Laboratorijska vježba 5.

Razvoj programskih rješenja vođen testovima

5.1. Kreiranje klasa na osnovu inicijalizacijskih testova

U okviru prethodne laboratorijske vježbe prikazan je način na koji se može izvršiti provjera ispravnosti definisanog programskog koda u okviru kreiranih klasnih sistema. U tom kontekstu, podrazumijevano je da se prvo vrši razvoj programskog koda, a zatim testova koji koriste definisane klase i njihove metode. Međutim, ovo nije jedini način za razvoj programskih rješenja. Jedan od mogućih načina podrazumijeva razvoj programskih rješenja vođen testovima (*test-driven development* - **TDD**), u okviru kojeg se prvo vrši razvoj testova koji opisuju očekivane ulazne i izlazne vrijednosti, a zatim razvoj klasa koje omogućavaju ostvarivanje datih funkcionalnosti. Na ovaj način omogućava se lakše razumijevanje šta određene klase i metode trebaju da rade i izbjegava nagomilavanje programskog koda za situacije u kojima kompleksna programska logika nije potrebna.

Kako bi se demonstriralo korištenje TDD principa, upotrijebiti će se jedna testna klasa koja koristi sve principe objektno-orijentisanog programiranja iz prethodnih laboratorijskih vježbi. Na samom početku vrši se inicijalizacija statičkih objekata koji će se koristiti u okviru testova, na način prikazan u Listingu 1. Iz definisanog programskog koda testne klase mogu se izvući sljedeće informacije:

- Klasni sistem se sastoji od tri klase: **Pacijent**, **Doktor** i **MedicinskaSestra**;
- Klasa *Pacijent* ima minimalno 5 atributa koji se koriste za inicijalizaciju u okviru konstruktora, pri čemu se na osnovu proslijeđenih vrijednosti parametara može zaključiti da se radi o imenu i prezimenu, dijagnozi, datumu rođenja, numeričkom parametru koji vjerovatno predstavlja neke novčane prihode, i jedan *Boolean* parametar s trenutno nepoznatom svrhom;
- Klasa *Doktor* ima minimalno 4 atributa koji se koriste za inicijalizaciju u okviru konstruktora, pri čemu se na osnovu proslijeđenih vrijednosti parametara može zaključiti da se radi o imenu i prezimenu, nekom datumu (koji vjerovatno nije datum rođenja, jer je specificirani datum preblizu u prošlosti - 01/01/2018), vrsti zaposlenja koja se definiše enumeracijskim tipom **Zaposlenje** i nekim numeričkim parametrom s trenutno nepoznatom svrhom;

- Klasa *MedicinskaSestra* također ima minimalno 4 atributa koji se koriste za inicijalizaciju u okviru konstruktora, pri čemu se na osnovu proslijeđenih vrijednosti parametara može zaključiti da se vjerovatno radi o istim parametrima za klasu *Doktor* - ime i prezime, neki datum i enumeracijski tip *Zaposlenje*, a četvrti parametar je također numerički, ali cjelobrojni;
- Pri inicijalizaciji može doći do pojave izuzetaka, što vjerovatno ukazuje na to da se koristi validacija parametara u konstruktorima.

```
public class MainTest {
    static Pacijent pacijent1, pacijent2;
    static Doktor doktor;
    static MedicinskaSestra medicinskaSestra;

    @BeforeAll
    public static void Inicijalizacija ()
    {
        try {
            pacijent1 = new Pacijent("Pacijent 1", "Prehlada", new
Date(90, 1, 1), 1000.0, false);
            pacijent2 = new Pacijent("Pacijent 2", "Gripa", new
Date(55, 7, 20), 500.0, true);
            doktor = new Doktor("Doktor", new Date(118, 0, 1),
Zaposlenje.ZaposlenNeodredjeno, 7.0);
            medicinskaSestra = new MedicinskaSestra("Medicinska
sestra", new Date(120, 4, 20), Zaposlenje.Zaposlen20Posto, 300);
        }
        catch (Exception ex)
        {
            assertFalse(true);
        }
    }
}
```

