6. Uvod u objektno-orijentisano modeliranje / Softverski inženjering

Kao što je u poglavlju 3 objašnjeno, objektno-orijentisani model razmišljanja je idealna osnova za softversko rešavanje inženjerskih problema. Konkretno rešenje inženjerskih praktičnih problema zahteva znanja iz objektno-orijentisanog pristupa navedena u poglavlju 3, koja sada treba primeniti.

Osnova svakog objektno-orijentisanog rešenja je takozvano **objektno-orijentisano modeliranje (OOM),** kojim se definiše principijelni postupak rešavanja, ali takođe i opšte-primenljivi mehanizam, ili, bolje rečeno, pogodan matematički opis problema, koji omogućava formalnu, ali takođe jasnu i preciznu formulaciju problema. Posledično se može izvršiti prevođenje u softver pomoću razumljivog i objektno-orijentisanog programskog jezika (u našem slučaju JAVA).

Pre nego što se pređe na formalizme i notaciju OOM, trebalo bi - radi pravilne klasifikacije objektno-orijentisanih metoda - kratko obraditi istorijski razvoj.

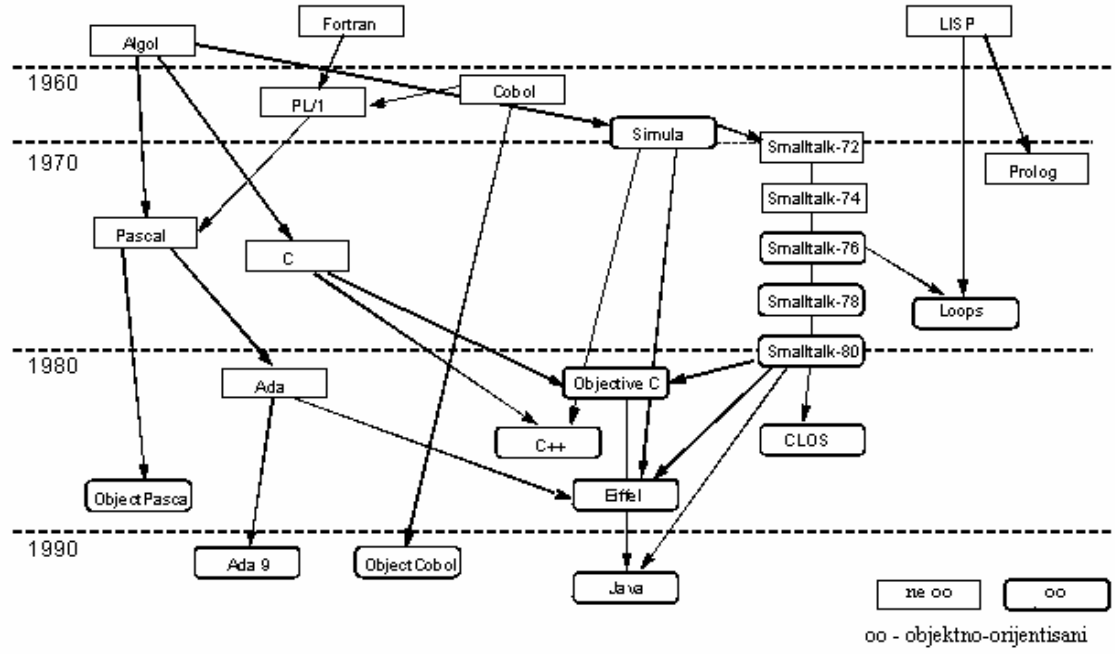
6.1. Istorijski razvoj objektno-orijentisanih metoda

Razvoj i projektovanje softvera brzo je od informatičara nazvan „inženjerstvom" i mogao se izrazom „programiranje" samo nepotpuno definisati: da bi se došlo do toga da je softver pravi inženjerski proizvod uporediv sa drugim inženjerskim proizvodima – na primer, mostom, mašinom, uređajem, i sl. – i da bi se učinilo razumljivim da je proces stvaranja softvera sličan drugim inženjerskim procesima – na primer procesu konstruisanja, uveden je izraz „softversko inženjerstvo". Ovaj izraz bi trebalo da objasni da softver mora da bude konstruisan po strogo postavljenim principima i kontrolisan jasno definisanom sistematikom, uz omogućavanje upravljanja njime.

Objektna orijentacija je najnovije dostignuće u razvoju softverskog inženjerstva: mnogi poznati principi softverskog inženjerstva su pritom preuzeti (npr. princip modularnosti, princip lokalnosti i dr.) i povezani sa novim idejama (jedinstvo podataka i operacija, asocijacije, izmena novosti, itd.). Ovo je imalo za posledicu da se se pored numeričkih problema – više nego do tada – nenumerički (kognitivni, tj. podacima orijentisani) problemi mogli rešavati; čak su i humanoidni aspekti (npr. interaktivnost, bolja uslužnost, individualnost) mogli biti bolje realizovani nego kod konvencionalnog softverskog inženjerstva.

Ideja objektne orijentacije je pritom stara više od 30 godina: upravo šezdesetih godina prošlog veka stvoren je prvi objektno-orijentisani programski jezik, SIMULA. No tek sa programskim jezikom SMALLTALK, početkom osamdesetih, šire je postao poznat objektno-orijentisani model mišljenja. Posle toga su nastali mnogi objektno-orijentisani programski jezici

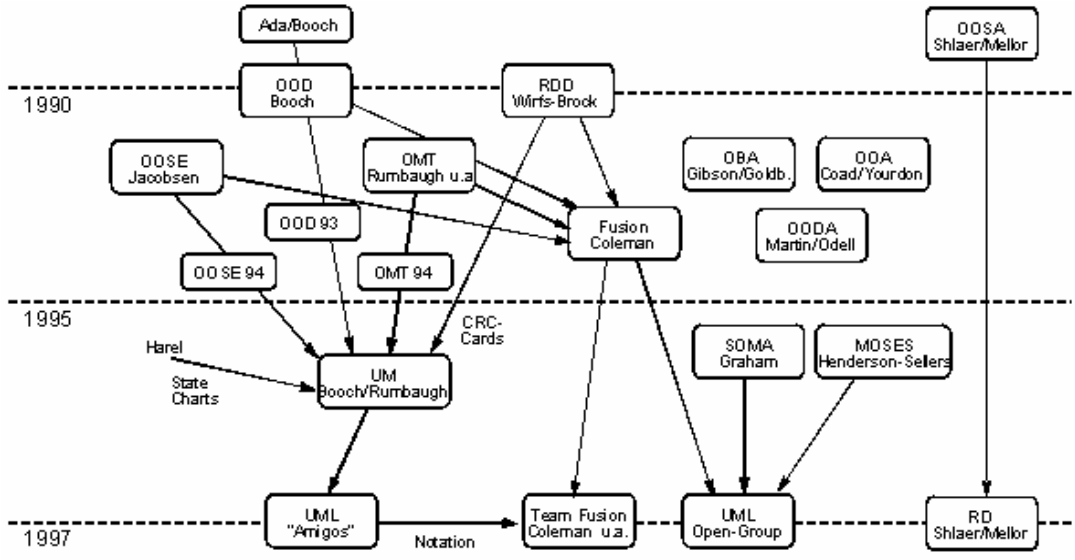
(Objective C, C++, Eiffel, Clos, Object Pascal, Object Cobol, Ada 9 i JAVA). Sledeći grafikon prikazuje razvojni tok objektno-orijentisanih jezika u kontekstu sa ostalim programskim jezicima.



Istorijski razvoj programskih jezika

Iako se objektno-orijentisani programski jezici već dugo vreme upotrebljavaju, iznenađujuće je da je objektno-orijentisano modeliranje, koje je u suštini koncepcionalna osnova objektno-orijentisanog programiranja, nastalo mnogo docnije. Prvi udžbenici o OOM pojavili su se tek početkom devedesetih. S time u vezi se mogu posebno navesti knjige autora Von Boocha, Coad/Yourdon, Rumbaugh i dr., Wirfs-Brock/Johnson, Shlaer/Mellor i Jacobsen.

Zatim se pojavio čitav niz različitih, međusobno konkurentnih OOM-metoda. Sledeći grafikon daje razvoj pojedinih metoda sa naznakom autora pojedinih metoda i pripadajućom skraćenicom metode.



Razvoj OOM (objektno-orijentisane metode)

Posebno su dominantne dve metode:

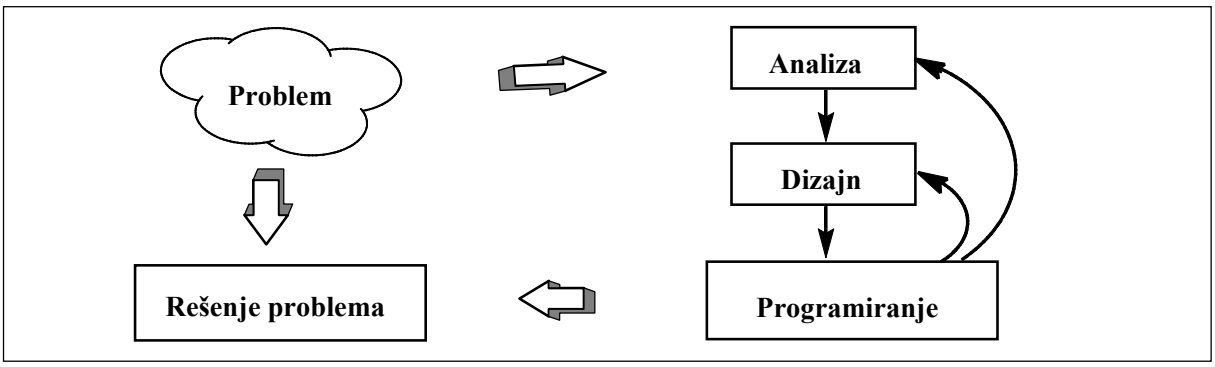
- Object-Modeling Technique (OMT – Tehnika za objektno modeliranje), autora Rumbaugha, i

- Object Oriented Design (OOD – metoda Booch-a).

OMT se jako oslanja na klasične struktuirane metode, dok je OOD više orijentisana ka komercijalnim i vremensko-kritičnim primenama.

Godine 1995. su saradnjom Rumbaugha i Boocha OMT i OOD metoda prevedene u jedan zajednički oblik, takozvani **Unified Method (UM)**, ujedinjeni metod. Kada je i Jacobsen, takođe pristalica OOM pristupio grupi Booch/Rumbaugh (Trijumvirat se nazvao „tri Amigosa") i uveo sopstvene principe, posebno „Use Cases" (tj. slučajeve primene), nastao je **Unified Modelling Language (UML)** (Jedinstveni jezik za modeliranje), i kao Verzija 1.0 1997. godine stigao kod **Object Management Group** (konzorcijum za popularizaciju Objektne tehnologije) za standardizaciju i odmah počeo da se razvija kao priznati standard za modeliranje.

UML je zadovoljio sve uslove jednog konzistentnog opisnog matematičkog alata i podržavao obe oblasti virtuelnog „programiranja", tehniku analize i dizajna – projektovanja.



Bazirajući se na definisanoj sistematici i formalnoj notaciji, može se pomoću UML-a od pojedinog problema razviti odgovarajući meta-model (tj. jedan slikoviti, ali od nepotrebnih detalja abstrahovan model problema), koji se relativno lako, pošto je konsekventno objektno-orijentisano postavljen, bez značajnijih gubitaka informacija, pomoću objektno-orijentisanog jezika prevesti u izvršni program. Striktna konzistentnost objektne orijentacije, od analize preko dizajna do programiranja, omogućuje čak i automatsko generisanje programskog koda. Tipičan primer za to je razvojni alat „Together J", pomoću koga se metamodel razvijen pomoću UML notacije može automatski prevesti u jezik JAVA i obrnuto.

6.2. Modeliranje pomoću UML i prevođenje u programu JAVA

Na osnovu činjenice da se UML u međuvremenu razvio u svetski priznati standard, u Inženjerskoj informatici je izabran za standardni jezik za opis problema, a u oblasti programiranja mu je data prednost. Drugim rečima, daje se prednost objektivno-orijentisanom modeliranju

problema, kojim se omogućava misaoni prodor u rešenje problema. Tek pošto je taj prodor uspešan, okrećemo se programiranju problemskog rešenja, na osnovu metamodela notiranog u UML jeziku. Da bi se ovaj školski model mogao slediti, moraju se izložiti najvažniji elementi i najvažniji formalizmi UML jezika.

6.2.1. Klase i objekti – Posebni oblici klasa

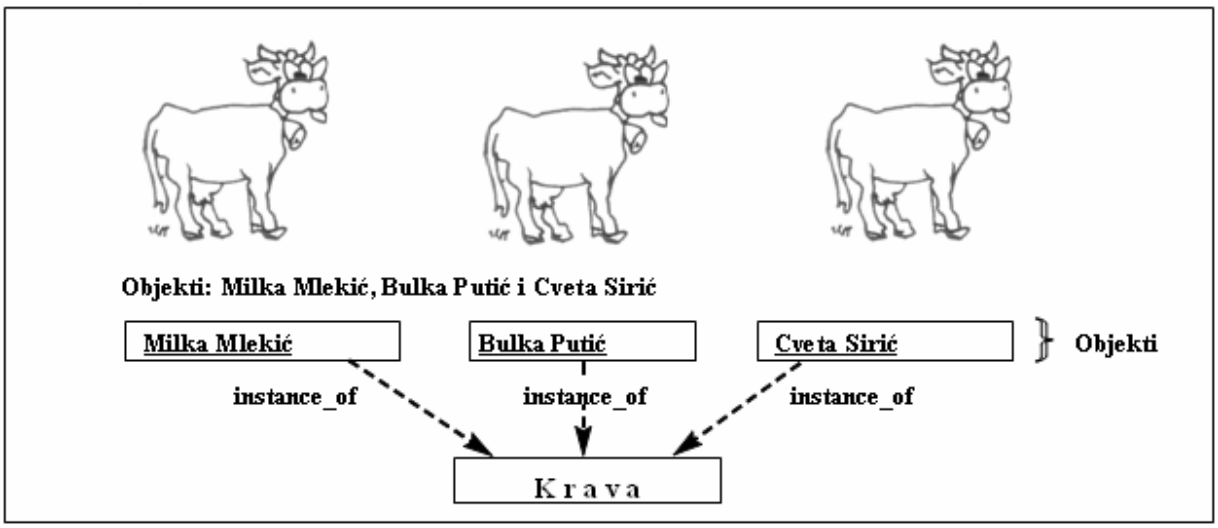
Objekti i klase su već kratko informaciono obrađeni (videti poglavlje 3), ali su za konkretno rešavanje problema na osnovu UML neophodni još neki detalji i pojedinosti.

Izlazni rezultat UML jezika su objekti, koji – kao što znamo – predstavljaju razumljive abstrakcije stvari iz odgovarajućeg okruženja (iz domena problema) i dobijaju se eliminisanjem nepotrebnih pojedinosti (detalja). Pored toga znamo da objekti imaju osobine koje sadrže s jedne strane atribute, s druge strane operacije, tj. metode (kao i funkcionalnost) i da su veze (asocijacije) između objekata vrlo važne.

Za rad sa modelima objekata su i klase, koje čine objekti iste semantike i stoje u vezi sa drugim klasama. Uvođenjem klasa, koje predstavljaju plan gradnje, pomoću koga se mogu automatski izvesti tj. proizvesti konkretni, nezamenljivi primeri klasa, postaje nepotrebno pojedinačno koncipiranje svakog objekta (kao kod klasičnog modela razmišljanja). Objekti pritom fizički zauzimaju mesto u memoriji odgovarajuće planu gradnje (strukturi) klase, radi docnije realizacije na računaru.

Ukoliko su gore obuhvaćeni osnovni koncepti relativno brzo objašnjeni, može se njihov način funkcionisanja razjasniti na konkretnim primerima; ovde će biti razmotrena dva mala praktična problema.

Primer (naučno-popularni):



Činjenično stanje

Jedna krava, kao Objekt, daje mleko; mnogo krava, kao Objekti iste semantike daju isto tako sve mleko, posedujući pritom iste osobine i operacije („davati mleko"), što sugeriše da se u OOM (Object Oriented Modeling) može modelirati jedna „Klasa Krava".

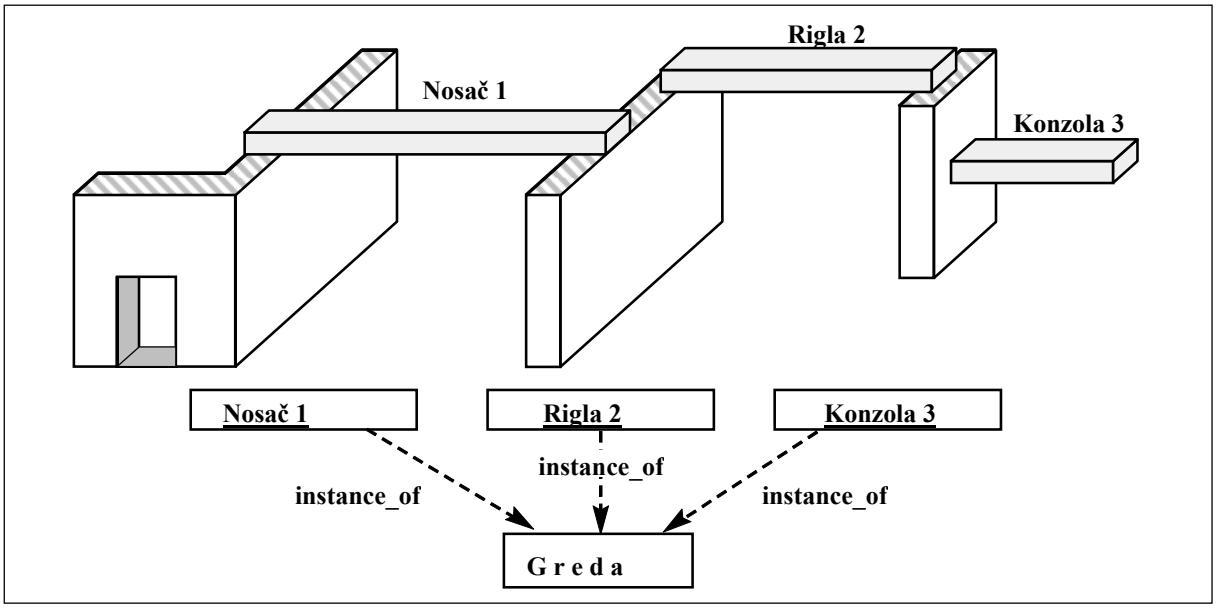
Notacija

U UML notaciji je prihvaćeno da se klase i objekti predstavljaju pravougaonicima. Pritom se pojedini pravougaonici obeležavaju imenima klase odnosno objekta i to masnim slovima (bold) sa početnim velikim slovom. Za razliku od imena klasa, imena objekata su podvučena.

Tako su sve gorenavedene krave, iako svaka za sebe „nezamenljiva", nastale po projektu „krava", tako da su one takozvane „instance" (objektne instance) klase (Objektne klase) „Krava". Izrazi „Instanca" tj. „Instanciranje" baziraju se na slobodnom prevodu engleskog izraza „instance" za objekte; ispravnije bi bilo reč „instance" prevesti sa „egzemplar" ili „primer" (primer objekta).

Odnos između instanca jedne klase i njoj dodeljene klase može se predstaviti isprekidanom strelicom $- - - >$ pri čemu svaka strelica ima značenje „instance-of" ili „primer-od".

Primer (iz projektovanja nosećih konstrukcija)



Činjenično stanje

Za objekte „Nosač", „Nadvratnik", „Konzola", modelira se klasa „greda", pošto se sva tri objekta po teoriji konstrukcija u smislu teorije deformacije mogu obuhvatiti gredama opterećenim na savijanje.

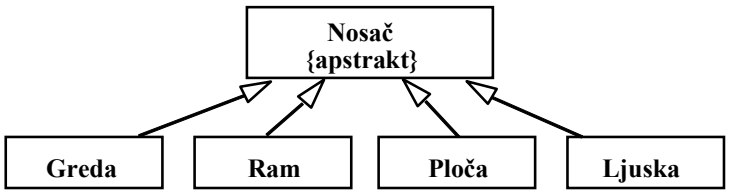
Apstraktne klase

Primer (Projektovanje nosača)

Primer za to da se klase mogu koristiti kao osnova za druge klase (podklase) i tek onda iz njih izvesti instance je klasa „Noseći elemenat". On direktno ne čini nijedan konkretni objekat, već prepušta kao viša klasa kreiranje svojim podklasama „Greda", „Ram", „Ploča", „Platno", itd. Ovakve klase, koje ne proizvode direktno objekte, nazivaju se apstraktne klase. Apstraktne klase predstavljaju često opšte izraze. Apstraktne klase su nepotpune, pošto ne sadrže nikakve atribute niti operacije.

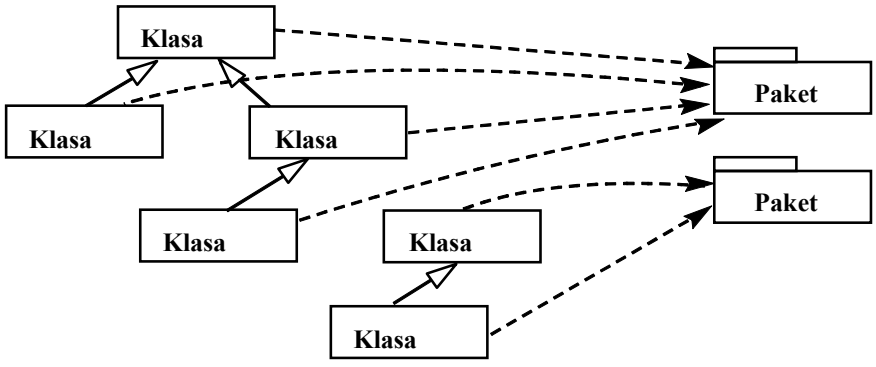
Notacija

Apstraktne klase se, kao i sve druge klase, označavaju pravougaonikom. Da bi se naznačilo da se radi o apstraktnoj klasi, ispod imena klase - uokvireno vitičastim zagradama - stavlja oznaka „Apstrakt". Za apstraktnu klasu „Noseći elemenat" biće tada sledeća notacija:



Paketi

Uvođenjem većeg broja klasa nastaje – kao što se može pretpostaviti - mreža klasa. Tako se može razmatrati jedna opšta mreža klasa (stablo klasa), bez ulaženja u detalje.

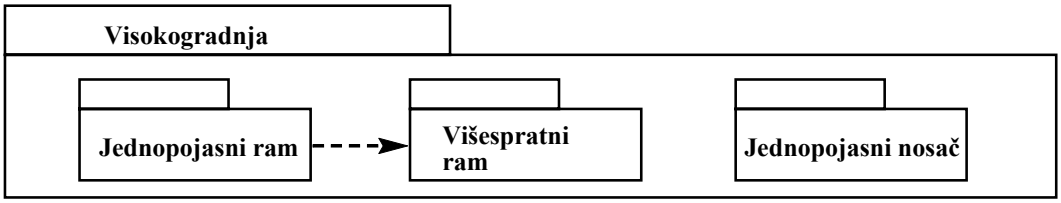


Ukoliko se iz logičkih, strukturalnih, organizacijskih ili drugih razloga klase spajaju u veće strukture, nastaju tzv. paketi ili packages (zvani takođe i podsistemi, subsystems). Paketi mogu pritom opet sadržati druge pakete. Paketi omogućavaju time struktuiranje problema i predstavljaju sredstvo za struktuiranje klasa.

Notacija

Paket se označava kao kartica kartoteke sa vođicom; ime paketa se unosi u sredinu vođice. Zavisnosti između paketa se označavaju isprekidanim strelicama.

Primer (Statika)



Paketi igraju u programiranju pomoću JAVA jezika važnu ulogu. Zato je za formiranje paketa predviđena posebna ključna reč „package". Da bi se uzela u obzir pravilna korelacija između klasa u paketu i posebno definisanih klasa, mora se paziti na pravila pri davanju imena i značenje strelica u strukturi paketa.

Paketi koji se isporučuju sa programom JAVA i koji povećavaju relativno mali vokabular JAVA programa imaju karakter biblioteka. Osam paketa (Application Program Interfaces - API) stoje na raspolaganju u JAVA (Version 1.0.2).

java.applet	za primenu apleta
java.awt	apstraktne windows alatke za postavljanje grafičkih korisničkih površina
java.awt.image	obrada slika
java.awt.peer	za preslikavanje awt na drugi sistem
java.io	ulaz/izlaz
java.lang	osnovni sastavni delovi jezika
java.net	za mrežne klase
java.util	za različite strukture podataka i pomoćne klase

6.2.2. Detalji: atributi, operacije, osiguranja, kapsuliranje podataka

Da bi se osobine klasa tj. objekata mogle preciznije utvrditi, potrebne su još neke osobine. To su:

- atributi (podaci),
- metode (operacije),
- osiguranja (uslovi).

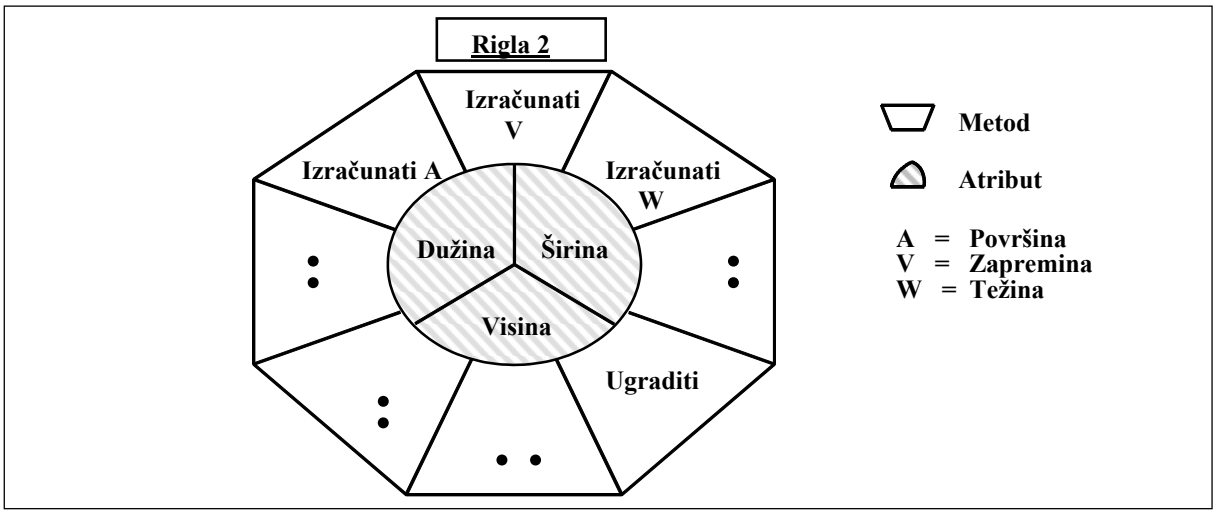
Atribute obezbeđuju podaci. Atributi mogu imati različite vrednosti i tako definisati „unutrašnje stanje" instanci za jedno određeno vreme. U gore obrađenom primeru „Greda", geometrijske veličine „Dužina", „Visina", „Širina" su važni atributi za statičko ponašanje (statično stanje) tj. gradnju (stanje gradnje). Pored ovih, mogu postojati i drugi atributi.

Metode, nazvane i operacije, govore nam o tome kako se objekti ponašaju i koje osobine poseduju (funkcionalitet). Metode imaju algoritamski karakter, pošto one po unapred datoj šemi obrađuju podatke. Vezano za primer „Greda" su proračuni površine preseka, zapremine ili krive savijanja (elasticitet) tipične metode; ali takođe i komplikovani postupci, kao što je ugradnja grede u noseću konstrukciju, mogu se formulisati kao operacije. Metode su slične funkcijama i procedurama, poznatim u klasičnom programiranju, ali se od njih prilično razlikuju, jer su u relaciji Klase-Objekti-Strukture značajno nezavisnije od klasičnih funkcija i procedura (ove stoje stalno pod kontrolom jedne jedinice koja se može uvek pozvati i kojoj se kontrola po izvršavanju vraća nazad!)

Osiguranja (prava pristupa) su uslovi, formulama slične premise ili druga pravila, koja objekti moraju na razumljiv način ispuniti, da bi se izbegla moguća greška. Na primer, kod primera „Greda", geometrijska osiguranja vrste „Dužina >> „Širina", tj. „Dužina >> Visina" bi bila razumljiva. Jedan fundamentalni princip rešavanja problema kod objektno-orijentisanog modeliranja je da se atributi objekata tako organizuju da mogu biti odgovorni samo nad specijalnim objektnim metodama. Drugim rečima: direktni pristup atributima sa mesta izvan određenih objekata se prekida. Ovaj konstrukcioni princip se naziva

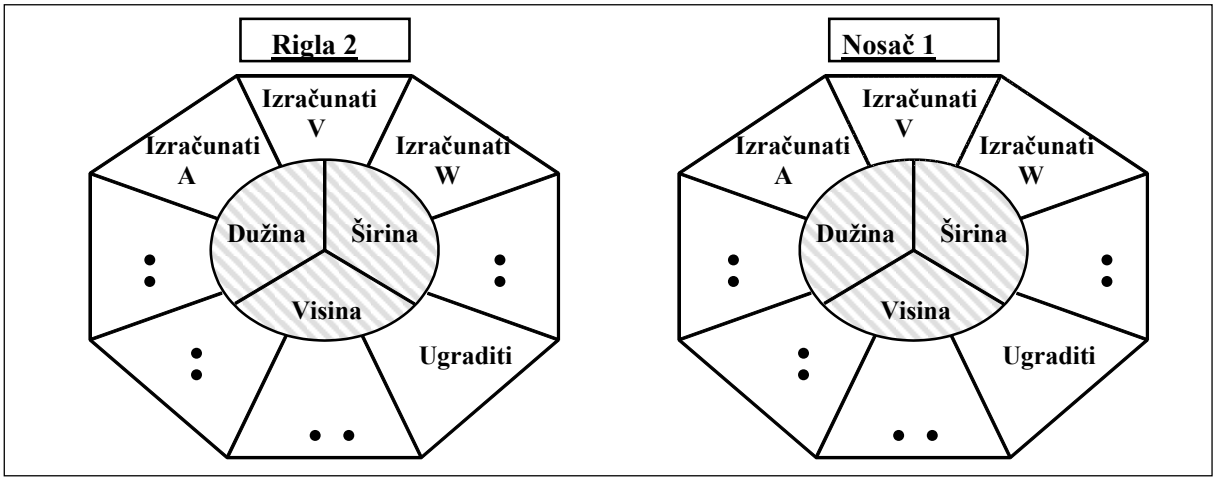
Information hiding (skrivanje informacija), Data Encapsulating (kapsuliranje podataka, Datenkapselung – nemački).

Na primeru „Greda" se može princip enkapsulacije kratko objasniti. Unutrašnja struktura objekta „Rigla 2" sa svojim atributima i metodama može se, na primer, predstaviti na sledeći način:



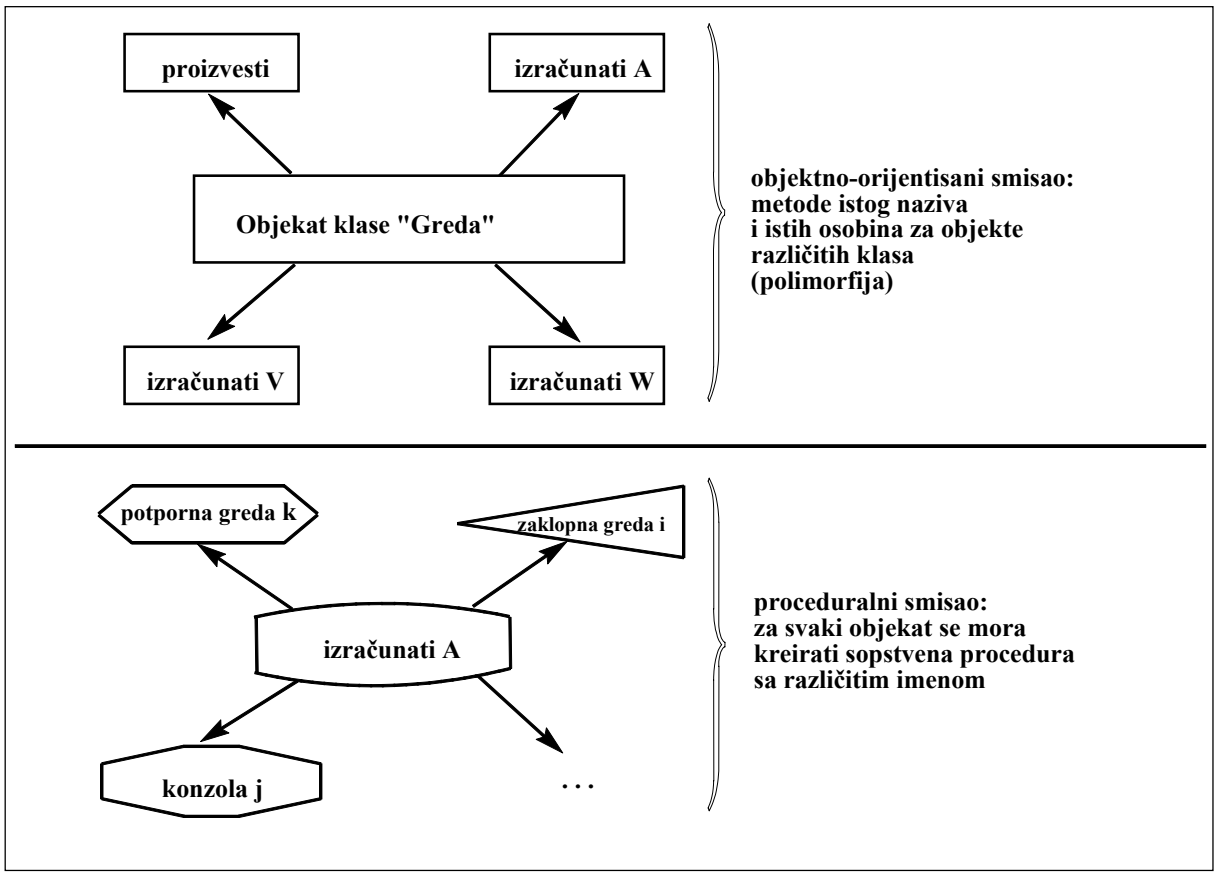
Kao što se raspoznaje, atributi „Dužina", „Širina", „Visina" su u suštini metodama koje deluju kao kvazi-ljuska zaštićeni od direktnog pristupa spolja. Konsekventnim pridržavanjem principa enkapsulacije povećava se sigurnost pri razvoju softvera, a takođe i ponovna upotrebljivost, proširljivost i pogodnost održavanja. To su osobine koje značajno utiču na cenu softvera.

