Nasleđivanje

Vrlo često novi programi nastaju proširivanjem prethodnih. Najbolji način za stvaranje novog softvera je imitacija, doterivanje i proširivanje postojećeg. Tradicionalni metodi razvoja softvera su zanemarivali ovakvo stanovište. U objektno orijentisanom programiranju to je osnovni način rada.

Zamislite slučaj kada imate izvorni kod neke klase. Kako biste mogli da taj kod ponovo iskoristite. Mogli biste da ga iskopirate i u kopiji menjate ono što je potrebno. Kod tradicionalnih jezika to je bila jedina mogućnost. Problem je kako da sve ostane dobro organizovano i kako da se do kraja razume originalni kod.

Ako biste imali nekoliko desetina klasa i ako je vašem programu potrebno još nekoliko desetina klasa, na kraju biste prethodno opisanom tehnikom kopiranja stigli do više desetina datoteka sa izvornim kodom. Bez pažljivog planiranja završili biste sa neorganizovanom gomilom koda, prepounom bagova.

Ovaj problem možemo posmatrati i sa druge strane. Pretpostavimo da imamo komplikovanu klasu koja u osnovi radi ono što nam je potrebno, ali je potrebno da se nešto malo doda. Ako promenimo izvorni kod, rizikujemo da nešto poremetimo. Zbog toga moramo da pažljivo studiramo izvorni kod da bismo bili sigurni da su promene na mestu. To nije nimalo lako.

Klase predstavljaju dobru tehniku za modularnu dekompoziciju i poseduju mnoge kvalitete koji se očekuju od komponenata za višekratnu upotrebu. One su homogeni i koherentni moduli, jasno se može razdvojiti interfejs od implementacije itd. Ipak da bi se postigao krajnji cilj višekratne upotrebe i proširivosti softvera potrebno je nešto više.

Da bi se izbeglo stalno prepisivanje istog koda iz početka, gubljenje vremena, nedoslednost i rizik od pojave grešaka, potrebna je tehnika za obuhvatanje očiglednih zajedničkih karakteristika, koje postoje unutar grupa slične strukture, na primer, svih editora teksta, svih tabela itd. Istovremeno treba voditi računa i o razlikama koje su karakteristične za pojedine slučajeve.

Da bi se dostigli višekratna upotreba i proširivost potrebno je da iskoristimo konceptualne odnose između klasa. Klasa može biti proširenje, specijalizacija ili kombinacija drugih klasa. Za definisanje i upotrebu tih odnosa potrebna je podrška jezika. Tu podršku obezbeđuje nasleđivanje.

Nasleđivanje je proces defnisanja nove klase koja nasleđuje neku već postojeću klasu. Nova klasa, koja se naziva potomak, ili podklasa, ili subklasa, sadrži sve osobine roditelja, ali ima mogućnost dodavanja novih osobina.

Nasleđivanje sa stanovišta modula i sa stanovišta tipa

Već smo pomenuli da se klasa može posmatrati i kao modul i kao tip. Kod nasleđivanja se ova dvojaka uloga posebno ispoljava. Sa stanovišta modula, naslednik predstavlja proširenje roditeljskog modula, dok sa stanovišta tipa on opisuje podtip roditeljskog tipa.

Sa tačke gledišta modula nasleđivanje je posebno korisno kao tehnika za višekratnu upotrebu koda.

Modul je skup usluga koje se nude spoljašnjem svetu. Bez nasleđivanja svaki novi modul bi morao da samostalno definiše sve usluge koje nudi. Ne postoji, međutim, način da se novi modul jednostavno definiše dodavanjem novih usluga onima koje već postoje u prethodno definisanim modulima.

Nasleđivanje daje tu mogućnost. Ako B nasleđuje A, onda sve usluge (metodi i članovi) klase A automatski postaju dostupni i u klasi B, bez ikakve potrebe da se u B dodatno definišu. Sa druge strane B slobodno može dodavati nove karakteristike za svoje potrebe. Dodatni nivo prilagodljivosti se postiže ponovnim definisanjem, odnosno mogućnošću da B izabere implementacije koje nudi A, pri čemu neke od njih ostavlja onakvim kakve jesu, dok druge menja i prilagođava svojim potrebama.

Ovo vodi ka stilu programiranja koji umesto da se svaki novi problem rešava iz početka, ohrabruje građenje programa na prethodnim dostignućima i njihovim prethodnim rezultatima.

