MOCKITO

NIVOI TESTIRANJA

- Jedinično (unit) testiranje
  - Verifikuje ponašanje svake softverske komponente nezavisno od ostatka sistema
- Integraciono (integrity) testiranje
  - Verifikuje međusobnu interakciju skupa komponenti prema definisanom dizajnu
- Sistemsko (system) testiranje
  - Verifikuje kompleto integrisan sistem kako bi se utvrdilo da li odgovara unapred definisanoj specifikaciji
- Test prihvatljivosti (acceptance)
  - Utvrđuje da li sistem zadovoljava potrebe, zahteve i očekivanja naručioca

JEDINIČNO (UNIT) TESTIRANJE

- Testiranje programskih komponenti nezavisno od ostalih delova sistema
- Predstavlja nivo testiranja u kome se verifikuje funkcionalnost određene sekcije koda testiranog programa, obično na funkcionalnom nivou
- Jedinično testiranje treba vršiti u kontrolisanom okruženju tako da tim za testiranje može komponenti koja se testira da predaje unapred definisan skup podataka i da posmatra izlazne akcije i rezultate. Takođe, tim za testiranje proverava unutrašnju strukturu podataka, logiku i granične uslove za ulazne i izlazne podatke
-

src/test/java/rs/ac/uns/testing/car/CarTest.java

JEDINIČNO (UNIT) TESTIRANJE

- Tipičan jedinični test se sastoji iz 3 faze:
    1. Inicijalizacija malog dela aplikacije koji želimo da testiramo (poznat i kao sistem koji se testira ili SUT)
    2. Primena stimulansa na sistem koji se testira (obično pozivanjem metoda nad njim)
    3. Posmatranje rezultirajućeg ponašanja. Ako je uočeno ponašanje u skladu sa očekivanjima, jedinični test prolazi, u suprotnom ne prolazi, što ukazuje da postoji problem negde u sistemu koji se testira
- Ove tri faze testa jedinice poznate su i kao Arange, Act i Assert, ili jednostavno `AAA`

```java
@Test
public void testGasTankAfterDriving() {
    // In the Arrange phase, we create and set up a system under test.
    // A system under test could be a method, a single object, or a graph of connected objects.
    // It is OK to have an empty Arrange phase, for example if we are testing a static method -
    // in this case SUT already exists in a static form and we don't have to initialize anything explicitly.
    Car test_car = new Car( make: "Toyota",  model: "Prius",  gasTankSize: 10,  milesPerGallon: 50);

    // The Act phase is where we poke the system under test, usually by invoking a method.
    // If this method returns something back to us, we want to collect the result to ensure it was correct.
    // Or, if method doesn't return anything, we want to check whether it produced the expected side effects.
    test_car.drive( miles: 50);
    double actual =  test_car.getGasTankLevel();

    // The Assert phase makes our unit test pass or fail.
    // Here we check that the method's behavior is consistent with expectations.
    assertEquals( actual: 9, actual, delta: .001);
}
```

CILJEVI TESTIRANJA

- Testiranje funkcionalnosti
  - Testira se input/output ponašanje test objekta
- Testiranje robusnosti
  - Testira se u kojoj meri su test objekti otporni na otkazivanje, fokus na stvari koje nisu dozvoljene ili nisu obuhvaćene specifikacijom
- Testiranje efikasnosti
  - Testira se zauzeće resursa (potrošnja memorije, vreme izračunavanja, zauzeće diska itd.)
- Testiranje održavanja
  - Struktura koda, modularnost, pokrivenost koda komentarima i dokumentacijom itd.

