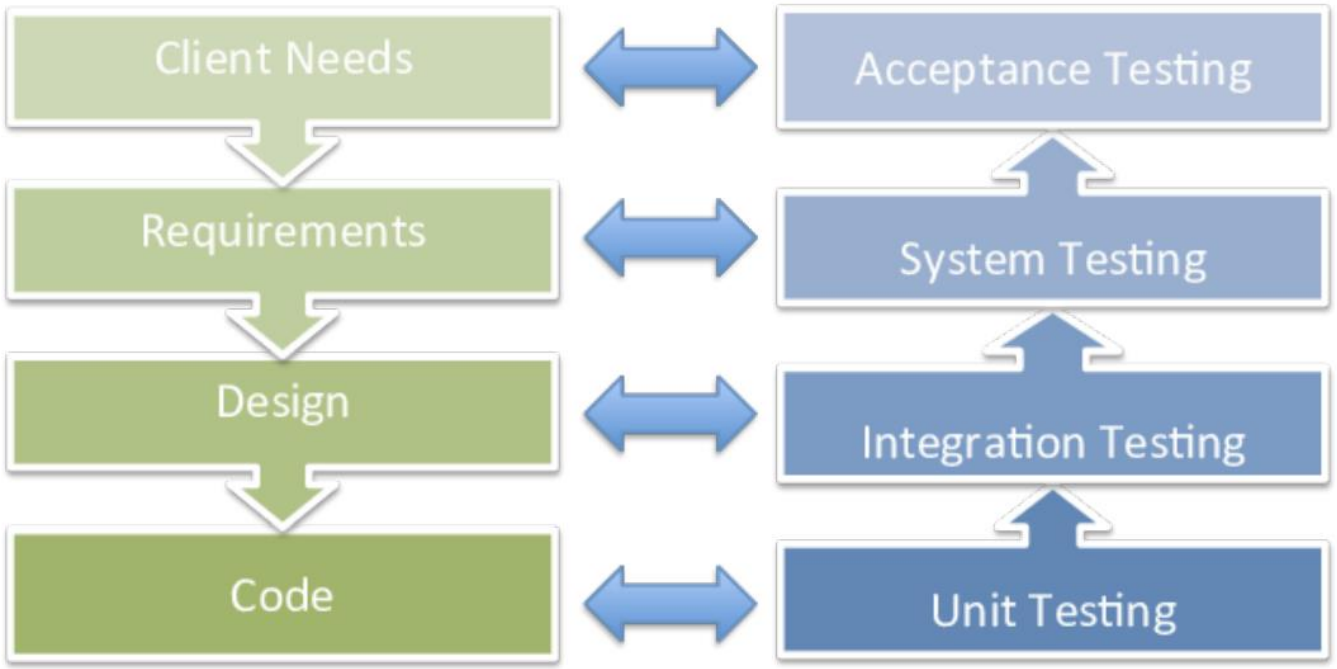
Testiranje

Nivoi testiranja

1. **Jedinično** (unit) testiranje
   - verifikuje ponašanje svake softverske komponente nezavisno od ostatka sistema
2. **Integraciono** (integrity) testiranje
   - verifikuje međusobnu interakciju skupa komponenti prema definisanom dizajnu
3. **Sistemsko** (system) testiranje
   - verifikuje kompletno integrisan sistem kako bi se utvrdilo da li odgovara unapred definisanoj specifikaciji
4. **Test prihvatljivosti** (acceptance)
   - utvrđuje da li sistem zadovoljava potrebe, zahteve i očekivanja naručioca

