Laboratorijska vježba 3.

Objektno-orijentisano programiranje

3.1. Principi nasljeđivanja

U prethodnoj laboratorijskoj vježbi opisan je način kreiranja klase *Student* i način vršenja operacija koristeći ovako definisanu klasu. Sada je moguće kreirati još klasa (npr. *Predmet*, *Ispit*, *Knjiga* i sl.) koje će povećati broj funkcionalnosti i omogućiti rad sa kompleksnim klasnim sistemom u okviru programskog rješenja u Java programskom jeziku. Međutim, veoma često postoji potreba za kreiranjem više sličnih klasa. Naprimjer, neka je potrebno čuvati informacije o nastavnom osoblju (koje se dijeli na profesore i asistente) i studentima. Za sve ove klase potrebno je čuvati lične podatke, preklapanje *toString()* metode koja ispisuje te podatke može biti isto ili veoma slično, a klase dodatno mogu podržavati i iste funkcionalnosti, poput određivanja općine stanovanja na osnovu adrese ili izračunavanja radnog staža.

Kako bi se omogućilo ponovno korištenje (*reuse*) atributa i metoda osnovnih klasa u drugim klasama, koje načelno predstavljaju njihov podtip, koristi se nasljeđivanje. Nasljeđivanje je implicitno upotrijebljeno u prethodnoj laboratorijskoj vježbi pri kreiranju korisnički definisanih tipova izuzetaka, pri čemu je naslijeđena klasa *Exception*. Neka se sada želi napraviti klasni sistem koji omogućava nasljeđivanje, sa sljedećom organizacijom klasa u tri nivoa nasljeđivanja:

1. Klasa *Osoba*
2. Klase *Student* i *NastavnoOsoblje* koje nasljeđuju klasu *Osoba*
3. Klase *Profesor* i *Asistent* koje nasljeđuju klasu *NastavnoOsoblje*.

Listing 1 prikazuje osnovnu strukturu ovako definisanog klasnog sistema. Kao što je vidljivo, osnovna klasa *Osoba* jedina sadrži atribute i konstruktor a sve naslijeđene klase su prazne. U ovom primjeru nisu dodane *get()* i *set()* metode zbog uštede prostora u okviru laboratorijske vježbe, međutim podrazumijeva se da je potrebno iskoristiti iste principe kao u prethodnoj laboratorijskoj vježbi. Za nasljeđivanje se koristi stereotip **extends**, i u Java programskom jeziku moguće je koristiti i duboko nasljeđivanje (više od dva nivoa dubine), kao i široko nasljeđivanje (više od jedne klase na istom nivou - isključivo preko interfejsa). Kao i u prethodnoj laboratorijskoj vježbi, potrebno je da svaka od kreiranih klasa bude definisana u zasebnoj datoteci u *src* folderu.

```
public class Osoba
{
   String ime, prezime, adresa;
   Date datumRodjenja;

   public Osoba (String ime, String prezime, String adresa, Date
datumRodjenja)
   {
       this.ime = ime;
       this.prezime = prezime;
       this.adresa = adresa;
       this.datumRodjenja = datumRodjenja;
   }
}

public class Student extends Osoba
{ }

public class NastavnoOsoblje extends Osoba
{ }

public class Profesor extends NastavnoOsoblje
{ }

public class Asistent extends NastavnoOsoblje
{ }
```

Listing 1. Definisanje klasnog sistema koristeći nasljeđivanje

Prvenstveno se može primijetiti da ovako definisan klasni sistem dovodi do pojave kompajlerskih grešaka. Razlog za to je što naslijeđene klase predstavljaju podtip osnovne klase, međutim osnovna klasa ne posjeduje konstruktor bez parametara koji je moguće automatski naslijediti. Kako bi se riješio ovaj problem, potrebno je za početak preklopiti osnovni konstruktor klase *Osoba* u naslijeđenim klasama. U Listingu 2 prikazan je način preklapanja konstruktora samo za klasu *Student*, dok za ostale klase ovo preklapanje nije prikazano (a vrši se na isti način). Kao što se može vidjeti, konstruktor klase *Student* prima iste parametre kao i konstruktor klase *Osoba*, koji su neophodni kako bi se mogla izvršiti inicijalizacija atributa osnovne klase (koje također posjeduje izvedena klasa). Poziv konstruktora klase *Osoba* u okviru konstruktora izvedene klase *Student* poziva se koristeći funkciju **super()**, koja prima parametre konstruktora klase *Osoba*, čime se gubi potreba za ponavljanjem koda koji je definisan u osnovnom konstruktoru. Na ovaj način, u ostatku programskog koda konstruktora moguće je izvršiti sve funkcionalnosti koje su neophodne za inicijalizaciju svih atributa koji su jedinstveni za izvedenu klasu.

```
public class Student extends Osoba
{
   public Student (String ime, String prezime, String adresa, Date
```

```
datumRodjenja)
    {
        super(ime, prezime, adresa, datumRodjenja);
    }
}
```

