Objektno orjentirano programiranje

Predavanje 6

Nasljeđivanje, kompozicija, apstraktne klase i sučelja

Nasljeđivanje i kompozicija

- Prilikom odlučivanja o dizajnu jedna od najvažnijih odluka je kada primijeniti nasljeđivanje, a kada kompoziciju
- Nasljeđivanja klasa u sebi implicira nasljeđivanje atributa i ponašanja drugih klasa
- Postoji stvarni odnos roditelj-dijete
- Kompozicija se odnosi na stvaranje novih objekata koristeći već postojeće objekte

Ponovno korištenje (eng. Reusing) objekata

- Reuse je jedan od najvažnijih razloga zašto nasljeđivanje i kompozicija postoje
- Nasljeđivanje predstavlja *is-a* odnos (*dog is-a animal)*
- Kompozicija uključuje druge klase da bi se stvorile nove kompleksnije klase. U ovom slučaju ne postoji *is-a* odnos
- Kompozicija predstavlja *has-a* odnos (*car has-a engine)*
- Klasa *engine* u sebi može imati druge klase (*cilindar, ventili …*)
- Danas je još aktualna rasprava da li je učinkovito uopće koristiti nasljeđivanje ili se bolje prikloniti kompoziciji

Ponovno korištenje (eng. Reusing) objekata

- I nasljeđivanje i kompozicija su ispravne tehnike dizajna klasa i svaka ima svoje mjesto u cjelokupnoj slici OO programiranja
- Korištenje nasljeđivanja možemo utvrditi sljedećim pravilom:
  - Ako možemo reći „*Klasa A je (is-a) Klasa B”*, tada je ovaj odnos dobar kandidat za nasljeđivanje
- Klasa Pas ima svoj svojstvene karakteristike koje je čine različitom od npr. klase Mačka
- Recimo da želimo napraviti klasu

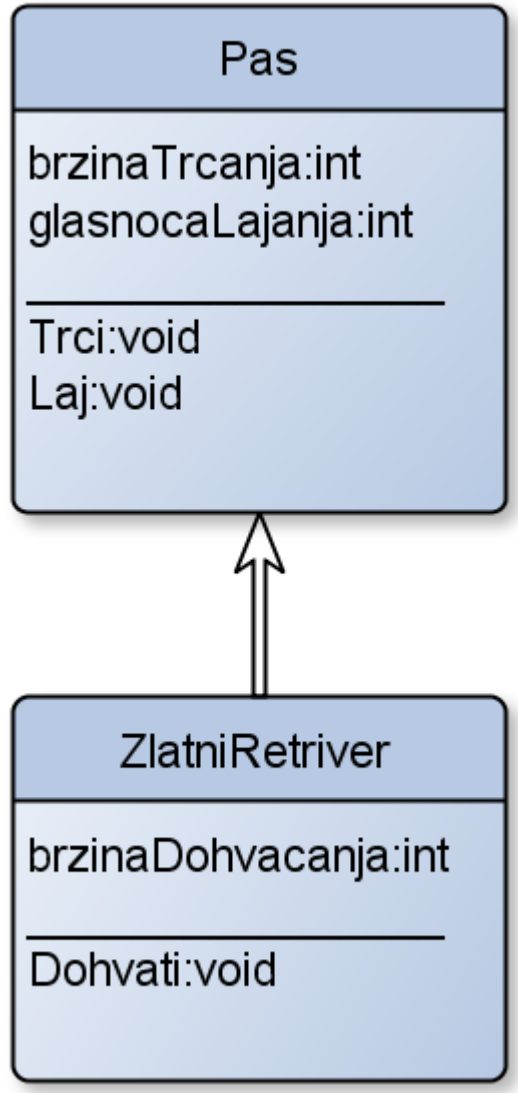
ZlatniRetriver

Nasljeđivanje

- ZlatniRetriver može biti svoja odvojena klasa ali može i nasljeđivati klasu Pas
- ZlatniRetriver je Pas (vrijedi *is-a* odnos)
- Sada Zlatni retriver sadrži karakteristike Psa,