Dobar modul treba da istovremeno bude i otvoren i zatvoren. On treba da bude zatvoren pošto su klijentima potrebni servisi određenog modula za sopstveni razvoj. Kada se odluče za neku verziju određenog modula, uvođenje novog servisa ne bi trebalo da utiče na klijente. Sa druge strane modul treba da bude i otvoren, pošto ne postoji garancija da će od početka obuhvatiti sve uslugekoje će klijentu potencijalno trebati.

Ovakav dvojni zahtev koji se postavlja pred module, predstavlja dilemu koju klasično programiranje ne rešava. Ona se može rešiti nasleđivanjem. Klasa je zatvorena pošto može da se kompajlira, sačuva u biblioteci klasa, odakle je klijentske klase mogu koristiti. Sa druge strana klasa je i otvorena, pošto je bilo koja nova klasa može koristiti kao svog roditelja i dodati nove karakteristike ili menjati postojeće. U tom procesu nema potrebe da se menja originalna klasa ili da se smeta njenim klijentima.

Jedan od najtežih problema prilikom planiranja višekratno upotrebljivih struktura modula je neophodnost da se iskoristi prednost zajedničkih osobina koje postoje između grupa povezanih apstrakcija podataka. To se upravo postiže nasleđivanjem.

Ideja kod nasleđivanja je da se definicija svake karakteristike podigne što više u hijerarhiji, tako da je može deliti što veći broj klasa potomaka.

Ako se govori o nasleđivanju sa stanovišta tipa onda nasleđivanje predstavlja odnos tipa je (engleski "is a"). Primer za ovo može biti iskaz "Svaki pas je životinja" i sl.

Nasleđivanje se poneka posmatra kao proširenje, a ponekad kao specijalizacija. I jedno i drugo tumačenje je ispravno, iako možda izgleda da su to kontradiktorne tvrdnje. Razlika dolazi od toga da li klasu posmatrate kao tip ili kao modul.

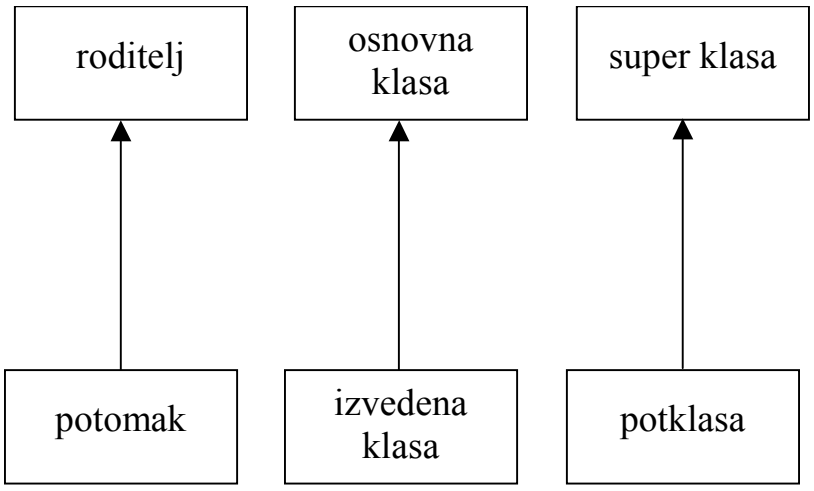
Ako je posmatrate kao tip, onda je očigledno u pitanju specijalizacija. Pas je mnogo uži pojam nego životinja.

Sa stanovišta modula se klasa posmatra kao davalac usluga, odnosno B (naslednik klase A) implementira sve karakteristike klase pretka A, ali i svoje karakteristike. Sa tog stanovišta, to je proširenje.

Nasleđivanje u Javi

Ako nasleđivanje u programiranju posmatrate kao analogiju sa ljudskom vrstom, Java tu nije dosledna. Dete bi prirodno nasledilo osobine oba roditelja. U Javi je dozvoljeno samo jednostruko nasleđivanje. To znači da izvedena klasa može da ima samo jednog roditelja. Neki objektno orijentisani jezici podržavaju i višestruko nasleđivanje. O tome će kasnije biti više reči. Kreatori Jave su zaključili da višestruko nasleđivanje donosi više štete nego koristi, pa ga zato nisu ugradili u jezik.

Šematski se odnos klase roditelja i klase potomka može predstaviti na sledeći način. Na slici su dati i termini koji se paralelno koriste, ali u osnovi označavaju istu stvar.



