

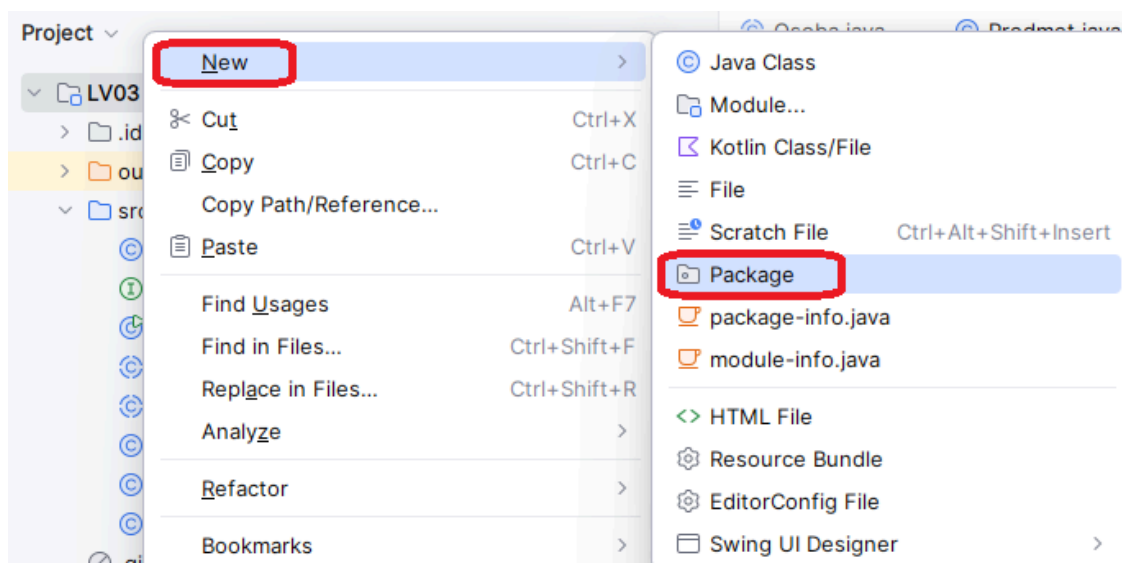
## Laboratorijska vježba 4.

# Unit testiranje programskih rješenja

### 4.1. Konfiguracija okruženja za testiranje

U prethodnim laboratorijskim vježbama, za provjeru ispravnosti definisanog programskog koda kreiranog klasnog sistema korištena je *Main* klasa u okviru koje se najčešće koristi korisnički unos i ispis dobivenih vrijednosti u konzoli. Ovo nije najbolji niti najbrži način za validaciju ispravnosti koda, posebno imajući u vidu primjene u kojima konzola ne postoji (npr. kreiranje programskih biblioteka). Iz tog razloga, potrebno je kreirati testove pomoću kojih će se provjeriti različite funkcionalnosti definisanog programskog koda, a pritom neće biti potrebe za korištenjem konzole, repeticijom niti radom s korisnikom.

Kao osnova za rad na ovoj laboratorijskoj vježbi koristiti će se klasni sistem iz prethodne laboratorijske vježbe. U prvom koraku, potrebno je izvršiti desni klik na osnovni folder projekta koji se nalazi s lijeve strane okruženja, a zatim odabrati opciju *New* → *Package*, na način koji je prikazan na Slici 1 (ponekad je moguće dodati i *Directory*, u tom slučaju koristiti tu opciju). Novom direktoriju potrebno je dodijeliti ime **test**, i nakon dodavanja prazan folder će biti dodan u folder strukturu programskog rješenja.