TEST OBJEKTI

- Kod ovog tipa testiranja test objekti su pojedinačne softverske komponente, najčešće klase
- Osnovna karakteristika unit testiranja jeste da se komponente testiraju pojedinačno i izolovano od ostatka sistema
- Izolacijom isključujemo spoljašnje uticaje drugih komponenti na komponentu koja se testira
- Detektovani problem onda definitivno ukazuje na nedostatak u testiranoj komponenti

PROBLEMI UNIT TESTIRANJA

- Prilikom unit testiranja želimo potpunu izolaciju dela softvera koji testiramo
- 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?

```
public String getTimeOfDay()
{
    Calendar time = Calendar.getInstance();

    if (time.get(Calendar.HOUR_OF_DAY) >= 0 && time.get(Calendar.HOUR_OF_DAY) < 6)
    {
        return "Night";
    }
    if (time.get(Calendar.HOUR_OF_DAY) >= 6 && time.get(Calendar.HOUR_OF_DAY) < 12)
    {
        return "Morning";
    }
    if (time.get(Calendar.HOUR_OF_DAY) >= 12 && time.get(Calendar.HOUR_OF_DAY) < 18)
    {
        return "Afternoon";
    }
    return "Evening";
}
```

REŠENJE 1

- Svakako jedno od rešenja jeste refactoring koda
- Problem malopređašnje metode jeste taj što rezultat jedničnog testa koji bi se za njega napisao nije deterministički, već će njegovo izvršavanje zavisiti od trenutka pokretanja testa
- Takođe, ovako napisana metoda narušava `SRP (Single Responsibility Principle)`
- Izmeštanjem kreiranja trenutnog vremena iz koda u argument funkcije dobijamo mogućnost kreiranja determinističkih testova

```
public String getTimeOfDay(Calendar time)
{
    if (time.get(Calendar.HOUR_OF_DAY) >= 0 && time.get(Calendar.HOUR_OF_DAY) < 6)
    {
        return "Night";
    }
    if (time.get(Calendar.HOUR_OF_DAY) >= 6 && time.get(Calendar.HOUR_OF_DAY) < 12)
    {
        return "Morning";
    }
    if (time.get(Calendar.HOUR_OF_DAY) >= 12 && time.get(Calendar.HOUR_OF_DAY) < 18)
    {
        return "Afternoon";
    }
    return "Evening";
}
```

```
@Test
public void getTimeOfDay_For6AM_ReturnsMorning() {
    SmartHouseSystemRefactored smartHouse = new SmartHouseSystemRefactored();

    Calendar time = Calendar.getInstance();
    time.set(Calendar.HOUR_OF_DAY, 6);

    String actual = smartHouse.getTimeOfDay(time);

    Assert.assertEquals(actual, expected: "Morning");
}
```

PROBLEMI UNIT TESTIRANJA

- Ukoliko je klasa `SmartHouseSystem` deo sistema za automatsko uključivanje i isključivanje svetala na osnovu doba dana, onda smo problem testiranja samo podigli na jedan nivo iznad
- Pozivanjem `Calendar.getInstance()` metode u klasi kontrolera, dobijamo situaciju od malopre
- Rešenje leži u dvema činjenica
  - metodu za kreiranje instance vremena moguće je dodeliti zasebnoj klasi `(IoC – Inversion of Controll)`. IoC je moguće implementirati na nekoliko načina, u konkretnom slučaju to će biti `Dependency Injection`
  - nakon toga, moguće je sa lakoćom upotrebiti objekte dvojnike, odnosno iskoristiti princip mokovanja

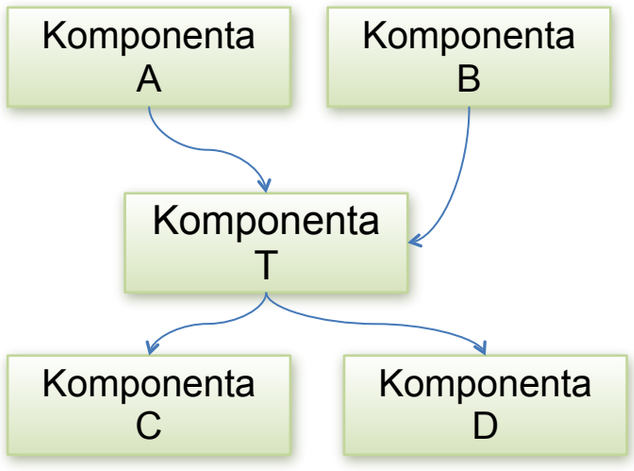
```
public String actuateLights(boolean motionDetected) {
    Calendar time = Calendar.getInstance(); // Ouch!

    // Update the time of last motion.
    if (motionDetected)
    {
        lastMotionTime = time;
    }

    // If motion was detected in the evening or at night, turn the light on.
    String timeOfDay = smartHouseSystem.getTimeOfDay(time);
    if (motionDetected && (timeOfDay == "Evening" || timeOfDay == "Night"))
    {
        return "ON";
    }
    // If no motion is detected for one minute, or if it is morning or day, turn the light off.
    else if ((time.getTimeInMillis() - lastMotionTime.getTimeInMillis()) > 60000 ||
            (timeOfDay == "Morning" || timeOfDay == "Noon"))
    {
        return "OFF";
    }

    return "OFF";
}
```