Slična struktura kao za objekat „Nadvratnik 2" može se dati i za objekat „Noseća greda-1", tako da se za sve objekte klase „Greda" mogu primeniti iste metode (izracunatiA, izracunatiV, itd.) sa istim imenima metoda.



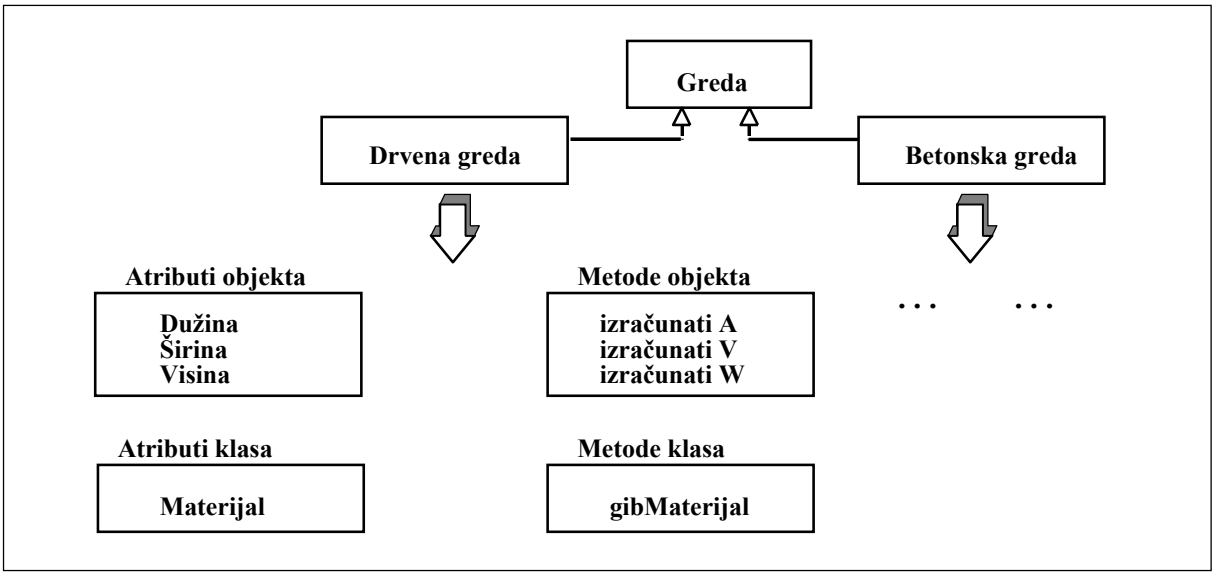
Ukoliko se neki objekat nekom porukom, jedinstvenim mehanizmom za komunikaciju između objekata, aktivira, on reaguje izvršavanjem neke njemu dodeljene metode, pri čemu se mogu opet aktivirati i drugi objekti.

Iz sledećeg primera se jasno vidi razlika u klasičnom proceduralnom (ALGOL, FORTRAN, BASIC, PASCAL, C, ...) i objektno-orijentisanom pristupu.



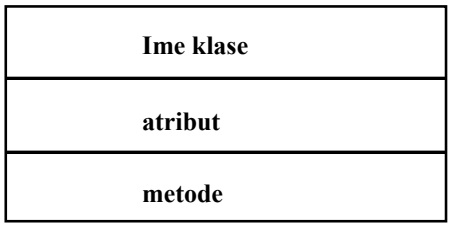
Treba napomenuti da kako kod atributa, tako kod metoda, između atributa objekata i atributa klasa, kao i između metoda objekata i metoda klasa traba činiti razliku. Atributi, koji važe za sve objekte jedne klase, dodeljuju se klasi, pri čemu svi objekti te klase imaju pristup atributima te klase. Slično važi i za metode. To znači da jedan atribut klase egzistira jednom za sve instance te klase, da se može iz svake pojedine instance promeniti i da je ta promena odmah raspoznatljiva za sve druge instance te klase.

U sledećem primeru, koji se sastoji iz nadklase „Greda" i podklasa „Drvena greda" i „Betonska greda", može se uvesti atribut „Materijal" kao promenljiva klase (što se u JAVA postiže pomoću takozvane static-definition) i definisati metoda za izdavanje materijala kao metoda klase. Atributi „Dužina", „Širina", „Visina", kao i proračunske metode su nasuprot tome definisane specijalno za nivo objekata.



Notacija

Atributi, metode i osiguranja se pri modeliranju klasa unose po strogo definisanoj notaciji. Za atribute i metode su pritom, unutar pravougaonika predviđenog za predstavljanje klase - izvan rubrike za ime klase i razdvojeno jednom horizontalnom linijom – predviđene sopstvene rubrike (videti sledeću principijelnu skicu).



Atributi se unose najmanje svojim imenom, pri čemu je utvrđeno da imena atributa počinju malim slovom. Pored imena u daljem postupku se daje tip atributa (celobrojan, realan, itd.), eventualne početne vrednosti (default values), posebne primedbe i osiguranja (pri čemu se

poslednja dva daju u vitičastim zagradama). Standardizovana notacija za atribute, kada su sve mogućnosti navedene, ima sledeći opšti oblik:

atribut: Tip_atributa = Početna_vrednost {karakteristika} {osiguranje}

Specifičnosti

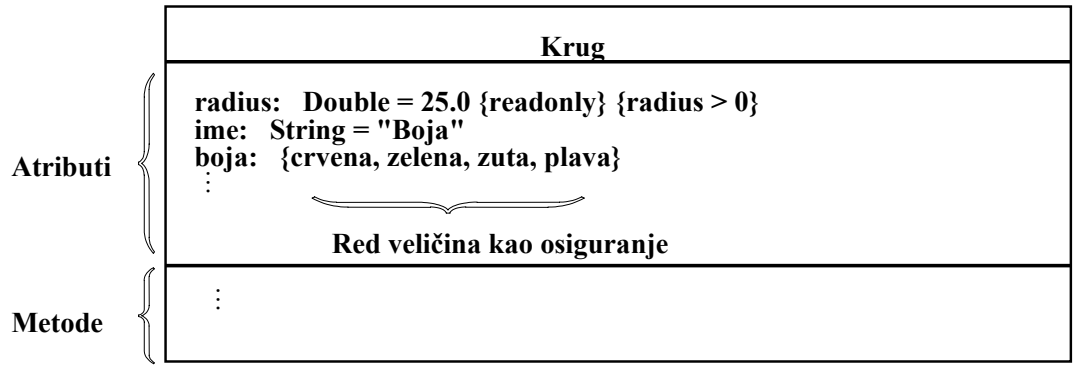
Atributi klasa se radi naglašavanja podvlače; izvedeni atributi, tj. atributi koji se dobijaju iz dotada raspoloživih atributa jedne klase prema strogo definisanim uslovima (na primer, prema nekom proračunu), sadrže „/" kao prefiks.

klasniAtribut
/ izvedeni atribut

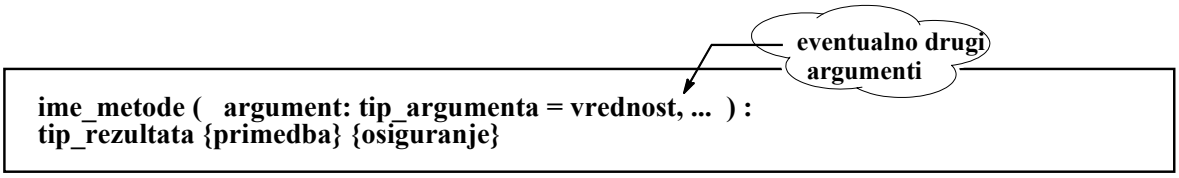
Karakteristike atributa mogu biti:

{apstrakt}
{readonly/samo učitavanje}
{transient/prolazne}
{persistent/stalne}

Jednostavni praktični primer

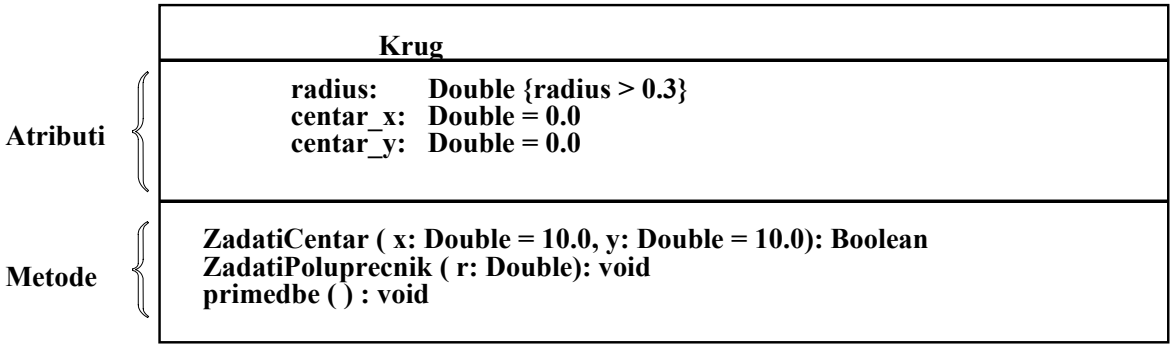


Metode se takođe daju najmanje svojim imenom, koje prema UML-konvenciji takođe počinje malim slovom. Kako će metode biti razmatrane kao „uslužni servisi", one će biti objašnjene signaturom, koja se sastoji od najviše jednog imena operacije, parametara (argumenata) i po potrebi, od tipa izlaznog rezultata. U opštem slučaju ima signatura (sintaksa) sledeći izgled:



Argumenti počinju malim slovima i opisuju se svojim tipom (u vezi sa simbolom „:") kao i, po potrebi, unapred datom početnom vrednošću.

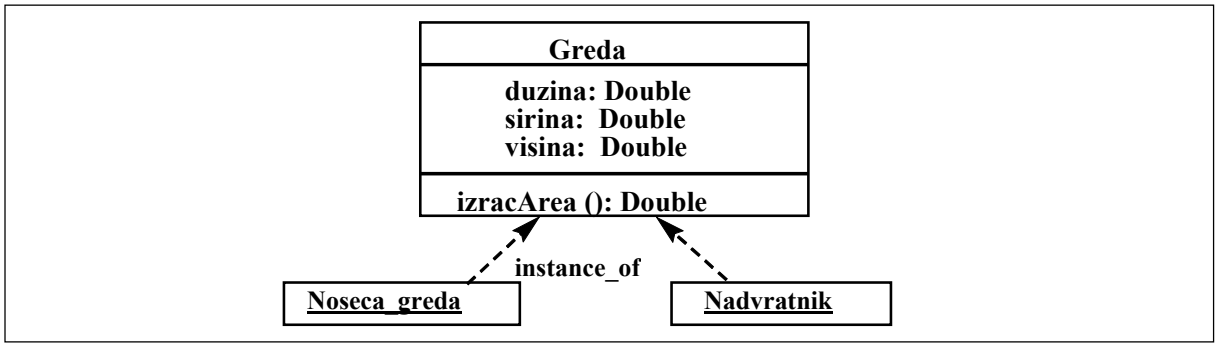
Primer



6.2.3. Umetanje ranije uvedenih klasa u vezi sa objektnim konceptom programskog jezika JAVA. Prvi kontakt sa JAVA programom.

Za primer projektovanja noseće konstrukcije može se do sada uvedeni koncept predstaviti u programu JAVA. To predstavljanje treba istovremeno da bude uvod u JAVA programiranje.

Elementarni problem, predstavljen u UML notaciji (minimalni nivo)



Dakle, problem se sastoji iz jedne klase pod imenom „Greda": biće kreirane dve instance klase, objekti „Noseca_greda" i „Nadvratnik".

Prevođenje sa detaljnim opisom pomoću komentara

```
/**
 * Program I.
 * Prvo se definise klasa „Greda". Ovde se primenjuje kljucna rec „class"
 */

public class Greda
{
```

```
// Telo klase;
// definise sadrzaj klase;
// prvo atribute saglasno principu tajnosti, pri cemu se pravo pristupa tj.
// vidljivost definise naredbom „private”; „private” znaci da je pristup
// moguc samo unutar klase;
// ukoliko nije dozvoljen pristup, onda je pristup moguc samo unutar
// određenog paketa;
// ostala prava pristupa su: „protected”, „private” ;
// Atributi

public double duzina, sirina, visina;

// ova deklaracija odgovara opstoj deklaraciji atributa u obliku:
// „pravo_pristupa tip_podatka ime_atributa”
// Metode
// specijalna metoda za konstruisanje objekata je tzv. Konstruktor, koji je
// predviđen sa pravom pristupa „public”; u suprotnosti sa „private”,
// omogucava „public” opsti pristup.
// Konstruktor

public Greda(double d, double s, double v)
{
        // Povezivanje sa atributima
        duzina = d;
        sirina = s;
        visina = v;

}

// Time prepisane standardne konstrukcije se ponovo definisu

public Greda ()
{
}

// Sve metode se definisu prema sledecem opstem obliku.
// Pravo_pristupa tip_rezultata metoda (lista parametara)
// {
// ...sadrzaj metode...
// }
// „Double” je pritom podatak tipa real; ukoliko se neki rezultat vraca
// preko metode, to se postize naredbom „return”
// Metode za izlaz podataka (rezultata)

public double izlDuzina()
{
        return duzina;
}
```

```
        public double izlSirina()
        {
                return sirina;
        }

        public double izlVisina()
        {
                return visina;
        }

        // Metod za izracunavanje povrsine poprecnog preseka
        public double izracArea()
        {
                return sirina*visina;
        }
}

        // Kraj klase Greda
        //
        // Za testiranje klase Greda – nezavisno od dalje primene – definise se
        // klasa GredaTest ;
        // njome ce biti konstruisani Greda-objekti (primeri) „Noseca_greda” i
        // „Nadvratnik”, uz pomoc gore definisanih konstruktora:

        public class GredaTest
        {
        // Da bi se klasa „GredaTest” iz JAVA-Byte-Code Interpretera mogla
        // startovati, mora se u JAVA pozivna rutina „main” izvrsiti; ovo je
        // polazna tacka glavnog programa za resavanje problema;
        // Pritom se treba pridrzavati sledece sintakse:

        public static void main(String[] args)
        {
                // kao parametri se moraju tzv. Komandni argumenti poziva, String args[], tj.
                // jedno polje – niz preneti. String je pritom jedna predefinisana klasa; u
                // ovom slucaju nije potrebno uneti nikakve argumente.
                // Konstruisanje objekata
                // Procedura: postavlja se jedan tzv. referentni atribut i tada u njega novi
                // objekat – sa standardnim konstruktorom – obezbeđuje; pritom se treba
                // pridrzavati sledeceg formalizma:
                // Ime_klase Referentni_atribut_klase = new Ime_klase( );
                // ili sa sopstveno definisanim konstruktorom, koji sadrzi listu parametara:
                // Ime_klase Referenca_klase = new Ime_klase(Lista_parametara);
                // Sa standardnim konstruktorom

                Greda Noseca_greda = new Greda();

                // Dodeljivanje atributa objektu „Noseca_greda” pomocu operatora
```

```
        // pristupa funkcionise samo ako su atributi u pravu pristupa „public”.

        Noseca_greda.duzina = 3.5;
        Noseca_greda.sirina = 0.2;
        Noseca_greda.visina = 0.35;

        // sto znaci da opsti oblik pristupa u vrednost atributa ima formu:
        // Referenca_klase.ime_atributa
        // Alternativno mozemo primeniti i sopstveno definisani konstruktor,
        // Kreiranje sa sopstvenim konstruktorom

        Greda Nadvratnik = new Greda (3.5, 0.2, 0.35);

        // Izdavanje (stampanje) atributa za objekat Nadvratnik pomocu metode
        // System.out.println iz JAVA paketa java.lang , u koji je smesten recnik
        // (jezicki interface)

        System.out.println ("Atributi nadvratnika");
        System.out.println ("Duzina=" + Nadvratnik.izlDuzina());
        System.out.println ("Sirina=" + Nadvratnik.izlSirina());
        System.out.println ("Visina=" + Nadvratnik.izlVisina());

        // Proracun povrsine poprecnog preseka nosece grede sa stampanjem

        System.out.println ("Pop. Pres. Nos. grede=" + Nadvratnik.izracArea());


    }
    // Kraj main
}
// Kraj klase GredaTest
```

Izlaz:

```
Atributi nadvratnika
Duzina=3.5
Sirina=0.2
Visina=0.35
Pop. Pres. Nos. grede=0.069999999999999993
```

Fizički tok prevođenja

Izvorni JAVA-program, koji se unosi pomoću nekog editora čini jednu celinu za prevođenje. U ovom slučaju sastoji se ta celina iz dve klase, klase „Greda” i klase „GredaTest”. Ta celina se memoriše kao izvorna datoteka „GredaTest.java” (produžno ime = „.java”).

Izvorna datoteka pod imenom „GredaTest.java” se može u sledećem koraku JAVA-kompilatorom „javac” prevesti u Java-Bytecode (*.class); ukoliko je u izvornoj datoteci definisano više klasa, za svaku klasu se generise jedna „.class” datoteka. Ograničenja u pogledu broja klasa ne postoje, no za jednu izvornu datoteku mora biti raspoloživa samo jedna „public” klasa.

6.2.4. Modeliranje hijerarhijski struktuiranih veza između Klasa / Objekata / Nasleđivanja / Generalizacija / Specijalizacija

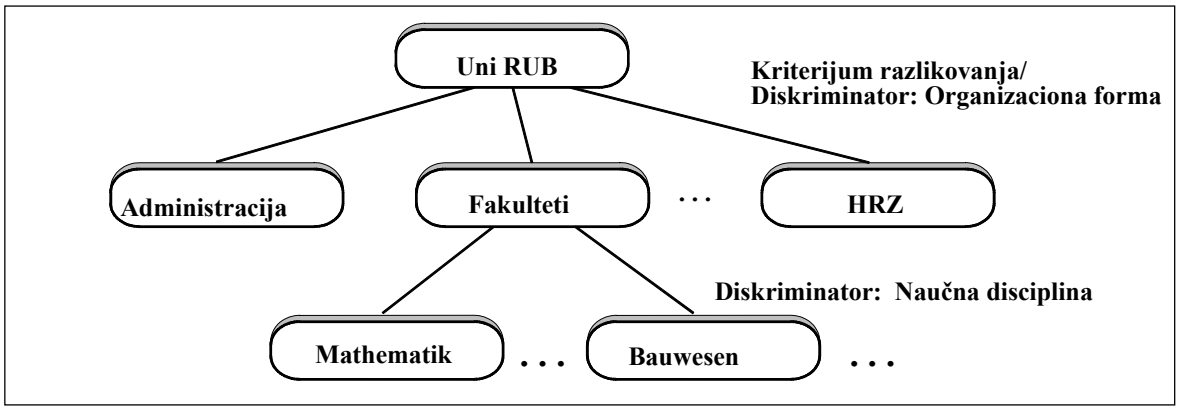
Dosada su klase – s obzirom na diskusiju o apstraktnim klasama – posmatrane poglavito izolovano. Problemi iz oblasti primene u osnovi predstavljaju mrežu međuodnosa različitih klasa. Zato je osnovni zadatak pri razvoju softvera za određenu postavku problema da utvrde osnovnu mrežu međusobnih odnosa klasa, da analiziraju njihove unutrašnje međuzavisnosti i (na osnovu UML) da modeliraju, da bi na osnovu toga mogli dobijene modele da prevedu u izvršni program.

U notaciji UML za ove zadatke stoje na raspolaganju klasni i objektni dijagrami, koji služe tome da se iznađene strukturalne međuzavisnosti formalno i razumljivo mogu (matematički) definisati. Ali, to ne znači da formalni opis veza mora da bude uvek jednoznačan. Dijagrami klasa imaju često osobinu da se za jedan isti problem može dati više reprezentacionih alternativa (ali svaka pojedina se može svesti i na neku treću alternativu).

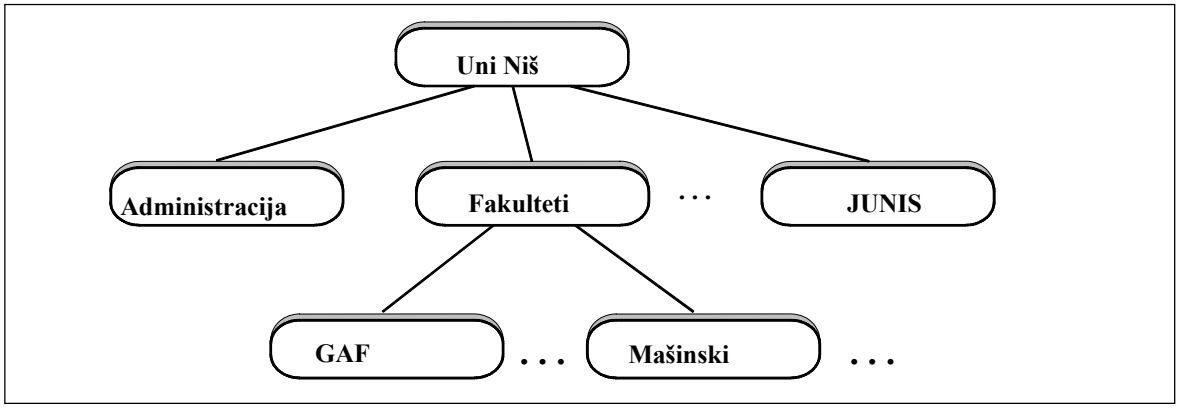
Kao u običnom životu, ali takođe i u mnogim naučnim disciplinama (biologija, matematika, mehanika, arhitektura, ...), naročito često zastupljena prirodna sistematika, „hijerarhija", primenjuje se u orijentaciji objekata kao oblik uređenja i struktuiranja. Njome se mogu dobro obuhvatiti sve one činjenice, koje se mogu podeliti na pojedinačne oblasti (mogu se klasifikovati), pri čemu se oblasti struktuiraju u obliku stabla. Pri tome se i preporučuje da se u praksi, pored hijerarhije uvode i druge vrste sistematizacije (heterarhije, mreže, itd.). Za sada ostanimo kod hijerarhija, pošto one predstavljaju najjednostavniji oblik sistematizacije.

Tipični primeri hijerarhija iz univerzitetske svakodnevnice

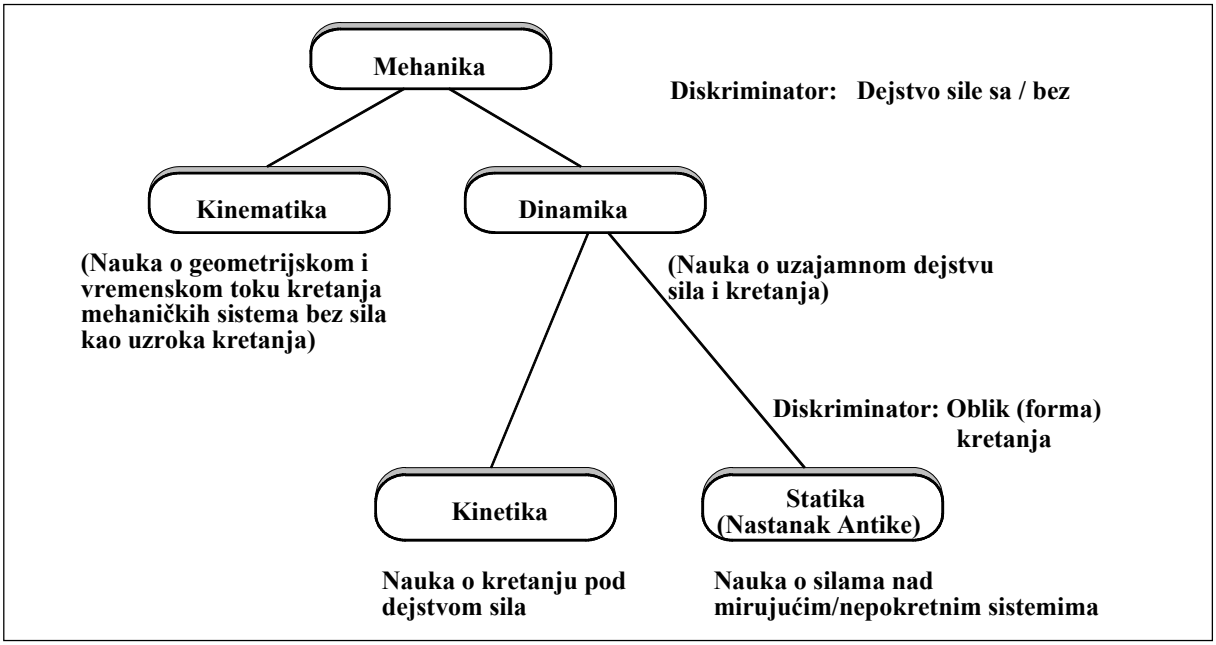
Hijerarhija: Univerzitet Bochum



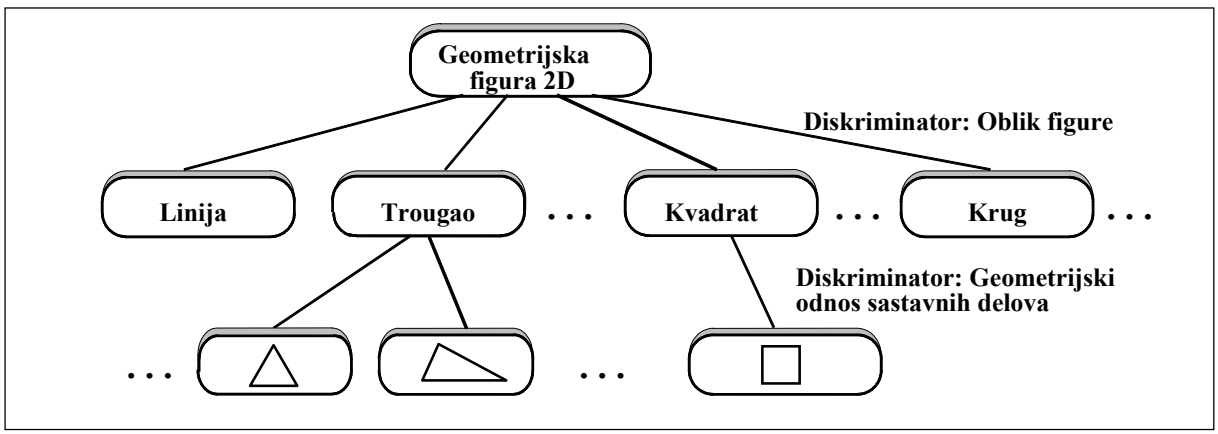
Hijerarhija: Univerzitet Niš



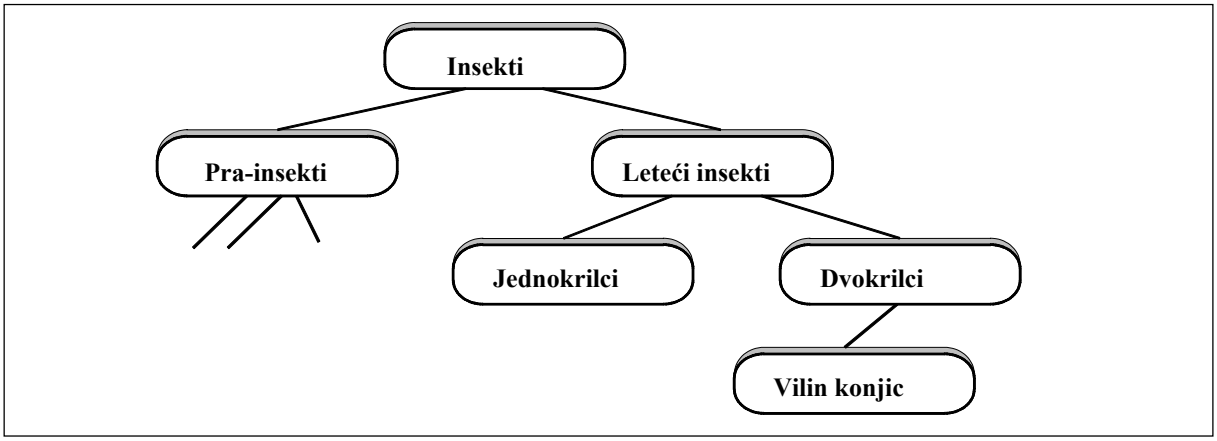
Mehanika



Primenjena informatika / CAD-sistem (2D)

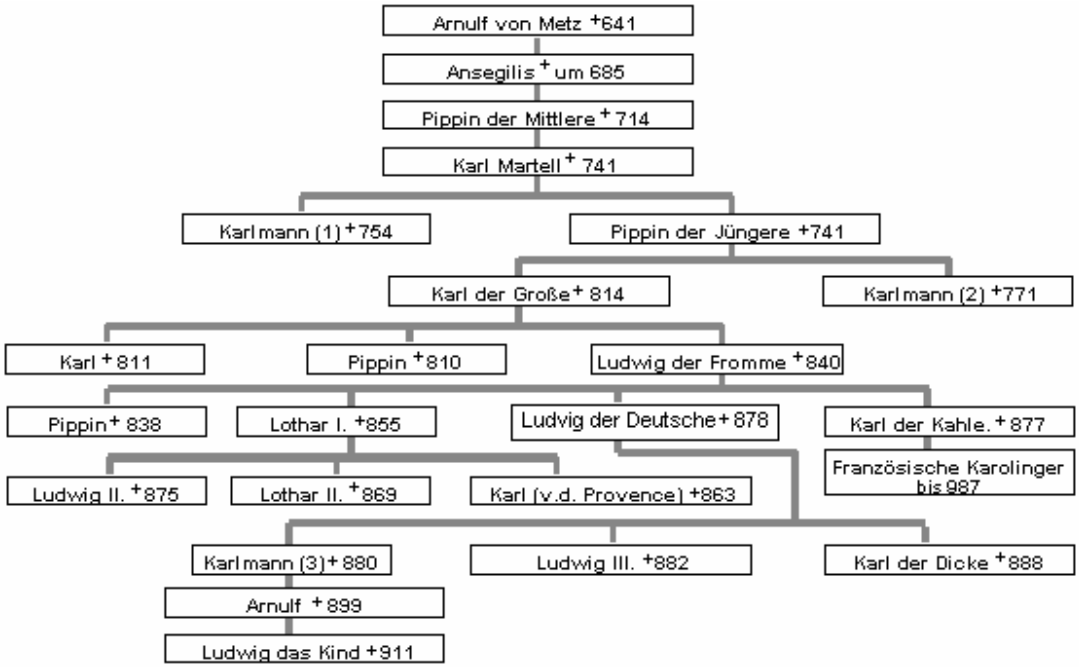


Biologija (etnomologija, „prethodnik" taksonomije)

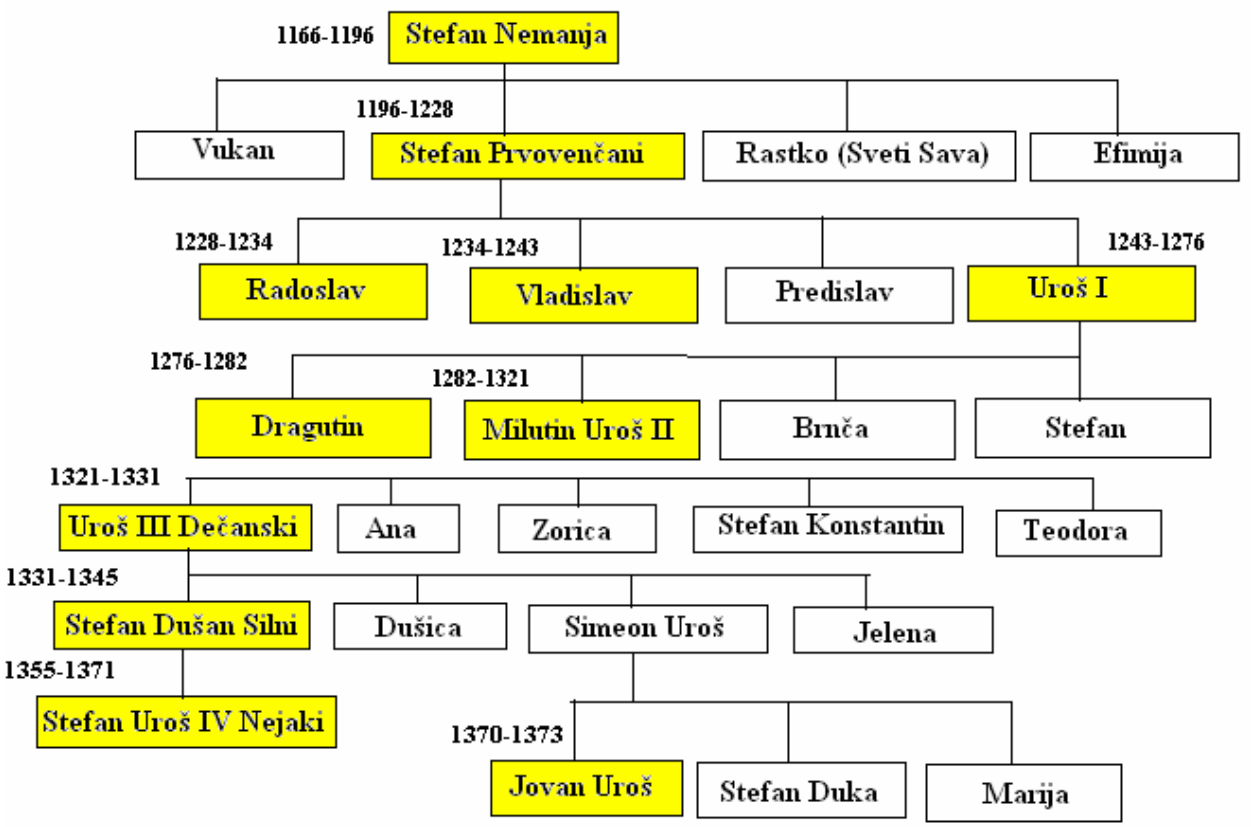


Za hijerarhije je karakteristično da svi članovi hijerarhije imaju slične osobine i kada se u pojedinoj jasno međusobno razlikuju. Takođe je karakteristično da svi članovi nadređenog nivoa predstavljaju jednu „specifičnost" nižeg nivoa; obrnuto, istovremeno su više postavljeni nivoi „generalizacije" niže postavljenih nivoa.