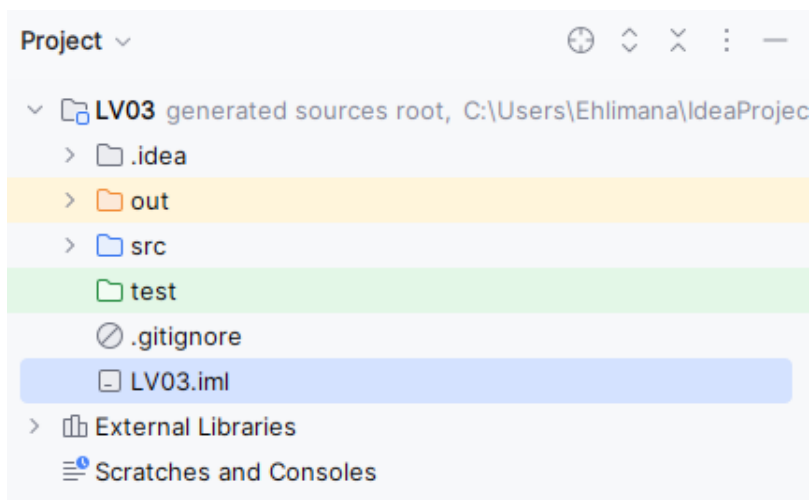


Slika 1. Dodavanje novog foldera u programsko rješenje

Prethodno definisani folder ima naziv *test*, ali okruženju nije označen kao takav. Iz tog razloga, potrebno je pristupiti **.iml datoteci** (koja se nalazi u osnovnom folderu projekta i ima isti naziv kao i naziv programskog rješenja) i dodati novu liniju koda **sourceFolder** koja specificira da je folder *test* testni folder (postaviti parametar **isTestSource="true"**), na način prikazan u Listingu 1. Nakon što se IML datoteka spasi, *test* folder biti će obojen zelenom bojom, što znači da je to sada testni direktorij (Slika 2). Važno je napomenuti da pri kreiranju novog programskog rješenja postoji mogućnost odabira opcije *Maven* kao *build* sistema, pri čemu će testni folder biti automatski generisan. Studentima se savjetuje da pokušaju i ovu mogućnost, ukoliko će im olakšati rad na predmetu.

```
<?xml version="1.0" encoding="UTF-8"?>
<module type="JAVA_MODULE" version="4">
  <component name="NewModuleRootManager"
inherit-compiler-output="true">
    <exclude-output />
    <content url="file://$MODULE_DIR$">
      <sourceFolder url="file://$MODULE_DIR$" isTestSource="false"
generated="true" />
      <sourceFolder url="file://$MODULE_DIR$/src"
isTestSource="false" />
      <sourceFolder url="file://$MODULE_DIR$/test"
isTestSource="true" />
    </content>
    <orderEntry type="inheritedJdk" />
    <orderEntry type="sourceFolder" forTests="false" />
  </component>
</module>
```

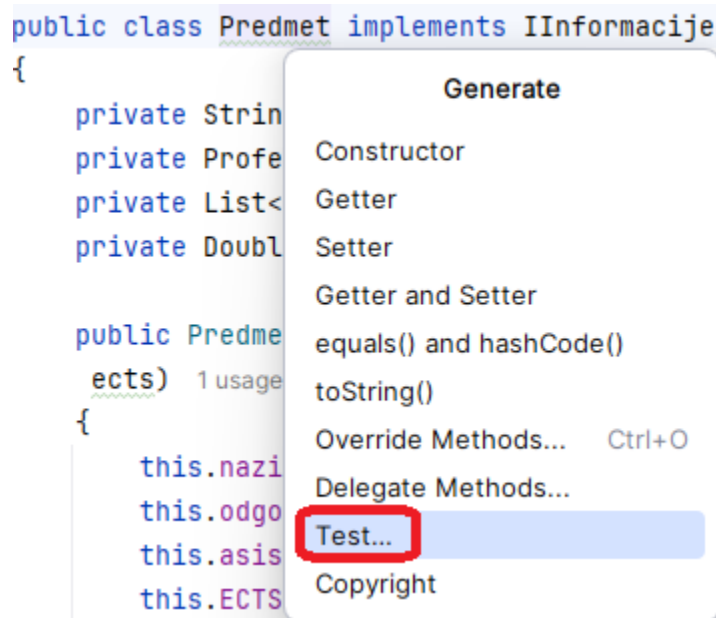
Listing 1. Specifikacija da je novi folder testni direktorij u IML datoteci



