Mocking

SADRŽAJ

- Uvod
- Mocking
- Mock vs Spy
- Mockito framework
- Mockito When/Then pravila
- Mockito Verify

UVOD

- Unit testiranje je testiranje u izolaciji testiranog dela softvera od ostatka
- Međutim, većina softverskih celina, komponenti ili delova koji se testiraju ne mogu da funkcionišu nezavisno u odnosu na ostatak softvera, te se tako ni njihova funkcionalnost ne može testirati izolovano
- Kako onda izvršiti testiranje?

MOCKING

- Da bi se testirani objekat testirao u izolaciji važno je da referencirani objektni ne unose grešku, zbog toga je potrebno simulirati njihov rad

- **Mocking** mehanizan omogućuje simulaciju ponašanja objekata koje testirani objekat koristi

- Umesto pravih referenciranih objekata postavljaju se **objektni dvojnici** koji pojednostavljuju ili simuliraju ponašanje referenciranih objekata

MOCKING

REAL SYSTEM

Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

CLASS IN UNIT TEST

Green = class in focus
Yellow = mocks for the unit test

MOCKING – OBJEKTNI DVOJNICI

- Postoji različiti tipovi dvojnika:

1. Dummy objekti
    - Dummy objekti su objekti koji se ne koriste u testu i obično ne sadrži nikakvu implementaciju. Služe kao placeholderi, kako bi se zadovoljili potpisi funkcija
2. Fake objekti
    - Fake objektima se nazivaju oni koji koriste određene prečice ili pojednostavljenu verziju koda u odnosu na radnu verziju
3. Stub objekti
    - Imaju unapred predefinisan skup podataka koje koristi kao odgovor na zahteve sistema. Koriste se u situacijama kada nije moguće ili se izbegava odgovor sa pravim podacima u fazi testiranja
4. Mock objekti
5. Spy objekti

MOCKING – MOCK OBJEKTI

- **Mock objekat** simulira ponašanje stvarnog objekta
- Ako stvarni objekat sadrži sekvencu poziva određenih metoda, mock objekat ponavlja ovu sekvencu
- Na ovaj način dobijamo dvojnika koji se i ponaša kao stvarni objekat
- Sami izlazi koji su rezultat ponašanja su i dalje simulirani

MOCKING – MOCK OBJEKTI PRIMENA

- Jedan slučaj kada su nam potrebni mock objekti:
  - Želimo da testiramo klasu A u izolaciji
  - (samo nju - jer su testovi koji izoluju unit-e obično bolji, jasno naglašavaju testirano ponašanje)
    - Klasa A ima referencu na klasu B
    - Ako u testiranje klase A uvrstimo i testiranje klase B naš test postaje preopširan i manje upotrebljiv.
- Rešenje koje predlaže Mock pristup:
  - Napraviti "lažan" / mock objekat klase B (zovimo ga mB) koji poštuje isti interfejs kao “pravi".
  - Mock objekat mB koristiti na mestu "pravog" B.

MOCKING – SPY OBJEKTI

- **Spy objekat** je modifikovani stvarni objekat
- Deo ponašanja spy objekta je stvarno, dok je deo simuliran za potrebe testiranja

MOCK VS SPY

- **Mock**
  - Lažni objekat koji **u potpunosti** zamenjuje pravi objekat
  - Mock se koristi kada imamo instancu kompleksne klase koja koristi eksterne resurse poput mreže, fajlova, baza podataka ili recimo gomilu nekih drugih objekata, da takve aktivnosti „lažiramo“
  - Objekat se mokuje da bi se izvršilo izolovanje objekta od ostatka sistema
- **Spy**
  - Pravi objekat, kome su **samo neke metode** zamenjene lažnim implementacijama
  - Ako neka metoda nije zamenjena (shadowed), njen poziv će u testu izazvati poziv prave metode originalnog objekta
  - Na ovaj način možemo lažirati ponašanje kompleksnih metoda, a one jednostavne koje nema smisla lažirati možemo pustiti da se izvrše u originalu
- Mock podrazumeva zamenu svih metoda, dok spy podrazumeva zamenu samo nekih, obično onih kompleksnih, dok se jednostavne ne lažiraju

MOCKITO FRAMEWORK

- Najpopularniji Java mocking framework
- Omogućuje pisanje jasnih i jednostavnih testova koji proizvode čitljive validacione greške
- Dokumentacija: https://static.javadoc.io/org.mockito/mockito-core/2.22.0/org/mockito/Mockito.html