Listing 1. Inicijalizacijska metoda i atributi testne klase

Ovako definisane varijable testne klase i inicijalizacijska *@BeforeAll* metoda omogućavaju da se definišu tri klase i jedan enumeracijski tip sa prethodno opisanim atributima. U ovom trenutku zbog istovjetnosti određenih parametara konstruktora za klase *Doktor* i *MedicinskaSestra* već se može pretpostaviti da će vjerovatno biti korišteno nasljeđivanje, međutim na osnovu inicijalizacijske testne metode još uvijek se ne može zaključiti tačno na koji način. Iz tog načina, potrebno je izvršiti analizu prve testne metode *TestValidacija()*, koja je definisana u Listingu 2. Ova metoda sadrži sintaksu koja nije korištena u prethodnoj laboratorijskoj vježbi, a odnosi se na korištenje metode **assertThrows()**, koja kao parametre prima klasu izuzetka čije se pojavljivanje želi validirati i lambda izraz koji sadrži programski kod za koji se očekuje da će dovesti do pojavljivanja datog izuzetka. Na osnovu testa, može se doći do sljedećih zaključaka:

- U klasnom sistemu postoje dva korisnički definisana tipa izuzetaka: **ValidacijaImenaException** i **ValidacijaDatumaException**;
- Klase *Doktor* i *MedicinskaSestra* nasljeđuju klasu **MedicinskoOsoblje**;
- Izuzetak za neispravno ime i prezime nastaje ukoliko se kao parametar proslijedi *null* varijabla ili prazan string;
- Izuzetak za neispravan datum nastaje ukoliko se kao parametar proslijedi datum u budućnosti (iskorišten je datum 01/01/2025).

```
@Test
public void TestValidacija ()
{
    assertThrows(ValidacijaImenaException.class, () ->
    {
        MedicinskoOsoblje doktor = new Doktor (null, new Date(118,
0, 1), Zaposlenje.ZaposlenNeodredjeno, 5.0);
    });

    assertThrows(ValidacijaImenaException.class, () ->
    {
        MedicinskoOsoblje medicinskaSestra = new MedicinskaSestra
("", new Date(118, 0, 1), Zaposlenje.ZaposlenNeodredjeno, 500);
    });

    assertThrows(ValidacijaDatumaException.class, () ->
    {
        Doktor doktor = new Doktor ("Doktor", new Date(125, 0, 1),
Zaposlenje.ZaposlenNeodredjeno, 5.0);
    });
}
```

Listing 2. Test validacije vrijednosti atributa za klase

Na osnovu ovih informacija, sada je moguće napraviti preliminarni klasni sistem koji će sadržavati definicije svih klasa, njihove atribute i konstruktore. Za početak neće biti korištene *get()* i *set()* metode zbog smanjenja količine programskog koda, jer iste nisu korištene u okviru testova. Cilj preliminarnog klasnog sistema je da omogući uspješan prolazak inicijalizacije i testa za validaciju vrijednosti atributa. U Listingu 3 prikazan je enumeracijski tip *Zaposlenje* koji zasad ima samo jednu moguću vrijednost dodanu na osnovu programskog koda testne klase.

```
public enum Zaposlenje
{
    ZaposlenNeodredjeno
}
```