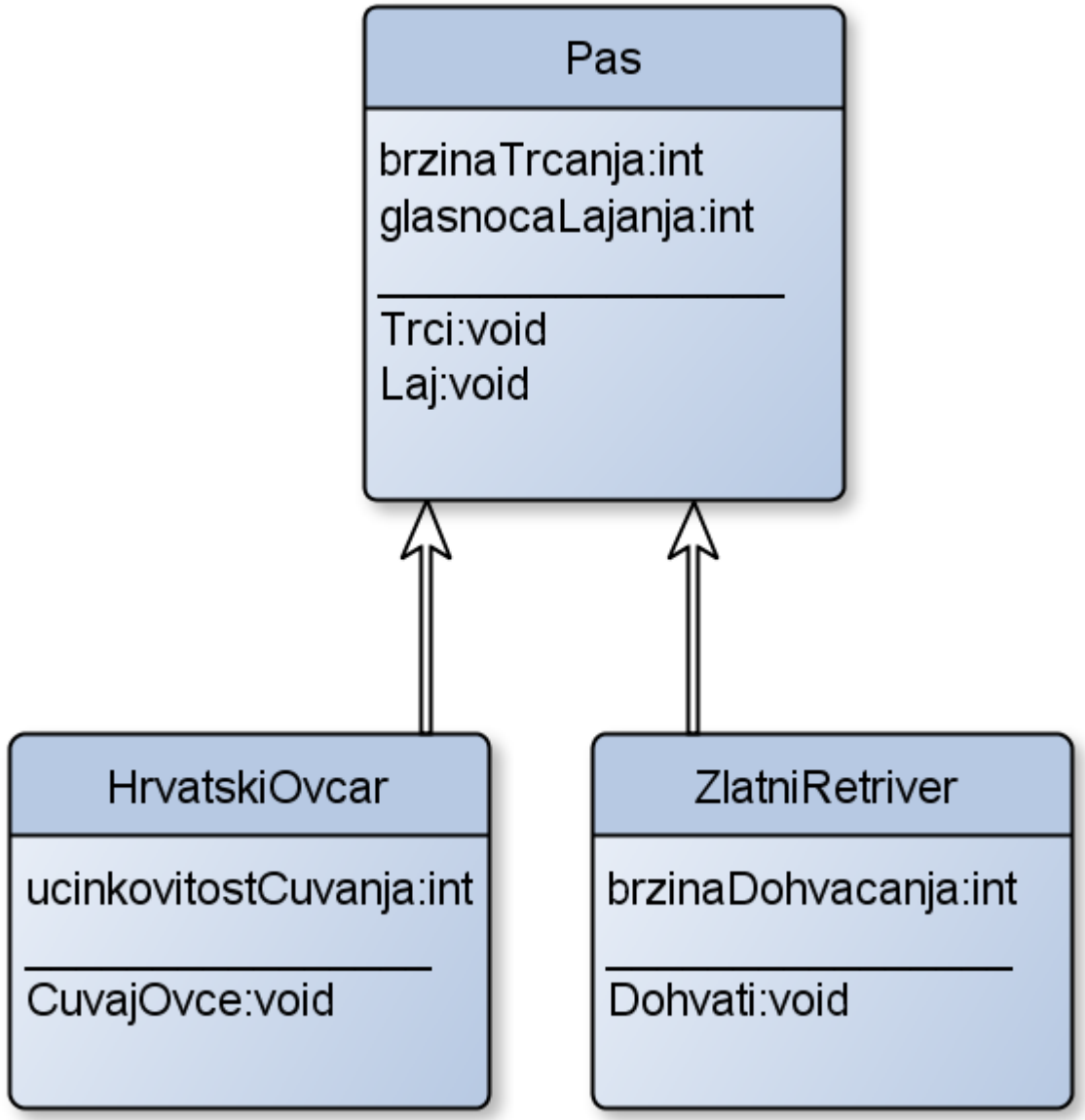
ali i neke svoj specifične atribute

- Metode koje smo naslijedili iz klase Pas ne trebamo ponovno kodirati što nam štedi vrijeme i resurse, i isto tako ih ne trebamo testirati jer su već testirane
- Isto vrijedi i za buduće održavanje
- Neka testiranja ipak trebamo obaviti radi novih interfejsa



Nasljeđivanje

- Recimo da želimo napisati klasu HrvatskiOvčar
- ZlatniRetriver ima funkciju *donošenja* a ovčar ima funkciju čuvanja ovaca
- Još jedna prednost nasljeđivanja je da je kod za funkcionalnost *lajanje* i *trčanje* na jednom mjestu, pa prilikom mijenjanja koda ne trebamo mijenjati sve klase



