TESTIRANJE SOFTVERA

Arhitektura testabilnog koda i napredne tehnike izolacije (Mocking)

DIZAJN ZA TESTABILNOST

TEST DOUBLES

OVER-MOCKING

Ciljevi predavanja i agenda

Čvrsto spregnut kod

Prepoznavanje problema direktnih zavisnosti od baze i eksternih servisa.

Dizajn za testabilnost

SOLID, DIP, IoC i ubrizgavanje zavisnosti kroz konstruktor.

Pet vrsta dvojnika

Dummy, Stub, Spy, Mock i Fake — svrha i upotreba.

Praktična realizacija

Java, Maven, JUnit 5 i Javalin kontroler.

Rizici

Over-mocking i krhki testovi (Fragile Tests).

Problem: čvrsto spregnut kod (Tightly Coupled Code)

Šta je problem?

Komponente sistema direktno zavise od konkretnih implementacija drugih komponenti.

Primer: poslovna logika neposredno poziva bazu podataka ili eksterni API.



Posledica: biznis logiku nije moguće testirati u izolaciji od spoljnog okruženja.

Primer loše prakse: zavisnost kreirana unutar klase

```
// Loša praksa: klasa sama kreira zavisnost
public class KorisnikServis {
    private UkrstenaBazaPodataka baza =
        new UkrstenaBazaPodataka(); // čvrsta sprega

    public void registrujKorisnika(Korisnik k) {
        if (k.isValidan()) {
            baza.sacuvaj(k);
        }
    }
}
```

Konkretna klasa

Servis direktno poznaje implementaciju baze.

Bez ubrizgavanja

Nema mesta na koje test može da ubaci dvojnika.

Ovakva zavisnost otežava zamenu baze lažnom komponentom u testu.

Zašto je ovakav kod teško testirati?

IZOLACIJA

Nedostatak izolacije

Test za KorisnikServis zavisi od toga da li je baza podignuta i da li su podaci u njoj konzistentni.

BRZINA

Brzina izvršavanja

Testovi koji komuniciraju sa bazom podataka sporiji su od provera koje ne zavise od nje.

NUSPOJAVE

Nuspojave

Testovi mogu menjati stvarne podatke u bazi i uticati jedni na druge.

Zavisnost od infrastrukture preliva se direktno u testove poslovne logike.

Koncept: dizajn za testabilnost (DfT)

Definicija

Pristup dizajnu softvera u kome se mogućnost lakog testiranja postavlja kao primarni arhitektonski zahtev.

ZLATNO PRAVILO

Kod koji se lako testira obično je i dobro dizajniran.

Kako to postižemo?

SOLID principi

Fleksibilnost, modularnost i proširivost.

Inverzija kontrole

Zavisnosti se uvode spolja, umesto da ih klasa sama kreira.

SOLID principi kroz prizmu testova

JEDNA ODGOVORNOST

S — Single Responsibility

Klasa radi samo jednu stvar.

Lakše se testira jer ima manje scenarija koje treba proveriti.

PROŠIRIVANJE

O — Open/Closed

Klasa je otvorena za proširenje.

Testovi ostaju netaknuti kada se dodaju nove funkcionalnosti.

APSTRAKCIJE

D — Dependency Inversion

Moduli zavise od apstrakcija.

Ključni princip za uvođenje testnih dvojnika.

U ovom kontekstu, DIP omogućava zamenu produkcione zavisnosti dvojnikom u testu.

Dependency Inversion Principle (DIP)

Moduli višeg nivoa ne treba da zavise od modula nižeg nivoa.
Oba treba da zavise od apstrakcija.



KorisnikServis ne zavisi od UkrstenaBazaPodataka, već od interfejsa KorisnikRepozitorijum.

Inversion of Control (IoC) i Dependency Injection (DI)

KONTROLA

Inversion of Control (IoC)

Klasi se oduzima pravo da sama upravlja životnim ciklusom svojih zavisnosti.

UBRIZGAVANJE

Dependency Injection (DI)

Zavisnosti se ubrizgavaju spolja — najčešće kroz konstruktor.

U testu: umesto prave baze ubrizgavamo lažnu komponentu (dvojnika).

Refaktorisan primer: uvođenje DI (Java)

```java
// 1. Apstrakcija
public interface KorisnikRepozitorijum {
    void sacuvaj(Korisnik k);
}

// 2. Constructor Injection
public class KorisnikServis {
    private final KorisnikRepozitorijum repozitorijum;

    public KorisnikServis(KorisnikRepozitorijum repozitorijum) {
        this.repozitorijum = repozitorijum;
    }

    public void registrujKorisnika(Korisnik k) {
        if (k.isValidan()) repozitorijum.sacuvaj(k);
    }
}
```

Interfejs

Opisuje potrebnu operaciju, bez detalja implementacije.