Slika 2. Prikaz testnog direktorija (označen zelenom bojom) u IntelliJ okruženju

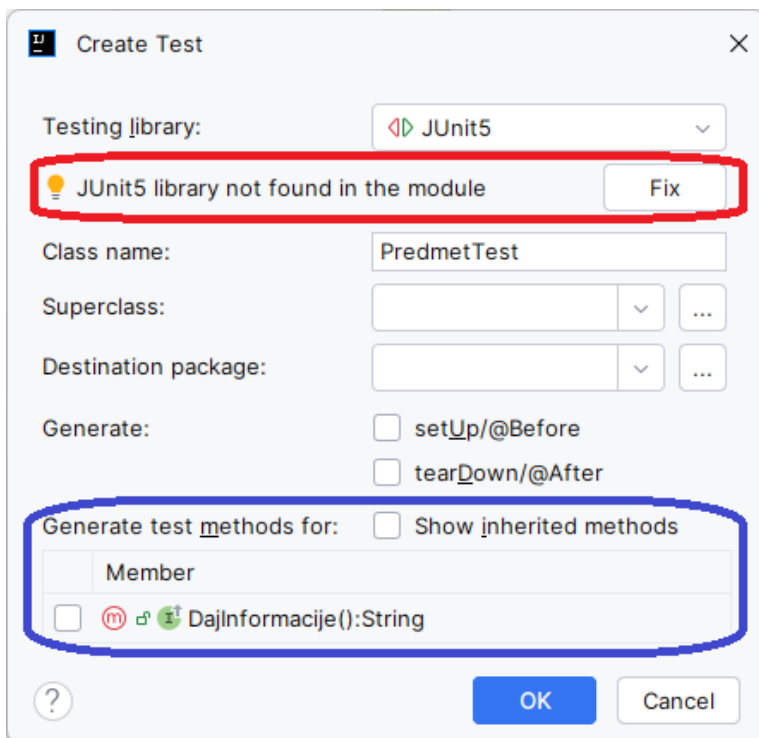
## 4.2. Testiranje metoda klasa

Sada je moguće dodavati nove testne klase. S obzirom da se na ovoj laboratorijskoj vježbi radi osnovni tip testiranja - testiranje jedinica programskog koda (*units*), za svaku klasu koja se želi testirati kreira se posebna testna klasa. Da bi se dodala nova testna klasa, potrebno je izvršiti desni klik na naziv klase u okviru *java* datoteke s programskim kodom, a zatim odabrati opciju *Generate...* → *Test...*, kao što je prikazano na Slici 3.



Slika 3. Dodavanje testne klase za željenu klasu

U idućem koraku dobiva se prikaz interfejsa koji prikazuje opcije za kreiranje nove testne klase (Slika 4). Ukoliko se dodaje nova testna klasa prvi put, dobiva se informacija o tome da biblioteka za testiranje *JUnit* nije instalirana i da je potrebno instalirati (označeno crvenom bojom na Slici 4). Iz tog razloga, prvo je potrebno odabrati opciju **Fix** kako bi se *JUnit* biblioteka instalirala. Nakon instalacije, ponovo se prikazuje isti interfejs, međutim bez upozorenja za biblioteku, koja je sada uspješno instalirana i može se koristiti. Sada se mogu odabrati druge opcije za kreiranje testne klase, poput njenog imena (preporučuje se da se ne vrše promjene u imenovanju testnih klasa), ali i opcije za automatsko generisanje nekih testova, poput detektovanih metoda klase (plavom bojom na Slici 4 označena je jedina detektovana metoda klase *Predmet* - *DajInformacije()*). O ostalim opcijama će biti riječi u narednim dijelovima laboratorijske vježbe. U ovom koraku potrebno je odabrati metodu *DajInformacije()* u okviru interfejsa za kreiranje testne klase, kako bi se kao rezultat automatski kreirao novi test u toj klasi.



