UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA

Dušan T. Malbaški

OBJEKTNO ORIJENTISANO PROGRAMIRANJE kroz programski jezik C++

Novi Sad, 2008.

Edicija: "TEHNIČKE NAUKE - UDŽBENICI"

Naziv udžbenika: "Objektno orijentisano programiranje kroz programski jezik C^{++}"
Autor: dr Dušan Malbaški, redovni profesor Fakulteta tehničkih nauka u Novom Sadu

Recenzenti: dr Miroslav Hajduković, redovni profesor FTN u Novom Sadu
dr Ivana Berković, redovni profesor Tehničkog fakulteta "Mihajlo Pupin" u Zrenjaninu

Izdavač: Fakultet tehničkih nauka u Novom Sadu

Glavni i odgovorni urednik: prof. dr Ilija Ćosić, dekan Fakulteta tehničkih nauka

Štampanje odobrio: Savet za izdavačko-uređivačku delatnost FTN u Novom Sadu

Predsednik saveta: dr Radomir Folić, profesor emeritus

PREDGOVOR

Ovaj udžbenik namenjen je studentima koji izučavaju objektnu metodologiju osloncem na model-jezik C++. Izrađen je na bazi monografije "Objekti i objektno programiranje kroz programske jezike C++ i paskal" (Fakultet tehničkih nauka, 2006.) istog autora, a sa ciljem da se monografija prilagodi potrebama onih koji se tek upoznaju sa objektnom metodologijom. U odnosu na monografiju izvršene su sledeće bitne izmene:

1. *Pre svega, organizacija i način izlaganja prilagođeni su udžbeničkoj prirodi knjige.*
2. *Bitno je promenjen tretman osnovnih pojmova - klase, objekta, nasleđivanja i inkluzionog polimorfizma - tako što je cela metodologija postavljena na novu osnovu formiranu konceptualnim pristupom.*
3. *Pojmovi modularnosti i odgovarajuće softverske komponente - modula, razdvojeni su i definisani saobrazno konceptualnom pristupu i aktuelnim trendovima u objektnoj metodologiji.*
4. *Način fukncionisanja konstruktora i destruktora obrađen je primenom novog ANSI standarda.*
5. *Iz istog razloga znatno je modifikovano poglavlje o generičkim klasama i funkcijama.*
6. *Dodat je nov odeljak vezan za prostore imena.*
7. *Svi primeri usklađeni su sa novim standardom, a dodati su i novi.*
8. *Uvedena je tzv. "camel" notacija za članove klase.*
9. *Uklonjen je tekst koji se odnosi na programski jezik paskal.*

Nadam se da će ovaj udžbenik poslužiti svrsi i omogućiti čitaocu da se lakše snađe u interesantnom, raznolikom, ali i kompleksnom svetu objekata, te da će mu pomoći da ostvari osnovni zadatak svakog programera - izradu kvalitetnog softvera.

U Novom Sadu, novembra 2008.
Autor

S A D R Ž A J

1. UVOD U OBJEKTNO PROGRAMIRANJE

Svaka programska aplikacija je inherentno složen sistem. Pre svega, softver je sistem jer se sastoji od elemenata (podsistema) i veza između njih. Elementi i podsistemi softverskog sistema mogu biti logički zaokruženi segmenti, potprogrami, moduli, programi, datoteke... Osnovni, ali ne i jedini, tip veza predstavlja međusobna razmena podataka[1]. Takođe, i za najjednostavije programe lako je definisati stanje, recimo kao tekuće vrednosti svih promenljivih i stanje steka. Ako uzmemo u obzir činjenicu da su se potprogrami pojavili još 1959. sa programskim jezikom FORTRAN II, izlazi ne samo da je softver sistem nego da tu karakteristiku ima od samog početka programiranja.

Softver je, pored toga što je po prirodi sistem, još i *veliki sistem* (large scale system, engl.) i "neki softverski proizvodi spadaju u najsloženije sisteme koje su ljudi realizovali" (Lipajev, [33]). Pri tom, pojam "veliki sistem" treba shvatiti u Lernerovoj [64] interpretaciji kao "skup uzajamno povezanih podsistema objedinjenih opštim ciljem funkcionisanja". Pošto je svaki pristojan program realizovan modularno, a modul je tipičan podsistem, zaključak se nameće sam po sebi.

Pre nego što se bliže pozabavimo tezom o kompleksnoj prirodi softvera, treba definisati sam pojam kompleksnosti (složenosti). Opredelićemo se za definiciju iz [37] prema kojoj je kompleksnost neke jedinice posmatranja *mera mentalnog napora* potrebnog da se ona razume.

Prvi uzrok (velike) kompleksnosti softvera kao sistema je kompleksnost indukovana *složenošću problema* koji se rešavaju na računaru. Ti problemi su, s obzirom na nivo tehnologije, uvek bili složeni: rešavanje sistema linearnih jednačina reda nekoliko desetina, u svoje vreme, pre tridesetak godina, bilo je isti takav izazov kao što je danas izrada komplikovanih mrežnih aplikacija ili upravljanje svemirskim letelicama pomoću računara. Rešavanje složenog problema, s druge strane, zahteva timski rad praćen svim uobičajenim poteškoćama vezanim za upravljanje projektom: planiranje, organizaciju, koordinisanje, komunikaciju, finansije itd. Buč (G.Booch, [3]) ističe činjenicu da je softver *diskretan sistem* te je,

[1] Pored ove veze mogu se uspostaviti i druge, npr. veza sadržavanja između komponenata programa ili različite semantičke veze između datoteka.

kao takav, manje upravljiv od kontinualnih sistema. Pri tom, poziva se na Parnasa koji tvrdi: "kada kažemo da je sistem opisan kontinualnom funkcijom, u stvari tvrdimo da ne sadrži skrivena iznenađenja. Male izmene na ulazu uvek izazivaju male izmene na izlazu". Softver nije takav sistem. Stanja su diskretna te se, prema tome, teško može govoriti o "malim" i "velikim" promenama ulaza odnosno izlaza. Svaka ulazna aktivnost izaziva skokovitu promenu stanja, a s obzirom na obično enormno velik broj tačaka u prostoru stanja, promena može biti neočekivana i neželjena. Ako izvedemo kosi hitac gotovo je sigurno da će lansirani predmet, opisavši parabolu, završiti na lokaciji koja se prilično precizno može predvideti proračunom. U programu što simulira taj isti hitac sasvim je moguće (zbog neizbežnih grešaka) da se predmet u toku leta zaustavi i ostane nepokretan! Pored toga, ne treba smetnuti s uma činjenicu da su kontinualni veliki sistemi u načelu inertni, dakle imaju relaksacioni vremenski interval u kojem se može reagovati na neželjene promene stanja. Ništa slično, naravno u principu, ne može se reći za softver. Promene stanja - željene i neželjene - trenutne su, bez mogućnosti za intervenciju te stoga nije čudno što softverski moduli obiluju segmentima za proveru korektnosti ulaza, tzv. obradu izuzetaka (engl. exception handling) čiji je zadatak da spreče zabranjene promene stanja i da izveste o tome. I trivijalan interaktivni program mora biti zaštićen, bar od pritiska na taster sa nedefinisanom ulogom. Mejer (B. Meyer, [1]) razvio je posebnu varijantu programiranja, nazvanu ugovornim programiranjem, gde se provera ispunjenosti preduslova za izvršavanje neke programske komponente prihvata kao neodvojivi deo kodiranja i ugrađena je u programski jezik. I na kraju: analitičari i projektanti kontinualnih sistema imaju na raspolaganju moćno oruđe - matematičku analizu - što nije slučaj sa softverom gde matematički aparat - diskretna matematika - ima ipak manju upotrebnu vrednost.

Povrh svega, ponašanje softverskog sistema je stohastičko, i to ne u specijalnom nego u opštem slučaju. Dajkstra (E.W. Dijkstra, [31]) je još 1976. dalekovido primetio: "... smatram indeterminizam za normalnu okolnost, dok se determinizam sveo na poseban slučaj - i to ne odviše interesantan poseban slučaj". Vreme mu je dalo za pravo: interaktivno i konkurentno programiranje, klijent - server arhitektura, programiranje na mreži, otkazi - sve to čini ponašanje softvera nužno stohastičkim. Ovde se ne radi o pukoj konstataciji nego o činjenici koja se itekako mora imati u vidu tokom celog procesa razvoja, od analize i projektovanja do testiranja i održavanja.

Isto tvrdi i Lipajev: "Stohastičnost sastava poruka i trenutaka njihovog pristizanja, fluktuacija u redosledu i dužini pojedinih poslova, otkazi (...) kao i velika količina povratnih skokova u programima, uopšte uzev dovode do *stohastičnog ponašanja* softverskih sistema" ([33], str. 8).

Kompleksnost softverskog sistema se, prema Martinu [2], sastoji od dve komponente: kompleksnosti problema i kompleksnosti rešenja. Martin je to izrazio (kvalitativnom) jednačinom

$$c_a = c_p + c_r \quad \text{... (1.1)}$$

gde su c_a, c_p i c_r redom kompleksnost aplikacije[2], problema i rešenja.

Virt u [32] iznosi stanovište da pored inherentne kompleksnosti ili kompleksnosti *po sebi* softver poseduje i - nazovimo je tako - nametnutu kompleksnost (self-inflicted complexity) koja je rezultat dejstva tržišta i potrošačke psihologije korisnika. S jedne strane, u cilju održanja u konkurenciji (ili njenog eliminisanja) proizvođači su primorani da neprestano plasiraju sve novije i novije verzije programa koje, po pravilu, imaju proširenu funkcionalnost, često bez adekvatnog pokrića u stvarnim potrebama korisnika. Tako, na primer, obični tekst - procesori dostižu takav nivo složenosti da se obim priručnika meri brojem od 1000 stranica! S druge strane, Virt zapaža da i sami konzumenti softverskih proizvoda utiču na povećanje njihove kompleksnosti jer ne razlikuju esencijalne osobine programa koji nabavljaju, od onih koje je "lepo imati". Ako ovu, treću, komponentu obeležimo sa c_n jednačina 1.1 dobija oblik

$$c_a = c_n + c_p + c_r. \quad \text{... (1.2)}$$

Očigledno, želimo li da smanjimo kompleksnost softverskog sistema, to možemo postići dejstvom na tri faktora sa desne strane 1.2. Pre svega, a iz već navedenih razloga, nametnutu kompleksnost c_n vrlo je teško redukovati osim, možda, posebnom edukacijom korisnika ili pak apelima poput Virtovog iz [32] koji je (naravno) ostao bez naročitog odjeka.

Složenost problema c_p još je manje podložna našem uticaju - naprotiv! Prvo, ljudskom duhu primereno je upravo suprotno: težnja ka rešavanju sve komplikovanijih i zahtevnijih zadataka. Drugo, Dajkstra [7] sasvim logično precizira: "The automatic computer owes its right to exist, its usefulness, precisely to its ability to perform large computations where we humans cannot", što bi ukratko značilo da su računari izmišljeni baš zato da bi izvršavali poslove koji su za nas, ljude, preobimni.

Ostaje, dakle, treći činilac: kompleksnost rešenja. Za razliku od prethodna dva, na njega je moguće uticati izborom sredstava (npr. viši programski jezik umesto asemblerskog), korišćenjem nekih gotovih rešenja, uprošćavanjem algoritama i pažljivim prilagođavanjem struktura podataka zadatom problemu (u [36] prikazane su tri međusobno potpuno različite programske strukture podataka koje sve odgovaraju logičkoj strukturi matrice, a primenjuju se u striktno različitim situacijama). Kompleksnost rešenja se, dakle, može smanjiti ali samo *do određenih granica* definisanih očekivanim kvalitetom softvera. Dalje se ne sme ići jer, kako je

[2] Originalni, Martinov, naziv za kompleksnost softvera.

govorio Ajnštajn, "sve treba rešavati na najprostiji način, ali ne prostije!". Imajući na umu razvoj hardvera i rast zahteva korisnika u skladu sa tzv. "efektom autoputa"[3], lako zaključujemo da su mogućnosti redukcije kompleksnosti rešenja prilično limitirane.

Sve u svemu, lepeza sredstava za smanjenje složenosti softvera nije posebno široka, te ostaje da se solucija potraži na drugoj strani: treba prihvatiti da softver jeste kompleksan sistem i iznalaziti načine da se tom kompleksnošću *upravlja*. Takav opšti zaključak izvodi i Dajkstra u [7] gde navodi da "moramo organizovati proračun[4] tako da naše ograničene moći budu dovoljne da garantuju ostvarenje željenog rezultata".

Postoje bar dva, ne obavezno nezavisna, razloga zbog kojih je neki problem kompleksan. Prvi proističe iz činjenice da je gotovo svaku jedinicu posmatranja, ma šta ona bila, moguće proučavati iz više uglova posmatranja: preciznije, broj osobina gotovo svake jedinice posmatranja je maltene proizvoljan. Papir na kojem je odštampan ovaj tekst ima boju, kvalitet, format ali i specifičnu provodnost, gustinu itd. Svaki zaposleni je ličnost za sebe, sa bezbroj osobina, no sa stanovišta informacionog sistema preduzeća samo su neke od njih relevantne: npr. ime i prezime, radno mesto, kvalifikacija i sl.

Drugi razlog za kompleksnost je obim problema meren odgovarajućom merom. Ta mera može da bude, recimo, broj vrsta jedinica posmatranja ili, ako se radi o softveru, procenjena veličina softverskog rešenja izražena bajtovima koda, brojem potprograma, brojem modula... Povodom ovim Dajkstra piše: "Široko rasprostranjeno potcenjivanje poteškoća vezanih za veličinu je jedan od glavnih uzroka softverskih neuspeha".

Dva su glavna sredstva za sučeljavanje sa kompleksnošću: *apstrakcija* i *dekompozicija*[5].

Apstrakcija (apstrahovanje) je vezana, uglavnom, za proučavanje individualnih jedinica posmatranja i ima dvostruku ulogu:

1. *Izdvajanje* (isticanje) bitnog u cilju formiranja generalizovanog, idealizovanog modela jedinice posmatranja. Tako se, na primer, jedinica posmatranja STUDENT u informacionom sistemu fakulteta modeluje osobinama BROJ-INDEKSA, PREZIME, IME, GODINA-STUDIJA itd., dok se osobine tipa VISINA ili TELESNA-MASA zanemaruju. Isto tako, običan korisnik "vidi" telefon kroz njegove komande; veoma mali broj ljudi je upoznat sa detaljima izrade i funkcionisanja. Matematičar piše A^{-1} retko se upuštajući u postupak određivanja inverzne matrice.
2. *Grupisanje* tako idealizovanih modela u skupove čiji su elementi istovrsni.

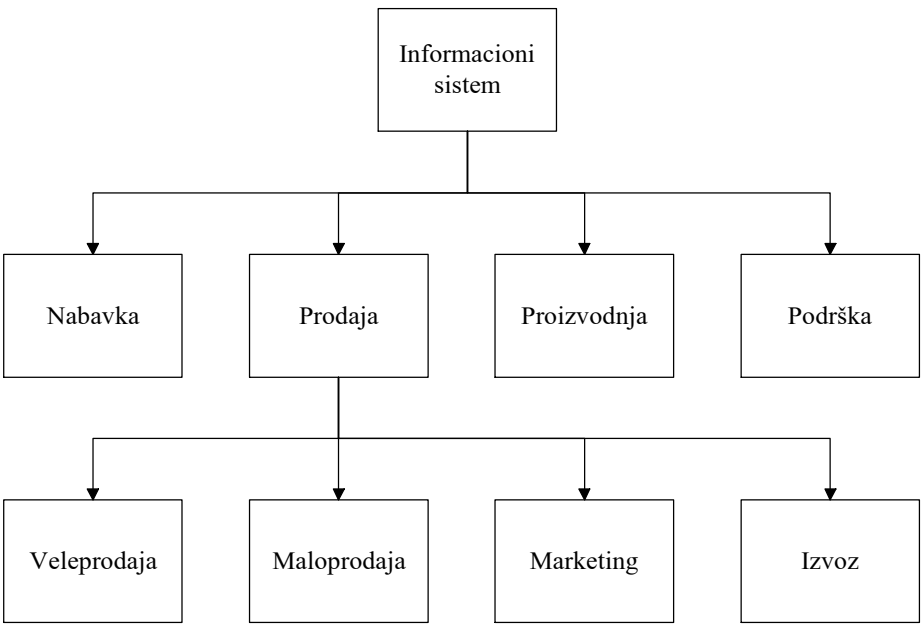
[3] Efekat autoputa: zamena starog puta između dva mesta povratno povećava frekvenciju saobraćaja.
[4] Termin je upotrebljen u širokom smislu (computation).
[5] Neki autori u sredstva za redukovanje kompleksnosti ubrajaju *nezavisnost* (modula) i *hijerarhiju*; međutim, nezavisnost je samo posledica apstrakcije, a hijerarhija jedna od vrsta dekompozicije.

Ovim mehanizmom se, prema primeru Hoara [7], od dve crvene jabuke, jedne zelene jabuke i jedne trule jabuke dolazi do skupa od četiri jabuke.

Mehanizam apstrahovanja poznat je odavno: primenjivao ga je eksplicitno još Aristotel u svom učenju o biću gde se od konkretne jedinice posmatranja, tzv. prve supstancije, postupkom opisanim napred dolazi do pojmova vrste i, na isti način, roda. Kako su vrsta i rod već nekonkretni, on ih naziva "drugom supstancijom". Aristotelovo rezonovanje lako se da proširiti i na više od 3 stupnja, primerice

1. Mustafa je BERBERSKI-LAV
2. BERBERSKI-LAV je LAV
3. LAV je MAČKA
4. MAČKA je SISAR
5. SISAR je KIČMENJAK itd.

Zapazimo, uzgred, da se i na ovoj, produženoj, lestvici apstrakcije samo prvi stupanj, "Mustafa", odnosi na konkretnu životinju, dok ostali predstavljaju manje ili više apstraktne pojmove. O definiciji apstrakcije i njenoj ulozi u procesu izrade softvera biće reči u narednim odeljcima.



Slika 1.1

Drugi osnovni mehanizam za bavljenje kompleksnošću je ***dekompozicija***. Suštinu postupka sažeo je Dajkstra u rimsku maksimu "Divide et impera", što ne

znači da su to otkrili tek Rimljani. Ako se "Divide et impera" shvati izvorno, dakle ne kao "Zavadi pa vladaj" nego kao "Podeli [državu na upravljive jedinice] pa vladaj", proizlazi da je princip nastao sa stvaranjem prvih država, a verovatno i ranije.

Dekompozicijom se složen sistem rastavlja na delove koji su jednostavniji za razumevanje. Po potrebi, delovi se mogu dalje razlagati sve dok se ne dostigne nivo detaljnosti koji omogućava neposrednu analizu i projektovanje, pri čemu je broj nivoa dekompozicije arbitraran i vezan za konkretan problem, odnosno procenu analitičara-projektanta (Buč, [3]). Neophodnost postepene dekompozicije i, u dobroj meri, način na koji se sprovodi diktirani su dobro poznatim Milerovim "Pravilom 7±2": maksimalni broj informacija koje pojedinac može simultano da obrađuje reda je veličine 7±2. Na primer, informacioni sistem nekog proizvodnog preduzeća možemo dekomponovati na podsisteme "Nabavka", "Prodaja", "Proizvodnja" i "Podrška"; podsistem "Prodaja" dekomponujemo dalje na "Veleprodaju", "Maloprodaju", "Marketing" i "Izvoz", kao što je prikazano na slici 1.1. Slično, STEREO-SISTEM dekomponujemo na RADIO, KASETOFON, CD, ZVUČNIK itd.

Dekart je bio taj koji je u filozofiju na velika vrata uveo postupak dekompozicije, kroz ideju da se, po ugledu na matematičku analizu, složen pojam razloži na prostije sastavne delove kojih [3] ima "svega nekoliko vrsta". Iz prostijih delova se indukcijom izvlače zaključci o ishodišnom pojmu.

Dekompozicija se može zasnivati na različitim kriterijumima kojih može biti i više, sve u zavisnosti od ugla posmatranja i primenjene metode. Sistem linearnih jednačina $Ax = b$ može se razložiti procesno, prema algoritmu rešavanja, na sastavne delove "Ulaz", "Određivanje rešenja", "Izlaz" i dalje. Alternativno, u sistemu se može uočiti jedinica posmatranja MATRICA i definisati operacije nad njom (ulaz, izlaz, invertovanje), zatim VEKTOR sa sopstvenim operacijama (ulaz, izlaz), kao i interakcija između MATRICE i VEKTORA (množenje). Tek potom se pristupa formulisanju solucije za dati zadatak. Dok je prvi način dekompozicije karakterističan za klasično i strukturirano programiranje, drugi je tipičan za objektno orijentisani pristup. Interesantno je primetiti da je objektni prilaz bolje usklađen sa izvornim Dekartovim principima, što je neke autore navelo da za tvorca objektno orijentisane metodologije proglase upravo njega (ni manje ni više).