Konstruktor

Prima konkretnu zavisnost spolja.

Ista tačka ubrizgavanja koristi se i u testu.

Šta su dvojnici testova (Test Doubles)?

METAFORA

Kao što kaskaderi menjaju glumce u zahtevnim scenama, testni dvojnici menjaju produkcione komponente u testovima.

Svrha

Izolacija jedinice koda koja se testira (Unit under test — UUT).

Taksonomija

Prema Gerardu Meszarosu razlikuje se pet tipova dvojnika.

Zavisnost se zamenjuje komponentom prilagođenom cilju testa.

Pregled taksonomije Gerarda Meszarosa

Tip dvojnika	Primarna svrha / karakteristika
Dummy	Popunjava parametar; nikada se ne poziva.
Stub	Vraća unapred definisane odgovore na pozive.
Spy	Beleži broj poziva i prosleđene argumente.
Mock	Ima unapred programirana očekivanja i sam verifikuje ponašanje.
Fake	Funkcionalna, pojednostavljena implementacija — npr. in-memory.

1. Dummy objekti

Šta rade?

Prosleđuju se komponenti, ali se ne koriste u metodi koja se testira.

Često služe da zadovolje potpis konstruktora ili metode.

Validacija imena pada pre upotrebe zavisnosti.

```
@Test
void registracijaBacaIzuzetakZaPraznoIme() {
    KorisnikRepozitorijum dummyRepo = null;
    EmailServis dummyEmail = email -> fail("Ne sme biti pozvan");
    KorisnikServis servis =
        new KorisnikServis(dummyRepo, dummyEmail);

    Korisnik nevalidan = new Korisnik("", "test@email.com");
    assertThrows(IllegalArgumentException.class,
        () -> servis.registrujKorisnika(nevalidan));
}
```

2. Stub — dvojnik sa odgovorom

Uloga Stub-a

Vraća unapred definisane podatke kako bi usmerio tok izvršavanja koda pod testom.

Ne prati koliko puta je pozvan niti sa kojim parametrima.

```
class KorisnikRepozitorijumStub
        implements KorisnikRepozitorijum {
    @Override
    public boolean postojiPoEmailu(String email) {
        return true; // fiksni odgovor
    }

    @Override public void sacuvaj(Korisnik k) {}
    @Override public int prebrojKorisnike() { return 0; }
}
```

Ovaj Stub simulira slučaj u kome email već postoji.

Praktičan test sa Stub-om (JUnit)

```
@Test
void registracijaPropadaAkoEmailVecPostoji() {
    // Arrange
    KorisnikRepozitorijum stubRepo =
        new KorisnikRepozitorijumStub();
    EmailServis dummyEmail = email -> {};
    KorisnikServis servis =
        new KorisnikServis(stubRepo, dummyEmail);
    Korisnik korisnik =
        new Korisnik("Petar", "postojeci@email.com");

    // Act + Assert
    assertThrows(IllegalStateException.class,
        () -> servis.registrujKorisnika(korisnik));
}
```

Arrange

Pripremamo Stub koji uvek vraća true.

Assert

Očekujemo IllegalStateException jer email već postoji.

3. Spy — špijun

Šta pamti?

Pored vraćanja vrednosti, beleži istoriju poziva:

- koje metode su pozvane
- koliko puta
- sa kojim argumentima

```
class EmailServisSpy implements EmailServis {
    int brojPoziva = 0;
    String poslednjiEmail = null;

    @Override
    public void posaljiPozdravniEmail(String email) {
        brojPoziva++;
        poslednjiEmail = email;
    }
}
```

Spy izlaže zabeleženo stanje koje test proverava nakon izvršavanja.

Praktičan test sa Spy-em (JUnit)

```
@Test
void uspesnaRegistracijaSaljeJedanEmail() {
    FakeKorisnikRepozitorijum fakeRepo =
        new FakeKorisnikRepozitorijum();
    EmailServisSpy spyEmail = new EmailServisSpy();
    KorisnikServis servis =
        new KorisnikServis(fakeRepo, spyEmail);
    Korisnik novi = new Korisnik("Marko", "marko@email.com");

    servis.registrujKorisnika(novi);

    assertEquals(1, spyEmail.brojPoziva);
    assertEquals("marko@email.com", spyEmail.poslednjiEmail);
}
```

Provera 1

brojPoziva == 1

Provera 2

poslednjiEmail == "marko@email.com"

4. Mock — programirani dvojnik

Teorija

Pre izvršavanja testa programiraju se očekivanja.