Slika 4. Konfiguracija nove testne klase

Nakon automatskog generisanja testne klase sa jednom metodom, dobiva se prikaz kao u Listingu 2. Prvenstveno se može primijetiti da je automatski uključena biblioteka za testiranje, kao i da automatski generisani test ima stereotip **@org.junit.jupiter.api.Test**, koji označava da je u pitanju metoda za testiranje. Ovako definisana testna klasa može se iskoristiti kao osnova za testiranje klase *Predmet*.

```

import static org.junit.jupiter.api.Assertions.*;

class PredmetTest {

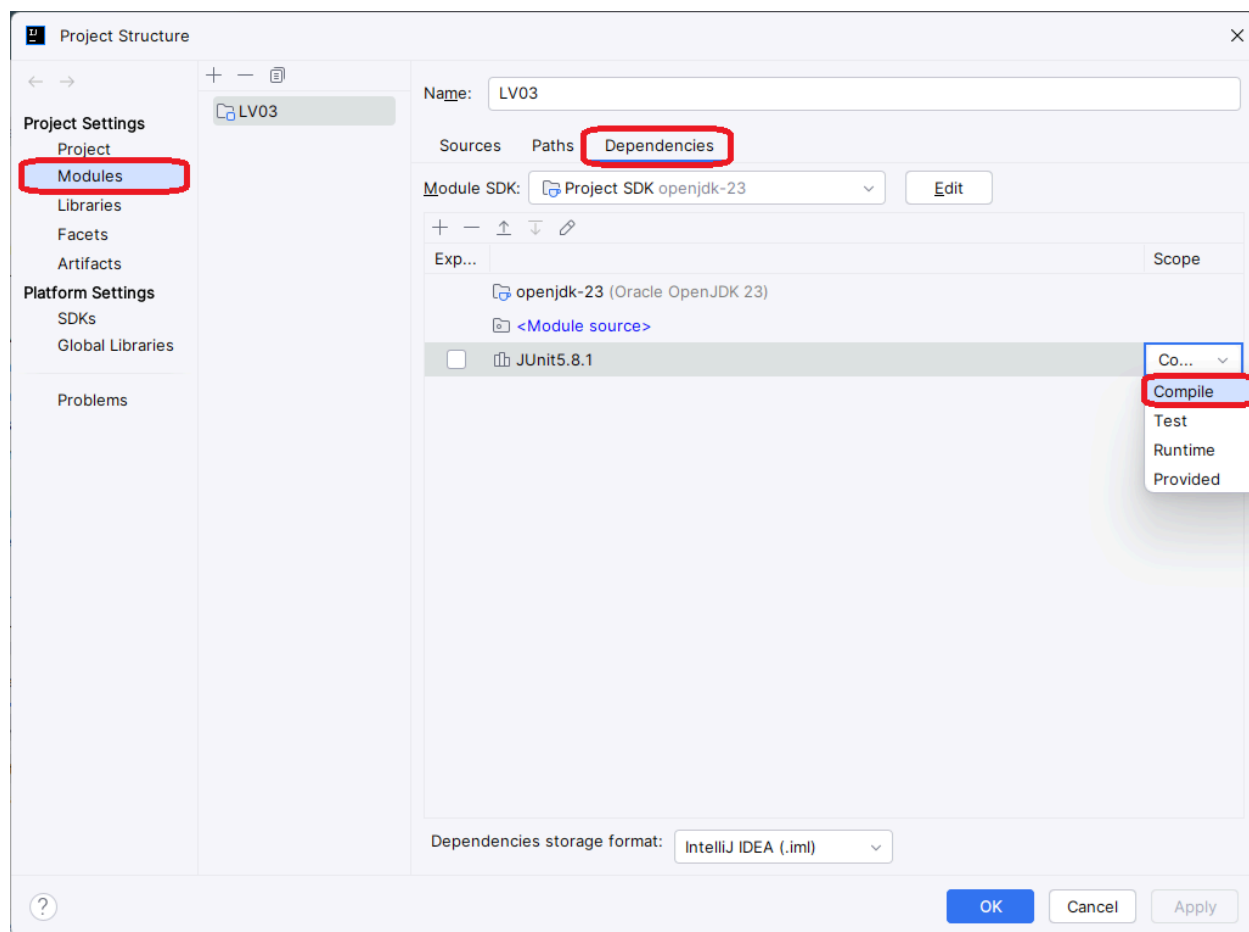
    @org.junit.jupiter.api.Test
    void dajInformacije() {

    }

}
  
```

Listing 2. Predefinisana testna klasa s jednom metodom

Prije dodavanja novog programskog koda za testiranje, potrebno je izvršiti posljednji korak kako bi se mogla koristiti *JUnit* biblioteka za testiranje - omogućiti njeno kompajliranje nakon prethodne uspješne instalacije. U tu svrhu, potrebno je izvršiti desni klik na osnovni folder projekta, a zatim odabrati opciju **Open Module Settings**. Nakon toga, potrebno je odabrati opciju *Modules* → *Dependencies*, nakon čega je za *JUnit* biblioteku potrebno odabrati opciju **Compile** umjesto predefinisane opcije *Test* (Slika 5). Ovime je završeno definisanje svega što je neophodno kako bi se moglo započeti sa testiranjem programskog koda.