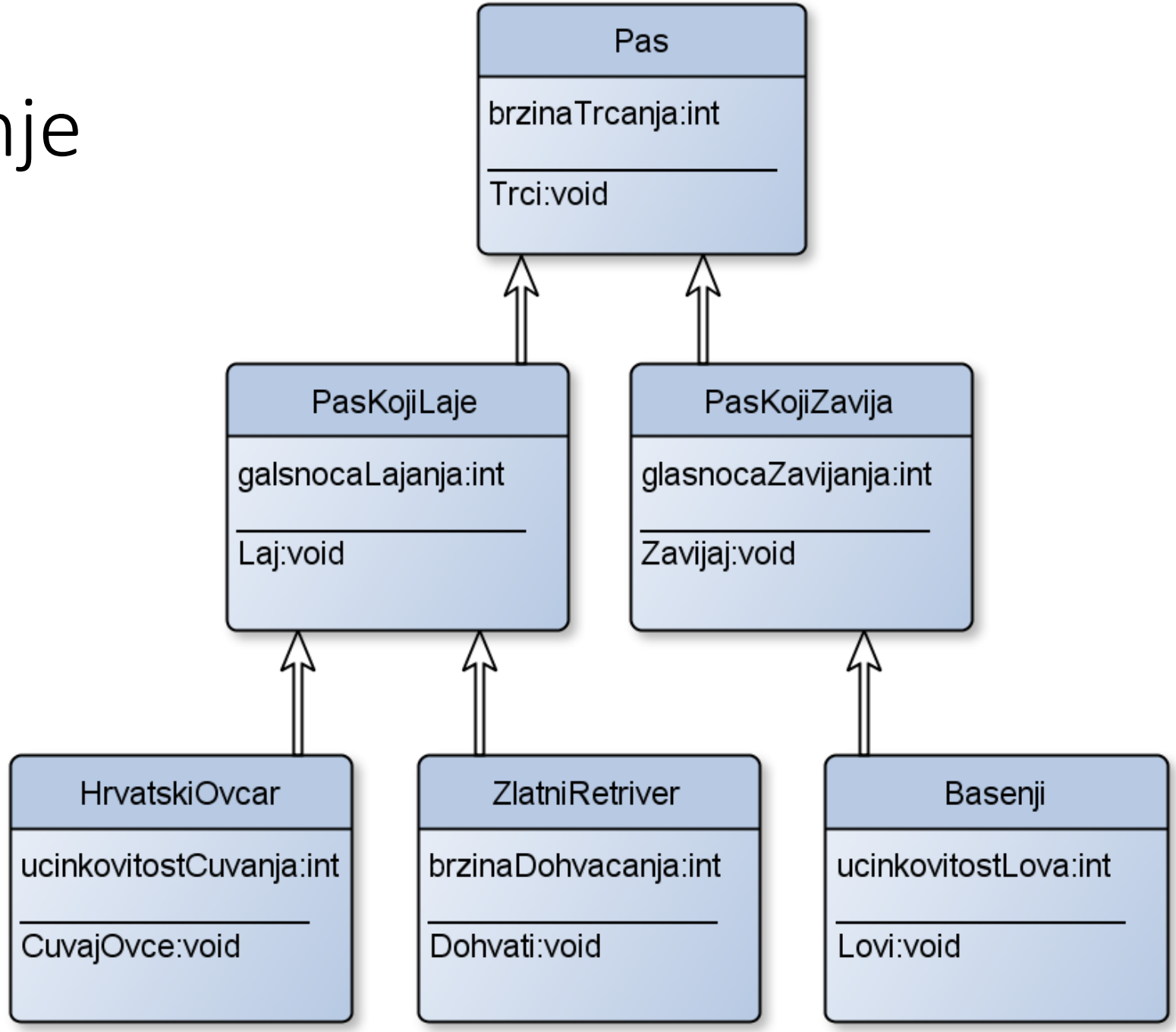
Nasljeđivanje

- Na ovom nivou nasljeđivanje izgleda da funkcionira bez problema
- Kako možemo biti sigurni da svi psi imaju ponašanje specificirano u klasi
- Uzmimo za primjer klasu Ptica, jedno od najprepoznatljivijih ponašanja je da ptica leti
- Napravimo klasu Ptica sa metodom *Poleti*
- Međutim, postoje ptice koje ne lete (noj, pingvin)
- Što napraviti kod ovakvih iznimki?
- Možemo lokalno preopteretiti metodu za iznimke, ali ona će se i dalje zvati *Poleti* što nema baš previše smisla za ptice koje plivaju ili hodaju

Nasljeđivanje

- Kada bi pingvin išao testirati svoju metodu *Poleti* iznad litice, neugodno bi se iznenadio
- Slična stvar je sa programerima koji bi koristili našu klasu
- U našoj klasi Pas koristili smo metodu *lajanje*, međutim postoje psi koji ne laju već *zavijaju*
- Kako da u ovom slučaju promijenimo naš dizajn?
- U programiranju ćemo se susretati s ovakvim situacijama kada trebamo mijenjati dizajn
- Dizajn je iterativan proces

Nasljeđivanje



Generalizacija i specijalizacija

- Prilikom definicije osnovne klase Pas pokušali smo izvući neke zajedničke i osnovne osobine pasa
- To je najgeneralniji slučaj
- Kako se spuštamo u stablu nasljeđivanja stvari postaju specifičnije/specijaliziranije
- Koncept generalizacija-specijalizacija (eng. Generalization-specialization)
- Mogli smo također odlučiti da nećemo posebne klase PasKojiLaje i PasKojiZavija već ćemo to koristiti kao svojstva specifičnih klasa

Odluke pri dizajnu

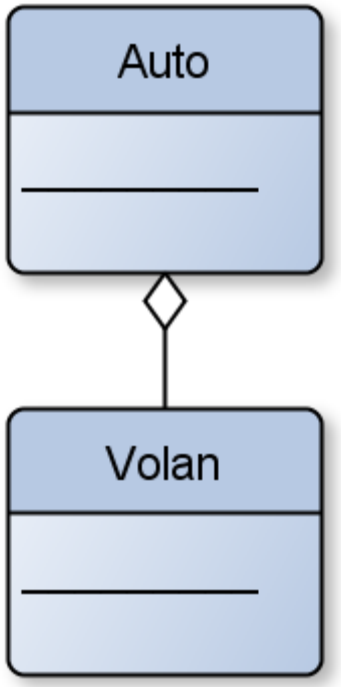
- Teoretski gledano – cilj je izvući što više zajedničkih značajki
- Na takav način bolje i pobliže predstavljamo stvarni svijet, ali u isto vrijeme naš sustav postaje kompleksniji
- Da li želimo što točniji model ili model koji je manje kompliciran?
- Ovakve odluke se donose temeljeno na danoj situaciji i ne postoje generalne upute
- U našem slučaju sa klasom Pas, sustav nije kompliciran, ali u velikim sustavima kompleksnost brzo raste
- Kod velikih sustava je najbolja praksa držati se pristupa koji omogućava manju kompleksnost i lakše razumijevanje

Odluke pri dizajnu

- Postojat će slučajevi kada točniji model ne znači nužno i veću kompleksnost
- Postoje situacije kada svjesno izbjegavamo točnost modela jer znamo da nam neće trebati
- Recimo da radimo aplikaciju za uzgajivača pasa
- Uzgajivač zna da neće imati vrste pasa koji bi spadali u klasu PsiKojiZavijaju (psi koji ne laju)
- Odluka veća točnost/manja kompleksnost je stvar balansiranja
- Cilj je da imamo sustav koji je i fleksibilan i jednostavan
- Trenutačni i budući troškovi su također bitna stavka pri donošenju odluke u dizajnu – kasnija mijenjanja sustava su skupa (što ako se odlučimo prodati sustav i uzgajivačima pasa koji ne laju)