Listing 2. Korištenje preklapajućeg konstruktora osnovnog tipa

Ukoliko se sada u *Main* klasi pokuša inicijalizirati varijabla klase *Osoba* i promijeniti vrijednost nekog od atributa, kao što je prikazano u Listingu 3, neće biti nikakvih poruka o greškama. Ovo je posljedica toga što je predefinisani tip atributa bilo koje klase **public**, odnosno ukoliko se ne navede tip atributa, oni će biti vidljivi izvan klase, čime se narušava princip enkapsulacije.

```
public class Main {
    public static void main(String[] args) {
        Osoba o = new Osoba("", "", "", new Date());
        o.ime = "a";
    }
}
```

Listing 3. Mogućnost korištenja atributa klase u drugim klasama

Ukoliko se za sve atribute klase **Osoba** postavi stereotip **private**, programski kod iz Listinga 3 rezultovati će pojavom greške, jer više nije dozvoljeno direktno pristupanje atributima klase iz drugih klasa. Dovoljno je promijeniti problematičnu liniju koda koja pokušava pristupiti privatnom atributu sa korištenjem *set()* metode za dati atribut, čime će biti riješen problem. Međutim, postavljanje svih atributa klase *Osoba* privatnima dovodi do mogućeg problema, ukoliko postoji potreba za direktnim korištenjem atributa u okviru izvedenih klasa (npr. pristup atributima osnovne klase u konstruktoru i metodama zahtijeva korištenje *get()* metoda). Kako bi se onemogućio direktan pristup atributima iz svih klasa osim izvedenih klasa date osnovne klase, može se koristiti ključna riječ **protected**, na način prikazan u Listingu 4. Važno je napomenuti da je, i nakon ovakve definicije atributa, programski kod Listinga 3 ispravan, jer se klase nalaze u istom paketu (za više informacija pogledati sljedeći link).

```
protected String ime, prezime, adresa;
protected Date datumRodjenja;
```

Listing 4. Promjena stereotipa atributa osnovne klase

Sada je neophodno dovršiti početne definicije svih klasa klasnog sistema (osim klase *Osoba*, koja je već potpuno definisana), uključujući:

- Dodavanje atributa koji su specifični za studenta, poput broja indeksa, godine studija, ukupnog prosjeka;
- Dodavanje atributa koji su specifični za nastavno osoblje, poput broja časova (norme), kancelarije u kojoj se nalaze i godine zaposlenja;

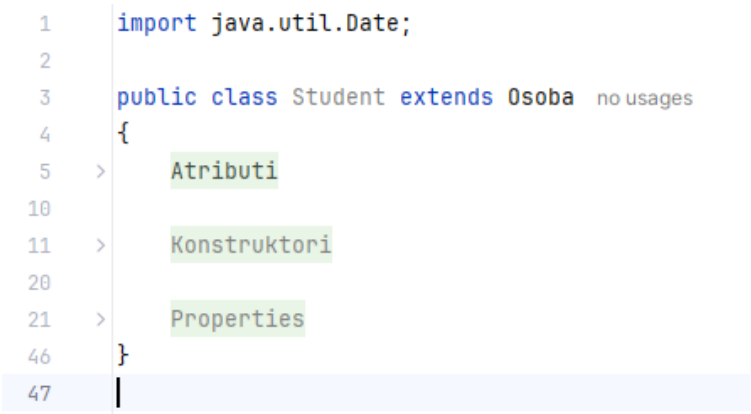
- Dodavanje atributa koji su specifični za profesore, poput zvanja i broja diplomanata kojima su mentori;
- Dodavanje atributa koji su specifični za asistente, poput laboratorije u kojoj održavaju vježbe i termina za konsultacije (dan u sedmici);
- Dodavanje *get()* i *set()* metoda za sve atribute svih pripadajućih klasa;
- Dodavanje pripadajućih konstruktora za sve klase, pri čemu svaka klasa koristi osnovni konstruktor klase iz koje je izvedena i *set()* metode za dodatne atribute.