Kako predstavljanje hijerarhija podseća na familijarni rodoslov (videti rodoslove Karolinga i Nemanjića) i karakteristike se prenose iz nivoa u nivo, izraz „nasledna hijerarhija" se odomaćio.



Rodoslov loze Karolinga



Rodoslov loze Nemanjića

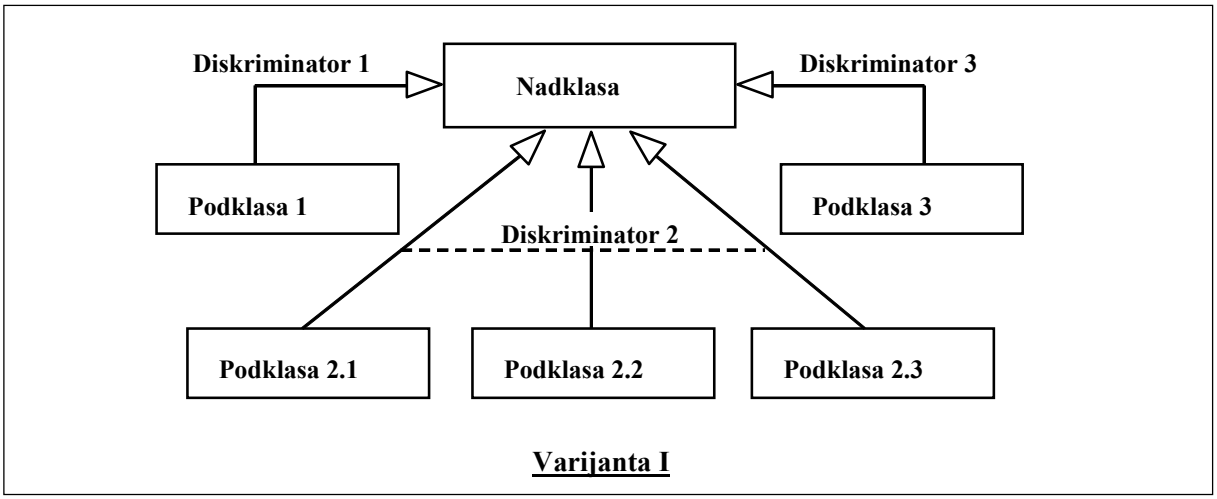
Zašto je razumljivo primenjivati naslednu hijerarhiju za informatička rešenja problema, ili, još više, zašto ona čini osnovni sastavni deo objektno-orijentisanog modeliranja i programiranja? (Svi značajni objektno-orijentisani programski jezici podržavaju nasledne hijerarhije direktno pomoću odgovarajućih jezičkih alatki u obliku ključnih reči!).

Odgovor glasi: Pored, sa nasleđivanjem povezanim principom uređivanja, glavna prednost leži pre svega u tome što se – saglasno logici „rodoslova" („naslednog drveta") – osobine određenih objekata kao što su atributi i metode, mogu automatski iz jednog hijerarhijskog nivoa u drugi hijerarhijski nivo (top-down) relativno lako obrađivati („nasleđivati"). Pritom je dovoljno „informaciju" samo jedanput tamo definisati, gde se ona prvi put koristi. Time lako ostvarljiv, implicitni (tj. automatski ugrađen) mehanizam nasleđivanja se tada stara za to da se ova „osobina" „prenosi" na dole. Ovom koncepcijom se značajno smanjuje rad softverista u smislu kodiranja, a time i mogućnost unošenja greške pri kreiranju softvera uopšte. Posebno se povišava ponovna upotrebljivost softvera (Re-usability).

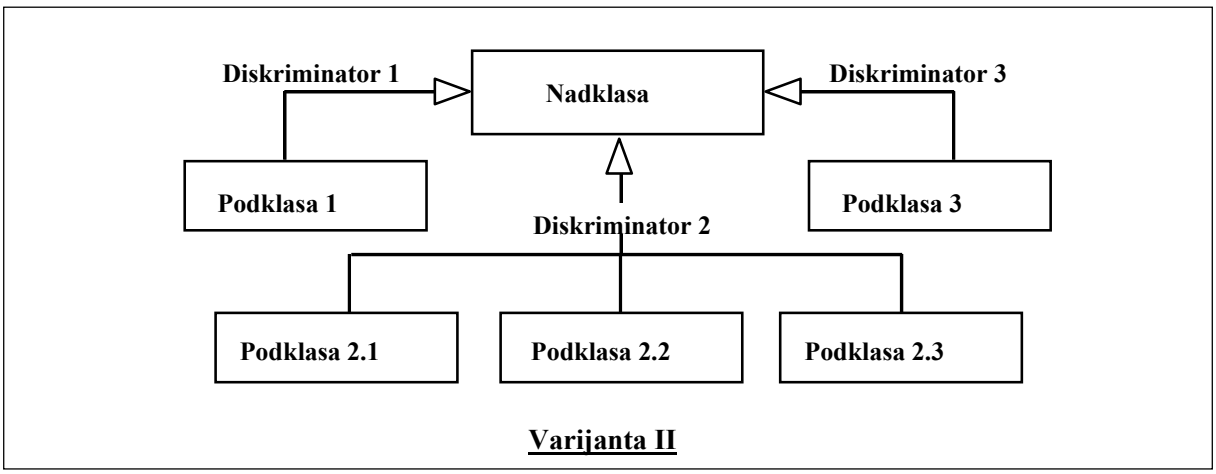
Princip nasleđivanja se pritom ne treba primeniti naivno; treba paziti da će se pri nasleđivanju informacija iz jedne tzv. gornje klase u odgovarajuće donje klase preneti takođe i sve odgovornosti i zadaci gornje klase, koji su dati pravom pristupa i metodama. Dalje treba paziti da se „nasleđene osobine" moraju izmenama prilagoditi zadacima donje klase: nasleđene metode treba uz pridržavanje određenih pravila (jednakost imena i saglasnost u listi parametara) prepisati! U određenim slučajevima se mogu i nove uvesti. Posebno se mora ispitati da li nasledna hijerarhija odražava realnu taksonomiju, ili će druge alternative biti bolje usaglašene.

UML- notacija

UML predstavljanje hijerarhije u obliku klasnog dijagrama široko se oslanja na logiku naslednog stabla. Nasledni odnos između gornjih klasa sa karakteristikama predodređenog značaja i direktnih podklasa se jasno označava strelicom sa neispunjenom vrhom strelice. Vrh strelice je pritom okrenut u pravcu gornje klase; pojedine strelice se mogu po izboru jednom zajedničkom linijom spojiti u jednu strelicu.



Varijanta I

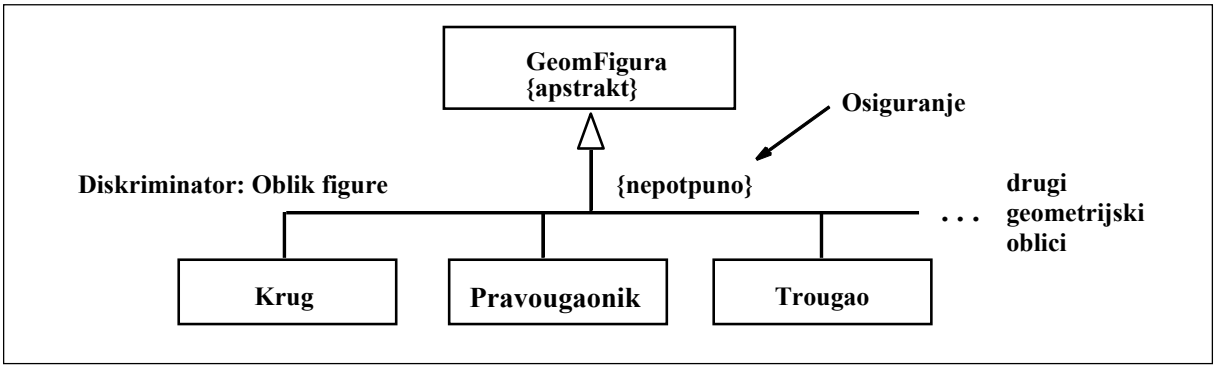


Varijanta II

Primer (Odsečak iz jednog CAD-sistema)

U radnom prozoru jednog CAD-sistema treba generisati krugove, pravougaonike, trouglove, itd. i izdati ih (nacrtati) kao grafičke elemente. Za krugove, pravougaonike i trouglove se uvode kao „nacrti" („mustre") klase „Krug", Pravougaonik" i „Trougao". Za različite geometrijske figure se, da bi se mogao proučiti princip nasleđivanja, formira (apstraktna) nadklasa „GeomFigura" (geometrijska figura), koja nastaje generalizacijom klasa „Krug", „Pravougaonik" i „Trougao". Obrnuto, klasa „GeomFigura" se dobija „specijalizacijom" klasa „Krug", „Pravougaonik", „Trougao", itd.

Nastali klasni dijagram se tako predstavlja kao na sledećoj šemi (gruba struktura bez detalja u oblasti atributa i metoda):



Kako klasa „GeomFigura" ne proizvodi sopstvene objekte (oni se isključivo generišu preko podklasa!), već je uvedena kao osnovna klasa za druge klase, to je ona apstraktna klasa: njoj će biti dodeljena primedba {apstrakt}. Pri primeni u jednom JAVA - programu ova predefinicija ima određeno dejstvo na formulisanje problema (apstraktna klasa ima apstraktne metode, koje moraju odgovarati zadatim JAVA pravilima!).

U klasnim dijagramima se takođe upotrebljavaju:

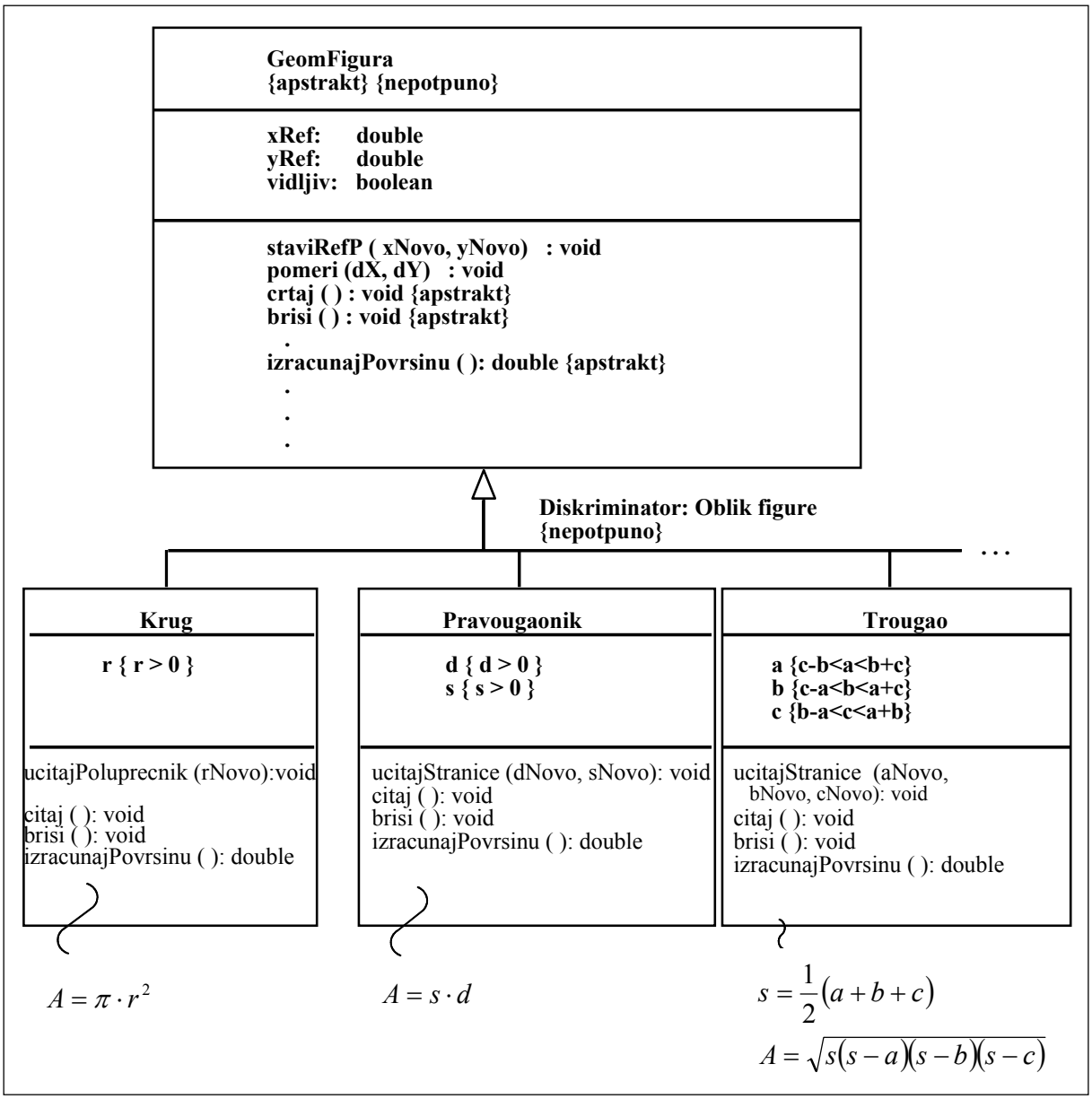
- Diskriminator hijerarhije,
  ovde „OblikFigure", koji „implicitno" definiše uređenu strukturu klasnog dijagrama (strogo uzev, imena klasa i podklasa predstavljaju vrednosti atributa „Diskriminator")

- Osiguranje hijerarhije je dato u vitičastim zagradama i ono označava da je hijerarhija još „nepotpuna" (Ostala osiguranja mogu biti „preklopljena", „disjunktna", „potpuna").

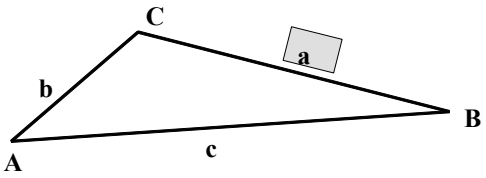
Uputstvo: o uređenju osobina

Osobine su unutar nasledne hijerarhije tačno tamo unesene, gde saglasno semantici problema imaju svoje poreklo. Drugim rečima: uređenje ne treba da bude urađeno prema svrsishodnosti ili radi uštede u kodiranju, već samo na osnovu unutrašnjih razmatranja. Cilj mora uvek biti da se svet problema (domen problema) predstavi na najprirodniji način.

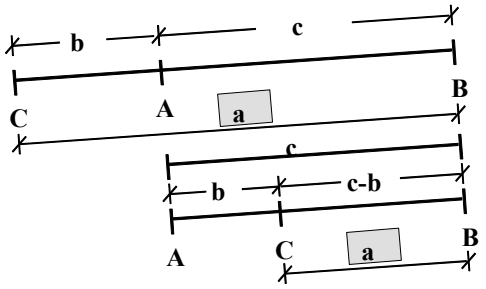
Po utvrđivanju gore prodiskutovane grube strukture, uvedene klase se upotpunjuju sledećim sadržajima (atributima i metodama):



Usvojena osiguranja služe tome da uvedeni geometrijski detalji budu uneseni korektno. Kod kruga i kod pravougaonika moraju poluprečnik, odnosno dužina i širina uvek biti pozitivni. Kod trougla su za svaku stranicu data odgovarajuća gornja i donja granica. Na primer, za stranicu a, ako je b+c=a, odnosno c-b=a, umesto trougla dobićemo duž. Zato mora da važi c-b <a < b+c. Analogno važi za stranice b i c.

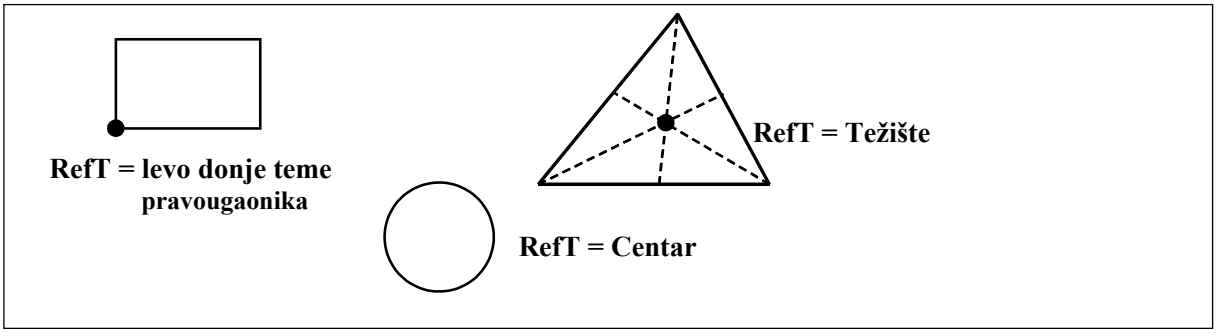


Za stranicu a, ako je b+c=a, odnosno c-b=a, umesto trougla dobićemo duž



Mora da važi c-b <a < b+c. Analogno važi za stranice b i c.

U apstraktnoj nadklasi „GeomFigure" biće za celokupno problemsko područje uvedene istovetne – i zato i sa istim imenom označene - metode „staviRef(erentnu)T(acku)", „pomeri", „crtaj", „brisi" i „izracunajPovrsinu". Te metode, koje su usko vezane sa fizičkim konceptom podklase, biće, što je potpuno razumljivo, prvo realizovane i definisane i stoga uvedene kao apstraktne metode. To znači da se te metode sa svojim sadržajem tek u izvedenim podklasama mogu definisati. Jedino što se mora uvesti pri primeni jezika JAVA u apstraktne klase je oznaka (signatura) te metode, tj. ime metode (sa pravom pristupa), lista parametara i tip izlaznih rezultata. Drugim rečima, apstraktna metoda ne sadrži definiciju, nema telo metode, već samo glavu metode (signaturu). U apstraktnu klasu se dalje uvode atributi xRef, yRef za referentnu tačku kao markantnu tačku pri unošenju neke geometrijske figure (videti sledeću skicu).



Takođe se uvodi prekidač sa vidljivošću (da/ne). Navedene veličine (xRef, yRef, vidljiv) uvode se u nadklasu „GeomFigura", pošto će one biti korišćene za sve klase hijerarhije.

Karakteristike podklasa su generalno samoobjašnjive. Svi karakteristični geometrijski elementi, kao „Radius r" za „Krug", „Duzina d" i „Sirina s" za „Pravougaonik", kao i dužine stranica „a", „b" i „c" za „Trougao" biće dati – sa određenim osiguranjima. U delu metoda podklasa „unesi ...(...) " biće poziva na određene delove (r tj. d i s, tj. a, b i c.), pri čemu će zadata osiguranja (opsezi vrednosti) biti proverena. Pored toga, u podklasama će apstraktne metode nadklasa biti „napunjene životom", tj. u zavisnosti od geometriske figure biće uvedeno crtanje („crtaj"), brisanje („brisi"), ili računski postupak („izracunajPovrsinu").

Mehanizam nasleđivanja deluje u ovom slučaju tako što se iz osnovne klase „GeomFigura" mogu preuzeti (naslediti) atributi „xRef", „yRef" i „vidljiv", kao i da se nasleđuju sve metode iz „GeomFigura".

6.2.5. Uvođenje nasleđivanja u JAVA

Za primer razmatran u prethodnom odeljku iz osnovne oblasti jednog 2D-CAD sistema treba primeniti saglasno gore sastavljenom klasnom dijagramu mehanizme nasleđivanja JAVA programa. Grafički postupci ovde ostaju još narazmotreni, pošto neophodna znanja o grafičkim elementima JAVA još nisu izložena. Stoga se primena JAVA odnosi samo na numerički deo gore datih klasnih dijagrama (tj. za proračun poprečnih preseka).

JAVA-Programski kôd

```
/**
 * Program II.
 * Osnovna klasa GeomFigura
 */

public abstract class GeomFigura
{
        // atributi za pravo pristupa unutar svih izvedenih klasa
        protected double xRef, yRef;
        protected boolean vidljiv;
        // metode (za sada samo metode za proracun poprecnog preseka!)
        public abstract double izracunajPovrsinu();
}

// Kraj klase GeomFigura
// Podklase nasledne hijerarhije (na primeru klase Pravougaonik)
// Nasleđivanje ostvareno kljucnom reci „extends"

public class Pravougaonik extends GeomFigura
{
        // Atributi
        private double d, s;
        // Metode
        public void ucitajStrane(double dNovo, double sNovo)
        {
                // Osiguranja
                if(dNovo > 0.0 && sNovo > 0.0)
                {
                        d = dNovo;
                        s=sNovo;
                }
```

```
                else
                {
                        System.out.println ("Greska ! Nije pravougaonik");
                }
        }

        public double izracunajPovrsinu()
        {
                return d*s;
        }
}

// Kraj podklase Pravougaonik

public class NasledjivanjeTest
{
        // Pozivna rutina main
        public static void main(String[] args)
        {
                // Kreiranje objekta
                Pravougaonik P1 = new Pravougaonik();
                P1.ucitajStrane(300.0, 200.0);
                System.out.println("Povrsina pravougaonika P1 = " + P1.izracunajPovrsinu());
        }
        // Kraj main
}
// Kraj Klase NasleđivanjeTest
```

Izlaz:

Povrsina pravougaonika P1 = 60000.0

6.2.6. Kontrolne strukture

6.2.6.1. Osnove / Motivacija

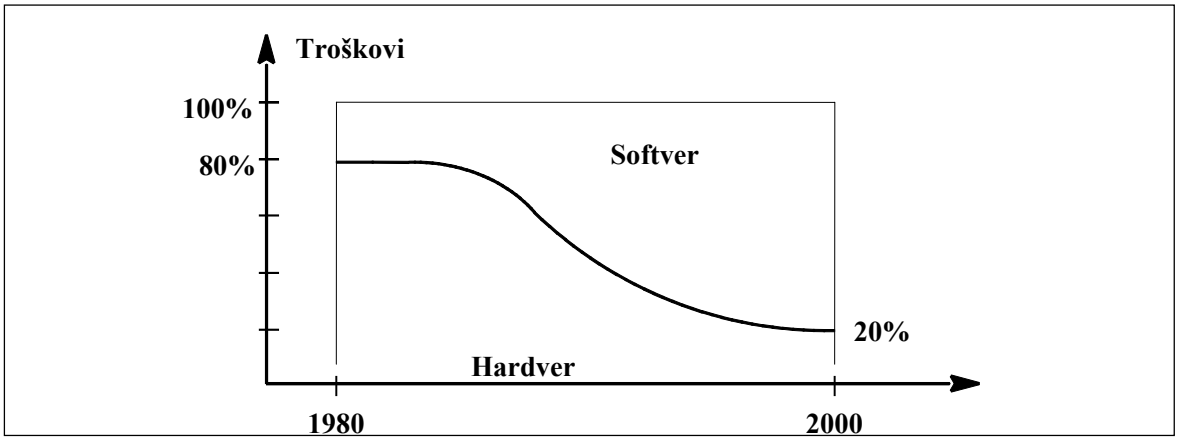
Do sada su od metoda uvedenih klasa uglavnom razmatrane samo jednostavne naredbe – iz aspekta primene u predhodnom odeljku razmatranog osiguranja (if – else – struktura). Jasno je da su za praktičnu primenu potrebne i druge strukture, da bi se i kompleksni zahtevi mogli ispuniti.

Izmene jednostavne (linearne) strukture vode u praksi vrlo brzo do nepreglednih situacija, gde se podesnim pravilima ne može intervenisati. Smisao i svrha ovih pravila je da se razvoj softvera, a to posebno važi za razvoj objektno-orijentisanog softvera, usmeri u uređenom i kontrolisanom pravcu. U drugom slučaju postoji opasnost da se u metodama objektno-orijentisanih programa proizvodi „špageti-kod", koji nije više pod kontrolom.

Metode koje se ne mogu rekonstruisati (nekontrolisane) ne odgovaraju standardima kvaliteta modernog softvera. Među različitim kriterijumima za ocenu kvaliteta softvera tipični su sledeći:

- pouzdanost,
- pogodno održavanje,
- laka izmenjivost,
- efikasnost,
- pogodnost korišćenja (user-friendly) i
- niski troškovi.

Troškovna komponenta, na koju sa svoje strane utiču osobine kao što su održavanje, efikasnost, itd., je pritom od posebnog značaja. To je stoga što su troškovi softvera poslednjih godina, u poređenju sa hardverom, u stalnom porastu. (Pogledati sledeći grafikon radi uvida u situaciju, odakle se vidi da se odnos troškova hardvera i softvera u periodu od 1980. do 2000. godine kretao od 80:20 do 20:80).



Opšte prihvaćeni princip je da kvalitativno dobar koncept softvera doprinosi smanjenju troškova softvera. Najvažniji preduslov za kvalitativno dobar projekat softvera je da se razvoj softvera odvija „kontrolisano i podložno kontroli", na sličan način kao i razvoj tehničkih proizvoda i konstrukcija u inženjerskim disciplinama.

U inženjerskim naukama se pre svega pokazalo da rad sa osmišljenim standardima i normama povoljno deluje na smanjenje troškova. Stoga je prenošenje tog iskustva na proceduru pri programiranju metoda blisko razmišljanju. Osnovno pravilo je: ne može se sve, što je negde ostvarljivo, prevesti u dijagram toka (flow-chart). Mnogo češće se sasvim određene konstrukcije nazvane strukturni elementi prihvataju, i to samo u kontrolisanom broju (Princip ograničenja konstruktivnih varijanti).

Interesantno je da je upravo manji broj strukturnih elemenata dovoljan da formuliše kompleksne algoritme visokog nivoa. U osnovi su dovoljna četiri osnovna strukturna elementa:

- **Sekvenca**,
- **Alternativa** ili **Grananje**, tj. **Selekcija**,
- **Višestruka alternativa**/**Razlikovanje slučajeva**,
- **Iteracija**.

Ukoliko se formulisanje metoda bazira na normiranim strukturnim elementima, dobija se još jedna značajna prednost: izbegava se zavisnost od jednog konkretnog programskog jezika. Drugačije rečeno: ukoliko se kontrolni (blok) dijagram operacija definiše isključivo pomoću normiranih strukturnih elemenata, odgovarajuće prevođenje u određeni programski jezik je sekundarno, pošto se postupak konverzije može sprovesti šematski (uporedi definiciju dijagrama klasa u objektno orijentisanom modeliranju, gde se uslovi slično memorišu).

Da bi se postupak prevođenja olakšao, uvedene su grafičke oznake za predstavljanje strukturnih elemenata, tzv. **struktogrami** (**Nassi-Schneider-dijagrami**). Radi usaglašavanja ovih struktograma u veće programske celine/programe uvedena su opšta „konstrukciona pravila". Oba, struktogrami i konstrukciona pravilla, služe tome da se u objektnom programiranju primenjene metode mogu kontrolisano i uz nadzor ugraditi u softver.

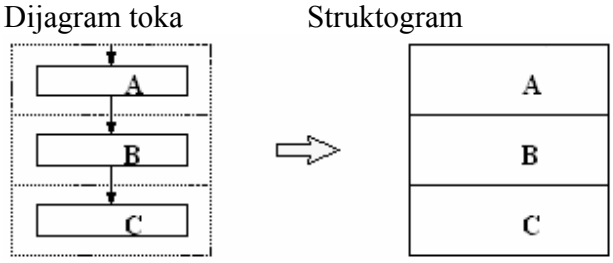
6.2.6.2. Sekvenca

Sekvenca, kao najjednostavnija algoritamska struktura u operacijama je već dovoljno poznata.

Definicija:

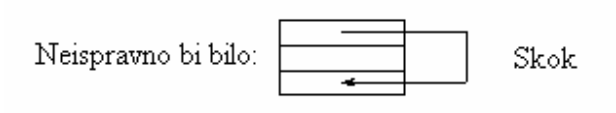
Jednostavan niz naredbi ili blokova naredbi, koji slede u nizu.

Grafički prikaz:



Konstrukciono pravilo / Pravilo Odozgo na dole (Top-Down)

Pojedini blokovi (ovde, A,B,C) se moraju počev odozgo u redosledu na dole izvršavati; tek po izvršenju poslednjeg bloka (ovde C), može se sekvenca napustiti.



Uputstvo:

Pojedini blokovi naredbi mogu se sastojati iz više posebnih blokova, koji moraju biti stavljeni između vitičastih zagrada; pritom je moguće definisati veličine, koje važe samo unutar jednog dela bloka.

JAVA-primer

```
// Naredbe A

int j = 100;
System.out.println("Naredbe A zavrsene");

// Naredbe B

{
        int i = 1;
        double x = 3.0;
        System.out.println("Naredbe B zavrsene");
}
```

6.2.6.3. Alternativa / Grananje / Selekcija – (if-else)

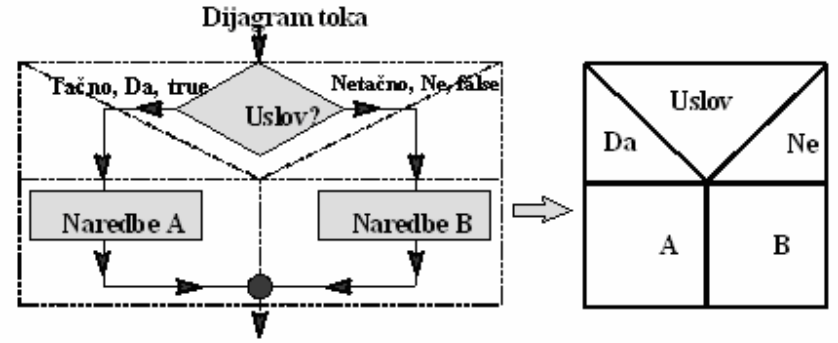
Kod određenih operacija je neophodno izvršiti izbor naredbi koje će se izvršiti; ovde kompjuter izvršava „odluke", koje omogućavaju određene forme „inteligentnog ponašanja". Pritom je izvanredno moćna relativno jednostavna struktura IF-ELSE, koja se nalazi u mnogim jezicima i upravo je bila primenjena pri realizaciji osiguranja (red veličine, oblast vrednosti). Da bi se ograničilo konstruktivno mnoštvo pri primeni na moguće komplikovanije razgranate strukture, njena je primena namerno izostavljena.

Alternativa / Selekcija

Kratak opis:

Obrada dva alternativna bloka naredbi u zavisnosti od vrednosti logičkog (Boolov) izraza, koji može imati saglasnost (vrednost = „true"), ili nesaglasnost (vrednost = „false") kao odgovor na neki zadati uslov; vrednost „true" odgovara iskazu istinitosti „tačno", tj. „DA", dok vrednost „false" odgovara iskazu istinitosti „netačno", tj. „NE".

Grafički prikaz:



Primena u JAVA sa naredbom if-else
Opšti oblik:

```
if (Bulov izraz ili uslov)
{
        naredbe A;
}

else
{
        naredbe B;
}
```

Semantika:

Izračunavaju se uslovi formulisani kao logički izraz. Kao izraz može pritom stajati neko poređenje, ali i podatak tipa „Boolean", tj. logički podatak. Ako je rezultat „true", izvršava se grupa naredbi A, koja sledi odmah iza uslova, dok se u drugom slučaju izvodi grupa naredbi B. Naredbe A tj. B se pritom nalaze u vitičastim zagradama.

Radi formiranja logičkih izraza u JAVA se primenjuju operatori poređenja i logički operatori. Operatori poređenja upoređuju objekte istog tipa.

Operator poređenja	**Matematički simbol**	**Primer**
= =	=	x = = y
!=	≠	x != y
<	<	x < y
<=	<=	x <= y
>	>	x > y
>=	>=	x >= y

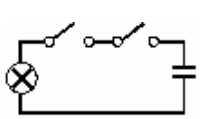
Logički operatori omogućavaju realizaciju veza između logičkih vrednosti.

Logički operator	**Matematički simbol**	**Primer**
!	(Negacija) ¬	! bool
&&	(I, Konjunkcija) ∧	bool1 && bool2
||	(ILI, Disjunkcija) ∨	bool1 || bool2
^	(Ekskluzivno ili)	bool1 ^ bool2

Dejstvo logičkih operacija (predstavljeno tablicama istinitosti)

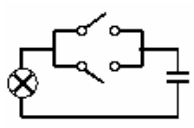
bool	! bool
true	false
false	true

! - Negacija



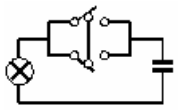
bool1	bool2	bool1 && bool2
true	true	true
true	false	false
false	true	false
false	false	false

&& - Konjunkcija



bool1	bool2	bool1 || bool2
true	true	true
true	false	true
false	true	true
false	false	false

||- Disjunkcija



bool1	bool2	bool1 ^ bool2
true	true	false
true	false	true
false	true	true
false	false	false

^ - Ekskluzivno ILI

Pri primeni operatora && ili || drugi izraz se tek onda izračunava kada prvi dobije vrednost „true" ili „false". Ukoliko, nasuprot tome, treba svaki izraz da se izračuna, moraju se primeniti obični operatori & i |.

Nadalje treba obratiti pažnju da operatori poređenja i logički (Boolovi) operatori treba da budu rangirani prema prioritetu. Pri tome, u jeziku JAVA važi sledeći prioritet:

Opadajući prioritet

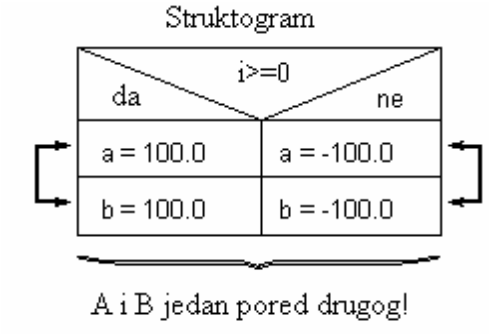
Operator (nepotpuno)	**Asocjativnost, utvrđuje u kom redosledu se izvršavaju operacije istog prioriteta**
Aritmetički operatori: ++ -- +- **Inkrement Dekrement Predznak**	**unarna**
< / % **+ -**	**s leva udesno** **s leva udesno**
Operatori poređenja: **< <= > >=**	**s leva udesno**
Logički operatori: **== !=** **&** **^** **|** **& &** **| |**	**s leva udesno** **s leva udesno** **s leva udesno** **s leva udesno** **s leva udesno** **s leva udesno**
Operatori dodeljivanja vrednosti: **= += -=** ***= /= %=**	**s desna ulevo** **s desna ulevo**

Operacije nad bitovima i šiftovanja nisu navedene. Operatorima nad bitovima mogu se obrađivati celi brojevi ili logičke vrednosti (važno kod posebnih operacija, na primer pri računanju sa stepenima od 2).