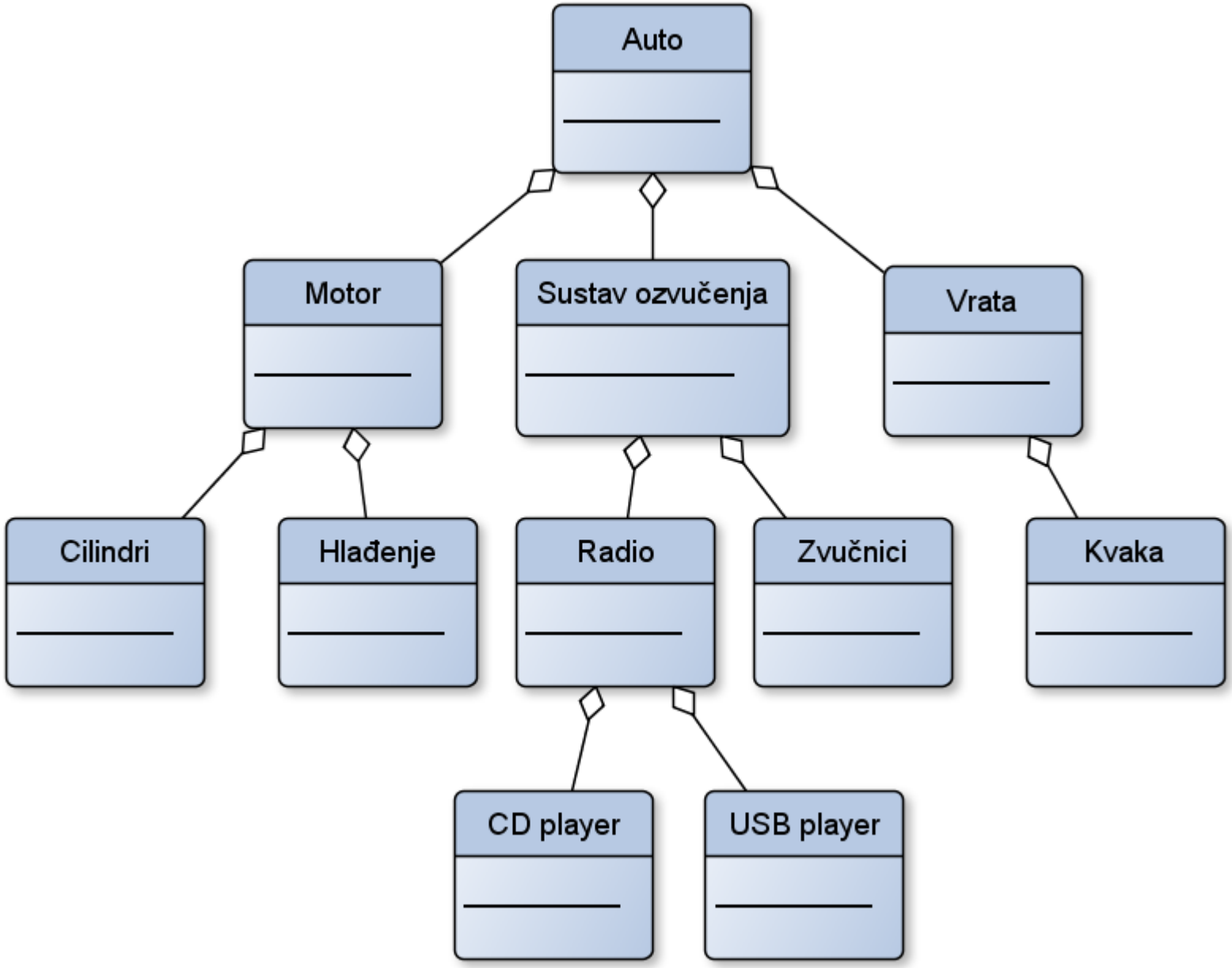
Kompozicija

- Tipičan primjer je automobil koji se sastoji od mnogo različitih djelova
- Računalo se sastoji od mnogo elemenata poput CPU,hard drive, …
- Predstavljanje kompozicije putem UML dijagrama
- *Has-a* relacija



Kompozicija

- Pretpostavimo da auto želimo predstaviti bez korištenja nasljeđivanja i samo koristeći kompoziciju



Kompozicija

- Kao i kod nasljeđivanja, previše korištenja kompozicije vodi većoj kompleksnosti
- Nivo kompleksnosti treba odrediti dizajner
- Potrebno je pronaći balans između detaljnog prikaza svijeta i lakoće održavanja i razumijevanja modela

Enkapsulacija i OO programiranje

- Gilbert i McCarty definiraju enkapsulaciju kao *„proces pakiranja programa, dijeleći svaku klasu u dva zasebna dijela: interfejs i implementaciju”*
- Enkapsulacija je fundamentalni koncept OO programiranja
- Nasljeđivanje je također jedan od tri osnovna koncepta OO programiranja
- Međutim, nasljeđivanje na jedan način slabi enkapuslaciju
- Na koji način nasljeđivanje slabi enkapsulaciju?