Primer: metoda X treba da bude pozvana tačno jednom sa parametrom Y.

Mock na kraju sam verifikuje ponašanje.

Spy naspram Mock-a

Spy: test naknadno proverava zabeleženo stanje pomoću assertEquals.

Mock: očekivanja su deo definicije samog objekta.

Razlika je u mestu i načinu verifikacije očekivane interakcije.

Praktičan primer Mock-a (ručna JUnit varijanta)

```
class KorisnikRepozitorijumMock
        implements KorisnikRepozitorijum {
    private boolean sacuvajPozvan = false;

    @Override
    public void sacuvaj(Korisnik k) {
        sacuvajPozvan = true;
    }

    public void verifikuj() {
        if (!sacuvajPozvan) {
            throw new AssertionError(
                "Metoda sacuvaj() nije pozvana!");
        }
    }
}
```

Pre poziva

Test može pripremiti očekivanja.

Posle poziva

verifikuj() prijavljuje ako očekivanje nije ispunjeno.

5. Fake — funkcionalan, pojednostavljen dvojnik

Teorija

Fake ima stvarnu funkcionalnu implementaciju poslovne logike, ali koristi prečicu koja nije pogodna za produkciju.

HashMap umesto prave baze podataka

```
class FakeKorisnikRepozitorijum
        implements KorisnikRepozitorijum {
    private final Map<Long, Korisnik> podaci =
        new HashMap<>();
    private long idCounter = 1;

    @Override
    public void sacuvaj(Korisnik k) {
        podaci.put(idCounter++, k);
    }
}
```

State Verification — verifikacija stanja

FOKUS

ŠTA je rezultat?

Proverava se konačno stanje sistema ili vraćena vrednost nakon operacije.

Primer

```
assertEquals("Uspešno", rezultat);
```

Primer

```
assertEquals(1, fakeBaza.prebrojKorisnike());
```

Koristi se uz Stub-ove i Fake-ove.

Behavior Verification — verifikacija ponašanja

FOKUS

KAKO je rezultat postignut?

Proverava koje su metode eksternih zavisnosti pozvane tokom izvršavanja.

Primer

Da li je EmailServis pozvan tačno jednom?

Provera

mockEmailServis.verifikujDaJePoslatMejl(email)

Koristi se uz Mock-ove i Spy-eve.

Poređenje: State Verification i Behavior Verification

Pristup	Šta proverava	Primer	Tipični dvojnici
State Verification	Konačno stanje / vraćena vrednost	assertEquals(1, fakeBaza.prebrojKorisnike())	Stub, Fake
Behavior Verification	Pozivi i interakcije sa zavisnostima	EmailServis je pozvan tačno jednom	Spy, Mock

State = rezultat operacije • Behavior = interakcija tokom operacije

Postavka projekta: Maven

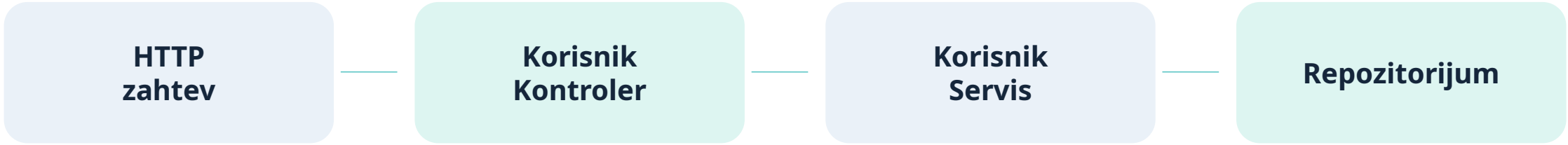
STRUKTURA PROJEKTA

```
kurs-testiranje/
├── pom.xml
└── src/
    ├── main/java/com/kurs/
    │   ├── App.java
    │   ├── model/
    │   ├── repo/
    │   ├── servis/
    │   └── kontroler/
    └── test/java/com/kurs/
        ├── dft/
        └── kontroler/
```

Komponenta	Verzija / uloga
Java	17
Javalin	6.1.3
Jackson databind	2.17.0
SLF4J Simple	2.0.12
JUnit Jupiter API + Engine	5.10.2
Maven Surefire	3.2.5

Zavisnosti se definišu u pom.xml.