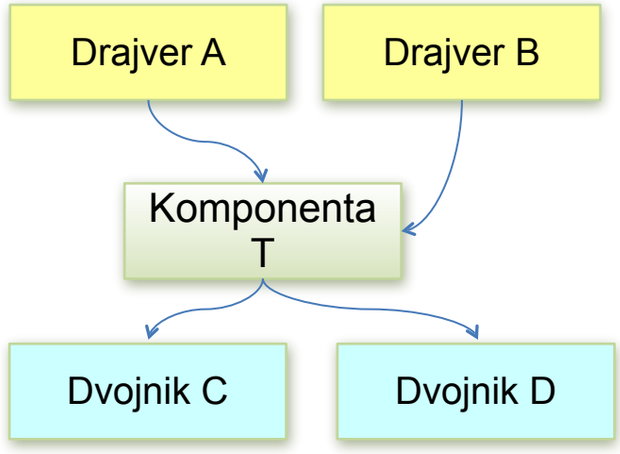
MOCKING

- 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

OKRUŽENJE ZA TESTIRANJE POJEDINAČNE KOMPONENTE

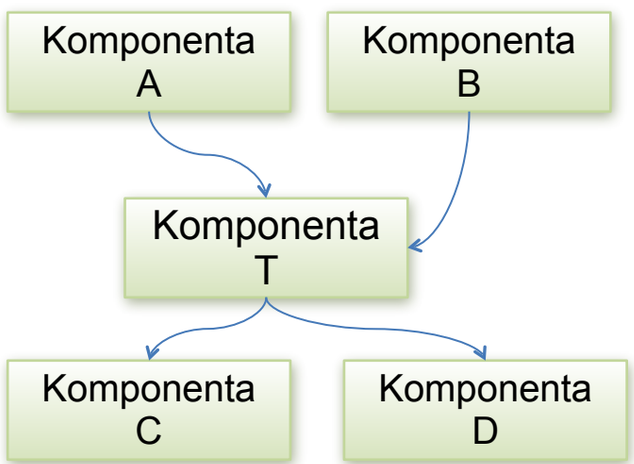
- Komponenta koja se testira se izvlači iz konteksta sistema u kome komunicira sa ostalim komponentama i ubacuje se u test kontekst koji je simulacija pravog sistema gde se nalaze komponente koje simuliraju rad stvarnih komponenti
- Primer je prikazan na slici – komponenta koja se testira T u realnom okruženju biva pozvana od strane komponenti A i B kojima pri pozivu vraća neke podatke i poziva komponente C i D koje joj daju informacije kada ih komponenta pozove