1.1. KOMPOZITNO, STRUKTURIRANO I OBJEKTNO PROGRAMIRANJE

Od pojave prvih široko upotrebljavanih programskih jezika, fortrana (FORTRAN, 1954./57.) i kobola (COBOL, 1959./61.), metodologija razvoja softvera usavršavana je, kako u dubinu, poboljšavanjem postojećih pristupa, tako i u širinu, ustanovljavanjem novih. Pri tom, pojam "pristup" ne podrazumeva samo

programski jezik, nego ukupan način razmišljanja u svim fazama životnog ciklusa softvera, od analize problema, preko projektovanja, kodiranja i testiranja sve do održavanja u eksploataciji. Da bi se podvukla razlika između neposredne izrade programa i celovitog prilaza, umesto termina "programiranje" često se koristi termin "paradigma" (grčki, paradigma = primer za ugled). Tako, razlikujemo tzv. procedurnu paradigmu zasnovanu na jezicima tipa fortrana i kobola, odnosno paskala (Pascal) u strukturiranoj varijanti, zatim objektnu paradigmu (objektni paskal, C^{++}, Smalltalk), logičku paradigmu (Prolog), funkcionalnu paradigmu (Lisp) i druge. Čini se, ipak, da je umesto "paradigma" opravdanije koristiti termin ***tehnologija*** (grčki, techne = veština, logos = moć mišljenja, rasuđivanja) koji se definiše kao

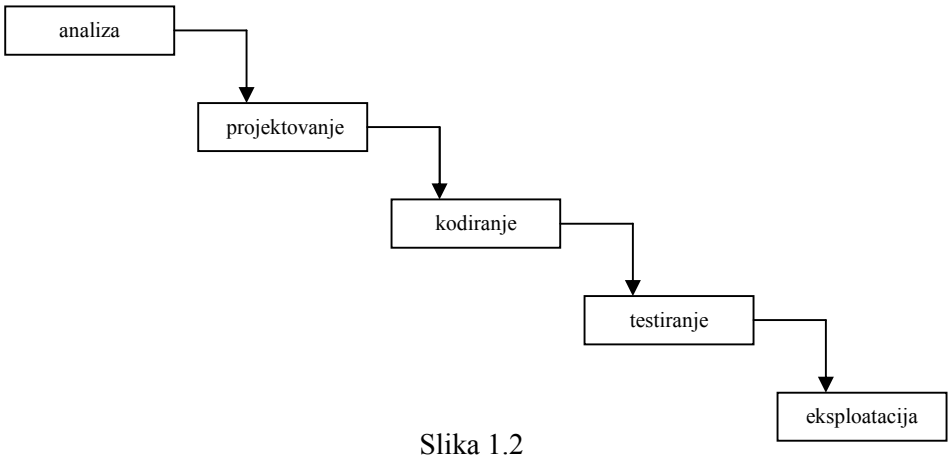
- skupni izraz za izučavanje oruđa, postupaka i metoda upotrebljavanih u raznim granama proizvodnje (prema Larousse).

Posebno, kada je u pitanju razvoj softvera, poslednjih godina intenzivno se upotrebljava termin "informaciona tehnologija" kao posebna vrsta tehnologije koja inkorporira sredstva, postupke i metode za prikupljanje, čuvanje, obradu, prenos i prezentovanje informacija. Saobrazno tome, govorimo o procedurnoj tehnologiji (klasičnoj i strukturiranoj), objektnoj tehnologiji itd. U svrhu razjašnjavanja posebnosti objektne tehnologije, od interesa je komparirati upravo je sa procedurnom (razdvajajući procedurnu na klasičnu ili, kako je Buč naziva, *kompozitnu* i strukturiranu) i to zbog bliskosti odgovarajućih domena problema. Pored toga, one se hronološki nadovezuju jedna na drugu, uz napomenu da pod hronološkim redosledom ne podrazumevamo samo redosled nastanka nego i period u kojem je svaka od njih bila dominantna. Uopšte uzev, kompozitna tehnologija dominirala je šezdesetih, strukturirana sedamdesetih i delom osamdesetih da bi, sve do danas, vodeću ulogu preuzela objektna tehnologija.

Životni ciklus softverskog proizvoda, definisan kao niz aktivnosti u periodu između donošenja odluke o izradi i povlačenja sa tržišta obuhvata, u ovoj ili onoj formi, faze analize, projektovanja, realizacije (kodiranja i testiranja) i eksploatacije. Imajući ovo u vidu mogu se identifikovati glavne komponente pomenutih triju tehnologija:

1. Kompozitna tehnologija: više različitih metoda analize i projektovanja sistema (npr. ADS, SOP, ISDOS), blok dijagrami algoritama i sistemski dijagrami, programski jezici tipa fortrana i kobola, tehnika dekompozicije na potprograme.
2. Strukturirana tehnologija: Strukturirana sistem analiza Jordona i De Marka (Yourdon, De Marco, [6]), strukturni dijagrami, tehnika hijerarhijske dekompozicije (top-down decomposition), programski jezici tipa paskala.
3. Objektna tehnologija: metode analize i projektovanja Jakobsona, Rumbaa i Buča (Jacobson, Rumbaugh, Booch, [2], [3], [24]) kasnije objedinjene u UML (Unified Modeling Language - Objedinjeni jezik modelovanja), objektni paskal, C^{++}, smoltok (Smalltalk), java.

Interesantno je zapaziti da objektni pristup ima i sve osobine metodologije [58]. Buč [3] pod *metodom* podrazumeva "disciplinovan proces generisanja modela koji opisuju razne aspekte softvera u izgradnji". Kada je u pitanju objektni prilaz takve metode ne samo da postoje, nego ih ima čitav niz: Bučova metoda, Rumbaova metoda OMT, Yin-Tanik-ova metoda, JSD metoda (Birchenough i Cameron) i metode iz UML (Unified Modeling Language Buča, Rumbaa i Jakobsona). ***Metodologiju*** Buč definiše kao "kolekciju metoda koje se primenjuju tokom životnog ciklusa softvera i koje su povezane zajedničkom filozofijom". Nisu sve od nave-



Slika 1.2

denih metoda nivoa metodologije; većina ne pokriva ceo životni ciklus, nego uglavnom fazu projektovanja. Da bismo ustanovili da li i koji od objektnih stilova može biti prihvaćen za metodologiju moramo prvo ustanoviti šta sadrži tipični model životnog ciklusa softvera tako da se može proveriti prekriva li sve faze u životnom ciklusu ili ne. Postoji nekoliko modela životnog ciklusa među kojima je, mada ne spada u najmodernije, u praksi još uvek veoma zastupljen tzv. *kaskadni ili vodopadni model* ("Waterfall Model"). I ovaj model ima više varijanata, no kako to nema uticaja na analizu odlučićemo se za osnovnu varijantu datu na slici 1.2. Sa slike se vidi da se faze mogu sažeti u četiri: analiza, projektovanje, implementacija (kodiranje sa testiranjem) i eksploatacija. Jedan od stilova koji konsekventno, dakle na bazi zajedničke filozofije, prekriva čitav životni ciklus softvera je UML:

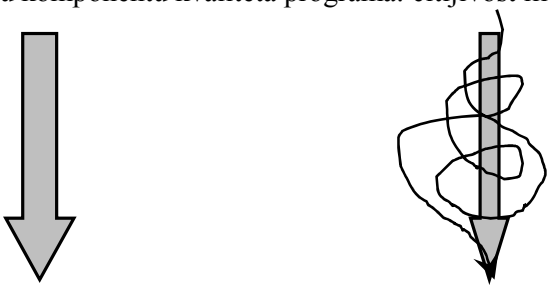
- *Analiza* se sprovodi metodom slučajeva korišćenja ("Use Case Analysis").
- *Projektuje* se upotrebom familije dijagrama koji uključuju kako statičke tako i dinamičke aspekte sistema, čak do nivoa projektovanja hardverske podrške.
- *Implementacija* nije propisana metodologijom (što je dobro), ali je ostavljen prostor za primenu bilo kojeg programskog jezika ili objektno

orijentisanog softverskog alata. Testiranje objektno orijentisanog softvera obavlja se dobro definisanim postupcima.

- Konačno, najvažnija aktivnost u fazi *eksploatacije* - modifikacija - ne samo što je obuhvaćena metodologijom, nego je rešenje tog problema, mogućnost tzv. višekratne upotrebe ("reusability") softvera, u samoj osnovi objektnog pristupa.

Osnovna odlika kompozitne tehnologije-metodologije bila je nekonsekventnost koja se ogledala u mnoštvu različitih sredstava i pristupa. Da bi se koliko-toliko uveo neki red, problemi i odgovarajuće aplikacije deljeni su na dve, međusobno disjunktne, grupe: tzv. naučne (scientific) aplikacije realizovane na fortranu i tzv. poslovne (business) aplikacije za čiju se izradu koristio kobol. Iako razdvojeni po domenu problema, softverski sistemi imali su jednu zajedničku karakteristiku - intenzivno korišćenje naredbi skoka po čemu se naročito "proslavio" fortran, a ni BASIC (Kemeny i Kurz, 1965.) nije bio ništa bolji.

Ovde moramo napomenuti da nije slučajno što se pri analizi kompozitne tehnologije (a i ostalih) vezujemo ponajviše za programski jezik jer, prema Straustrupu [29]: "Na programskom jeziku razmišljamo-programiramo. Dakle, programski jezik predstavlja i sredstvo pomoću kojeg prezentujemo računaru posao koji treba da uradi, ali i skup koncepata koji se koriste kada se razmišlja o rešenju datog problema". Prema tome, programski jezik nije puki alat što će biti upotrebljen da se računaru zada posao; on uslovljava način razmišljanja u fazi realizacije, pa i pre nje. Način razmišljanja diktiran fortranom i kobolom doveo je do prve velike "softverske krize šezdesetih" izazvane nesaglasjem mogućnosti hardvera i zahteva korisnika s jedne, i ograničenog dometa softverske tehnologije s druge strane[6]. Tačnu dijagnozu postavio je Dajkstra u čuvenom članku "Goto Statement Considered Harmful" iz 1968. godine, u kojem je utvrdio da je broj grešaka u programu direktno srazmeran broju programskih skokova čime je, pored brzine i memorijskih zahteva, uveo i treću komponentu kvaliteta programa: čitljivost ili razumljivost.



fizički redosled naredbi trasa izvršavanja

Slika 1.3

[6] Radi se o već pominjanom efektu autoputa.

Dajkstrini zaključci pokrenuli su lavinu kritika na račun fortrana[7]. Kompozitno programiranje nazvano je "haotičnim", a rezultujući kod dobio je prilično pežorativno ime "špageti kod" (za razlog videti sliku 1.3). Sam Dajkstra je, sa osećajem za meru, ovakav kod nazvao "baroknim" [31]. Pojavilo se proročanstvo da u kratkom roku fortran silazi sa scene. Proročanstvo još čeka na ispunjenje.

Kompozitno programiranje prevaziđeno je iz jednostavnog razloga što tehnologija izrade programa - koja je *postojala* - u jednom trenutku više nije bila u stanju da prati zahteve tržišta. Straustrup [29] primećuje: "Jezik obezbeđuje programeru skup konceptualnih alata; ako oni ne odgovaraju datom poslu, jednostavno se ignorišu". Povećane mogućnosti hardvera su, efektom autoputa, generisale nove apetite korisnika, koji više nisu mogli biti ispunjeni postojećim softverskim sredstvima. Nema zanatlije, ma kako bio vešt, koji je u stanju da, njemu raspoloživim sredstvima, napravi svemirsku stanicu ... Drugim rečima, u Martinovoj jednačini (1.1)

$$c_a = c_p + c_r$$

važi $c_r=f(c_p)$, to jest kompleksnost rešenja zavisi od kompleksnosti problema. Pri tom, za zadati problem kompleksnost rešenja ima donju granicu, bez obzira na hardversku komponentu tehnologije. Kada ta donja granica postane suviše visoka mora se menjati softverska tehnologija i to je pravi uzrok svih "softverskih revolucija"; u tom pogledu ništa se nije promenilo, sem što su danas izmene u tehnologiji izrade softvera znatno učestalije.

Strukturirano programiranje, koje je tokom sedamdesetih preraslo u tehnologiju, najstariji je (uspešan) pokušaj da se postupak razvoja softvera unifikuje, tj. da se programiranje fundira kao inženjerska delatnost. Počeci strukturiranog programiranja sežu negde u drugu polovinu šezdesetih godina kada je formulisana Strukturna teorema Bema i Jakopinija (C. Boehm i G. Jacopini) godine 1966. i kada je Dajkstra objavio pomenutu raspravu "Goto Statement Considered Harmful" u *Communications of the ACM* godine 1968. Pri tom, od posebnog interesa je upravo ovo poslednje, jer je Dajkstra ukazao na glavni razlog neupravljivosti složenih softverskih paketa - skokove. Negativan uticaj udela naredbi skoka u programu uslovljen je činjenicom da prilikom rešavanja bilo kojeg problema čovek rezonuje sistematski, dakle rešava ga sekvencijalno, korak po korak, a ne haotično, preskačući sa jednog potproblema na drugi i vraćajući se na ranije započeto.

Osnovne postavke i sredstva strukturiranog programiranja bili su definisani relativno brzo, naročito radovima Dajkstre, Hoara (C.A.R. Hoare) i Dala (O.J. Dahl) (knjiga *Structured Programming* iz 1972. po kojoj je čitava metodologija

[7] Ali tek iz drugog pokušaja. Članak je prvo odbijen. Objavljen je u konkurentskom časopisu.

dobila ime), kao i pojavom programskog jezika paskal Niklausa Virta i saradnika 1969. godine. Metode i tehnike koje čine strukturirano programiranje intenzivno su se razvijale i u narednih deset do petnaest godina predstavljale su jedinu relevantnu tehnologiju proizvodnje softvera. U poslednjih petnaestak godina strukturirano programiranje nije više dominirajuća tehnologija prepustivši primat novijim pristupima - konkretno objektnom. Iz toga se, međutim, ne sme izvući zaključak da je strukturirana tehnologija zastarela (programerski svet je pomalo sklon preterivanju), jer način razmišljanja prilikom izrade programa - od opšteg ka pojedinačnom - nije se promenio niti je u izgledu da se to desi u skoroj budućnosti. Takođe, upravljačke strukture koje se koriste u objektnom programiranju (sekvence, selekcije, iteracije) su iste one koje su definisane u strukturiranom prilazu.

Strukturirano programiranje bilo je čvrsto vezano za tehniku razvoja programa tzv. sukcesivnom (hijerarhijskom) dekompozicijom (top-down program development), koja se izvodi postepenom detaljizacijom programa od grube šeme do nivoa spremnog za kompilaciju. Ova veza bila je toliko čvrsta da se tehnika sukcesivne dekompozicije izjednačavala sa metodologijom u celini, što je pogrešno, naročito u novije vreme kada sama tehnika više ne predstavlja osnovu za izgradnju softverskih sistema, dok je strukturirani način razmišljanja i dalje u potpunosti aktuelan.

Velika popularnost tehnike sukcesivne dekompozicije ponešto je zamaglila druge - kasnije se pokazalo mnogo značajnije - doprinose strukturirane tehnologije-metodologije. Tu se, pre svega, misli na promociju celovite *teorije programiranja*, čime je učinjen odsudan iskorak iz područja veštine u područje nauke. Pored teorije sintakse razrađene još pedesetih i šezdesetih u radovima Čomskog, Bekusa i Naura (N. Chomsky, J. Backus, P. Naur), oformljeni su teorijski osnovi i razvijeni brojni modeli semantike programskih jezika, čemu su bitan doprinos dali Knut (D. Knuth), Dajkstra i Hoar (videti npr. [31] ili [7]).

Takođe, uočen je značaj struktura (tipova) podataka koje čine nedeljivu celinu sa algoritmom, što je efektno formulisao Virt dajući jednoj od svojih knjiga - danas klasičnoj - naslov "Algorithms + Data Structures = Programs". Virt je utvrdio da su, kao osnovne komponente programa, struktura podataka i algoritam ravnopravni, pa čak i da je struktura podataka u izvesnoj prednosti, jer je stabilniji deo programa. Pored toga, algoritam operiše nad strukturom podataka, a ne obrnuto. Ako više programskih jedinica radi na jednoj strukturi podataka, tada izmena u nekoj od njih neće imati mnogo uticaja na ostale, dok će modifikacija strukture podataka izazvati promene u mnogim, a najverovatnije svim, jedinicama. Kako za oblast semantike naredbi, tako i za strukture (tipove) podataka razvijena je konsistentna, algebarski zasnovana, teorija apstraktnih tipova podataka. Nije bez značaja činjenica da je koncept apstraktnog tipa podataka blizak konceptu objekta (tačnije konceptu tzv. klase objekata), tako da ih Virt čak naziva "starijim" i "novijim" konceptom. I još nešto, veoma važno: upravo insistiranje na strukturi podataka kao

neizdvojivoj i ravnopravnoj komponenti programa izazvalo je prve naprsline u strukturiranoj tehnologiji, te direktno dovelo do stvaranja nove: objektne.

Strukturirano programiranje je definitivno skinulo s dnevnog reda pitanje programskih skokova. Strukturnom teoremom utvrđeno je da se praktično svi moduli (izuzimajući egzotične, sa više ulaznih tačaka ili sa beskonačnim ciklusima i sl.) mogu realizovati superpozicijom samo tri upravljačke strukture: sekvence (u paskalu begin-end), selekcije tipa if-then-else i ciklusa while-do, dakle bez upotrebe bilo koje naredbe skoka. Tehnika sukcesivne dekompozicije i odgovarajući programski jezici (paskal pre svih) omogućili su da se program piše *a priori* bez skokova, tako da se ne zahtevaju naknadne intervencije u svrhu njihovog eliminisanja. Trebalo je da prođe gotovo decenija i po da se proskribovani skokovi rehabilituju, ali u vrlo skromnom obliku iskoka iz ciklusa (*break*, *continue*), iskoka iz potprograma odn. završetka programa (*exit*) i krajnje limitirane primene *goto* na iskok iz skupa uklopljenih ciklusa. Činilo se da su dveri industrijske proizvodnje softvera širom otvorene!

Nažalost, ispostavilo se da situacija nije bila baš tako idilična. Naime, epohalno otkriće da su struktura podataka i algoritam nedeljivi (najbliže stvarnosti je reći da se *prožimaju*) neumitno je postavilo zahtev da metodologija projektovanja i izrade softvera obuhvati oba aspekta i to *ravnopravno* i *istovremeno*. S druge strane, tehnika sukcesivne dekompozicije u potpunosti je usmerena na algoritam, iako je Hoar još u "Strukturiranom programiranju" uočio ovaj problem i pokušao da ga reši. Uspeh je izostao jer rešenje nije ni moglo da se nađe u okvirima sukcesivne dekompozicije algoritma.

Iz neusaglašenosti logičkog prožimanja algoritma i strukture podataka s jedne strane, odnosno njihove fizičke razdvojenosti na "deklaracioni" i "izvršni" deo s druge, proistekla je većina nedostataka strukturiranog programiranja. U najvažnije ubrajamo [1]:

- problem kompatibilnosti
- problem kontinuiteta i
- problem višekratne upotrebe softvera (engl. software reusability).

Problem kompatibilnosti je direktna posledica algoritamske orijentisanosti strukturirane metodologije, tj. fizičke razdvojenosti algoritma i strukture podataka. Uopšte uzev, primena tehnike sukcesivne hijerarhijske dekompozicije može rezultovati modulima što operišu nad istom strukturom podataka koja je u pojedinim modulima različito realizovana, naročito ako se rad obavlja timski i ako zakaže koordinacija. Primer (ekstreman) je program za rad sa retkim matricama čiji moduli nemaju jednaku fizičku realizaciju takvih matrica. Problem kompatibilnosti nije ni malo bezazlen jer usklađivanje nekompatibilnih struktura podataka u principu zahteva ponovnu izradu nekih ili svih modula.

Problem kontinuiteta vezan je za modifikaciju programa, a posebno za proširenje njegove funkcionalnosti. Pre svega, vrlo je pogrešno shvatiti program

statički, kao nešto što se radi jednom za svagda. Naprotiv, programi podležu čestim izmenama i dopunama produkujući tokom životnog ciklusa niz verzija i podverzija. Softver realizovan sukcesivnom dekompozicijom teško je modifikovati (naročito dodavanjem funkcija) iz najmanje dva razloga:

- Zbog algoritamske usmerenosti tehnike gotovo je nemoguće izvršiti ozbiljnije izmene u strukturi podataka.
- Dodavanje novih funkcija otpočinje najčešće na visokim hijerarhijskim nivoima te zahteva dugotrajan, često mukotrpan, postupak za modifikovanje hijerarhije pre nego što se stigne do izvršnog koda.

Problem višekratne upotrebe softvera je, možda, najbolje poznat od svih. Radi se o najnormalnijoj težnji da se jednom napisan i testiran softver, u celini ili delimično, koristi u različitim projektima, dakle kad god se pojavi potreba za njim. Strogo teorijski, strukturirani program (pisan, recimo, na klasičnom paskalu) realizuje se kao monolit sastavljen od glavnog programa i potprograma fizički uklopljenih u njega. To, pak, ima za posledicu da se ni jedan od potprograma ne može koristiti u nekom drugom programu ako se ne raspolaže izvornim kodom[8]. Inače, hronološki starija kompozitna tehnologija ovaj problem imala je rešen posredstvom biblioteka gotovih potprograma. Ne čudi, stoga, što su već prve operativne verzije strukturiranih programskih jezika imale mogućnost modularizacije, iako to teorijski nije sasvim u skladu sa izvornim postupkom sukcesivne dekompozicije. Međutim, ni prilično moćan mehanizam modula u paskalu ili C-u ne nudi sveobuhvatno rešenje navedenog problema jer preuzimanje gotovog softvera počesto ide uz delimičnu modifikaciju, za šta moduli pružaju relativno ograničene mogućnosti. Tek u kombinaciji sa objektima, a kao sastavni deo objektne tehnologije, moduli će pokazati pravu upotrebnu vrednost.