Nasljeđivanje i enkapsulacija

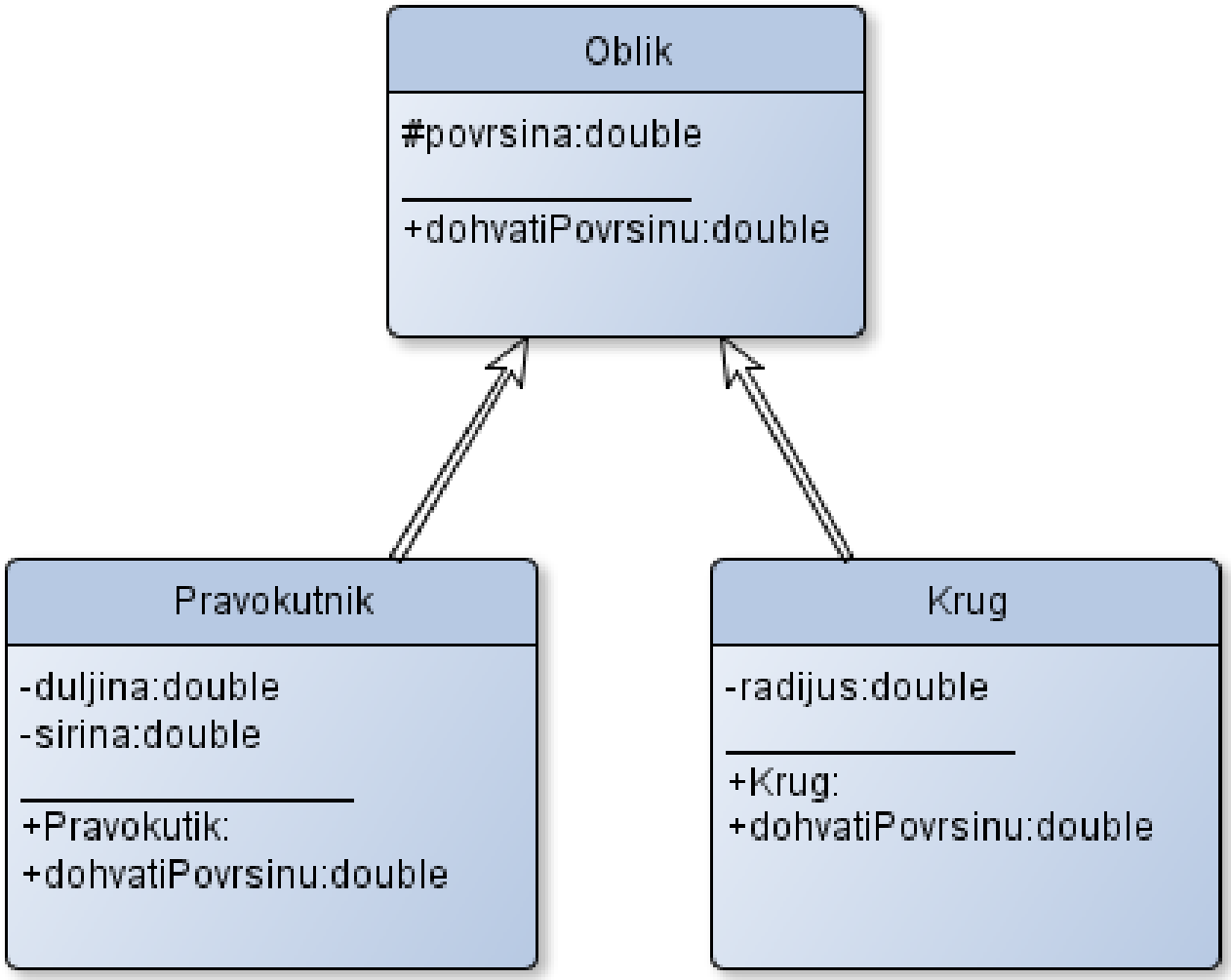
- Enkapsulacija inherentno oslabljuje unutar hijerarhije klasa
- Nasljeđivanje održava enkapsulaciju prema ostalim klasama, ali oslabljuje enkapsulaciju između nad-klase (eng. superclass) i pod-klasa (eng. subclasses)
- Ako naslijedimo implementaciju iz nad-klase i kasnije mijenjamo tu implementaciju ta promjena se propagira kroz hijerarhiju klase
- Ta promjena potencijalno utječe na sve pod-klase
- Na prvi pogled se ovo ne čini kao veliki problem, ali ta promjena može prouzrokovati neočekivane situacije
- Testiranje može postati veliki problem jer se ponovno treba testirati cijela hijerarhija

Polimorfizam

- Mnogi ljudi smatraju polimorfizam jedim od temeljnih koncepata OO preogramiranja
- Svrha dizajniranja klasa je stvaranje zasebnih objekata
- Objekt bi trebao biti odgovoran za sebe
- Polimorfizam znači – *mnogo oblika*
- Kada se poruka pošalje objektu – on treba znati kako odgovoriti
- U hijerarhiji nasljeđivanja sve pod-klase nasljeđuju interfejs on nad-klase
- Međutim, budući da je svaka pod-klasa zasebna cjelina postoji mogućnost da može imati svoj specifičan odgovor na tu poruku

Polimorfizam

- Premisa polimorfizma je da možemo poslati poruke različitim objektima, a njihov odgovor ovisi o tipu objekta
- Svaki objekt je odgovoran za sebe
- Preopterećenje metoda znači da implementaciju roditelja zamijenimo sa svojom implementacijom
- Klasa Oblik ne može vratiti površinu jer ne zna kako je izračunati



Polimorfizam - primjer

- Svaki objekt je odgovoran za sebe

```
class Oblik
{
public:
	virtual void Iscrtaj()=0;
};
```

```
class Pravokutnik : public Oblik
{
public:
	void Iscrtaj()
	{
		cout << "Ja sam pravokutnik!"
	}
};
```

```
class Krug : public Oblik
{
public:
	void Iscrtaj()
	{
		cout << "Ja sam krug!" << endl;
	}
};
class Trokut :public Oblik
{
public:
	void Iscrtaj()
	{
		cout << "Ja sam trokut!" << endl;
	}
};
```

Polimorfizam - primjer

```
int main()
{
        Pravokutnik p;
        Krug k;
        Trokut t;
        p.Iscrtaj();
        k.Iscrtaj();
        t.Iscrtaj();
        return 1;
}
```

```
Ja sam pravokutnik!
Ja sam krug!
Ja sam trokut!
```

Polimorfizam - primjer

- Iako klasa Oblik ne zna kako iscratati trokut, klasa Trokut to zna.

```
class Trokut :public Oblik
{
public:
        void Iscrtaj()
        {
                cout << "Ja sam trokut!" << endl;
        }
};
```