*Slika 5. Konfiguracija JUnit biblioteke za korištenje u testnim klasama*

Sada se može započeti s definisanjem programskog koda za testove. Prvenstveno je potrebno dodati ključnu riječ **public** za definisanu testnu klasu, kako bi se moglo vršiti testiranje. Nakon što testna klasa postane javna, s lijeve strane editora programskog koda pojaviti će se ikone za testiranje (klikom na neku od ovih ikona, pokreće se dati test, ili svi testovi definisani u testnoj klasi). Listing 3 prikazuje primjer testiranja *DajInformacije()* metode u klasi *Predmet*. Ova metoda nema *if-else* strukturu niti više različitih tokova izvršavanja, zbog čega je jedan test dovoljan za validaciju njene ispravnosti. Za testiranje je iskorišten gotovo isti programski kod kao u *Main* klasi prethodne laboratorijske vježbe, uz jednu veoma važnu izmjenu - kriterija za validaciju ispravnosti. U tu svrhu koristi se *Assertions* klasa, koja posjeduje veoma veliki broj metoda za provjeru (za detaljne informacije, pregledati [sljedeći link](#)). U Listingu 3 upotrijebljena je metoda **assertEquals()**, koja vrši provjeru da li su vrijednosti dva proslijeđena parametra jednake. U tu svrhu za prvi parametar kreiran je predefinisani string koji sadrži očekivani rezultat, a drugi parametar predstavlja rezultat funkcije koja se testira *DajInformacije()*.

```

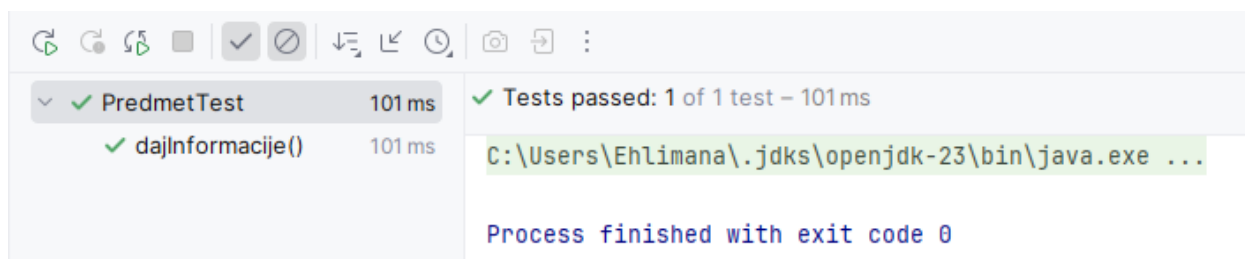
public class PredmetTest {

    @org.junit.jupiter.api.Test
    void dajInformacije() {
        Profesor prof = new Profesor("Profesor", "1", "", new
Date(99, 0, 1), 150, 2000, "3-00", "red. prof. dr.", 50);
        Predmet p = new Predmet("RPR", prof, null, 5.0);
        String ocekivaniRezultat = "Predmet: RPR, odgovorni
profesor: Profesor: red. prof. dr. Profesor 1";
        assertEquals(p.DajInformacije(), ocekivaniRezultat);
    }
}

```

*Listing 3. Dodavanje novog testa*

Nakon pokretanja testa, na donjem dijelu okruženja dobiva se prikaz kao na Slici 6, odakle je vidljivo da je test uspješno prošao, čime je validirana funkcija *DajInformacije()* klase *Predmet*. U slučaju više tokova izvršavanja jedne metode, mogu se kreirati novi testovi koji testiraju svaki pojedinačni tok, ili se u okviru jedne testne metode mogu dodavati *assert* naredbe za validaciju. Kreirani testovi mogu se pokretati nakon vršenja izmjena u programskom kodu kako bi se ponovo provjerila validnost funkcionalnosti, bez potrebe za korištenjem *Main* klase niti za korištenjem konzole za ulazne i izlazne informacije, čime je ostvarena nezavisnost programskog koda o načinu njegovog izvršavanja.



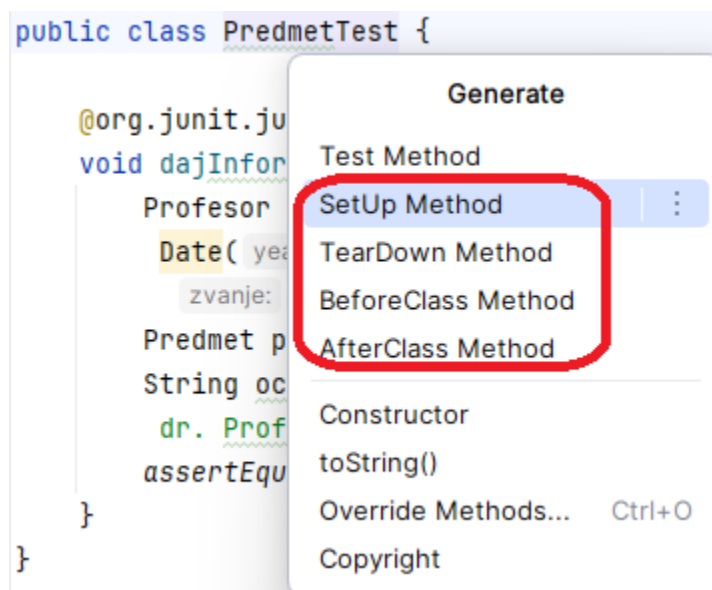
*Slika 6. Rezultati nakon izvršavanja definisanog testa*

