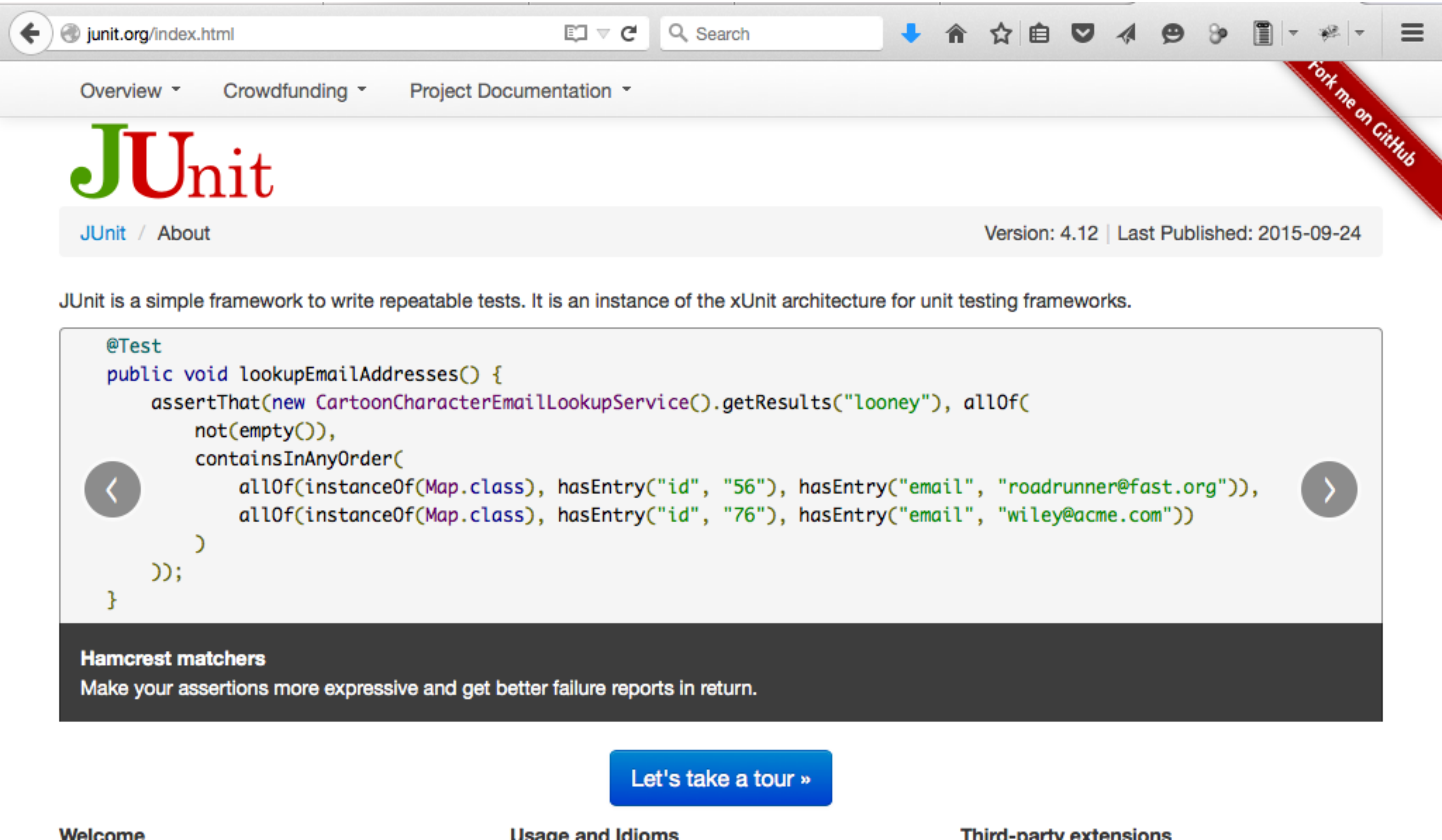
Add a Test

Write Code to Pass the test

Refactor and Apply Design

Test-Driven Development with Mockito, packt 2013

# JUnit.org



JUnit / About

Version: 4.12 | Last Published: 2015-09-24

JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.

```
@Test
public void lookupEmailAddresses() {
    assertThat(new CartoonCharacterEmailLookupService().getResults("looney"), allOf(
        not(empty()),
        containsInAnyOrder(
            allOf(instanceOf(Map.class), hasEntry("id", "56"), hasEntry("email", "roadrunner@fast.org")),
            allOf(instanceOf(Map.class), hasEntry("id", "76"), hasEntry("email", "wiley@acme.com"))
        )
    ));
}
```

**Hamcrest matchers**

Make your assertions more expressive and get better failure reports in return.

Welcome

Usage and Idioms

Third-party extensions

# Hamcrest

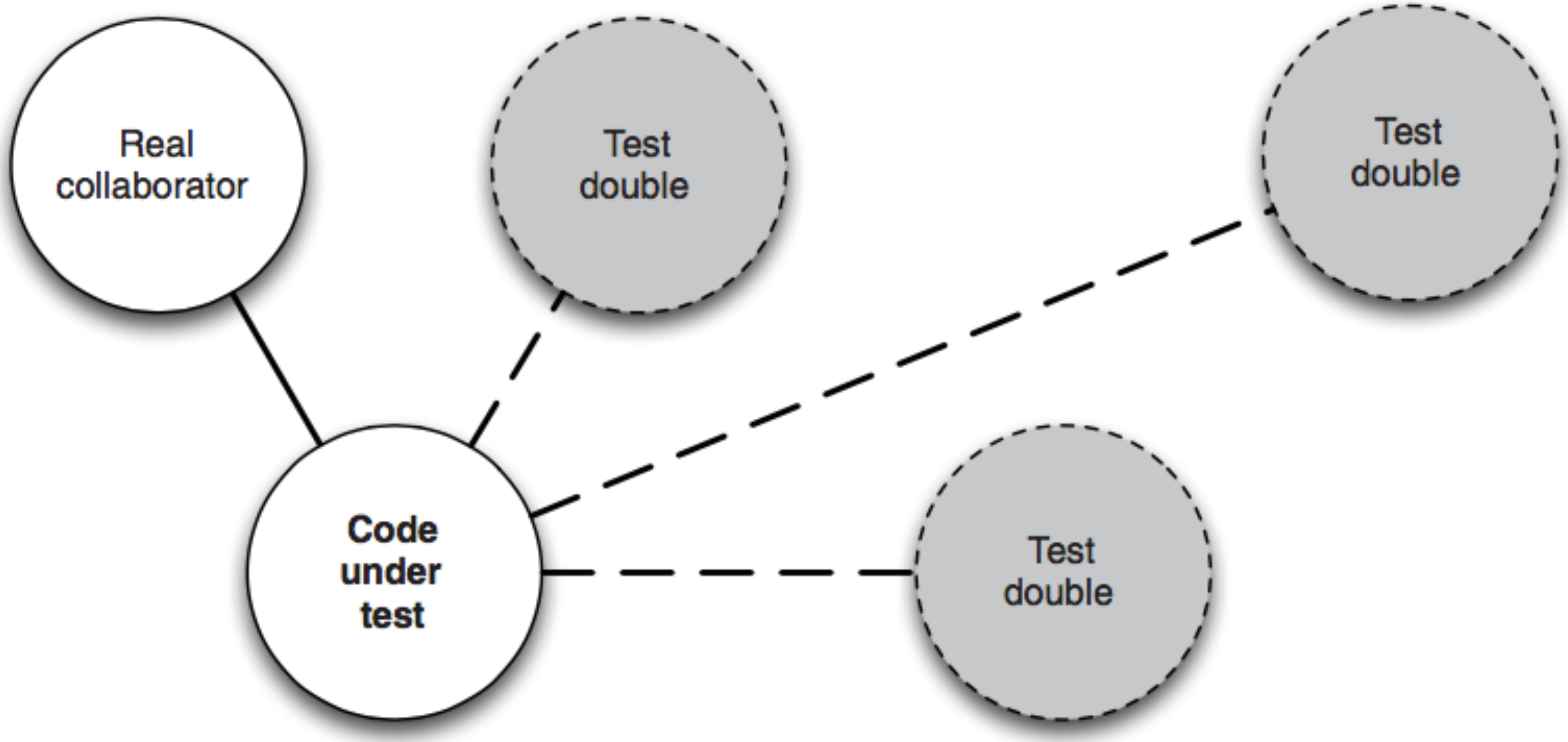
- Hamcrest is a library of matchers, which can be combined in to create flexible expressions of intent in tests. They've also been used for other purposes
- http://hamcrest.org/
- https://github.com/hamcrest/JavaHamcrest
- http://www.leveluplunch.com/java/examples/hamcrest-collection-matchers-junit-testing/
- http://grepcode.com/file/repo1.maven.org/maven2/org.hamcrest/hamcrest-library/1.3/org/hamcrest/Matchers.java

# Test Doubles

# Warum Test-Doubles?

- A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. The achievement of this desired goal is typical complicated by the fact that Java classes usually depend on other classes.
- To solve this, you can use test doubles.

