Podrška objektno orijentisanom programiranju u jezicima C++, Objective C, Java, C#, Ada i Ruby

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Katarina Popović, Dušan Pantelić, Dejan Bokić, Nikola Stojević
ai17210@alas.matf.bg.ac.rs, pantelic.dusan@protonmail.com,
mrdejan995@gmail.com, nikolastojevic@gmail.com

6. april 2019

Sažetak

U ovom tekstu je ukratko prikazana osnovna forma seminarskog rada. Obratite pažnju da je pored ove .pdf datoteke, u prilogu i odgovarajuća .tex datoteka, kao i .bib datoteka korišćena za generisanje literature. Na prvoj strani seminarskog rada su naslov, apstrakt i sadržaj, i to sve mora da stane na prvu stranu! Kako bi Vaš seminarski zadovoljio standarde i očekivanja, koristite uputstva i materijale sa predavanja na temu pisanja seminarskih radova. Ovo je samo šablon koji se odnosi na fizički izgled seminarskog rada (šablon koji *morate* da koristite!) kao i par tehničkih pomoćnih uputstava. Pročitajte tekst pažljivo jer on sadrži i važne informacije vezane za zahteve obima i karakteristika seminarskog rada.

Sadržaj

1 Uvod **2**

2 C++ **3**

3 Objective C **4**

4 Java **6**

5 C# **7**

6 Ada **9**

7 Ruby **10**

8 Zaključak **11**

Literatura **11**

1 Uvod

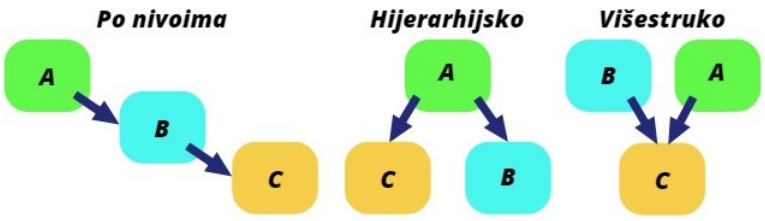
Programska paradigma je osnovni stil programiranja, način klasifikovanja programskih jezika na osnovu njihovih karakteristika. Programski jezici (u daljem tekstu jezici) često mogu da se klasifikuju u više od jedne paradigme.

Najčešće programske paradigme su: proceduralna (fokus stavlja na promenljive), funkcionalna (fokus na matematičkim funkcijama), logička (fokus na logičkim izrazima) i objektno-orijentisana paradigma (fokus je na **objektima**). Objekti imaju svoje interno stanje (atribute) i svoje ponašanje (metode). I metode i atributi mogu a ne moraju biti vidljivi spoljašnjem svetu. Srž objektno-orijentisanog programiranja je u slanju poruka i odgovaranju na njih. Ovu paradigmu najbolje opisuje tzv. *Tell dont ask* princip-

> Nemoj da tražiš informaciju koja ti treba da bi uradio posao. Umesto toga, pitaj objekat koji ima tu informaciju da odradi taj posao za tebe. *(Allen Hollub- Hollub on Patterns)*

Osnovni principi objektno-orijentisane paradigme:

1. **Princip jedinstvene odgovornosti-** Svaka klasa treba da ima samo jedan zadatak. Čim klasa ima više od jednog zadatka, verovatno bi mogla da se podeli u 2 klase.
2. **Enkapsulacija-** i *Tell dont ask* princip su usko vezani. Vidljivost podataka je osnovni način enkapsulacije. U većini programskih jezika, postoje 3 osnovna nivoa vidljivosti- public, protected i private.
3. **Nasleđivanje-** podrazumeva da jedna klasa "nasledišve metode i atribute svoje nadklase, sem kad su metode i atributi proglašeni privatnim. Uspostavlja odnos "jeste"izmedju klasa- ako klasa A nasleđuje klasu B, kaže se da A "jeste"B. Klasa B se naziva nadklasom klase A. Vrste nasleđivanja prikazane su na slici 1.
4. **Polimorfizam-** označava da klase koje implementiraju isti interfejs mogu da rade različite stvari, dok god se drže principa tog interfejsa.
5. **Apstrakcija-** je termin koji se veže za *interfejse* i *apstraktne klase*. **Interfejsi** se koriste za opisivanje ponašanja klase, definišu javni API (Application programming interface) klase, odnosno metode koje klasa mora da implementira da bi program radio. Oni ne mogu biti instancirani, mogu imati samo potpise metoda, bez tela, ne mogu imati atribute i sve metode moraju biti javne. **Apstraktne klase** su slične interfejsima, ali za razliku od njih metode apstraktnih klasa mogu da imaju telo, mogu da imaju atribute i mogu imati sva moguća prava pristupa (public, protected, private).