Alternacija (grananje)

Konkretni primer:

```
if (i >= 0)
{
//   naredbe A
 a = b = 100.0;
}
else {
 //  naredbe B
a = b = -100.0;
}
```

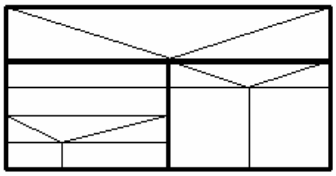


Uputstvo za konstruisanje

a) Do niza naredbi A ili B stiže se isključivo preko izračunavanja uslova; naredbe A i B nemaju dakle nikakvu međusobnu vezu; drugim rečima, alternacija se može samo tada napustiti, kada su A ili B potpuno obrađeni.

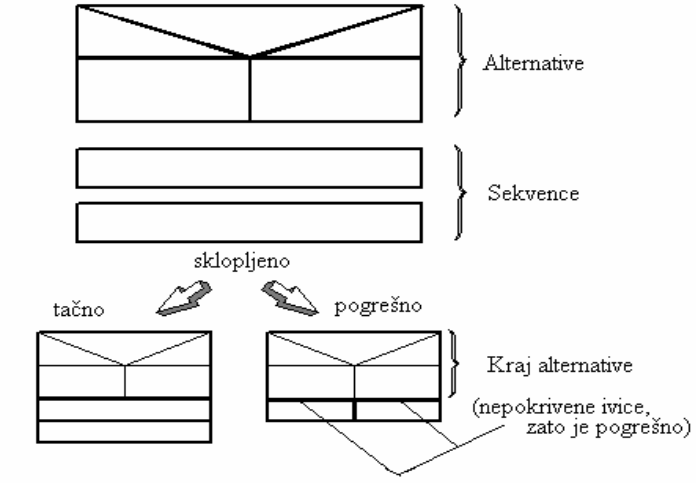
b) Naredbe A ili B mogu sadržati i druge strukturne elemente:

Primer uz b):



Konstrukciono pravilo

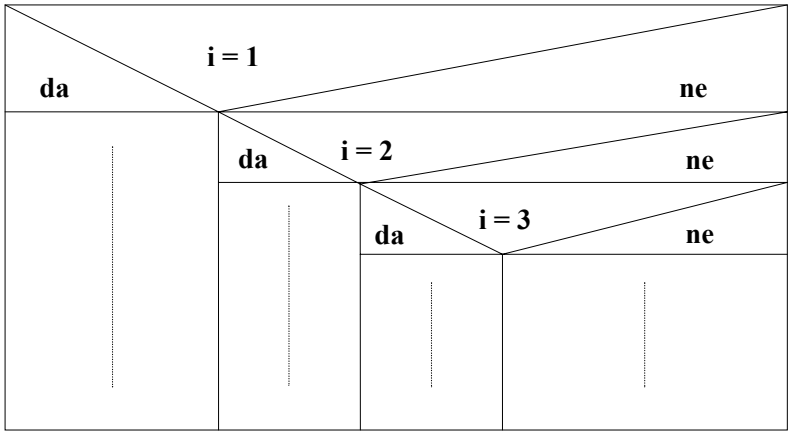
Strukturni blokovi istog nivoa moraju se oslanjati vertikalnim stranicama. Primeri:



6.2.6.4. Višestruko grananje / Razlikovanje slučajeva (switch)

U praktičnim primenama treba često uvesti razlikovanje slučajeva. Iako se razlikovanje slučajeva može realizovati ugnježdavanjem jednostavnih alternativa, iz razloga bolje preglednosti i čitljivosti je bolje višestruko grananje realizovati jednim strukturnim elementom – pogotovu onda kada je broj mogućih slučajeva veliki.

Višestruko grananje sa jednostavnim alternativama (mali broj alternativa)

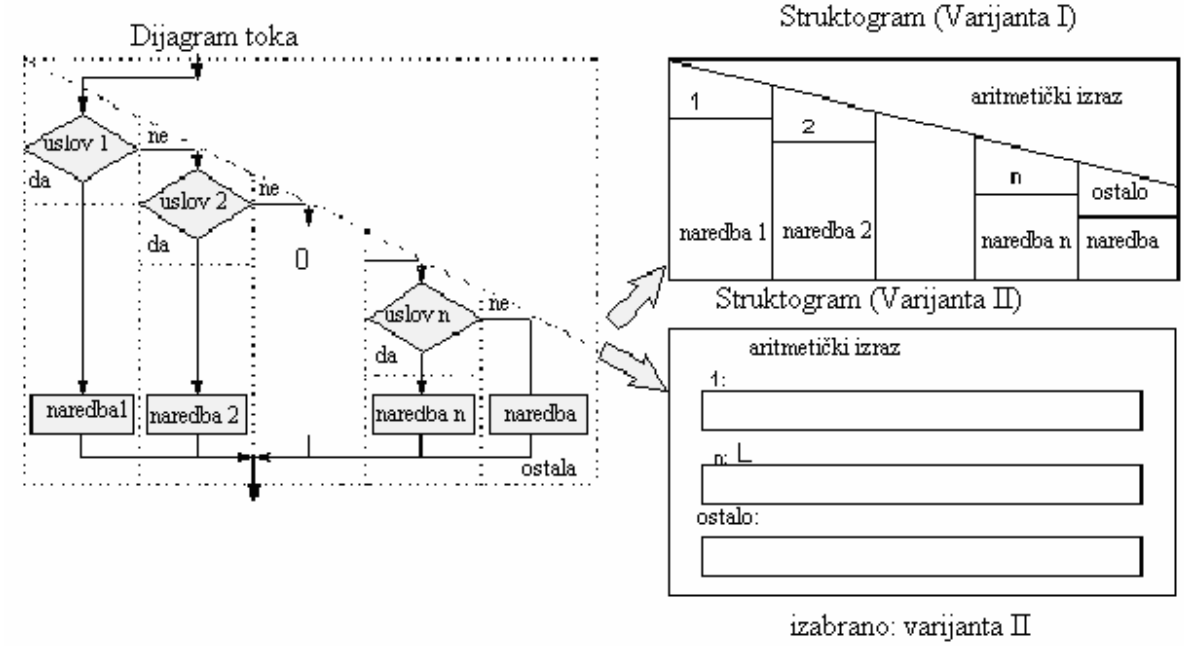


Višestruka alternativa

Kratak opis

To je uređena grupa alternativa, pri čemu svakoj alternativi pripadaju sasvim određene operacije; nepripadajući slučaj se posebno reguliše, jer bi u suprotnom nastala nekontrolisana stanja.

Grafički prikaz



Prevođenje u JAVA vrši se naredbama „switch”, „case”, „default” i „break”:

```
switch (aritm. izraz.)
{
        case 1:
                naredbe 1;
                break;
        case 2:
                naredbe 2;
                break;
                // dalje naredbe sa pripadajucim naredbama …
        case n:
                naredbe n;
                break;
        default:
                ostale naredbe;
}
```

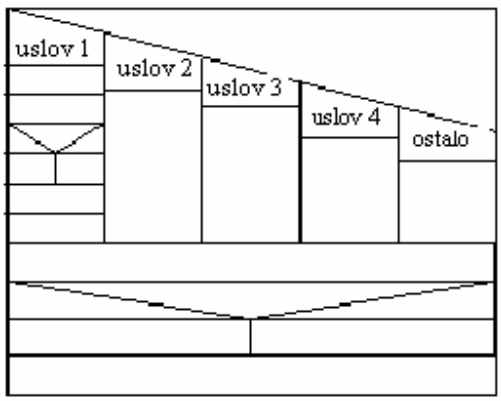
Semantika

Vrednost aritmetičkog izraza biće upoređena po redosledu sa svakom iz „case” sledećom konstantom 1, 2, ..., n; ukoliko jednakost postoji, biće izvršene naredbe koje pripadaju toj konstanti. Ukoliko se ne pojavi jednakost, izvršiće se naredbe koje slede iza „default”. Da bi se sprečilo da se izvrše naredbe koje slede iza slučaja koji je zadovoljen (default je isključen), mora se niz naredbi završiti naredbom „break”; „break” se stara za to da se kontola programa prenese na sledeću konstrukciju iza višestruke alternative/razlikovanja slučajeva.

Uputstvo

Pri doslednoj primeni objektno orijentisanog modela je moguće da se „switch” konstrukcija izbegne koristeći objektnoorijentisane mehanizme kao što su nasleđivanje ili polimorfizam. Stoga će višestruka alternativa, pošto je značajan elemenat algoritamskog načina razmišljanja, ovde biti detaljno predstavljena (ona je preglednija i jednostavnija nego neka objektno-orijentisana konstrukcija).

Struktogramski primer



Primer: Proračun različitih površina (stub)

```
int i;
double povrsina, ivica, sirina, visina;
ivica = 7.0; sirina = 13.0; visina = 19.0;
// Definisanje na osnovu logike problema; ovde staviti
switch (i)
{
        case 1:
                povrsina = ivica * ivica;
                break;
        case 2:
                povrsina = sirina * visina;
                break;
        default:
                System.out.println("Odgovarajuci slucaj ne postoji!");
        }
```

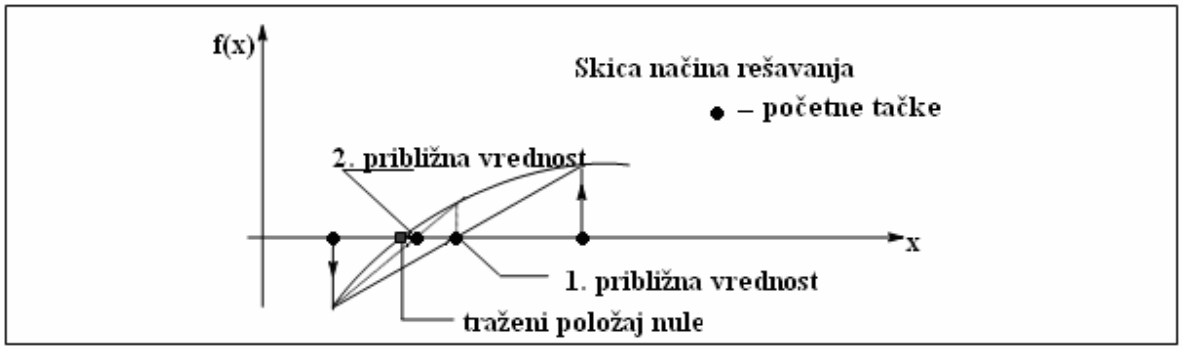
6.2.6.5. Iteracija

Numeričke operacije zahtevaju često ponovljeno izvršavanje određenih naredbi, dok ne bude ispunjen određeni uslov prekida. Pritom je broj ponavljanja nepoznat. Isto tako može biti razumno naredbama koje se ponavljaju dodati jedan iteracioni uslov i deo koji se ponavlja izvršavati samo kada je uslov za iteracije ispunjen. U mnogim slučajevima je broj ponavljanja takođe poznat, pri čemu su tada predviđene specijalne konstrukcije za ponavljanje.

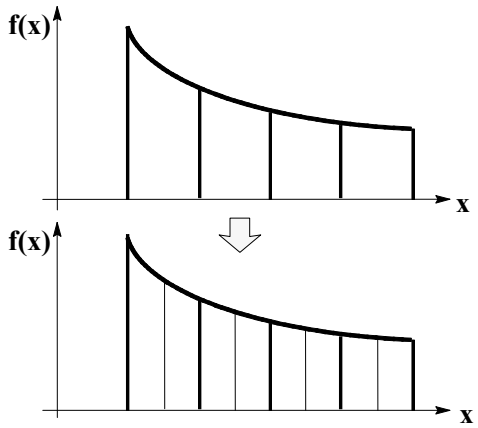
Dakle, iteracije su osnovni princip u numeričkim aplikacijama. Navešćemo kao primer sledeće mogućnosti primene:

- Izračunavanje najmanjeg zajedničkog sadržaoca (pogledati Euklidov algoritam u odeljku za objašnjenje algoritamskog modela razmišljanja),

- Izračunavenje nule neke funkcije,



- Izračunavanje površina.



Za iterativno rešavanje problema stoje u JAVA na raspolaganju tri mogućnosti:

- Iteracija sa iteracionim uslovom na početku dela koji se ponavlja (while), tzv. odložena ili unapred ispitana petlja,
- Iteracija sa uslovom prekida na kraju dela koji se ponavlja (do-while), tzv. neodložena, ili petlja sa kontrolom na kraju,
- Iteraciona petlja (for), ili tzv. brojčana petlja.

Sva tri iteraciona oblika biće nadalje razmatrana.

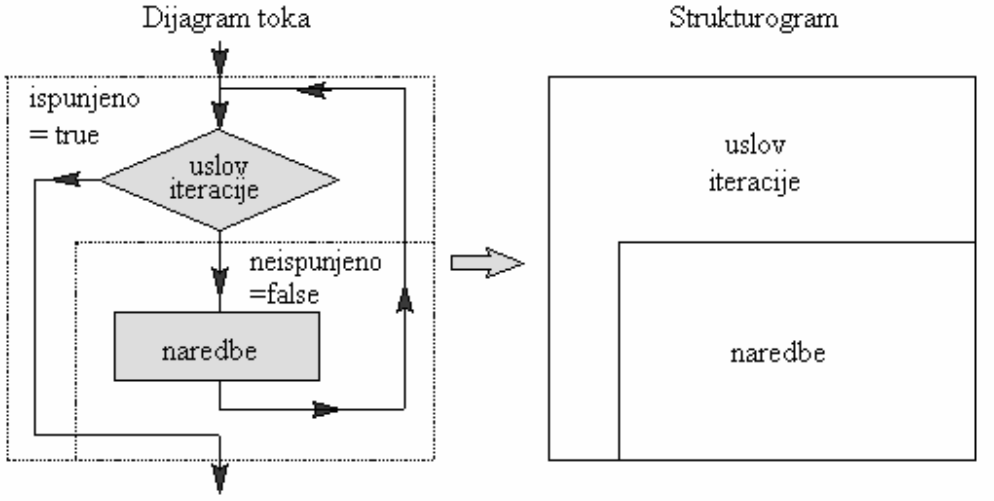
a) **Iteracija sa iteracionim uslovom na početku (while)**

Iteracija / while

Kratki opis

Operacija ponavljanja, pri čemu se na početku ponavljjućeg dela ispituje da li se ovaj mora obraditi. Ukoliko uslov nije ispunjen, sledi „otkazivanje”, tj. izvršavanje će se nastaviti naredbom iza „while” konstrukcije.

Grafički prikaz



Primena u JAVA sa naredbom „while”

```
while (logički izraz) {

    naredbe;

}
```

Semantika

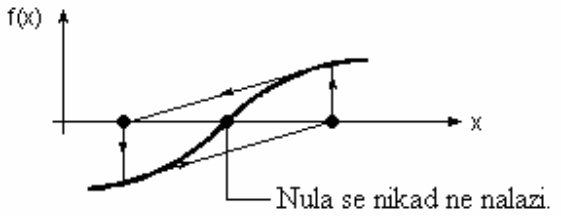
Logički izraz se izračunava pre izvršenja naredbi koje se nalaze između vitičastih zagrada, „{” i „}”. Ukoliko ovaj izraz, kao uslov za izvršenje iteracija, ima vrednost istinitosti „true”, naredbe koje treba da se ponavljaju biće izvršene; ukoliko je vrednost „false”, preći će se na strukturni elemenat iza „while”. Ponavljanja će se izvršavati doklegod je iteracioni uslov ispunjen.

Uputstvo

Ukoliko se, iz bilo kojih razloga, uslov „false” nikada ne dogodi, (dakle grananje „neispunjeno” - nikada se ne aktivira), nastaje tzv. beskrajna petlja. To bi bilo neispravno, jer je onda neophodan „ručni prekid”. Stoga treba unapred paziti da se takav slučaj ne dogodi.

Primer za ovu situaciju

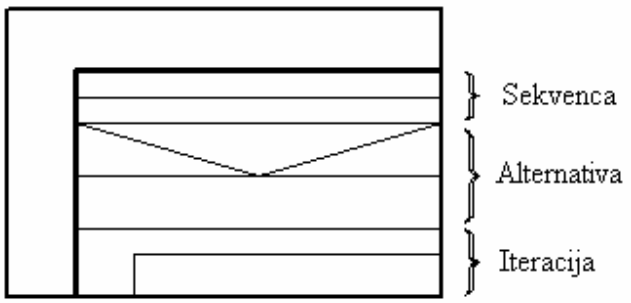
Traženje nule sa divergencijom



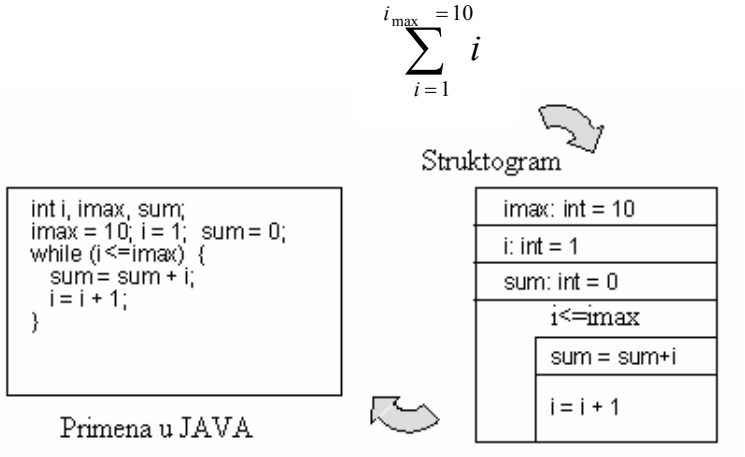
Konstrukciono pravilo

Deo koji se ponavlja može i sam sadržati druge strukturne elemente

Primer



Primer primene:



b) **Iteracija sa iteracionim uslovom na kraju (do - while)**

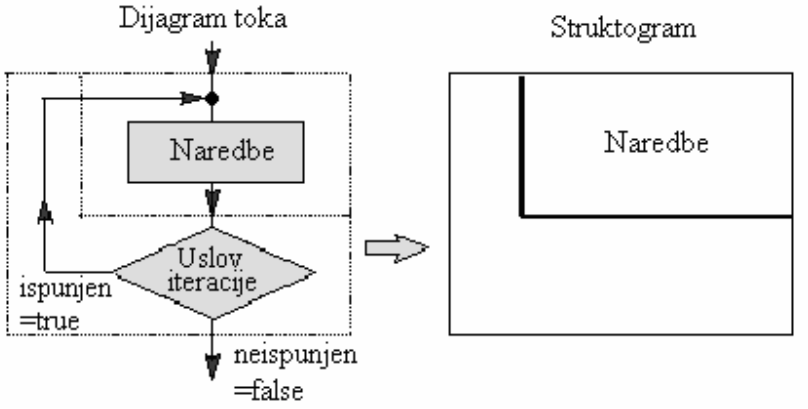
Kod mnogih problema u praksi je pametno da se deo koji se ponavlja – bez obzira na iteracioni uslov - izvrši najmanje jednom, da bi se tek na kraju dela koji se ponavlja ispitalo da li treba prekinuti iteraciju ili ne. Za slučajeve takve vrste stoji na raspolaganju, kao mala modifikacija „while" konstrukcije, tzv. „do-while" konstrukcija, koja se označava kao petlja sa kontrolom na kraju.

Iteracija / do-while

Kratak opis:

Operacija ponavljanja, pri čemu se tek na kraju dela koji se ponavlja ispituje da li će deo koji se ponavlja biti ponovo obrađen.

Grafički prikaz:



Primena u JAVA sa naredbama „do" i „while"

```
do {
    naredbe;
} while (logicki izraz);
```

Uputstvo:

Obratiti pažnju na „ ; ” iza logičkog izraza!

c) Iteraciona petlja (for)

Glavna karakteristika napred opisanih itaracionih varijanti („while” i „do-while”) se sastoji u tome da u oba slučaja broj ponavljanja nije neophodno a priori tačno utvrditi (navedeni primer izračunavanja sume nije posebno dobro prilagođen jednoj „while” petlji). U inženjerskim primenama ima doduše često problemskih postavki, kod kojih je broj ponavljanja unapred tačno poznat – tipična oblast primene je vektorski račun.

Jedan primer: Skalarni proizvod dva vektora

$$\vec{V}^T \cdot \vec{W} = \sum_{i=0}^{2} V_i \cdot W_i = V_0 \cdot W_0 + V_1 \cdot W_1 + V_2 \cdot W_2$$

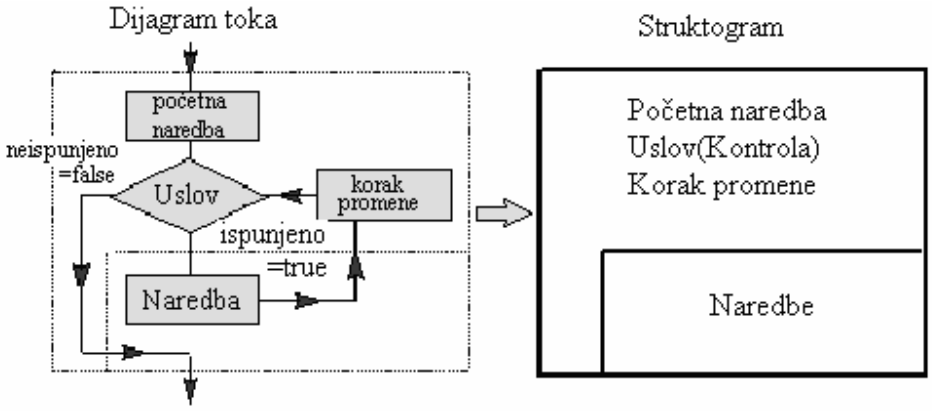
Ovde je utvrđen broj ponavljanja (3) za operacije $V_i \cdot W_i$ sa $i = 0,1,2$. Za izvršenje iteracija sa utvrđenom strukturom ponavljanja većina programskih jezika nudi, kao uostalom i JAVA, posebnu konstrukciju, takozvanu petlju („for”-loop, „for”- petlja, „for”- Schleife (nemački), ili brojčana petlja).

Iteraciona petlja / for

Kratak opis:

Ponavljanje operacija sa poznatim brojem ponovljenih prolaza.

Grafički prikaz:



Primena u JAVA sa naredbom „for”

```
for (pocetna vrednost; kontrolna vrednost; korak) {
   naredbe;
}
```

Semantika:

Po izračunavanju početnih vrednosti ispituje se uslov iteracije (operacija poređenja); ukoliko je vrednost logičkog izraza = „true" (tj. iteracioni uslov ispunjen), naredbe između vitičastih zagrada biće izvršene; zatim se izvršava naredba za korak. Ponavljanja se izvršavaju sve dok vrednost logičkog izraza ne postane = „false" (tj. uslov iteracije ne bude više ispunjen).

Uputstvo:

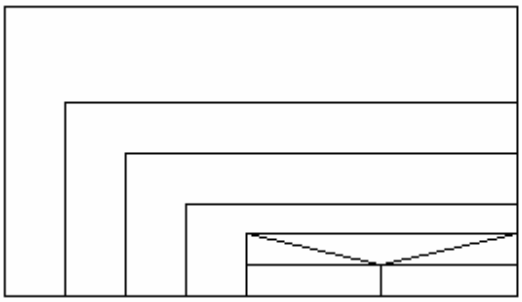
a) Petlje kao konstrukcije imaju smisla kada deo koji se ponavlja može da se formuliše u zavisnosti od koraka.

b) Treba obratiti pažnju da su moguće konstelacije pri kojima uslovi itaracije nisu ili nikada ne mogu biti ispunjeni, tako da naredbe „for" petlje nikada ne mogu da budu ispunjene.

Konstrukciono pravilo:

for-petlje se mogu višestruko jedna unutar druge ugnezditi.

Primer:

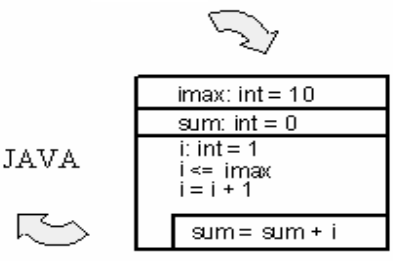


Primer primene (videti primer u 6.6.5.1.):

$$\sum_{i=1}^{i_{max}=10} i$$

```
int imax, sum;
imax = 10; sum = 0;
for (int i =1; i <= imax; i++) {
    sum = sum + i;
}
```

Realizacija u JAVA programu.



6.2.6.6. Poboljšanja

Pri radu sa strukturnim elementima/struktogramima oformile su se pojedine dopune tj. poboljšanja, koja mogu doprineti i poboljšanjima pri formulisanju metoda. Na kratko ćemo preći na jednu svrsishodnu dopunu.

Iteracija sa unutrašnjim prekidom

Do sada objašnjene iteracione konstrukcije su tako korišćene, da se delovi koji se ponavljaju uvek izvršavaju u potpunosti. Međutim, u praksi postoje češći slučajevi da se delovi koji se ponavljaju moraju pre vremena prekinuti, dakle, deo koji se ponavlja ne treba da se kompletno izvrši.

Izumitelji struktograma, Nassi i Shneiderman, su stoga predložili da se za slučajeve te vrste koriste iteracione konstrukcije „while", odnosno „do-while" da bi se realizovao interni kriterijum prekida. Ukoliko se on dogodi pri obradi dela koji se ponavlja (telo petlje), petlja se napušta prevremeno (tzv. neprirodan izlaz).

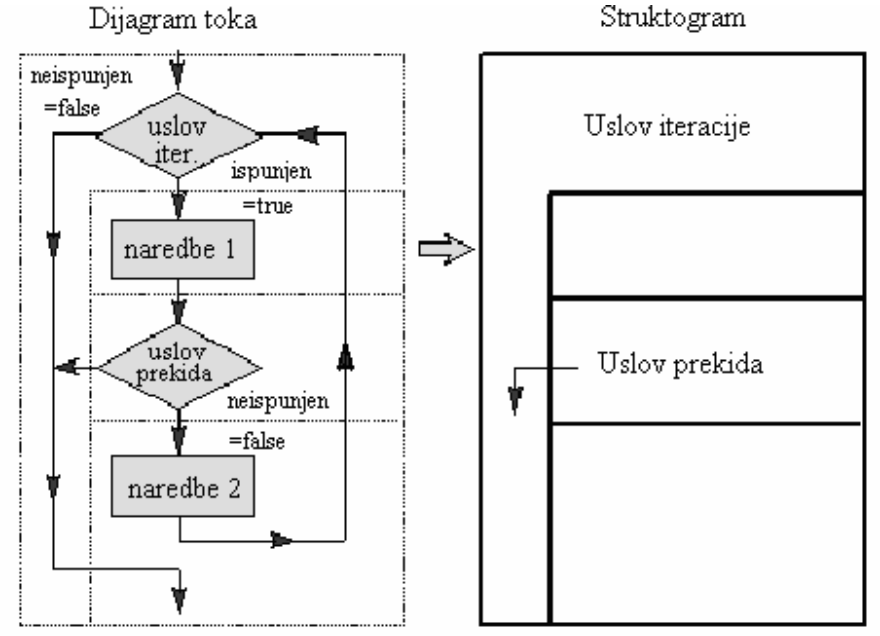
Iteracija sa prekidom

Kratak opis:

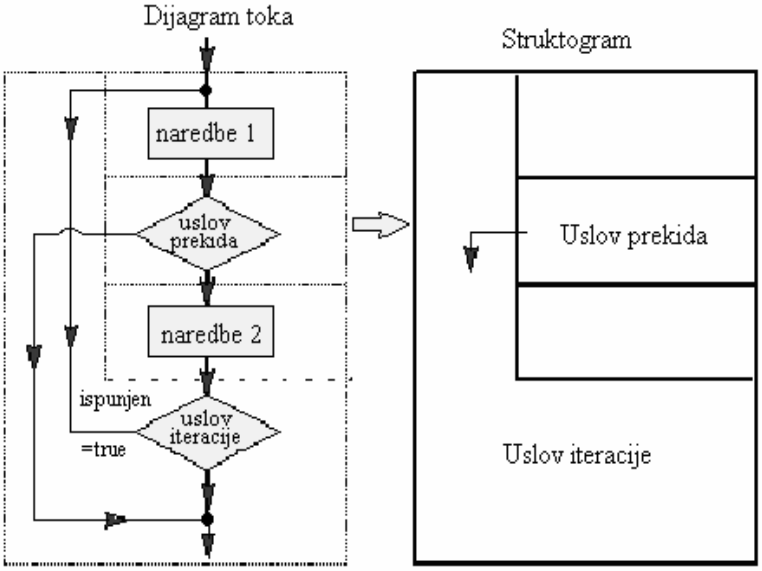
Ponavljanje operacije sa mogućnošću da se deo koji se ponavlja može prevremeno napustiti, u zavisnosti od nekog unutrašnjeg kriterijuma prekida.

Grafički prikaz:

a) „while" sa prekidom



b) „do-while” sa prekidom



Primena uslova prekida u JAVA naredbom „break” kao i „if” naredbom unutar naredbi „while” ili „do-while”

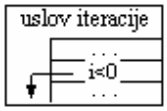
```
if (logicki izraz / uslov prekida)
    break;
```

Semantika:

Kao uslov prekida ispituje se izraz „uslov prekida” (operacija poređenja); ukoliko se dobije vrednost „true”, tj. uslov je ispunjen, iteracija se prekida i nastavlja se prvim strukturnim elementom iza iteracije; ukoliko je vrednost „false”, tj. uslov prekida nije ispunjen, nastaviće se sa izvršavanjem naredbi 2 unutar iteracione petlje.

Primer:

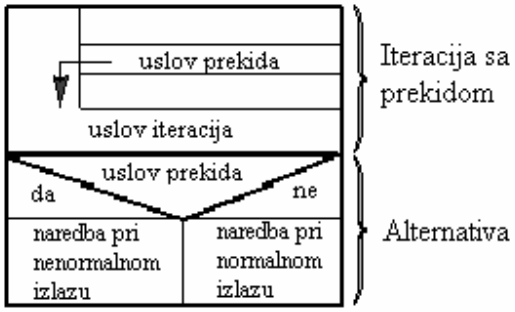
```
if (i < 0)
  break;
```



Konstrukciono pravilo:

Struktura prekida se može dalje tako poboljšati da se anomalijski prekid u nastavku na strukturni elemenat učini „raspoznatljivom” pomoću jedne ubačene alternative, što se vidi iz sledećeg primera.

Primer:



6.2.6.7. Vrednovanje u smislu softver inženjerstva

Za projektovanje dobrog softvera rad sa strukturnim elementima/struktogramima ima izvanredno veliki značaj. U tom smislu se pokazuju sledeće prednosti:

- Procesi osmišljavanja se odvijaju struktuisano; oni su pregledni i uvek se mogu opisati. Moto: „Prvo misli, potom kodiraj".
- Tokovi izvršenja programa su i bez izvršavanja na računaru pod kontrolom (statička kontrola).
- Softver se može lakše modifikovati tj. menjati, a time lakše administrirati i održavati; razvoj softvera će time biti ekonomičniji u poređenju sa ne-struktuiranim konceptom.
- Razvoj softvera orijentiše se na „standarde", tako da se razumljivost i transparentnost značajno poboljšavaju.

6.2.6.8. Uvođenje kontrolnih struktura u praktičnu primenu

Na osnovu jednog egzemplarno većeg primera primene u oblasti projektovanja konstrukcija (statika) treba pokazati kako se kontrolne strukture koriste za realizaciju „obuhvatnijih" operacija. Pritom je primer tako izabran da se objektno-orijentisano modeliranje u obliku klasa i objektdijagrama odvija u pozadini. Predmet interesovanja treba da bude pre svega nastajanje metoda. Ovde treba još jednom naglasiti da i algoritamskom (proceduralnom) modelu razmatranja pri objektnoorijentisanom modeliranju inženjerskih problema pripada veliki značaj. Praktični primer treba takođe da bude iskorišćen i za to da se pokaže kako se značajna interakcija između korisnika softvera i računara – posebno u inženjerskim naukama – može realizovati. Mogućnost da se utiče na tok podataka u računaru, na primer, interaktivnim unošenjem podataka, upravljanjem ili kontrolom (Prekid, mere korekcije sa mogućnošću restartovanja, između ostalog), je karakteristika modernih softverskih sistema i podržan je pre svega pomoću JAVA programa.

U programu JAVA posebno stoji na raspolaganju paket „java.awt" (abstract windowing toolkit), koji nudi bogat repertoar „windows-tehnika" (analogno u inženjerskoj informatici primenjivom operativnom sistemu Windows NT). Integrisanjem Windows-tehnika u naše (objektno-orijentisane) programe, unosi se svakako višestruka kompleksnost u poređenju sa proceduralnim programskim jezicima, koji nisu sadržali windows-tehnike, već samo elementarne, ali zato „brzo programirljive instrukcije". Neophodna kompleksnost u JAVA programu se ipak isplati, jer je konfor korisnika windows-baziranih programa (user-friendly) značajno poboljšan, a time je i prihvatanje programa od strane korisnika znatno povećano.

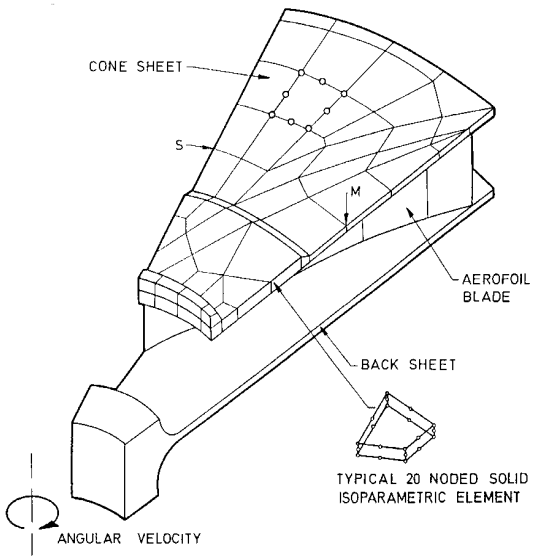
Postavka problema iz statike (Mehanike)

Danas je uobičajeno da se statični proračun nosećih sistema, posebno kada su komplikovani, izvršavaju programima. Kao opšte upotrebljiva i vrlo efikasna računska metoda se pokazala tzv.