### 4.3. Metode životnog ciklusa testne klase

Na prethodno opisani način mogu se kreirati testne klase i metode. Međutim, često postoji potreba da se određene informacije koriste u više testova, što ponovo dovodi do bespotrebne repeticije i nagomilavanja programskog koda. Naprimjer, neka se želi provjeriti validnost funkcije *ProvjeriMaticniBroj()* koja je u prethodnoj laboratorijskoj vježbi definisana u okviru apstraktne klase *Osoba*. S obzirom da je u testu iz Listinga 3 definisan objekat podtipa te klase (*Profesor*), ovaj objekat moguće je upotrijebiti kako bi se provjerila validnost date funkcije. Međutim, u novom testu bilo bi neophodno ponovo inicijalizirati isti objekat. Kako bi se to izbjeglo, može se koristiti predefinisana inicijalizacijska funkcija za testnu klasu. Specijalne funkcije moguće je automatski generisati pri kreiranju testne klase (Slika 4 - opcija **Generate: setUp/@Before** za inicijalizacijske metode, **tearDown/@After** za destruktorske metode), moguće ih je ručno

kreirati, ali i automatski generisati naknadno, desnim klikom na testnu klasu i odabirom opcije *Generate...* Nakon toga dobiva se prikaz kao na Slici 7, gdje su dostupne četiri vrste metoda (za više informacija pregledati [sljedeći link](#)):

- *SetUp Method*, koja će se izvršiti prije svakog pojedinačnog testa. Ovom metodom može se izvršiti ponovna inicijalizacija objekata i postavljanje parametara koji su eventualno promijenjeni u okviru izvršavanja testova;
- *TearDown Method*, koja će se izvršiti poslije svakog pojedinačnog testa. Ovom metodom može se izvršiti brisanje objekata koji su kreirani u okviru testa, kao i brisanje posljedice njihovog djelovanja;
- *BeforeClass Method*, koja će se izvršiti samo jednom, prije svih testova. Ovom metodom se najčešće inicijaliziraju objekti koji će se koristiti u okviru svih testova;
- *AfterClass Method*, koja će se izvršiti samo jednom, nakon svih testova. Ovom metodom najčešće se definišu logovi i neki izlazni rezultati, ukoliko postoji potreba za time.



Slika 7. Odabir testnih metoda koje se žele generisati

U konkretnom slučaju, nema potrebe da se objekat klase *Profesor* inicijalizira prije svakog testa, jer metode *DajInformacije()* i *ProvjeriMaticniBroj()* ne vrše nikakve izmjene nad objektima, nego samo vraćaju rezultate provjera. Iz tog razloga potrebno je odabrati opciju **BeforeClass Method**, što će za rezultat imati dodavanje nove testne metode **beforeAll()** koja ima stereotip **@BeforeAll** koji određuje način izvršavanja ove metode. Ova metoda je označena kao **static**, s obzirom da ne zahtijeva instancu testne klase (koja se uništava i nanovo kreira prije i poslije svakog testa). Ukoliko se sada inicijalizacija varijable tipa *Profesor* izdvoji u ovu metodu, potrebno je da inicijalizirana varijabla bude vidljiva i dostupna izvan inicijalizacijske metode i drugim testnim metodama. Kako bi se to postiglo, potrebno je deklarirati atribut testne klase (koji također mora biti **static** tipa kako bi se koristio u statičkoj metodi). Nakon toga, ova varijabla biti će dostupna za korištenje u testu *dajInformacije()*, kao i u svim dodatnim testovima

ove testne klase koji se eventualno žele kreirati. Cljela klasa *PredmetTest* nakon izvođenja prethodno opisanih promjena prikazana je u Listingu 4.

```
public class PredmetTest {
    private static Profesor prof;

    @org.junit.jupiter.api.Test
    void dajInformacije()
    {
        Predmet p = new Predmet("RPR", prof, null, 5.0);
        String ocekivaniRezultat = "Predmet: RPR, odgovorni
profesor: Profesor: red. prof. dr. Profesor 1";
        assertEquals(p.DajInformacije(), ocekivaniRezultat);
    }

    @org.junit.jupiter.api.BeforeAll
    static void beforeAll()
    {
        prof = new Profesor("Profesor", "1", "", new Date(99, 0, 1),
150, 2000, "3-00", "red. prof. dr.", 50);
    }
}
```