Slika 1: Vrste nasledjivanja

2 C++

Jezici koji podržavaju sve 4 funkcionalnosti objektno orijentisane paradigme ali ne u potpunosti se obično nazivaju delimično objektno orijentisanim jezicima. Zbog sledećih karakteristika, C++ se smatra delimično objektno orijentisanim jezikom.

1. **Main funkcija je izvan klase :** U C++ može da se napiše validan, ispravan kod bez kreiranja ijednog objekta. Main funkcija je obavezna ali se ona nalazi izvan svake klase, što nije karakteristično za druge OOP jezike.
2. **Koncept globalne promenljive:** U C++ može da se kreira globalna promenljiva koja je dostupna svugde u kodu, dok u drugim OOP jezicima promenljive mogu biti deklarisane samo u okviru klase gde mogu da se koriste modifikatori pristupa (private, protected, public).
3. **Postojanje friend funkcija:** Friend (prijateljska) funkcija može pristupiti privatnim poljima klase kojoj je deklarisana kao prijateljska. Ovo je jedna veoma korisna karakteristika C++, ali i dalje narušava neke koncepte objektno orijentisane paradigme.

2.1 Enkapsulacija

U C++ ne mora eksplicitno za svaki atribut ili metod klase da bude naglašeno pravo pristupa, nego se prave sekcije, i na početku sekcije se stavi modifikator pristupa (podržani su private, public i protected, ne podržava package modifikator). Ukoliko se modifikator ne navede eksplicitno, kod klasa se podrazumeva private, dok kod struktura se podrazumeva public (što je jedna od osnovnih razlika između struktura i klasa u C++).

```
1000 class Employee {
     private:
1002   int salary;
     public:
1004   Employee(int e_salary) { salary = e_salary;}
       int getSalary(){ return salary;}
1006   void setSalary(int newSalary) { salary = newSalary;}
       void display() {
1008         std::cout << "Hello I'm the employee!" << std::endl;
       }
1010 };
```

Listing 1: Primer deklarisanja klase sa enkapsulacijom

2.2 Nasleđivanje

U jeziku C++ i nasleđivanje može biti private, protected ili public. Ukoliko je nasleđivanje public, to znači da sva nasleđena polja ostaju javna, ukoliko je protected, tada ce sva public polja postati protected, a ukoliko je nasleđivanje private, to znači da će sva public i protected polja postati private. Takođe, za razliku od drugih OOP jezika, u C++ je podržano i višestruko nasleđivanje, gde jedna klasa može da nasledi više od jedne klase. Zbog problema koje višestruko nasleđivanje može da uvede, u C++ je uvedena jos jedna ključna reč prilikom nasleđivanja- vritual, koja sprečava tzv. dijamant strukturu.

```
1000 class Driver: public Employee  {
     private:
```

```
1002      std::string truck = "FAP";
public:
1004      Driver(int salary, std::string truck)
    : Employee(salary), truck(truck)
1006    {};
      void display() {
1008      std::cout << "My truck is " << truck << "!" << std::endl; };
};
```

Listing 2: Primer nasleđivanja klasa u C++

2.3 Polimorfizam

U C++ su podržana dva osnovna tipa polimorfizma- **polimorfizam u vreme kompilacije** i **polimorfizam u vreme izvršavanja**. Prvi obezbeđuje preopterećenje metoda i operatora. Preopterećenje operatora je isto jedna od C++ specifičnih mogućnosti, gde možemo sami da definišemo ponašanje operatora npr "+đok god ispunjava svoje osnovne karakteristike (da ima 2 argumenta). Drugi tip polimorfizma omogućuje premošćavanje metoda, tj. za metod se kaže da je premošten ukoliko izvedena klasa poseduje metod sa identičnim potpisom.