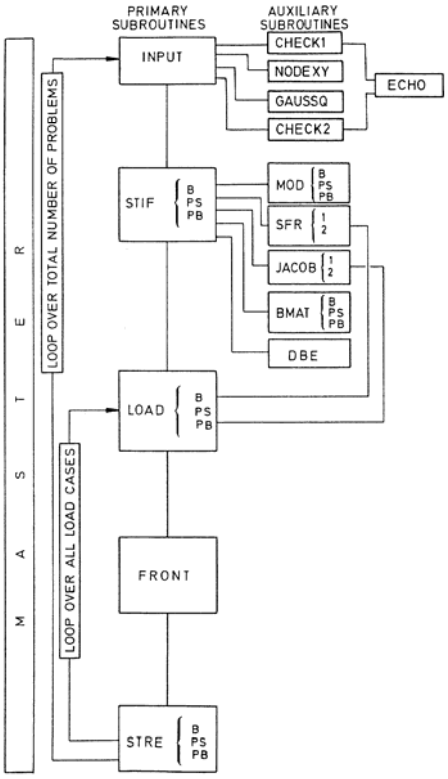
Final Element Method (FEM), Metoda konačnih elemenata. Pomoću FEM je moguće i po sebi najkomplikovanije noseće sisteme dobro i brzo pomoću kompjutera proračunati. Metodika proračuna se u principu pri tome sastoji od odgovarajućeg skupa većeg broja procedura, kojima u OOP „odgovaraju" metode.

FEM je u svakom slučaju materija koja se uči u višim semestrima u statici. U detalje te metode se zato ovde neće ulaziti. Principijelni postupak pri primeni obuhvatnih računskih operacija može se ipak objasniti, bez ulaženja u pojedinosti. Kao što se iz donje skice vidi, radni delovi su podeljeni na module (Subroutine, Functions), koji se pozivaju iz jednog glavnog modula („Master-" ili „Main-Modul"). Tok obrade (kontrolni tok) se pritom predstavlja u proceduralnim okvirima u tzv. organizacionom dijagramu (šema strukture programa). Direktno prevođenje konvencionalne šeme strukture programa u OOM doduše nije moguće, ali ipak postoje sličnosti.

Na sledećem crtežu je prikazana FORTRAN-aplikacija jednog FEM-programa sa procedurama baziranim na principu modularizacije:



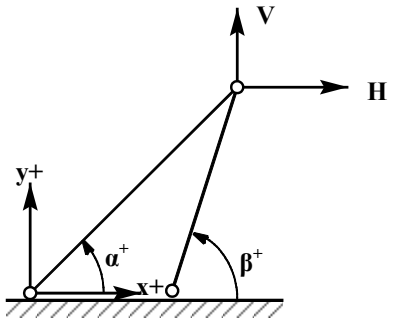
Odsečak konstrukcije (FEM-Sistem)



Strukturni plan programa

Jezgro proračuna kod FEM analize se sastoji u tome da se proračun mehaničko/statičkih veličina za koje smo zainteresovani (po pravilu su to pomeranja čvorova mreže konačnih elemenata) svede na sistem linearnih jednačina; ova ideja rešavanja biće primenjena i na ovde posmatrani jednostavni nosač, Luk_na_tri_zgloba.

Razmatrani noseći sistem: Luk_na_tri_zgloba



zadato:
Sile u čvorovimaV, H, α, β
traži se:
Sile u štapovima S1, S2

Prethodno datim zadatkom je izvršena

definicija problema

koja uvek prethodi primeni obuhvatnijeg računskog postupka.

Kao i kod objektnoorijentisanog modeliranja, pri primeni proceduralnog koncepta rešenja sprovodi se

analiza problema

u okviru koje se razmatraju različiti postupci rešavanja s obzirom na njihove prednosti i mane. Ova analiza se u datom slučaju relativno jednostavno sprovodi.

Kod ovog nosača radi se o ramovskoj konstrukciji (sistem štapova, koji se primenjuje, na primer, kod krovnih nosača, skela, mostova, kranova). Ovde moraju biti ispunjeni sledeći preduslovi, koji determinišu oblast važnosti primenjenih postupaka rešavanja problema:

1. Sile (opterećenja) moraju delovati samo u štapovima koji povezuju čvorove,

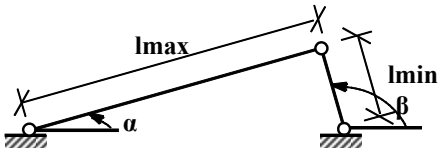
2. Čvorovi se uzimaju kao zglobovi bez trenja, tako da pri razmatranju uticaja sila pod 1. štapovi mogu preuzeti sile pritiska ili istezanja, ali ne poprečne sile niti momente savijanja.

Primedba:

U stvarnosti zglobovi nisu bez trenja; posebno, kod mnogih ramovskih konstrukcija su čvorovi izvedeni kao nitnovane ili švajsovane veze. Ocena pomoću jedne finije teorije pokazuje jasno da su razlike između jedne krute veze i veze pomoću idealnog zgloba (zglob bez trenja) zanemarljivo male.

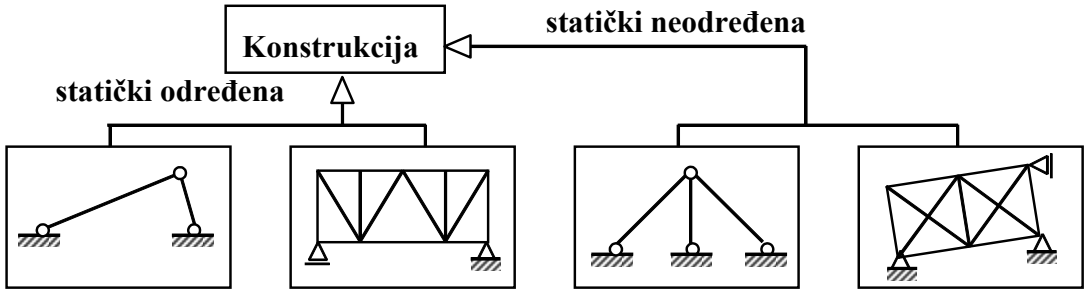
Pretpostavka:

Štapovi u ramu ne smeju biti previše kratki (na primer, $l_{\min} \geq \frac{1}{10} l_{\max}$, ili α, β birati pogodno.

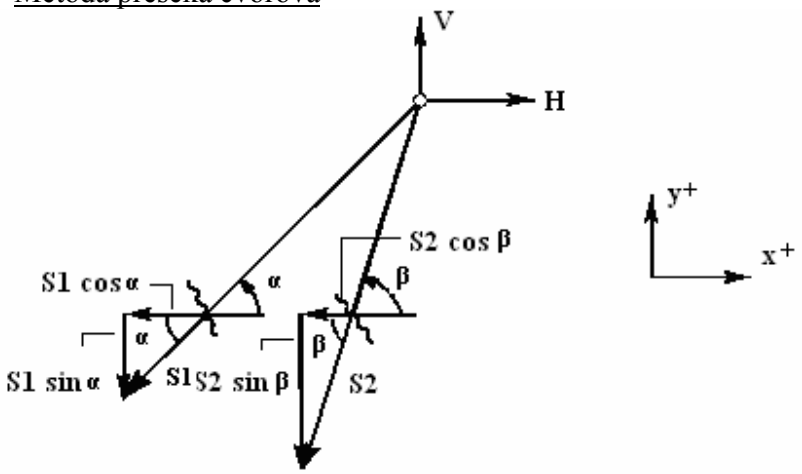


Izabrani postupak proračuna

Za dati slučaj primene „Luk_na_tri_zgloba" (kao najjednostavniji slučaj u dole datom klasnom dijagramu „Ramovska konstrukcija") najpogodnija je metoda preseka čvorova („metoda deformacija", itd.).



Metoda preseka čvorova



Da bi bio ispunjen uslov ravnoteže čvorova, mora da važi:

$$\sum K_x \overset{!}{=} 0 = -S_1 \cos\alpha - S_2 \cos\beta + H = 0$$
$$\sum K_y \overset{!}{=} 0 = -S_1 \sin\alpha - S_2 \sin\beta + V = 0.$$

Sređivanjem se dobijaju jednačine

$$S_1 \cos\alpha + S_2 \cos\beta = H$$
$$S_1 \sin\alpha + S_2 \sin\beta = V,$$

koje predstavljaju linearni sistem jednačina sa nepoznatima, silama u štapovima S_1, S_2, koeficijentima $\cos\alpha, \cos\beta, \sin\alpha, \sin\beta$ i na desnoj strani H i V.

Računarska obrada sistema linearnih jednačina (obično sa n linearnih jednačina i n nepoznatih) se tako sprovodi, da se postigne visoka opšta tačnost (princip generalizacije u softverskom inženjerstvu). Računarska obrada tako čestog problema kao što je rešavanje sistema linearnih jednačina treba da bude što je moguće efikasnija.

Sistemi jednačina se zato pri rešavanju na računaru u opštem slučaju tako sređuju da se koeficijenti nepoznatih predstavljaju u matričnoj formi; same nepoznate se predstavljaju kao vektor i pišu se sa desne strane matrice koeficijenata. U našem slučaju se dobija sledeći način pisanja:

$$m_{1,1}S_1 + m_{1,2}S_2 = R_1$$
$$m_{2,1}S_1 + m_{2,2}S_2 = R_{2,}$$

tj. u matričnoj formi

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{Bmatrix} S_1 \\ S_2 \end{Bmatrix} = \begin{Bmatrix} R_1 \\ R_2 \end{Bmatrix}$$

a potom i u sasvim opštem obliku

$$\mathbf{M} \cdot \vec{\mathbf{S}} = \vec{\mathbf{R}}$$

Prednost matrične notacije leži pre svega u tome da se pomoću primenjenih indeksa – posebno primenom brojčane petlje (for) - pojedini elementi mogu direktno adresirati, ali i u tome da se bez previše složenog pisanja mogu obrađivati (princip formalizacije).

Za matrične i vektorske strukture podataka postoje u većini programskih jezika odgovarajuće jezičke konstrukcije, tzv. nizovi i polja. To važi takođe i za JAVA, tako da ćemo ovde predstaviti nizove i polja.

Nizovi i polja

Kratak opis:

Linearni niz komponenata podataka istog tipa.

Definicija

Opšti oblik kod vektora

Tip_podataka Referenca [] = new Tip_podataka [n];

Opšti oblik kod 2-dimenzionalnih matrica

Tip_podataka Referenca [] [] = new Tip_podataka [n] [m];

Primeri

```
double matrix [][] = new double[2][2];
long fa[] = new long [10];
double r [] = new double [2];
double s [] = new double [2];
```

Semantika

Uvek se generiše Referenca (npr. matrix, fa, r, s) na neko jednodimenziono ili dvodimenziono polje datog tipa podataka (double tj. long). Operatorom „new" se povezuje neki objekat datog tipa zadate veličine (= dužina polja) u uglastim zagradama ([]) kome se može pristupiti pomoću odgovarajuće reference.

Pristup elementima polja se vrši pomoću indeksa. Vrednost indeksa pritom leži između 0 i (dužina polja-1).

Primer

Pomoću

```
Double m [][] = new double [2][2];
```

biće generisano polje brojeva dimenzije (2x2) , tipa „double", u sledećem redosledu memorisanja:

$$\begin{bmatrix} m[0][0], & m[0][1], \\ m[1][0], & m[1][1] \end{bmatrix}$$

Uvođenjem polja može se sprovesti rešavanje sistema linearnih jednačina na računaru. (Moto: Kompjuter je ljubitelj matrica, „matricofil"!). Sve uobičajene numeričke metode (npr. Gausov metod, metod Choleskog, itd.) primenjuju indeksirane veličine (polja, nizove) za formulisanje odgovarajućih algoritama.

U ovim okvirima nećemo navoditi neki opšti metod za rešavanje sistema linearnih jednačina, već ćemo primeniti Kramerova pravila kojima se lako rešavaju sistemi jednačina sa najviše dve, odnosno tri nepoznate, tj. jednačine.

Dakle, sistem jednačina

$$S_1 \cos\alpha + S_2 \cos\beta = H$$
$$S_1 \sin\alpha + S_2 \sin\beta = V,$$

pisaćemo u obliku

$$m_{00} S_1 + m_{01} S_2 = h$$
$$m_{10} S_1 + m_{11} S_2 = v,$$

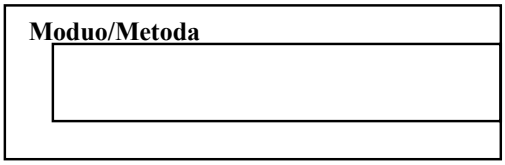
pri čemu se **m** memoriše kao matrica koeficijenata. Obe nepoznate se uvode kao skalarne veličine S_1, S_2, a isto tako i slobodni članovi na desnoj strani h i v. Po Kramerovim pravilima imaćemo

$$s_1 = \frac{\begin{vmatrix} h & m_{01} \\ v & m_{11} \end{vmatrix}}{\begin{vmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{vmatrix}} = \frac{h \cdot m_{11} - v \cdot m_{01}}{m_{00}m_{11} - m_{01}m_{10}}$$

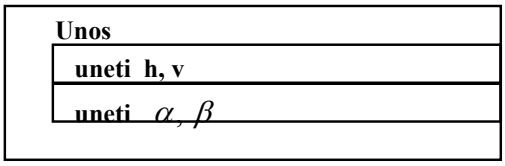
i

$$s_2 = \frac{\begin{vmatrix} m_{00} & h \\ m_{10} & v \end{vmatrix}}{\begin{vmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{vmatrix}} = \frac{v \cdot m_{00} - h \cdot m_{10}}{m_{00}m_{11} - m_{01}m_{10}}.$$

Kontrola toka za izvršenje proračuna može se sada obuhvatiti struktogramom, pri čemu ćemo posmatrati delove „Unošenje", „Proračun" i „Izlaz". Za svaki od tih delova razvija se sopstveni struktogram. Da bi se doprinelo utisku da su pojedini delovi nezavisni moduli, biće uveden novi strukturni element „moduo" pomoću koga će metode biti predstavljene.



Odatle se dobijaju sledeća tri struktograma:
Struktogram za unos



Struktogram za proračun sila u štapovima

Proračun po Krameru

//Prevodjenje u radijane(neophodno za programiranje!)

$\overline{\alpha} = \frac{\alpha}{180} \cdot \pi \qquad \overline{\beta} = \frac{\beta}{180} \cdot \pi$

//Matrica koeficijenata

$m[0][0] = \cos(\overline{\alpha})$
$m[0][1] = \cos(\overline{\beta})$
$m[1][0] = \sin(\overline{\alpha})$
$m[1][1] = \sin(\overline{\beta})$

//Proracun glavne determinante
determinanta = m [0] [0] * m [1] [1] - m [0] [1] * m [1] [0]

| determinanta | $< 10^{-4}$

da

Izlaz:
Determinanta singularna!

ne

z1Det = h * m [1] [1]
- v * m [0] [1]

z2Det = v * m [1] [1]
- h * m [1] [0]

s1 =z1Det / determinanta

s2 =z2Det / determinanta

Struktogram za izlaz

Izlaz

izdati s1, s2

Gore sprovedena jednostavna podela na samostalne module odgovarala bi pristupu koji se primenjuje pri korišćenju tradicionalnih (proceduralnih) programskih jezika. Pri prevođenju u objektnoorijentisani program sa programom JAVA su neophodne modifikacije, jer objektno orijentisanje zahteva drugi ugao gledanja. Posebno treba upamtiti da

a) Treba uvesti objekte koji se aktiviraju komunikacijama sa drugim objektima ili akcijama (events) korisnika,

b) JAVA podržava tehniku prozora i postavljanje grafičkih korisničkih interfejsa (GUI = Graphical User Interface), čime postiže interaktivno povezivanje korisnika drugačije nego kod čistih proceduralnih programskih koncepata.

Za ovde posmatrani primer primene za Luk_na_tri_zgloba najvažnije poznate veličine (podaci o strukturi) za proračun sila u štapovima, tj. opterećenje u zglobu nosača,

- horizontalna sila h,
- vertikalna sila s ,

kao i geometrijski podaci za uglove nagiba štapova,

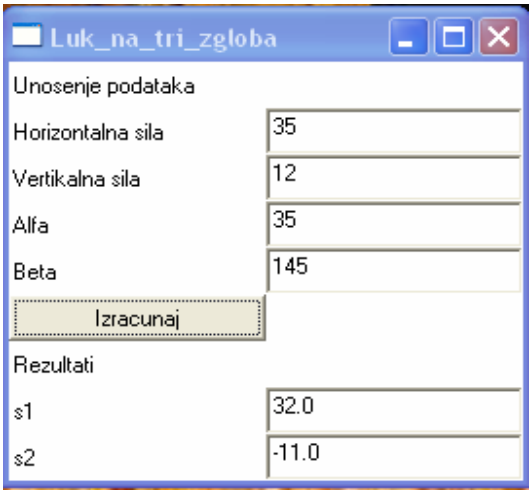
- ugao α i
- ugao β

mogu da se unesu pomoću jednog prozora. Isti prozor se može (opet radi pojednostavljenja) pored toga koristiti za iniciranje proračuna sila u štapovima. Ovde se koristi tzv. Button („Dugme"), jedna prekidačka površina. Isto tako, da bi se smanjilo programiranje, izlaz rezultata, tj.

- sila S_1 i
- sila S_2

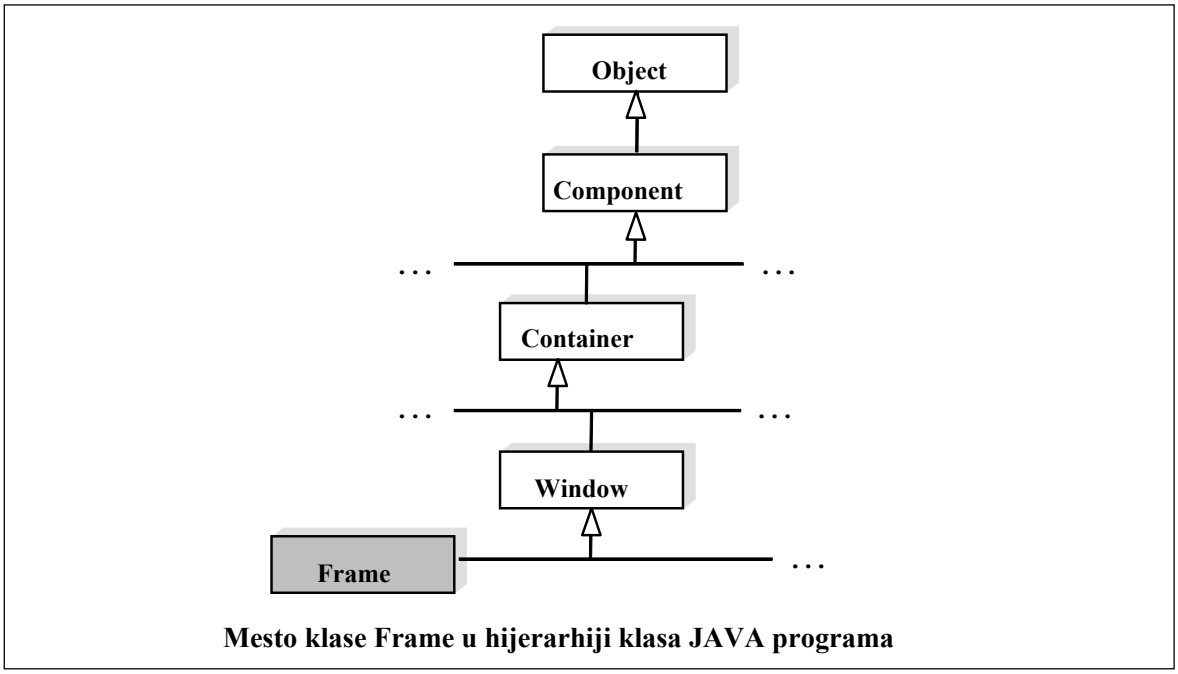
unosi se u isti prozor (Za komplikovanije slučajeve nego što je ovaj, za interakciju sa korisnikom i izlaz rezultata, koriste se posebni prozori).

Ovde prikazana „Jedno-prozorna-logika" vodi u suštini do sledećeg „Jednostavnog prozora" koji se generiše kao podklasa „Luk_na_tri_zgloba" jedne u JAVA već uvedene osnovne klase „Frame".



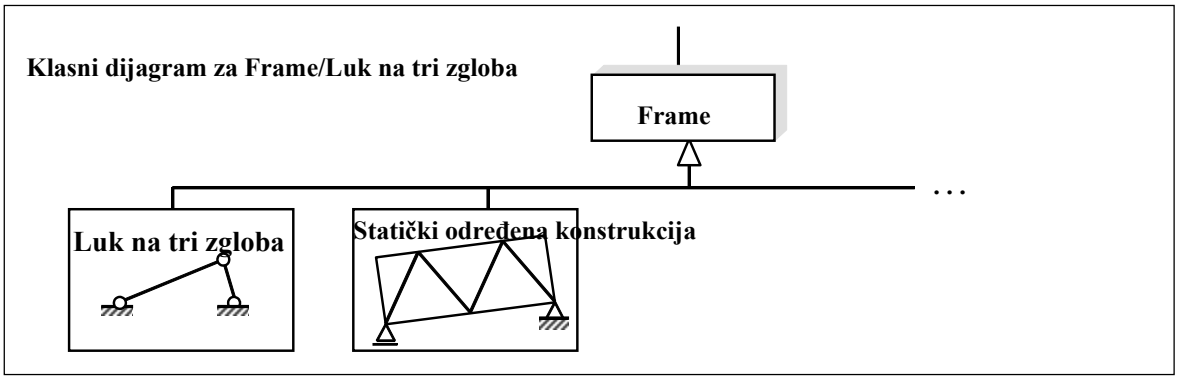
Klasa „Frame" je deo JAVA-paketa „awt" (abstract windowing toolkit), koji nudi različite klase za kreiranje grafičkih korisničkih površina – nezavisno od korisničke platforme MS-Windowsa ili drugih.

Klasom „Frame" se posebno može kreirati prozor na najvišem nivou (tzv. toplevel window), koji sadrži telo menija, kursor i ikonu (Videti 6.2.1.: paket „java.awt.*" sadrži grafičke elemente (boje, liste, menije, dijalog-prozore) i tzv. LayoutManager, koji preuzimaju uređenje komponenata). Mesto klase „Frame" u klasnoj hijerarhiji JAVA – da bi se shvatile mogućnosti programa JAVA – predstavljeno je na sledećoj skici.

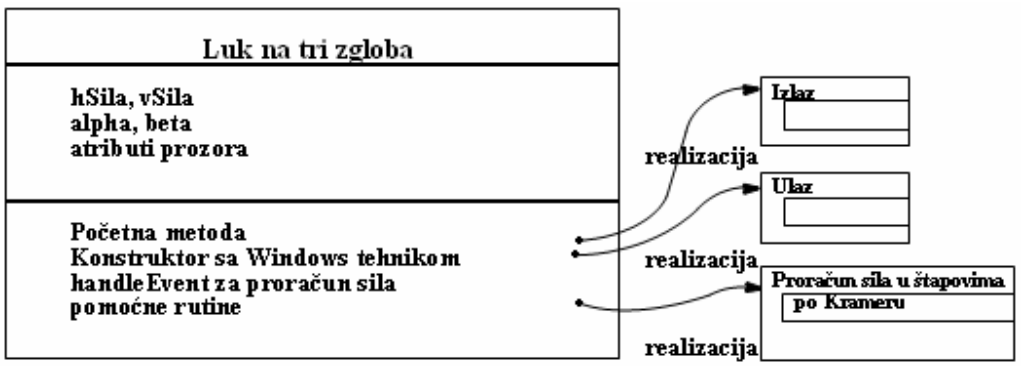


Mesto klase Frame u hijerarhiji klasa JAVA programa

Objektno-orijentisana implementacija proračuna sila u štapovima za Luk_na_tri_zgloba može se, sa gore datim objašnjenjima, najbolje na sledeći način pokazati:



Generisanje klase „Luk_na_tri_zgloba", koja je izvedena iz osnovne klase „Frame" i kojom treba da se reši problem „Luk_na_tri_zgloba", ima pritom sledeći oblik (Minimalni formalizam):



Kada se prevede klasni dijagram „Frame/Luk_na_tri_zgloba/...” sa odgovarajućim metodama u JAVA-naredbe, dobija se sledeći program:

```
// ================================================
// Program III.
import java.awt.*;
public class Luk_na_tri_zgloba extends Frame
{
        double hSila;
        double vSila;
        double alfa,beta;

        TextField hField = new TextField("");
        TextField vField = new TextField("");
        TextField aField = new TextField("");
        TextField bField = new TextField("");
        TextField s1Field = new TextField("");
        TextField s2Field = new TextField("");

        Button okButton = new Button("Izracunaj");

        public Luk_na_tri_zgloba()
        {
                super("Luk_na_tri_zgloba");

                setLayout(new GridLayout(9,2));

                add(new Label ("Unosenje podataka"));
                add(new Label (""));
                add(new Label ("Horizontalna sila"));
                add(hField);
                add(new Label ("Vertikalna sila"));
                add(vField);
                add(new Label ("Alfa"));
                add(aField);
                add(new Label ("Beta"));
                add(bField);

                add (okButton);
                add(new Label (""));
                add(new Label ("Rezultati"));
                add(new Label (""));

                add(new Label ("s1"));
                add(s1Field);
                add(new Label ("s2"));
                add(s2Field);

                pack();
                show();
        }
```

```
public boolean handleEvent( Event e )
{
	if ( e.id == Event.WINDOW_DESTROY)
	{
		System.exit(0);
	}

	else if ( e.target == okButton && e.id == Event.ACTION_EVENT)
	{
		hSila = toDouble (hField);
		vSila = toDouble (vField);
		alfa = toDouble(aField)/180*Math.PI;
		beta  = toDouble(bField)/180*Math.PI;

		double matrix[][] = new double[2][2];

		matrix[0][0] = Math.cos(alfa) ;
		matrix[0][1] = Math.cos(beta) ;
		matrix[1][0] = Math.sin(alfa) ;
		matrix[1][1] = Math.sin(beta) ;

		double determinanta = matrix[0][0]*matrix[1][1]- matrix[1][0]*matrix[0][1] ;

		if (Math.abs(determinanta) <= 0.0001 )
		{
			s1Field.setText("Determinanta");
			s2Field.setText("singularna");
		}

		else
		{
			double z1Det = hSila*matrix[1][1]-vSila*matrix[0][1] ;
			double z2Det = vSila*matrix[0][0]-hSila*matrix[1][0] ;

			double s1 = z1Det/determinanta ;
			double s2 = z2Det/determinanta ;

			s1 = roundTo(s1,0) ;
			s2 = roundTo(s2,0) ;

			s1Field.setText(toString(s1));
			s2Field.setText(toString(s2));
		}
	}

			return super.handleEvent(e);
}
```

```
        double roundTo (double d, int i)
        {
                return ((double)Math.round(d*(int)Math.pow(10,i)));
        }

        double toDouble (TextField s)
        {
                return Double.valueOf (s.getText().trim()).doubleValue();
        }

        String toString (double d)
        {
                return new Double(d).toString();
        }

        public static void main(String[] args)
        {
                new Luk_na_tri_zgloba();
        }
            }
// Kraj III programa
//  ============================================
```

6.2.7. Obrada izuzetaka

Upravo pri primeni računskih metoda pomoću struktura objašnjenih u prethodnom odeljku postaje jasno da se pritom mogu načiniti mnoge greške, bilo da se unesu pogrešni podaci koji mogu prouzrokovati besmislene operacije, ili da nastaju greške u izvršavanju kontrolnih naredbi usled, na primer, nedozvoljenih operacija (na primer, deljenje sa nulom, pristup nedozvoljenom području memorije, pogrešan pristup sistemskim resursima, itd.). Mogućnosti greške su skoro neiscrpne.

Ni objektno-orijentisanim pristupom se to stanje nije izmenilo, čak su se pojavile mnoge nove mogućnosti greške. Sa greškama pri razvoji softvera se mora živeti, a posebno se mora tretiranje nastalih grešaka ispravno sprovoditi.

Tradicionalni programski jezici FORTRAN, ALGOL, BASIC i C pri tome ne nude naročito prefinjene metode da bi se greške u toku izvršavanja programa razmatrale ili sprečile. Svako ko piše softver u jednom od navedenih programa mora da se sam stara za to da nastalu grešku raspozna i ispravno je tretira. S obzirom na nezamislivo veliki broj mogućnosti nastajanja grešaka, ovo je često jedan beznadežan posao. Pri tom nastaje velika šteta, pošto greške ne samo da ne moraju da budu otkrivene, već se ne može ni odrediti kako program po nastanku greške treba da reaguje.

Nasuprot tome, u JAVA, kao uostalom i u C++ je moguće tretirati greške tokom izvršenja programa. Predviđene su prefinjene metode za sistematično i logički struktuirano tretiranje grešaka, čime softver postaje sigurniji i robusniji nego ranije.

Greške se pritom u JAVA označavaju kao izuzeci (exceptions), pri čemu se svako nastajanje incidenta koji prekida normalno odvijanje programa ocenjuje kao izuzetak. Kada nastupi greška, ona se mora, pre nego što se nastavi normalno izvršenje programa, ispraviti.

Primer: izračunavanje korena

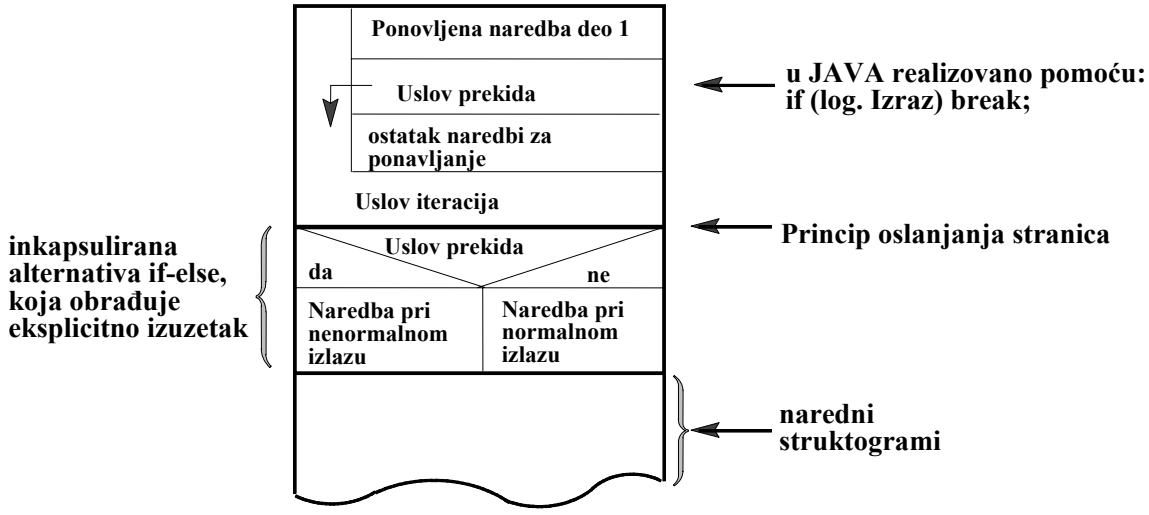
$\sqrt{x^2 - y^2}$, sa $x = 4,\ y = 8.$

Potkorena veličina će biti u tom slučaju negativna, tako da sa

$\sqrt{4^2 - 8^2} = 4\sqrt{-3},$ dobijamo imaginarni broj kao rezultat, što je izuzetak.

Interesantno je da strukturno programiranje sa struktogramima, kao što je u prethodnom odeljku za opis računskih operacija i tokova obrada podataka pomoću metoda objašnjeno, poseduje i eksplicitna pravila za obradu izuzetaka. Kod iteracija sa unutrašnjim prekidom, koji iz određenih razloga prekida obradu naredbi koje se ponavljaju, prekid se isto tako ocenjuje kao izuzetak i direktno u nastavku iteracije eksplicitno navodi kao nezavisna alternativa (predlog poboljšanja pronalazača Nassi-Schneiderman-Struktograma).

Primer: do-while-struktogram sa internim prekidom



Ovo je jedan dobar primer za to, kako se tretiranje izuzetaka može posmatrati kao značajan konstruktivni elemenat razvoja softvera.

Semantičke mogućnosti JAVA programa su značajno bogatije i konfornije od gore datog tretmana koristeći struktograme. Da bi se razumeo osnovni princip, dajemo realizaciju principijelnog postupka u programu JAVA.