Listing 3. Preliminarna definicija enumeracijskog tipa Zaposlenje

U Listingu 4 prikazana je klasa *MedicinskoOsoblje* koja je proglašena apstraktnom jer se u testovima ne instancira direktno, koja sadrži tri atributa, tri *set()* funkcije koje vrše njihovu validaciju i konstruktor koji vrši poziv *set()* funkcija. U ovim funkcijama koriste se korisnički definisani tipovi izuzetaka.

```
public abstract class MedicinskoOsoblje {

    //region Atributi
    protected String imeIPrezime;
    protected Date datumZaposlenja;
    protected Zaposlenje vrstaZaposlenja;
    //endregion

    //region Properties
    public void setImeIPrezime(String imeIPrezime) throws
ValidacijaImenaException {
        if (imeIPrezime != null && imeIPrezime.length() > 0)
            this.imeIPrezime = imeIPrezime;
        else throw new ValidacijaImenaException("Neispravno uneseno
ime i prezime!");
    }
    public void setDatumZaposlenja(Date datumZaposlenja) throws
ValidacijaDatumaException {
        if (datumZaposlenja.before(new Date()))
            this.datumZaposlenja = datumZaposlenja;
        else throw new ValidacijaDatumaException("Neispravno unesen
datum zaposlenja!");
    }
    public void setVrstaZaposlenja(Zaposlenje vrstaZaposlenja) {
this.vrstaZaposlenja = vrstaZaposlenja; }
    //endregion

    //region Konstruktor
    public MedicinskoOsoblje (String imeIPrezime, Date
datumZaposlenja, Zaposlenje vrstaZaposlenja) throws
ValidacijaImenaException, ValidacijaDatumaException
    {
        setImeIPrezime(imeIPrezime);
        setDatumZaposlenja(datumZaposlenja);
        setVrstaZaposlenja(vrstaZaposlenja);
    }
    //endregion
}
```

Listing 4. Preliminarna definicija apstraktne klase MedicinskoOsoblje

Korisnički definisani tipovi izuzetaka *ValidacijaImenaException* i *ValidacijaDatumaException* prikazani su u Listingu 5, pri čemu se za njihovo definisanje koriste isti principi kao u prethodnim laboratorijskim vježbama, a nasljeđivanje klase *Exception* omogućava njihovo korištenje u testnoj metodi *Inicijalizacija()*.

```
public class ValidacijaImenaException extends Exception
{
   public ValidacijaImenaException (String message) {
super(message); }
}

public class ValidacijaDatumaException extends Exception
{
   public ValidacijaDatumaException (String message) {
super(message); }
}
```

Listing 5. Preliminarna definicija korisnički definisanih tipova izuzetaka

U Listingu 6 prikazana je preliminarna definicija klase *Pacijent*, pri čemu se za početak vrši definicija samo atributa i konstruktora, s obzirom da se u testovima ne koriste *get()* i *set()* metode. Pretpostavljeno je značenje trećeg i četvrtog parametra, a u slučaju nerazumijevanja moguće im je dati imena po želji.

```
public class Pacijent
{
   //region Atributi
   private String imeIPrezime, dijagnoza;
   private Date datumRodjenja;
   private Double ukupnaPrimanja;
   private Boolean penzioner;
   //endregion

   //region Konstruktor
   public Pacijent (String imeIPrezime, String dijagnoza, Date
datumRodjenja, Double ukupnaPrimanja, Boolean penzioner)
   {
       this.imeIPrezime = imeIPrezime;
       this.dijagnoza = dijagnoza;
       this.datumRodjenja = datumRodjenja;
       this.ukupnaPrimanja = ukupnaPrimanja;
       this.penzioner = penzioner;
   }
   //endregion
}
```

Listing 6. Preliminarna definicija klase Pacijent

Klasa *Doktor* definisana je u Listingu 7, pri čemu je definisano da nasljeđuje klasu *MedicinskoOsoblje* i koristi jedan dodatni atribut čije je značenje također pretpostavljeno.

```
public class Doktor extends MedicinskoOsoblje
{
    //region Atributi
    private Double koeficijentPlate;
    //endregion

    //region Konstruktor
    public Doktor (String imeIPrezime, Date datumZaposlenja,
Zaposlenje vrstaZaposlenja, Double koeficijentPlate) throws
ValidacijaImenaException, ValidacijaDatumaException
    {
        super(imeIPrezime, datumZaposlenja, vrstaZaposlenja);
        this.koeficijentPlate = koeficijentPlate;
    }
    //endregion
}
```

Listing 7. Preliminarna definicija klase Doktor

Na sličan način definisana je i klasa *MedicinskaSestra* koja također nasljeđuje klasu *MedicinskoOsoblje* i sadrži jedan dodatni atribut s pretpostavljenim značenjem, koja je definisana u Listingu 8.

```
public class MedicinskaSestra extends MedicinskoOsoblje
{
    //region Atributi
    private Integer brojNormaSati;
    //endregion

    //region Konstruktor
    public MedicinskaSestra (String imeIPrezime, Date
datumZaposlenja, Zaposlenje vrstaZaposlenja, Integer brojNormaSati)
throws ValidacijaImenaException, ValidacijaDatumaException
    {
        super(imeIPrezime, datumZaposlenja, vrstaZaposlenja);
        this.brojNormaSati = brojNormaSati;
    }
    //endregion
}
```