Polimorfizam - primjer

```
Pravokutnik p;
Krug k;
Trokut t;
Test test;
Oblik* o1, *o2, *o3;
o1 = &p;
o2 = &k;
o3 = &t;
test.CrtanjeLika(o1);
```

```
class Test
{
public:
        void CrtanjeLika(Oblik* o)
        {
                cout << "Klasa Test iscrtava:";
                o->Iscrtaj();
        }
};
```

- Metoda koja barata s oblicima ne mora znati s kojim točno oblikom trenutno radi

Apstraktne klase

- Apstraktna klasa ima apstraktne/virtualne metode
- Iz tog razloga ne može bit instancirana
- Klasa Oblik ima apstraktnu metodu *dohvatiPovrsinu()* koja nema implementaciju
- Implementaciju (preopterećenje) imaju samo klase koje ju nasljeđuju

```
//Implementacija apstraktne klase C++
class Oblik
{
        public:
        virtual void Iscrtaj()=0;
}
```

```
//Implementacija apstraktne klase Java
public abstract class Oblik
{
        public abstract void Iscrtaj();
}
```

Framework

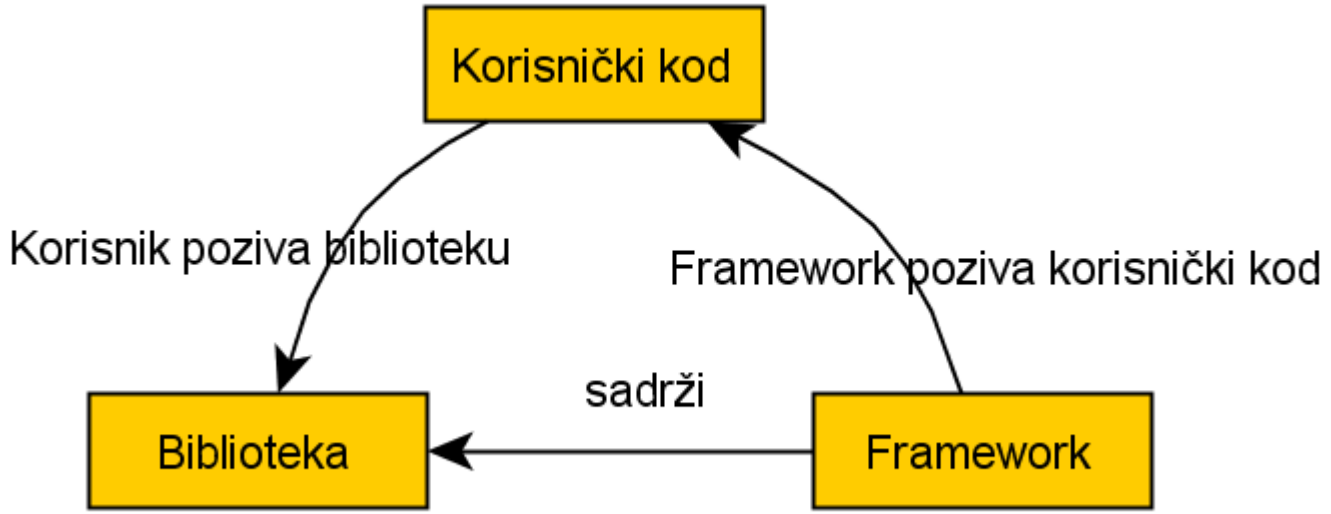
- Ponovno korištenje koda – nije isključivo za OO programiranje
- Framework i standardizacija – jednoznačnost korištenja za programera i za krajnjeg korisnika programa
- Stvaranje prozora u Windowsima je standardizirano sa istim funkcionalnostima
- Kod korišten u frameworku je već testiran
- Postoje framework za različite primjene (npr. word processing – definirane su standardne radnje kao copy-paste, otvaranje, spremanje, pretraga, …)
- Učenje i iznova programiranje već napravljenih stvari nije isplativo

Framework

- Očito pitanje je – ako trebamo koristiti dijalog-box koji nam pruža framework – kako ga upotrijebiti?
- Pravila korištenja se obično nalaze u dokumentaciji framework-a
- Osoba koja piše klasu/klase je zadužena za pisanje dokumentacije kako koristiti javni interfejs klase
- Veće tvrtke obično napišu i koriste svoj framework specifičan za njihove aplikacije
- Postoje i general-purpose frameworks i libraries
- Postoje različite definicije frameworka

Framework

- Po nekim definicijama framework predstavlja veću cjelinu/generalniji pojam od biblioteke, može biti i skup biblioteka
- Framework za razliku od biblioteke upravlja tokom programa – *inversion of control* – tok programa ne diktira korisnik već framework



Framework vs biblioteka

- Biblioteke su stariji koncept, frameworks su „noviji”
- Oboje imaju definiran API sa uputama za korištenje
- Biblioteka se može promatrati kao specifična funkcionalnost aplikacije, a framework kao kostur aplikacije
- Ponekad se ova dva pojma miješaju ovisno o tome tko ih koristi
- Framework nudi proširivost na način na možemo naslijediti klase i pružiti novu/prilagođenu funkcionalnost
- Korištenjem frameworka programeri se mogu posvetiti software zahtjevima, a ne low-level detaljima i stvaranju stand-alone sustava

Contracts

- Mehanizam koji zahtjeva od developera da se pridržava specifikacija API-ja ili frameworka
- To znači da se pridržava načina imenovanja metoda (standard imenovanja) i korištenja putem kojeg se osigurava dobra praksa pisanja koda
- Standard nema smisla ako ga ljudi ne koriste što rezultira nerazumljivim kodom
- U Javi i .NET jezicima pridržavanje se osigurava apstraktnim klasama i interfejsima