U Listingu 5 prikazan je primjer početne definicije klase *Student* nakon primjene uputstava iznad, pri čemu su *get()* i *set()* metode izostavljene zbog uštede prostora. Također je uvedeno korištenje specifičnih komentara za označavanje regija koda, kako bi se omogućilo lakše snalaženje kroz programski kod (na Slici 1 prikazan je izgled klase kada se izvrši sažimanje koda, vidjeti primjere na sljedećem linku). Atributi klasa koje nemaju podtipova (*Student*, *Profesor* i *Asistent*) definisani su stereotipom **private**, dok su atributi osnovnih klasa (*Osoba* i *NastavnoOsoblje*) definisani stereotipom **protected**. Konstruktori svih izvedenih klasa sadrže dodatne parametre kojima se inicijaliziraju vrijednosti dodatnih atributa ovih klasa putem odgovarajućih *set()* metoda, nakon poziva **super()** konstruktora osnovne klase.

```
public class Student extends Osoba
{
    //region Atributi
    private String brojIndeksa;
    private Integer godinaStudija;
    private Double prosjek;
    //endregion

    //region Konstruktori
    public Student (String ime, String prezime, String adresa, Date
datumRodjenja, String brojIndeksa, Integer godinaStudija, Double
prosjek)
    {
        super(ime, prezime, adresa, datumRodjenja);
        setBrojIndeksa(brojIndeksa);
        setGodinaStudija(godinaStudija);
        setProsjek(prosjek);
    }
    //endregion
}
```

Listing 5. Definisanje izvedene klase uz korištenje regija koda



Slika 1. Način korištenja regija koda za lakše snalaženje

3.2. Korištenje polimorfizma

Osim zajedničkih atributa, klase mogu imati i zajedničke metode. Naprimjer, može biti potrebna metoda koja na osnovu ličnih podataka osobe provjerava validnost matičnog broja, kao i metoda koja vraća osnovne informacije u obliku stringa (koja u ovom primjeru neće biti definisana kao preklapanje *toString()* metode, već kao zasebna metoda). U Listingu 6 prikazan je način na koji je ove dvije metode moguće definisati za osnovnu klasu *Osoba*, pritom koristeći metode *getDate()*, *getMonth()* i *getYear()* klase *Date*, pri čemu je potrebno voditi računa da mjesec započinje nulom, a godina sa 1900.

```
public boolean ProvjeriMaticniBroj(String maticniBroj)
{
   boolean danIsti = datumRodjenja.getDate() ==
Integer.parseInt(maticniBroj.substring(0, 2)), mjesecIsti =
datumRodjenja.getMonth() + 1 ==
Integer.parseInt(maticniBroj.substring(2, 4)), godinaIsta =
datumRodjenja.getYear() + 900 ==
Integer.parseInt(maticniBroj.substring(4, 7));
   return (danIsti && mjesecIsti && godinaIsta);
}

public String DajInformacije()
{
   return "Ime i prezime: " + ime + " " + prezime;
}
```

Listing 6. Definisanje osnovnih implementacija metoda

Sada je u *Main* klasi moguće koristiti definisane metode, i to ne samo nad osnovnom klasom *Osoba*, već i nad svim njenim podtipovima. Listing 7 prikazuje programski kod u okviru kojeg se na isti način vrši poziv metode *ProvjeriMaticniBroj()* i za osnovnu klasu *Osoba*, i za jednu od njenih naslijeđenih klasa *Student*. Na isti način moguće je koristiti metodu *DajInformacije()* nad svim prethodno definisanim klasama. Na Slici 2 prikazan je izlaz u konzoli nakon pokretanja

ovako definisanog programskog koda, odakle je vidljivo da je metoda za provjeru matičnog broja u oba slučaja vratila pozitivan rezultat.

```
public static void main(String[] args) {
   Osoba o = new Osoba("", "", "", new Date(99, 0, 1));
   System.out.println("Provjera 1: " +
o.ProvjeriMaticniBroj("0101999123456"));
   Student s = new Student("", "", "", new Date(98, 2, 2), "12345",
2, 0.0);
   System.out.println("Provjera 2: " +
s.ProvjeriMaticniBroj("0203998123456"));
}
```