Listing 4. Konfiguracija inicijalizacijske metode u testnoj klasi

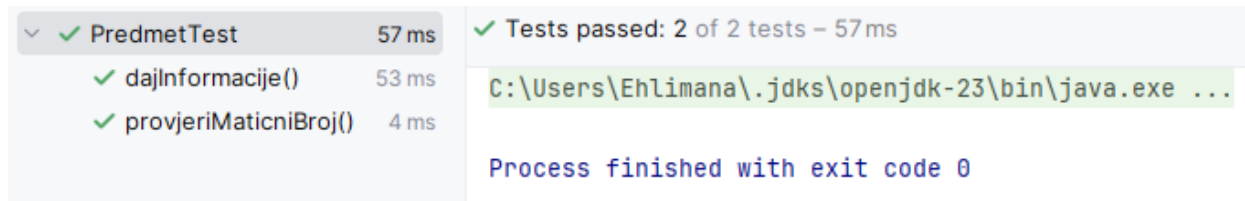
Sada je neophodno kreirati novi test *provjeriMaticniBroj()* koji testira istoimenu metodu klase *Osoba* koristeći statički atribut *prof*. U Listingu 5 prikazan je izgled ove metode, u okviru koje nema potrebe ni za kakvim inicijalizacijama. Iskorištene su dvije metode *Assertions* klase: metoda **assertTrue()** koja provjerava da li je vrijednost parametra *true* i metoda **assertFalse()** koja provjerava da li je vrijednost parametra *false*. Pritom se kao parametar šalje rezultat izvršavanja funkcije *ProvjeriMaticniBroj()*, za koji se u prvom slučaju postavljaju ispravne vrijednosti datuma rođenja profesora u stringu, a u drugom slučaju neispravne vrijednosti.

```
@org.junit.jupiter.api.Test
void provjeriMaticniBroj()
{
    assertTrue(prof.ProvjeriMaticniBroj("0101999123456"));
    assertFalse(prof.ProvjeriMaticniBroj("3112991123456"));
}
```

Listing 5. Test koji koristi inicijalizirani objekat testne klase

S obzirom da sada testna klasa ima definisana dva testa i jednu inicijalizacijsku metodu, najlakše je pokrenuti oba testa odjednom putem pokretanja svih testova klase. Dobiveni rezultati prikazani su na Slici 8, odakle je vidljivo da su oba testa uspješno prošla, čime je validirana ispravnost metoda klase *Predmet* i *Osoba*, bez potrebe za višestrukom inicijalizacijom objekta klase *Profesor* koji se koristi u oba testa.