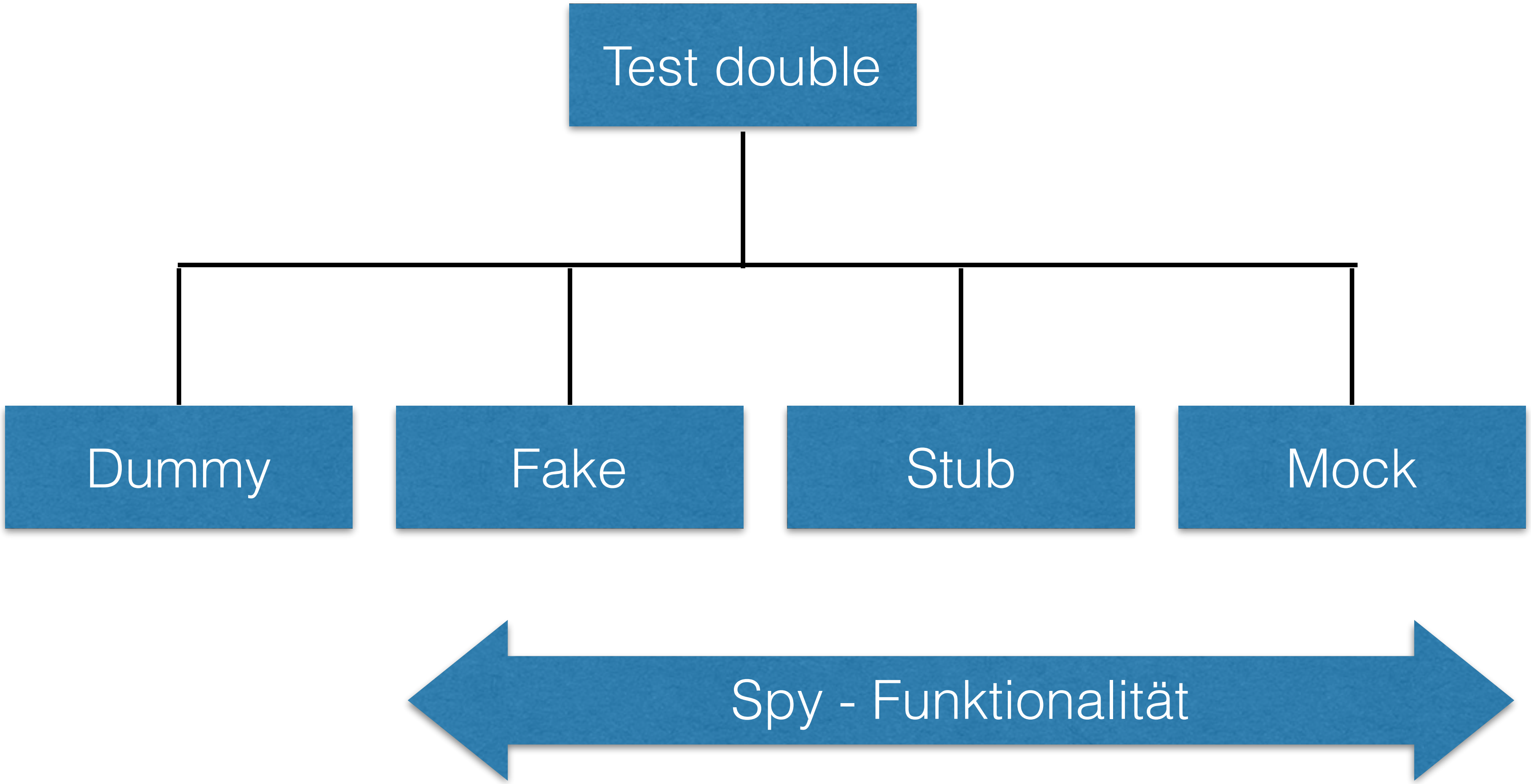
# Test Doubles



***The power of a test double***

Effective Unit Testing, Manning 2013, p.29

# Test doubles



http://www.vogella.com/tutorials/Mockito/article.html

# Dummy

- A dummy object is passed around but never used, i.e., its methods are never called. Such an object can for example be used to fill the parameter list of a method.

- Ein Objekt, das im Code weitergereicht, aber nicht verwendet wird. Werden eingesetzt um Parameter mit Werten zu befüllen [Wikipedia]

# Fake object

- Fake objects have working implementations, but are usually **simplified**, for example they use an in memory database and not a real database.

- Whereas a test stub can return hard-coded return values and each test may instantiate its own variation to return different values to simulate a different scenario, a fake object is more like an optimized, thinned-down version of the real thing that replicates the behavior of the real thing, but without the side effects and other consequences of using the real thing.

- Ein Objekt mit Implementierung. Die Implementierung ist dabei jedoch eingeschränkt, wodurch ein Einsatz in der Produktionsumgebung nicht möglich ist. Ein typisches Beispiel für ein Fake ist eine Datenbank, die Daten nur temporär im Speicher hält. [Wikipedia]

# Stub

- A stub class is an **partial implementation** for an interface or class with the purpose of using an instance of this stub class during testing. Stubs usually do responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls

- A test stub's (or just stub for short) purpose is to stand in for the real implementation with the simplest possible implementation. The most basic example of such an implementation would be an object with all of its methods being one-liners that return an appropriate default value.

- Ein Objekt, welches beim Aufruf einer bestimmten Methode unabhängig von der Eingabe die gleiche Ausgabe liefert. [Wikipedia]

# Mock object

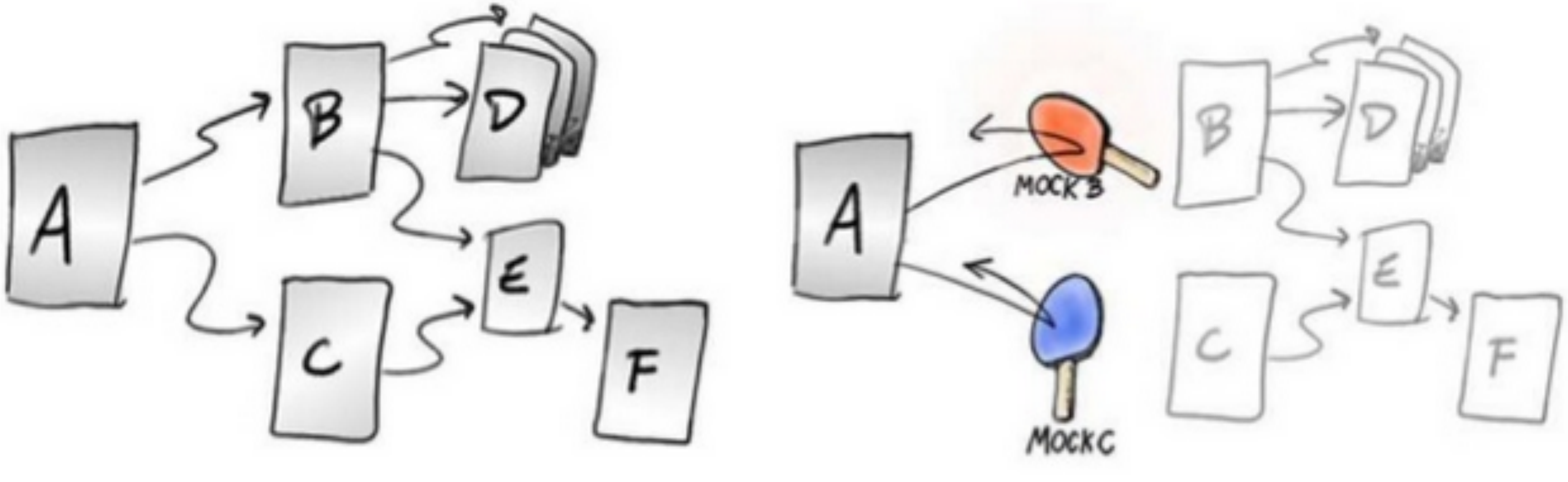
- A mock object is a dummy implementation for an interface or a class in which you define the output of certain method calls.

- Mock objects are typically configured. Mock objects typically require less code to configure and should therefore be preferred.

- It’s an object that’s configured to behave in a certain way under certain circumstances. For example, a mock object for the User-Repository interface might be told to return null when findById() is invoked with the parameter 123 and to return a given instance of User when findById() is invoked with 124. At this point we’re basically talking about stubbing certain method calls, considering their parameters.