Nivoi testiranja



Jedinično testiranje

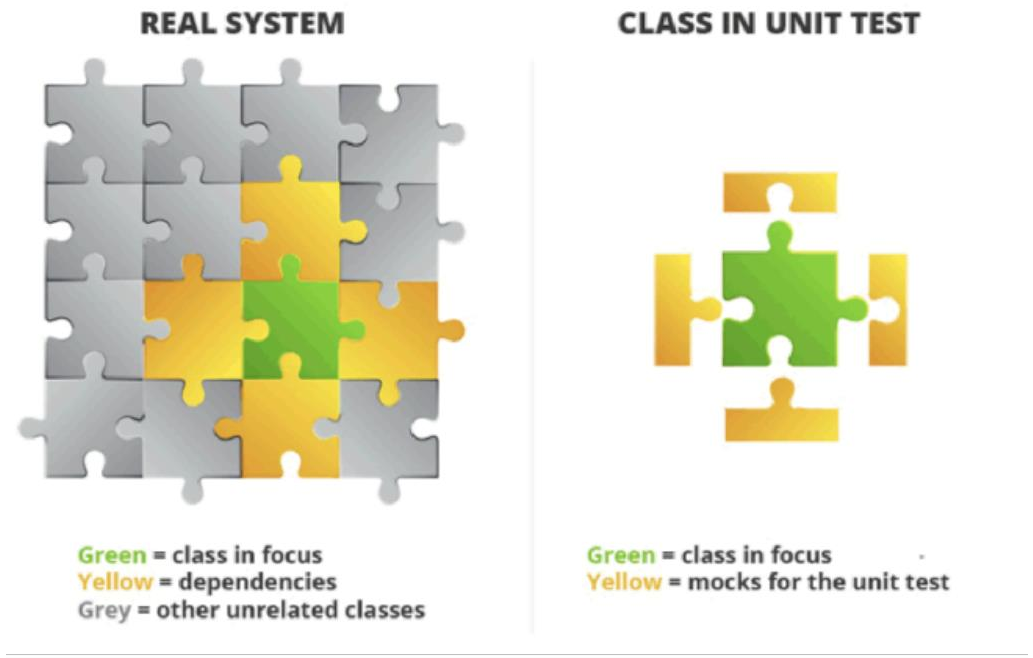
Jedinično testiranje

- Testiranje programskih komponenti **nezavisno** od ostatka sistema
- Osnovna karakterisitka: komponente se testiraju odvojeno i izolovano od ostatka sistema
- Izolacijom se isključuju spoljašnji uticaji drugih komponenti na komponentu koja se testira
- Detektovani problem onda definitivno ukazuje na nedostatak u testiranoj komponenti
- Komponenti koja se testira se prosleđuje **unapred definisani skup podataka**, na osnovu kojeg se posmatraju dobijeni rezultati i porede sa očekivanim

Test objekti

- Test objekti su kod ovog tipa testiranja pojedinačne softverske komponente, najčešće **klase**
- Da bi se testirani objekat testirao u izolaciji, važno je da referencirani objektni ne unose grešku, pa je zbog toga potrebno simulirati njihov rad
- **Mocking mehanizam** omogućuje simulaciju ponašanja objekata koje testirani objekat koristi
- Umesto pravih referenciranih objekata postavljaju se objektni dvojnici koji pojednostavljuju ili simuliraju ponašanje referenciranih objekata – **mock objekti**

Mocking mehanizam



JUnit

- Java *framework* predviđen za jedinično testiranje pojedinačnih Java klasa
- *Test-first* projektovanje - programer sam paralelno sa kodiranjem piše i testove
- Nudi različite metode za asertacije (tvrdnje)
- Integracija sa popularnim Java razvojim okruženjima: Eclipse, NetBeans, IntelliJ, JBuilder.

JUnit asertacije

- Služe za proveravanje rezultata testa
- Asertacija je deo koda u kojem se zahteva da određeni izlaz bude istinit
- Najčešći format naredbi za asertaciju:

```
assertEquals(expected, actual)
```

očekivana vrednost

stvarna vrednost koja se računa u kodu koji se testira

- Ako ne dođe do poklapanja, *assert* metoda baca izuzetak `java.lang.AssertionError` i prekida se izvršavanje tekućeg testa, ali ne i ostalih

JUnit asertacije

- `assertEquals(expected, actual)`
- `assertFalse(condition)`
- `assertTrue(condition)`
- `assertSame(expected, actual)`
- `assertNotSame()`
- `assertThat(T actual, Matcher<? super T> matcher)`
- Kompletna specifikacija metoda za asertaciju: https://junit.org/junit4/javadoc/latest/org/junit/Assert.html

JUnit anotacije

- **@Test –** označava test metodu
- **@BeforeClass -** označava metodu koja se poziva pre prvog testa neke klase
- **@AfterClass -** označava metodu koja se poziva nakon izvršavanja svih testova neke klase
- **@BeforeMethod -** označava metodu koja se poziva pre svake test metode
- **@AfterMethod -** označava metodu koja se poziva pre svake test metode

Mock objekat

- Simulira ponašanje stvarnog objekta
- U potpunosti zamenjuje pravi objekat
- Sami izlazi koji su rezultat ponašanja su i dalje simulirani
- Koristi se kada imamo instancu kompleksne klase koja koristi eksterne resurse poput mreže, fajlova, baze podataka ili referencira dosta drugih objekata