Listing 7. Korištenje iste implementacije metode nad više klasa u Main klasi

```
Provjera 1: true
Provjera 2: true

Process finished with exit code 0
```

Slika 2. Izlaz u konzoli nakon primjene iste metode nad više izvedenih klasa

Mogućnost korištenja osnovne (*default*) implementacije osnovne klase od strane naslijeđenih klasa je samo jedna od prednosti korištenja polimorfizma. Druga prednost predstavlja mogućnost izmjene implementacije metoda od strane naslijeđenih klasa, samo ukoliko je to potrebno. Ovakvo nešto je moguće i spriječiti (korištenjem stereotipa *final*, za više informacija vidjeti sljedeći link), međutim najčešće nije neophodno zbog korištenja objektno-orijentisanih principa pri nasljeđivanju klasa. Ukoliko se sada želi promijeniti implementacija metode *DajInformacije()* u nekoj od naslijeđenih klasa, nije potrebno korištenje nikakvih stereotipa (iako postoji stereotip **@Override**, koji se može koristiti za lakšu čitljivost koda, pri čemu je više informacija dato na sljedećem linku). Listing 8 prikazuje izmijenjenu implementaciju metode *DajInformacije()* u okviru klase *Student*.

```
public String DajInformacije()
{
   return "Student: " + ime + " " + prezime + ", broj indeksa: " +
brojIndeksa;
}
```

Listing 8. Izmijenjena implementacija metode istog naziva u drugoj klasi

Ovako definisane metode sada je moguće upotrijebiti u *Main* klasi, na način prikazan u Listingu 9. Iako na prvi pogled nema razlike između poziva metoda u odnosu na Listing 7, u ovom slučaju ne koristi se ista implementacija metode *DajInformacije()*, nego klase *Osoba* i *Student* posjeduju različite implementacije za ovu metodu. Iz tog razloga, na Slici 3 dobivaju se

dva različita izlaza u konzoli, s obzirom da su implementacije ovih metoda različite. Na ovaj način ostvaren je polimorfizam u definisanom klasnom sistemu.

```
public static void main(String[] args) {
   Osoba o = new Osoba("Osoba", "1", "", new Date(99, 0, 1));
   System.out.println(o.DajInformacije());
   Student s = new Student("Student", "1", "", new Date(98, 2, 2),
"12345", 2, 0.0);
   System.out.println(s.DajInformacije());
}
```

Listing 9. Korištenje različitih implementacija metode nad više klasa u Main klasi

```
Ime i prezime: Osoba 1
Student: Student 1, broj indeksa: 12345

Process finished with exit code 0
```

Slika 3. Izlaz u konzoli nakon primjene iste metode s različitim implementacijama nad više izvedenih klasa

Preostalo je još razmotriti da li u klasnom sistemu ima potrebe instancirati klasu *Osoba*, s obzirom da nema potrebe koristiti nedovršenu klasu koja se koristi kao baza za ostale klase koje predstavljaju srž sistema (studenti, profesori i asistenti). Do istog zaključka dolazi se za klasu *NastavnoOsoblje*, koja samo služi kao međunivo sa zajedničkim atributima i metodama za asistente i profesore. Kao rezultat, potrebno je klase *Osoba* i *NastavnoOsoblje* proglasiti apstraktnim, koristeći ključnu riječ **abstract**, a kao što je prikazano i u Listingu 10. Ovo će rezultovati da programski kod Listinga 9 sada dovodi do pojave greške, jer više nije moguće kreirati instancu klase *Osoba*.

```
public abstract class Osoba
```