- Ein Objekt, das bei vorher bestimmten Funktionsaufrufen mit bestimmten übergebenen Werten eine definierte Rückgabe liefert. [Wikipedia]

# Mocking

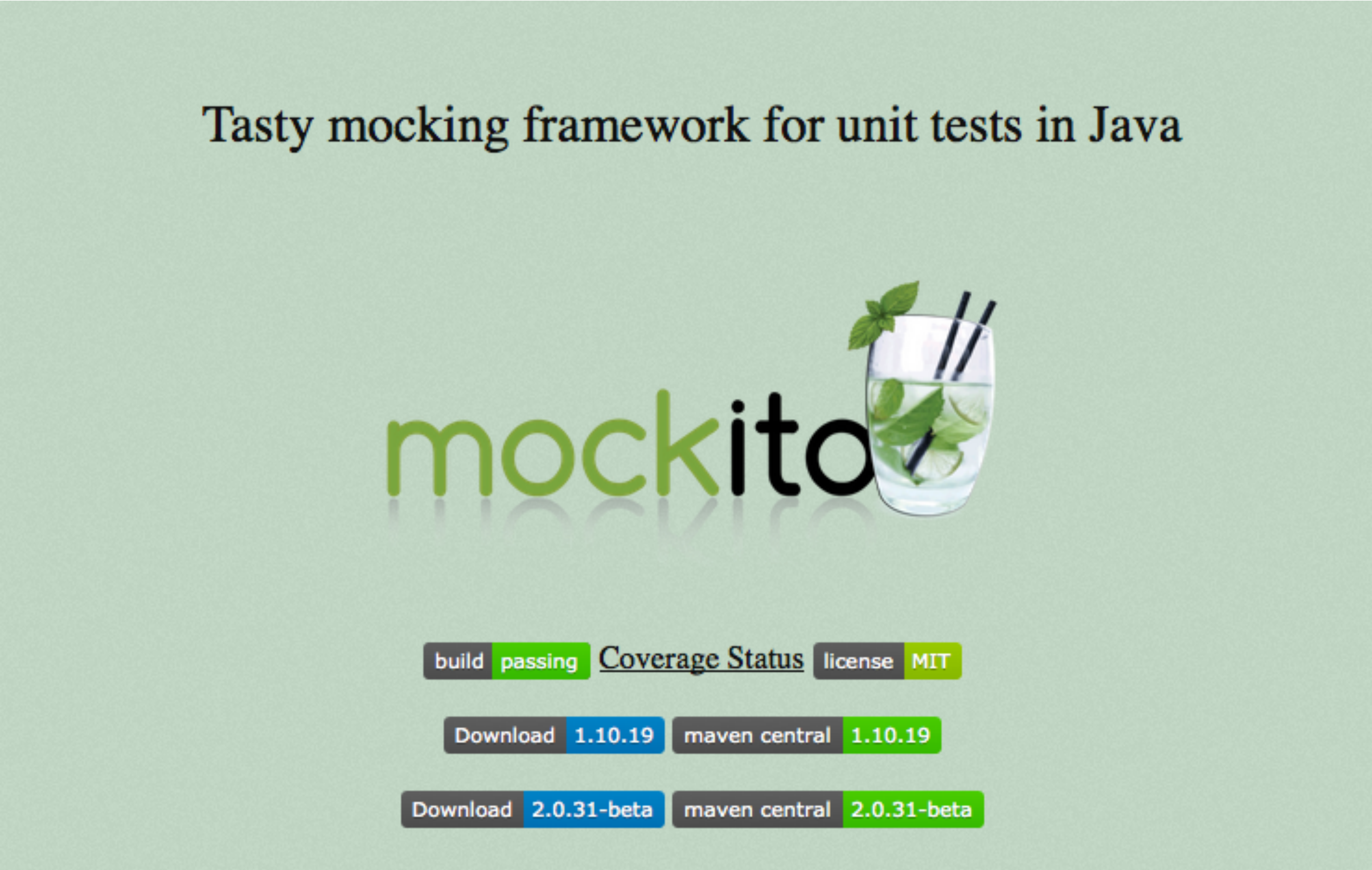


real objects

mocked objects

```
public class ATMTest {

  @Mock Dispenser failingDispenser;
  @Before
  public void setUp() throws Exception {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void transaction_is_rolledback_when_hardware_fails()
    throws DispenserFailed {
    Account myAccount = new Account(2000.00, "John");
    TransactionManager txMgr =
      TransactionManager.forAccount(myAccount);
    txMgr.registerMoneyDispenser(failingDispenser);

    doThrow(new
      DispenserFailed()).when(failingDispenser).
        dispense(isA(BigDecimal.class));
    txMgr.withdraw(500);
    assertTrue(2000.00 == myAccount.getRemianingBalance());
    verify(failingDispenser, new
      Times(1)).dispense(isA(BigDecimal.class));

  }

}
```