Apstraktne klase

- Apstraktna klasa ima jednu li više metoda čija implementacija nije definirana
- Zove se apstraktna je je nije moguće instancirati
- Kako se koristi za *contract?*
- Pretpostavimo da imamo klasu Oblik. Cilj nam je da predstavimo svaku vrstu oblika naknadno (pravokutnik, krug, trokut)
- Želimo da svo oblici koriste istu sintaksu za crtanje, npr. želimo da svi oblici imaju metodu Iscrtaj()
- Također želimo da svaka klasa imaju svoju implementaciju te metode koja je specifična za nju

Apstraktne klase

- Da bi smo implementirali funkcionalnost potrebno je nasljediti klasu Oblik
- Npr. klasa Krug nasljeđuje Oblik
- Ako klasa Krug ne implementira virtualnu metodu tada se i ona smatra apstraktnom klasom i njena djeca trebaju definirati implementaciju

```
//Implementacija apstraktne klase C++
class Oblik
{
        public:
        virtual void Iscrtaj()=0;
}
```

```
class Krug : public Oblik
{
    public:
    void Iscrtaj()
    {
        cout << "Ja sam krug!" << endl;
    }
};
```

Apstraktne klase

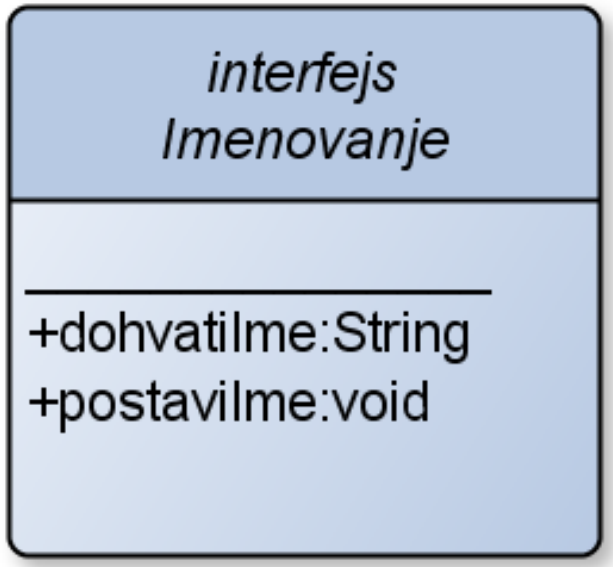
- Definicija apstraktne klase je da ima jednu ili više čistih virtualnih metoda
- To ne znači da apstraktna klasa ne može implementirati neke od svojih metoda
- Neki jezici poput C++ koriste apstraktne klase da bi implementirali ugovore (eng. contracts), a neki, poput Jave i .NET koriste mehanizme koji se zovu interfejsi (eng. interfaces)
- Napomena: nije isto što i interfejsi koji označavaju javno sučelje neke klase, ili možda GUI nekog programa

Sučelja– ver 2.

- Ako apstraktne klase pružaju istu funkcionalnost kao i sučelje zašto Java i .Net imaju posebne mehanizme za interfejse?
- C++ dozvoljava višestruko nasljeđivanje a Java i .Net ne dozvoljavaju
- Zbog toga mogu implementirati više sučelja
- Korištenje više od jedne roditeljske apstraktne klase (na istom nivou hijerarhije) je višestruko nasljeđivanje
- Korištenjem sučelja ne moramo se brinuti oko formalne strukture nasljeđivanja – teoretski možemo dodati sučelje na bilo koju klasu ako ima smisla s aspekta dizajna

Sučelja – ver 2.

- UML dijagram sučelja



- Implementacija sučelja

```
public interface Imenovanje{
    String dohvatiIme();
    void postaviIme(String infs_ime);
};
```

Primjer – nasljeđivanje, kompozicija i sučelja

- Želimo dizajnirati klasu Pas sa mogućnošću dodavanja još sisavaca
- Možemo za početak definirati apstraktnu klasu Sisavac