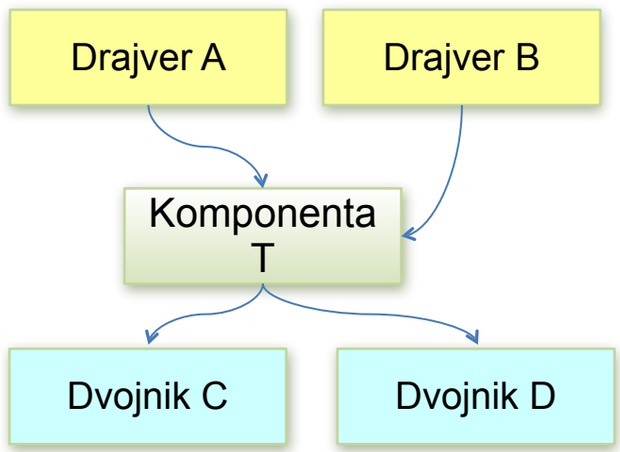




OKRUŽENJE ZA TESTIRANJE POJEDINAČNE KOMPONENTE

- Umesto sa realnim komponentama iz svog okruženja, komponenta u test kontekstu komunicira sa simuliranim komponentama koje je ili pozivaju ili joj odgovaraju na pozive na isti način kao i realne komponente. U zavisnosti od toga da li simulirane komponente pozivaju ili bivaju pozvane one se dele na dve vrste:
  - Pokretači (Drajveri) – komponente koje simuliraju rad realnih komponenti koje pozivaju druge komponente i očekuju neki odgovor. Ove komponente pokreću test pošto iniciraju pozive ka komponenti koja se testira. Za realizaciju drajvera često se koriste alati familije xUnit
  - Dvojnici (Doubles) – komponente koje simuliraju rad realnih komponenti koje primaju pozive i vraćaju iste rezultate kao i realne komponente





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

- Postoje tri glavna moguća tipa zamenskih objekata
  - FAKE objekti su objekti koji imaju realne implementacije, ali ne iste kao i objekti u produkciji. Obično koriste neku prečicu i imaju pojednostavljenu verziju produkcionog koda
  - STUB objekti su objekti koji sadrže unapred definisane podatke koji služe kao odgovori na pozive tokom testiranja. Koriste se u situacijama kada nije moguće ili nije poželjno uključiti objekte koji bi odgovarali stvarnim podacima ili imaju neželjene nuspojave
  - MOCK objekti predstavljaju sofisticiraniju verzija Stub-a. I dalje će vraćati vrednosti kao Stub, ali se takođe mogu biti programirati sa očekivanjima u smislu koliko puta treba pozvati svaki metod, kojim redosledom i sa kojim podacima

MOCKING FRAMEWORK

- Mocking radni okrviri se koriste za generisanje zamenskih objekata kao što su Stubs i Mock
- Njihova uloga jeste dopunjavanje testing radnih okvira kao što su JUnit ili TestNG izolovanjem zavisnosti. Izolujući zavisnosti, oni pomažu procesu jediničnog testiranja i pomažu programerima u pisanju fokusiranijih i sažetijih jediničnih testova. Testovi takođe rade brže tako što se deo celokupnog sistema izoluje iz testiranja
- **Mocking radni okviri nisu zamena za unit testing framework-e**
- Korišćenje mocking radnih okvira nije nužno
- Lažni objekti se mogu kreirati i održavati ručno, ali ovaj proces je dugotrajan i na kraju neproduktivan. Mocking radni okviri omogućavaju programerima da se usredsrede samo na testiranje sistema koji se testira. Lažni objekti se automatski kreiraju u memoriji kada se pokreću testovi na osnovu jednostavne konfiguracije
- U Java programskom jeziko postoji nekoliko mocking radnih okvira