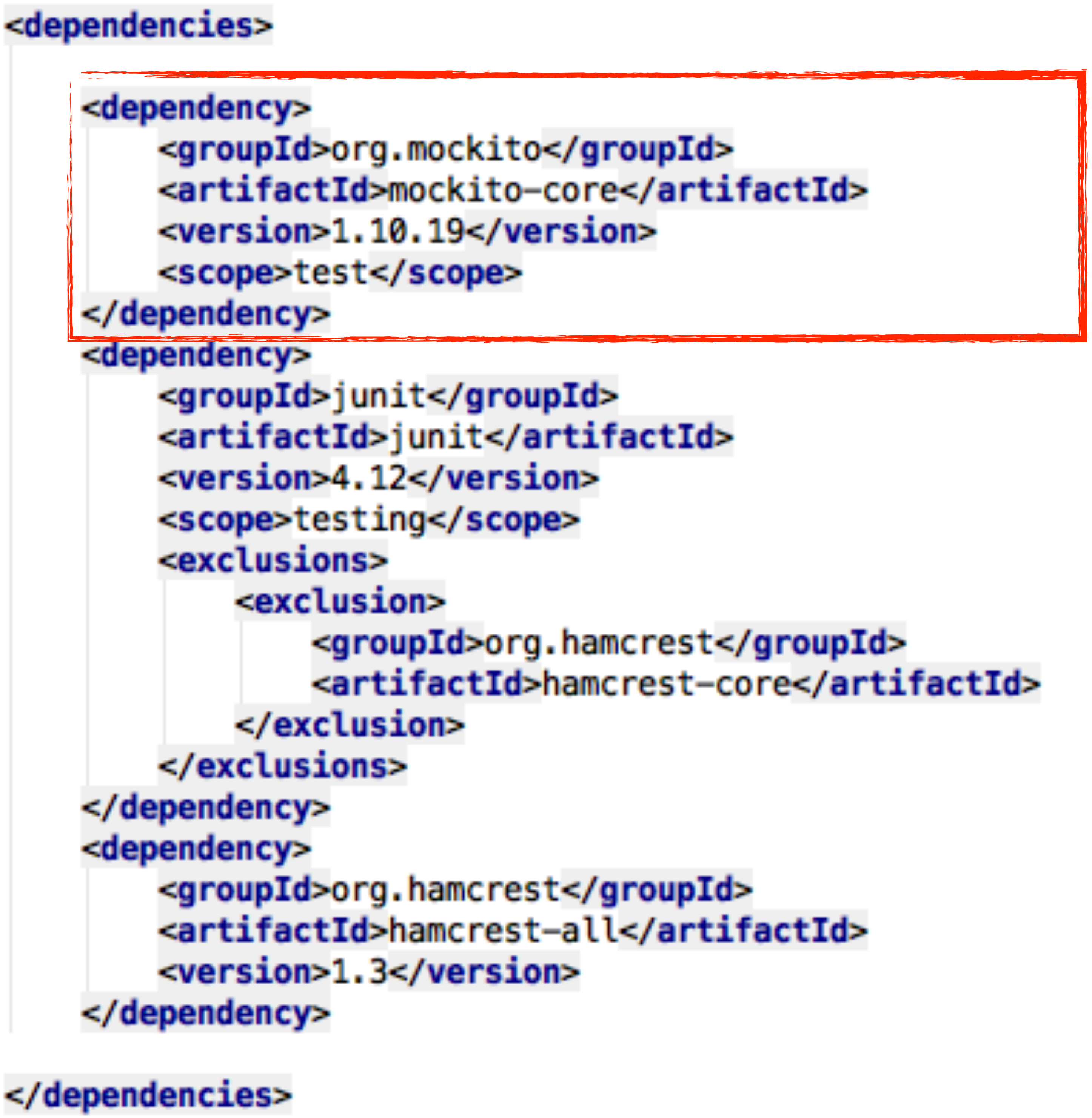
# mockito.org



Tasty mocking framework for unit tests in Java

# Mockito

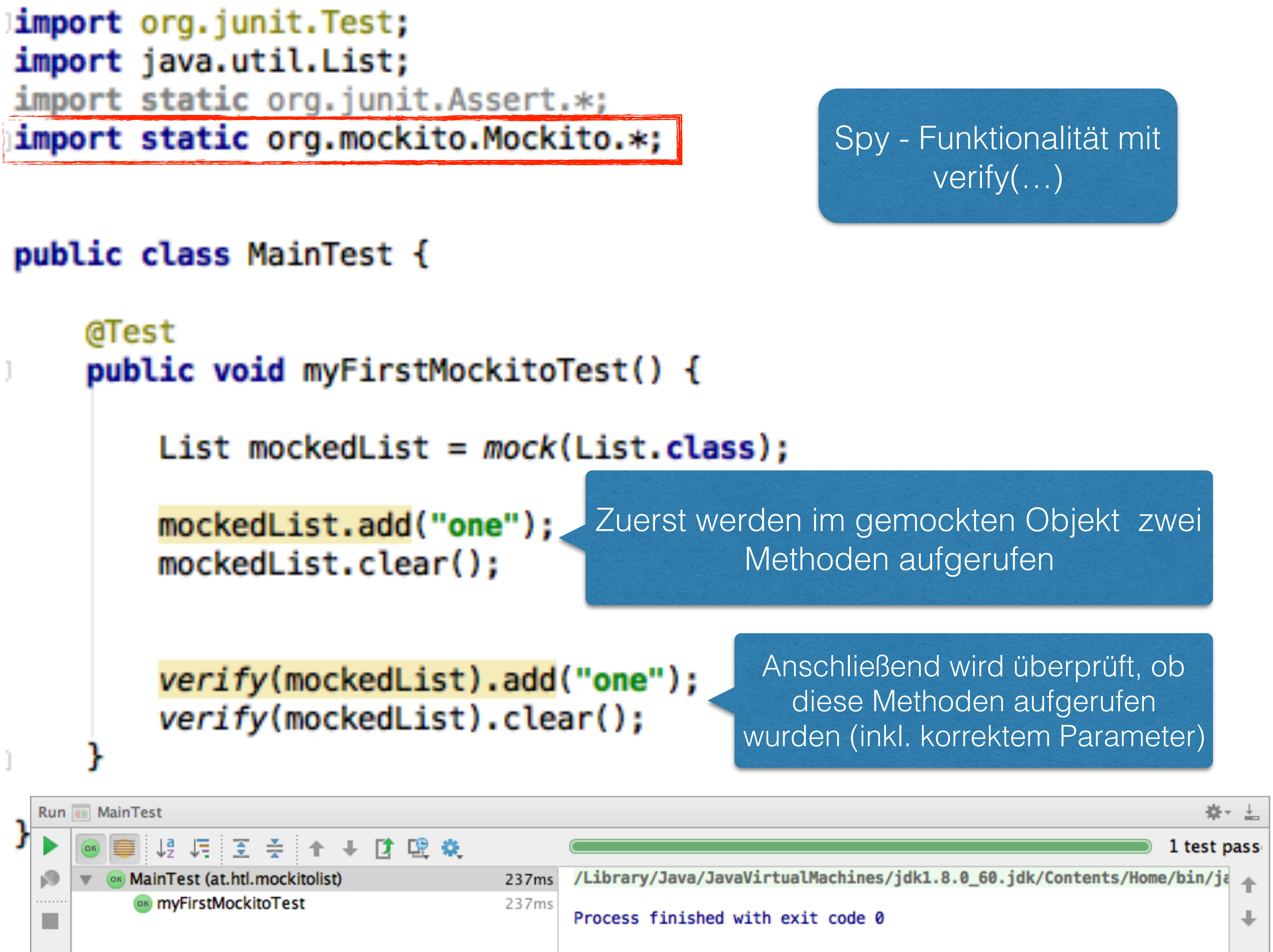
# Dependencies



```
<dependencies>

    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>1.10.19</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>testing</scope>
        <exclusions>
            <exclusion>
                <groupId>org.hamcrest</groupId>
                <artifactId>hamcrest-core</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-all</artifactId>
        <version>1.3</version>
    </dependency>

</dependencies>
```

# Stubbing Method Calls with Mockito

Spy - Funktionalität mit verify(…)



```
import org.junit.Test;
import java.util.List;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

public class MainTest {

    @Test
    public void myFirstMockitoTest() {

        List mockedList = mock(List.class);

        mockedList.add("one");
        mockedList.clear();


        verify(mockedList).add("one");
        verify(mockedList).clear();
    }
}
```

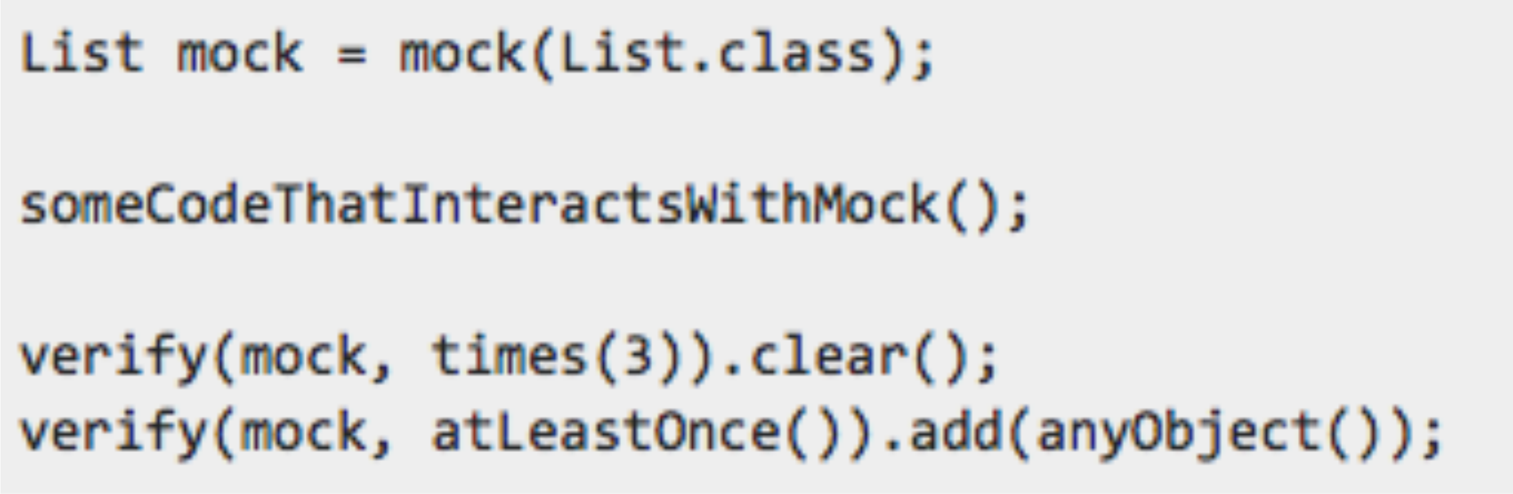
Zuerst werden im gemockten Objekt zwei Methoden aufgerufen

Anschließend wird überprüft, ob diese Methoden aufgerufen wurden (inkl. korrektem Parameter)

```
MainTest (at.htl.mockitolist)    237ms
    myFirstMockitoTest           237ms
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/bin/ja

Process finished with exit code 0
```