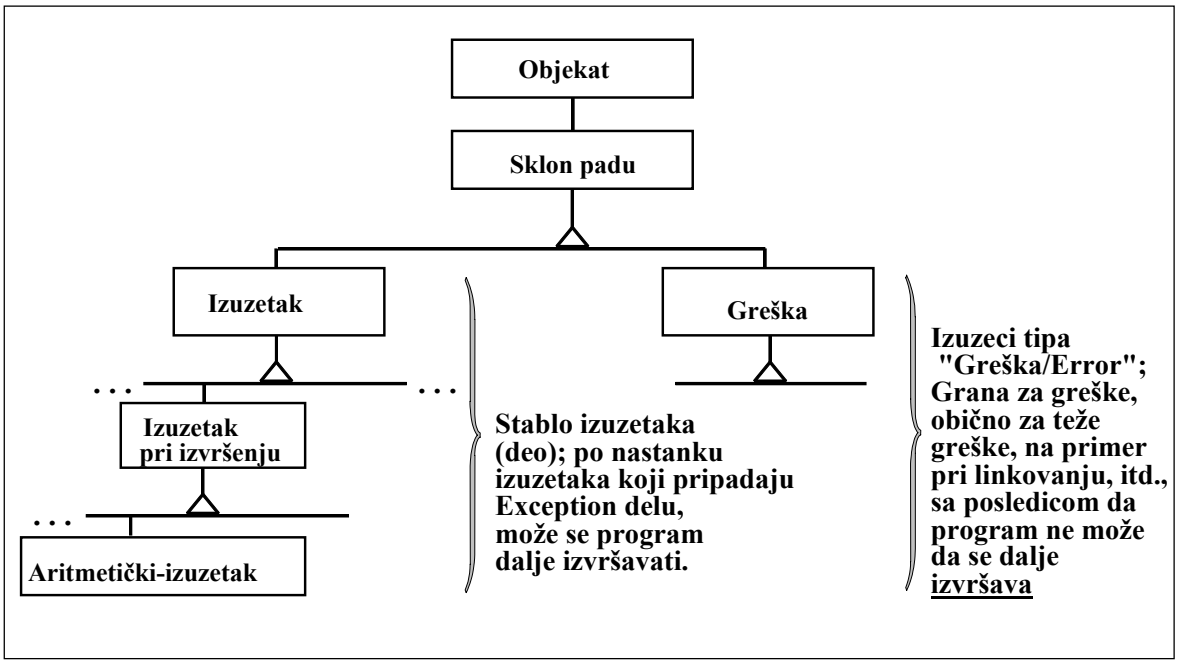
Principijelna procedura

Kada se u JAVA programu desi izuzetak, na primer, izračunavanje korena sa negativnom potkorenom veličinom, metoda koja sadrži tu naredbu generisaće tzv. instancu za izuzetak

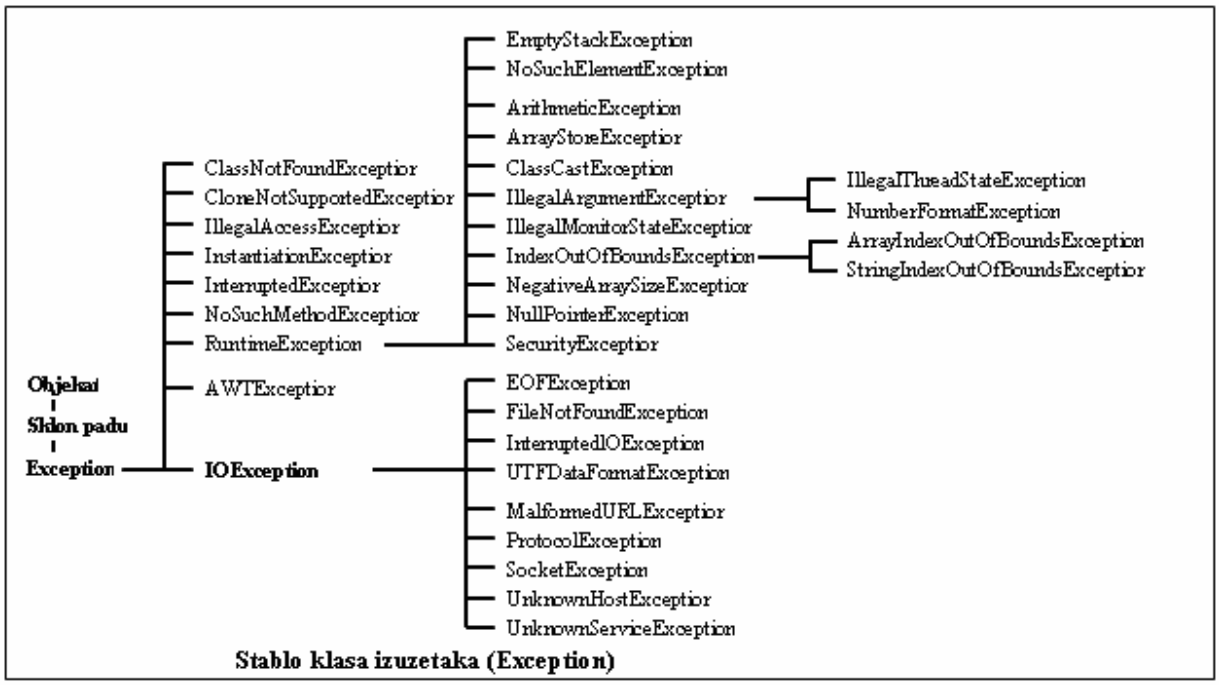
(exception-instance), a ova će instancu proslediti run-time okolini (jedinica za nadgledanje programa koji se izvršavaju). Generisana instanca za izuzetak sadrži niz važnih informacija kao što su vreme nastanka izuzetka, status programa u tome trenutku, tip izuzetka, itd. Sve informacije će biti od strane run-time sistema vrednovane, da bi se po proceni izabrale odgovarajuće mere za tretman greške-izuzetka. U JAVA terminoligiji se instanciranje izuzetka (tj. generisanje instance za izuzetak) i njeno prenošenje run-time sistemu označava kao „Izbacivanje", tj. „Throwing an exception" i očekivanje reakcije; u stvari se sistemu za kontrolu procesa dostavlja informacija o nastalom problemu i očekuje reakcija sistema.

Reakcija sistema zavisi od dostavljenih informacija koje se nalaze u jednom automatski aktiviranom staku (zove se trace stack); on sadrži – po redosledu pojavljivanja – sve incidente koji dovode do izuzetaka. Počevši od najvišeg ulaza, run-time sistem bira način tretiranja izuzetka (Exception Handler). Izbor zavisi od toga da li je tip izuzetka isti sa tipom „Exception Handlera". Postupak nalaženja i izbora označava se u JAVA-svetu kao „hvatanje", tj. „to catch an exception". Slučaj da neki izuzetak ne može da pronađe odgovarajući tretman se ne može dogoditi, pošto je unutrašnja organizacija tretmana izuzetaka tako postavljena, da uvek postoji „Exception Handler" (ako se dogodi, vodi do prekida tekućeg programa, što bi trebalo izbegavati).

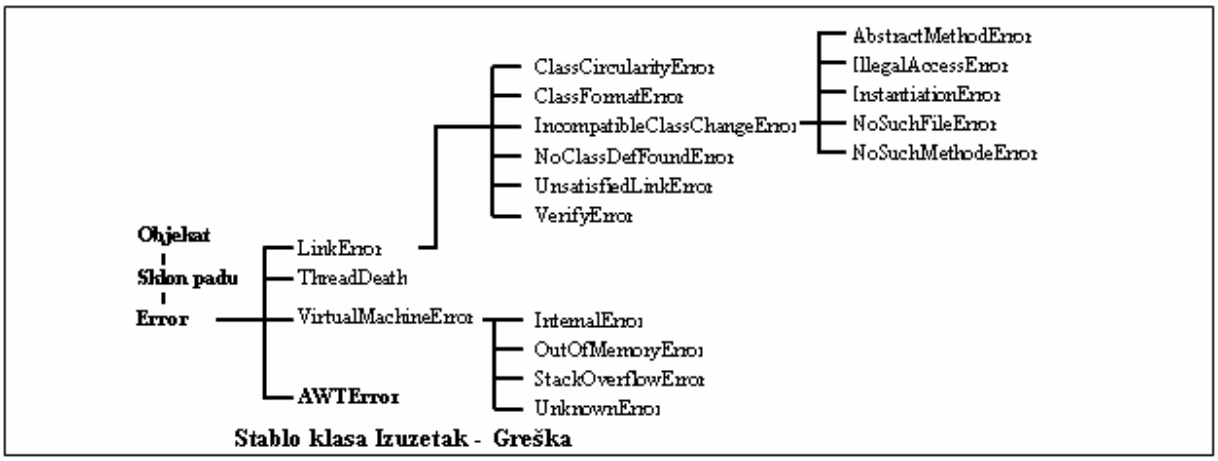
Sve instance koje se „Izbacuju" su pritom – sledeći objektno-orijentisanu logiku – instance jedne klase, ili klase pod imenom „Exception", ili podklase izvedene iz ove klase. Klasa „Exception" je sa svoje strane opet jedna podklasa klase „Throwable", koja se nalazi na najvišem mestu mehanizma za tretiranje izuzetaka. Pregled hijerarhije klasa za tretman izuzetaka u JAVA programu daje sledeći grafik, iako je dat samo jedan deo stabla klase „Throwable" („Izbacivanje").



Detaljni prikaz pojedinog izuzetka i tipa greške se može videti iz sledećeg prikaza.



Stablo klasa izuzetaka (Exception)



Stablo klasa Izuzetak - Greška

Kao što se vidi, radi se o vrlo kompleksnom konceptu klasa, koji je pritom i tako organizovan da ima važnost za proizvoljne ciljne platforme. Stoga je jasno da jedan tako obuhvatni mehanizam za tretiranje grešaka i izuzetaka podleže strogoj sintaksi, koja će biti nadalje obrađena.

Sintaksno uređenje tretiranja izuzetaka

Svaki niz naredbi koji može voditi nekom izuzetku (exception), mora biti uključen u takozvani „try-block", koji počinje JAVA naredbom „try". Ovo se sprovodi sledećom formalnom konstrukcijom:

```
try {
   JAVA kod;
}
```

Preciznije:

```
try {
        // grupa naredbi saglasno strukturnom dijagramu sa jednom ili vise naredbi koje
        // mogu da ukazu na greske i stampaju ih pomocu naredbe „throw”.
}
```

Bloku „try-block” mora da sledi, bez prekida drugim JAVA kodom, najmanje jedan tzv. „catch-block”, u kome se izuzeci „hvataju” naredbom „catch”. Pritom se koristi sledeći šablon:

```
try
{
        // JAVA kod try-bloka
}
catch (IzuzetakKlasa1 Instanca1)
{
        // tretman Izuzetka1
        // pristup Instanci1
}
catch (IzuzetakKlasa2 Instanca2 )
{
        // tretman Izuzetka2
        // pristup Instanci2
}

// itd. npr. naredne Catch-konstrukcije.
```

Primer realizacije naredbe „catch”

```
catch (ArithmeticException W) {
        System.out.println („Pronadjeni izuzetak= ” + W.getMessage());
}
```

Semantika:

Naredbe koje se nalaze unutar „catch”-bloka biće izvršene samo onda kada se dogode izuzeci koji korespondiraju sa izuzecima navedenim u bloku. Što se viša klasa izuzetaka pritom izabere, to se više izuzetaka može tretirati. Ali, opšte deklaracije imaju malo smisla, jer tada se može reagovati na izuzetke za koje tretman uopšte nije bio uziman u obzir. Zato je potrebno da „IzuzetakKlasa” bude prilagođena konkretnom slučaju.

Iza „try-catch” – kombinacije može da opcionalno sledi jedan „finally-block”. Ukoliko je „finally-block” predviđen, on će biti u svakom slučaju izvršen i kada nije nastupio izuzetak. „finally-block” se po pravilu upotrebljava za to da bi se oslobodili resursi koji za rad više nisu potrebni. S time imamo ukupno sledeću JAVA-strukturu za obradu izuzetaka:

```
try
{
        // Programski kod koji moze, ali ne mora, voditi izuzecima
}
```

```
catch (IzuzetakKlasaN InstancaN)
{
        // tretman izuzetaka prema redosledu, za N=1,2,3,
}
finally (IzuzetakKlasa2 Instanca2 )
{
        // naredbe koje se u svakom slučaju izvrsavaju
}
```

Navedenim oblikom tretiranja izuzetaka posebno se povezuju sledeće prednosti:

- normalni programski kod se može osloboditi programskih delova koji se bave obradom grešaka i time program čine nepreglednim (bolja preglednost, ponovna upotrebljivost i portabilnost),

- obrada grešaka može biti sistematski i „inteligentno" struktuirana (ušteda rada, ponovna upotreba u sličnim slučajevima).

Obrada izuzetaka pomoću strukture „try-catch-finally" mora se aktivirati metodama koje mogu generisati poseban slučaj izuzetka. To znači da kod tih metoda treba ostvariti odgovarajuće povezivanje sa regulativom „try-catch-finally". Cilj ovih metoda je da se pomoću dodeljenih ključnih reči realizuje „izvršni mehanizam" za izbacivanje („Hinwerfen", „throw") izuzetaka. Pri tome se posebno primenjuju:

- ključna reč „throws", koja se preuzima u zaglavlje metode (tj. u signaturu), da bi idicirala koje izuzetke može metoda da generiše.

  Primer:

  ```
  public void imemetode throws IzuzetakKlasa
  {
          …;
  }
  ```

- ključna reč „throw", kojom se inicira obrada izuzetaka i uvodi jedna throw naredba. Iza ključne reči „throw" mora biti navedena jedna instanca, tj. objekat klase „Throwable" ili njene podklase.

Primer:

```
throw new ArithmeticException(„Imaginarni broj");
```

U ovom slučaju se operatorom „new" generiše istovremeno instanca klase „ArithmeticException" kao podklasa klase „Exception". Prenos jednog teksta za tretman izuzetaka („Imaginarni broj") je moguć stoga na tako jednostavan način zato što klasa „Throwable" između ostalog sadrži i strukturu tipa niz za memorisanje poruke o greški, kao i metodu „getMessage" za izlaz teksta. Inicijalizacija niza znakova se pritom odvija pomoću konstruktora, koja se pri generisanju nove instance sa operatorom „new" automatski poziva.

Naredbama „throws/throw" može se formulisati i metoda za izračunavanje korena, koja reguliše slučaj negativne potkorene veličine.

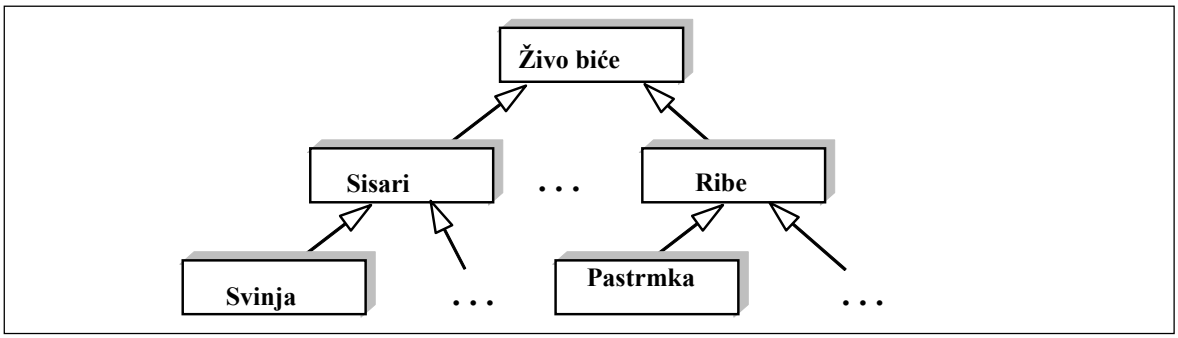
Primer: Izračunavanje korena

```
// ============================================
// Program IV.
public static double koren (double d) Throws ArithmeticException
{
        if ( d < 0 )
                throw new ArithmeticException ("Imaginarni broj");
        else
                return Math.sqrt(d);
}
```

6.2.8. Višestruko nasleđivanje / Modeliranje interfejsa

Do sada predstavljeni odnosi između klasa dozvoljavali su samo kreiranje jednostavnijih hijerarskijskih mreža (videti 6.2.4.). Kod ovih odnosa se govori o jednostrukom nasleđivanju, kod koga – kao što je upravo rečeno – svaka klasa ima najviše jednu nadklasu (superklasu).

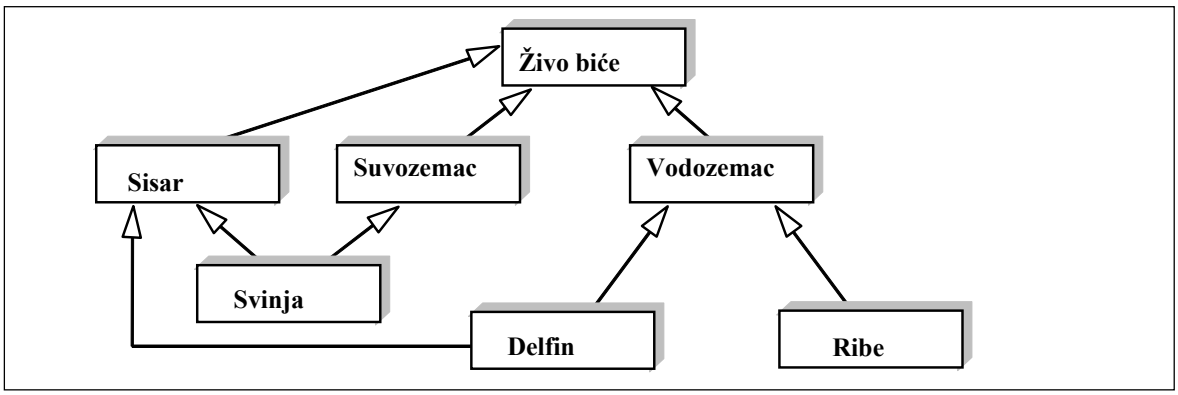
Primer: Klasifikacija (u biologiji)



Kod jednostrukog nasleđivanja svaka klasa raspoznaje svoje veze automatski, tako da se korisnik ne mora zamarati uređivanjem. Može se lako predpostaviti da samo jednostruko nasleđivanje nije dovoljno da bi se opisala realnost. Zato postoje u praksi mnogo različitih mreža odnosa. Po složenosti prvi viši stepen složenosti je heterarhija, čije modeliranje treba da bude obrađeno u ovom odeljku. Na opšte, tj. još kompleksnije mreže odnosa, preći će se u sledećim odeljcima.

Heterarhije se pojavljuju kada jedna klasa ima dve ili više nadklasa (superklasa).

Primer: produžena klasifikaciona šema (iz biologije)



Klase „svinja" i „delfin" imaju obe dve nadklase, tako da klasa „svinja" nasleđuje osobine sisara i suvozemaca. Na taj se govori o višestrukom nasleđivanju, kada se sa više od jedne klase prenose osobine na donje klase.

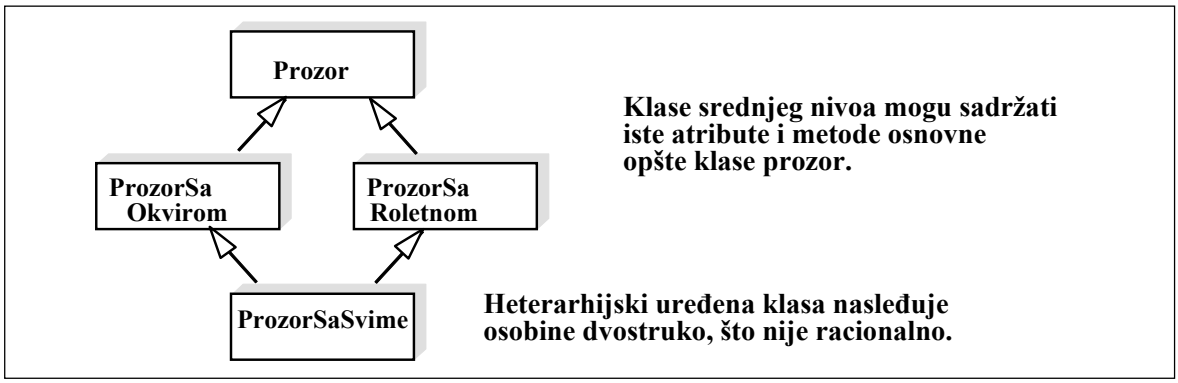
Višestruko nasleđivanje nije podržano u programu JAVA i to iz dobrih razloga. Višestruko nasleđivanje može biti vrlo problematično i voditi situacijama, koje, ukoliko nisu praćene odgovarajućim protivmerama, mogu uneti kontradiktornosti i komplikacije. Zbog tih teškoća, koje će biti bliže objašnjene, odlučeno je da se višestruko nasleđivanje ne implementira u JAVA (u C++ je to nasuprot moguće). Međutim, konceptom interfejsa/priključaka poseduje JAVA jake mehanizme, koji – po mišljenju mnogih stručnjaka - realizuju mreže relacija uz redukciju grešaka, na način sličan višestrukom nasleđivanju. Konstruktorima JAVA je u dobroj meri uspelo da prednosti jednostrukog nasleđivanja (jasna i jednostavna struktura) povežu sa prednostima heterarhije. Tako se u JAVA mogu osobine heterarhije iskoristiti da se generišu ravne i ne – kao što je kod hijerarhija uobičajeno – duboke mreže relacija.

Problemi kod višestrukog nasleđivanja

1. problem: generisanje nepotrebne redudance

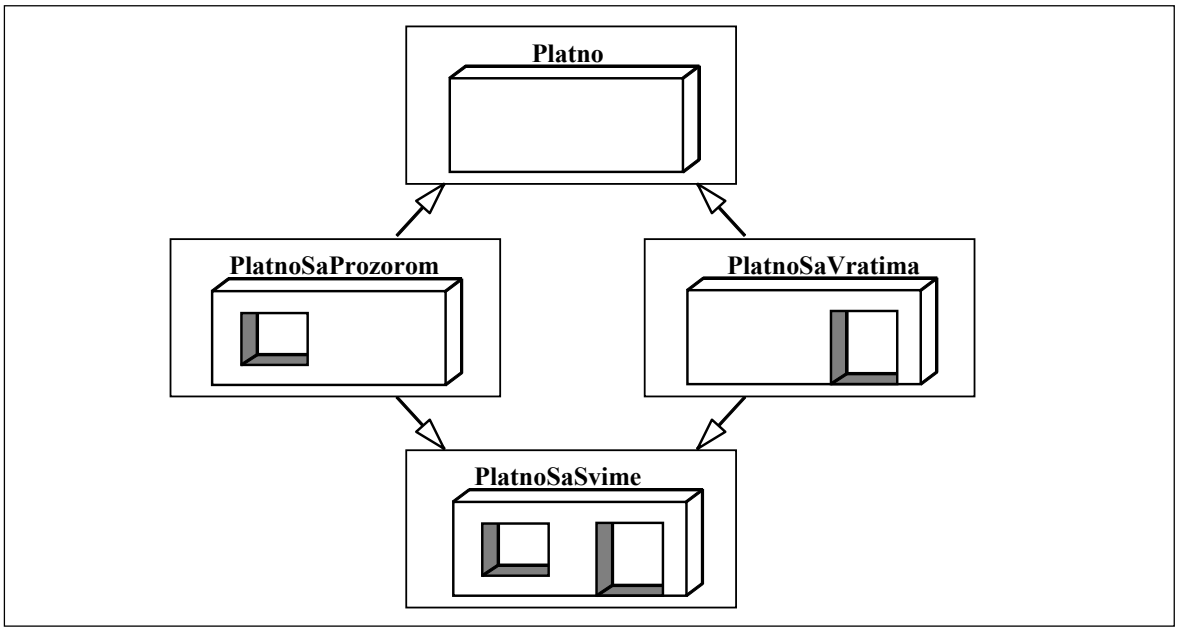
Ukoliko neka klasa ima dve nadklase, može se desiti, da se iz obe nadklase iste informacije (atributi, tj. metode) prenose na zajedničku podklasu. Ovaj slučaj može nastupiti tada kada su obe nadklase izvedene iz jedne osnovne klase.

Primer: grafička nadklasa



Za rešavanje tog problema koristi se klasa Prozor interfejs. Pomoću interfejs mehanizma se izbegava da klasa Prozor poseduje druge osnovne elemente kao stalne atribute i signature metoda, koje preko klasa ProzorSaOkvirom i ProzorSaRoletnom može dalje preneti. Time se pomoću interejsa omogućava da se nasleđeni atributi i metodi samo jedanput sadrže u ukupnoj heterarhiji, u ovom slučaju u klasi ProzorSaSvim. Time se ne sprovodi nikakvo nasleđivanje suvišnog koda. U objektno orijentisanje uvedeni princip tajnosti je posledično konsekventno sadržan (kod višestrukog nasleđivanja u smislu C++ uvaj princip je isključen).

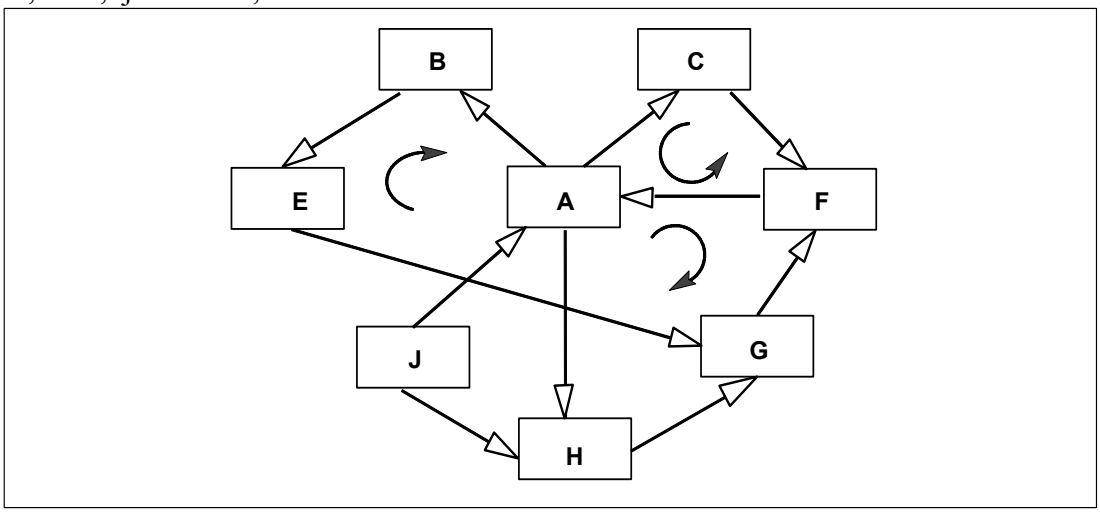
Primer: projektovanje nosećih konstrukcija



2. problem: generisanje cirkularnih odnosa (petlji)

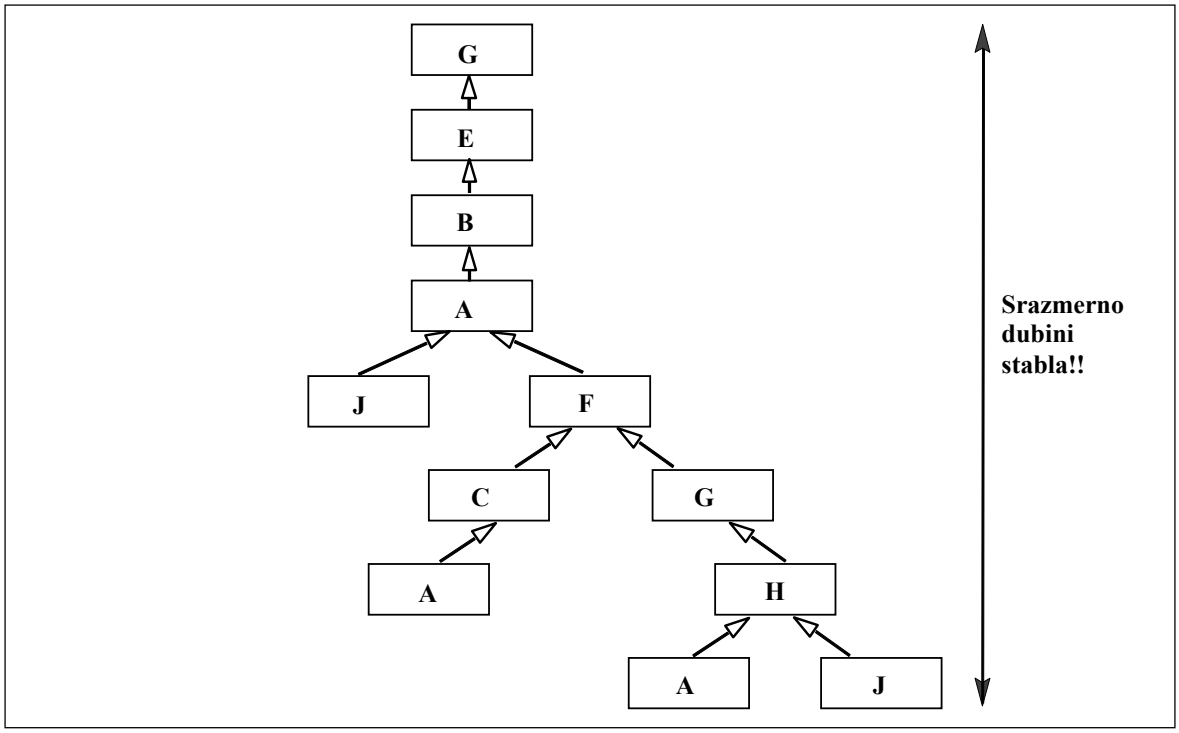
Uvođenjem klasa sa više nadklasa mogu nastati vrlo komplikovane, ne više bezuslovno jednoznačne povratne veze. Tada se govori o nasleđivanju unazad.

U sledećem primeru nasleđuje klasa E, na primer, direktno od klase G, ali indirektno i od klasa H,A i F, tj. od B i C, itd.



Situacije ove vrste vode do konfuzije i nepreglednih situacija. U C++ se zato moraju isključivati mogući pogrešni putevi i izvori grešaka pomoću komplikovanih dodatnih mehanizama.

Prednost heterarhije da omogućava ravno ugnježdavanje, biće objašnjena pomoću petlji povećane složenosti, koje su na prethodnom grafiku obeležene kružnom strelicom. U slučajevima te vrste je svrsishodno heterarhiju rešiti prevođenjem u hijerarhiju. To je moguće uvek, ali kao što će se iz sledećeg videti po cenu dubljeg ugnježdavanja. Pored toga, moraju se uzeti u obzir i redundance.



Mehanizam interfejsa

Umesto da zaobiđe goreopisane dileme teškoća pri višestrukom nasleđivanju, JAVA nudi takozvani „Interfejs-mehanizam". Ovaj daje jedan dobro osmišljeni, fleksibilno upotrebljivi koncept interejsa, koji je postavljen slično kao i koncept klasa. On superponira tj. preklapa obrađene dijagrame klasa i tako može da bude posmatran kao drugi struktuirani elemenat. Kombinacija između ovih omogućava u značajnoj meri izgradnju heterarhijskih mreža odnosa, bez odricanja prednosti hijerarhije (uniformno, jednostavno struktuirano stablo klasa).

Za sintaksnu formulaciju stoje programu JAVA na raspolaganju naredbe „interface", kao i „implements"; istovremeno može da se upotrebi, radi modeliranja jednostavnih struktura nasleđivanja unutar interfejsa, već uvedena naredba „extends" (videti odeljak 6.2.5.).

Opšti oblik (Deklaracija)

```
interface ImeInterface < extends Basisinterface > {
 Definicija konstanti;
 Deklaracija metoda;
 }
```

Opšta predstava interfejsa pokazuje značajnu sličnost sa konceptom klasa. Posebno se moraju uzimati u obzir sledeća pravila i konvencije:

- Interfejs, uveden pomoću naredbe „interface", može biti deklarisan samo kao „public" i kao „abstract". Kako se svaki interfejs bez izuzetka definiše kao „public", specifikacija „public" može izostati. Međutim, radi jasnoće će to biti stavljeno. Dakle, pišemo uvek:

  public interface...

- Interfejs može biti po potrebi izveden iz neke nadklase. Koliko je dosada o klasama poznato, izvođenje se sprovodi pomoću naredbe „extends". U skladu sa jednostavnim principom nasleđivanja, biće pritom svi atributi i sve apstraktne metode od osnovnog interejsa (Basisinterface) nasleđeni od strane iz njega izvedenih interfejsa, pri čemu za atribute treba obratiti pažnju na sledeća pravila:

- svi atributi (komponente podataka) jednog interfejsa su implicitno „static" i uz to „final". Posledice „static" i „final" su sledeće:

a) pomoću „static" se deklariše atribut klase koji ima tačno jedno memorijsko mesto za eventualno kreirane instance i ima sa interfejsom povezane klase; to znači da sve instance klase imaju jednom utvrđenu vrednost.

b) pomoću „final" se označava da se atribut klase instalira sa jednom vrednošću, koja se u docnijem izvršenju programa ne sme više menjati.

Pomoću „static" i „final" se dakle deklarišu prave konstante. Stoga ima smisla preuzeti atribute u interejse samo onda kada ovi predstavljaju prave konstante. To je na primer slučaj kod korisničke površine, gde boja pozadine jednog prozora ili širina okvira mogu da se uvedu kao prave konstante, ili u slučaju betonskoarmiranog rama, gde bi boja materijala (približno „kameno sivo") trebalo da bude konstantna.

- Metode – koje su u suštini tačka oko koje se okreću i za koju su vezani interfejsi – su uvek apstraktne metode. Saglasno tome, za njih se ne daje telo metode; na njegovom mestu stoje samo tačka i zapeta (semicolon, „ ; "). Tako interfejsi predstavljaju samo jedan šablon za klase; tačne aktivnosti metoda se regulišu u klasama na koje se odnose.

Primer:

```
public interface InterA extends Basisinterface
{
        //  konstanta (uvek samo jednom raspoloziv atribut)
        static final int a1 = 1;
        static final int a2 = 2;
        // apstraktna metoda
        public void imeMetode (int param);
}
```

- Konkretno ugrađivanje interfejsa u neku određenu klasu vrši se naredbom „implements".

Primer:

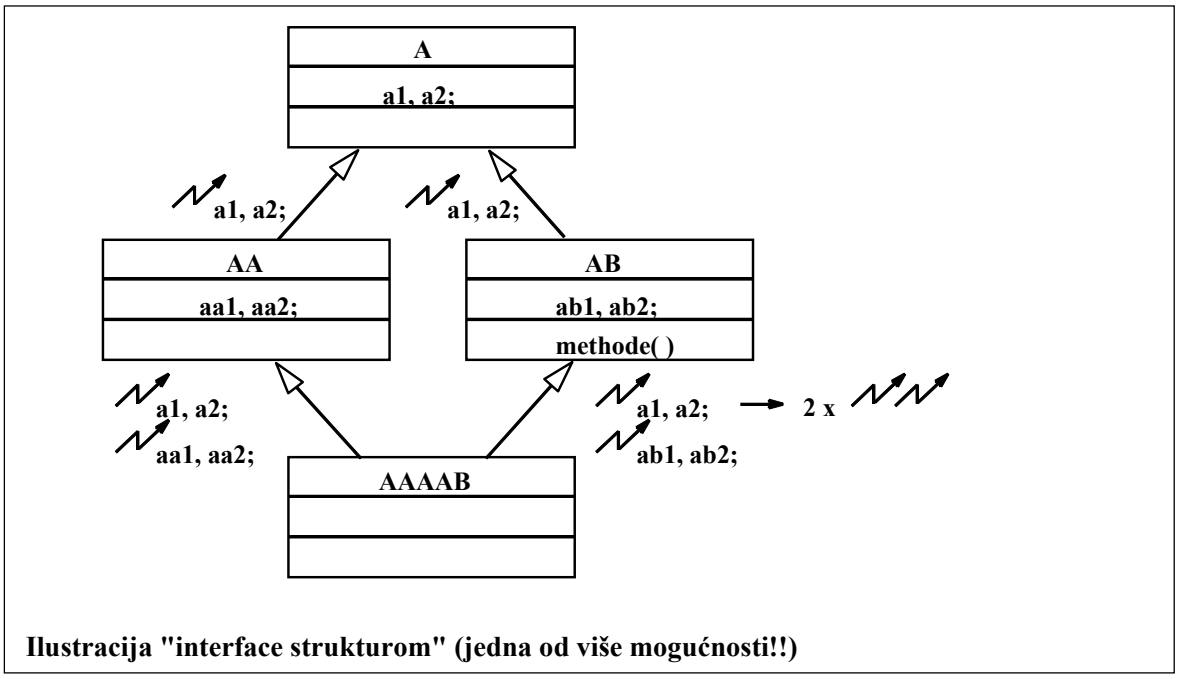
```
Class A extends BasisKlasa implements InterA
{
        // Atributi (ukoliko nisu automatski raspolozivi)
        // ...
        // Metode
        // ...
        // Primena apstraktne metode interfejsa InterA
        public void imeMetode (int param)
        {
          //  Uputstva za konkretizaciju apstraktnih metoda
        }
}
```

Rezime:

Interfejsi su međusobno usaglašavanje konstanti i apstraktnih metoda, čije tačne funkcije još nisu poznate. Saglasno tome, interfejsi se samo deklarišu, a ne definišu.