Javalin arhitektura: web sloj i konzolni ispis



Uloga kontrolera

Prima HTTP zahtev, delegira posao servisu i pakuje HTTP odgovor.

Granica za spoljni servis

EmailServis je u App.java realizovan konzolnim ispisom: System.out.println("Saljem email na: " + email).

Kontroler ne treba da sadrži biznis logiku niti direktne upite ka bazi.

Javalin kontroler sa Dependency Injection (Java)

```
public class KorisnikKontroler {
    private final KorisnikServis korisnikServis;

    public KorisnikKontroler(KorisnikServis servis) {
        this.korisnikServis = servis;
    }

    public void registracijaRuta(Context ctx) {
        try {
            Korisnik k = ctx.bodyAsClass(Korisnik.class);
            korisnikServis.registrujKorisnika(k);
            ctx.status(201).result("Korisnik uspesno kreiran");
        } catch (IllegalArgumentException e) {
            ctx.status(400).result(e.getMessage());
        } catch (IllegalStateException e) {
            ctx.status(409).result(e.getMessage());
        }
    }
}
```

Konstruktor

Servis se ubrizgava spolja.

201

Uspešno kreiran korisnik.

400 / 409

Nevalidno ime / postojeći email.

Kako testirati Javalin kontroler?

Javalin Context

Objekat koji predstavlja HTTP zahtev i odgovor.

Kontroler kroz Context čita telo zahteva i postavlja status i rezultat.

Cilj izolacije

Testirati KorisnikKontroler bez podizanja stvarnog servera na portu 8080.

Napraviti dvojnika (Mock / Spy) za Javalin Context.

Testiranje Javalin kontrolera u izolaciji (JUnit)

```
@Test
void kontrolerVracaStatus201ZaUspesnuRegistraciju() {
    KorisnikServis fakeServis = new KorisnikServis(
        new FakeKorisnikRepozitorijum());
    KorisnikKontroler kontroler =
        new KorisnikKontroler(fakeServis);

    // Ručni Spy za Javalin Context – skraćeni prikaz
    ContextMock ctx = new ContextMock();
    ctx.PostaviBody("{\"ime\":\"Petar\"}");

    kontroler.registracijaRuta(ctx);

    assertEquals(201, ctx.DohvatiStatus());
}
```

Arrange

Pripremaju se servis, kontroler i ContextMock.

Act

Poziva se registracijaRuta(ctx).

Assert

Očekuje se status 201.

Šta je Over-mocking antipattern?

Preterano mock-ovanje svake zavisnosti i kreiranje prekomerno komplikovanih očekivanja.

Simptom

Testira se detalj implementacije umesto stvarnog ponašanja koda.

Šta se gubi?

Test postaje tesno vezan za način na koji je kod trenutno napisan.

Posledice Over-mocking-a

INTEGRACIJA

Gubitak integracione vrednosti

Svi unit testovi mogu prolaziti, a aplikacija ipak pada kada se pokrene jer stvarne komponente ne sarađuju dobro u produkciji.

POKRIVENOST

Lažni osećaj sigurnosti

Visoka pokrivenost koda ne znači da aplikacija nema fundamentalne bagove.

Izolovani testovi ne zamenjuju proveru saradnje stvarnih komponenti.

Fragile Test antipattern — krhki testovi

Definicija

Test pada kada se promeni interna implementacija, iako su krajnji izlaz i ponašanje aplikacije ostali isti.

```
// Implementacija A
baza.nadji();

// Posle refaktorisanja
baza.pretraziPoId();
```

Test koji strogo proverava naziv interne metode pada posle refaktorisanja.

Posledica: promena implementacije zahteva nepotrebno prepisivanje testova.

Kako izbeći krhke testove?

1

Prednost dajte verifikaciji stanja

Kad god je moguće, proveravajte rezultat umesto internih poziva.

2

Mock-ujte granice sistema

Eksterni API, slanje email-a i fajl sistem.

3

Ne mock-ujte čiste podatke

Value Objects i Entities ne treba zamenjivati mock objektima.

Zaključak i rezime

Dizajn za testabilnost

Preduslov za održiv sistem.

Test Doubles

Moćan alat koji zahteva disciplinu.

Stub i Fake

Za stabilne testove zasnovane na podacima.

Mock i Spy

Oprezno — za protokol komunikacije između komponenti.

Test treba da bude dokumentacija ponašanja, a ne čuvar strukture koda.