Mock Objects

and other test doubles

PRAGMATISTS

www.pragmatists.pl

Classical testing

- All objects which participate in testing are created and set up
- The state of all objects is being verified
- Mocks are created only for objects really hard to test

Mockist testing

- THE ONLY real object is the object being tested (System Under Test – SUT)
- ALL other objects are mocked
- The state is verified only on SUT
- For other objects only interaction is being verified (behaviour requested by SUT)

Classical testing - example

```
public class OrderTestClassical {

  private static final String POTATOES = "potatoes";
  private Warehouse warehouse = new Warehouse();

  @Before
  public void setUpWerhouse() {
    warehouse.add(POTATOES, 50);
  }

  @Test
  public void orderShouldBeFilled() {
    Order order = new Order(POTATOES, 50);

    order.fill(warehouse);

    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(POTATOES));
  }

  @Test
  public void orderShouldNotBeFilled() {
    Order order = new Order(POTATOES, 51);

    order.fill(warehouse);

    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(POTATOES));
  }
}
```

Real impl of Warehouse class is used.

Should Warehouse have errors, tests for Order class will fail – it will be hard to locate the erronous code

So... it's not a unit test ;-)

Test doubles

Test doubles types, after Gerard Meszaros (xUnit Patterns book):

- Dummy – object passed as a method parameter, but never really used. Needed to fill in the parameters' list.
- Fake – object having a working, but simplified implementation.
- Stub – returns predefined result for given parameter values. It can remember method calls.
- Mock – object with programmed expectations. One can verify behaviour on it.
- Spy – a Stub with a verifiable behaviour.

Mockist approach
jMock

```
@RunWith(JMock.class)
public class OrderTestJMock {

  private static final String POTATOES = "potatoes";
  private Mockery context = new JUnit4Mockery();
  private IWarehouse warehouse = context.mock(IWarehouse.class);

  @Test
  public void orderShouldBeFilled() {
    context.checking(new Expectations() {
      {
        oneOf(warehouse).hasInventory(POTATOES, 50);
        will(returnValue(true));
        oneOf(warehouse).remove(POTATOES, 50);
      }
    });
    Order order = new Order(POTATOES, 50);

    order.fill(warehouse);

    assertTrue(order.isFilled());
  }
(…)

}
```

Additional object

Mocking on interfaces

Definition of mock's behaviour

Mockist approach
Mockito

```
public class OrderTestMockito {

  private static final String POTATOES = "potatoes";
  private Warehouse warehouse = mock(Warehouse.class);

  @Test
  public void orderShouldBeFilled() {
    when(warehouse.hasInventory(POTATOES, 50)).thenReturn(true);
    Order order = new Order(POTATOES, 50);

    order.fill(warehouse);

    assertTrue(order.isFilled());
    verify(warehouse).remove(POTATOES, 50);
  }
}
```

Mock on a class

Definition of returned values

Verification of a method call

Creating mocks

v1

```
public class OrderTestJMockito {
        private Warehouse warehouse = mock(Warehouse.class);
```

v2

```
@RunWith(MockitoJUnitRunner.class)
public class OrderTestJMockito {
        @Mock
        private Warehouse warehouse;
```

Mockito – Definition of behaviour

```
when(warehouse.hasInventory(POTATOES, 50)).thenReturn(true);

when(warehouse.hasInventory(anyString(), anyInt())).thenReturn(true, false);

when(warehouse.hasInventory(anyString(), anyInt())).thenThrow(new RuntimeException("ooops!"));
```

```
when(warehouse.hasInventory(POTATOES, 50))
        .thenReturn(true)
        .thenReturn(false)
        .thenThrow(new RuntimeException("ooops!"));


doThrow(new RuntimeException("ops!")).when(warehouse).remove(any(String.class), any(Integer.class));

doNothing()
  .doThrow(new RuntimeException("ooops!"))
  .when(warehouse).remove(any(String.class), any(Integer.class));
```

Mockito – Verification

```
verify(warehouse).remove(POTATOES, 50);

verify(warehouse).remove(anyString(), eq(50));

verify(warehouse, times(10)).remove(POTATOES, 50);

verify(warehouse, times(0)).remove(POTATOES, 50);

verify(warehouse, never()).remove(POTATOES, 50);

verifyNoMoreInteractions(warehouse, anotherMock, (...));

verifyZeroInteractions(warehouse, anotherMock, (...));
```

Mockito – Verification of call order

Normally the order of calls is not verifed

```
Object mock1 = mock(Object.class);
Object mock2 = mock(Object.class);
(…)

InOrder inOrder = inOrder(mock1, mock2, (...));
inOrder.verify(mock1).toString();
inOrder.verify(mock2).hashCode();
inOrder.verify(mock1).getClass();
```

Mockito – verification of argument values

```
verify(warehouse).assign(argThat(new ArgumentMatcher<Person>() {
  @Override
  public boolean matches(Object object) {
    Person person = (Person) object;
    return "Наталия".equals(person.getName());
  }
}));
```

```
ArgumentCaptor<Person> argumentCaptor = ArgumentCaptor.forClass(Person.class);
verify(warehouse).assign(argumentCaptor.capture());
assertEquals("Наталия", argumentCaptor.getValue().getName());
```

Mockito – Answer

```
Answer<Void> answer = new Answer<Void>() {
  @Override
  public Void answer(InvocationOnMock invocation) throws Throwable {
    Object[] params = invocation.getArguments();
    // do something with the params
    return null;
  }
};

doAnswer(answer).when(warehouse).remove(anyString(), eq(1));

when(warehouse.hasInventory(anyString(), eq(1))).thenAnswer(answer);
```

Mockito – forget previous interactions

```
Repository repository = mock(Repository.class);

Service service = new Service(repository);

// something doing e.g. repository.save();

reset(repository);

// code under test

verifyZeroInteractions(repository);
```

Mockito – mocking chained calls

```
Person me = mock(Person.class, RETURNS_DEEP_STUBS);

when(me.homeAddress().street()).thenReturn("Хращатик");

assertEquals("Хращатик", me.homeAddress().street());

verify(me.homeAddress()).street();
```

Mockito – Spy

A spy in mockito is a real object, of which some methods can be mocked and their behaviour verified

```
@RunWith(MockitoJUnitRunner.class)
public class OrderTestMockito {

    @Spy
    private Warehouse warehouse = new Warehouse();
```

or

```
    private Warehouse warehouse = spy(new Warehouse());
```

It's created on a real object, not a class!

Spy - mocking methods

- We cannot mock spy's methods like mock's – we'd then call real method implementation!

```
when(spy.hasInventory(POTATOES, 50)).thenReturn(true);
```

- So we need to reverse the order:

```
doReturn(true).when(spy).hasInventory(POTATOES, 50);

doNothing().when(spy).remove(POTATOES, 50);
```

Будьмо!