2.4 Apstrakcija

Za razliku od drugih OOP jezika, C++ ne poseduje ključnu rec *abstract*. Apstraktne klase se u C++ kreiraju tako što se napravi virtuelna metoda i u potpisu joj se dodeli 0. Takva klasa ne može biti instancirana, ali može biti napravljen pokazivač na nju. Takođe, apstraktna klasa može da ima konstruktor i destruktor. Klasa koje ne premosti virtuelnu metodu takođe postaje apstraktna klasa. Postoji nekoliko pravila koja moraju da važe za apstraktne klase:

1. Moraju biti proglašene javnim (inače potklasa ne može da ih premosti).
2. Virtuelne metode ne mogu biti static i ne mogu biti prijateljske metode neke druge klase.
3. Virtuelnim metodama se mora pristupati preko pokazivača na baznu klasu da bi se dobio pravi polimorfizam u vreme izvršavanja.
4. Potpis virtuelne metode mora biti identičan i u baznoj i izvedenoj klasi (povratna vrednost, tipovi argumenata, konstantnost argumenata,...).
5. Klase mogu imati virtuelni destruktor, ali ne mogu da imaju virtuelni konstruktor.

3 Objective C

Proces definisanja klasa se malo razlikuje kod jezika Objective C. Obavlja se u dve sekcije koje se označavaju sa ključnom reči **@interface** i **@implementation**, gde se vrši deklaracija i implementacija metoda klase. Obe sekcije se završavaju ključnom reči **@end**. Svaka klasa je izvedena iz super klase NSObject, čiji konstruktor init je podrazumevani konstruktor, može predefinisati. Moguće je kreirati i svoje konstruktore sa proizvoljnim argumentima,slično ostalim programskim jezicima i mogu se proizvoljno imenovati. Opširnije o ovome u dokumentaciji [2]. U primeru (3) vidimo definisanje klase zaposleni (detaljnije 3.1).

```
@interface Employee : NSObject {
    double salary; @public int age;
}
@property(nonatomic, readwrite) double salary;
- (void)display;
@end

@implementation Employee
@synthesize salary;
- (void)display { NSLog(@"Employee salary is %f", salary); }
@end

@interface Driver : Employee {
    NSString* truck;}
- (id)initWithTruck:(NSString*)model;
@end

@implementation Driver
- (id)initWithTruck:(NSString*)model {
    truck = model; return self;
    }
- (void)display { NSLog(@"Driver salary is %f", salary); }
@end
int main(int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Employee *empl = [[Driver alloc]initWithTruck:@"Mercedes"];
    empl.salary = 5.0;  empl->age = 33;
    [empl display];
    [pool drain];
    return 0;
}}
```

Listing 3: Primer koda u Objective C jeziku

3.1 Enkapsulacija

Atributi klase su automatski privatni, što pogoduje enkapsulaciji. Njima se omogućuje pristup izvan klase preko ključne reči **@property**, uz implementaciju pristupnih metoda. Moguće automatsko generisanje metoda pristupa, navođenjem atributa uz ključnu reč **@synthesize**. Primer(3), main funkcija sadrži pozive pristupnim metodama pomoću operatora (.), dok se javnim atributima pristupa preko operatora (->).

3.2 Nasleđivanje

Ovaj koncept označavamo sa (:) i imenom klase koju nasleđujemo. Primer (3), klasa vozač nasleđuje klasu zaposleni .Postoji nasleđivanje u više nivoa i hijerarhijsko. Ako želimo iz izvedene klase da zovemo metode bazne, to radimo referisanjem na baznu klasu pomoću ključne reči **super**.