Listing 8. Preliminarna definicija klase MedicinskaSestra

Na ovaj način kreirane su sve preliminarne definicije klasnog sistema, koje će zatim biti nadograđene na osnovu ostalih testova u okviru programskog rješenja.

5.2. Kreiranje metoda klasa na osnovu testova

Naredna testna metoda koja će se analizirati je *TestPacijenti()*, a koja je prikazana u Listingu 9. Ova testna metoda koristi metodu klase *Doktor* **RadSaPacijentima()**, koja prima tri parametra:

- Vrijednost enumeracijskog tipa **VrstaOperacije**, koji u okviru testa poprima jednu od tri vrijednosti: *Dodavanje*, *Izmjena* i *Brisanje*;
- Objekat klase *Pacijent*, koji se koristi pri dodavanju i izmjeni, ali ne i pri brisanju;
- String koji označava ime i prezime pacijenta koji se pretražuje, a koji se koristi pri izmjeni i brisanju.

Na osnovu programskog koda testa, očigledno je da se očekuje da se omogući vršenje dodavanja, izmjene i brisanja pacijenata za doktore. Pritom se koristi metoda klase *Doktor* **getPacijenti()** koja kao rezultat vraća listu pacijenata, što znači da je potrebno dodati novi atribut u klasu *Doktor*. Osim toga, u metodi se koristi i *getImeIPrezime()* metoda klase *Pacijent* za koju već postoji atribut.

```
@Test
public void TestPacijenti ()
{
   doktor.RadSaPacijentima(VrstaOperacije.Dodavanje, pacijent1, "");
   assertEquals(doktor.getPacijenti().size(), 1);

   doktor.RadSaPacijentima(VrstaOperacije.Izmjena, pacijent2,
pacijent1.getImeIPrezime());

assertTrue(doktor.getPacijenti().get(0).getImeIPrezime().equals(paci
jent2.getImeIPrezime()));

   doktor.RadSaPacijentima(VrstaOperacije.Brisanje, null,
pacijent2.getImeIPrezime());
   assertFalse(doktor.getPacijenti().size() > 0);
}
```

Listing 9. Testna metoda za rad doktora s pacijentima

Nakon dodavanja atributa *pacijenti* u klasi *Doktor* i odgovarajućih *get()* metoda, može se preći na definisanje metode *RadSaPacijentima()*. Prije toga potrebno je dodati i novi enumeracijski tip *VrstaOperacije* koji je prikazan u Listingu 10. Programski kod ove metode koji omogućava uspješan prolazak testa *TestPacijenti()* prikazan je u Listingu 11, pri čemu se iterira kroz listu pacijenata kako bi se izvršio pronalazak pacijenta koji se želi izmijeniti ili obrisati, a za rad sa listom koriste se metode iz prethodnih laboratorijskih vježbi *add()*, *get()*, *set()* i *remove()*.

```
public enum VrstaOperacije
{
   Dodavanje, Izmjena, Brisanje
}
```

Listing 10. Enumeracijski tip VrstaOperacije

```
public void RadSaPacijentima(VrstaOperacije operacija, Pacijent
pacijent, String imeIPrezime)
{
   if (operacija == VrstaOperacije.Dodavanje)
       pacijenti.add(pacijent);
   else
   {
       int indeksPacijenta = -1;
       for (int i = 0; i < pacijenti.size(); i++)
           if
(pacijenti.get(i).getImeIPrezime().equals(imeIPrezime)) {
               if (operacija == VrstaOperacije.Izmjena)
                   pacijenti.set(i, pacijent);
               else
                   indeksPacijenta = i;
               break;
           }
       if (operacija == VrstaOperacije.Brisanje)
           pacijenti.remove(indeksPacijenta);
   }
}
```