Pored tri pomenuta problema, nije na odmet istaći još jedan: problem startovanja sukcesivne dekompozicije. Naime, originalna tehnika podrazumeva da dekompozicija otpočne kratkim tekstom datim u obliku komentara koji opisuje šta program treba da radi i koji je osnova za dekomponovanje. Kako, međutim, opisati kratkim tekstom, pogodnim za dekompoziciju, operativni sistem? Kako opisati moderne multifunkcionalne programe kao što su programi za obradu teksta, slika ili tabela? Algoritamska dekompozicija ne nudi odgovor na ova pitanja.

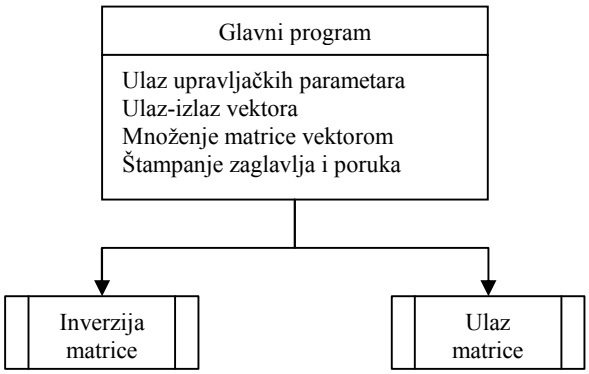
Primer 1.1. Da bismo ilustrovali sličnosti i razlike između kompozitnog i strukturiranog programiranja i istovremeno prikazali osnovnu ideju objektnog pristupa razmotrićemo jednostavan zadatak:

- Odrediti rešenje sistema linearnih jednačina Ax = b reda n, koristeći inverziju matrice A.

Klasično kompozitno programiranje ne predviđa sistematizovan postupak (tj. algo-

[8] Može se samo zamisliti sudbina grafičkog korisničkog interfejsa koji se svaki put pravi iznova!

ritam) za rešavanje problema, tako da su programi raznih autora imali malo toga zajedničkog. Sredstvo za dekompoziciju bio je potprogram kojeg je prosečni programer često shvatao ne kao logičku celinu (dakle kao jednu vrstu apstrakcije) nego prvenstveno kao način za skraćenje izvornog koda - segment koji se pojavljuje u programu na više mesta izdvaja se u potprogram. S obzirom na (pre)veliku slobodu u kreiranju arhitekture programa ne može se unapred reći kako bi ona izgledala, tj. koji bi delovi bili realizovani u vidu potprograma i kakav bi bio redosled izrade. Realno je, ipak, pretpostaviti da bi iskusan programer izdvojio komplikovanije delove - a to su, u ovom slučaju, inverzija i ulaz matrice - u potprograme, najviše stoga što mogu zatrebati i u drugim aplikacijama. Jednostavniji segmenti kao što su ulaz-izlaz vektora, množenje matrice vektorom, a takođe i naredbe za ulaz upravljačkih podataka, štampanje raznih zaglavlja i poruka mogli bi biti smešteni u glavni program. Ilustracija je, inače, bazirana na soluciji datoj u zbirci potprograma SSP (Scientific Subroutine Package, [99]) firme IBM. Rezultujuća arhitektura prikazana je na slici 1.4.



Slika 1.4

Strukturirani pristup se bitno razlikuje od prethodnog po tome što se dekompozicija algortima sprovodi *sistematski*. Polazi se od opšteg opisa zadatka, datog u vidu slobodnog, ali u najvećoj meri preciznog, teksta[9]:

> 0. Odrediti rešenje sistema linearnih jednačina Ax = b reda n (n je najviše ...), koristeći inverziju matrice.

U prvom koraku zadatak se dekomponuje na osnovne podzadatke koje je, u ovom primeru, lako definisati:

> 1. Ulaz: A, b

[9] Formalna tehnika predviđa da tekst ima sintaksnu formu komentara.

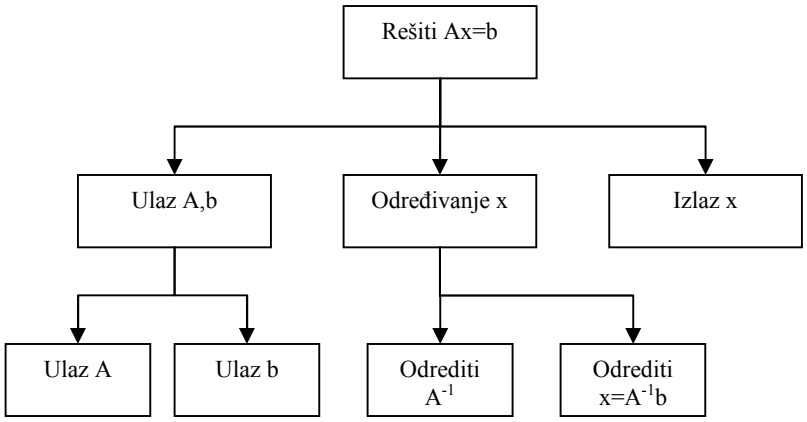
2. Određivanje rešenja x
3. Izlaz x.

U nastavku podzadaci se dekomponuju na isti način i to posebno i nezavisno jedan od drugog:

1.1 Ulaz A
1.2 Ulaz b

2.1 Invertovanje A
2.2 Množenje invertovane matrice A sa b i dodela vektoru x

Postupak se teorijski završava kada se dekompozicijom stigne do naredbe na ciljnom programskom jeziku. Praksa se, kao i obično, ponešto razlikovala. Naime, odluku o završetku dekompozicije na pojedinačnim granama hijerarhije donosio je programer kada zaključi da je rezultujući tekst sam po sebi jasan i da se može neposredno realizovati kao programska celina. Postupak dekompozicije prikazan je na slici 1.5.



Slika 1.5

Objektni programer sasvim drugačije pristupa rešavanju ovog zadatka. Pre svega, a kako podvlači Mejer [1], on se koncentriše ne na analizu rešenja već na analizu problema, zadržavajući se što je moguće duže u domenu problema. Cilj analize je identifikovanje elemenata (jedinica posmatranja) i njihovih veza u domenu problema. Potom se pristupa modelovanju pojedinačnih elemenata i veza da bi se traženi program, po pravilu algoritamski kratak, realizovao tek na kraju. Drugim rečima, objektno orijentisani postupak, za razliku od strukturiranog, ne polazi od pitanja "šta program treba da radi" nego se njime faktički završava.

Jedinice posmatranja modeluju se grupisanjem u tzv. klase i opisom klasa kroz naziv, bitne osobine i repertoar operacija primenljiv na primerke klase koji nose naziv *objekti*.

U domenu problema rešavanja sistema jednačina identifikacija klasa je jednostavan posao što, podvucimo to odmah, predstavlja izuzetak, a ne pravilo. Radi se o klasama MATRICA i VEKTOR. Njihov opis, a i realizacija, izgledali bi približno ovako:

Naziv:	MATRICA
Podaci:	m, n (broj vrsta-kolona)
	P (elementi)
Operacije:	Učitati
	Prikazati
	ZadatiElement(i, j, vrednost)
	OčitatiElement(i, j, vrednost)
	Invertovati
	Transponovati
	itd.

Naziv:	VEKTOR
Podaci:	k (broj elemenata)
	V (elementi)
Operacije:	Učitati
	Prikazati
	ZadatiElement(i, vrednost)
	OčitatiElement(i, vrednost)
	Moduo
	itd.

Množenje matrice vektorom modelovalo bi se kao operacija u jednoj od klasa, u obema ili u formi tzv. slobodne operacije (slobodnog potprograma). Odlučićemo se, ilustracije radi, za poslednju varijantu:

Pomnožiti(Matrica M, VEKTOR v, VEKTOR rezultat)

Na kraju, dakle posle modelovanja i izrade odgovarajućeg softvera, pristupa se rešavanju zadatka jednostavnim programom sledećeg oblika:

MATRICA A
VEKTOR b, x

```
A.Učitati
b.Učitati
A.Invertovati
Pomnožiti(A, b, x)
x.Prikazati
```

gde je konstrukt $\alpha.op$ uobičajena oznaka za primenu operacije *op* na primerak klase (objekat) α.

Pažljiviji čitalac će odmah prokomentarisati: "Šta se dobija ovakvim pristupom? Umesto da odmah rešimo sistem jednačina, trošimo vreme na realizaciju svake klase posebno uz obilje nepotrebnih operacija poput transpozicije i prikazivanja matrice ili određivanja modula vektora!". Odgovor na to je da se klasa realizuje *jedan jedini put*, a zatim u *prevedenom obliku*, dakle bez korišćenja izvornog koda, koristi gde god je to potrebno, čak uz mogućnost modifikacije i proširivanja, opet bez intervencija u izvornom kodu. Prvo - i najvažnije - što se primećuje je činjenica

- da objekat predstavlja fizičku celinu strukture podataka i algorit(a)ma datih u vidu operacija,

te je, samim tim, problem kompeticije između strukture podataka i algoritma rešen tako što je jednostavno skinut s dnevnog reda. Svaki objekat predstavlja celinu za sebe i ima jasnu logičku barijeru koja ga odvaja od drugih objekata, kako u pogledu strukture podataka tako i u pogledu algoritma.

Samim tim i problem kompatibilnosti nestaje jer ne postoji nekakva zajednička struktura podataka koja bi stvarala nekompatibilnost.

Problemi kontinuiteta i višekratne upotrebe softvera rešavaju se uz pomoć posebnog mehanizma, tzv. nasleđivanja, koje omogućuje da se objekat menja i proširuje bez upotrebe izvornog koda. O nasleđivanju će biti dosta reči u sledećim poglavljima.

1.2. KRATAK OSVRT NA ISTORIJSKI RAZVOJ OBJEKTNE METODOLOGIJE

Hronološki sled formiranja kompozitne i strukturirane metodologije kretao se linijom implementacija - projektovanje - analiza, dakle suprotno od redosleda u životnom ciklusu. Razlog nije teško otkriti: programiranje je najodređenija delatnost od navedene tri; projektovanje već ima više stepeni slobode, dok je analiza sistema još uvek prevashodno veština. Objektna metodologija prošla je kroz *identične* razvojne faze i to *istim redosledom*. Prema tome, istorijski razvoj softverskih tehnologija - metodologija treba pratiti kroz razvoj osnovnih sredstava - programskih jezika.

Procedurni programski jezici, prethodnici objektnih, mogu se hronološki

razvrstati u 3 generacije (prema [3], uz dopune):

- Jezici prve generacije, 1954.-1958. (FORTRAN I): obrada matematičkih izraza.
- Jezici druge generacije, 1959.-1961. (FORTRAN II, Algol 60, COBOL): potprogrami, blokovi, upravljanje datotekama.
- Jezici treće generacije, 1962.-1970. (Pascal): moduli, složeni tipovi podataka.

Sedamdesete godine čine, kako tvrdi Buč, "generacijski jaz" (generation gap) karakterisan pojavom mnogih neuspelih programskih jezika ali, dodajmo, i konsolidacijom prikupljenog znanja i iskustva u ovoj oblasti. U toku tog perioda zaokružene su doslovce *sve* osnovne koncepcije koje će biti implementirane u objektnoj tehnologiji, od apstrakcije i modularizacije pa do nasleđivanja.

Sama ideja o objektu kao fizičkom spoju strukture podataka i pripadajućih operacija bila je ugrađena u programski jezik simula-67 još polovinom šezdesetih godina, te zato mnogi autori dovode ovaj jezik u vezu sa početkom objektnog programiranja. Simula-67 bio je, doduše, zamišljen kao programski jezik opšte namene, ali je prvenstveno služio kao sredstvo za pravljenje softvera iz oblasti simulacije diskretnih sistema. Njegova koncepcija iskorišćena je kao temelj na kojem su Alen Kej (Alan Kay) i saradnici na samom kraju sedamdesetih izgradili jezik smoltok (Smalltalk) koji i danas spada među popularne objektne programske jezike. Treba, međutim, podvući da je tokom sedamdesetih objektna orijentacija bila na sporednoj liniji razvoja softverskih tehnologija, duboko u senci strukturiranog programiranja, zato što je za njenu široku primenu bilo jednostavno *prerano*. Sličnu sudbinu imala je Bebidžova (John Babbage) računska mašina iz XIX veka, pa donekle i Tjuringova (Alan Turing) koncepcija apstraktnog računara iz tridesetih godina XX veka.

Širenje personalnih računara i prateće softverske industrije, kao i promena profila korisnika, od programera ka neprofesionalcu, dovela je do usvajanja objektne orijentacije za glavnu softversku metodologiju-tehnologiju. Otprilike u isto vreme kad i smoltok, dakle početkom osamdesetih godina, pojavljuju se drugi objektno orijentisani jezici: C^{++} (Bjarne Stroustrup), objektni paskal (tačnije Turbo Pascal 5.5 firme Borland) i ada (Ada, po imenu prvog programera, Ade lady Lovelace, inače Bajronove kćerke) koji su i danas vodeći. Razlika između smoltoka s jedne i ostala tri jezika s druge strane je u tome što je smoltok čist objektni jezik u kojem nema ničeg osim objekata[10], dok C^{++}, objektni paskal i ada predstavljaju hibridne jezike jer imaju i neobjektnu komponentu, zato što su direktno izvedeni iz odgovarajućih procedurnih jezika.

Tokom devedesetih godina, naročito kroz radove Buča, Rumbaa, Jakobsona i Šoove razrađene su metode za objektno orijentisano projektovanje i sistem anal-

[10] Čak se i konstante tretiraju kao (konstantni) objekti.

izu, čime je objektna orijentacija dostigla nivo metodologije. Ekspanzija računarskih mreža, naročito interneta, ispostavila je programerima nove zahteve. Naime, mreže povezuju međusobno vrlo različite računare - operativne sisteme ("platforme"), a to je u prvi plan izbacilo problem portabilnosti programa. Program u prevedenoj (mašinskoj) formi postao je za mrežni ambijent neupotrebljiv, jer nije mogao biti izvršen na bilo kojoj platformi u mreži. Ovo je uslovilo prodor novog objektnog jezika nazvanog java (Java) napravljenog u firmi Sun Microsystems i zvanično predstavljenog 1995. godine. Java je izvedena iz C^{++} sa generičkom idejom da se program pisan na tom jeziku može izvršiti na proizvoljnoj platformi (takvi jezici se nazivaju "platform neutral" ili "platform independent").

2. OSNOVNI POJMOVI I TERMINI

Prvi korak pri izučavanju neke stručne ili naučne oblasti jeste upoznavanje sa osnovnim pojmovima i terminima te njihovo povezivanje definicijama. Definisati fundamentalne pojmove u određenoj disciplini nikad nije lako, a naročito ne u situacijama kada se nova oblast razvija na više mesta simultano. Drugim rečima, najmanji problem čine one definicije koje formuliše jedna osoba ili grupa naučnika pri inaugurisanju sasvim novog poglavlja u nekoj naučnoj oblasti. Tada autori imaju punu slobodu izbora naziva i kreiranja definicija, tako da novoizgrađeni pojmovni sistem, pod uslovom da je konsistentan, ima prirodu *norme* koja se predlaže ili uspostavlja. Definicije koje spadaju u ovu klasu nose naziv normativne ili postulacione definicije sa glavnom karakteristikom da "ne konstatuju postojeću, već propisuju buduću upotrebu", [9]. Definicije u matematici mahom su normativne.

Nažalost, objektna paradigma nije uvedena jednim potezom nego je nastala evolutivno, kao rezultat postepenog povećavanja zahteva s jedne i znanja s druge strane. Shodno tome, u toku izgrađivanja teorije i prakse uspostavilo se faktičko stanje u kojem egzistiraju slični - ali nikako identični - pogledi na središnje koncepte objektne metodologije: objekat i klasu objekata. To povlači za sobom nužnost usvajanja tzv. empirijske ili leksičke definicije [9]. Empirijska definicija dobija se prikupljanjem činjenica i mišljenja o pojmu koji se definiše (u ovom slučaju objekat), njihovim sučeljavanjem te izvlačenjem zajedničkih karakteristika. Formulisanje empirijske definicije nije jednostavno baš stoga što mora da uvaži zatečeno stanje, u našem slučaju posebno zato što se radi o pojmu što ga mnogi "definišu" sa "zna se šta je to". Pojam *algoritma* tipičan je primer: toliko je poznat da ne postoji normativna definicija, a Uspenski čak predlaže da se uopšte ne definiše (poput tačke, prave ili ravni u geometriji).

Ni sa terminologijom stvar ne stoji ništa bolje. Za iste - i to fundamentalne - pojmove postoje po dva, po tri naziva. Već sam objekat naziva se i instancom klase. Delovi objekta zovu se polja, podaci-članovi ili atributi[11]; delovi objekta koji su i sami objekti zovu se objekti-članovi, ali i polja (opet). Operacije nose naziv

[11] Posebno je nejasan termin "atribut", videti dalje..

metode ili funkcije-članice. Termini "bazna klasa", "predak", "roditeljska klasa", "natklasa" i "klasa-davalac(!)" odnose se na isti pojam u jednoj važnoj relaciji koja se, sa svoje strane, zove "nasleđivanje", ali i "izvođenje".

Pored sinonimije, objektni terminosistem boluje i od tzv. motivisanih termina, tj. termina koji su poznati i izvan njega, a mogu da izazovu pogrešnu asocijaciju. Nažalost, eklatantan primer je upravo termin "objekat", koren naziva čitave metodologije. Uobičajeno značenje reči *objekat* ili *objekt*[12], "stvar, predmet, korelat od subjekat", ne odgovara u potpunosti semantici koju ima u objektnoj metodologiji. Izraz "nasleđivanje" je preterano specifičan i niukoliko ne ukazuje na to da se radi o fundamentalnom pojmu koji daleko prelazi granice ne samo objektne metodologije nego i računarstva uopšte.

2.1. OBJEKAT I KLASA

Za objekat kao jedan od dva centralna koncepta (drugi je klasa objekata), u objektnoj metodologiji postoji obilje definicija od kojih ćemo izložiti samo najpoznatije. Pre toga - a da bismo bili u mogućnosti da analizujemo definicije - moramo se pozabaviti definicijom kao takvom. Prema [57] "Definicija (lat. definitio = određenje) je logički postupak kojim se određuje odn. utvrđuje sadržaj nekog pojma". Marković u [9] daje nešto precizniju definiciju definicije:

- Definicija je svaki onaj iskaz ljudskog jezika u kojem se značenje jednog jezičkog simbola objašnjava pomoću skupa drugih jezičkih simbola".

Jezički simbol koji se definiše zove se *definiendum*, a skup jezičkih simbola koji ga objašnjavaju nosi zajednički naziv *definiens*. U definiciji

◊ preslikavanje iz skupa A u skup B je podskup skupa Dekartovog proizvoda A i B u kojem su prve komponente uređenih parova obavezno različite

definiendum je "preslikavanje", dok definiens čine jezički simboli "skup", "Dekartov proizvod", "uređeni par" i "(prva) komponenta". Jedna od esencijalnih osobina definicije je njena nekreativnost [59] koja zahteva da se definiendum u potpunosti izvodi iz definiensa. Pored toga, definicija mora biti jasna [57] što znači prvo da su svi članovi definiensa ili unapred definisani ili, bar, intuitivno spoznatljivi i drugo da je odnos definiendum - definiens precizan, tj. nedvosmislen.

U definiciji preslikavanja, naročito ako se izrazi jezikom predikatskog računa svi zahtevi su ispunjeni. "Definicija" *čovek je živo biće* čini se jasnom, ali je neprecizna jer pored čoveka postoje i druga živa bića, te se, dakle, uopšte ne radi o

[12] "Objekt" i "objekat" su gramatički pravilni oblici. Izraz je nastao od latinskog *objectum* = nešto što je nasuprot, suprotstavljeno [7].

definiciji nego samo o iskazu.

U definiens ulaze [57] (1) tzv. *genus proximum* što označava najbliži viši pojam po rodu i (2) razlika u odnosu na druge pojmove iz roda (tzv. *differentia specifica*). U definiciji preslikavanja genus proximum je "podskup Dekartovog proizvoda (skupova)", a razlika kojom se preslikavanje odvaja od ostalih podskupova Dekartovog proizvoda (differentia specifica) je uslov da prve komponente moraju biti međusobno različite.

Polazeći od ovakve strukture definicije, lako zapažamo manjkavost navedene definicije čoveka jer nema differentia-e, a zatim "živo biće" nije najbliži viši pojam.

S obzirom na to da je definicija objekta po prirodi empirijska, za početak je neophodno nabrojati njegove karakteristike uočene od strane relevantnih autora. Sa kognitivnog stanovišta objekat je [3]:

- opipljiva ili vidljiva stvar
- nešto što se može intelektualno pojmiti
- nešto na šta se mogu usmeriti misli ili akcije.

Dalje, praktično svi autori prihvataju gledište da su bitne ***osobine objekta***:

- identitet,
- stanje i
- ponašanje.