Izvođenje klasa se šematski obično prikazuje strelicom usmerenom od potomka prema roditelju. Važno je primetiti da se nasleđivanje odnosi na klase, a ne na objekte.

Sintaksa nasleđivanja u Javi je:

```
class IzvedenaKlasa extends OsnovnaKlasa{
```

```
   // ovde idu karakteristike izvedene klase
}
```

Klasa može biti roditelj neke klase, ali istovremeno može biti potomak neke druge klase. Isto kao i kod ljudi. Svako od nas je nečije dete, a mnogi od nas imaju svoju decu.

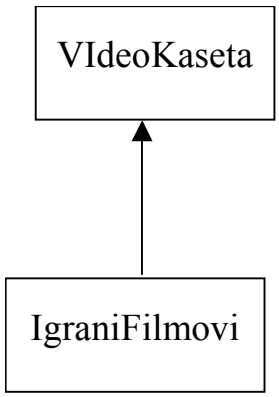
U sledećem primeru su prikazane dve klase koje se koriste u video klubu. U ovom primeru ne postoji nasleđivanje.

```
class VideoKaseta{
  String  naslov;    // naslov filma
  int     duzina;   // broj minuta
  boolean raspoloziva;    // da li je kaseta na raspolaganju?

  // konstruktor
  public VideoKaseta ( String ttl )  {
    naslov = ttl; duzina = 90; raspoloziva = true;
  }

  // konstruktor
  public VideoKaseta ( String ttl, int lngth )  {
    naslov = ttl; duzina = lngth; raspoloziva = true;
  }

  public void prikazi()  {
    System.out.println( naslov + ", " + duzina + " min. raspoloziva:" +
raspoloziva );
  }

}

class VideoKlub{
  public static void main ( String args[] )  {
    VideoKaseta k1 = new VideoKaseta ("Matrix", 120 );
    VideoKaseta k2 = new VideoKaseta ("Ratovi zvezda" );

    k1.prikazi();
    k2. prikazi ();
  }
}
```

Klasa VideoKaseta je u redu za eventualno dokumentarne filmove, jer sadrži osnovne informacije. Šta ako bismo želeli da o igranim filmovima imamo više informacija, na primer rezisera i zanr. U tom slučaju bismo mogli da napravimo novu klasu IgraniFilmovi, koja bi nasledila klasu VideoKaseta i dodala atribute koji su potrebni.

```
class IgraniFilmovi extends VideoKaseta{
  String  reziser;     // ime rezisera
  String  zanr;       // K, SF, A

  // konstruktor
  public IgraniFilmovi(String ttl, int lngth, String dir, String rtng){
    super( ttl, lngth );    // konstrktor super klase
    reziser = dir;  zanr = rtng;  // inicijalizuje se ono što je novo
                                  // za klasu IgraniFilmovi
  }
}
```

Klasa IgraniFilmovi je potklasa klase VideoKaseta. Šematski se to prikazuje ovako:



Objekat tipa IgraniFilmovi ima sledeće atribute:

član	
naslov	nasleđen iz klase VIdeoKaseta
duzina	nasleđen iz klase VIdeoKaseta
raspoloziva	nasleđen iz klase VIdeoKaseta
prikazi()	nasleđen iz klase VIdeoKaseta
reziser	definisan u klasi IgraniFilmovi
zanr	definisan u klasi IgraniFilmovi

Konstruktor klase roditelja

Klasa VideoKaseta ima konstruktor koji inicijalizuje atribute objekata tog tipa. U klasi IgraniFilmovi postoji konstruktor koji inicijalizuje podatke za objekte tog tipa. Konstruktor klase IgraniFilmovi je:

```
public IgraniFilmovi(String ttl, int lngth, String dir, String rtng){
    super( ttl, lngth );    // konstrktor super klase
    reziser = dir;  zanr = rtng;  // inicijalizuje se ono što je novo
                                  // za klasu IgraniFilmovi
  }
```

Iskaz super(ttl, lntgh) poziva konstruktor klase roditelja, koji treba da inicijalizuje neke podatke. Dobra programerska praksa je da na početku konstruktora izvedene klase pozivate konstruktor klase roditelja (na ovaj način), ali to nije obavezno. Ako se poziva konstruktor osnovne klase, to mora biti prvi iskaz u konstruktoru izvedene klase.