Listing 11. Metoda doktora za rad s pacijentima definisana na osnovu testne metode

Naredna testna metoda je *TestPlateSvihOsoba()* i prikazana je u Listingu 12. U okviru ove metode prvenstveno se može primijetiti korištenje interfejsa **IIzracunPlate** kako bi se u istu listu mogli dodati i pacijenti, i doktor, kao i medicinska sestra. Nakon toga, nad objektima svih ovih klasa primjenjuje se metoda **DajPlatu()**, pri čemu se prilikom izračunavanja očekivanog rezultata može doći do zaključka da:

- Za izračunavanje plate pacijenta, uzima se vrijednost ukupnih primanja ukoliko nije penzioner, a u suprotnom se određuje vrijednost 0 (s obzirom da je razlika između objekata *pacijent1* i *pacijent2* u vrijednosti posljednjeg parametra);
- Za izračunavanje plate doktora, uzima se vrijednost koeficijenta plate pomnožena s 385;
- Za izračunavanje plate medicinske sestre, uzima se vrijednost osnovice plate pomnožena s brojem norma sati.

```
@Test
public void TestPlateSvihOsoba ()
{
   List<IIzracunPlate> osobe = new ArrayList<IIzracunPlate>();
```

```
    osobe.add(pacijent1);
    osobe.add(pacijent2);
    osobe.add(doktor);
    osobe.add(medicinskaSestra);
    Double ukupniRezultat = 0.0, ocekivaniRezultat = 0.0;

    for (IIzracunPlate osoba : osobe)
        ukupniRezultat += osoba.DajPlatu();

    ocekivaniRezultat = pacijent1.getUkupnaPrimanja() + 0.0 +
doktor.getKoeficijentPlate() * 385 + medicinskaSestra.getOsnovica()
* medicinskaSestra.getBrojNormaSati();

    assertEquals(ukupniRezultat, ocekivaniRezultat);
}
```

Listing 12. Testna metoda za izračunavanje plata svih klasa u sistemu

Kako bi ovaj test uspješno prolazio, potrebno je dodati novi interfejs *IIzracunPlate* koji je prikazan u Listingu 13, novi atribut *osnovica* u klasu *MedicinskaSestra* koji se ne inicijalizira koristeći parametar konstruktora, već ima neku *default* vrijednost (može se uzeti naprimjer 385, kao i kod doktora) i odgovarajuće *get()* metode koje se koriste u okviru testa.

```
public interface IIzracunPlate
{
    public Double DajPlatu();
}
```

Listing 13. Definicija interfejsa za izračunavanje plate

Sada je neophodno da kreirani interfejs naslijede sve izvedene klase (*Doktor* i *MedicinskaSestra*), kao i klasa *Pacijent*, a da se zatim izvrši implementacija metode *DajPlatu()* za sve tri klase na prethodno opisani način (Listing 14 - *Pacijent.DajPlatu()* pri čemu je demonstrirano i korištenje ternarnog operatora, Listing 15 - *Doktor.DajPlatu()* i Listing 16 - *MedicinskaSestra.DajPlatu()*).

```
public Double DajPlatu()
{
    return (penzioner) ? 0 : ukupnaPrimanja;
}
```

Listing 14. Implementacija metode za izračunavanje plate za klasu Pacijent

```
public Double DajPlatu()
{
    return koeficijentPlate * 385;
}
```

Listing 15. Implementacija metode za izračunavanje plate za klasu Doktor

```
public Double DajPlatu ()
{
   return osnovica * brojNormaSati;
}
```

Listing 16. Implementacija metode za izračunavanje plate za klasu MedicinskaSestra

Posljednja testna metoda *TestPlateDoktora()* prikazuje sve načine na koje se izračunava plata za doktore, a prikazana je u Listingu 17. Ova testna metoda otkriva sve ostale vrijednosti enumeracijskog tipa *Zaposlenje* i prikazuje da na osnovu njegovih vrijednosti plata biva smanjena na 20% ili 50% osnovnog iznosa.

```
@Test public void TestPlateDoktora ()
{
   Double plata = doktor.getKoeficijentPlate() * 385;
   assertEquals(doktor.DajPlatu(), plata);
   doktor.setVrstaZaposlenja(Zaposlenje.Zaposlen20Posto);
   assertEquals(doktor.DajPlatu(), plata * 0.2);
   doktor.setVrstaZaposlenja(Zaposlenje.Zaposlen50Posto);
   assertEquals(doktor.DajPlatu(), plata * 0.5);
}
```