U objektnom sistemu svaki objekat je jedinstven (unikalan). Identitet, u širem smislu, predstavlja osobinu datog objekta koja ga odvaja od svih ostalih objekata, tj. osobinu po kojoj je on *prepoznatljiv*. Više objekata mogu da budu u istom stanju ili da imaju jednako ponašanje, a ipak se po nečem moraju razlikovati: to "nešto" nosi naziv identitet. Da bismo ilustrovali ulogu identiteta kao osnovnog i neizbežnog svojstva objekta uporedimo matrice sa slika (a) i (b).

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

(a) (b)

Pitanje glasi: da li su na slikama (a) i (b) prikazane *dve* jednake matrice ili (b) predstavlja matricu sa slike (a) koja je jednostavno *prepisana*? Odgovor se može dobiti samo ako matricama dodelimo identitet u vidu, recimo, imena. Ako je na slici (a) prikazana matrica M, a na slici (b) matrica N, tada se, očito, radi o dvema matricama za koje važi M=N. Ako im se, pak, dodeli ista oznaka, recimo M, onda je to

jedna matrica napisana na dva mesta. Promenimo li vrednost elementa u gornjem levom uglu slike (a) sa 1 na 10, u prvom slučaju će jednostavno prestati da važi M=N, dok se u drugom podrazumeva promena vrednosti odgovarajućeg elementa i na slici (b). U primeru sistema linearnih jednačina Ax=b vektori b i x mogu da budu jednaki, ako je matrica A jedinična; pored toga, repertoar operacija nad njima je isti. Ono po čemu se razlikuju je, u ovom slučaju, *ime* "x" odnosno "b". Dodeljivanje imena objektu nije jedini način za njegovo identifikovanje o čemu će biti reči kasnije u ovom poglavlju.

I koncept stanja je sasvim generalizovan jer seže daleko van granica računarstva uopšte. U smislu u kojem je ovde upotrebljen, potiče iz opšte teorije sistema. Arbib i dr. [60] daju intuitivnu definiciju stanja sistema kao dela sadašnjosti i prošlosti sistema potrebnog[13] za opredeljivanje njegovog sadašnjeg i budućeg odziva na pobude (a to nije ništa drugo do ponašanje). U slučaju objekta to znači da ishod dejstva nad njim zavisi kako od konkretnog zahteva tako i od stanja objekta. Rezultat dejstva "invertovati" nad matricom zavisi od trenutnih vrednosti njenih dimenzija i sadržaja elemenata koji su, po sebi, slika stanja matrice. Štaviše, matrica može biti u takvom stanju da se ne može invertovati (ako je singularna ili ako nije kvadratna).

Kada se kaže da objekat ima ponašanje (engl. behaviour, am. behavior) ima se na umu činjenica da je objekat aktivan, tj. da reaguje na pobudu (stimulans) promenom stanja i generisanjem odziva. Isto tako, objekat može pobuđivati druge objekte. Ponašanje objekta određeno je operacijama, pri čemu konkretni obrazac ponašanja zavisi kako od same operacije tako i od tekućeg (zatečenog) stanja. Operacija "transponovati" menja stanje matrice ne samo u pogledu vrednosti elemenata nego i u pogledu dimenzija. Operacija "ZadatiElement(i, j, vrednost)" takođe menja stanje objekta. Operacija "OčitatiElement(i, j)" samo generiše izlazni rezultat ne menjajući, , stanje (ili, ako se nekom dopada, menjajući stanje iz tekućeg u isto to).

2.1.1. Pregled definicija objekta

Razmotrimo, za početak, nekoliko definicija objekta, [72]. Na terminološki značajnim mestima dati su originalni izrazi.

- **D1.** Objekat predstavlja individuu, pojedinost koja se može identifikovati ("identifiable item"), jedinicu ("unit") ili entitet, realan ili apstraktan, sa dobro definisanom ulogom u domenu problema (Smith, Tockey, [61]).
- **D2.** Objekat definišemo kao bilo šta ("anything") sa oštro i jasno definisanim granicama (Cox, [62]).
- **D3.** Objekat je stvar ("thing") koja se može identifikovati i koja igra određenu ulogu u odnosu na zahtev za izvršavanje operacija (Banerjee, prema [27]).

[13] Ne i dovoljnog!

D4. Objekat je kolekcija operacija koje dele ("share") stanje (Wegner, prema [27]).

D5. Objekat je inkapsulacija (rezultat objedinjavanja) skupa operacija ili metoda[14] koje se mogu eksterno pobuđivati i stanja koje pamti efekte primene metoda (Blair, prema [27]).

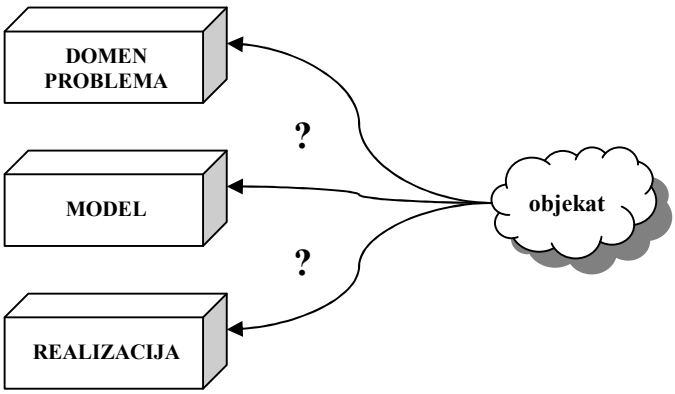
D6. Objekat je instanca (primerak) klase objekata u vreme izvršavanja (Meyer, [82]).

D7. Objekat je model entiteta koji ima identitet, stanje i ponašanje, gde se pod entitetom podrazumeva postojanje nečeg (Elmer [27], modifikovana u [58]).

Pre no što pristupimo analizi navedenih definicija, moramo ustanoviti na kojem stupnju razvoja softverskog sistema objekat može da se uvede, jer mogućnosti ima tri. Naime, prva faza izrade programskog sistema jeste *analiza sistema* koja se sprovodi na nivou tzv. *domena problema* (zove se i konceptualni nivo). Domen problema (engl. "entity world") je deo realnog sveta, materijalne ili idealne prirode, koji je predmet analize sistema. Svrha ograničavanja realnosti na domen problema jeste da unapred eliminišemo, ponekad nepremostive, poteškoće vezane za univerzalije gde se postavljaju najteža filozofska pitanja poput egzistencije ili saznanja. Na primer, ako bismo želeli da opišemo entitet "građanin", morali bismo uzeti u obzir da se radi o ljudskom biću koje je sisar, kičmenjak, životinja, živo biće itd. U domenu problema informacionog sistema nekog grada, biološke osobine građanina (osim pola) nisu od interesa te ih, iako postoje, jednostavno zanemarujemo. Drugi nivo jeste *modelovanje*, odnosno izrada homomorfne (uprošćene) slike stvarnog entiteta iz domena problema, pri čemu se radi o softverskom modelu, dakle o modelu prilagođenom realizaciji na računaru. Konačno, treći stupanj je sama *realizacija* ili *implementacija* koja podrazumeva neposrednu izradu softvera (kodiranje, testiranje, dokumentovanje).

Pojam objekta moguće je uvesti na bilo kojem od pomenuta tri nivoa: već na nivou domena problema, na nivou modela ili tek na nivou realizacije. Razlike između gornjih definicija objekta treba tražiti, pre svih drugih, u nivou na kojem se on uvodi. Odmah se primećuje da su navedene definicije međusobno nesaglasne u pogledu mesta objekta: prve dve smeštaju objekat u domen problema, treća je negde na granici, dok ga D4, D5 i D7 postavljaju u ravan modela. Konačno, Mejer u D6 izvodi objekat iz pojma klase objekata (videti dalje) i to u domenu realizacije.

[14] Ovde se termini "operacija" i "metoda" smatraju sinonimima, što nije uvek slučaj.



Slika 2.1

Izvršićemo sada kratku analizu navedenih definicija. Definicija D1 ima (najmanje) jednu ozbiljnu manjkavost: pojmovi navedeni u genusu nisu na istom nivou apstrakcije. Tamo je, kao četvrti po redu(!), ravnopravno sa ostala tri, naveden *entitet*. Ima određenih neusaglašenosti oko ovog pojma koji se u oblasti informacionih sistema tradicionalno interpretira kao stvar, događaj, subjekt ili apstraktni pojam. Ova interpetacija je u dobroj meri pojednostavljena jer je entitet jedan od najopštijih filozofskih (ontoloških) koncepata, generalizovan do minimuma semantike. Prema [57] definicija entiteta glasi

- Entitet (novolatinski entitas, izvedeno iz ens = biće, postojeće) postojanje nečega. Pojam entitet označuje da nešto *jeste* [...]

Pojam entiteta definisan je slično i međunarodnim standardom ISO/IEC [101] kao

- Bilo koja konkretna ili apstraktna stvar ("thing") koja postoji, koja je postojala ili je mogla postojati, uključujući i veze između ovih stvari.

Kao što se vidi, pojam entiteta je uopšten do jednog jedinog zahteva: treba da postoji i ništa više. Pritom, reč "postojanje" treba shvatiti kao egzistenciju u realnom svetu ili u apstraktnom mišljenju. Ovako definisan, entitet može, naravno, da posluži kao genus proximum za objekat. Ako se vratimo na definiciju D1, primetićemo da individua kao pojam predstavlja vrstu entiteta (čak vrlo tipičnu), što važi i za "pojedinost koja se može identifikovati", tako da struktura D1 podseća na rečenicu tipa "lav je mačka i sisar" ili "funkcija je relacija i skup"). Kada tome dodamo ne baš jasnu diferenciju (šta znači "dobro definisana uloga"?), sledi da D1 nije najbolji kandidat za definiciju objekta.

Definicija D2 ima suviše širok genus ("bilo šta"), a definicija D3 ponovo nejasan deo što se odnosi na diferenciju ("određena uloga").

Definicija D4 svodi objekat na kolekciju operacija (dakle, na ponašanje) i stanje ispuštajući identitet. Posebno, sintagma "operacije koje dele stanje" prilično je neodređena.

Definicija D5 je dosta određena, ali takođe ne sadrži eksplicitno identitet. Pored toga, oslanja se na inkapsulaciju koja, u objektnom programiranju, nije fundamentalan nego izveden pojam.

Mejerova definicija D6 prilično se razlikuje od prethodnih, jer prvo pojam objekta izvodi iz pojma klase, i drugo objekat i klasu uvodi tek na nivou realizacije. Mejer definiše klasu kao jedan od mogućih načina implementacije apstraktnog tipa podataka (Abstract Data Type, ADT). Prednost ove definicije leži u činjenici da se ona naslanja na precizno definisan, matematički pojam - apstraktni tip podataka te, shodno tome, ima čvrsto, aksiomatizovano uporište. Ni ova definicija nije bez mana. Prvo, klasa se u praksi pojavljuje već na nivou modela jer objektni sistemi zahtevaju posebne postupke i za modelovanje, a ne samo za realizaciju. Prema tome, ova definicija ne poštuje *faktičko stanje stvari*. Pored toga, javljaju se i problemi sa konceptom stanja, jer se ADT zasniva na vrednostima i preslikavanjima tih vrednosti, a ne na stanjima. Najzad, dubiozno je da li je klasa fundamentalniji pojam od objekta. Naime, klasa[15] i objekat stoje u odnosu grupno-individualno, a šta je od toga "starije" teško je prosuditi.

Ostaje još definicija D7. Ona ima dosta jednostavan genus i razumljivu diferenciju i, kao takva, dosta je bliska faktičkom stanju stvari. Ipak, primedba se može staviti. Radi se o terminu "entitet" koji je semantički gotovo prazan, pa stoga i slabo određen, i oko kojeg već postoje nesaglasnosti u pogledu interpretacije (videti u diskusiji uz definiciju). Posebno, ISO definicija entiteta proširuje njegovo značenje i na veze (relacije) što, videćemo, nije opravdano jer se radi o terminima koji odgovaraju nesvodljivim pojmovima (tzv. kategorijama). Drugim rečima, stvari i veze ne mogu se podvesti pod zajednički viši pojam, što se najlakše ustanovljava na primeru najvažnije od svih veza, tzv.nasleđivanja. Konačno, domen problema nosi i naziv konceptualni nivo upravo zbog toga što se oslanja na koncept ili *pojam*. Zašto onda uvoditi posebnu kategoriju - entitet - kada se domen problema po sopstvenom značenju već sastoji od pojmova? U narednom odeljku pokazaćemo da se i objekat i klasa objekata prirodno izvode iz pojma i ne samo to: njihove najvažnije osobine direktno se izvode iz već poznatih postavki logike, pri čemu se pitanje "treba li izvesti objekat iz klase ili obrnuto?" uopšte ne postavlja jer se i klasa objekata i objekat izvode neposredno iz pojma.

2.1.2. Konceptualna definicija klase i objekta

Dok izvodi analizu i modelovanje, recimo, informacionog sistema fakulteta, softverski inženjer ne barata realnim studentima, nastavnicima ili inventarom. On se bavi, u stvari, njihovim *mentalnim slikama*. Drugim rečima, projektant se bavi *mislima*, i to ne bilo kakvim nego mislima o učesnicima u informacionom sistemu, a isto važi kada su u pitanju softverski proizvodi ma kojeg

[15] Tačnije, klasa objekata (biće definisana u sledećim odeljcima).

tipa: jedinica posmatranja tretira se uvek preko misli o njoj, odnosno misli o njenim bitnim karakteristikama. Ova činjenica navodi na ideju da se i klasa i objekat vežu direktno za misli, bez ikakvih posrednika u vidu entiteta ili nečeg trećeg.

- Misao o bitnim karakteristikama predmeta jeste ***pojam*** ili ***koncept***[16],

pri čemu se reč "predmet" ovde koristi u najširem značenju, nešto kao "ono o čemu se misli".

Sam pojam odlikuje se sadržajem i opsegom. Pre svega, misao o ma kakvim karakteristikama (bitnim i nebitnim) zove se *oznaka pojma*. Oznaka koja je bitna nosi naziv *bitna oznaka* ili *atribut*, a razlika između bitnih i nebitnih oznaka leži u tome da se ove druge mogu izvesti iz prvih.

Sadržaj pojma definišemo kao skup njegovih bitnih oznaka. Na primer, sadržaj pojma FUNKCIJA jeste "relacija" i "Ako (a,b) i (a,c) pripadaju funkciji tada je b=c". Sadržaj pojma TROUGAO jeste "geometrijska figura u ravni" i "tri stranice". Jednakost visina jednakostraničnog trougla nije bitna oznaka jer proizlazi iz jednakosti dužina njegovih stranica. Očigledno, sadržaj pojma jeste u direktnoj vezi sa već razmatranom definicijom definicije, jer ga treba tražiti u definiensu, tj. u genusu i diferenciji.

Neka su *A* i *B* dva pojma. Ako za pojam *B* možemo reći "svako *B* je istovremeno i *A*" tada je *A generički (rodni) pojam* u odnosu na *B*, a *B* je *vrsni pojam* u odnosu na *A*. Tako, svaki jednakostranični trougao jeste trougao, a isto važi i za svaki pravougli ili ravnokraki trougao. Pod ***opsegom*** pojma podrazumevamo skup njegovih vrsnih pojmova. Opseg pojma MAČKA[17] obuhvata pojmove LAV, TIGAR, LEOPARD, JAGUAR, RIS itd. Zapazimo još i to da je relacija *biti generički odn. vrsni pojam*" tranzitivna u smislu da ako je *A* generički pojam u odnosu na *B*, a *B* generički pojam u odnosu na *C* tada je *A* generički pojam u odnosu na *C*. Za navedene primere to bi značilo "pošto je svaki lav mačka, a svaka mačka sisar, svaki lav je sisar" ili "kako je svaki pravougli trougao - trougao, a svaki trougao geometrijska figura u ravni, to je svaki pravougli trougao geometrijska figura u ravni". Veza između generičkog pojma i genusa u definiciji očigledna je: genus proximum je, u stvari, najbliži generički pojam u odnosu na definiendum.

Za sadržaj i opseg pojma kaže se da su u obrnutoj srazmeri: niži (vrsni) pojam ima veći sadržaj od generičkog, ali zato mu je opseg manji. Ovo je logično jer se vrsni pojam dobija iz generičkog dodavanjem semantike u vidu oznaka, ali se time sužava opseg jer je vrsni pojam specifičniji. Pojam SISAR ima u opsegu mnoge pojmove tipa MAČKA, PAS, SLON, KIT itd., dok MAČKA obuhvata samo pripadnike porodice mačaka.

Pojmovi se klasifikuju na mnogo načina. Mogu biti pojmovi o fizičkim

[16] Materijal vezan za pojam izrađen je prema [102].
[17] Misli se na porodicu mačaka.

predmetima (pojam kuće npr.) koji postoje u realnom svetu, ili logičkim (idealnim) predmetima koji ne egzistiraju u realnom svetu (npr. pojam matrice). Za naše razmatranje, tj. u svrhu izgradnje konceptualnog pogleda na objekte, od posebnog interesa je podela pojmova na individualne i klasne. *Individualni pojmovi* odnose se na individualne predmete: pojmovi "Mocart" ili "Mikelanđelo" odnose se na dve pojedinačne osobe. Prvi se može opisati kao kompozitor opere "Figarova ženidba" i ronda "Alla turca", a drugi kao umetnik koji je oslikao Sikstinsku kapelu i vajao "Mojsija". Pri tom, veoma je važno razlučiti Mocarta i Mikelanđela od pojmova o njima. Mocart i Mikelanđelo su realne osobe koje su postojale u vremenu, a odgovarajući pojmovi su samo misli o njihovim bitnim oznakama.

Pojedinačni predmeti koji imaju zajedničke oznake čine klasu, a misao o klasi ima prirodu *klasnog pojma*. Pojam "barokni kompozitor" obuhvata sve pojedinačne pojmove o Bahu, Hendlu, Vivaldiju, Liliju itd.

Zastaćemo na ovom mestu da razjasnimo još jednu potencijalnu nedoumicu. Moguće je, kada se radi o pojmovima za stvari[18], da individualni i klasni pojam nose isti naziv i čak imaju isti sadržaj (ne i opseg). Pojam "mačka" može se odnositi na porodicu mačaka, ali i na (apstraktnu) individuu koja poseduje sve oznake vezane za porodicu mačaka i samo njih. Ovakav par sastavljen od klasnog i individualnog pojma može da ima isti sadržaj, ali individualni pojam nema opseg jer se odnosi na pojedinost. Klasni i odgovarajući individualni pojam stoje u odnosu celina-deo u kakvom su, recimo, skup i njegov element. I još nešto: rekli smo da klasni i odgovarajući individualni pojam *mogu* imati isti sadržaj, ali to nije obavezno. Naime, klasni pojam obuhvata sve oznake koje ima odgovarajući individualni pojam, ali može imati još oznaka koje postoje samo na nivou klase. Na primer, klasnom pojmu LAV može se pridružiti oznaka "broj lavova" koju je besmisleno vezivati za pojedinačne lavove. Prema tome, u opštem slučaju,

- sadržaj klasnog pojma obuhvata sadržaj odgovarajućih individualnih pojmova, ali klasni pojam može da ima i neke sopstvene oznake.

Još od Aristotela vodi se polemika o tome da li postoje i koji su to najopštiji pojmovi, tj. pojmovi za koje ne postoje generički pojmovi. Takvi pojmovi najvišeg reda zovu se ***kategorije***. Pominje ih već Aristotel, ali nedosledno: neki put navodi deset kategorija, neki put osam, pa četiri. Ilustracije radi navodimo nekoliko: stvar (supstanca), kvalitet, kvantitet, odnos. Kant navodi četiri: kvantitet, kvalitet, odnos i modalitet. U modernom sistemu Ingvara Johansona (1989.) i Roderika Čisholma (1996.) navode se devet kategorija, videti [104]. Bez namere da ulazimo u detalje ili da analizujemo različite skupove kategorija, možemo uočiti da se u svim sistemima kao kategorije ili bar kao pojmovi neposredno podređeni kategorijama pojavljuju tri pojma što predstavljaju podlogu za konceptualne definicije objekta i klase objekata. To su:

[18] ne i za npr. svojstva ili odnose koji su takođe pojmovi.

1. Pojam za stvar (čovek, kuća, matrica, simfonija) i pojam za proces
2. Pojam za svojstvo (boja, masa, prezime, ali i mogućnost letenja ili - za časovnik - mogućnost pomeranja kazaljki).
3. Pojam za odnos (biti veći, biti manji, posedovati, pripadati).

Pojmovi za stvar i svojstvo pojavljuju se ili kao kategorije ili kao pojmovi neposredno izvedeni iz kategorije nazvane "kvalitet", dok se odnos u svim sistemima pojavljuje kao kategorija. Pojam za proces je tradicionalno vezan za objekat i klasu. U svetlu toga, možemo primetiti da ISO definicija entiteta nije u saglasju ni sa jednim sistemom kategorija jer povezuje stvar i odnos, iako za pojam odnosa nema višeg pojma. Zato ćemo u nastavku

- ***entitet*** smatrati za treći sinonim reči stvar ili supstanca.