"Nije obavezno" iz prethodne priče je samo uslovan iskaz i odnosi se na to da programer nije obavezan da lično napiše odgovarajući kod za poziv konstruktora osnovne klase. Sa druge strane, ako to ne uradi programer, uradiće Java kompajler. Pogledajmo sledeći primer:

```
// predloženi konstruktor
public IgraniFilmovi( String ttl, int lngth, String dir, String rtng ){
  naslov = ttl;       // ovde se radi ono što radi konstruktor
                      // klase roditelja.
  duzina = lngth;
  raspoloziva = true;
  reziser = dir;
  zanr = rtng;
}
```

Iako izgleda da ovde nije potrebno pozivati konstruktor osnovne klase, on se ipak implicitno poziva. Kompajler smatra da prethodni kod izgleda ovako:

```
public IgraniFilmovi( String ttl, int lngth, String dir, String rtng ){
  super(); // poziva se podrazumevani konstuktor klase roditelja
  naslov = ttl;
  duzina = lngth;
  raspoloziva = true;
  reziser = dir;
  zanr = rtng;
}
```

Kompajler implicitno poziva podrazumevani konstruktor osnovne klase. Ako ta klasa nema podrazumevani konstruktor može , ali ne mora, doći do sintaktičke greške.

Kada klasa ima podrazumevani konstruktor?

- Ako je programer napisao konstruktor bez argumenata?
- Ako programer nije napisao nijedan konstruktor za klasu, onda će kompajler sam napraviti podrazumevani konstruktor.
- Ako je programer napisao makar jedan konstruktor za klasu, onda se podrazumevani konstruktor ne pravi automatski.

U primeru koji smo dali, definicija klase VideoKaseta sadrži konstruktor, tako da se podrazumevani konstruktor neće napraviti. Prema tome predloženi konstruktor iz prethodnog primera bi javio grešku.

Da vidimo sada kako bismo koristili ove dve klase:

```
class VideoKlub{
  public static void main ( String args[] )  {
    VideoKaseta k1 = new VideoKaseta ("Mikrokosmos", 90 );
    IgraniFilmovi k2 = new IgraniFilmovi ("Ratovi zvezda", 120,
"Lukas", "SF" );
    k1.prikazi();
    k2. prikazi ();
  }
}
```

Nakon izvršenja ovog programa trebalo bi da dobijete sledeći izlaz:

```
Microcosmos, 90 min. raspoloziva:true
Ratovi zvezda, 120 min. raspoloziva:true
```

Iskaz k2.prikazi() poziva metod prikazi() objekta k2. Ovaj metod je nasleđen iz klase VideoKaseta i nije menjan. Ovaj metod ne zna da su u klasi IgraniFilmovi dodati novi atributi, tako da ih i nije odštampao.

Redefinicija (nadjačavanje) metoda osnovne klase

Jedna od ideja koja leži iza koncepta nasleđivanja jeste mogućnost da izvedena klasa izabere da li će osobine roditelja naslediti bez ikakvih promena, ili će im promeniti definiciju u skladu sa svojim potrebama. Ovde se pod osobinama prvenstveno misli na metode.

Za metod izvedene klase se kaže da nadjačava (engl. override) metod osnovne klase, ako oba metoda imaju istu signaturu (ime metoda i lista argumenata). U tom slučaju roditeljska klasa ima svoju verziju metoda, a izvedena klasa svoju.

Kada se napravi objekat tipa osnovne klase on koristi metod iz osnovne klase. Kada se napravi objekat izvedene klase, on koristi svoju verziju metoda.

U prethodnom primeru mogli bismo da u klasi IgraniFilmovi nadjačamo metod prikazi() iz osnovne klase VideoKaseta.

```
  public void prikazi()  {
    System.out.println( naslov + ", " + duzina + " min. raspoloziva:" +
raspoloziva );
    System.out.println( režiser + ", " + reziser + ", žanr:" + zanr);
  }
```

Ako sada ponovimo program koji smo napisali:

```
class VideoKlub{
  public static void main ( String args[] )  {
    VideoKaseta k1 = new VideoKaseta ("Mikrokosmos", 90 );
    IgraniFilmovi k2 = new IgraniFilmovi ("Ratovi zvezda", 120,
"Lukas", "SF" );
    k1.prikazi();
    k2. prikazi ();
  }
}
```

on će dati sledeći izlaz:

```
Microcosmos, 90 min. raspoloziva:true
Ratovi zvezda, 120 min. raspoloziva:true
režiser: Lukas, žanr: SF
```

U redu k1.prikazi() se poziva metod prikazi iz klase VideoKaseta, a u redu k2.prikazi() se poziva metod prikazi iz klase IgraniFilmovi.