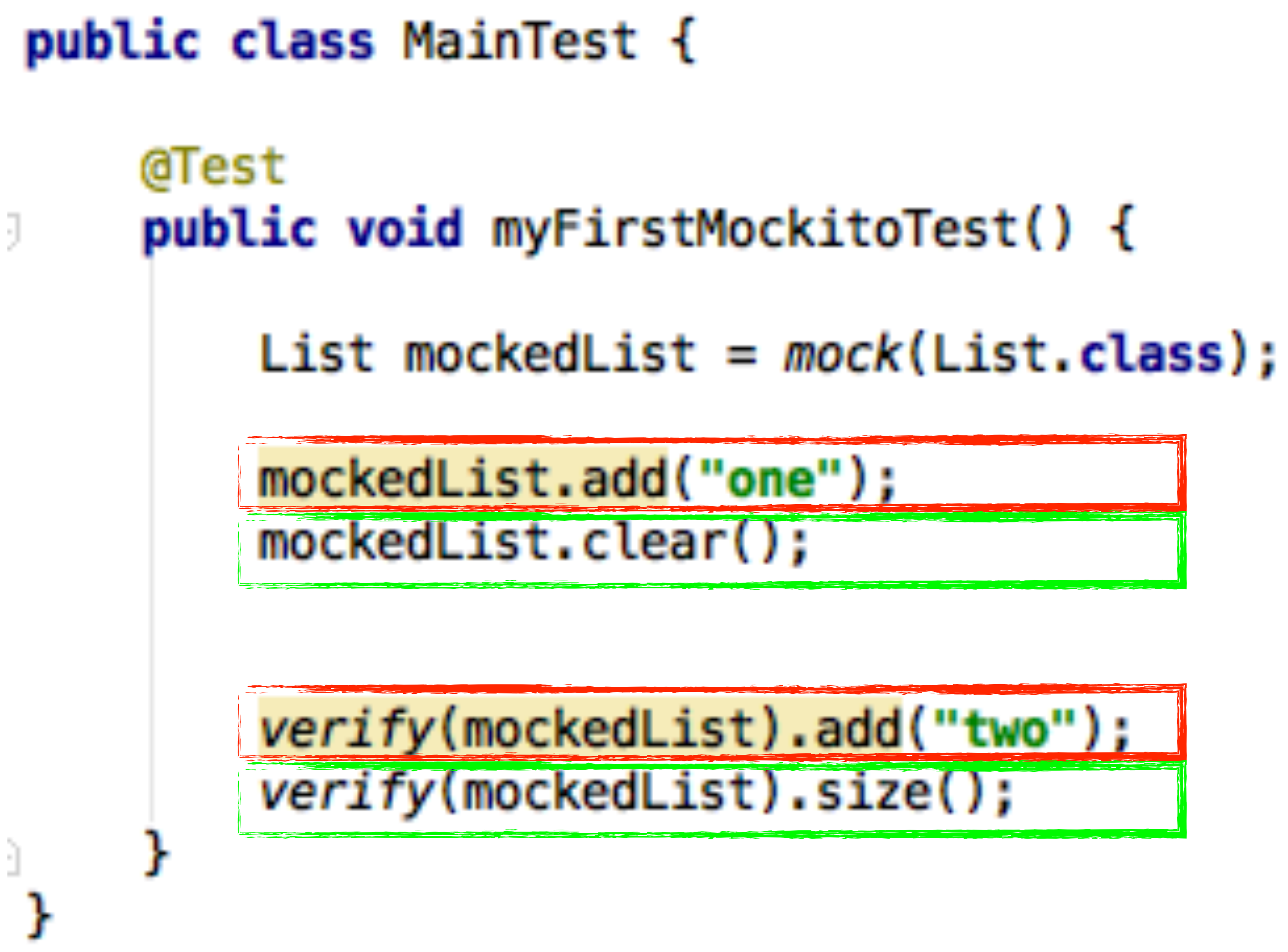
# Example



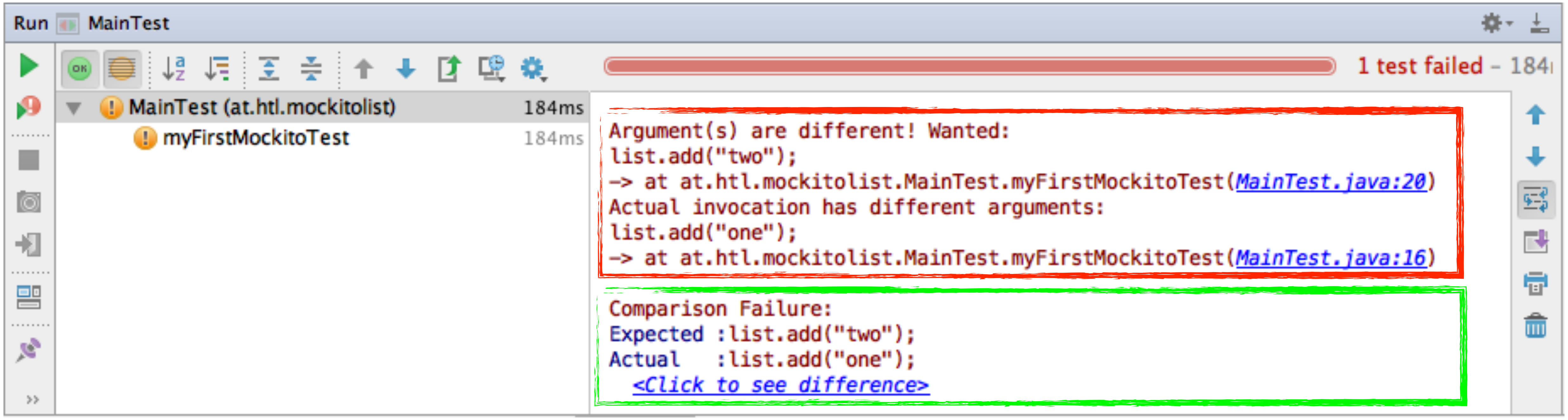
```
List mock = mock(List.class);

someCodeThatInteractsWithMock();

verify(mock, times(3)).clear();
verify(mock, atLeastOnce()).add(anyObject());
```

- In der Liste wurde clear() 3 x aufgerufen

- add() wurde mind. einmal mit einem beliebigen Parameter aufgerufen



```java
public class MainTest {

    @Test
    public void myFirstMockitoTest() {

        List mockedList = mock(List.class);

        mockedList.add("one");
        mockedList.clear();


        verify(mockedList).add("two");
        verify(mockedList).size();
    }
}
```

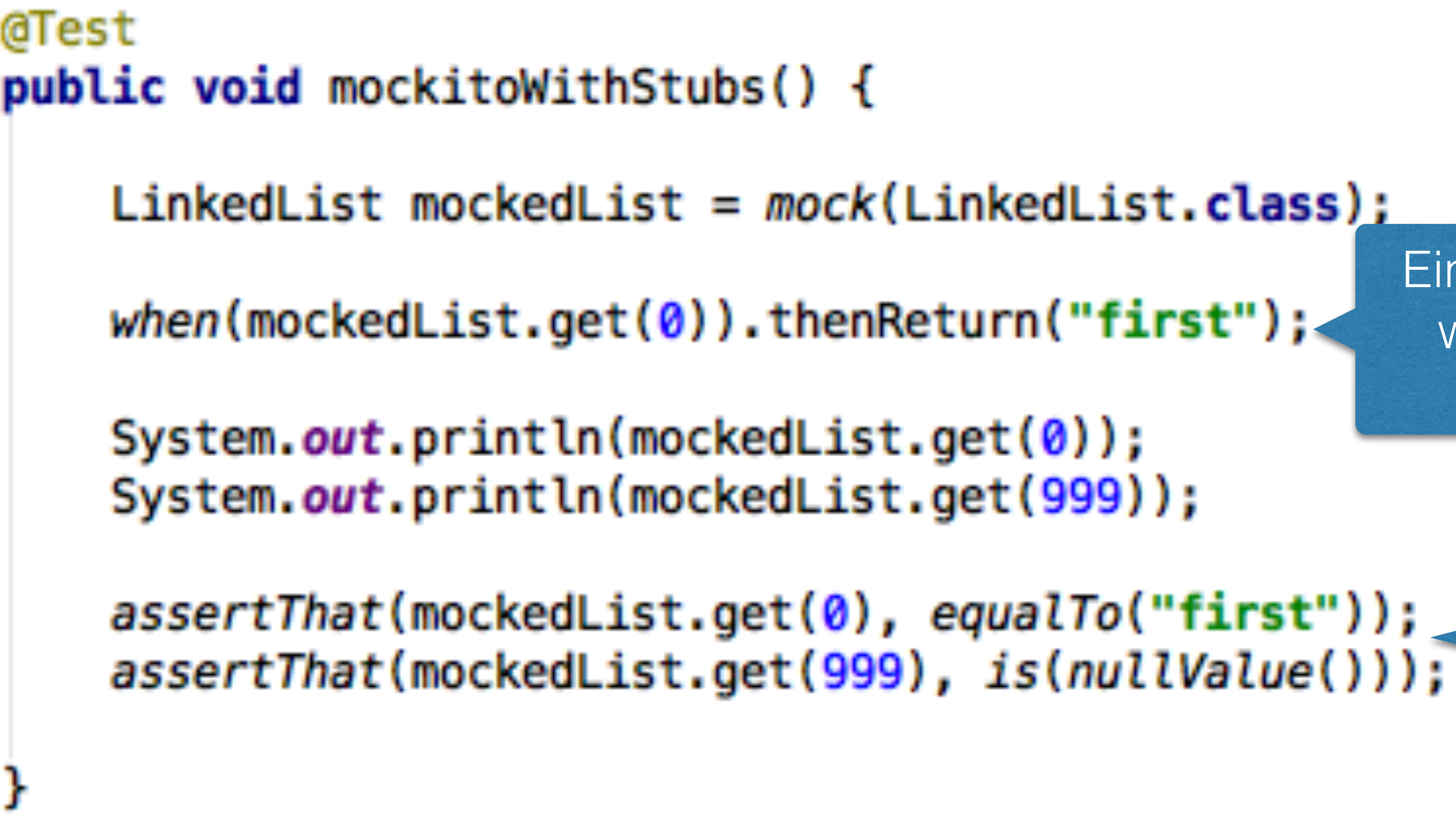


```
MainTest (at.htl.mockitolist)    184ms
    myFirstMockitoTest           184ms
```

```
Argument(s) are different! Wanted:
list.add("two");
-> at at.htl.mockitolist.MainTest.myFirstMockitoTest(MainTest.java:20)
Actual invocation has different arguments:
list.add("one");
-> at at.htl.mockitolist.MainTest.myFirstMockitoTest(MainTest.java:16)
```

```
Comparison Failure:
Expected :list.add("two");
Actual   :list.add("one");
  <Click to see difference>
```