Vratimo se na sadržaj pojma, čije ustanovljavanje nije ni izbliza jednostavan posao. Pokušajmo sa pojmom GRAĐANIN. Ovaj pojam ima za bitne oznake svakako matični broj, adresu i broj lične karte. Šta je, međutim, sa činjenicom da je svaki građanin istovremeno i ljudsko biće, pa ima dve ruke, dve noge, sposobnost razmišljanja ...? Upravo na ovom mestu pokazuje se opravdanost uvođenja domena problema. Naime, a za razliku od logičara, projektant softvera ne mora da vodi računa o *svim* bitnim oznakama pojma, već samo o onim što su *relevantne* za domen problema koji je predmet analize i modelovanja. Činjenica da je nastavnik ljudsko biće nije od interesa za domen problema informacionog sistema fakulteta, a isto važi i za građanina u okviru informacionog sistema grada, što neobično olakšava posao oko ustanovljavanja bitnih oznaka. Oznake pojma koje su od interesa u datom domenu problema nazivaćemo ***relevantnim oznakama*** pojma (relevantnim atributima, prema [103], modifikovano). Zamena bitnih oznaka relevantnim rešava još i problem ustanovljavanja da li je neka oznaka bitna ili nije. Recimo, oznaka "matični broj građanina" tamo gde postoji sigurno je bitna, ali u nekim zemljama još uvek nije uvedena , te je uopšte nema!

Sada možemo navesti konceptualne definicije klase i objekta. Skrećemo odmah pažnju na to da ove definicije imaju dve prednosti u odnosu na sve ostale iz prethodnog odeljka:

a) definicije su zasnovane na pojmovima, tj. na dobro razrađenom i jasnom sistemu termina i njihovih značenja
b) definicije klase i objekta su ravnopravne - klasa se ne definiše preko objekta niti obrnuto.

Klasa i objekat su u semantičkom smislu učinjeni ravnopravnim zahvaljujući tome što su izvedene iz pojmova i to kao njihovi *modeli*, gde model predstavlja homomorfnu (uprošćenu) sliku onoga što se modeluje. Dajemo prvo ***konceptualnu definiciju klase***:

- Klasa objekata, ili kratko *klasa*, jeste softverski model klasnog pojma za stvar ili proces zasnovan na koncepciji stanja, i sa sadržajem koji čine oznake relevantne za dati domen problema.

Pre svega, relevantne oznake klasnog pojma, po svojoj prirodi, takođe su pojmovi. One mogu biti dvojake: opet pojmovi za stvar (nazovimo ih ***fragmentima***) ili pojmovi za svojstvo. Klasa *Duž* ima za fragmente dva temena koji su, sa svoje strane, opet pojmovi za istu stvar - tačku. Oznake koje su pojmovi za svojstvo (zovu se ***odlike***, engl. "features") mogu biti čisto deskriptivne, kao što su boja, marka (automobila), masa i sl., ali mogu biti i operacione kakve su npr. oznake vezane za mogućnost kretanja kod automobila ili reprodukcije zvuka kod CD plejera. Pri tom, veoma je važno podvući da objektna metodologija *ne pravi nikakvu razliku između deskriptivnih i operacionih oznaka*! ***Konceptualna definicija objekta*** glasi:

- Objekat je softverski model individualnog pojma za stvar ili proces, zasnovan na koncepciji stanja i sa sadržajem koji čine oznake relevantne za dati domen problema.

Očigledno, konceptualni pogled postavlja klasu i objekat u ravnopravan položaj. Razlika je samo u tome što klasa odgovara klasnom, dok se objekat vezuje za individualni pojam. Klasa i objekat povezani su jednom pretpostavkom koja ima, u stvari, snagu postulata i glasi:

- za svaki objekat postoji klasa koja poseduje sve njegove relevantne oznake. U tom slučaju, kaže se da objekat *pripada* klasi.

Svaki pojedinačni ravnokraki trougao (sa konkretnim vrednostima stranica) pripada klasi *RavnokrakiTrougao*. Između deskriptivnih oznaka-fragmenata klasnog pojma odn. individualnog pojma postoji (sasvim prirodna) razlika utoliko što za njih važi da se u prvom slučaju pojavljuju u klasnoj, a u drugom u individualnoj varijanti. Na primer, odlika klase "boja" kod individue se pojavljuje kao "bela" ili "zelena".

Pojmovi uopšte, pa prema tome i pojmovi za stvar, mogu biti složeni što znači da njihovi fragmenti mogu imati svoje fragmente, ovi svoje itd. Na primer, udžbenik se sastoji poglavlja, poglavlja sadrže pasuse, pasusi linije, a linije sadrže znake. Skup sastavljen od pojma za stvar, njegovih fragmenata, njihovih fragmenata itd. uređen relacijom "biti fragment" čini *strukturu* takvog pojma. Kada se individualni pojam za stvar modeluje objektom, njegova struktura postaje ***struktura objekta***.

U prethodnom odeljku istakli smo identitet, stanje i ponašanje kao esencijalne osobine svakog objekta. Navedena definicija objekta nije u koliziji sa ovim pretpostavkama. Pre svega, identitet objekta mora da postoji jednostavno zato što, ako toga nema, za neke objekte ne bismo bili u stanju da razlučimo da li se radi o objektima sa istim sadržajem ili o kopijama istog objekta. Koncepcija stanja predstavlja, u stvari, fundament objektnog modelovanja kao takvog. Stanje se, kako ćemo videti, modeluje tzv. podacima-članovima, a odlike se *izvode iz stanja* kao funkcije stanja, odn. funkcije promene stanja (zato se i ne mogu izvesti iz apstraktnog tipa podataka, kao što je urađeno u definiciji D6). Odlike koje su operacionog karaktera determinišu ponašanje objekta.

Kada se iz klasnog pojma izuzmu oznake nivoa klase, dobija se skup oznaka (fragmenata i odlika) koje poseduje svaki individualni pojam vezan za tu klasu. Imajući u vidu definiciju objekta, logično sledi da

- objekti iste klase imaju istu strukturu i isto ponašanje.

Inače, klasa i objekat stoje u identičnom odnosu u kojem su tip podataka i promenljiva u standardnim programskim jezicima. Promenljive istog tipa, npr. *int*, predstavljaju primerke (pojave) pomenutog tipa; na isti način pojedini objekti predstavljaju primerke (pojave, instance) svoje klase. Mattos (prema [27]) u pogledu namene, uočava tri aspekta klase:

1. Klasa po definiciji služi za grupisanje objekata.
2. Klasa ima namenu da opredeli strukturu i ponašanje svih objekata koji joj pripadaju (tj. ima prirodu *šeme*).
3. Klasa služi kao generator objekata. Nešto šire, ona se može shvatiti kao svojevrsno skladište (warehouse) objekata vezano za generator (nazvan "fabrikom" - factory) čiji je zadatak da "proizvede" (konstruiše, stvori) pojedine objekte. U tom smislu najosnovnija operacija u okviru klase je operacija kreiranja objekta koja nosi naziv *konstruktor* (videćemo kasnije da ih može biti i više). Obrnuto, objekti se mogu i uništavati takođe bazičnom operacijom sa nazivom *destruktor*.

Recimo, na kraju, nešto i o terminu ***atribut***. U filozofiji, atribut (od latinskog attributum = dodeljen) predstavlja dobro definisan pojam sa definicijom

- atribut je bitno, nužno svojstvo neke stvari koja se može pomišljati i nezavisno od stvari same (prema [57])

i po smislu slaže se sa terminom "bitna oznaka", kao što je dato u ovom tekstu. Međutim, već smo uočili da postoje dva vida bitnih oznaka: one koje se odnose na deskriptivne oznake, ali i one koje su po prirodi operacije. Pod uticajem klasične informatike, neki autori i u objektnom ambijentu atributima nazivaju samo deskriptivne odlike, iako se suština objektne metodologije sastoji upravo u izjednačavanju svih odlika, kako deskriptivnih tako i operacionih. Očigledno je, na primer, da odlika "može da leti" jeste legalan atribut koji zadovoljava definiciju jer je prvo bitna, a zatim i nezavisna od stvari, jer se može pridružiti ptici, insektu, avionu... Amadi i Kardeli u [78] eksplicitno navode da atributi obuhvataju *sve* bitne oznake, kako deskriptivne tako i operacione. Sve u svemu, gde god postoji mogućnost nesporazuma izbegavaćemo upotrebu termina "atribut".

2.2. MODELOVANJE I REALIZACIJA

U dosadašnjim razmatranjima vezanim za objekte i klase pažnja je bila usmerena ka domenu problema i, malim delom, ka nivou modela (objekti i klase objekata). U ovom odeljku detaljnije ćemo razmotriti modelovanje i realizaciju (koja nosi i naziv implementacija).

Modelovanjem se moraju obuhvatiti sve relevantne oznake i to, pre svega, klase jer se one definišu na nivou klase, a konkretizuju na nivou objekta. Kao što je već napomenuto, osnovu objektnog modela čini koncepcija stanja. Stanje se modeluje tzv. ***podacima-članovima*** koji se u programskim jezicima implementiraju kao promenljive procedurnog tipa (skalarne svih vrsta, nizovi itd.). I ovde dugujemo jedno objašnjenje. Naime, podaci-članovi mogu imati semantiku, ali ne moraju. Podatak-član klase *Kocka* sa nazivom *ivica* može nositi to ime ili ma koje drugo – recimo *q*. Ovo nema suštinskog uticaja, jer primarna namena podataka-članova jeste da *modeluju stanja objekta*. Iz ovako modelovanih stanja odlike se izvode ***funkcijama-članicama*** koje se, čak češće, nazivaju ***metodama***. Drugim rečima, između stanja i odlike uspostavlja se uzročno-posledična veza u kojoj je stanje uzrok, a odlika posledica. Razlog za to ilustruje se prostom činjenicom da je npr. banana žuta (odlika) zato što je zrela (stanje), a ne obrnuto! Još jedna ilustracija data je u primeru 2.2 (videti dalje). Po svojoj prirodi, metode su skoro identične sa potprogramima, a osnovna razlika leži u činjenici da metode ne egzistiraju samostalno nego su deo klase. Pošto u hibridnim jezicima poput C^{++} postoje obe vrste funkcija, potprograme (funkcije u terminologiji programskog jezika C) nazivamo ***slobodnim funkcijama*** ili slobodnim potprogramima. Metode su tako koncipirane da dejstvuju na objekte, pa je uobičajeno da se, umesto izraza "aktivirati metodu" ili "pozvati metodu" kaže poslati ***poruku***.

Nešto je drukčija situacija sa fragmentima. Ovde se radi o oznakama koje su po prirodi pojmovi za stvar i sa matičnim pojmom stoje u odnosu deo-celina. Pošto se radi o pojmovima za stvar, fragmenti se modeluju objektima koji nose naziv ***objekti-članovi***. U nekim programskim jezicima (npr. java, zbog načina realizacije) podaci-članovi i objekti-članovi nose zajednički naziv ***polja***. Shodno ranije rečenom, strukturu objekta čine njegovi objekti-članovi, njihovi objekti-članovi itd. Buč [3] opisujući karakteristike kompleksnog objekta navodi:

- Odluka o tome koji su objekti-članovi datog objekta[19] primitivni (tj. atomarni, bez sopstvenih objekata-članova) relativno je arbitrarna i u velikoj meri zavisi od nahođenja posmatrača[20].

Dodajmo da i za sam objekat postoje dva alternativna termina: ***objekat*** i ***instanca klase*** (drugi termin se u poslednje vreme sve češće primenjuje).

Relevantna oznaka	koja je pojam za	modeluje se kao
fragment	stvar (entitet)	objekat-član
odlika	svojstvo	funkcija-članica (metoda)

[19] Buč upotrebljava reči "komponenta" umesto "objekat-član" i "sistem" umesto "objekat".
[20] što je, uzgred, još jedan dokaz opravdanosti zamene bitnih oznaka relevantnim oznakama!

Samo u trivijalnim sistemima pojmovi egzistiraju kao izolovane celine. U realnim domenima problema, individualni i klasni pojmovi stoje u različitim međusobnim *odnosima*. Već između pojma i njegovih fragmenata postoji odnos "posedovati kao deo sadržaja". Odnosi između individualnih i odgovarajućih klasnih pojmova semantički su raznovrsni. Individualni pojam "Proizvodnja" iz klase SEKTOR stoji u odnosu "pripada" sa individualnim pojmom "Mašinprodukt" iz klase PREDUZEĆE. Odnos "pripada" spada u odnose tipa deo-celina. Odnos "predavati" između individualnog pojma klase NASTAVNIK i individualnog pojma klase PREDMET je očigledno sasvim drukčiji, jer niti je nastavnik deo predmeta niti je predmet deo nastavnika. Pored toga, odnos je promenljiv u vremenu. Odnos između individualnog pojma klase PC_RAČUNAR i individualnog pojma klase MONITOR je takav da prvi ne može da postoji bez drugog, dok obrnuto ne važi. Između klasnih pojmova MAČKA i SISAR postoji odnos koji se može opisati rečju "je", "jeste". Odnosi između klasnih kao i između individualnih pojmova modeluju se ***vezama*** koje imaju prirodu matematičkih relacija snabdevenih semantikom.

Primer 2.1. Posmatrajmo klasni pojam geometrijske kocke. Klasa (objekata) *Kocka* koja modeluje ovaj pojam može se oformiti različitim postupcima. Jedan od načina je da se definiše skalarni podatak-član *ivica* čije vrednosti interpretiramo neposredno kao stanja. Na bazi ovako zadatih stanja opredeljujemo ponašanje putem metoda

- kreiranje objekta uz zadavanje ivice (čime se odmah određuje i stanje)
- *PromenitiDuzinuIvice*
- *Povrsina* kao šestostruki kvadrat ivice
- *Zapremina* kao kub ivice
- *PovrsinaStrane* itd.

Alternativno, može se oformiti klasa *Kvadrat* sa podatkom-članom *stranica* (koji svojom vrednošću određuje stanja kvadrata) i metodama

- kreiranje objekta uz zadavanje stranice
- *PromenitiDuzinuStranice*
- *OcitatiDuzinuStranice*
- *Povrsina* (kvadrata).

Na osnovu klase *Kvadrat* gradi se klasa *Kocka* čiji svaki primerak sadrži po jedan objekat-član *strana* klase *Kvadrat*, a koji igra ulogu strane kocke. Metode u ovoj verziji klase *Kocka* su

- kreiranje objekta; objekat klase *Kocka* može se kreirati na bazi zadate strane koja je objekat klase *Kvadrat*, na bazi zadate stranice pošto se prethodno kreira strana ili i jedno i drugo
- *Povrsina* kao šestostruka površina strane
- *Zapremina* kao umnožak površine strane i njene stranice.

Ove dve verzije, mada očigledno različite jer se u drugom rešenju pojavljuje i

samostalna klasa *Kvadrat* ipak generišu isto ponašanje. Naime, i u drugom slučaju mogu se primeniti operacije promene i očitavanja dužine ivice kocke te određivanje površine strane, ali posredstvom odgovarajućih metoda iz klase *Kvadrat* koje se primenjuju nad stranom kocke. Dakle, u istom domenu problema, klasni pojam za entitet može se modelovati na više načina.

Ako se, međutim, domen problema promeni, objekti klase *Kocka* počinju da se ponašaju drukčije. Neka je domen problema trodimenzionalni koordinatni sistem. Deskriptivne osobine odgovarajućeg modela - objekta postaju sada koordinate temena, a promenjeno ponašanje diktirano je mogućnošću postojanja metoda za translaciju ili rotaciju pored ranije pomenutih.

Treći, ne manje važan, aspekt objektne metodologije je programska (računarska) realizacija klase odnosno objekta. Videli smo da se određeni klasni pojam za stvar može modelovati na više načina kao što je pokazano primerom 2.1; isto tako se i klasa objekata, kao model klasnog pojma, može različito *realizovati*. Realnu matricu realizujemo primenom standardnog tipa matrice koji se nalazi u svakom programskom jeziku, ali to nije jedina varijanta: matrica se može realizovati i kao niz slogova koji sadrže vrednost elementa, njegovu vrstu i njegovu kolonu što se, inače, praktikuje kod velikih, retkih matrica, a može se realizovati čak i spregnuto, primenom Iliffe-ovih vektora. Čak i u sasvim jednostavnom slučaju tačke u Dekartovom koordinatnom sistemu postoje bar dve mogućnosti: koordinate se mogu izvesti kao dva podatka-člana tipa *double*, ali i kao dvokomponentni *double* niz čija prva komponenta igra ulogu apscise, a druga ordinate.

Pre nego što pređemo na razmatranje implementacije, moramo skrenuti pažnju na činjenicu da objektni terminosistem nije potpun - objekat nosi isti naziv i kao model i kao njegova programska realizacija iako je razlika velika, u najmanju ruku zbog toga što realizacija zavisi od programskog jezika. Isto važi i za ostale termine: podatak-član, objekat-član, polje, funkcija-članica odn. metoda i poruka nose iste nazive i u modelu i kod realizacije. U daljem tekstu, osim kada su u pitanju definicije ili kada postoji mogućnost nesporazuma, nećemo preterano insistirati na preciznosti terminologije [68].

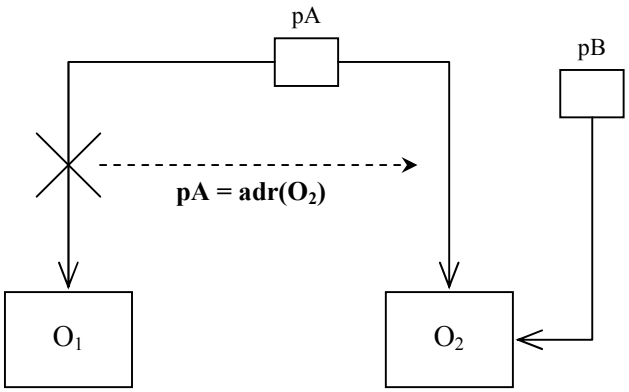
Kao što smo videli, objekat poseduje identitet, stanje i ponašanje, a objekti koji imaju istu strukturu i ponašanje pripadaju istoj klasi. Prema tome, pri razmatranju programske realizacije moraju se diskutovati sva četiri aspekta: identitet, stanje, ponašanje i struktura.

Identitet je svojstvo objekta da se razlikuje od svih ostalih objekata u modelu (i realizaciji). Od objekata drugih klasa dati objekat se odvaja strukturom i ponašanjem, no kada su u pitanju objekti iste klase struktura i ponašanje su, po definiciji, jednaki, tako da se identitet mora tražiti drugde. Da bi objektni sistem bio konsistentan, identifikatori objekata moraju ispunjavati dva uslova:

1. Identifikator mora biti unikalan, tj. dva objekta ne mogu imati isti identifikator

2. Za svaki identifikator mora postojati objekat kojeg on identifikuje, tj. identifikator ne može egzistirati *per se*, mimo objekta.

U skladu sa tačkom 1 i sa definicijom objekta, Elmer u [27] citira (sa dozom humora): "Dva objekta su identična samo ako je u pitanju isti objekat". Alen Kej, jedan od pionira objektnog programiranja i tvorac programskog jezika smoltok, formulisao je 5 principa na kojima je zasnovan pomenuti jezik. U trećem po redu, eksplicitno se tvrdi da "svaki objekat poseduje sopstvenu memoriju [...]". S obzirom na to, objekat kao programska kategorija se, u krajnjoj liniji, razlikuje od svih ostalih po adresi. Uobičajeni način za direktan pristup objektu na bazi adrese je upotreba *pokazivača*. Veza između pokazivača i objekta čiju adresu on sadrži nije čvrsta: pokazivač sa istim imenom, recimo pA, može u jednom trenutku da pokazuje na objekat O_1, a u drugom na O_2, pri čemu objekti O_1 i O_2 čak ne moraju pripadati istoj klasi. Isto tako, dva pokazivača mogu pokazivati na isti objekat, što se sve vidi na slici 2.2.



Slika 2.2

Uz sliku 2.2 sleduje dodatno objašnjenje. Naime, dodela vrednosti pokazivaču pA= adr(O_2) ne sme ostaviti objekat O_1 neidentifikovanim, tj. ne sme izazvati tzv. curenje memorije (memory leak, engl.)[21]. To znači da objekat O_1 mora imati alternativnu identifikaciju ili se dodela druge vrednosti pokazivaču pA mora propratiti destrukcijom objekta O_1.

Nešto čvršća veza između objekta i njegovog identifikatora, uobičajena kod baza podataka, ostvaruje se putem tzv. *ključa*. Neformalno, ključ je polje ili skup polja čije su vrednosti unikalne za svaki objekat date klase (ne mogu se pojaviti kod drugih objekata). Na primer, ključ za objekte klase *Građanin* jeste matični broj, a za objekte klase *Student* broj indeksa.

Najčvršća veza između objekta i njegovog identifikatora ostvaruje se do-

[21] Ne važi za programski jezik java.

delom *imena* pri njegovom kreiranju (tzv. konstruisanju). Usput, za akciju konstruisanja objekta široko se koristi i termin ***instanciranje klase***, nastao od alternativnog naziva za objekat - instance klase. U hibridnim jezicima poput C^{++} ili paskala ime objekta ni po čemu se ne razlikuje od imena promenljive, a instanciranje klase suštinski je isto što i definisanje promenljive (sa inicijalizacijom ili bez nje). Iako najčvršća, veza imena i objekta nije uvek uzajamno jednoznačna. Primer za to je C^{++} koji dozvoljava postojanje sinonima (alijasa) pod nazivom "referenca" što, međutim, nije nedostatak već poboljšanje u odnosu na programski jezik C.