3.3 Polimorfizam

Višestruka upotrebljivost koda u vidu preopterećenosti (eng. ***overloading***) nije omogućeno u Objective C jeziku, tako da se metodi moraju različito imenovati[3]. Koncept važnosti metoda(eng. ***overriding***) postoji, i prikazan je u primeru (3). Gde klase zaposleni i vozač imaju isti metod display i poziv [empl display] će izvršiti metod klase vozač, zato što promenljiva zaposleni empl referiše na objekat tipa vozač.

3.4 Apstrakcija

Jezik Objective C nema definisan koncept apstraktnih klasa[3], sličan efekat je moguće postići programerskom snalažljivošću i ne instancirati klasu koja bi trebalo biti apstraktna. Za interfejse koje ovde nazivamo protokoli, vezana je ključna reč **@protocol** jer je **interface** rezervisana za klase. Sekcija protokola se završava sa **@end** i može sadžati dve podoblasti **@required** za metode koji se obavezno moraju implementirati u klasi i **@optional** za metode čija je implementacija opciona.

4 Java

Kako su klase su u centru zbivanja krenućemo od njih. Definišemo ih pomoću ključne reči **class**. Sledeći primer (4) prikazuje kreiranje klase zaposleni, koja ima svoje atribute i metode(detaljnije 4.1). Objekte klasa instanciramo pomoću metoda konstruktora, koji nema povratni tip i uvek se zove isto kao i klasa. Ako ne definišemo konstruktor, automatski se generiše podrazumevani konstruktor[5], koji inicijalizuje objekat na podrazumevane vrednosti.

```
1000 public class Employee {
       private int salary;
1002   #this je referenca na tekuci objekat
       public Employee(int salary) { this.salary = salary;}
1004   public int getSalary(){ return salary;}
       public void setSalary(int newSalary) { salary = newSalary;}
1006   public void display() {
             System.out.println("Hello i'm employee!");
1008    }
       public static void main(String[] args) {
1010         Employee Marko = new Driver(600,"Mercedes");
             Marko.display();}
1012 }
     class Driver extends Employee {
1014     String truck = "FAP";
       #super vrsi poziv konstruktora bazne klase
1016     public Driver(int salary,String truck) {
         super(salary); this.truck = truck;}
1018     public void display() {
         System.out.println("My truck is "+truck+"!");
1020   public void display(String x) {
         System.out.println("My truck is "+truck+x+"!");
1022 }
```

Listing 4: Primer deklarisanja klase sa enkapsulacijom i nasleđivanjem

4.1 Enkapsulacija

Ograničavanje pristupa internim podacima klase postižemo navođenjem ključne reči ***private*** ispred deklaracije promenljive u klasi. Private modifikator pristupa znači da se podacima može pristupiti isključivo iz deklarisane klase. Modifikatori i njihova vidljivost prikazani u tabeli 1. Ako su podaci ipak potrebni van klase, omogućuje se njihovo čitanje i menjanje, preko javnih metode pristupa (eng. *getters and setters*)[5]. U primeru(4), vrednosti privatnog atributa plata možemo pristupiti metodom getSalary() ili menjati sa setSalary(newSalary).

4.2 Nasleđivanje

Za nasleđivanje koristimo ključnu reč **extends**. Obratiti pažnju na primer(4), gde klasa vozač nasleđuje svojstva klase zaposleni. U javi postoji

Tabela 1: Vidljivost različitih modifikatora pristupa.

Modifikator	Klasa	Paket	Podklasa	Svet
public	Da	Da	Da	Da
protected	Da	Da	Da	Ne
podrazumevani	Da	Da	Ne	Ne
private	Da	Ne	Ne	Ne

nasleđivanje po nivoima i hijerarhijsko. Višestruko ne postoji direktno[6], već se implementira preko interfejsa (detaljnije 4.4).Ako želimo iz izvedene klase da zovemo metode bazne, to radimo referisanjem na baznu klasu pomoću ključne reči **super**.

4.3 Polimorfizam

Višestruka upotrebljivost koda, tj. pripadnost metoda objektu se obavlja u vreme izvršavanja(eng. *run time execution*) i predstavlja koncept važnosti metoda(eng. ***overriding***)[5]. U primeru koda 4, Marko.display(); pozvaće metod klase vozač, iako postoji isti metod bazne klase.