Listing 10. Proglašavanje klase apstraktnom

3.3. Interfejsi

Sve što je prethodno opisano služi kako bi se mogao kreirati kompleksni klasni sistem u kojem klase međusobno koriste atribute i metode. Međutim, nasljeđivanje je moguće primijeniti samo ukoliko naslijeđene klase predstavljaju podtip osnovne klase (naprimjer, nema smisla naslijediti klasu *Predmet* iz klase *Student* ili *Osoba*, jer nastavni predmeti nisu vrste osoba, a niti posjeduju iste atribute i ne ponašaju se na isti način). S druge strane, ponekad postoji potreba da različiti objekti posjeduju iste atribute ili, mnogo češće, da podržavaju pozivanje istih metoda. Kako bi se kreirao zajednički sadržilac za objekte koji nemaju ništa zajedničko, koriste se **interfejsi**.

Za početak, neka je potrebno da postoji klasa *Predmet* koja nema nikakve veze s prethodno kreiranim klasnim sistemom i sadrži informacije o nazivu, odgovornom profesoru i asistentima, kao i broju ECTS bodova. Početna definicija ove klase (bez prikaza *get()* i *set()* metoda, kao i konstruktora) data je u Listingu 11. Neka je sada potrebno da i ova klasa podržava metodu *DajInformacije()*, kao i sve ostale klase klasnog sistema, kako bi se ispisale informacije o datom predmetu. Ova metoda je također definisana u Listingu 11 i u sebi sadrži poziv istoimene metode za odgovornog profesora.

```
public class Predmet
{
   private String naziv;
   private Profesor odgovorniProfesor;
   private List<Asistent> asistenti;
   private Double ECTS;

   public String DajInformacije()
   {
       return "Predmet: " + naziv + ", odgovorni profesor: " +
odgovorniProfesor.DajInformacije();
   }
}
```

Listing 11. Kreiranje nove klase koja nije povezana s nasljeđivanjem

Ukoliko se sada želi izvršiti poziv metode *DajInformacije()* nad različitim klasama klasnog sistema, isto je jednostavno moguće uraditi nakon dodavanja svih objekata izvedenih klasa u listu elemenata osnovnog tipa (klase *Osoba*), a što je izvršeno u Listingu 12. Međutim, iako objekat klase *Predmet* posjeduje svoju implementaciju metode *DajInformacije()*, nije ga moguće dodati u listu, jer on ne predstavlja podvrstu klase *Osoba*.

```
public static void main(String[] args) {
   Profesor p = new Profesor("Profesor", "1", "", new Date(99, 0,
1), 150, 2000, "3-00", "red. prof. dr.", 50);
   Student s = new Student("Student", "1", "", new Date(98, 2, 2),
"12345", 2, 0.0);
   Predmet pr = new Predmet("RPR", p, null, 5.0);

   List<Osoba> osobe = new ArrayList<Osoba>();
   osobe.add(p);
   osobe.add(s);
   for (Osoba o : osobe)
       System.out.println(o.DajInformacije());
}
```

Listing 12. Korištenje osnovnog tipa za poziv iste metode nad više izvedenih objekata

Kako bi se riješio ovaj problem i omogućilo da se objekti klase *Predmet* poistovijete sa objektima koji su podtipovi klase *Osoba*, a koji ne predstavljaju dio istog klasnog sistema, dodati

će se interfejs *IInformacije* koji će sadržavati definiciju zajedničke metode za ove klase (interfejsima se najčešće dodaju imena koja započinju sa *I*, kako bi se označilo da nije u pitanju klasa). S obzirom da je interfejs apstraktan (kao i osnovne klase), nije ga moguće direktno instancirati, a niti je moguće definisati tijelo metoda koje sadrži (za sva pravila pri korištenju interfejsa, pregledati sljedeći link). Listing 13 prikazuje definiciju interfejsa *IInformacije* (koji je također u zasebnoj datoteci, kao i sve klase) koji samo sadrži definiciju metode koja će biti zajednička za klase *Predmet* i klasni sistem zasnovan na osnovnom tipu *Osoba*.

```
public interface IInformacije
{
   public String DajInformacije();
}
```

Listing 13. Način definisanja interfejsa

Ovako definisan interfejs nasljeđuje se od klase *Predmet* i klase *Osoba*, a neophodno je da samo izvedene klase klasnog sistema koje se mogu instancirati (*Student*, *Profesor* i *Asistent*) imaju implementaciju datog interfejsa. Za označavanje da neka klasa nasljeđuje interfejs koristi se ključna riječ **implements** (Listing 14).

```
public abstract class Osoba implements IInformacije
```