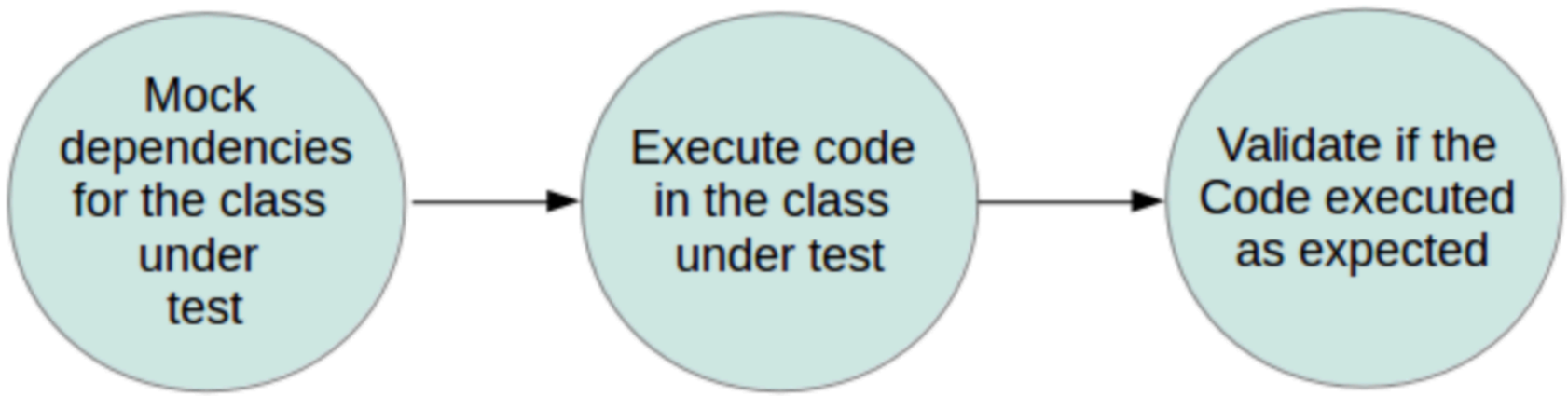
    1. JMockit
    2. EasyMock
    3. PowerMock
    4. Mockito

MOCKITO

- Mockito je Java mocking radni okvir
- Mockito koristi princip refleksije kako bi kreirao lažne objekte. Mock objekti se mogu posmatrati kao neka vrsta proksija ka pravim implementacijama objekata
- Prednosti korišćenja Mockita

1. Automatizam pri kreiranju lažnih objekata – nema potrebe za manuelnim pisanjem lažnih objekata
2. Bezbedan refaktoring – preimenovanje naziva metoda interfejsa ili promena redosleda parametara neće narušiti testni kod jer se lažni objekti kreiraju tokom izvršavanja
3. Podrška povratne vrednost
4. Podrška za izuzetke
5. Podrška za proveru redosleda poziva – omogućava praćenje poziva mokovanih objekata
6. Podrška za deklarativno programiranje – podržava kreiranje lažnih objekata putem anotacija

MOCKITO


Mock dependencies for the class under test
Execute code in the class under test
Validate if the Code executed as expected

KREIRANJE MOCK OBJEKTA

- Mock objekti se mogu kreirati na nekoliko načina:
    1. Pozivom `mock()` metode
        - `Mockito.mock()` omogućava kreiranje mock objekta nad klasom ili interfejsom
        - Kreiranje mock objekta pozivom metode ne zahteva nikakvu dodatnu konfiguraciju

```
PasswordEncoder mPasswordEncoder = Mockito.mock(PasswordEncoder.class);
```

KREIRANJE MOCK OBJEKTA

- Mock objekti se mogu kreirati na nekoliko načina:
    2. Pomoću `@Mock` anotacije
        - Predstavlja skraćeni zapis poziva mock() metode
        - Ovako definisanje mock objekte moguće je koristiti samo unutar test klasa
        - Korišćenje anotacije zahteva dodatnu konfiguraciju u vidu postavljanja `@RunWith(MockitoJUnitRunner.class)` anotacije iznad definicije test klase ili implicitnim pozivom `MockitoAnnotations.openMocks()` metode

```
@Mock
private PasswordEncoder mPasswordEncoder;

@BeforeMethod
public void setUp() throws Exception {

    MockitoAnnotations.openMocks(this);
}
```

KREIRANJE MOCK OBJEKTA

- Mock objekti se mogu kreirati na nekoliko načina:
    3. Pomoću `@MockBean` anotacije
        - Služi za mokovanje objekata u kontekstu Spring aplikacije
        - Lažni objekti će zameniti Bean objekte prave implementacije ukoliko ona postoji u kontekstu
        - Kako bi se omogućilo kreiranje objekata na ovaj način, neophodno je test klasu anotirati sa `@RunWith(SpringRunner.class)` anotacijom