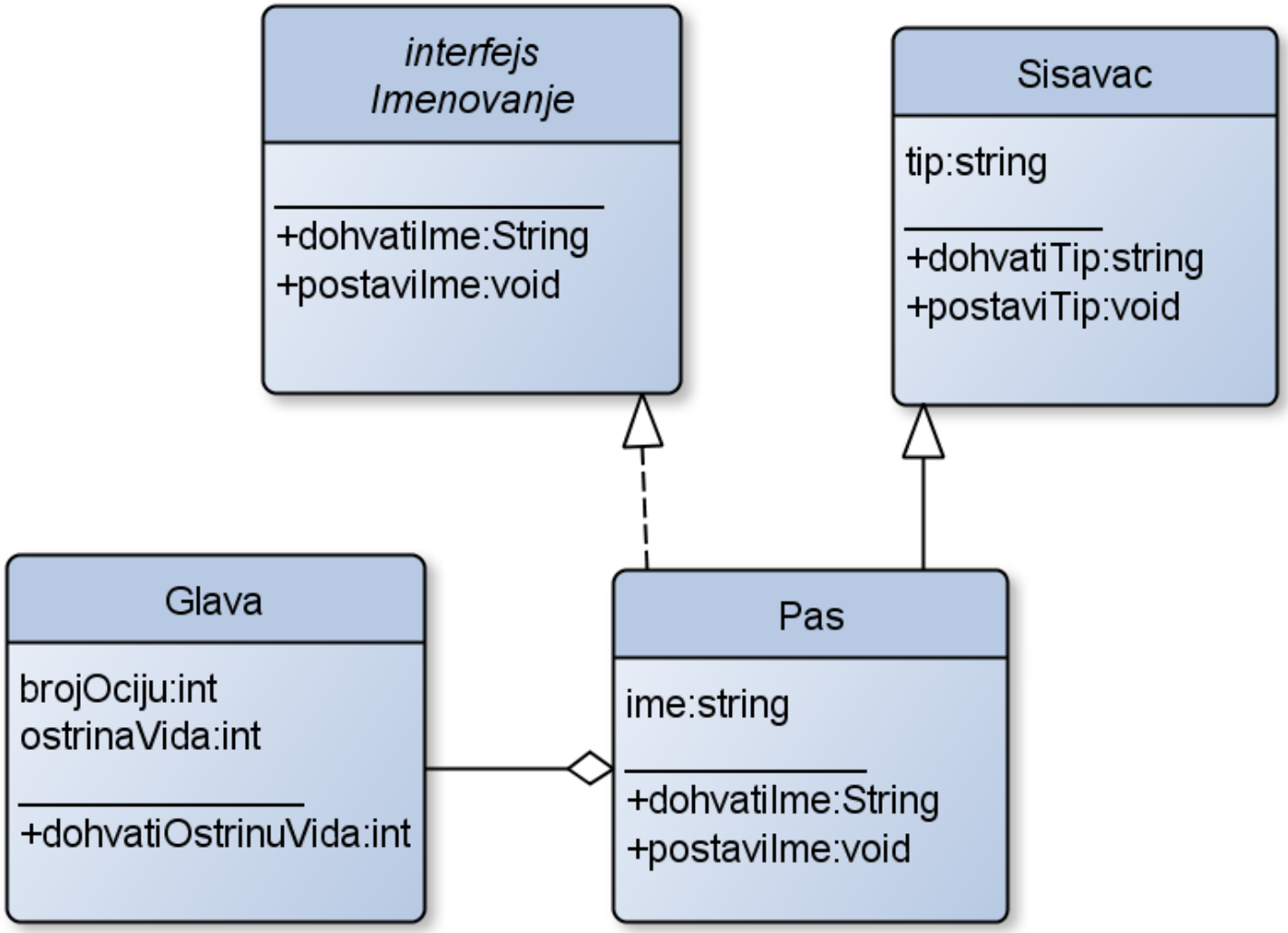
```
public class Sisavac {
     string tip;
    public void PomakniSe()
    {
        System.println("Pokrećem se");
    }
    public abstract void GlasajSe();
};
```

Primjer – nasljeđivanje, kompozicija i sučelja

- Definiramo klasu Glava koja se koristi u kompoziciji

```
public class Glava {
    int brojOciju;
    int ostrinaVida;
    public dohvatiOstrinuVida()
    {
        return ostrinaVida;
    }
};
```

Primjer – nasljeđivanje, kompozicija i sučelja



Primjer – nasljeđivanje, kompozicija i sučelja

- Kod za definiciju klase Pas u Javi

```
public class Pas extends Sisavac implements Imenovanje {
    String ime;
    Glava glava;
    public void GlasajSe(){
        System.out.println("Lajanje");
    }
    public void postaviIme(String infs_ime)
    {
        ime = infs_ime;
    }
    public void dohvatiIme(){
        return ime;
    }
};
```

Razlika između nasljeđivanja i sučelja

- Nasljeđivanje je stroga is-a relacija a interfejs nije
- Pas **je** (*is-a*) Sisavac
- Gmaz **nije** (*is-not*) Sisavac
- Klasa Gmaz prema tome ne može nasljeđivati Sisavca
- Interfejsi nadilaze različitosti klasa
- Pas **je** *imenovan* (može biti imenovan)
- Gušter **je** *imenovan* (može biti imenovan)
- Interfejs **nikada** ne pruža nikakvu vrstu implementacije, samo specificira koje ponašanje neka klasa treba imati

Stvaranje *contract-a*

- Jednostavno pravilo je da se pruži nedefinirana metoda kroz apstraktnu klasu ili interfejs
- Kada se napravi pod-klasa koja nasljeđuje tu klasu ili prihvaća interfejs potrebno je napraviti implementaciju te metode
- Jedna od prednosti contract-a je standardiziranje konvencija kodiranja
- Primjer što se događa kada se ne koriste standardi:

```
public class nogometniKlub{
    string imeNogometnogluba;
    public string dohvatiImeNogometnogKluba(){
        return imeNogometnogKluba;
     }
};
```

```
public class Drzava{
        string imeDrzave;
        public string dohvatiImeDrzave(){
                return imeDrzave;
        }
};
```

```
public class Grad{
    string imeGrada;
    public string dohvatiImeGrada(){
        return imeGrada;
    }
};
```

Contract - primjer

- Javlja se problem da bilo tko ako koristi klase mora proučiti dokumentaciju za svaku od ovih klasa da sazna kako dohvatiti ime
- Dobra praksa je da klase dijele zajedničku konvenciju imenovanja
- Radi se contract za bilo koji tip klase koja koristi ime
- Korisnici ovih klasa bi jednostavnije radili s klasama sa istom sintaksom

Contract - primjer

```
public interface Imenovanje{
  String dohvatiIme();
  void postaviIme(String intfs_ime);
};



public class Drzava implements Imenovanje{
  string imeDrzave;
  public string dohvatiIme(){
    return imeDrzave;
  }
  public void postaviIme(string intfs_ime)
  {
    imeDrzave = intfs_ime;
  }
};
```

```
public class nogometniKlub{
  string imeNogometnogluba;
  public string dohvatiIme(){
    return imeNogometnogKluba;
  }
  public void postaviIme(string intfs_ime)
  {
    imeNogometnogluba = intfs_ime;
  }
};

public class Grad{
  string imeGrada;
  public string dohvatiIme(){
    return imeGrada;
  }
  public void postaviIme(string intfs_ime)
  {
    imeGrada = intfs_ime;
  }
};
```