Koncept preopterećenosti metoda(eng. ***overloading***)[5], određuje metode u vremenu kompajliranja(eng.*compile time*) na osnovu razlika u potpisu metode(različito ime metoda ili tipovi i broj parametara). U primeru koda [4], Marko.display(2); pozvaće metod display(int x) klase vozač.

4.4 Apstrakcija

Izdvajanje skupa metoda sa kojima spoljašnji korisnik komunicira, prema artiklu [6], vršimo pomoću apstraktnih klasa ili interfejsa.

Za apstraktne klase navodimo ključnu reč ***abstract***(kod 5). Ne mogu se instancirati, ali može biti tip promenljive. Sadrže apstraktne metode koje treba da predefiniše neka podklasa.

```
1000 public abstract class Employee {
       public abstract void display(); ...
```

Listing 5: Apstraktna klasa

Interfejs predstavlja nacrt klase. Sadrži apstraktne, statične, podrazumevane metode(mogu se predefinisati u klasi) i statičke promenljive. Da implementiramo interfejs navodimo ključnu reč ***implements***(kod 6) i zatim ime interfejsa. Prednost interfejsa[5] je da klasa može implementirati više interfejsa, dok može da nasleđuje samo jednu klasu.

```
1000 interface Employee {
       public void display(); #podrazumevano apstraktna
1002   default void work(){System.out.println("Working"); }
```

Listing 6: Interfejs

5 C#

C# je jednostavan, moderan, objektno-orijentisan jezik, razvijen od strane Microsoft-a I odobren od strane ECMA-e(European Computer Manufacturers Association). Nudi punu podršku objektno orijentisanom pro-

gramiranju uključujući nasleđivanje, enkapsulaciju, apstrakciju, I polimorfizam.

Ne podržava druge paradigme ali koristi svoje imperativne strukture. Veome je slična podrška OOP-u kao kod Java programskog jezika takođe su iste I klase I strukture.

5.1 Enkapsulacija

Enkapsulacija je kada se grupa od povezanih metoda, svojstava I ostalih članova tretira kao jedan isti objekat. C# podrzava sledeće specifikatore pristupa:

- Public: Dopušta klasi da izloži varijable i funkcije članova drugim funkcijama i objektima. Svakom public članu se moze pristupiti izvan klase.
- Private: Dopušta klasi da sakrije promenljive i funkcije članova od ostalih funkcija i objekata.Samo funkcije iz iste klase mogu pristupiti svojim privatnim članovima. Čak i instanca klase ne moze pristupiti svojim privatnim članovima.
- Protected: Dopušta podređenoj klasi da pristupi varijablama i funkcijama članova svoje osnovne klase. Na taj način pomaže u implementiranju nasleđivanja.
- Internal: Dopušta klasi da izlozi varijable i funkcije članova drugim funkcijama i objektima u trenutnom skupu. Tačnije svakom članu sa ovim specifikatorom pristupa se moze pristupiti iz bilo koje klase ili metode definisane unutar aplikaciju u kojoj je definisan clan.
- Protected internal: Dopušta klasi da sakrije promenljive i funkcije članova od ostalih funkcija i objekata klase, osim podređene klase unutar iste aplikacije.To se takodje koristi kod implementacije nasleđivanja.

5.2 Nasleđivanje

Nasleđivanje je, kao sto samo ime kaze, mogućnost da "nasleđuje" metode I svojstva od postojećih klasa. Kod c# nasleđivanja, klasa čiji su članovi nasleđeni se zove bazična (roditeljska) klasa, a klasa koja nasleđuje članove bazične klase se zove izvedena (podređena) klasa.

C# Koristi sintaksu c++ za definisanje klasa. Nasleđena metoda od roditeljske klase moze biti zamenjena u izvedenoj klasi tako sto se definiše kao new(novo). Verzija roditeljske klase se I dalje moze zvati eksplicitnom sa prefiksom base(baza): base.Draw().