Spy objekat

- 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 se može lažirati ponašanje kompleksnih metoda, a one jednostavne koje nema smisla lažirati 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

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

Mockito mock I spy objekti

- **Mock**
  - podrazumevano ponašanje za metode koje nisu lažirane jeste da ne rade ništa
  - metoda koja nema povratni tip podataka *(void)* neće raditi ništa
  - metoda koja ima povratnu vrednost će vraćati *null, empty, nulu* ili *default* vrednost

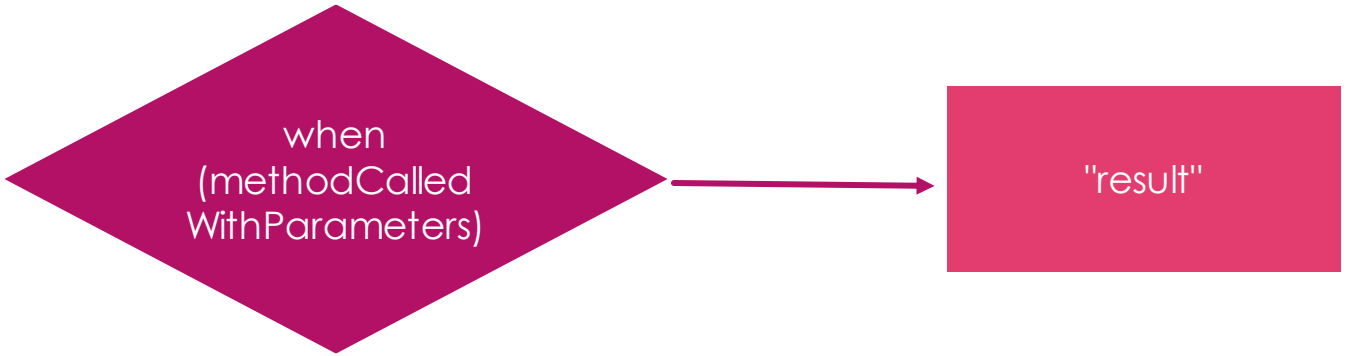
```
MyList listMock = Mockito.mock(MyList.class);
```

- **Spy**
  - ukoliko metoda nije lažirana, njen poziv će biti preusmeren na pravu 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 povratne vrednosti *mock* objekta

```
// kreiranje mock objekta
MyClass test = mock(MyClass.class);

// definisanje povratne vrednosti metode getUniqueId()
when(test.getUniqueId()).thenReturn(41);

// potreba mock objekta u testu
assertEquals(test.getUniqueId(), 41);
```

alternativa:

```
doReturn(41).when(test).getUniqueId();
```

Mockito when/then pravila

- Konfigurisanje *mock* objekta da baca izuzetak na poziv metode

```
// kreiranje mock objekta
MyClass test = mock(MyClass.class);

// definisanje izuzetka koji se baca baca prilikom poziva metode getUniqueId()
when(test.getUniqueId()).thenThrow(NullPointerException.class);
```

ukoliko je metoda void:

```
doThrow(NullPointerException.class).when(test).getUniqueId();
```

Mockito when/then pravila

- Poziv metode bez mokovanja vrednosti

```
// kreiranje mock objekta
MyClass test = mock(MyClass.class);

when(test.getUniqueId()).thenCallRealMethod();
```

Mockito verify

- Za testiranje interakcije za *mock* objektom

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 da li je nad mock objektom pozvana metoda .size() tačno 2 puta
verify(mockedList, times(2)).size();
```

Mockito verify

- Provera interakcije sa mock objektom

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

// provera da nije bilo interakcije sa objektom
verifyZeroInteractions(mockedList);

// provera da nije bilo poziva neke konkretne metode
verify(mockedList, times(0)).size();
verify(mockedList, never()).size();
```