Stanje objekta, zajedno sa njegovim ponašanjem, predstavlja ključ za razumevanje njegovih dinamičkih karakteristika. Stanje i ponašanje objekta su međusobno uslovljeni i to u oba smera. S jedne strane, stanje je deo prošlosti i sadašnjosti objekta potreban (videćemo, ne i dovoljan) za određivanje njegovog ponašanja u sadašnjosti i budućnosti. Obrnuto, stanje objekta predstavlja kumulativni rezultat njegovog ponašanja u prošlosti, što znači da se ove dve karakteristike *prožimaju*. Kao primer, možemo se pozvati na stek: odziv na operacije očitavanja, upisa ili brisanja elementa zavisi od tekućeg stanja koje je, sa svoje strane, uslovljeno hronološki prethodnim operacijama upisa-brisanja. Sličnih primera ima bezbroj.

Sledeća stvar koju treba uočiti u intuitivnoj definiciji stanja je izraz "potreban" pri određivanju njegovog upliva na ponašanje. Treba ga shvatiti u matematičkom smislu, što će reći da je za određivanje ponašanja objekta neophodno poznavati njegovo trenutno stanje, ali i još ponešto. Konkretno, ponašanje objekta, kao reakcija na aktiviranje neke metode (pobudu, poruku) u opštem služaju zavisi od tri faktora, a to su:

a) sama pobuda
b) stanje datog objekta
c) stanje drugih objekata, iste ili druge klase, koji nisu objekti-članovi, ali ostvaruju uticaj u sklopu pobude.

Neka je, na primer, klasa *Matrica* što predstavlja realnu matricu snabdevena operacijom *PomnožitiSleva* čiji je parametar realni vektor iz klase *Vektor*. Ako se stanje matrice opisuje brojem vrsta-kolona i vrednostima elemenata, očigledno je da odziv na pobudu *PomnožitiSleva* - promena stanja matrice - zavisi od same pobude, zatečenog stanja matrice *i* stanja vektora kojim se množi matrica, a koji ne pripada istoj klasi. Drugi objekat od kojeg zavisi promena stanja može biti iz iste klase, recimo ako postoji operacija množenja matrice matricom. Konačno, broj i vrsta objekata koji utiču na ponašanje datog objekta nisu ograničeni.

Prvo - i glavno - pitanje vezano za realizaciju stanja na računaru, a u okviru nekog programskog jezika, predstavlja odnos stanja objekta i odlika odgovarajućeg individualnog pojma. Ovo zbog toga što postoji dosta široko rasprostranjen stav (npr. u [23], str. 22) da stanje objekta proizlazi iz vrednosti njegovih polja. Recimo odmah, radi se o tipičnom *previdu* koji smo već ilustrovali: banana je žuta (odlika) zato što je zrela (stanje), a ne obrnuto! Razlog za pomenuti previd verovatno leži u

činjenici da je u mnogim slučajevima stanje zaista moguće odrediti iz vrednosti njegovih deskriptivnih odlika, ali samo kada postoji uzajamno jednoznačna veza (banana je žuta ako i samo ako je zrela). Iako je i navedeni primer dovoljan, navešćemo još dva, sa ciljem da se ukaže na moguće konkretne greške do kojih dolazi u procesu modelovanja, a usled pogrešnog tumačenja uzročno-posledične veze između stanja i odlike.

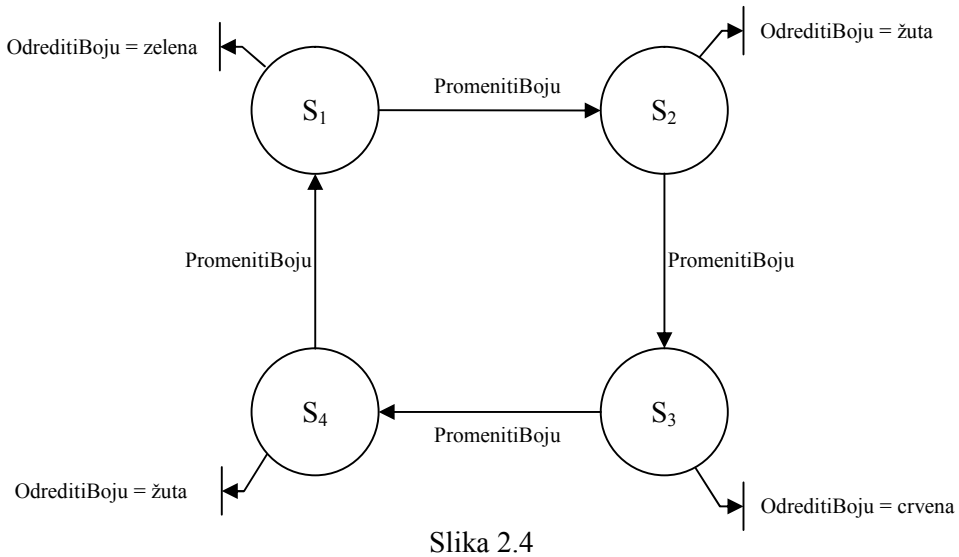
Primer 2.2. Svojevremeno su automobilski semafori funkcionisali tako što su u svakom trenutku pokazivali tačno jednu od tri boje: zelenu, žutu ili crvenu. Nema potrebe dokazivati da je BOJA odlika semafora jer ne samo da ona to jeste, nego se radi o najvažnijoj odlici (u domenu upravljanja saobraćajem). Shodno tome, prvo što projektantu pada na pamet je da u deskripciju objekta - modela semafora uključi podatak-član *boja*. Od dejstava nad semaforom sigurno je jedno od najvažnijih promena boje. Nema nikakve prepreke da se taj segment ponašanja modeluje metodom *PromenitiBoju* bez parametara, jer tako i funkcioniše semafor u realnim okolnostima[22]. Stanja semafora određena su - bar se tako čini - aktuelnom bojom, sa promenama kao na slici 2.3.

Slika 2.3

Problemi nastaju pri pokušaju realizacije. Naime, rezultat operacije *PromenitiBoju* u "stanju" *žuto* nije određen: objekat može preći u bilo koje od preostala dva "stanja". Uzrok tome nije bezazlen, a leži upravo u činjenici da ovako definisan podatak-član(!) *boja* ne određuje stanje semafora jer se odnosi isključivo na sadašnjost, tj. ne poseduje definicijom zahtevanu istorijsku komponentu. U datom slučaju poteškoća se relativno lako otklanja. Jedan od načina je da se memoriše prethodna boja (tj. prethodno "stanje") pa, ako je bila zelena, operacija *PromenitiBoju* iz žute prevodi u crvenu, odnosno ako je bila crvena prevodi u zelenu. Međutim, sada se u opisu klase pojavljuje *bivša boja* koja nije oznaka. Pored toga, postavlja se pitanje kako slične probleme rešiti kada je promena stanja objekta zavisna ne samo od tekućeg i prethodnog "stanja" nego od sadašnjeg i *n* prethodnih stanja, gde (videti sledeći primer) *n* može da bude promenljivo? Prirodnije rešenje - jer je u skladu sa definicijom - treba tražiti u takvom modelovanju stanja da opis trenutnog stanja sadrži i prošlost i sadašnjost, tj. da bude potreban *i* dovoljan za određivanje odziva. Odlike postaju, umesto elemenata opisa stanja, njegove funkcije. Dakle, odustanemo li od dijagrama stanja sa slike 2.3 i zamenimo ga dijagramom na slici 2.4 problem je rešen sam po sebi: sada postoje četiri, a ne tri,

[22] Nova vrednost boje *može* da bude zadata i eksplicitno, kao parametar, ali *ne mora*.

stanja i svako od njih u potpunosti određuje reakciju na dejstvo *PromenitiBoju* bez referisanja na bivša stanja, zato što je informacija o prošlosti ugrađena u tekuće stanje. Podatak-član *boja* više ne postoji kao takav; aktuelna boja na semaforu ustanovljava se operacijom *OdreditiBoju* sa vrednostima naznačenim na slici 2.4.



Slika 2.4

Programsko rešenje je prosto: uvodimo celobrojni podatak-član *t* sa vrednostima 0, 1, 2 i 3 takvim da svaka od njih kodira po jedno stanje (nota bene: promenljiva *t* nema nikakvu semantiku u domenu problema). Operacija *PromenitiBoju* programski se realizuje zamenom aktuelne vrednosti t vrednošću t+1 (mod 4). Metoda *OdreditiBoju* daje vrednosti odlike "boja" prema sledećoj tabeli:

t	***OdreditiBoju***
0	zelena
1	žuta
2	crvena
3	žuta

Inače, za opis stanja u ovom primeru namerno je izabrana promenljiva *t* čiji naziv nije mnemonički, da bi se podvukla činjenica da su stanja uzrok, a odlike posledica. U praksi, naravno, podaci-članovi, kad god je to moguće, imaju mnemoniku, ali samo zbog jednostavnije realizacije.

Primer 2.3. Za objekte klase *BinarnoStablo* praktično je nemoguće definisati smislene deskriptivne odlike, što znači da podaci-članovi odgovarajuće klase ne bi ni imali semantiku. Model klase obuhvatio bi samo odlike tipa *PronaćiElement*, *DodatiElement*, *UklonitiElement* i sl. Moguća implementacija sastojala bi se od

strukture podataka tipa binarnog stabla i odgovarajućih realizacija metoda. Slično je i sa drugim poludinamičkim i dinamičkim strukturama podataka (stek, red, dek, liste, n-arna i uopštena stabla, mreže).

Ostaje još da se raspravi pitanje kako se određuje prostor stanja klase. Odgovor je jednostavan: u opis stanja ulaze *svi podaci-članovi* sadržani u strukturi objekta. Tu su, dakle, podaci-članovi date klase, ali i podaci-članovi svih objekata-članova, njihovih objekata-članova itd. Posmatrajmo klasu *Duž* koja nema podataka-članova, ali ima dva objekta-člana klase *Tačka* koji predstavljaju temena duži. Ako klasa *Tačka* ima dva podatka-člana (koordinate tačke) tipa npr. *double*, tada prostor stanja klase *Duž* čine četiri podatka tipa *double* gde jedan par čine koordinate jednog, a drugi koordinate drugog temena.

Ponašanje objekta modeluje se metodama (funkcijama-članicama) koje menjaju stanje ili na bazi stanja daju vrednost deskriptivne odlike ili pak i jedno i drugo. Po opštim osobinama metode sasvim podsećaju na potprograme: imaju ime, formalne parametre i, po potrebi, generišu izlaz odnosno, kako se to obično kaže, vraćaju vrednost. Bitna, razlika između "običnih" funkcija (nazvali smo ih slobodnim) i metoda ogleda se u tome što su metode nesamostalne, tj. vezane su za klasu, što znači da se metode izvršavaju nad objektom. Tipičan poziv metode (slanje poruke) u svim objektno orijentisanim jezicima ima izgled

```
ob.met(argumenti)
```

gde se metoda *met* izvršava nad objektom *ob*, manjajući njegovo stanje ili generišići neki rezultat na bazi tekućeg stanja ili i jedno i drugo. Pored metoda, implementacija klase može da sadrži i *pomoćne (interne) funkcije* koje nisu realizacija svojstava, nego igraju istu ulogu kao pomoćni potprogrami u procedurnim programskim jezicima, s tim što su, poput metoda, vezane za klasu. Treću grupu rutina sadržanih u implementaciji klase čine *metode klase* što predstavljaju realizaciju odlika nivoa klase (npr. kreiranje objekta). Kao i slobodna funkcija, funkcija-članica ima dva dela:

1. *Signatura* što obuhvata naziv funkcije-članice, njen tip kao i spisak formalnih parametara definisanih sopstvenim nazivima i tipovima. Tip funkcije-članice odgovara tipu izlazne vrednosti ili označava da se ne generiše nikakav izlaz (poput procedura u paskalu ili void funkcija u C-u). Signature metoda su deo deklaracije klase. Inače, signatura je poznata i pod imenom zaglavlje ili prototip; imaju je i obični potprogrami.
2. *Implementacija* funkcije-članice, tj. njen programski kod na konkretnom jeziku. U skladu sa opšteprihvaćenim načelima, implementacija se fizički razdvaja od signature, osim tzv. "inline" funkcija u C^{++} koje su izuzetak, a ne pravilo. Potreba razdvajanja signature od implementacije nastala je usled toga što se u toku životnog ciklusa signatura u principu ne menja što

ne mora da važi za implementaciju, a što je, opet, posledica dobro poznatog fakta da se isti posao može algoritamski realizovati na više načina. Primera za to ima mnogo: na primer, klase koje realizuju neke tabele uvek poseduju metodu za sortiranje čija je signatura nepromenljiva, npr. *Sortirati*, bez parametara. S druge strane, poznato je da postoji čitav niz algoritama za sortiranje koji čak nemaju nikakvih sličnosti, izuzev to što obavljaju isti posao. Identičan zaključak važi i za metodu *Invertovati* klase *Matrica*. To, pak, znači da promena implementacije nema (preciznije, ne bi smela da ima) uticaja na ponašanje objekata.

Očigledno, signatura metode definiše se već na nivou modela, dok se implementacija ostavlja za fazu realizacije na računaru. Prema dejstvu na objekat, operacije i odgovarajuće metode dele se na tri grupe:

a) Metode koje menjaju stanje i ne vraćaju nikakvu vrednost. Takvim metodama dodeljuje se poseban tip (poznati tip "void" iz C/C^{++}) ili se realizuju kao posebna podvrsta metoda (*procedure* u paskalu).
b) Metode što vraćaju rezultat bez promene stanja kojima se kao tip dodeljuje tip izlaza (u paskalu se realizuju kao podvrsta *function*).
c) Metode koje rade i jedno i drugo. Poznate su pod nazivom "metode sa bočnim efektima"[23]. Rezultat vraćaju ili putem imena ili kao izlazni parametar ili i jedno i drugo ako ima više izlaza.

Pored operacija vezanih za klasu, u objektnim sistemima figurišu i slobodne funkcije (potprogrami) koje nisu deo ni jedne klase. Obično se radi o funkcijama sa više operanada iz iste klase ili iz različitih klasa, kao što je, recimo, funkcija za množenje matrice vektorom. Slobodni potprogrami su, u stvari, obični potprogrami iz paskala ili funkcije iz C-a sa formalnim parametrima objektnog ili neobjektnog tipa. Posebnu podvrstu slobodnih potprograma čine tzv. kooperativne ili prijateljske funkcije ("friend functions") u C^{++} koje se odlikuju time što imaju mogućnost direktnog pristupa svim članovima klase.

Dozvoljavanje pristupa strukturi klase, tj. njenoj "unutrašnjosti", nije pravilo već, naprotiv, izuzetak rezervisan samo za specijalne slučajeve poput kooperativnih funkcija. Tu se ne radi o nekoj ustaljenoj praksi nego o jednom od fundamentalnih principa objektne metodologije, koji će biti iscrpno tretiran u sledećim poglavljima. Na ovom mestu, zadovoljićemo se konstatacijom da su za spoljnog klijenta podaci-članovi gotovo uvek nedostupni ili, kako se još kaže, zatvoreni, a suštinski razlog leži u navedenoj činjenici da podaci-članovi primarno služe za opis stanja.

Članovi klase koji su dostupni (koristi se i izraz "vidljivi") spolja - a radi se, uglavnom, o metodama - čine ***interfejs klase***. Već samo postojanje interfejsa

[23] "Bočni efekat" je ne naročito uspešan prevod engleskog izraza "side effect" čije je značenje otprilike "nuz-pojava" ili "nuz-efekat".

tumačimo kao postavljanje određenih ograničenja pri korišćenju usluga klase. To, međutim, ne iscrpljuje listu restrikcija. U opštem slučaju, ne mogu se u svakom stanju aktivirati sve metode: primerice, prazan stek se ne može očitati niti nad njim izvršiti operacija uklanjanja elementa. Na isključenom semaforu ne može se promeniti boja. Interfejs klase snabdeven (softverski ugrađenim) pravilima za njeno korišćenje nosi naziv ***protokol***.

Već je pomenuto da se aktiviranje metode zove slanje poruke, pri čemu se isti termin upotrebljava i u modelu i na nivou implementacije, zato što za formiranje poruke nije neophodan uvid u implementaciju odgovarajuće metode. U računarskoj realizaciji slanje poruke analogno je pozivu potprograma u proceduralnim jezicima. Takođe, treba imati u vidu da su metode definisane u okviru klase, dok poruke razmenjuju objekti, tj. instance klase. Tipična poruka sastoji se iz tri komponente:

- *Receptor*: objekat kojem se šalje poruka.
- *Selektor metode*: metoda koja će se aktivirati porukom.
- *Argumenti*: stvarni parametri (argumenti) metode. Argumenti poruke i parametri metode stoje u identičnom odnosu u kojem su stvarni i formalni parametri potprograma. Naravno, postoje i metode i odgovarajuće poruke bez parametara.

Neka je *Matrica A* deklaracija instance klase Matrica sa identifikatorom A. U poruci *A.OčitatiElement(i,j)* receptor je objekat "A", selektor metode je "OčitatiElement", a parametri su "i" i "j". Poruka *A.Invertovati* nema parametara.

U odnosu na ulogu u procesu razmene poruka objekti se mogu ponašati trojako:

- Objekat šalje poruke, ali ih ne prima. Ovi objekti zovu se *klijenti*.
- Objekat prima poruke, ali ih ne šalje. U ovom slučaju govorimo o *serverima*.
- Objekat može i da prima i da šalje poruke. Takvi objekti nose naziv *agenti*.

Period između kreiranja (konstruisanja) objekta i njegovog poništavanja (destrukcije) zove se *životni vek objekta*. U toku životnog veka objekat trpi promene diktirane porukama koje prima. Na početku tog perioda objekat nema definisano stanje, što znači da je neupotrebljiv sa stanovišta prosleđivanja poruka. Na primer, konstruisanje objekta klase *Vektor* oblika *Vektor x* podrazumeva zauzimanje memorijskog prostora i dodelu identiteta, pri čemu nije obavezno da se podacima-članovima odmah pridruže vrednosti koje određuju njegovo stanje. Objektom se može rukovati tek posle dovođenja u početno stanje što se postiže ***inicijalizacijom***. U gornjem primeru inicijalizacija bi obuhvatila zadavanje broja i vrednosti komponenata. Aktivnosti konstruisanja i inicijalizacije često se združuju u jednu metodu.

Po izvršenoj inicijalizaciji, sve do destrukcije, objekat se uvek nalazi u nekom od stanja. U zavisnosti od tekućeg stanja on reaguje na poruke tako što obavlja zadati posao ili odbija poruku ako je neregularna (npr. inverzija singularne ma-

trice). Međutim, objekat nije u svakom trenutku spreman da momentalno reaguje na poruku izvršavanjem odgovarajućih rutina. Primera radi, po prijemu poruke *A.PomnožitiSleva(x)* čiji je parametar objekat x klase *Vektor*, objekat A klase *Matrica* u određenom, konačnom, vremenskom intervalu neće biti u mogućnosti da reaguje na nove poruke, sve dok se množenje ne završi. Drugim rečima, nakon prijema pomenute poruke matrica prelazi u *pseudostanje čekanja na događaj* - završetak množenja - i tek posle toga prelazi u novo stanje određeno rezultatom operacije. Vreme prelaza u novo stanje u nekim slučajevima može se zanemariti, ali to nije pravilo. Ako je, na primer, događaj na koji objekat čeka vezan za akciju korisnika sa tastature ili odziv sa mreže, interval može biti itekako osetne dužine. Zbog toga, kada govorimo o ponašanju objekta u vremenu, moramo voditi računa kako o stanjima tako i o njihovim promenama. Saobrazno tome, objekat se u toku životnog veka uvek nalazi u jednom od tri ***režima***:

1. Režim inicijalizacije koji počinje konstrukcijom objekta, a završava se postavljanjem u početno stanje. U ovom režimu objekat prima samo poruku o inicijalizaciji.
2. Ustaljeni režim u kojem objekat ima definisano stanje i reaguje na poruke.
3. Prelazni režim u kojem objekat čeka na neki događaj; nakon tog događaja vraća se u ustaljeni režim uz odgovarajući odziv.

U vezi sa programskom realizacijom objekata, a prilikom kreiranja jezika smoltok Alen Kej formulisao je sledećih 5 principa [30]:

1. Sve je objekat ("Everything is an object"). Drugim rečima, u programu nema ničeg drugog osim objekata, čak i kada su u pitanju konstante koje se interpretiraju kao konstantni objekti, tj. objekti koji ne mogu da menjaju stanje. Promenljive su isključivo instance klase. Napominjemo da u hibridnim jezicima poput C^{++} ili paskala ovaj princip nije bilo moguće poštovati do kraja. Jezik C^{++} omogućava, štaviše, istovremenu upotrebu "običnih" konstanti i konstantnih objekata.
2. Program je skup (Kej kaže "bunch") objekata koji zadaju poslove jedan drugom putem slanja poruka.
3. Svaki objekat poseduje sopstvenu memoriju sastavljenu od drugih objekata (i drugih promenljivih u hibridnim programskim jezicima).
4. Svaki objekat pripada nekoj klasi[24].
5. Svi objekti iz iste klase primaju iste poruke. Ovaj princip proističe iz definicionog zahteva da svi objekti iste klase imaju isto ponašanje.