5.3 Polimorfizam

U c#-u, polimorfizam omogućava klasama da da implementiraju različite metode koje se nazivaju istim imenom i omogućava pozivanje metoda izvedene klase kroz referencu bazične klase tokom izvođena, na osnovu naših zahteva.

U c#-u imamo dve različite vrste polimorfizma a to su: Polimorfizam vremena kompiliranja i polimorfizam vremena izvođenja.

5.4 Apstrakcija

Apstrakcija je proces kod koga programer krije sve osim relevantnih podataka o datom objektu u cilju pojednostavljanja I povećanja efikasnosti. U c#-u mozemo napraviti klasu sa potrebnim metodama i svojstvima. Izlažemo samo potrebne metode i svojstva koristeći modifikatore pristupa na osnovu nasih zahteva, videti primer (7).

```
abstract class MobilePhone {
    public void Calling();
    public void SendSMS();
}
public class Nokia1400: MobilePhone {}
public class Nokia2700: MobilePhone {
    public void FMRadio();
    public void MP3();
    public void Camera();
}
public class BlackBerry: MobilePhone {
    public void FMRadio();
    public void MP3();
    public void Camera();
    public void Recording();
    public void ReadAndSendEmails();
}
```

Listing 7: Primer deklarisanja apstraktivne klase u C#-u

6 Ada

Ada sledi model klase zasnovan na odvojenim karakteristikama (tipovima(eng. *types*) deklarisanim u jedinicama paketa(eng. *package*)), a ne na jednom konstruktoru klase[4]. Klasa u Adi ima koncept klase tipa koja se sastoji od skupa tipova kreiranih unutar deklaracije osnovnog tipa(eng. *root type*) unutar paketa. Svakom tipu je pridružena oznaka(eng. *tag*) koja ga razlikuje od ostalih tipova dodavanjem ključne reči **tagged** u deklaraciju osnovnog tipa. Unutar paketa možemo imati procedure(nemaju povratnu vrednost) i funkcije(imaju povratnu vrednost). U nastavku možemo videti primer(8) OOP u Adi.

```
package Employees is
  type Employee is tagged
  record
    Name: String;
  end record;
  procedure Set_Name(Obj: in out Employee; Name: Name_Type);
end Employees;
package Drivers
type Driver is new Employees.Employee with record
  null;
end record;
```

Listing 8: Primer objektno orijentisanog programiranja u jeziku Ada.

6.1 Enkapsulacija

U Adi, enkapsulacija je malo drugačija od većine OOP jezika jer je privatnost generalno određena na nivou paketa[1]. Pomoću ključne reči **private** u deklaraciji baznog tipa obezbeđujemo da podacima unutar tipa pristupamo jedino pomoću funkcija i procedura. Još veći stepen privatnosti dobijamo dodavanjem ključne reči **limited** ispred ključne reči **private**.

Procedure i funkcije unutar paketa deklarišemo kao privatne ukoliko ispred niza deklaracija procedura i funkcija dodamo ključnu reč **private**.

6.2 Nasleđivanje

Nasleđivanje po nivoima i hijerarhijsko nasleđivanje je direktno podržano u Adi, dok je višestruko nasleđivanje moguće implementirati pomoću interfejsa(eng. *interface*) uz dodatni rad programera. U primeru(8) deklaracija klase *Driver* je tipičan primer kreiranja izvedenog tipa. Izvedeni tip nasleđuje sve podatke unutar tipa, kao i sve procedure i funkcije rodjiteljskog tipa.

6.3 Polimorfizam

6.4 Apstrakcija

7 Ruby

Podršku u programskom jeziku Ruby ilustrovaćemo primerom(9) koji pokriva sve bitnije aspekte objektno orijentisanog programiranja. Standardni metod klase je **initialize**, on se poziva automatski prilikom kreiranja objekta i ponaša se skoro identično kao konstruktori u drugim programskim jezicima.

```
class Employee
attr_accessor :name
def initialize(name)
  @name = name
  print()
end
def print
  puts "Employee: #{@name}."
end
end
class Driver < Employee
  def initialize(name)
    @name = name
    print()
  end
  private
  def print
    puts "Driver: #{@name}."
  end
end

emp = Employee.new("John")
drv = Driver.new("John")
```

Listing 9: Primer objektno orijentisanog programiranja u jeziku Ruby.