POVRATNE VREDNOSTI MOCK OBJEKATA

- Nakon kreiranja lažnih metoda, moguće je njihovo pozivanje
- Međutim, postavlja se pitanja - šta su povratne vrednosti metoda mokovanih objekata?
- Po pravilu, metode mokovanjih objekata će vratiti “uninitialized” ili “empty” vrednosti

```
interface Demo {
   int getInt();
   Integer getInteger();
   double getDouble();
   boolean getBoolean();
   String getObject();
   Collection<String> getCollection();
   String[] getArray();
   Stream<?> getStream();
   Optional<?> getOptional();
}
```

```
Demo demo = mock(Demo.class);
assertEquals(0, demo.getInt());
assertEquals(0, demo.getInteger().intValue());
assertEquals(0d, demo.getDouble(), 0d);
assertFalse(demo.getBoolean());
assertNull(demo.getObject());
assertEquals(Collections.emptyList(), demo.getCollection());
assertNull(demo.getArray());
assertEquals(0L, demo.getStream().count());
assertFalse(demo.getOptional().isPresent());
```

Za više primera stubbinga pogledati src/test/java/rs/ac/uns/testing/user/MockitoCookbook.java

STUBBING

- Stubbing je proces je moguće izvršiti konfigurisanje mokovanih objekata tako što će se definisati ponašanje njenih metoda nakon poziva
- Mockito nudi dva načina stubbovanja metoda:

1. When Then pravila
   - *When this method is called, then do something*

```
when(passwordEncoder.encode("1")).thenReturn("a");
```

2. Do When pravila
   - *Do something when this mock's method is called with the following arguments*

```
doReturn("a").when(passwordEncoder).encode("1");
```

Za više primera argument matchera pogledati src/test/java/rs/ac/uns/testing/user/MockitoCookbookArgumentMatcher.java

ARGUMENT MATCHERS

- U prethodnim primerima mokovani objekti su stubbovani sa tačnim vrednostima kao argumentima. U tim slučajevima, Mockito interno poziva `equals()` kako bi proverio da li su očekivane vrednosti jednake stvarnim vrednostima
- Ponekad, međutim, ove vrednosti ne znamo unapred
- Ukoliko nije važno koja stvarna vrednost se prenosi kao argument ili ukoliko je potrebno definisati reakciju za širi opseg vrednosti moguće je koristiti argument matchers
- Ideja je jednostavna: umesto da se obezbedi tačna vrednost, definišu se AM sa kojima će Mockito porediti stvarne vrednosti
- Napomena: argument matchers ne mogu biti korišćeni kao povratne vrednosti stubbovanja

```
// when the password encoder is asked to encode any string, then return the string ‘exact.
when(mPasswordEncoder.encode(anyString())).thenReturn("exact");

assertEquals("exact", mPasswordEncoder.encode("1"));
assertEquals("exact", mPasswordEncoder.encode("abc"));
```

Za više primera argument matchera pogledati src/test/java/rs/ac/uns/testing/user/MockitoCookbookArgumentMatcher.java

ARGUMENT MATCHERS

- Ukoliko metoda koja treba biti stubbovana prima više od jednog parametra, Mockito ne dozvoljava da se za neke od njih prosleđuje tačna vrednost, a za neke argument matcheri
- Ovakvi pozivi izazvaće greške

```
abstract class AClass {
    public abstract boolean call(String s, int i);
}

AClass mock = Mockito.mock(AClass.class);
//This doesn't work.
when(mock.call("a", anyInt())).thenReturn(true);
```

- Ukoliko je potrebno neki od parametara porediti sa stvarnim vrednostima, neophodno je koristiti `eq()` metodu

```
when(mock.call(eq("a"), anyInt())).thenReturn(true);
```

Za više primera verifikacije pogledati src/test/java/rs/ac/uns/testing/user/MockitoCookbookVerify.java