2.3. OBJEDINJENI JEZIK MODELOVANJA - UML

U dosadašnjim razmatranjima bavili smo se pitanjima definicije objekta i

[24] U originalnoj interpretaciji Kej je upotrebio izraz "tip" umesto "klasa". Ispravku je uneo Ekel [30], jer je pojam tipa širi, tako da može doći do terminološke konfuzije.

klase, kao i njihovim karakteristikama. Pitanje kojim ćemo se pozabaviti u ovom odeljku je *kako* doći do objektnog modela domena problema. S obzirom na izrazito veliku složenost problema, taj zadatak je sve samo ne jednostavan. Podrazumeva dobro poznavanje problema kao i projektantsko znanje i iskustvo. To, još uvek, nije dovoljno: objektno modelovanje i objektna metodologija-tehnologija uopšte, zahtevaju i *sredstva*. Jedno od najpoznatijih sredstava objektne tehnologije je Objedinjeni jezik za modelovanje (Unified Modeling Language, skraćeno UML). Nastao je 1995. objedinjavanjem tri srodne metodologije: OMT (Object Modeling Technique) koju je razvio Rumba (J. Rumbaugh), Bučove (G. Booch) metodologije i Jakobsonove (I. Jacobson) metode OOSE (Object Oriented Software Engineering). Svrha UML je da obezbedi standardne alate i postupke za razvoj objektno orijentisanog softvera u fazama analize i projektovanja, delimično i implementacije.

Osnova UML je pogled na domen problema kao skup objekata i klasa koji stoje u određenim odnosima i koji podležu promenama u vremenu. Ovo je važno napomenuti jer postoje i drugi, sasvim različiti, pogledi na domen problema, a neki od njih izloženi su u [51]. Nije na odmet zapaziti da je na sličnim temeljima izgrađen i čuveni ER (Entity Relationship) model, odavno u upotrebi kod projektovanja baza podataka.

UML je gotovo u potpunosti grafički orijentisan. Model sistema prezentuje se putem dijagrama različitih formi i namene. Dijagrami su grupisani u 4 klase u skladu sa fazama životnog ciklusa i aspektima modela:

1. Dijagrami slučajeva korišćenja (Use Case Diagrams) predstavljaju sredstvo za prikazivanje rezultata analize sistema.
2. Statički model sistema sadrži opis klasa objekata, samih objekata i veza između njih. U model su uključeni dijagrami klasa (Class Diagrams) i dijagrami objekata (Object Diagrams).
3. Dinamički model sistema pokazuje kako se on ponaša u vremenu. Sadrži četiri vrste dijagrama: dijagrame sekvenci (Sequence Diagrams), dijagrame saradnje (Collaboration Diagrams), dijagrame stanja (Statechart Diagrams) i dijagrame aktivnosti (Activity Diagrams). Svaka od navedenih vrsta dijagrama pokazuje poseban aspekt dinamičkog ponašanja sistema.
4. Fizički model sistema opisuje strukturu programa i hardvera. Sastoji se od dijagrama komponenata (softver) i dijagrama razmeštaja (hardver).

UML je obimna metodologija sa mnoštvom dijagrama koji koriste različite simbole (ima i simbola-sinonima) i obiljem detalja, tako da ćemo se u ovom tekstu osvrnuti samo na najosnovnije delove, a iscrpan opis metodologije može se naći u [13] ili [23].

2.3.1. Dijagram klasa

Uloga dijagrama klasa je da na pregledan način prikaže sadržaj pojedinih klasa sistema i njihove međusobne veze. Pošto će o vezama između klasa biti reči u

posebnim odeljcima, ovde ćemo se zadržati samo na opisu pojedinačnih klasa.

Slika 2.5

Na slici 2.5 prikazan je simbol klase[25]. Simbol - pravougaonik - podeljen je na tri dela od kojih je samo deo sa nazivom klase obavezan. *Naziv klase* ima sve osobine identifikatora u standardnim programskim jezicima. U delu *polja* navode se podaci-članovi i objekti-članovi. Opšti oblik prikaza polja je

vidljivost ime[višestrukost]: tip = početna_vrednost

gde "[višestrukost]" i "početna_vrednost" nisu obavezni. Prvi simbol, "vidljivost", označava nivo zaštite pristupa objektima date klase od strane drugih objekata. Naime, jedan od temeljnih principa objektne metodologije je tzv. *princip skrivanja informacija* (engl. information hiding principle) koji, iz razloga što će biti detaljno diskutovani, ograničava ili čak onemogućava direktan pristup pojedinim članovima klase u ovom slučaju atributima. "Onemogućiti direktan pristup poljima" znači da se očitavanje i zadavanje vrednosti vrši samo posredstvom odgovarajućih operacija (metoda) koje su takođe delovi klase. Postoje tri nivoa zaštite (tri nivoa vidljivosti):

1. Pristup dozvoljen bez ograničenja. Simbol je +.
2. Pristup je ograničen (o detaljima kasnije). Simbol je #.
3. (Direktan) pristup nije moguć; moraju se koristiti metode. Simbol je -.

Ime atributa je tipičan identifikator. Višestrukost se odnosi na atribute tipa niza. Navodi se kao

[najmanji_indeks .. najveći_indeks]

npr. [1..100] (sintaksa je preuzeta iz paskala). Tip polja je jedan od standardnih tipova koji se sreću u programskim jezicima (integer, real, string, ...), ali i izvedenih tipova sa dosegom u datom domenu problema. Početna vrednost navodi se ako to ima smisla. Konačno, ako je polje nivoa klase kompletna deklaracija se podvlači. Primeri su

[25] Slika sadrži samo najvažnije delove simbola; ima ih još, ali nisu od interesa za ovo izlaganje.

- Prezime: String
+ BrojPrimeraka: Integer
- x: Real
- y: Real

gde polje BrojPrimeraka pripada klasi, a ne pojedinačnim objektima. Treći deo simbola sadrži *metode*. Ne mora se navesti, kao ni deo sa poljima. Ako se polja ne navedu, a metode navedu, deo sa poljima postoji i ostavlja se prazan. Sintaksa metode ima sledeći izgled:

stereotip vidljivost naziv_metode(lista_parametara): tip_rezultata

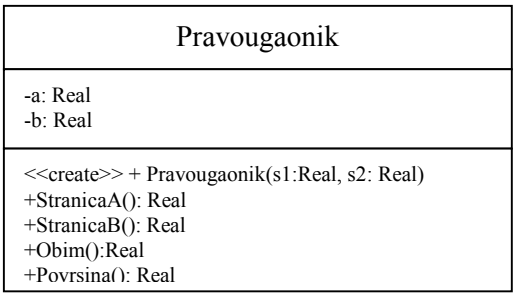
Stereotip je mehanizam koji se dosta intenzivno koristi u UML, a ima prirodu službene (rezervisane) reči. To je kraći tekst oblika <<tekst>> čiji sadržaj ima određeno, nepromenljivo značenje. Prilikom definisanja metode, stereotip može biti <<create>> kada operacija služi za kreiranje (konstrukciju) objekta, odnosno <<destroy>> ako se radi o poništavanju (destrukciji). Vidljivost i naziv metode imaju istu interpretaciju kao i kod polja. Lista parametara sadrži opis parametara - kao kod potprograma - s tim što se svaki parametar predstavlja sintaksnim konstruktom

naziv_parametra : tip_parametra = podrazumevena_vrednost

gde je mogućnost uvođenja podrazumevane vrednosti (neobavezna je) direktno preuzeta iz programskog jezika C i ima iste osobine. Ako ima više parametara razdvajaju se zapetama. Ako nema parametara navode se samo zagrade. Konačno, ako operacija vraća vrednost, mora se naznačiti tip, a u suprotnom ovaj deo se ispušta. Metode nivoa klase se takođe podvlače. Primeri:

+ PromenitiKoordinate (x: Real, y: Real)
+ Invertovati()
<<create>> + Complex(re: Real= 0, im: Real =0)

Na slici 2.6 prikazan je kompletan primer klase *Pravougaonik*.



Slika 2.6

2.3.2. Dijagram stanja

Dijagram stanja je tipičan primer grupe dinamičkih dijagrama jer povezuje skup stanja i ponašanje objekata, i to objekata jedne klase. Nije izum objektne metodologije jer je primenjivan u nešto jednostavnijem obliku još kod prvih prikaza ponašanja diskretnih sistema. Svrha dijagrama stanja date klase je da se na prikladan način prezentuje ponašanje objekata. Sastoji se od simbola stanja, oznaka promena stanja i još nekih simbola od kojih za sada izdvajamo dva: simbol početka i simbol završetka (u terminologiji UML početak i završetak zovu se pseudostanja).

Stanje objekta predstavlja se pravougaonikom sa zaobljenim temenima, koji može biti podeljen na dva polja, slika 2.7. U gornje se upisuje ime stanja, a u donje opis ponašanja pri ulasku u stanje, izlasku iz njega i u vreme opstajavanja u stanju. Ni jedan od navedenih delova nije obavezan.

Slika 2.7

Oznaka *do/aktivnost* definiše sekvencu akcija (dejstava) koja se izvršava u stanju, dok oznake *entry* i *exit* određuju aktivnosti koje se vrše prilikom ulaska u stanje, odnosno pri njegovom napuštanju.

Promene stanja prikazuju se strelicama koje povezuju odgovarajuća stanja. Strelice se snabdevaju oznakama vezanim za neke od sledećih stavki (ili sve):

- aktivnost koja izaziva promenu stanja
- uslov koji mora biti ispunjen da bi došlo do promene stanja

- aktivnost koja se izvršava u toku promene stanja.

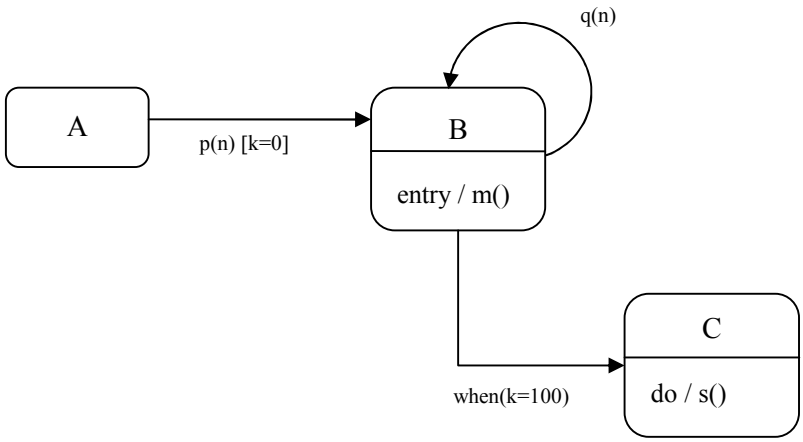
Sintaksa oznake je sledeća:

naziv_aktivnosti(lista_parametara) [uslov] / aktivnost

gde su elementi liste parametara isti kao kod sintakse operacije.

Na slici 2.8 promena stanja iz A u B inicira se porukom *p(n)* koja je deo definicije klase na dijagramu klasa. Do promene stanja doći će samo ako je k = 0. Kako se vidi, neke poruke, poput *q(n)* sa slike 2.8 zadržavaju objekat u istom stanju, ali sa promenjenom vrednošću nekih podataka-članova. Pri svakom ulasku u stanje B izvršava se aktivnost *m*. Od oznaka promene treba izdvojiti još jednu, prikazanu na slici 2.8,

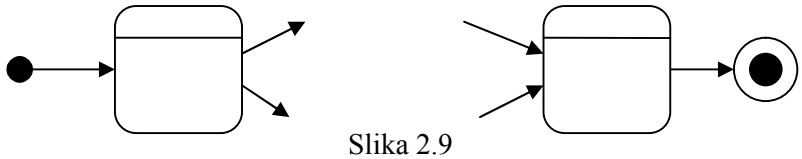
when(uslov)



Slika 2.8

koja se interpretira sa: kada je uslov ispunjen obavezno dolazi do odgovarajuće promene stanja. U gornjem primeru, kada u stanju B vrednost k postane jednaka 100 dolazi do obaveznog prelaza u stanje C.

Početno pseudostanje označava se popunjenim krugom, a završno popunjenim krugom zaokruženim još jednim, slika 2.9.



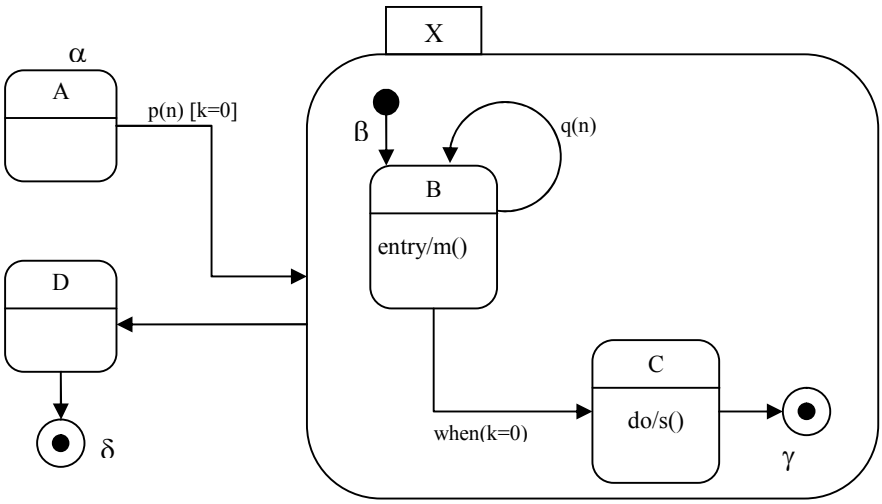
Slika 2.9

Pseudostanje početka pokazuje početno stanje objekta ili početno podstanje

složenog stanja (videti dalje). Završno pseudostanje odgovara destrukciji objekta ili izlasku iz složenog stanja.

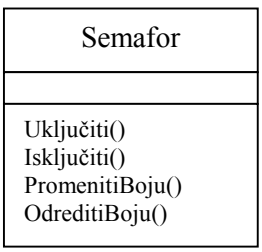
Postoje stanja objekta koja su složena, tj. sastoje se od podstanja. Projektant ih definiše u slučaju kada želi da izoluje deo ponašanja sistema. Simbol je isti kao i simbol stanja, samo što se naziv daje u pravougaoniku iznad oznake, a sadržaj simbola je subdijagram dijagrama stanja, slika 2.10.

Na slici 2.10 simbol α pokazuje da je *A* početno stanje objekta, simbol β da se po ulasku u složeno stanje *X* prelazi u stanje *B*, simbol završetka γ da po izlasku iz stanja *C* objekat više nije u složenom stanju *X*, a simbol završetka δ da je objekat poništen (izvršena je operacija destrukcije).



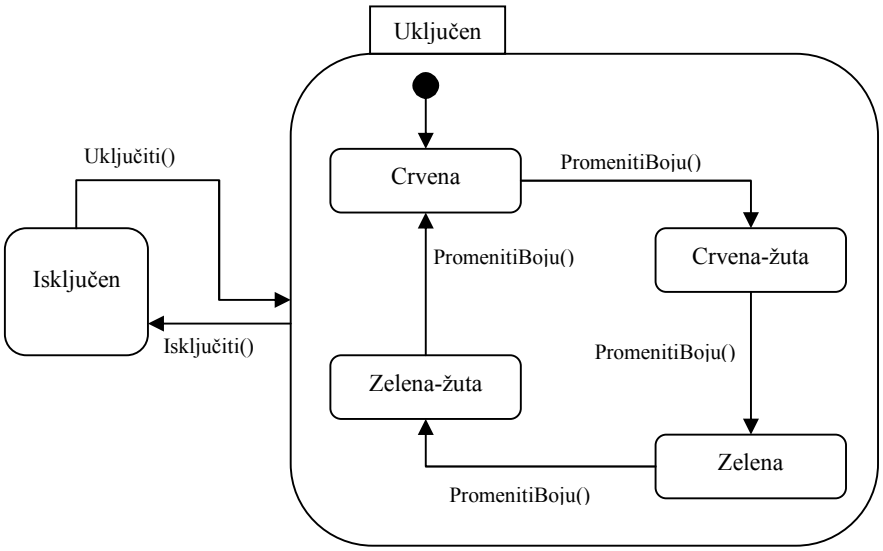
Slika 2.10

Primer 2.4. Kao primer razmotrićemo opet klasu *Semafor* iz primera 2.2, ali ovog puta nešto formalnije i detaljnije. Pored promena boje predvidećemo i mogućnost uključivanja odnosno isključivanja semafora, s tim da pri uključenju semafor treba da pokaže npr. crvenu boju. Simbol klase *Semafor* ima sledeći oblik:



Slika 2.11

Dijagram stanja prikazan je na slici 2.12.



Slika 2.12

Zapazimo da je "uključen" složeno stanje tako da stanje nazvano "crvena", znači, u stvari, "uključen *i* crvena". Takođe, nazivi stanja su mnemonički.

3. TEMELJI OBJEKTNE METODOLOGIJE

Objektna metodologija zasniva se na nekoliko osnovnih principa i tehnika od kojih - slobodno se može reći - nijedna nije nastala u sklopu objektne paradigmom, a jedna od njih - apstrahovanje - koristi se bukvalno oduvek. Čak i ako se ograničimo samo na računarstvo, ispostavlja se da su svi osnovni principi i tehnike bili poznati od ranije, samo što nisu igrali (bar ne svi) presudnu ulogu pri izradi kvalitetnog softvera.

Principe i tehnike koji sačinjavaju temelje objektne metodologije razni autori grupišu na različite načine i pridaju im različitu važnost. Upravljajući se mišljenjem većine, u osnovne elemente objektne metodologije ubrojaćemo sledeće:

1. Apstrakcija i skrivanje informacija
2. Inkapsulacija i modularnost
3. Polimorfizam
4. Veze između klasa, a naročito nasleđivanje.

Napominjemo da elemenata objektne metodologije ima još (Buč npr. pominje i tipiziranje), ali na njima se nećemo posebno zadržavati.

Apstrahovanje i njegov rezultat - apstrakciju već smo pominjali u prvom poglavlju kao sredstvo za sučeljavanje sa kompleksnim problemima, ne samo u računarstvu nego u gnoseologiji uopšte. Pod pojmom *skrivanja informacija* (engl. "information hiding") podrazumeva se princip koji nalaže da se spreči ili ograniči pristup pojedinim komponentama softverskog sistema (u objektnoj metodologiji radi se o članovima klase). Konkretnije, princip skrivanja informacija u izvornom obliku zahteva da detalji realizacije budu nedostupni klijentu.

Inkapsulacija je tehnika kojom se pri modelovanju i realizaciji objedinjavaju podaci-članovi, objekti-članovi i funkcije-članice (metode), poštujući pritom princip skrivanja informacija. *Modularizacija* predstavlja tehniku razlaganja složenog softverskog sistema u domenu implementacije na jednostavnije komponente koje imaju unutrašnju logiku. Kako i sama objektna metodologija podrazumeva razlaganje sistema na klase, moduli i klase nisu nezavisne softverske jedinice; naprotiv, među njima postoje međuzavisnosti o kojima će biti reči u narednom poglavlju.

Polimorfizam je osobina koju je najteže definisati jezgrovitim iskazom.

Sama reč nastala je od grčkih reči "poly" koja znači "mnogo" i "morfe" sa značenjem "oblik", tako da izraz može da se protumači kao "uzimanje više oblika". Nešto konkretnije, polimorfizam je osobina (ili pak mogućnost) da se softverska komponenta ponaša zavisno od konteksta ili okolnosti. Na primer, paskalska funkcija *succ(x)* generisaće izlaz 'b' ako je x znakovna promenljiva sa tekućom vrednošću 'a', ili 6 ako je x celobrojna promenljiva sa tekućom vrednošću 5. Slično tome, promenljivom C-a koja je tipa *char* može se slobodno rukovati kao promenljivom celobrojnog tipa. Shodno tome, u teoriji tipova ([3], str. 72, 115) polimorfizam se pominje kao osobina da određena promenljiva ili vrednost pripada većem broju tipova (koji, međutim, nisu potpuno nezavisni). Konačno, postoji i takav pogled na polimorfizam koji ga tretira kao dovođenje u vezu imena i tipa.

Vezama se ostvaruje uređenje apstrakcija. Naime, pojmovi, klasni i individualni, u domenu problema ne egzistiraju pojedinačno, izolovano već se, naprotiv, nalaze u najrazličitijim međusobnim odnosima. Ti odnosi modeluju se vezama između njihovih modela - objekata odnosno klasa što je, uostalom, i suština ranije navedenih principa Alena Keja. Među mogućim vezama između klasa, u objektnoj metodologiji najveću važnost, sa stanovišta i modela i implementacije, ima tzv. *nasleđivanje* (engl. "inheritance"). Radi se o specifičnoj vrsti veze koja omogućuje prenos osobina jedne klase na drugu uz mogućnost dopune i modifikacije. Na primer, putem nasleđivanja može se oformiti klasa *MyWindow* od gotove klase *Window* preuzimanjem svih članova te klase (a ima ih mnogo i nisu ni malo jednostavni za realizaciju) uz dodavanje osobina specifičnih za novu klasu. Zapazimo da je to vrlo delotvoran način za rešavanje ranije pomenutog problema ponovnog korišćenja softvera ("software reuse"). U praksi, relacija nasleđivanja se najčešće formira tako da klase uređene tom relacijom čine strukturu tipa stabla, te verovatno stoga Buč umesto opštijeg termina "veze" za ovaj element objektne metodologije upotrebljava izraz "hijerarhija ("hierarchy").