7.1 Enkapsulacija

Kako u samom jeziku ne postoji mogućnost direktnog pristupa podacima unutar klase(podaci su privatni), njima možemo pristupiti jedino pomoću metoda klase. Svi metodi klase su javni, osim ako nije eksplicitno naznačeno drugačije ključnim rečima **public protected**, **private** neposredno pre definicije jednog ili više metoda. Ruby nam pruža mogucnost ugrađenih metoda za pristup(eng. *accessor methods*). U primeru(9) **attr_accessor** omogućava čitanje i menjanje vrednosti promenljivih klase. Pomocu **attr_reader** i **attr_writer** možemo pojedinačno dopustiti samo čitanje odnosno samo menjanje vrednosti promenljivih.

7.2 Nasleđivanje

Kada nakon imena klase u njenoj definicije dodamo znak $<$ za kojim sledi ime već postojeće klase, dobijamo efekat nasleđivanja koji možemo videti u prethodnom primeru(9) gde klasa *Driver* nasleđuje klasu *Employee*(primetiti da u klasi *Driver* nismo implementirali **attr_accessor** jer se nasledjuje). Nasleđivanje po nivoima i hijerarhijsko nasleđivanje je moguće dok višestruko nasleđivanje nije(više o tipovima nasleđivanja u 4.2).

7.3 Polimorfizam

Osnovni vid polimorfizma možemo postići nasleđivanjem tako sto ce razliciti objekti odgovoriti različito na iste metode. U primeru(9) u klasi *Driver* smo definisali metod *print* koji je istog naziva kao i nasleđeni metod čime postižemo da instanca klase odgovori različito na metod od instance roditeljske klase.

Drugačiji vid polimorfizma postižemo takozvanim "pačijim kucanjem"(eng. *duck typing*) u kojem nisu bitni tipovi objekata već skup istoimenih metoda koje poseduju. Za primer uzmimo klasu *Duck* koja poseduje metod *quack* i funkciju koja za argument uzima objekat tipa *Duck* i poziva metod *quack*. U tom slučaju funkciji možemo proslediti bilo koji objekat koji poseduje metod naziva *quack*(sa istim ili različitim ponašanjem metoda) i gledati na njega kao da je tipa *Duck* bez obzira što on to nije.

7.4 Apstrakcija

Ruby nema direktnu podršku za apstrakciju klasa ali se sličan efekat može dobiti korišćenjem biblioteke "abstract". Takođe je moguće implementirati apstakciju pomoću nasleđivanja gde rodjiteljska klasa definiše apstraktne metode koji pokrecu "NotImplementedError"grešku tako da se ne mogu instancirati, pa mora postojati dete klasa koja će pomocu gore opisanog polimorfizma(7.3) nasleđivanjem implementirati željene apstraktne metode. Metode koje su zajedničke implementiramo u roditeljskoj klasi.

8 Zaključak

Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak.

Literatura

[1] Introduction to Ada. on-line at: https://learn.adacore.com/courses/intro-to-ada/index.html.

[2] Object C apple documentation. on-line at: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjectiveC.

[3] Gary Bennett, Brad Lees, and Mitchell Fisher. *Objective-C for Absolute Beginners: IPhone, iPad and Mac Programming Made Easy*. Apress, Berkely, CA, USA, 3rd edition, 2016.

[4] AdaCore experts. *High-Integrity Object-Oriented Programming in Ada*. AdaCore(www.adacore.com), 1.2 release edition, 2011. on-line at: http://extranet.eu.adacore.com/articles/HighIntegrityAda.pdf.

[5] Cay S Horstmann. *Core Java SE 9 for the Impatient*. Addison-Wesley Professional, 2017.

[6] Aayushi Johari. Object Oriented Programming – Java OOPs Concepts With Examples, 2018. on-line at: https://www.edureka.co/blog/object-oriented-programming/.