Ako pogledate kod metoda prikazi u klasi IgraniFilmovi videćete da se tu ponavlja deo koda koji već postoji u metodu priakzi klase VideoKaseta. U ovakvim slučajevima, kada u metodu izvedene klase želimo da iskoristimo metod osnovne klase, to možemo uraditi na sledeći način:

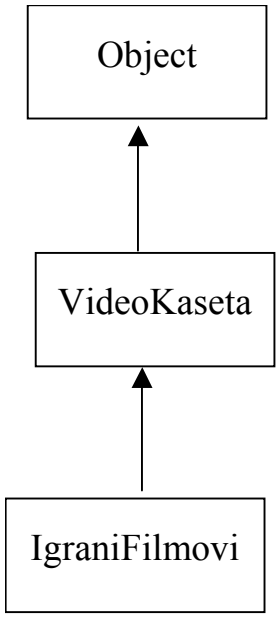
```
public void prikazi()  {
  super.prikazi();
  System.out.println( režiser + ", " + reziser + ", žanr:" + zanr);
}
```

Kao što vidite preko ključne reči super smo pozvali metod osnovne klase. Za razliku od slučajeva kada se super koristi u konstruktorima, ovde iskaz super ne mora biti prvi u metodu.

Klasa Object

Nasleđivanje je u Javi tako organizovano da sve klase uvek nasleđuju jednu klasu, klasu Object. U pitanju je klasa koja je unapred definisana.

Čak i kada radite sa običnim klasama, kod kojih niste eksplicitno uveli nasleđivanje, ono implicitno postoji. U Javi se podrazumeva da su sve klase izvedene iz klase Object. U skladu sa ovim hijerarhija klasa za naš primer bi izgledala ovako:



U klasi Object je definisan određen broj metoda koje nasleđuju sve druge klase u Javi. To su na primer, metode toString(), equals() i sl. Ove metode se kasnije mogu nadjačati, ali se ponekad mogu koristiti i u izvornom obliku.

Višestruko nasleđivanje

Višestruko nasleđivanje je u suštini samo varijacija već pomenutog koncepta nasleđivanja. Razlika je u tome što se osobine nasleđuju od više osnovnih klasa. Različiti autori se različito odnose prema višestrukom nasleđivanju. Neki od njih smatraju da ono donosi više problema nego koristi, dok drugi ističu njegov značaj i smatraju da se pravilnom upotrebom mogu iskoristiti samo prednosti, a da se ne zapadne u probleme. Autori Jave spadaju u prvu grupu, tako da višestruko nasleđivanje nisu ni ubacili u jezik.

Primer višestrukog nasleđivanja

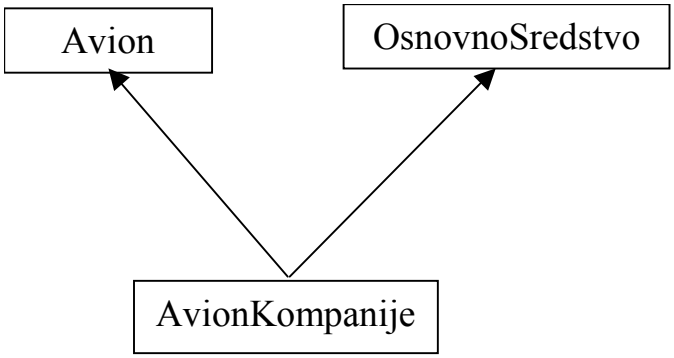
Višestruko nasleđivanje se najbolje može shvatiti na primeru.

Pretpostavimo da postoji klasa Avion. Kao što joj i ime kaže ona opisuje apstrakciju aviona. Njeni atributi mogu biti brojPutnika, visina, brzina itd. Metode mogu biti poleti, podesiBrzinu itd.

U drugom domenu može postojati klasa OsnovnaSredstva, koja opisuje knjigovodstveni pojam osnovnog sredstva. Atributi osnovnih sredstava mogu biti nabavnaCena, prodajnaVrednost itd.