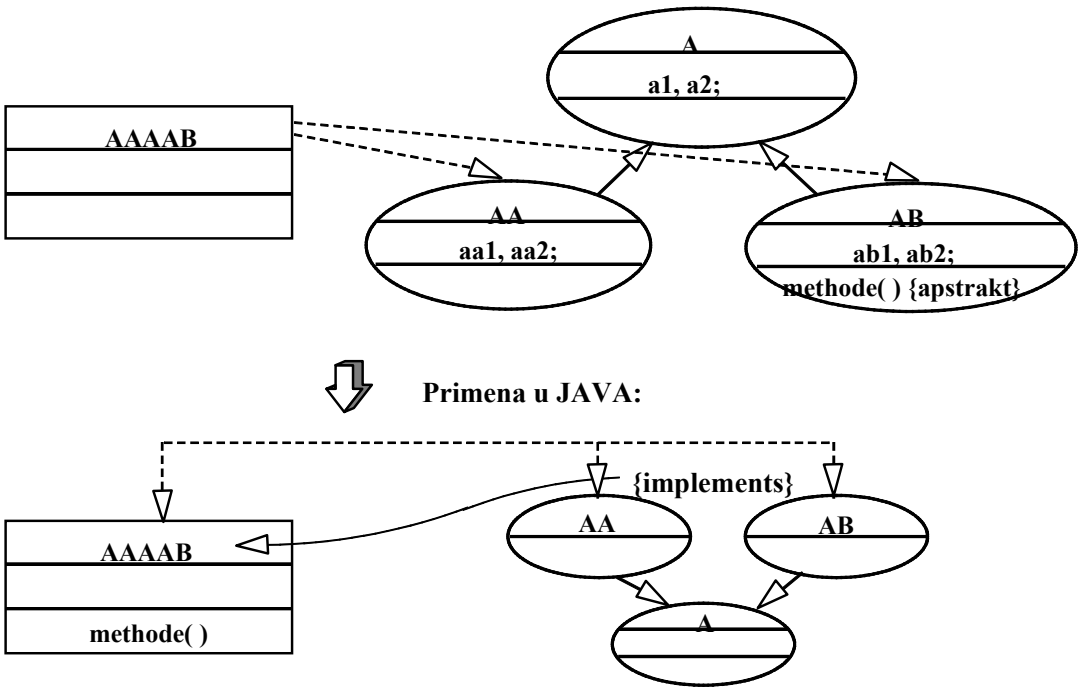
Ukoliko neka klasa primenjuje interfejs, tada je za tu klasu utvrđeno kako se njenim metodama može spolja pristupiti. Dakle, interejsi definišu šablon za ponašanje klasa i služe kao osnova za komunikaciju između međusobno komunicirajućih objekata.

Zaključni primer: Heterarhija „odslikana" pomoću interfejsa
(uporediti sa primerom: heterarhija ploča)



Ilustracija "interface strukturom" (jedna od više mogućnosti!!)

Dijagram klasa plus struktura interfejsa:



Uputstvo!
Pri zadavanju imena paziti da klase i interfejsi pripadaju istoj vrsti imena; odgovarajuća imena se moraju, saglasno tome, razlikovati.

JAVA - program

```
public interface A
        {
                static final int a1 = 1;
                static final int a2 = 2;
                ...
        }

        public interface AA extends A
        {
                static final int aa1 = 3;
                static final int aa2 = 4;
                ...
        }
        public interface AB extends A
        {
                static final int ab1 = 5;
                static final int ab2 = 6;
                ...
                public abstract void methode ();
        }
```

```
        class AAAAB implements AA, AB
        {
                // Atributi
                // Metode
                public methode ()
                }
        ...
}
// Kraj klase AAAAB.
```

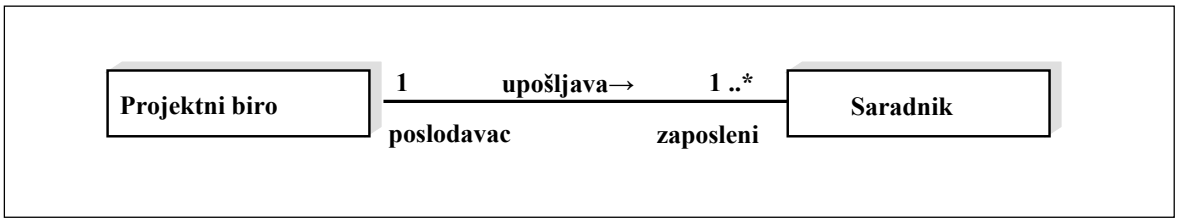
6.2.9. Modeliranje opštih relacionih mreža (asocijacije)

Problemi koji se javljaju u praksi u mnogim slučajevima ne mogu se izričito, a često i uopšte, opisati pomoću prethodno diskutovanih odnosa nasleđivanja. U tom slučaju se moraju odnosi eksplicitno modelirati u dijagramu klasa i tada adekvatno implementirati. Opšta veza između dve klase biće označena kao asocijacija, kojom se takođe mogu zatvoriti i veze između angažovanih objekata odgovarajućih klasa (objekat/veze između objekata se često pritom označavaju kao „link"). Posmatrano iz aspekta objektne orijentacije linkovi su „objekti" asocijacija.

Označavanje (notacija):

U najjednostavnijem slučaju predstavlja se asocijacija kao linija između dve klase. Ali, mogu nastupiti i slučajevi, kod kojih asocijacije sadrže sveobuhvatne detalje. To znači da asocijacije – pored klasa i objekata – prenose značajne, kognitivne međusobne veze u opisu problema. Pojedinostima njihovog modeliranja mora se stoga posvetiti najveća pažnja.

Primer 1 (iz radne prakse):

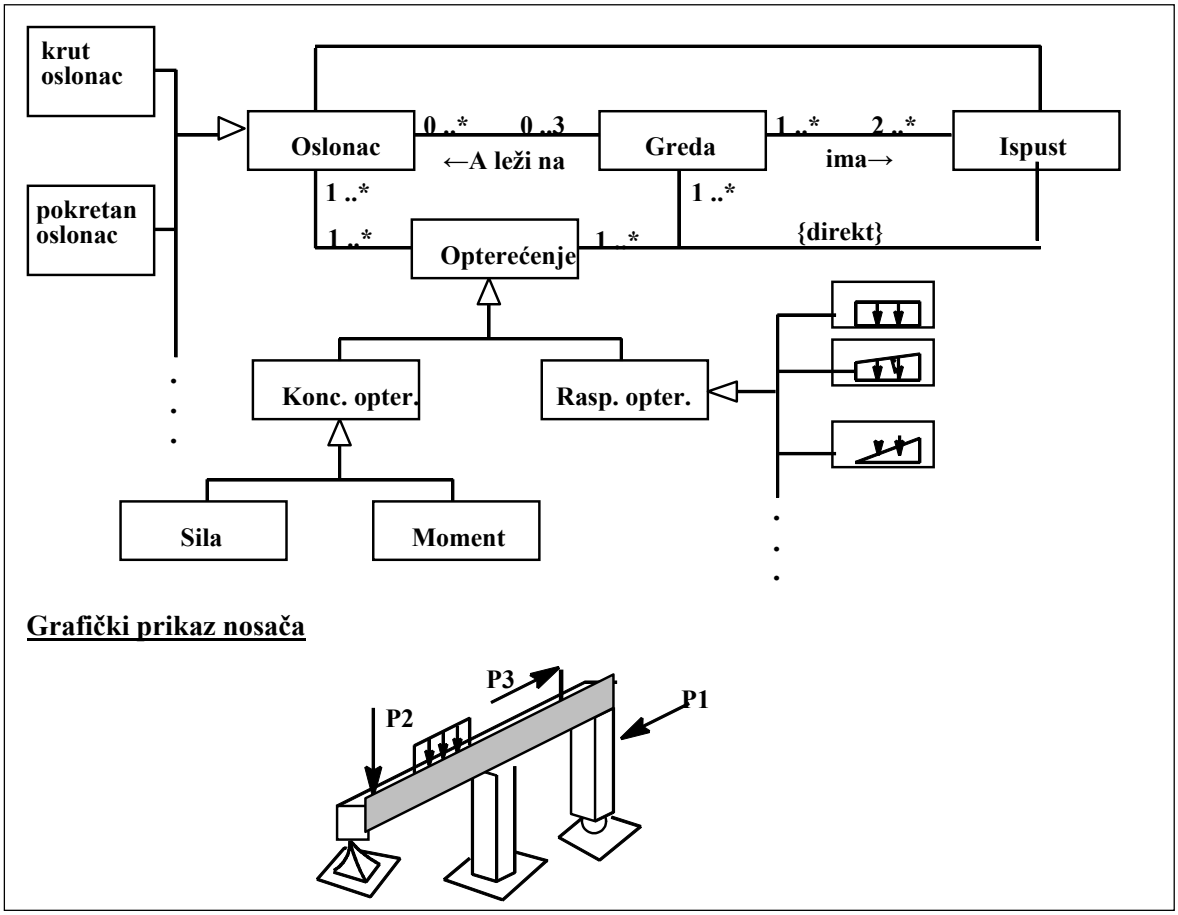


Predstavljena suština:

Jedan „projektni biro" u ulozi „poslodavca" zapošljava više „saradnika" u ulozi „uposlenika". Ovaj verbalni opis strukture odnosa između obe klase „projektni biro" i „saradnik" može se predstaviti jednom asocijacijom (linijom). Pri tome asocijacija sadrži predikat („upošljava") kao naziv odnosa sa naznakom pravca dejstva (→) kao i naznakama broja (podaci o multiplicitetu), da bi se tačno definisalo koliko objekata na jednoj strani je povezano sa koliko objekata na drugoj strani. U datom slučaju treba najmanje jedan objekat (1 ..*) klase „projektni biro" sa jednim, ali eventualno i sa više „saradničkih objekata" biti poveziv; oblast (domen) se označava pomoću „..", dok se specifikacija „više" zadaje pomoću „*", kao tzv. „džokera".

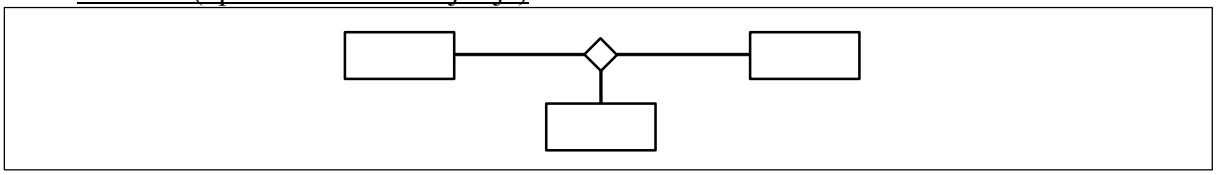
Razmatrani slučaj razjašnjava da asocijacije sadrže kako semantiku, tako i strukturu objektnih veza. Ovakvim predstavljanjem asocijacija se mogu – kao što naredni primer pokazuje – odslikati komplikovane međuzavisnosti.

Primer 2 (projektovanje nosećih konstrukcija)

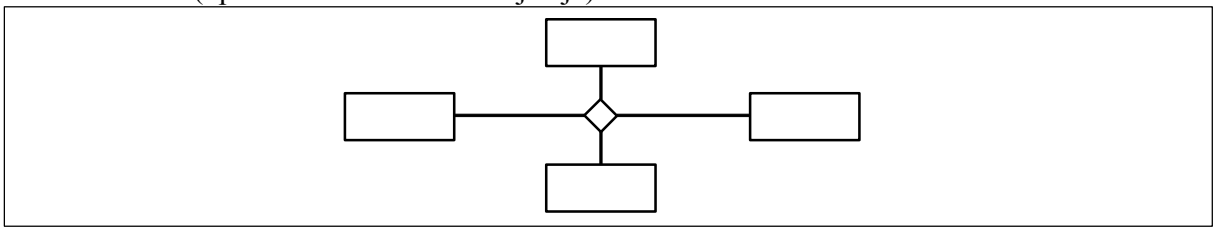


Moguće su takođe višečlane asocijacije, npr. tročlane (ternarne), četvoročlane (kvaternalne), ili čak n-to člane asocijacije.

Primer 3 (opšte ternarne asocijacije)

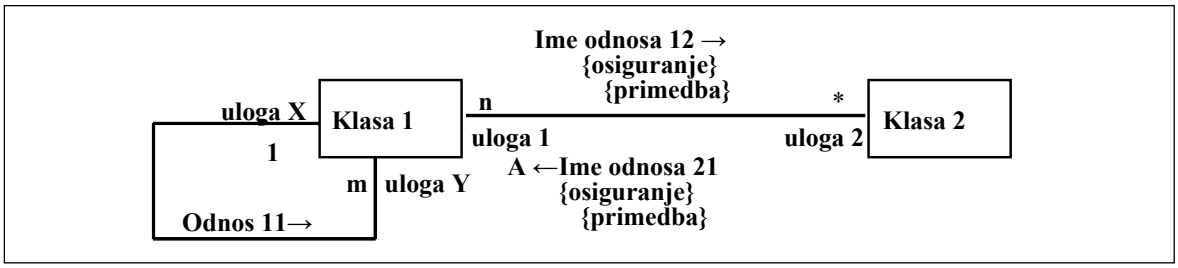


Primer 4 (opšte četvorostruke asocijacije)



Pored toga, asocijacije čine objektno-orijentacioni princip izmene informacija za rešavanje problema preglednim kao i lako sledivim, čime pojednostavljuju uklapanje u docniji objektno-orijentisani program. Istovremeno sadrže asocijacije po potrebi obuhvatnu povezujuću semantiku, koja daje sliku o značajnim logičkim (kognitivnim) detaljima strukture veza u problemu koji se rešava.

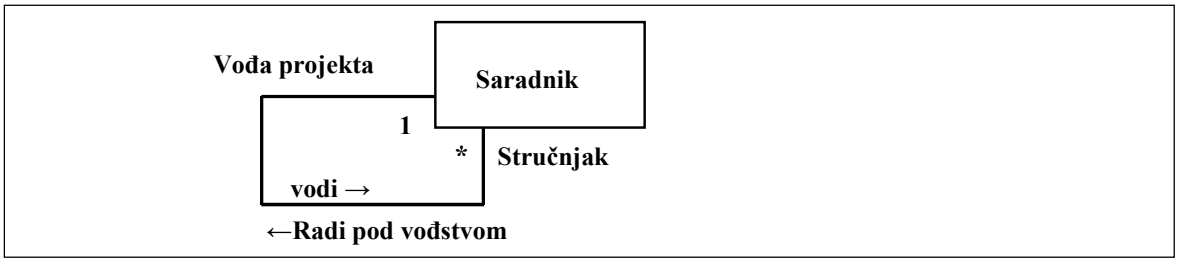
Opšti oblik notacije za jednu asocijaciju ima sledeći izgled:



Značenje

U opštem slučaju jedna asocijacija može biti dvostrano orijentisana (→, ←); ona može sadržati oznake uloga, koje su kod rekurzivnih asocijacija (tj. kod odnosa između objekata iste klase) čak obavezno predviđene (videti „uloga X" i „uloga Y").

Primer za rekurzivnu asocijaciju



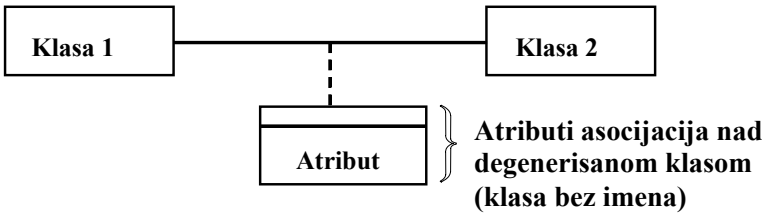
Količinski aspekti se uzimaju u obzir pomoću specifikacija o multiplicitetu (podaci o broju). Pored toga su, gde je to moguće, predviđeni osiguranja i primedbe. U praksi postoje veze objekata koje su privremene, tako da je u tom slučaju imamo primedbu „privremen".

Za zadavanje multiplikativnosti su moguće različite varijante; ovde dajemo tipične primere:

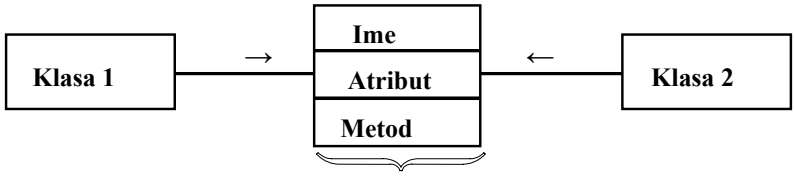
1 tačno jedan
0, 1 nula ili jedan
0..n između nule i celog broja n
k, j tačno k ili tačno j
0 .. * veće ili jednako nula (standardno – default – kada nema podatka)
1 .. * veće ili jednako 1, (ranije 1+)
k .. j, r .. s između k i j ili između r i s
itd.

Zadavanje tačno jednog broja označava se i kao kardinalitet.

Dalje tehnike predstavljanja asocijacija, posebno atributa asocijacija i klasa asocijacija, pored ostalih, ostaju za sada nerazmatrane.



Asocijativne klase imaju sledeći izgled:



Asocijativna klasa (težiti tome!)

Kako opšte važeće asocijacije nisu direktni deo objektnoorijentisanih jezika, mora onaj koji razvija softver eksplicitno naznačiti kako su asocijacije konstruisane. Asocijacije se normalno realizuju pomoću kontejnera (tzv. Container) preko kontejnerskih klasa (U ovom slučaju oni imaju ulogu da održavaju opšte objekte; oni se ne smeju mešati sa klasom Container iz AWT – Abstract Windowing Toolkit - koja će biti obrađena). Kontejnerske klase su pritom definisane u bibliotekama standardnih klasa objektnoorijentisanih programskih jezika. Sve kontejnerske klase povezuje osobina da moraju da održavaju određenu količinu objekata (uporediti sa multiplicitetom). Tako kontejnerske klase pomoću odgovarajućih metoda uvode i izbacuju objekte, ispituju raspoloživost objekata i prebrojavaju objekte.

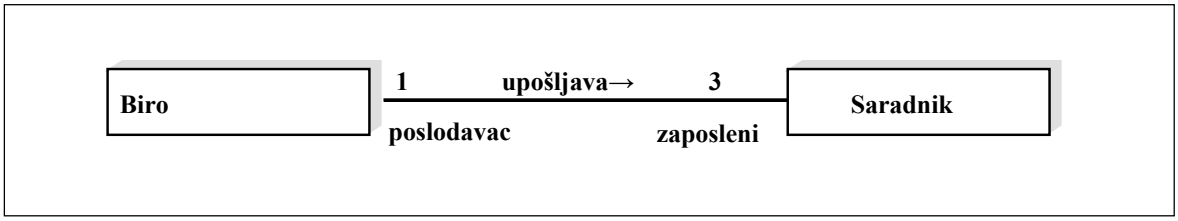
Kod kontejnera postoje dve grupe:

a) sekvencijalni kontejneri, kod kojih su objekti sakupljeni u linearnoj strukturi podataka (referentni atributi, polja – array, vektor);

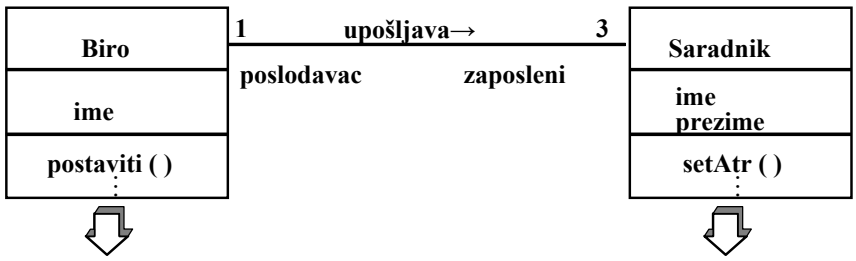
b) asocijativni kontejneri, kod kojih se objekti mogu identifikovati pomoću određenog ključa (rečnik, hash tabela).

Kontejneri pripadaju najvažnijim alatkama koje se znaju u softverskoj tehnici. Skoro svaka banka podataka, svaka tablična kalkulacija, svaki program sadrži kontejner. Kontejnerske klase koje se primenjuju u objektno-orijentisanom programiranju, razlikuju se pritom, pored svoje sekvencijalne ili asocijativne prirode, značajno prema drugim markantnim osobinama, kao što su broj raspoloživih funkcija (korisnički spektrum), efikasnost (vremenska dimenzija), zahtev za memorijom (utrošak memorije) i produživost (dinamički rast). Navedene kontejnerske forme (Array, Vector, Dictionary, Hashtable) i odatle izvodive strukture (stack, queue, dvostruko ulančane liste, strukture stabla) morale bi detaljnije da budu prodiskutovane nego što je ovde moguće. Ovde će biti obrađeni samo referentni atributi i vektorski kontejneri. Stoga se ponovo vraćamo na primer „projektni biro” / „saradnik”.

Primer: najjednostavnija forma primene asocijacija sa referentnim atributima (multiplicitet 1 tj. 3)



Asocijacija „zaposlen" se može u ovom slučaju realizovati na sledeći način: u klasi „saradnik" se definiše atribut „poslodavac", koji saglasno multiplicitetu 1 sadrži referencu na objekat klase „biro". Naprotiv tome, u klasi „biro" biće uvedena tri različita atributa „zaposlen", koji usled multipliciteta 3 imaju jednu referencu na objekat „saradnik".




```
public class Biro {

 String ime;
// tri referentna atributa
Saradnik zaposlen1;
Saradnik zaposlen2;
Saradnik zaposlen3;

// Konstruktor
public Biro (String s) {
   ime = new String (s) ;
}

// Metode
public void postaviti (
  Saradnik m1, Saradnik m2,
  Saradnik m3) {
  zaposlen1 = m1;
  zaposlen2 = m2;
  zaposlen3 = m3;
  }
} // Kraj klase Biro
```

```
class Saradnik  {


        String ime;
        String prezime;
        // jedan referentni atribut
        Biro poslodavac;

        // Konstruktor
        public Saradnik (Biro b) {
        poslodavac = b;
        }

        //Metode
      public void setAtr (String vn,
      String nn) {
      ime = new String (vn) ;
      prezime = new String (nn);
      }
  } // Kraj klase Saradnik
```

Objašnjenja radi uvođenja u JAVA-strukture

Obe klase, „Biro" i „Saradnik", se kreiraju pod kontrolom semantičkih pravila koja su sadržana u asocijacijama (podaci o „poslodavcu" i „zaposlenom" sa podacima o multiplikativnosti 1 i 3). Pri implementaciji treba saglasno tome asocijaciju između obe klase (public class Biro, odnosno class Saradnik) uneti u JAVA-konstrukciju.

U „class Biro" se na kraju definiše atribut „ime" za održavanje jednog tekstualnog objekta JAVA-klase „String". Najzad se definišu tri referentna atributa,

```
Saradnik zaposlen1;
Saradnik zaposlen2;
Saradnik zaposlen3;
```

koji mogu obezbediti veze između objekta klase „Biro" (u ovom slučaju multipliciteta 1) i tri objekta klase „Saradnik" (osobina kontejnera). Oznake uloge „zaposlen" za tri saradnika biroa pritom predstavljaju objekte klase Saradnik (class Saradnik {...}).

U klasi „Saradnik" se na sličan način, pomoću JAVA-klase „String" prvo upisuju najvažniji lični podaci saradnika (ovde ograničeni na atribute „Prezime" i „Ime"). Na kraju se – analogno postupku u klasi „Biro" – definiše jedan referentni atribut, u našem slučaju

```
Biro poslodavac;
```

čime se upućuje klasa „Saradnik" na instancu klase „Biro" (kontejnerska osobina).

Kao sledeće se obrađuju metode obe klase. U klasi „Biro" se definiše naredba

```
public Biro (String s)
```

koja brine za to da se može kreirati objekat klase „Biro"; u datom slučaju se primenjuje kao parametar jedna referenca na jedan string-objekat. Objekat generisan ovom naredbom se predstavlja nizom znakova (koji se sastoji od slova).

Zatim sledi naredna metoda

```
public void postaviti (Saradnik m1, Saradnik m2, Saradnik m3)
```

Primena metode (kôd u telu metode) brine za to, da tri reference saradnika budu dodeljene referentnim atributima biroa. Objekti saradnika pritom moraju predhodno da budu generisani naredbom u klasi „Saradnik":

```
public Saradnik (Biro b)
```

Naredba sadrži u svojoj listi parametara objekat „b" klase „class Biro", da bi se time u klasi „Saradnik" naznakom

```
poslodavac = b;
```

realizovala asocijacija „Biro kao poslodavac zapošljava saradnike kao zaposlene".

U metodi „postaviti" klase „Biro" utvrđuju se veze sa klasom „Saradnik" dodeljivanjem argumenata m1, m2, m3 referentnim atributima zaposlen1, zaposlen2, zaposlen3 naredbama

```
zaposlen1 = m1;
zaposlen2 = m2;
zaposlen3 = m3;
```

U klasi „Saradnik" se metodom

```
public void setAttr (String vn, String nn)
```

memorišu personalni podaci saradnika. Ostale metode se u obe klase mogu pretpostaviti, ali ovde neće biti dalje spominjane.

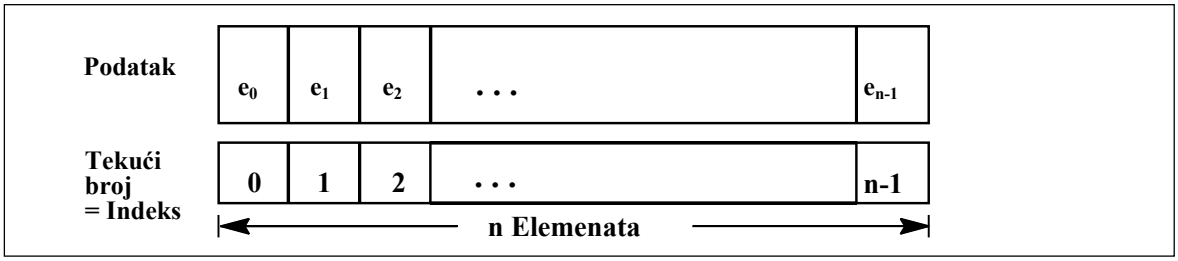
Ukoliko se žele testirati primenjene klase, mora se kreirati odgovarajuća klasa pod nazivom „main". U ovom slučaju će biti primenjena klasa „TestReferentnihAtributa" (kraće razmatranje):

```
import Biro;
import Saradnik;

public class TestReferentnihAtributa
{

        public static void main ( String s [] )
        {
                // Kreirati Biro
                Biro b = new Biro ( "StatikBiro Vagres" ) ;

                // Kreirati saradnike
                Saradnik Inzenjer1 = new Saradnik ( b ) ;
                Saradnik Inzenjer2 = new Saradnik ( b ) ;
                Saradnik Inzenjer3 = new Saradnik ( b ) ;

                // Postavljanje saradnika
                b.postaviti ( Inzenjer1, Inzenjer2, Inzenjer3 ) ;

                // Ubacivanje licnih podataka
                Inzenjer1.setAttr ( "Schultz", "Sigi" ) ;
                Inzenjer2.setAttr ( "Mueller", "Max" ) ;
                Inzenjer3.setAttr ( "Wagner", "Willy" ) ;
        }
}       // Kraj class TestReferentnihAtributa
```

Ubacivanje asocijacija uz pomoć referentnih atributa nije – kao što se iz prethodnog primera vidi – praktično, jer se pri pojavi novih objekata ne vrši automatsko produženje. Svako pojedinačno produženje se mora ručno unositi; kod mnogo velikog broja objekata ne bi se moglo primenjivati takva dugotrajna procedura.

Fleksibilnije i pre svega efikasnije ubacivanje nego što je navedeno može se sprovesti pomoću struktura (kontejnera) „Array", „Vector", „Dictionary" i „Hashtable".

Kod „Array", čija struktura odgovara linearnom (jednodimenzionalnom) polju, sa uzastopno numerisanim pojedinim elementima podataka, za razliku od strukture stabla, koja predstavljaju nelinearne strukture podataka, mora se veličina polja („Array"), tj. dimenzija polja, unapred definisati; pri tome je dimenzija polja tokom izvršenja programa statička (uzimajući u obzir elemente podataka, pomoću „new" se može dinamički generisati). Ukoliko se hoće pristupiti određenom podatku – bilo radi čitanja ili upisivanja - , mora vrednost indeksne promenjive biti poznata. Ukoliko se poveća broj indeksa, nastaju višedimenzionalna polja (matrice), koji se mogu memorisati u jednodimenzionalnom nizu (u JAVA vrsta po vrsta – red za redom).

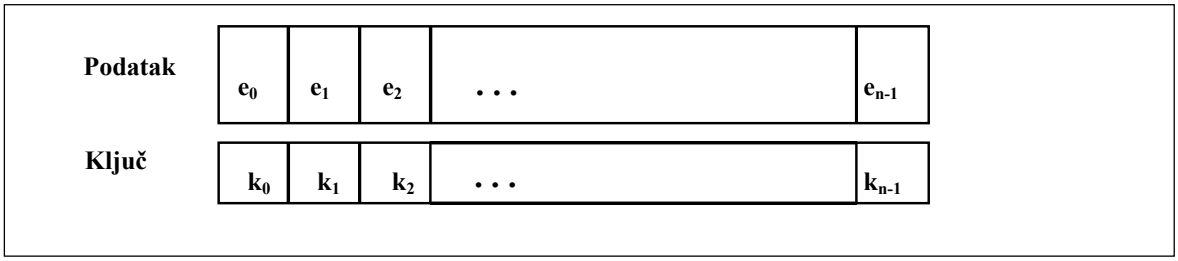


Uvođenje jednog elementa na kraj linearnog polja je jednostavno, pod uslovom da ima dovoljno rezervisanog mesta za niz. Ali, ako treba da se uvede neki element u unutrašnjost niza, bez upisivanja preko nekog postojećeg elementa, mora da se sprovedu operacije šiftovanja, što ima cenu u utrošenom vremenu. Isto važi i za brisanje nekog elementa, kada pri brisanju treba popuniti nastalu prazninu. Za indekse se formiraju nizovi brojeva, koji značajno pojednostavljuju pristup podacima i štede dosta vremena.

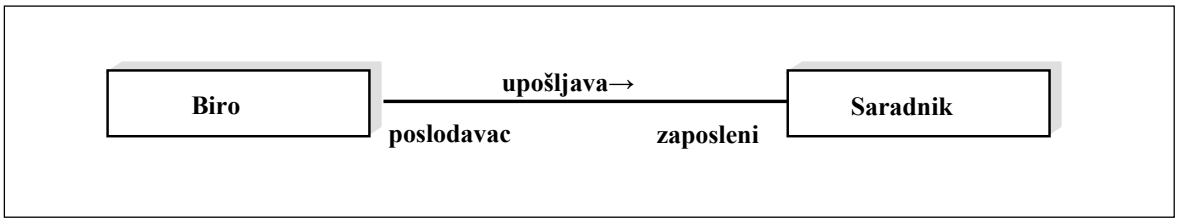
Rečnik („Dictionary") je isto tako jedno polje, ali sa specijalnom metodom indeksiranja; on se zove i asocijativni niz (Array). Dok kod normalnih nizova, pri pristupu nekom određenom elementu polja, odgovarajući indeks mora biti uvek poznat, može da se kod rečnika („Dictionary") indeks dobija iz sadržaja traženog elementa metodom indeksacije (o kojoj ovde neće dalje biti reči); pri traženju nekog elementa, čiji indeks nije poznat, ovaj postupak je izuzetno efikasan.

I asocijativni niz se ne može tokom izvršenja programa dinamički produžiti.

„Hashtable" je takođe jedno polje (Array), kojim se podaci mogu efikasno održavati. Međutim, Hash tabele se isplate samo kod velikih količina podataka. Kod malih količina podataka su troškovi (složenost) suviše veliki, a takođe je i zauzeće memorijskog prostora neprihvatljivo. Kod „Hashtable" se indeks određenog elementa podataka izračunava pomoću tzv. hash-funkcije, tj. pomoću postupka kodiranja, kod koga hash-funkcija daje formulu za dobijanje vrednosti indeksa. „Hashtable" je prema tome isto tako asocijativan niz, kod koga se ključ jednog elementa dobija iz sadržaja, tj. dela sadržaja toga elementa.

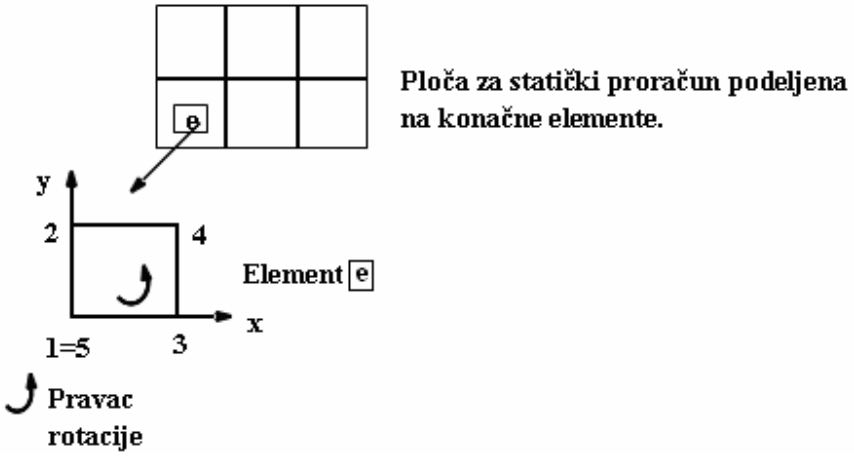


Strukture podataka koje se u JAVA pomoću kontejnera (Container) klase „Vector" mogu kreirati nude široku lepezu metoda za manipulisanje i održavanje. Iz tog razloga se kontejneru klase „Vector" u JAVA programu pridaje centralno značenje. Gore upravo prezentiran primer za asocijacije, primer „Biro"/„Saradnik", biće upravo zato još jednom razmatran, kao ugledni primer, primene kontejnere tipa „Vector" u programu JAVA.