VERIFY

- Mockito omogućava da nakon što je mock ili spy objekat korišćen potvrdimo da li je zaista i došlo do određenih interakcija
- U tu svrhu koristi se `verify()` metoda
- Verify po pravilu verifikuje pozivanje metode jednom, međutim moguće je proveriti i tačan broj poziva metode
  - `times(int wantedNumberOfInvocations)`
  - `atLeast( int wantedNumberOfInvocations )`
  - `atMost( int wantedNumberOfInvocations )`
  - `calls( int wantedNumberOfInvocations )`
  - `only( int wantedNumberOfInvocations )`
  - `atLeastOnce()`
  - `never()`
- Istom metodom je moguće proveriti i redosled interakcija sa lažnim objektima

```
PasswordEncoder mPasswordEncoder = mock(PasswordEncoder.class);

when(mPasswordEncoder.encode("a")).thenReturn("1");
mPasswordEncoder.encode("a");

verify(mPasswordEncoder).encode("a");
```

Za više primera verifikacije pogledati src/test/java/rs/ac/uns/testing/user/MockitoCookbookVerify.java

VERIFY

- Odsustvo poziva se takođe može proveriti pomoću `veriyZeroInteractions()` metode
  - Ovaj metod prihvata mock ili mocks kao argument i neće uspeti ukoliko je bilo poziva bilo koje metode koja se nalazi u prosleđenom mock objektu
- Takođe je vredno pomenuti metod `verifyNoMoreInteractions()`, koja prima mock objekte kao argumente i može se koristiti za proveru da li je svaki poziv na tim objektom verifikovan

SPY

- Spy je druga vrsta test dvojnika koje Mockito kreira. Za razliku od mock objekata, kreiranje spy objekta zahteva instancu pravog objekta
- Podrazumevano, spy delegira sve pozive metoda na pravi objekat i beleži koji je metod pozvan i sa kojim parametrima. To je ono što ga čini špijunom: špijunira pravi objekat
- Razmislite o korišćenju mock umesto spy kad god je to moguće. Spy objekti bi mogli biti korisni za testiranje zastarelog koda koji se ne može redizajnirati tako da se lako može testirati
- Praksa da se kreiraju spy objekti u cilju delimičnog mokovanja objekta je striktan pokazatelj narušavanja Single Responsibility principa (klasa ima previše metoda ili ima preveliku odgovornost i verovatno treba biti refaktorisana)

KREIRANJE SPY OBJEKATA

- Spy objekte je moguće kreirati na 3 načina:
    1. Pozivom `spy()` metode
        - Za razliku od `mock()` metode, statička `spy()` metoda kao argument prima instancu objekta kog špijunira
    2. Pomoću `@Spy` anotacije
    3. Pomoću `@SpyBean` anotacije
- Pozivanje metoda spy objekta će pozvati stvarne metode osim ako su te metode blokirane. Ovi pozivi se snimaju i činjenice ovih poziva mogu biti verifikovane

```
Demo sDemo = Mockito.spy(new Demo());

// if not stubbed, spy will call real method
assertEquals(1, sDemo.returnOne());

// spy objects can be verified
verify(sDemo).returnOne();

// stubbing spy
when(sDemo.returnOne()).thenReturn(5);
assertEquals(5, sDemo.returnOne());
```

DODATAK

PREPORUKE PISANJA TESTOVA

- Poželjno je testove nove funkcionalnosti pisati sledećim redom:

    1. **Granični slučaj** - započnite sa testom koji radi na „praznoj“ vrednosti poput nula, null, prazan niz ili slično. Pomoći će vam da zadovoljite interfejs osiguravajući pritom da se vrlo brzo može doneti.

    2. **Jedan ili nekoliko testova osnovnog uspešnog scenarija tzv. happy path tests** - Takav test / testovi postaviće temelje implementacije, dok ostaju fokusirani na osnovnu funkcionalnost.

    3. **Testovi koji pružaju nove informacije ili znanje** - Ne kopajte na jednom mestu. Pokušajte da pristupite rešenju iz različitih uglova tako što ćete napisati testove koji aktiviraju različite njegove delove i koji će vas naučiti nečemu novom o problemu.

    4. **Rukovanje greškama i negativni testovi** - Ovi testovi presudni su za ispravnost, ali retko za dizajn. U mnogim slučajevima mogu se bez opasnosti pisati na kraju.