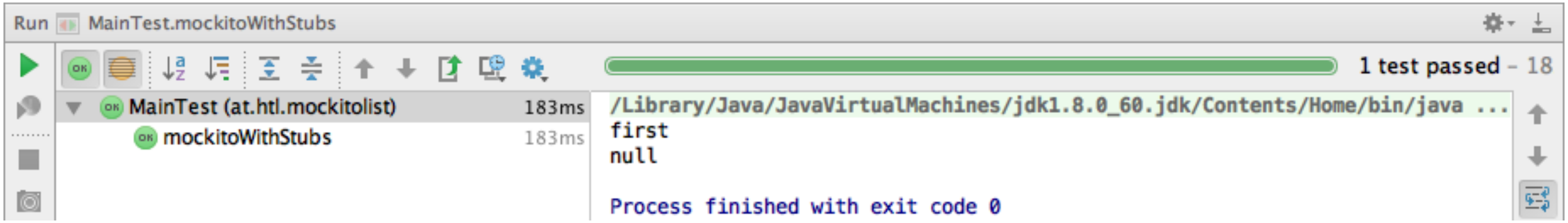
# Mockito mit Stub



```
@Test
public void mockitoWithStubs() {

    LinkedList mockedList = mock(LinkedList.class);

    when(mockedList.get(0)).thenReturn("first");

    System.out.println(mockedList.get(0));
    System.out.println(mockedList.get(999));

    assertThat(mockedList.get(0), equalTo("first"));
    assertThat(mockedList.get(999), is(nullValue()));

}
```

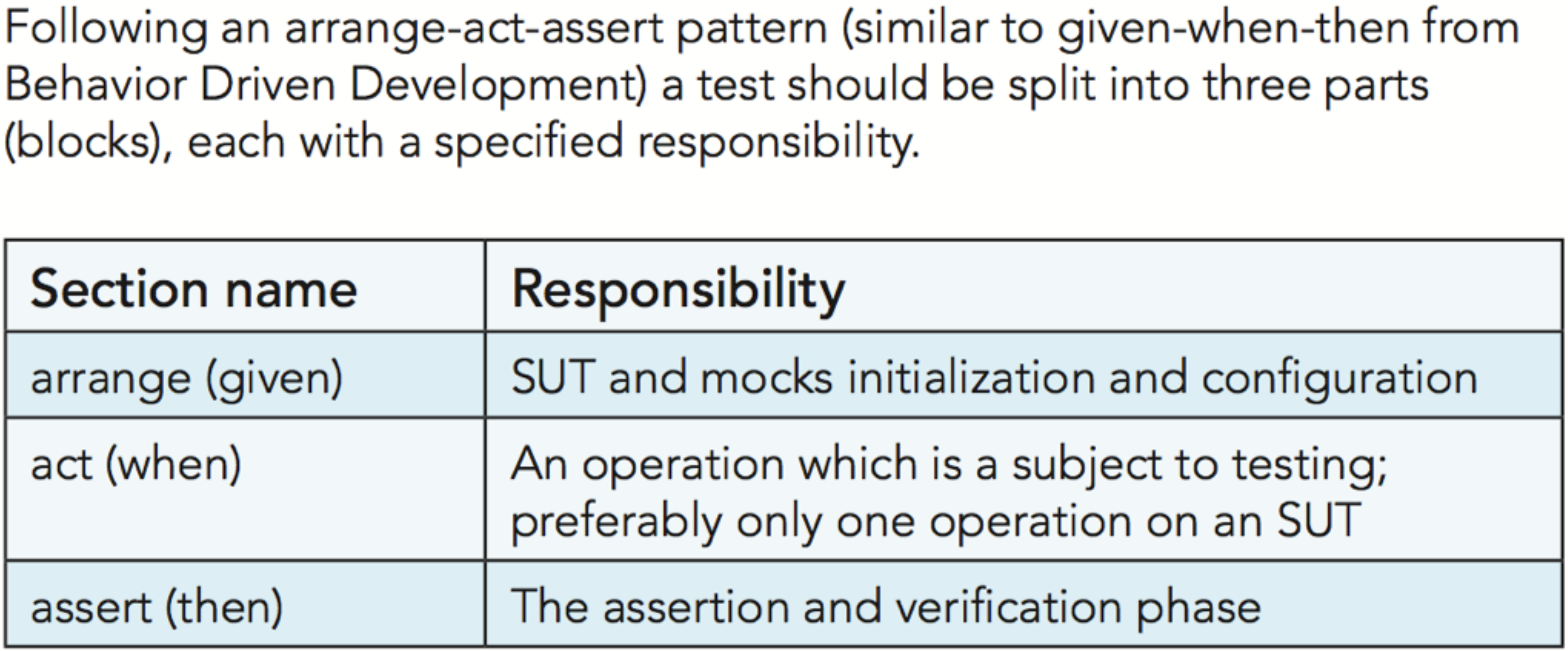
Eine einfache Funktionalität wird in das Mock Objekt implementiert

… und anschließend überprüft


Run
MainTest.mockitoWithStubs
1 test passed – 18
MainTest (at.htl.mockitolist) 183ms
mockitoWithStubs 183ms