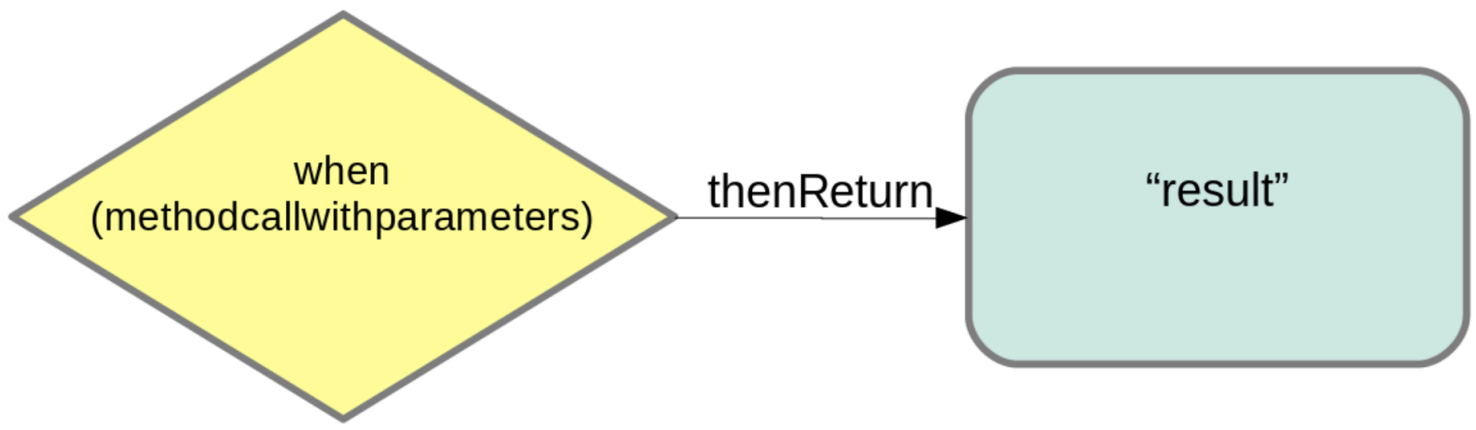
MOCKITO FRAMEWORK – MOCK I SPY

- Kada se koristi **Mock**:
  - Podrazumevano ponašanje za metode koje nisu lažirane jeste da ne rade ništa
  - Metoda koja nema povratni tip podataka (**void**) onda neće raditi ništa
  - Metoda koja ima povratnu vrednost će vraćati **null**, **empty**, **nulu** ili **default** vrednost
  - `MyList listMock = Mockito.mock(MyList.class);`
- Kada se koristi **Spy**:
  - Ukoliko metoda nije lažirana, njen poziv će biti preusmeren na stvarnu implementaciju te metode
  - `MyList listMock = Mockito.spy(MyList.class);`

MOCKITO WHEN/THEN PRAVILA

- Mokovani objekti mogu da vrate različite povratne vrednosti u zavisnosti od argumenata koji su prosleđeni metodi

- Ulančavanjem when(...).thenReturn(...) metoda možemo definisati povratnu vrednost za predefinisane parametre



MOCKITO WHEN/THEN PRAVILA

- Postavljanje jednostavne povratne vrednosti:

```
//  create mock
MyClass test = mock(MyClass.class);

// define return value for method getUniqueId()
when(test.getUniqueId()).thenReturn(43);

// use mock in test....
assertEquals(test.getUniqueId(), 43);
```

- Alternativan način postavljanja jednostavne povratne vrednosti:

```
//  create mock
MyClass test = mock(MyClass.class);

// define return value for method getUniqueId()
doReturn(43).when(test).getUniqueId();

// use mock in test....
assertEquals(test.getUniqueId(), 43);
```

- Primer: src/test/java/unittesting.mock/MockitoExampleTest.java testSimpleReturnValue(), testSimpleReturnValue2()

MOCKITO WHEN/THEN PRAVILA

- Konfigurisanje mock objekta da baca izuzetak na poziv metode:

```
@Test(expectedExceptions = IllegalStateException.class)
public void testThrowException() {
    MyList listMock = Mockito.mock(MyList.class);
    when(listMock.add(anyString())).thenThrow(IllegalStateException.class);
    listMock.add(randomAlphabetic(6));
}
```