Funkcionalnost „Vector"-a se može postići na sličan način kao i kod tzv. dvostruko ulančane liste, tj. pomoću jedne posebne strukture podataka, koja se, kao polja, sastoji od linearno uređenog skupa elemenata, ali se, za razliku od polja, može tokom izvršavanja programa dinamički proširivati. Pojednostavljeno rečeno, kontejner tipa „Vector" predstavlja dinamički pandam – s obzirom na vreme izvršenja – kontejneru statičkog tipa „Array". Ulančavanje između elemenata podataka jedne ulančane liste vrši se preko referenci ili pointera. Princip ulančavanja biće objašnjen na jednoj jednostruko ulančanoj listi:

Primer (Tehnika konačnih elemenata)

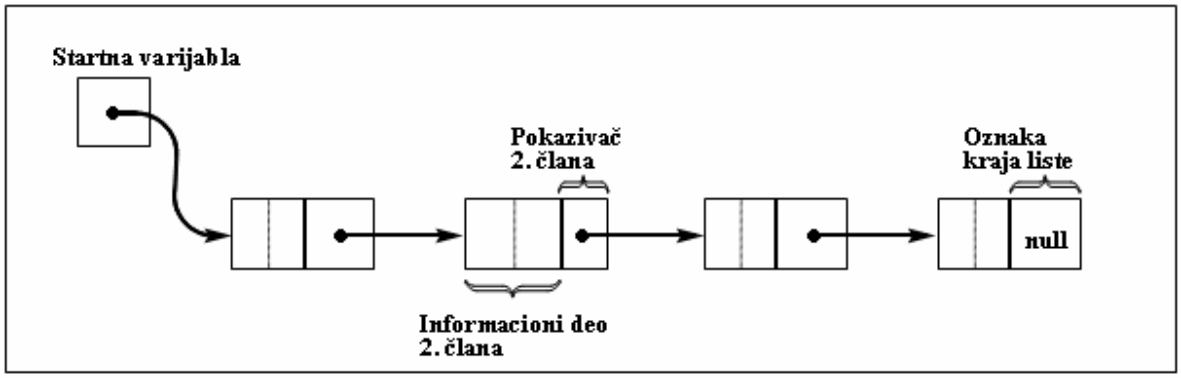


Ploča za statički proračun podeljena na konačne elemente.

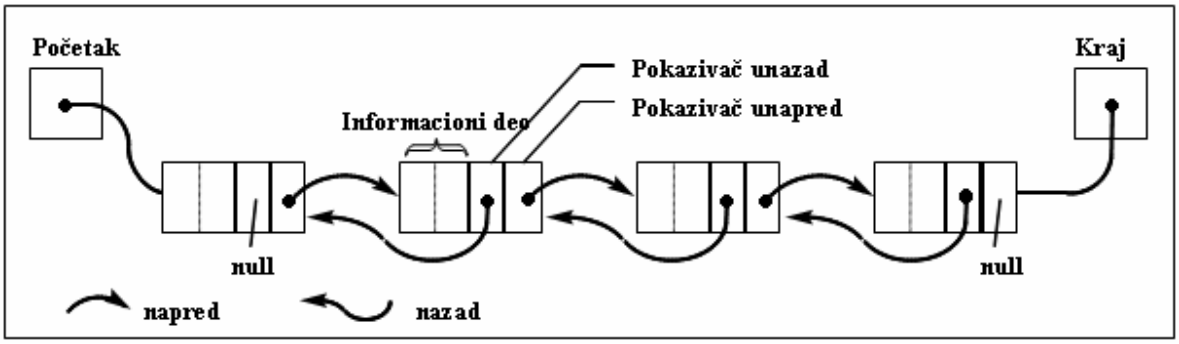
Uređenje temena (čvorova) kao lančane liste

čvor broj	x - koordinata	y – koordinata	pokazivač na naredni čvor
1	x_1	y_1	3
2	x_2	y_2	5
3	x_3	y_3	4
4	x_4	y_4	2
5=1	x_1	y_1	0 (null)

Jednostruko lančana lista može se grafički predstaviti kao jedan lanac, čiji se članci sastoje od stvarnih podataka i jednog pointera, pri čemu pointer ukazuje na sledeći članak u lancu. Poslednji članak u lancu ima posebnu vrednost, koji se označava kao „nula". Početak liste se uobičajeno označava jednom sopstvenom startnom varijablom, kojom se može pozvati prvi element liste.



Za fizičku realizaciju u memoriji koriste se posebno delovi za informacije i pointere. Pomoću ulančavanja i za ne neophodno uzajamno zavisne memorisane sadržaje mogu se sve aplikacije sa visokom frekvencijom upisa i brisanja efikasno realizovati, pošto se sve operacije umetanja i brisanja, kao i sortiranja, mogu izvršavati vrlo brzo. Nasuprot tome, traženje je sporo. Jednostruko ulančane liste se – kao što se iz šematskog prikaza vidi – pretražuju spreda unazad, ali ne i u obrnutom pravcu. Zbog toga nije moguće pri pretraživanju jednostruko ulančane liste pristupiti predhodnim člancima lanca. Ovaj nedostatak se rešava dvostruko ulančanom listom: kod dvostruko ulančane liste se pretraživanje može sprovesti kako unapred – od početka ka kraju liste -, tako i unazad – od kraja ka početku liste. Dvostruko ulančane liste se mogu šematski predstaviti na sledeći način:



Najvažnija prednost dvostruko (ili dvostrano) ulančane liste je da se troškovi pretraživanja po jednom ulazu znatno redukuju.

JAVA-klasa „Vector" (videti priloženu definiciju klase) sadrži tri različite konstrukcije, koje omogućavaju dinamičko proširenje kontejnera tipa „Vector" tokom izvršenja programa.

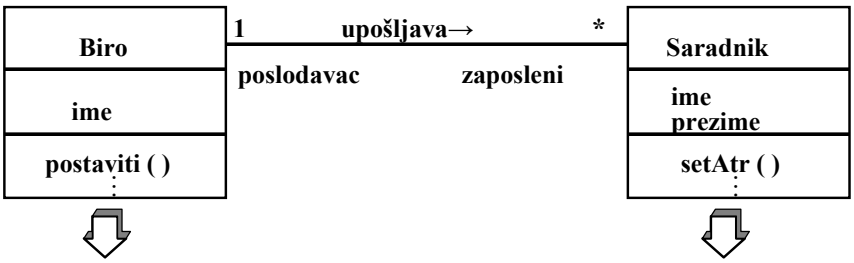
```
public class Vector extends Object implements Cloneable, Serializable
{
        // public konstruktori
        public Vector(int initialCapacity, int capacityIncrement);
```

```
    public Vector(int initialCapacity);
    public Vector();

    // protected instance promenljiva
    protected int capacityIncrement;
    protected int elementCount;
    protected Object[] elementData;

    // public instance metoda
    public final synchronized void addElement(Object obj);
    public final int capacity();
    public synchronized Object clone(); // Pretovaruje Object.clone()
    public final boolean contains(Object elem);
    public final synchronized void copyInto(Object[] anArray);
    public final synchronized Object elementAt(int index);
    public final synchronized Enumeration elements();
    public final synchronized void ensureCapacity(int minCapacity);
    public final synchronized Object firstElement();
    public final int indexOf(Object elem);
    public final synchronized int indexOf(Object elem, int index);
    public final synchronized void insertElementAt(Object obj, int index);
    public final boolean isEmpty();
    public final synchronized Object lastElement();
    public final int lastIndexOf(Object elem);
    public final synchronized int lastIndexOf(Object elem, int index);
    public final synchronized void removeAllElements();
    public final synchronized boolean removeElement(Object obj);
    public final synchronized void removeElementAt(int index);
    public final synchronized void setElementAt(Object obj, int index);
    public final synchronized void setSize(int newSize);
    public final int size();
    public final synchronized String toString(); // Pretovaruje Object.toString();
    public final synchronized void trimToSize();
}
```

Definicija klase „Vector”

Prvi konstruktor koji se primenjuje u posmatranom slučaju kreira jedan kontejner sa početnim kapacitetom od 10 instanci, pri čemu je vrednost 10 unapred data vrednost (default). Kada raspoloživi memorijski prostor uvođenjem novih elemenata biva prekoračen, tako da kapacitet vektora nije dovoljan, biće kapacitet kontejnera tokom izvršenja programa povećan za sledećih 10 instanci. Ostala dva konstruktora omogućavaju da se početni kapacitet „Vector”-a može unapred definisati (public Vector(int initialCapacity)), a takođe i dodatni kapacitet (public Vector(int initialCapacity, int capacityIncrement)).

Sa stečenim znanjima o kontejneru tipa „Vector” može se sada preformulisati primer za primenu asocijacije „Biro zapošljava saradnika”; ovde se klasa „Vector” inicijalizira ulaznom naredbom („import java.util.Vector;”):




```
public class Biro {

    String ime;
    // kontejner tipa Vector
    Vector team = new Vector();


    // Konstruktor
    public Biro (String s) {
    ime = new String (s) ;
    }

    // Postaviti metodu
    public void postaviti (
    Saradnik m ) {
    Team.addElement (m);
    }
} // Kraj klase Biro
```

```
class Saradnik   {


    String ime;
    String prezime;
    // referentni atribut
    Biro poslodavac;

    // Konstruktor
    public Saradnik (Biro b) {
  poslodavac = b;
    }

    // Metoda za licne podatke
public void setAtr (String nn) {

ime = new String (vn);
    prezime = new String (nn);
  }
}// Kraj klase Saradnik
```

Objašnjenja radi primene „Vector" kontejnera

Velika razlika pri upotrebi referentnih atributa se sastoji u tome da se sada u klasi „Biro" može održavati proizvoljan broj saradnika; konstruktorom klase „Vector"

```
Vector team = new Vector();
```

primenjeni „Vector" - objekat Team može sadržati najpre 10 instanci (uporediti sa default-pravilom), po potrebi sadržati još 10 instanci, koje se tada u metodi „postaviti" unose u kontejner. Poziv ovih instanci se pritom odvija u metodi „main" klase za testiranje. Za unošenje referenci je u klasi „Vector" zadužena jedna jedina metoda „addElement(Object obj)". Jednostavnije se realizuje generisanje pojedinih saradnika u main-metodi, pošto se od sada može koristiti jedna for-petlja; pojedinačna unošenja, koja sobom donose mnogo pisanja, mogu se sada izbeći. Ostale naredbe, posebno one klase „Saradnik", odgovaraju primenama sa referentnim atributima, tako da se ne moraju ponovo pominjati.

Test-klasa za korišćenje sa „Vector”- klasom ima tako sledeći principijelni izgled:

```
import Biro;
import Saradnik;

public class TesrReferentnihAtributa
{
        public static void main (String s [])
        {
                // kreirati Biro
                Biro b = new Biro ("Projektni Biro Vagres") ;
                // kreirati i postaviti saradnike
                int broj = 3;

                for (int i = 0; i < broj; i++ )
                {
                        Saradnik Inzenjer = new Saradnik (b) ;
                        b.postaviti ( Inzenjer ) ;
                }
        }
}
```

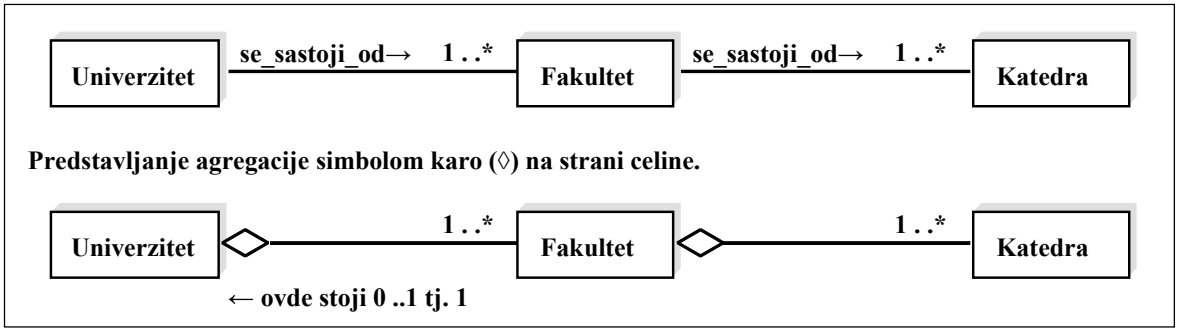
Uputstvo: Umesto broja 3 mogla bi stajati promenljiva, npr. n, čime bi se povećala opštost programa.

6.2.10. Modeliranje agregacija kao posebnih slučajeva asocijacija

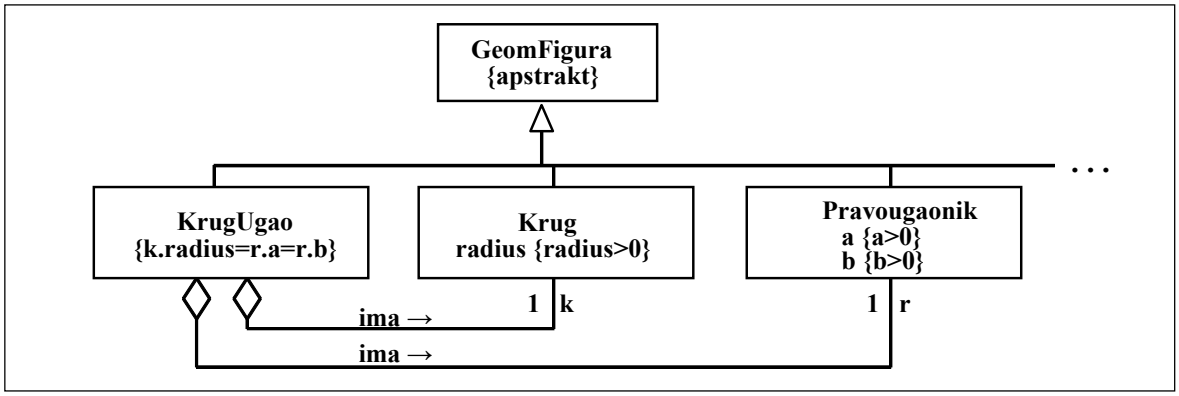
Specijalan slučaj asocijacija je tzv. agregacija, koja upravo u inžinjerstvu ima posebnu ulogu, koja se pojavljuje u mnogim tehničkim primenama. Kod agregacija se radi o odnosima klasa, koji slede jednu zajedničku logiku gradnje, što je upravo tipično za mnoge inženjerske probleme (sklapanje delova konstrukcije u jednu građevinu, sklapanje neke mašine, konfigurisanje uređaja ili tehničkih sistema, sklapanje projekata iz pojedinačnih delova, itd.). Posebnost agregacija se takođe sastoji u tome da se asocijacije između delova i (od delova sklopljene) celine mogu modelirati (odnos deo-celina, tj. Part-Whole).

Predstavljanje u obliku asocijacija

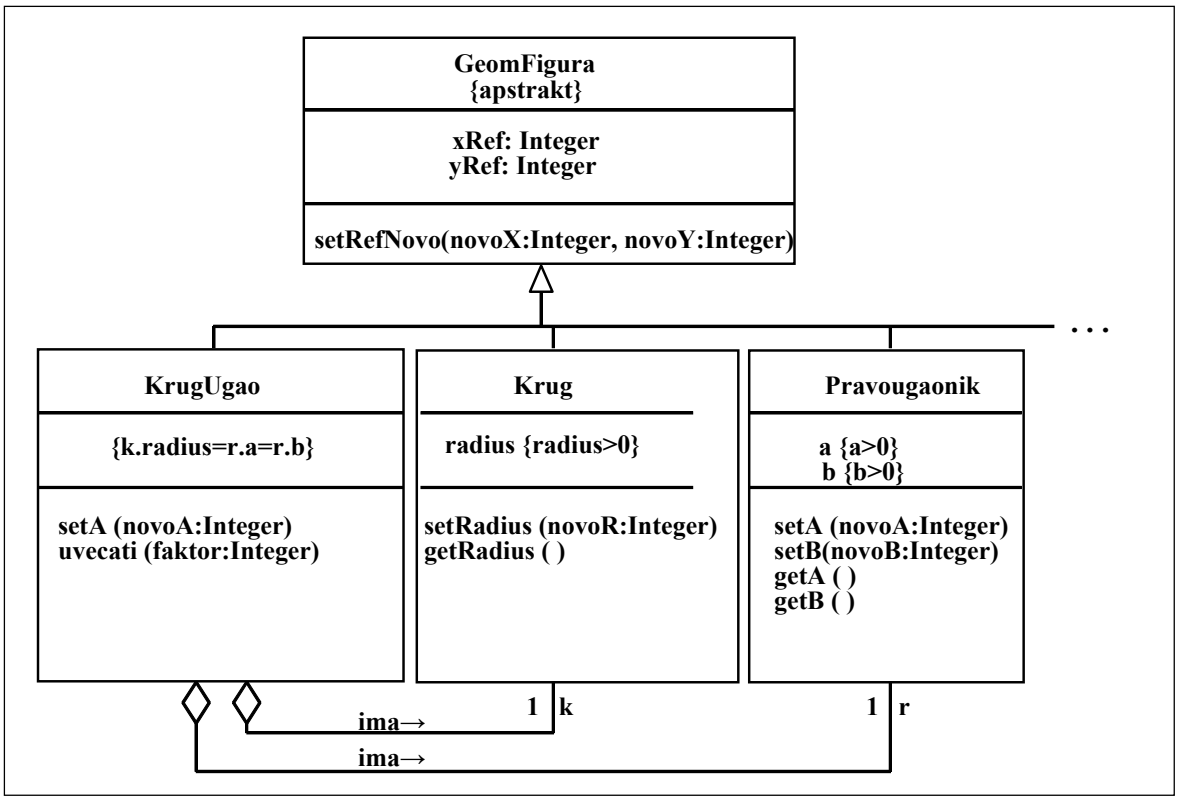
Primer iz svakodnevnog života



Primer: odsečak iz jednog CAD-sistema



Proširenjem na odgovarajuće atribute i metode, dobija se sledeći klasni dijagram:



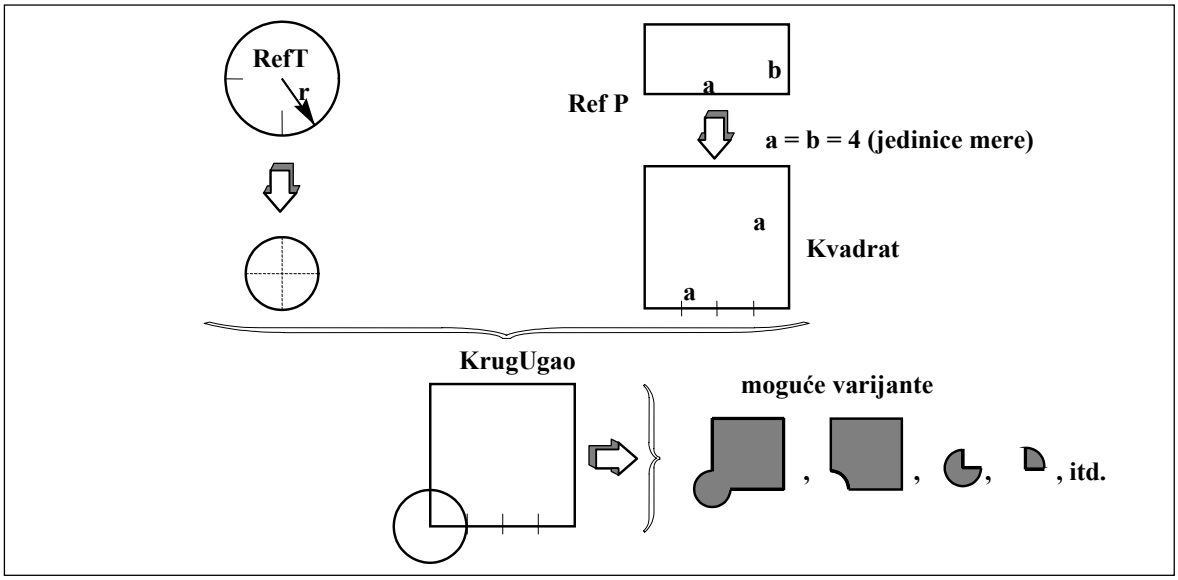
Postavka problema

U postojeći klasni dijagram može se grafičkom superpozicijom od jednog kvadrata i jednog kruga formirati nova klasa, nazovimo je „KrugUgao”, koja predstavlja jedan novi osnovni grafički elemenat.

JAVA-aplikacija sa jednostavnim referentnim atributima

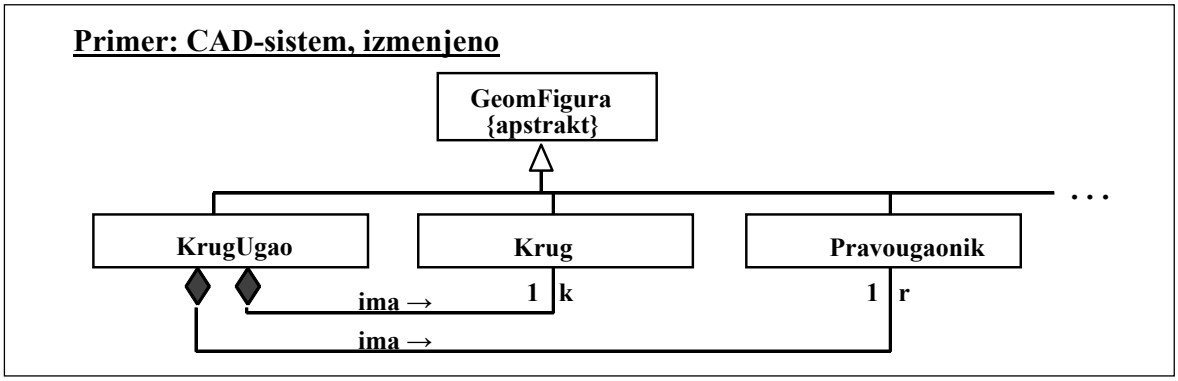
```
class KrugUgao extends GeomFigura
{
        Krug k;
        Pravougaonik p;

        public void setA (int novoA)
        {
                k.setRadius (novoA/4);
                p.setA (novoA);
                p.setB (novoA); // Osiguranje kvadrata
        }
        public void povecanje (int faktor)
        {
                int a;
                a = r.getA();
                setA (a*faktor);
        }
} // Kraj class KrugUgao
```

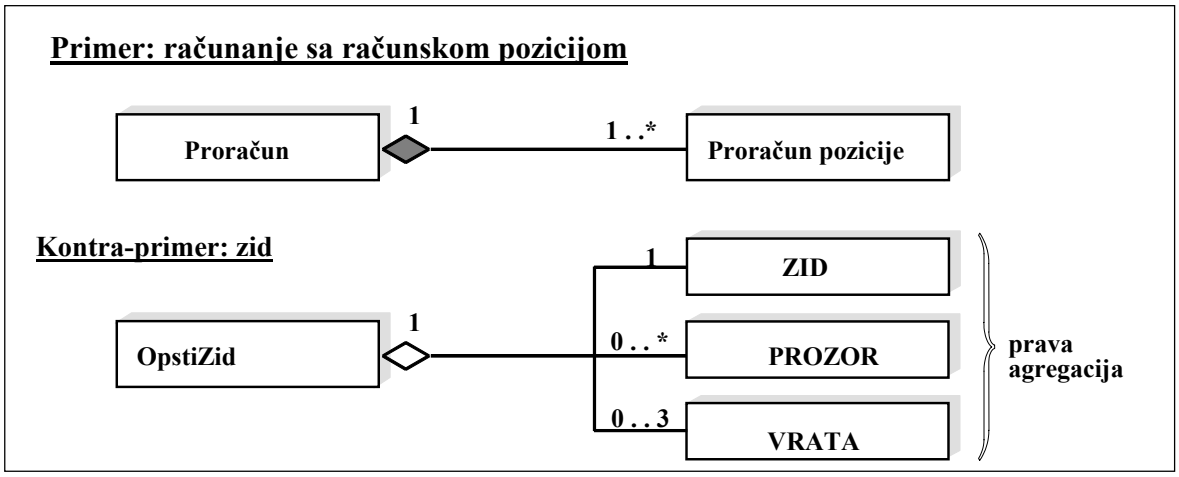
Ovim se mogu dobiti sledeći geometrijski oblici, što se zatim može daljim koracima obraditi:



Egzaktno uzeto, kod agregacije „Krug"-a „Pravougaonik"-a radi se o specijalnom slučaju agregacije, o takozvanoj kompoziciji, pošto „KrugUgao" bez kruga nije krug-ugao već pravougaonik. Kompozicija, nasuprot uobičajenoj agregaciji, ima osobinu da sastavni delovi, od kojih se kompozicija sastoji, sami za sebe ne mogu egzistirati. Nasuprot tome, kod obične agregacije mogu sastavni delovi potpuno nezavisno pojedinačno egzistirati. Da bi se kompozicija razlikovala od agregacije, romb kao simbol se popunjava crnim.

Primeri





Zid bez prozora može postojati, ali prozor bez zida ima manje smisla. Jedna proračunska pozicija u nekom proračunu gubi svrhu postojanja, kada proračun više ne postoji; drugim rečima, kod neke kompozicije je postojanje pojedinog dela podređeno celini.

Asocijacije, tj. njihove posebne forme, agregacije i kompozicije, mogu se tu dobro primeniti, radi razjašnjenja razmena informacija između objekata asociranih klasa, što je jedan od najvažnijih principa rešavanja problema kod objektnoorijentisanih modela. Predstavljeni primer za „KrugUgao" služi tome da se pri razmeni informacija tačnije sagledavaju izvršene instance.

Primer: razmena informacija za gore upravo primenjen CAD-sistem (jedan izvod)

Uzima se da se jednom kreiranom objektu klase „KrugUgao" šalje poruka „uvecati ()"; poruka se realizuje pomoću imena jedne metode dodavanjem odgovarajućoj listi argumenata.

Jedan pogled na JAVA-kôd pokazuje da jedan objekat klase „KrugUgao" unutar metode „uvecati ()", između ostalih sam kreira jednu poruku „setA (neuA: Integer)", koja sa svoje strane prenosi dalje poruke „setRadius (...)" na objekat krug, tj. „setA (...)" kao i „setB (...)" na objekat pravougaonik. Ovo znači da agregacija tj. kompozicija prenosi akcije dalje na svoje delove (to se

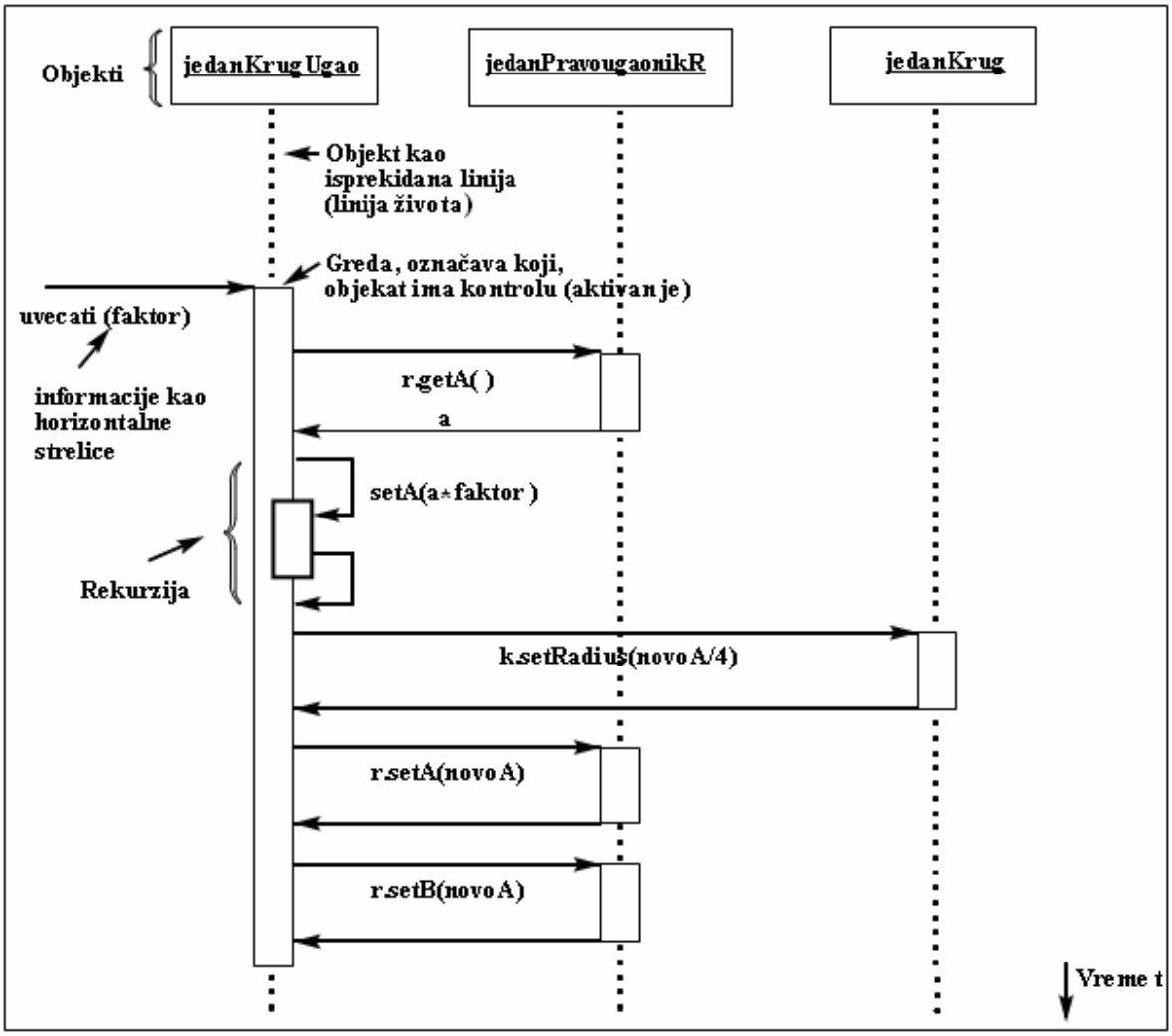
naziva propagacija metoda). Ovaj prenos na delove metoda se pritom može izvršavati samo tada, kada se ovo „razume" od strane delova, tj. kada su one tamo odgovarajuće definisane.

Da bi se razmena poruka mogla pratiti i kontrolisati, primenjuju se u UML (Unified Markup Language), pored klasnih dijagrama, posebni dijagrami:

- Sekvencijalni dijagrami (interakcioni dijagrami, dijagrami događaja), kojima se dokumentuje vremenski tok razmene poruka.

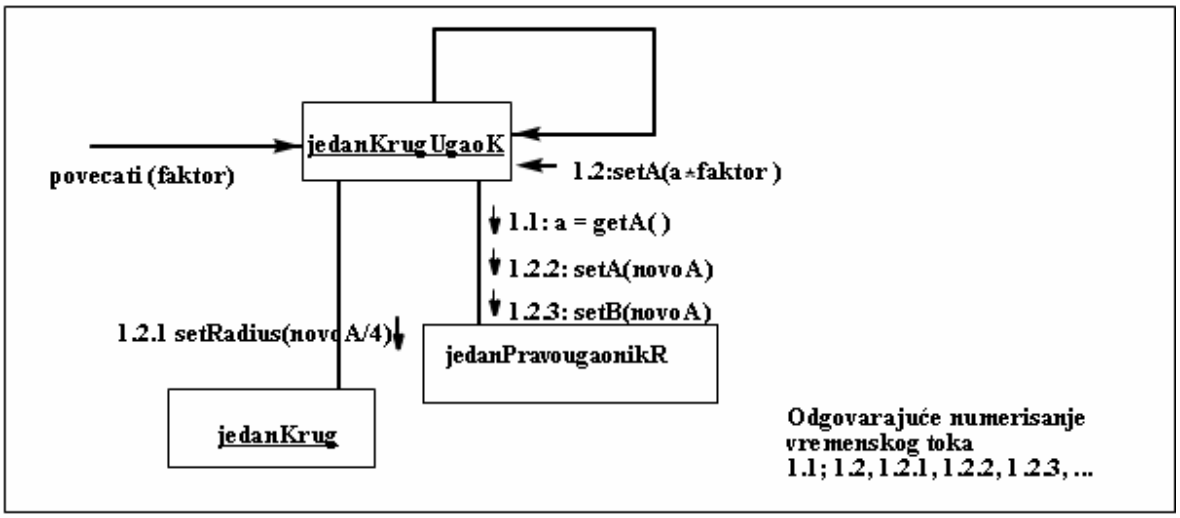
- Relacioni (kolaboracioni) dijagrami, kojima se predstavlja razmena informacija unutar spektra odnosa klasa koje učestvuju u razmeni informacija.

Za posmatrani primer, u kome objekat „KrugUgao" sadrži poruku „uvecati ()", može se za navedene objekte klasa „KrugUgao", „Krug" i „Pravougaonik" dati sledeći sekvencijalni dijagram:



Pripadajući kolaboracioni dijagram, koji stavlja u prvi plan međusobne odnose objekata, bazira se na klasnom dijagramu i jednoj razumljivoj numeraciji nastalih razmena informacija: u

osnovi, on prikazuje iste činjenice kao i sekventni dijagram, ali iz aspekta klasnih dijagrama. U našem slučaju on ima sledeći izgled:



Na osnovu grafičkog prikaza izmena informacija jasna je prednost u odnosu na poziv procedure (subroutine ili slično) u klasičnoj obradi podataka i to iz sledećih razloga:

- Poruke se mogu generisati samo pozivom na objekte, koji preko odgovarajućih metoda obezbeđuju njihovu obradu.

- Kontrola nad obradom poruka pripada odgovarajućem objektu, dok kod klasičnih programerskih paradigmi pozvani objekat zadržava tu kontrolu.