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/bin/java ...
first
null

Process finished with exit code 0
```

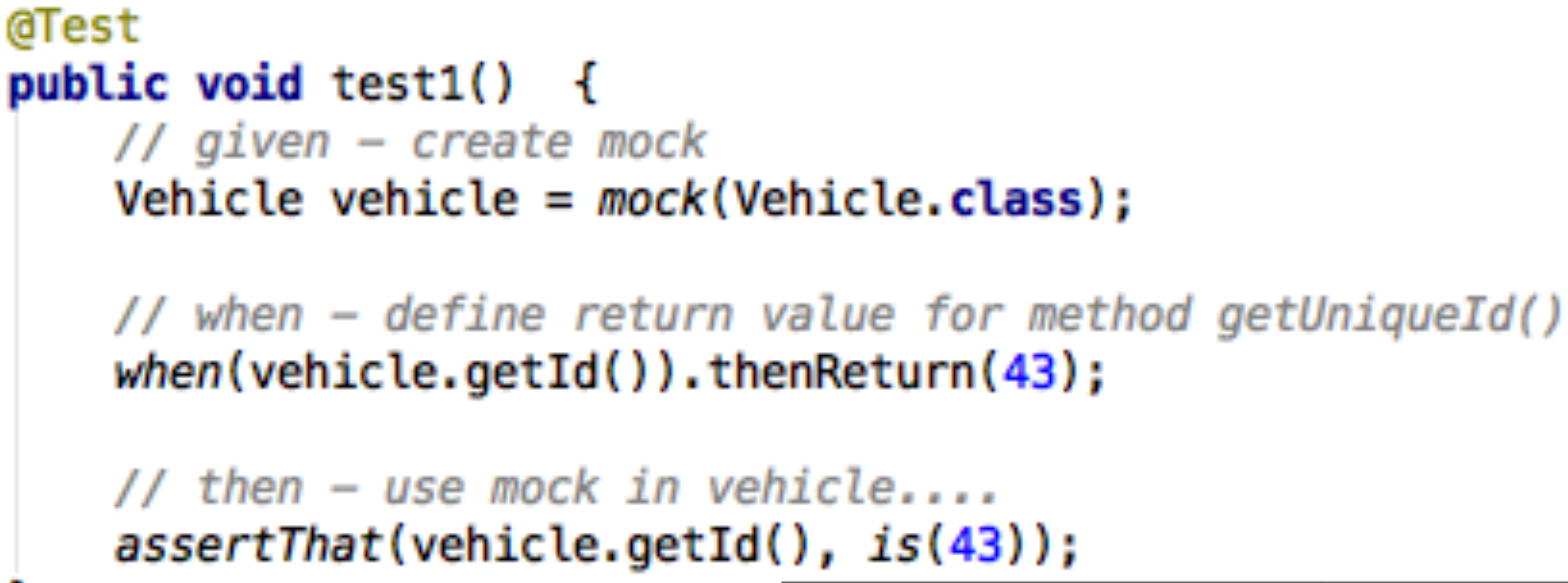
# arrange-act-assert



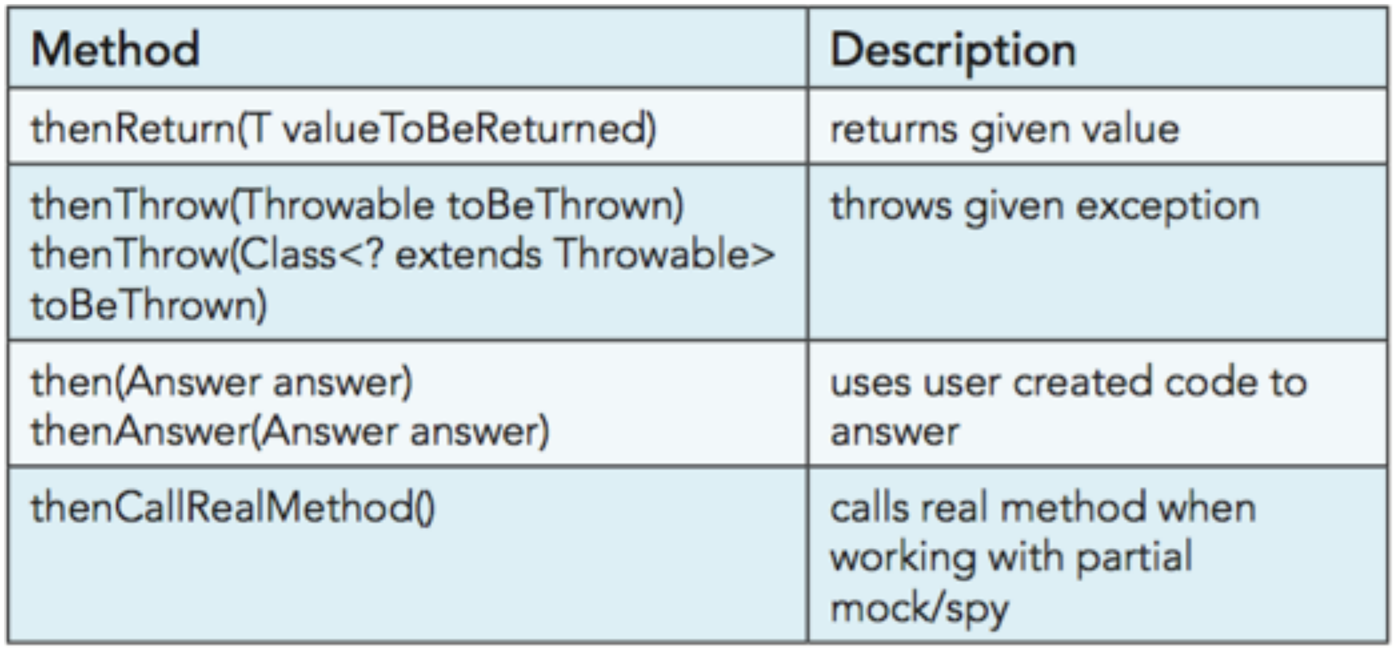
Following an arrange-act-assert pattern (similar to given-when-then from Behavior Driven Development) a test should be split into three parts (blocks), each with a specified responsibility.

| Section name | Responsibility |
|---|---|
| arrange (given) | SUT and mocks initialization and configuration |
| act (when) | An operation which is a subject to testing; preferably only one operation on an SUT |
| assert (then) | The assertion and verification phase |

# Stubbing Method’s returned value

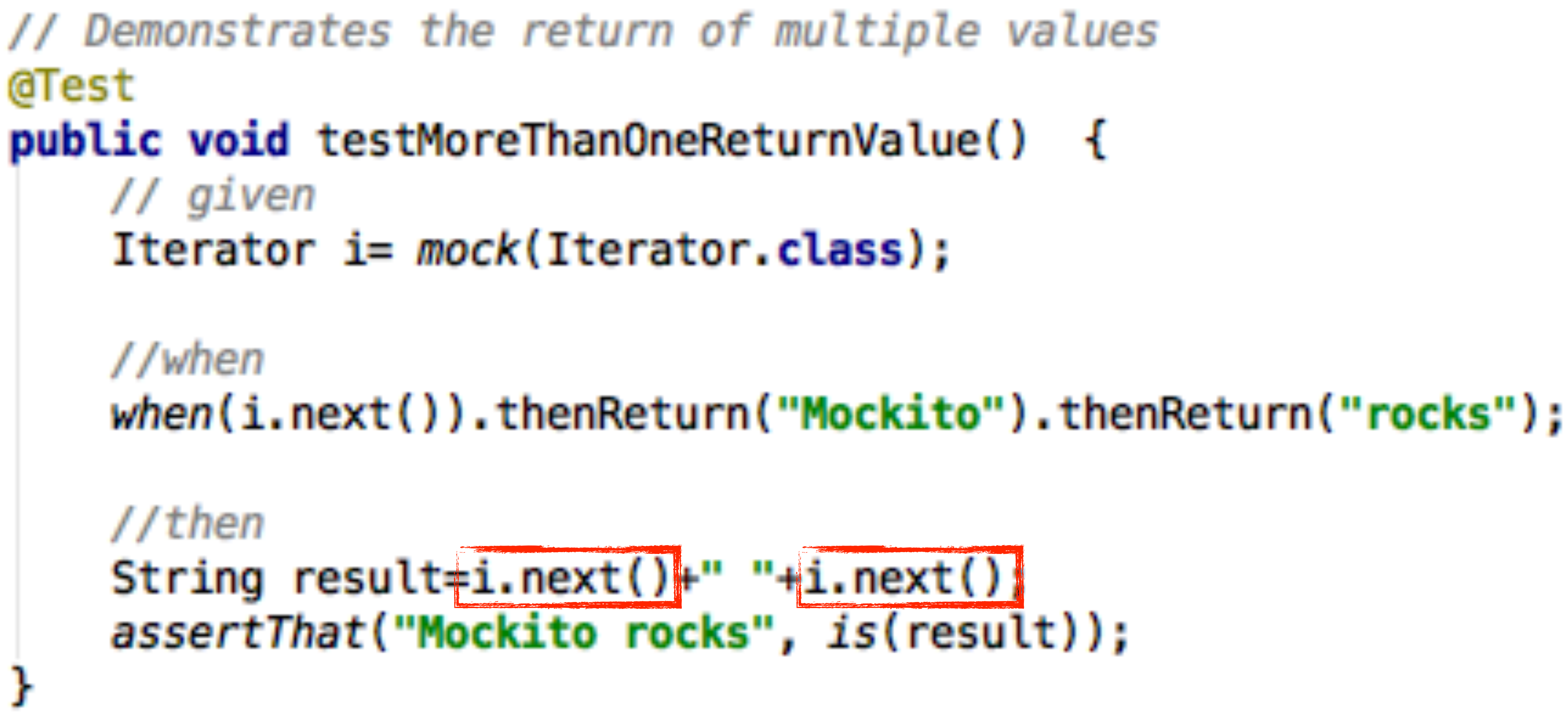


```
@Test
public void test1()  {
    // given - create mock
    Vehicle vehicle = mock(Vehicle.class);

    // when - define return value for method getUniqueId()
    when(vehicle.getId()).thenReturn(43);