Pogledajmo sada jednu avionsku kompaniju. Kompanije poseduju avione. Za pilota je taj avion samo običan avion sa svim svojstvima koje imaju obični avioni. On poleće, sleće, ima određenu brzinu itd. Sa tačke gledišta knjigovođe to je osnovno sredstvo, koje ima svoju nabavnu cenu, procenjenu vrednost itd.

Avion kompanije možemo modelirati preko višestrukog nasleđivanja:



U jeziku C++, koji podržava višestruko nasleđivanje, to bi izgledalo ovako:

```
class AvionKompanije : public Avion, public OsnovnoSredstvo {
….
}
```

Klasa AvionKompanije nasleđuje osobine klase avion, ali i osobine klase OsnovnoSredstvo.

Višestruko nasleđivanje i dvosmislenost kod pristupa članovima klase

Ovaj problem se javlja ako klase roditelji imaju isto ime za različite karakteristike. U tom slučaju će se javiti dvosmislenost kod pristupa članovima koji potiču iz tih klasa.

Postoje dva osnovna tipa dvosmislenosti. Jedan je slučajna dvosmislenost, kada postoje dva ista imena, koja se ne odnose na isti atribut. U tom slučaju je reč o slučajnosti, koja se desila kod izbora imena za atribute tih klasa.

```
class A{
   public: int i;
}
class B{
   public: int i;
}
class C : public A, public B {
   public: void f();
```

```
}
```

Druga varijanta dvosmislenosti nastaje ako postoji ponovljeno nasleđivanje. U tom slučaju je reč o istom imenu, ali i o istom entitetu.

```
class X{
   public: int i;
}
class A : public X{};
class B : public X {};
class C : public A, public B {
   public: void f();
}
```

Kako bi se ovo rešilo. Ponovo nekoliko mogućih pristupa. Jedan je princip zabijanja glave u pesak. U ovoj varijanti se zabranjuje takvo nasleđivanje. Problem se vraća dizajneru osnovnih klasa i on mora da reši problem, odnosno promeni ime. Ovo nije ni razumno ni uvek moguće. Nije razumno, zato što su dobra smislena imena retka, a višestruko nasleđivanje u puno slučajeva spaja hijerarhije koje dolaze od različitih proizvođača.

Druga varijanta je da se sve prebaci na onog ko pravi novu klasu. U tom slučaju programer mora da proba da sam reši sve probleme koji nastaju usled dvosmislenosti.

Treća mogućnost je da se ostavi klijentu da sam rešava probleme.

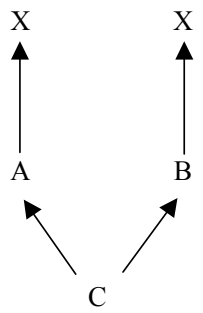
U jeziku C++ se ovakvi problemi uglavnom rešavaju primenom operatora za razrešavanje oblasti važenja. Pogledajmo ponovo primer sa ponovljenim nasleđivanjem:

```
class X{
   public: int i;
}
class A : public X{};
class B : public X {};
class C : public A, public B {
   public: void f();
}
```

Objekti klase C u ovom primeru će imati dva člana sa imenom i. U svakom objektu klase C postoje dva podobjekta tipa X. Jedan je nasleđen od A, a drugi od B. Pristup članu i bez eksplicitnog navođenja imena klase kojoj pripada, nije dozvoljen. Razlog je dvosmislenost. Članu i se može pristupiti samo eksplicitnim navođenjem imena klase kojoj pripada i operatora razrešavanja oblasti važenja.

```
void C::f(){
   i = 0; // greska, dvosmislenost da li je i iz A ili B
   A::i = B::i // ovako moze
}
```

Relacija između prikazanih klasa se grafički može prikazati kao na sledećoj slici. Treba primetiti da su podobjekti tipa X, unutar objekta tipa C, potpuno nezavisni, odvojeni objekti.



Posledica ovog je da nije dozvoljeno da jedna klasa bude višestruka direktna osnovna klasa, inače se ne bi mogla izbeći dvosmislenost.

```
class C : public B, public B ...// greška
```

Realizacija višestrukog nasleđivanja u jeziku C++

Pogledajmo sledeći primer:

```
class A {/* …. */};
class B {/* …. */};
class C : public A, public B {/* … */}
```

Objekti klase C mogu se predstaviti kao strukture sledećeg izgleda:

Deo A
Deo B
Deo C

Redosled smeštanja podobjekata osnovnih klasa nije definisan jezikom, već zavisi od implementacije prevodioca.