IMENOVANJE TEST KLASA

- Test klase se imenuju na sledeći način:
  - **[ClassName]Tests**
  - UserServiceTests.java

IMENOVANJE TEST METODA

- Postoji veliki broj prihvaćenih naming strategija kada je u pitanju imenovanje test metoda
- Neke od najpopularnijih možete pročitati na linku
- Međutim, u poslednje vreme sve je popularniji novi vid imenovanja test metoda koji nalaže sledeća pravila:
    1. Ne koristite stroge politike imenovanja testova
    2. Imenujte testove kao da osobi, upoznatoj sa domenom problema opisujete problem (osobu ne smatramo programerom)
    3. Razdvojite reči u nazivu testa donjom crtom
    4. Nemojte uključivati naziv metode koja se testira u naziv testa
- Ideja potiče od Vladimira Khorikova

IMENOVANJE TEST METODA

```
[Fact]
public void IsDeliveryValid_InvalidDate_ReturnsFalse()
{
    DeliveryService sut = new DeliveryService();
    DateTime pastDate = DateTime.Now.AddDays(-1);
    Delivery delivery = new Delivery
    {
        Date = pastDate
    };

    bool isValid = sut.IsDeliveryValid(delivery);

    Assert.False(isValid);
}
```

- Test metoda je imenovana po strogoj konvenciji
  - public void IsDeliveryValid_InvalidDate_ReturnsFalse()
- Renaming1:
  - public void Delivery_with_invalid_date_should_be_considered_invalid()
  - Ime postaje čitljivije neprogramerima
  - Naziv metode je izbačen iz naziva testa
  - Problem?
    - Šta znači “InvalidDate”

IMENOVANJE TEST METODA

- Test metoda je imenovana po strogoj konvenciji
  - public void IsDeliveryValid_InvalidDate_ReturnsFalse()
- Renaming1:
  - public void Delivery_with_invalid_date_should_be_considered_invalid()
  - Ime postaje čitljivije neprogramerima
  - Naziv metode je izbačen iz naziva testa
    - Problem?
      - Šta znači “InvalidDate”
- Renaming2:
  - public void Delivery_with_past_date_should_be_considered_invalid()

Nepotrebna reč, njenim izbacivanjem se ne gubi značenje

- Renaming3:
  - public void Delivery_with_past_date_should_be_invalid()

Anti patternt - test je jedinstvena, atomska činjenica o jedinici ponašanja. Nema mesta željama kada se navode činjenice.

- Renaming4:
  - public void Delivery_with_past_date_is_invalid()

Gramatika engleskog jezika

- Renaming5:
  - public void Delivery_with_a_past_date_is_invalid()

Ispravna verzija

UNIT TESTOVI I PRIVATNE METODE

- Prebacivanje metoda iz private u public, samo zarad testiranja nije dozvoljeno i smatra se anti-patternom
- Privatne metode se mogu testirati kao deo metoda koje ih pozivaju. U tom slučaju bile bi deo integracionog testa
- Ukoliko je privatna metoda kompleksa, razmotrite da li je moguće odraditi refactoring

```
public class Order
{
    private Customer _customer;
    private List<Product> _products;

    public string GenerateDescription()
    {
        return $"Customer name: {_customer.Name}, " +
            $"total number of products: {_products.Count}, " +
            $"total price: {GetPrice()}";
    }

    private decimal GetPrice()
    {
        decimal basePrice = /* Calculate based on _products */;
        decimal discounts = /* Calculate based on _customer */;
        decimal taxes = /* Calculate based on _products */;
```

The complex private method is used by a much simpler public method.

Complex private method

```
public class Order
{
    private Customer _customer;
    private List<Product> _products;

    public string GenerateDescription()
    {
        var calc = new PriceCalculator();

        return $"Customer name: {_customer.Name}, " +
            $"total number of products: {_products.Count}, " +
            $"total price: {calc.Calculate(_customer, _products)}";
    }
}

public class PriceCalculator
{
    public decimal Calculate(Customer customer, List<Product> products)
    {
        decimal basePrice = /* Calculate based on products */;
        decimal discounts = /* Calculate based on customer */;
        decimal taxes = /* Calculate based on products */;
        return basePrice - discounts + taxes;
    }
}
```