    // then - use mock in vehicle....
    assertThat(vehicle.getId(), is(43));
}
```



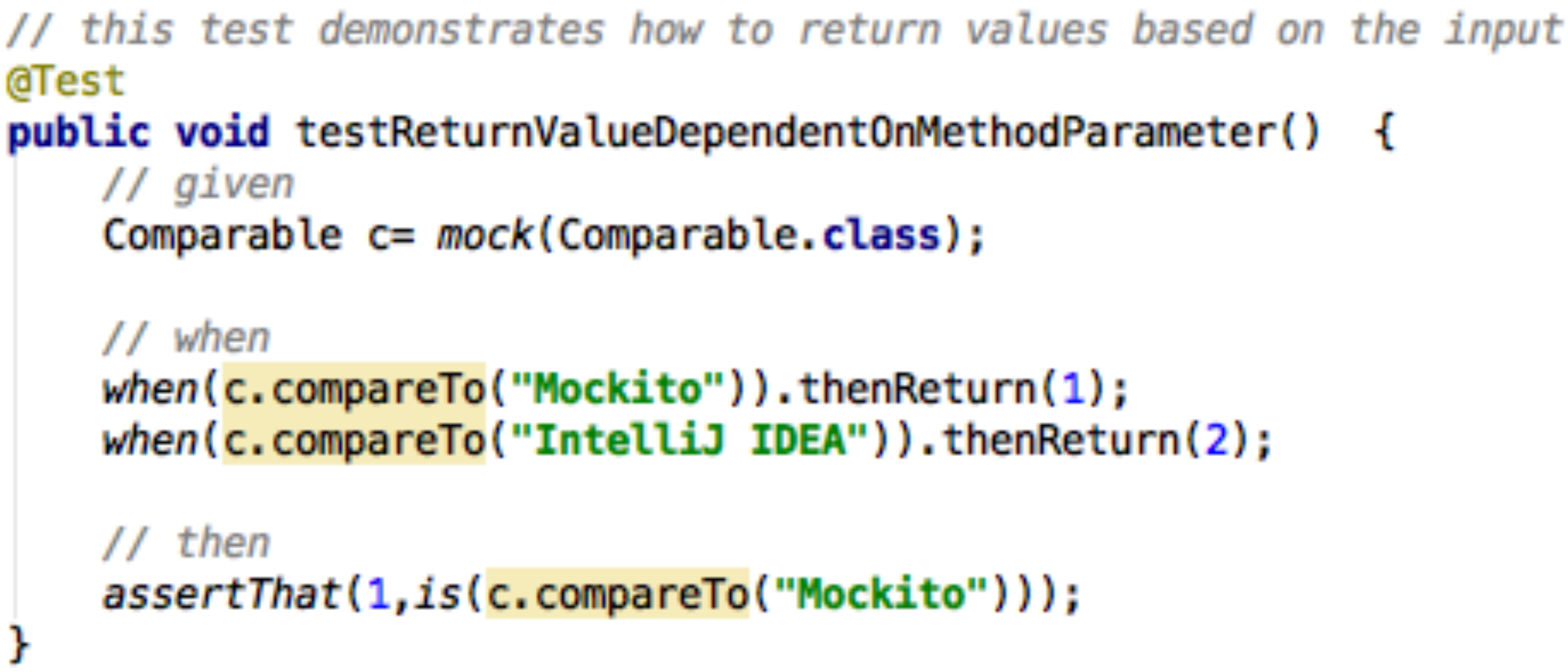
| Method | Description |
|---|---|
| thenReturn(T valueToBeReturned) | returns given value |
| thenThrow(Throwable toBeThrown)<br>thenThrow(Class<? extends Throwable> toBeThrown) | throws given exception |
| then(Answer answer)<br>thenAnswer(Answer answer) | uses user created code to answer |
| thenCallRealMethod() | calls real method when working with partial mock/spy |

# Mehrfacher Aufruf einer Methode



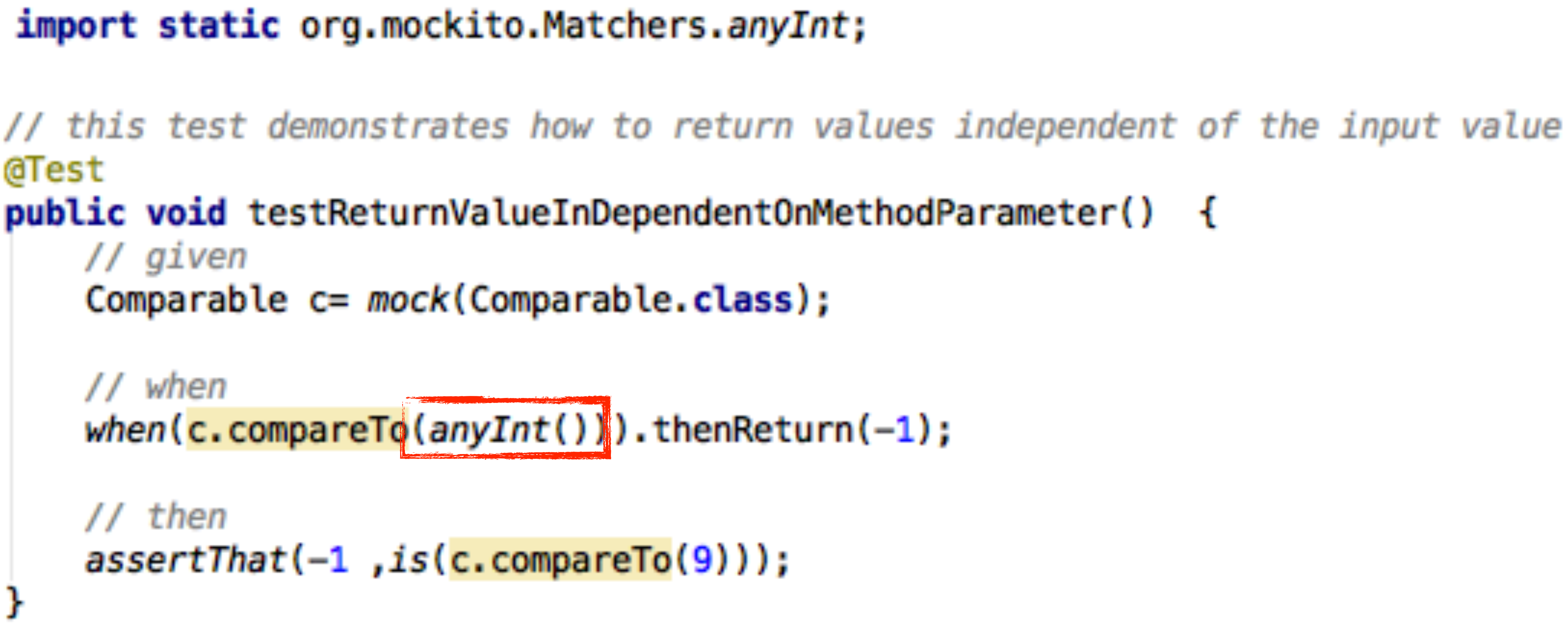
```
// Demonstrates the return of multiple values
@Test
public void testMoreThanOneReturnValue()  {
    // given
    Iterator i= mock(Iterator.class);

    //when
    when(i.next()).thenReturn("Mockito").thenReturn("rocks");

    //then
    String result=i.next()+" "+i.next();
    assertThat("Mockito rocks", is(result));
}
```

# Mit Eingangsparameter…



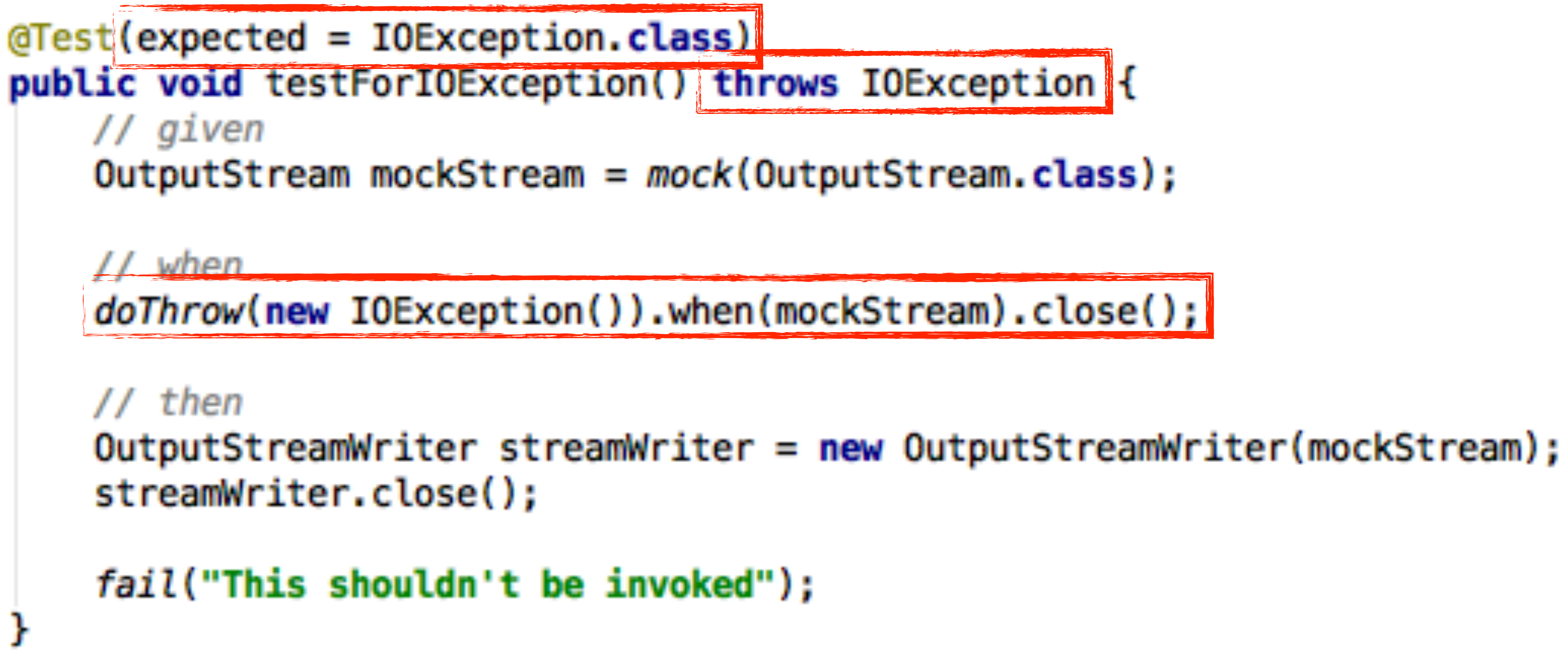
```
// this test demonstrates how to return values based on the input
@Test
public void testReturnValueDependentOnMethodParameter()  {
    // given
    Comparable c= mock(Comparable.class);

    // when
    when(c.compareTo("Mockito")).thenReturn(1);
    when(c.compareTo("IntelliJ IDEA")).thenReturn(2);

    // then
    assertThat(1,is(c.compareTo("Mockito")));
}
```

# anyInt()



```
import static org.mockito.Matchers.anyInt;

// this test demonstrates how to return values independent of the input value
@Test
public void testReturnValueInDependentOnMethodParameter()  {
    // given
    Comparable c= mock(Comparable.class);

    // when
    when(c.compareTo(anyInt())).thenReturn(-1);

    // then
    assertThat(-1 ,is(c.compareTo(9)));
}
```
)

# Exceptions



```java
@Test(expected = IOException.class)
public void testForIOException() throws IOException {
    // given
    OutputStream mockStream = mock(OutputStream.class);

    // when
    doThrow(new IOException()).when(mockStream).close();

    // then
    OutputStreamWriter streamWriter = new OutputStreamWriter(mockStream);
    streamWriter.close();

    fail("This shouldn't be invoked");
}
```

Was ist der Unterschied zwischen assertThat(…) und verify(…)?

# Mocking eines Entity Managers

http://www.adam-bien.com/roller/abien/entry/mocking_jpa_entitymanager_with_query to be continued …

verify()

@Mock

@Spy

# Resources

- http://www.vogella.com/tutorials/JUnit/article.html
- https://www.jetbrains.com/idea/help/testing.html

Noch
Fragen?