Mockito verify

- Provera da se interakcije desila bar/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();
```

Integraciono tesiranje

Integraciono tesiranje

- Jedinično testiranje osigurava da su sve pojedinačne komponente softverskog sistema testirane i da svaka od njih pojedinačno radi ispravno
- Jedinično testiranje ipak ne garantuje da li će ove komponente raditi ispravno kada se integrišu u sistem
- Neke greške se javljaju tek kada se komponente spoje. Integraciono testiranje je faza u testiranju softvera u kojoj se pojedinačne komponente softverskog sistema kombinuju i testiraju kao grupa i tako se otkrivaju greške

Testiranje Spring aplikacija

Testiranje Spring i SpringBoot aplikacija

- https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#testing-introduction
- https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html

Kreiranje testova

- Kreirati klasu koju je potrebno anotirati sa:
  1. `@RunWith(SpringRunner.class)`
  2. `@SpringBootTest`

} → Da bi se Spring *web* aplikacija uspešno inicijalizovala i da bi se *ApplicationContext* kreirao (omogućava da se bez problema koriste Spring komponente)

- Test metodu anotirati sa `@Test` anotacijom
  - naznačava Springu da će se anotirana metoda izvrši prilikom testiranja
  - ukoliko se izostavi, test metoda se neće izvršiti

→ **mock web environment**

Kreiranje testova

- Test metoda ne sme da vrši trajne izmene nad bazom podataka
- Test metode anotirane sa `@Transactional`:
  - za svaku test metodu se pokreće nova transakcija
  - transakcija se automatski poništava *(rollback)* na kraju metode
  - izmene nad bazom se automatski poništavaju nakon metode
  - iako je *rollback* podrazumevano ponašanje, može da se stavi i anotacija `@Rollback(true)` da bi bilo čitljivije

Kreiranje negativnih testova

- Test može da definiše da očekuje da se pri pozivu metode desi određeni izuzetak. Test prolazi ako se takav izuzetak desi
- Ovo se koristi kada je potrebno verifikovati ponašanje na nevalidne ulaze
- `@Test(expected = SomeException.class)`

Jedinično testiranje

- **JUnit**, **Mockito**
- `@Mock`
  - navodi se iznad atributa klase koji se testira
  - alternativa: poziv `Mockito.mock(Object o)` metode
- `@InjectMocks`
  - ukoliko komponenta koja se mokuje ima reference ka drugim komponentama, neophodno je kreirati *mock* objekte za svaku zavisnu komponentu
  - injektuje mokovane komponente

Ukoliko objekat koji se mokuje ima zavisnosti koje je takođe neophodno mokovati, ne može se koristiti ova alternativa!!!

Integraciono testiranje

- **MockMVC**
- Programski poziv REST servisa se vrši putem *Spring MockMvc* klase
- *MockMvc* simulira kompletnu Spring *web* MVC arhitekturu
- Nije *mock* objekat u ranije korišćenom značenju!
- Omogućuje stvarno, a ne lažno predefinisano ponašanje kao kod *mock* objekata koje koristi *Mockito*

Integraciono testiranje

```
@Autowired
private WebApplicationContext webApplicationContext;
```

Omogućava pristup konfiguraciji *web* aplikacije. Učitava sve Spring komponente i kontrolere

```
@Before
public void setup() {
    this.mockMvc =
    MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
}
```

Inicijalizacija mockMvc objekta u metodi anotiranoj sa **@Before** da se ova inicijalizacija ne bi radila u telu svake test metode

Verifikacija HTTP odgovora

- URL: *http://localhost:8080/test/hello*
- Očekivani odgovor: `{ message: "Hello World!" }`
- Kod test metode:

```
@Test
public void getMethodTest() throws Exception {
    MvcResult mvcResult = mockMvc.perform(get(url))
      .andExpect(status().isOk())
      .andExpect(jsonPath("$.message").value("Hello World!"))
      .andReturn();

      Assert.assertEquals("application/json;charset=UTF-8", mvcResult.getResponse().getContentType());
}
```

Verifikacija HTTP odgovora

- **andExpect(MockMvcResultMatchers.status().isOk())**:

  verifikuje da je HTTP status kod *200 Success*

- **andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!"))**:

  verifikuje da objekat iz tela HTTP odgovora sadrži atribut *message* koji ima vrednost "*Hello World!*". Koristi se jsonPath izraz

- **andReturn()**:

  vraća *MvcResult* objekat koji se koristi za verifikaciju nečega što nije deo biblioteke

jsonPath

- jsonPath je jezik za opis sadržaja JSON objekta
- Omogućuje selekciju delova JSON objekta
- Ima istu ulogu kao XPath za XML
- http://goessner.net/articles/JsonPath/
- **$** se odnosi na ceo JSON objekat
- Ukoliko je odgovor neka lista JSON objekata, sa **$.[*]** se pristupa bilo kojem elementu liste. Mogu se prosleđivati i konkretni indeksi umesto *