Listing 14. Deklaracija nasljeđivanja interfejsa

Sada je u klasi *Main* moguće obuhvatiti klasu *Predmet* i podklase klase *Osoba* u okviru jedne liste tipa interfejsa *IInformacije*, na način prikazan u Listingu 15. Na Slici 4 prikazan je izlaz u konzoli nakon pokretanja ovako definisanog programskog koda, odakle je vidljivo da je korištenjem interfejsa ispravno omogućeno pozivanje metode za prikaz informacija i za studente, i za profesore, i za asistente, kao i za predmete, iako oni ne predstavljaju vrstu osoba.

```
public static void main(String[] args) {
   Profesor p = new Profesor("Profesor", "1", "", new Date(99, 0,
1), 150, 2000, "3-00", "red. prof. dr.", 50);
   Student s = new Student("Student", "1", "", new Date(98, 2, 2),
"12345", 2, 0.0);
   Predmet pr = new Predmet("RPR", p, null, 5.0);

   List<IInformacije> objekti = new ArrayList<IInformacije>();
   objekti.add(p);
   objekti.add(s);
   objekti.add(pr);
   for (IInformacije o : objekti)
       System.out.println(o.DajInformacije());
}
```

Listing 15. Korištenje interfejsa za poziv iste metode nad različitim objektima

```
Profesor: red. prof. dr. Profesor 1
Student: Student 1, broj indeksa: 12345
Predmet: RPR, odgovorni profesor: Profesor: red. prof. dr. Profesor 1

Process finished with exit code 0
```

Slika 4. Izlaz u konzoli pri korištenju interfejsa

3.4. 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.

Kreirati sljedeći klasni sistem, pritom ispravno primjenjujući principe nasljeđivanja i polimorfizma, enkapsulacije i apstrakcije, kao i korištenja interfejsa:

- *Voće* ima svoj latinski naziv i zemlju porijekla, kao i listu nutritivnih vrijednosti (npr. količina masti, ugljikohidrata, proteina, vlakana, vitamina). Potrebno je podržati metodu *DajBrojKalorija()* koja na osnovu nutritivnih vrijednosti (koje predstavljaju pojedinačne brojeve kalorija) izračunava ukupan broj kalorija. Potrebno je podržati i metodu *Zdravlje()* koja označava voće zdravim ako je broj kalorija manji od 50, a vrijednost parametra *koeficijentZdravlja* veći od 0.75.
- *Povrće* ima svoj latinski naziv i zemlju porijekla, kao i listu nutritivnih vrijednosti (isto kao za voće). Potrebno je također podržati metodu *DajBrojKalorija()*, koja izračunava broj kalorija na isti način kao za voće. Potrebno je podržati i metodu *Zdravlje()* koja označava povrće zdravim ako je broj kalorija manji od 100, a vrijednost parametra *koeficijentZdravlja* između 0.5 i 0.7.
- *Meso* ima svoju vrstu, koja može biti jedna od četiri mogućnosti (piletina, puretina, teletina, janjetina) i zemlju porijekla, kao i listu nutritivnih vrijednosti (isto kao za voće i povrće). Potrebno je također podržati metodu *DajBrojKalorija()*, koja izračunava broj kalorija na sliča način kao za voće i povrće, s tim što na kraju skalira vrijednost povećavajući je na 120% iznosa broja kalorija. Potrebno je podržati i metodu *Zdravlje()* koja označava meso zdravim samo ako je *koeficijentZdravlja* veći od 0.95.
- *Prodavač* ima ime i prezime, broj štanda i ID broj licence. Potrebno je podržati metodu *Zdravlje()*, koja označava prodavača zdravim ukoliko ID završava sa ciframa “01”.

Obavezno je upotrijebiti jednu apstraktnu klasu i jedan interfejs, kao i primijeniti nasljeđivanje između nekih od klasa u klasnom sistemu. Obavezno kreirati i programsku logiku u *main()* metodi koja demonstrira korištenje metoda *DajBrojKalorija()* i *Zdravlje()* za sve klase u okviru sistema.