- Konfigurisanje mock objekta da metoda bez povratne vrednosti (void) baca izuzetak:

```
MyList listMock = Mockito.mock(MyList.class);
doThrow(NullPointerException.class).when(listMock).clear();
listMock.clear();
```

- Primer: src/test/java/unittesting.mock/MockitoExampleTest.java testForIOException()

MOCKITO WHEN/THEN PRAVILA

- Postavljanje ponašanja nakon više poziva:

```
// demonstrates the return of multiple values
@Test
public void testMoreThanOneReturnValue()  {
    Iterator<String> i= mock(Iterator.class);
    when(i.next()).thenReturn("Mockito").thenReturn("rocks");
    String result= i.next()+" "+i.next();
    //assert
    assertEquals("Mockito rocks", result);
}
```

- Primer: src/test/java/unittesting.mock/MockitoExampleTest.java testMoreThanOneReturnValue()

MOCKITO WHEN/THEN PRAVILA

- Pozivanje prave metode bez mockovanja vrednosti:

```
MyList listMock = Mockito.mock(MyList.class);
when(listMock.size()).thenCallRealMethod();
assertThat(listMock.size(), equalTo(1));
```

- Primer: src/test/java/unittesting.mock/MockitoExampleTest.java testCallMethodThatIsNotMocked()

MOCKITO VERIFY

- Do sada smo mogli da testiramo samo stanje objekta i za tu namenu smo koristili asertacije
- Mockito framework nam omogućava da testiramo i interakciju sa mock objektom
- Na taj način možemo da proverimo da li je naš objekat korišćen, na koji način itd
- Primer: ako smo putem mock objekta simulirali snimanje u bazu, možemo proveriti da li se metoda za snimanje pozvala tačno jednom

MOCKITO VERIFY

- Provera da li je postojala interakcija sa mock objektom:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
// provera da li je ikada nad mock objektom pozvana metoda .size()
verify(mockedList).size();
```

- Provera ukupnog broja interakcija sa mock objektom:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
// provera da li je nad mock objektom pozvana metoda .size() tačno 2 puta
verify(mockedList, times(2)).size();
```

- Primer: src/test/java/unittesting.mock/MockitoVerify.java testVerify()

MOCKITO VERIFY

- Provera interakcije sa mock objektom:

```
List<String> mockedList = mock(MyList.class);
// provera da li je bilo interakcije sa mock objektom
verifyZeroInteractions(mockedList);
```

- Provera interakcija sa mock objektom pozivom neke konkretne metode:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
// provera da nije bilo interakcije nad pozivom neke konkretne metode
verify(mockedList, times(0)).size();
```

- Primer: src/test/java/unittesting.mock/MockitoVerify.java testVerify()

MOCKITO VERIFY

- Provera da nema neočekivanih interakcija (ovaj test treba da padne):

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
mockedList.clear();
verify(mockedList).size();
verifyNoMoreInteractions(mockedList);
```

- Provera redosleda interakcija:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
mockedList.add("a parameter");
mockedList.clear();

InOrder inOrder= Mockito.inOrder(mockedList);
inOrder.verify(mockedList).size();
inOrder.verify(mockedList).add("a parameter");
inOrder.verify(mockedList).clear();
```

- Primer: src/test/java/unittesting.mock/MockitoVerify.java testVerifyNoMoreInteractions(), testVerifyInOrder()

MOCKITO VERIFY

- Provera da se interakcija nije desila:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList, never()).clear();
```

- Provera da se interakcija desila bar određen broj puta ili manje od određenog broja puta:

```
List<String> mockedList = mock(MyList.class);
mockedList.clear();
mockedList.clear();
mockedList.clear();

verify(mockedList, atLeast(1)).clear();
verify(mockedList, atMost(10)).clear();
```

- Primer: src/test/java/unittesting.mock/MockitoVerify.java testVerify()

MOCKITO VERIFY

- Provera interakcije sa konkretnim argumentom:

```
List<String> mockedList = mock(MyList.class);
mockedList.add(“test”);
verify(mockedList).add(“test”);
```

- Provera interakcije sa fleksibilnim/bilo kojim argumentom:

```
List<String> mockedList = mock(MyList.class);
mockedList.add(“test”);
verify(mockedList).add(anyString());
```