Listing 17. Testna metoda za izračunavanje plate za doktora na različite načine

U Listingu 18 prikazana je konačna definicija enumeracijskog tipa *Zaposlenje*, a u Listingu 19 prikazana je izmijenjena definicija metode *DajPlatu()* za klasu *Doktor* koja omogućava uspješan prolazak testa *TestPlateDoktora()*. Na ovaj način završen je razvoj klasnog sistema koristeći predefinisane testove.

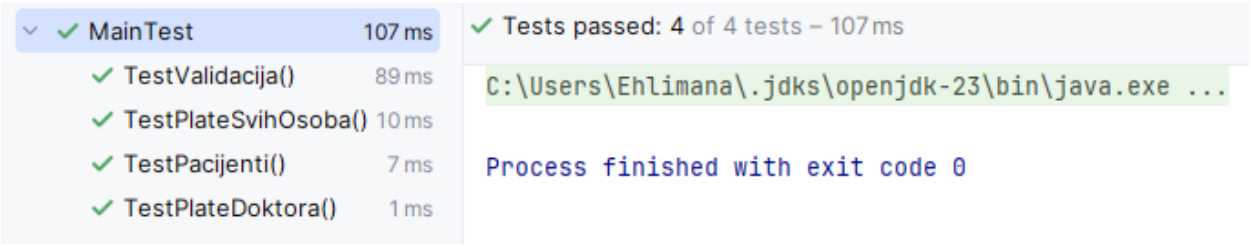
```
public enum Zaposlenje
{
   Zaposlen20Posto, Zaposlen50Posto, ZaposlenNeodredjeno
}
```

Listing 18. Enumeracijski tip Zaposlenje sa svim neophodnim vrijednostima

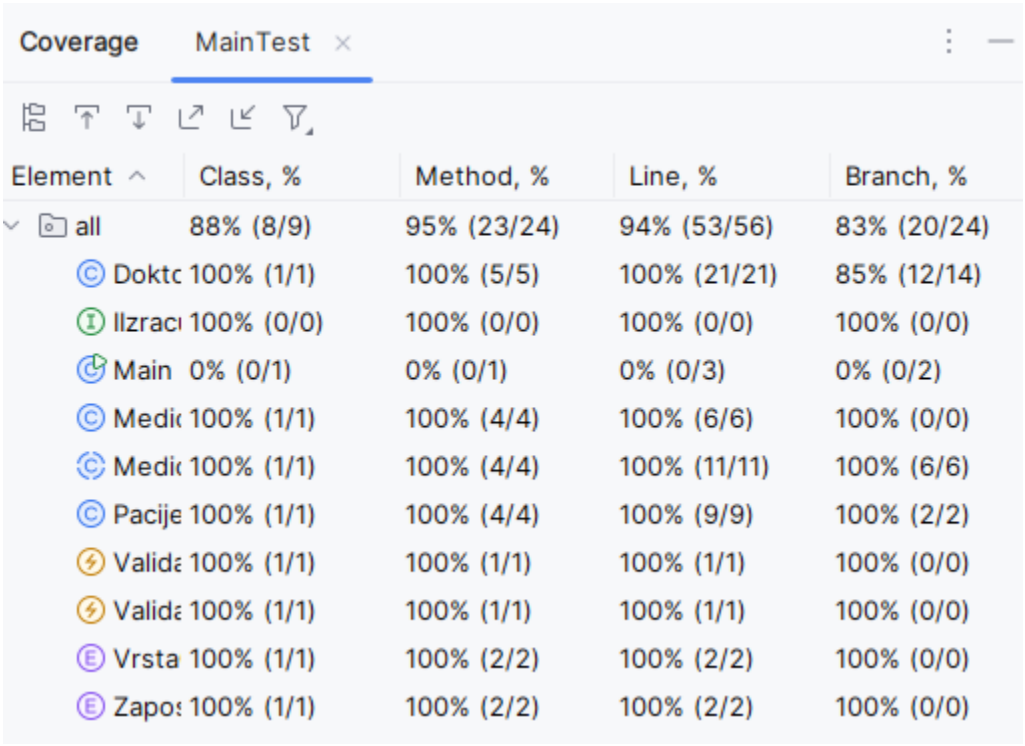
```
public Double DajPlatu()
{
   Double plata = koeficijentPlate * 385;
   if (vrstaZaposlenja == Zaposlenje.Zaposlen20Posto)
       plata *= 0.2;
   else if (vrstaZaposlenja == Zaposlenje.Zaposlen50Posto)
       plata *= 0.5;
   return plata;
}
```