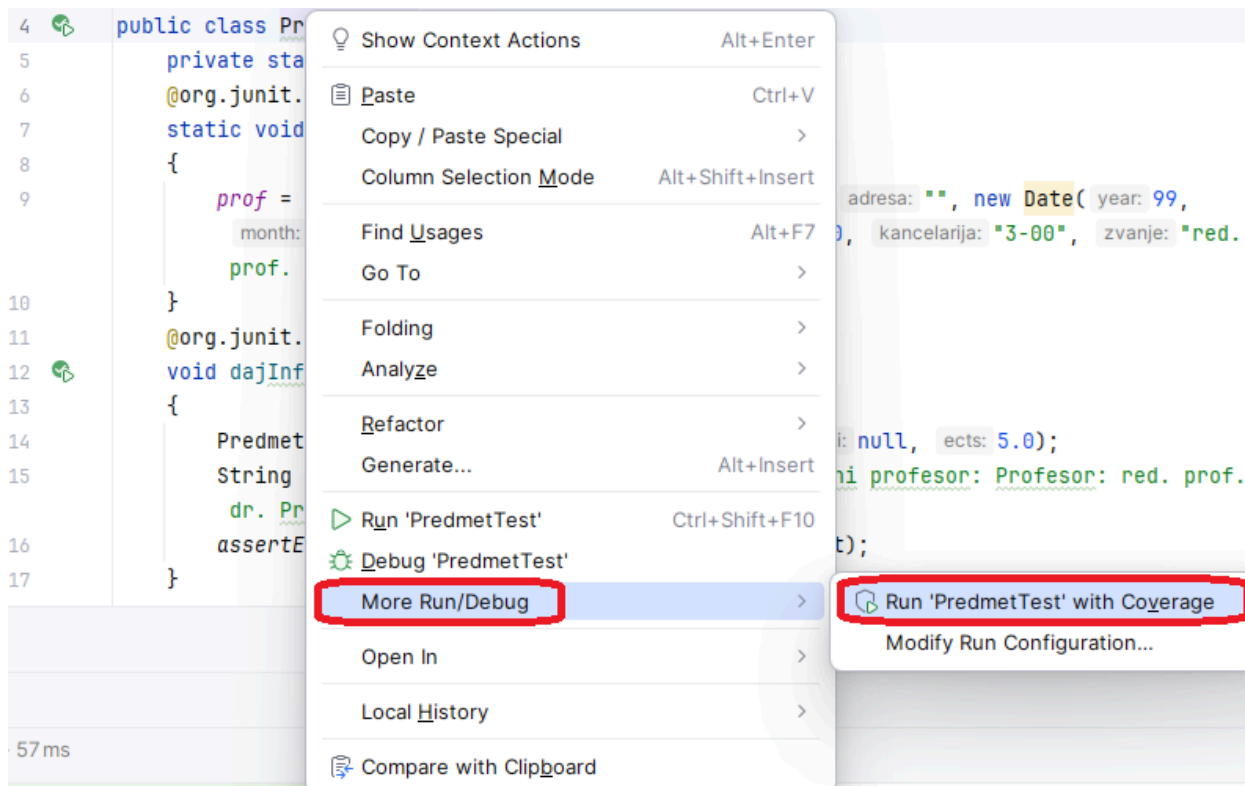




Slika 8. Rezultati nakon izvršavanja više definisanih testova

## 4.4. Pregled pokrivenosti koda testovima

Nakon završetka definisanja testova i njihovog uspješnog pokretanja, može se provjeriti koji dijelovi programskog koda klasnog sistema su ostali nepokriveni testovima. Kako bi se to izvršilo, potrebno je izvršiti desni klik na testnu klasu i odabrati opciju *More Run/Debug* → *Run 'PredmetTest' with Coverage*, na način prikazan na Slici 9.




Slika 9. Opcija za prikaz pokrivenosti koda testovima

Kao rezultat, prvo se pokreću svi testovi, a nakon njihovog izvršavanja otvara se novi tab s desne strane okruženja koji prikazuje informacije o pokrivenosti programskog koda testovima (Slika 10). U okviru ovog taba nalaze se informacije o postotku klase, metoda, linija koda i tokova izvršavanja koji su pokriveni testovima. Može se primijetiti da je za klase *Predmet* i *Profesor* postignuta potpuna pokrivenost koda, jer:

- Klasa *Predmet* ima konstruktor i metodu *DajInformacije()* koje su obje upotrijebljene u testovima;
- Klasa *Profesor* ima konstruktor, naslijeđeni konstruktor klase *Osoba*, naslijeđenu metodu *ProvjeriMaticniBroj()* klase *Osoba* i svoju implementaciju metode *DajInformacije()*, a koje su sve iskorištene u okviru testova.

Osim toga, i klase koje nisu direktno obuhvaćene testovima implicitno su testirane (npr. konstruktor klase *Osoba* koristi se u konstruktoru klase *Profesor*), dok interfejs *IIInformacije* ne sadrži programski kod koji je moguće testirati (testirano je 0/0 metoda).

Coverage PredmetTest x				
				
Element ^	Class, %	Method, %	Line, %	Branch, %
▼ all	57% (4/7)	58% (7/12)	53% (21/39)	75% (9/12)
Asistent	0% (0/1)	0% (0/1)	0% (0/3)	100% (0/0)
IIInformacije	100% (0/...)	100% (0/0)	100% (0/0)	100% (0/0)
Main	0% (0/1)	0% (0/1)	0% (0/9)	0% (0/2)
NastavnoOsoblje	100% (1/1)	100% (1/1)	100% (4/4)	100% (0/0)
Osoba	100% (1/1)	66% (2/3)	87% (7/8)	90% (9/10)
Predmet	100% (1/1)	100% (2/2)	100% (6/6)	100% (0/0)
Profesor	100% (1/1)	100% (2/2)	100% (4/4)	100% (0/0)
Student	0% (0/1)	0% (0/2)	0% (0/5)	100% (0/0)

Slika 10. Prikaz informacija o pokrivenosti koda testovima

## 4.5. 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.

Izvršiti testiranje preostalih klasa klasnog sistema iz laboratorijske vježbe 3 kako bi se postigla 100%-na pokrivenost programskog koda testovima. Programsko rješenje dostupno je na [sljedećem linku](#). Obavezno je dodati nove testne klase za svaku klasu klasnog sistema i koristiti inicijalizacijsku metodu za objekte koji će se koristiti u testovima.

Ukoliko tokom laboratorijske vježbe ne bude uspješno izvršen zadatak 1, moguće je steći bodove na prisustvo ukoliko se do sljedeće laboratorijske vježbe uradi i sljedeći zadatak:

## **Zadatak 2.**

Izvršiti testiranje svih klasa klasnog sistema iz Zadatka 1. laboratorijske vježbe 3 kako bi se postigla 100%-na pokrivenost programskog koda testovima. Obavezno je dodati nove testne klase za svaku klasu klasnog sistema i koristiti inicijalizacijsku metodu za objekte koji će se koristiti u testovima.