Listing 19. Implementacija metode za izračunavanje plate za doktora ovisno o vrsti zaposlenja

Ukoliko se sada pokrenu testovi, dobiva se prikaz kao na Slici 1, odakle je vidljivo da su svi testovi uspješno prošli. Prikaz pokrivenosti koda vidljiv je na Slici 2, odakle se vidi da je postignua 100%-na pokrivenost klasa testovima, *Main* klasa nije testirana i u klasi *Doktor* postoje neki uslovi koji nisu potpuno testirani (što je posljedica neophodnosti da za svaki logički uslov budu izvršene provjere svih kombinacija vrijednosti). Važno je napomenuti da u prethodno prikazanim listinzima koda nisu prikazani svi dijelovi finalnih klasa, već preliminarne klase i najvažnije izmjene, te je neophodno dodavanje atributa, *get()* i *set()* metoda, kao i različitih stereotipa kako bi konačno programsko rješenje bilo potpuno ispravno definisano.



Slika 1. Uspješan prolazak svih testova nakon definicije klasnog sistema



Slika 2. Postizanje potpune pokrivenosti koda testovima

5.3. Zadaci za samostalni rad

Za sticanje bodova na prisustvo laboratorijskoj vježbi, potrebno je uraditi sljedeće zadatke tokom laboratorijske vježbe i postaviti ih u repozitorij studenta na odgovarajući način:

Zadatak 1.

Za testnu klasu koja je data u Listingu 20, kreirati klasni sistem koji omogućava uspješan prolazak svih testova.

```
public class TestClass
{
    static List<Ljubimac> ljubimci = new ArrayList<Ljubimac>();
    static Veterinar veterinar;

    @BeforeAll
    public static void Inicijalizacija ()
    {
        ljubimci.add(new Pas("Pas", new Date(118, 4, 20), "Nema
bolesti", VrstaPsa.ZlatniRetreiver));
        ljubimci.add(new Macka("Macka", new Date(120, 10, 1), "Nema
bolesti", VrstaMacke.Sijamska));
        veterinar = new Veterinar("Doktor", Specijalizacija.Psi);
    }

    @Test
    public void TestVeterinarskihUsluga ()
    {
        try {
            veterinar.PregledajLjubimca(ljubimci.get(0));
        }
        catch (Exception ex)
        {
            assertFalse(true);
        }
        assertTrue(veterinar.getPregledi().size() == 1);
        assertThrows(ValidacijaVrsteException.class, () ->
        {
           veterinar.PregledajLjubimca(ljubimci.get(1));
        });
    }

    @Test
```

```
    public void TestPrikazaInformacija ()
    {
        String rezultat = "";
        List<Objekat> objekti = new ArrayList<Objekat>();
        objekti.add(ljubimci.get(0));
        objekti.add(ljubimci.get(1));
        objekti.add(veterinar);

        for (Objekat o : objekti)
        {
            rezultat += o.PrikaziInformacije() + " ";
        }
        assertEquals(rezultat, "Pas: Zlatni Retreiver Mačka: Sijamska
Veterinar: Doktor ");
    }
}
```

Listing 20. Testna klasa na osnovu koje je potrebno razviti klasni sistem