3.1. APSTRAKCIJA I SKRIVANJE INFORMACIJA

Od svih elemenata objektne metodologije apstrakcija je svakako najopštiji jer se susreće u svim oblastima ne samo nauke nego i umetnosti. Usled toga, njenu definiciju treba tražiti van računarstva - u filozofiji. "Filozofijski rečnik" [57] ***definiše apstrakciju*** preko postupka apstrahovanja:

- "Apstrahovati (lat. abstrahere = odvući), najopštije: odvojiti nešto od nečega, ostaviti nešto po strani, ne obratiti pažnju na nešto; nešto uže: ostaviti po strani ono nebitno, sporedno, slučajno [...] zadržavajući ono bitno i opšte [...]"
- "Apstrakcija: [...] radnja kojom izostavljamo pojedinačno, slučajno, sporedno, a zadržavamo opšte, bitno, nužno, važno. Takođe: rezultat te radnje [...]"

Vidimo, dakle, da apstrakcija[26] nema za svrhu samo rešavanje kompleksnih problema: suština joj je znatno dublja i zadire u same osnove ljudskog mišljenja.

U prirodnim naukama i matematici apstrakciju susrećemo na svakom koraku. Već ako definišemo skup S = {x | P(x)} tako što u njega uključimo sve elemente koji zadovoljavaju predikat P izvršili smo apstrahovanje: sve sličnosti i razlike između elemenata skupa svedene su na jednu jedinu, a to je da li zadovoljavaju predikat P ili ne. Slično, u skupu {voz, avion, helikopter, brod, automobil} element "avion" predstavlja sve avione, klipne, mlazne, putničke, transportne itd.

Apstrakcija se sreće i u umetnosti, u svojstvu načina ekspresije, a sa ciljem isticanja onog što umetnik smatra bitnim u svom delu. Takođe je i u umetnosti apstrakcija poznata oduvek (setimo se samo crteža životinja u pećini Lasko). Ipolit Ten (Hipolite Tain), klasični filozof umetnosti, u knjizi "Filozofija umetnosti" ističe: "Svrha umetničkog dela je da otkrije neku bitnu ili istaknutu karakteristiku, a potom i neku važnu ideju, jasnije i potpunije no što to čine istinski predmeti". Nije na odmet istaći da Ten nema u vidu ono što se danas smatra apstraktnom umetnošću nego, naprotiv, umetnost u svim vremenima. Pikasova "Gernika" savršeno jasno prikazuje užas rata i ljudskog stradanja, iako je sve samo ne fotografski realistična.

Slika 3.1

U računarstvu na apstrakciju nailazimo u svim segmentima: svaki program, čak i asemblerski[27] predstavlja apstrakciju izvedenu iz mašinskog koda; kod procesora

[26] U daljem tekstu termin "apstrakcija" upotrebljavaćemo i kao radnju i kao rezultat te radnje, u skladu sa navedenom definicijom.

[27] Šoova [70] smatra asemblerski jezik za prvu primenu apstrakcije u programiranju.

nas retko interesuju detalji realizacije itd. U toku životnog ciklusa softverskog sistema apstrakcija se pojavljuje u svim fazama, od definisanja zahteva, specifikacije i modelovanja do neposredne realizacije. Štaviše, glavna linija razvoja softverskog proizvoda može se shvatiti kao niz prelazaka sa viših na niže (konkretnije) nivoe apstrakcije.

Ako se detaljnije pozabavimo navedenom filozofskom definicijom apstrakcije primetićemo da ona krije bar dva odnosa važna za razumevanje uloge apstrakcije u izradi softvera. To su odnos bitnog (važnog) prema nebitnom (sporednom) i odnos opšteg prema pojedinačnom. Zavisno od toga koja se uloga stavlja u prvi plan, postoje različite uloge apstrakcije u računarstvu.

Tako Hoar u [7] apostrofira odnos bitno - nebitno kada govori o apstrakciji koja je rezultat "prepoznavanja sličnosti između nekih stvari[28], situacija ili procesa u realnom svetu i odluke o koncentrisanju na te sličnosti uz ignorisanje razlika". Šoova (M. Shaw, [70]) je na sličnom stanovištu definišući apstrakciju kao pojednostavljeni opis ili specifikaciju sistema koji ističu neke od detalja, dok neke zanemaruju ("suppress"). Ovaj tip apstrakcije Buč naziva *apstrakcijom entiteta*[29], (entity abstraction, [3]).

Tipičan primer apstrakcije entiteta jeste materijalna tačka u kinematici. Još jedna, računarska, ilustracija je semafor iz primera 2.2 i 2.5 gde se kao bitne karakteristike ističu aktuelna boja i mogućnost njene promene odn. uključivanja - isključivanja. Ostale osobine poput, recimo, marke, proizvođača, godine proizvodnje ili cene zanemaruju se. Uočimo da je i sam pojam po prirodi apstrakcija što direktno sledi iz definicije.

Bercins, Grej i Nauman ([71], cit. [3]) posmatraju apstrakciju iz drugog ugla kada kažu "da se pojam kvalifikuje kao apstrakcija samo ako može da se opiše, razume i analizuje nezavisno od mehanizma kojim će biti realizovan". Ovaj pogled izvodi se, u stvari, iz odnosa opšte - pojedinačno zato što se ista apstrakcija može realizovati na više načina. Tako, tačku u Dekartovom koordinatnom sistemu možemo programski realizovati kao dvokomponentni realni vektor ili pak kao slog, a da to ne utiče na naše shvatanje ovog pojma. Kada programer napiše

$$y = \log(x*z-3)$$

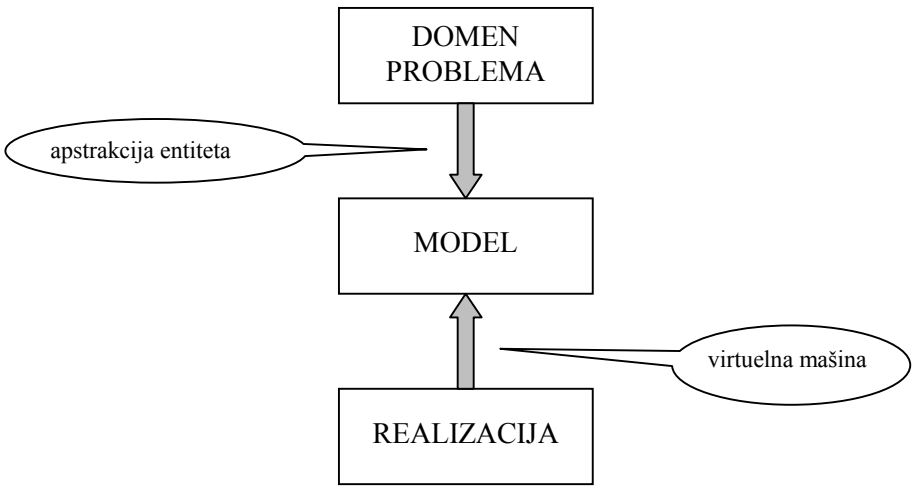
njega ne interesuje (najčešće ni ne zna) kako se memorišu x, y, z i 3 niti kako se računaju x*z-3 i (naročito) logaritam. Štaviše, ponašanje gornjih naredbi invarijantno je u odnosu na promene verzije kompilatora. Dajkstra u [7] navodi: "Postoji jaka analogija između korišćenja operacije u programu nezavisno od načina na koji je realizovana i korišćenja teoreme nezavisno od načina na koji je ona dokazana".

[28] Hoar upotrebljava izraz "objekat" koji smo, iz razumljivih razloga morali da prevedemo sa "stvar".
[29] Izraz "entitet" treba shvatiti kao "stvar" (videti prethodno poglavlje).

Koncepcija modula u C/C^{++} (zaglavlje + kod u izvornoj datoteci) i paskalu ("unit" sa delovima "interface" i "implementation") direktno je izvedena iz ovog ugla posmatranja. U javi je koncepcija u osnovi ista, ali se u detaljima ponešto razlikuje.

Martin stoji na istom stanovištu kada apstrakciju definiše kao "eliminaciju irelevantnog i isticanje esencijalnog" ([2], str. 9). Iako na prvi pogled ova definicija pripada prvoj grupi, on odmah u nastavku daje primer automobila koji vozač "vidi" samo kroz komande, dok su detalji realizacije za njega irelevantni.

Ovu vrstu apstrakcije Buč u [3] naziva *virtuelnom mašinom* (virtual machine abstraction). Naziv je očigledno inspirisan konceptom virtuelne mašine koji je izložen u [7], a radi se o hipotetičkom računaru koji direktno izvršava naredbe pisane na višem programskom jeziku, tj. njegov "mašinski" jezik sastoji se od naredbi višeg programskog jezika. Na apstrakciju ovog tipa direktno nailazimo kako kod slobodnih potprograma tako i kod klase. U oba slučaja interakcija sa klijentom obavlja se putem interfejsa, samo što on kod potprograma ne nosi to ime nego se zove zaglavlje i sastoji se od imena, parametara i tipa ili vrste potprograma. Drugi konstitutivni element potprograma sadrži lokalne promenljive, imenovane konstante, eventualno lokalne tipove i, naravno, kod izvršnog dela. Ova komponenta obično se zove telo potprograma ("procedure body", engl.) ili blok i po pravilu je nedostupna klijentu. Situacija sa klasom je gotovo identična: komunikacija se obavlja preko ***interfejsa klase*** sastavljenog od dostupnih članova, a neposredna realizacija članova klase (***telo klase***) zaštićena je od neautorizovanog pristupa primenom principa skrivanja informacija. I potprogram i klasu klijent "vidi" samo kao interfejs, ne vodeći računa o detaljima realizacije.



Slika 3.2

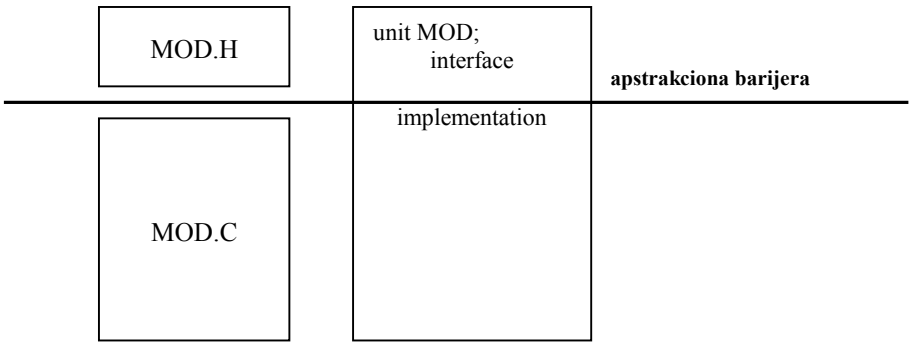
U objektnoj metodologiji apstrahovanje i apstrakcija kao rezultat pojavljuju se na dva mesta i to

- prilikom modelovanja gde se bavimo pojmovima, koji su po definiciji apstrakcije i
- pri realizaciji kada apstraktni model snabdevamo detaljima realizacije.

U prvom slučaju deluje apstrakcija entiteta jer se model dobija zadržavanjem relevantnih oznaka uz zanemarivanje onih koje to nisu. Rezultat apstrahovanja jeste model, kako klase tako i odgovarajućih objekata.

U fazi prelaska sa modela na realizaciju, klasa kao model dopunjuje se detaljima realizacije, naravno na ciljnom programskom jeziku: definišu se podaci-članovi i objekti-članovi, a prototipovi metoda razvijaju se u kompletan kod. Uočimo da ulogu apstrakcije ponovo ima model jer se, u ovom slučaju, radi o apstrakciji virtuelne mašine.

Granica koja razdvaja model od njegove računarske implementacije, nezavisno od toga da li se radi o objektnoj ili nekoj drugoj metodologiji, nosi naziv *apstrakciona barijera*. U procedurnim jezicima C i paskal, apstrakciona barijera se prepoznaje kod modula, jer se tehnički izvodi kao granica između zaglavlja (*.H) i izvorne datoteke (*.C) kod C-a, odnosno kao granica između segmenata "interface" i "implementation" u paskalskom modulu (slika 3.3).



Slika 3.3

Kada su u pitanju objektne verzije, C^{++}, java ili objektni paskal, apstrakciona barijera se pojavljuje na liniji razdvajanja interfejsa od tela klase. Inače, pravilno definisana apstrakciona barijera obezbeđuje vrlo važnu karakteristiku klase, a to je *invarijantnost ponašanja u odnosu na promene realizacije*.

U bliskoj vezi sa apstrakcijom virtuelne mašine je poznati ***princip skrivanja informacija*** (Information Hiding Principle). Princip skrivanja informacija je hronološki stariji od objektnog programiranja. Formulisao ga je Parnas još 1972. kao jedan od kriterijuma za dekompoziciju složenog sistema na module, [74][30]. Pošto se pominje u vezi sa načinom konstruisanja modula uopšte, Martin [2]

[30] Neki autori, na primer Buč [3, str. 49], princip skrivanja informacija implicitno shvataju kao *meha-*

ga formuliše na sledeći način:

- Sve informacije o modulu moraju biti skrivene ("private") osim onih koje su eksplicitno deklarisane kao javne ("public").

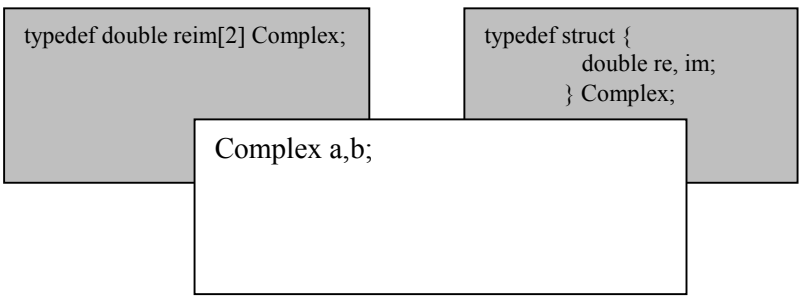
U objektnoj metodologiji princip skrivanja informacija izvodi se, kao što smo rekli, iz virtuelne mašine i može se formulisati pravilom:

- Neposredna realizacija članova klase (tj. telo klase) mora biti nedostupna klijentu.

Lako se uočava da liniju razdvajanja dostupnog (otvorenog, javnog) i nedostupnog (zatvorenog, skrivenog) dela, kako klase tako i modula uopšte, čini apstrakciona barijera, tako da princip skrivanja informacija možemo izraziti i ovako:

- deo implementacije iza apstrakcione barijere mora biti nedostupan klijentu.

Upravo princip skrivanja informacija obezbeđuje invarijantno ponašanje klase (i svakog modula) u odnosu na izmene u realizaciji, što je prikazano na slici 3.4. Na slici se vidi da tip (zasad se ograničavamo na tipove podataka, ali isto važi i za klase) *Complex* možemo realizovati kao dvokomponentni vektor ili kao slog, ali klijent to ne sme da oseti.



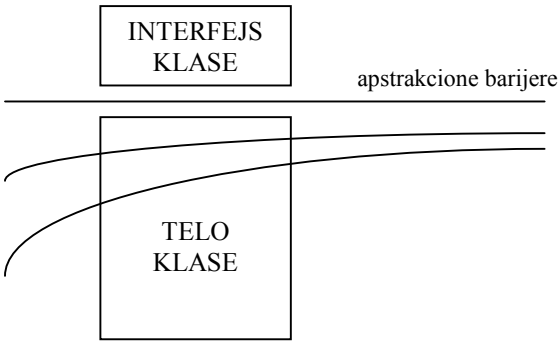
Slika 3.4

U originalnom obliku princip skrivanja informacija je isključiv: ako je nešto dostupno, dostupno je svim klijentima, a nedostupni delovi su zatvoreni za sve. U toku razvoja objektne metodologije princip skrivanja informacija je razrađen, tačnije evoluirao je u *skalu nivoa zaštite* članova klase (kaže se i *kontrole pristup*a). Stupnjevi zaštite, umesto pravila "ili-ili ", obezbeđuju da neki članovi budu dostupni svakom klijentu, neki ni jednom, a neki pak samo klasama sa različitim posebnim ovlašćenjima. To samo znači da apstrakciona barijera nije više "pravolinijska" kao na slici 3.3, kao i da nije ista za sve klijente, što je prikazano na slici 3.5.

U praksi, za sada, koriste se dva do četiri nivoa zaštite. Paskal ih ima dva, C^{++} tri, a java četiri. Posebno, C^{++} ima još jedan mehanizam za kontrolu pristupa, tzv. kooperativne ili prijateljske ("friend") funkcije i klase kojima se eksplicitno

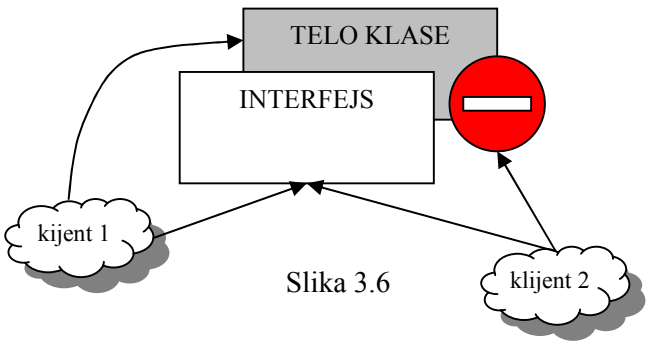
nizam za ostvarivanje inkapsulacije. Mi ćemo se držati originalnog, Parnasovog, shvatanja.

omogućuje pristup internim delovima date klase.



Slika 3.5

Postojanje više nivoa zaštite (tj. više apstrakcionih barijera) zahteva neka razjašnjenja u pogledu definicije interfejsa, odn. tela klase. Radi se o tome da u slučaju stroge primene principa skrivanja informacija (jedinstvena) apstrakciona barijera jasno razgraničava interfejs od tela klase jer interfejs sadrži članove koji su dostupni svim klijentima, a telo članove koji nisu dostupni ni jednom klijentu. Međutim, ako ima više apstrakcionih barijera, određenih različitim nivoima zaštite, postavlja se pitanje šta je interfejs, a šta telo klase?



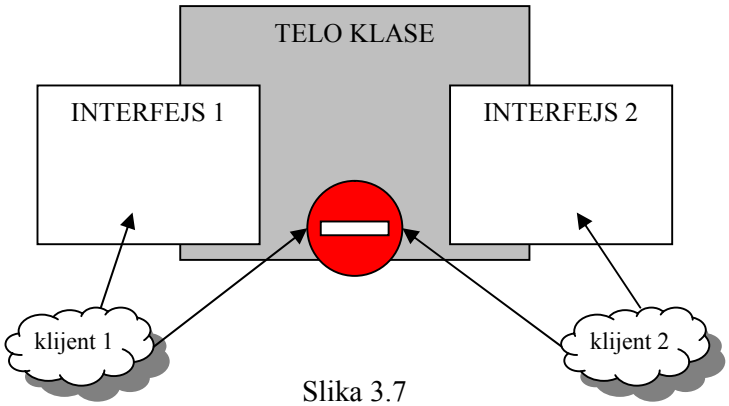
Slika 3.6

Postoje dva, alternativna, pogleda na ponašanje klase u smislu komunikacije sa klijentima:

1. Ako pod interfejsom klase i dalje podrazumevamo njene članove koji su dostupni *svim* klijentima, tada određeni, "privilegovani" klijenti mogu da pristupe i telu klase, dok "obični" klijenti tu mogućnost nemaju (slika 3.6).
2. Ako pod interfejsom podrazumevamo članove klase dostupne *određenoj kategoriji* klijenata, tada klasa komunicira sa klijentima kao da ima *više interfejsa*, po jedan za svaku kategoriju klijenata. U ovom slučaju telo klase je onaj njen deo koji nije dostupan ni jednom klijentu (slika 3.7).

Ako eksplicitno ne naznačimo suprotno, pod interfejsom ćemo podrazumevati slu-

čaj 1, tj. skup članova klase dostupan svim klijentima.



Slika 3.7

U načelu, glavni kandidati za skrivanje (zatvaranje) su podaci-članovi *zato što modeluju stanje i ne moraju imati nikakvu semantiku*, a zatim i objekti-članovi. Funkcije-članice zatvaraju se u slučaju kada su interne (pomoćne). Strogo gledano, princip skrivanja informacija zahteva da *svi* podaci-članovi budu zatvoreni i da se pristup ostvaruje putem posebnih funkcija-članica (metoda), što uglavnom važi i za objekte-članove. Inženjerska praksa, kao i obično, dozvoljava odstupanja od tog pravila, tada kada programer smatra da za to ima razloga. Na primer, ako se vrednost podatka-člana i čita i menja (tj. on ima semantiku), umesto da u tu svrhu predvidi dve metode, programer može jednostavno da ga ostavi otvorenim, smanjujući na taj način broj metoda koji se neretko meri desetinama. Inače, za odluke ovog tipa postoji "pravilo tri prsta": ako se želi odstupiti od dosledne primene pravila skrivanja informacija programer za to mora, barem samom sebi, pružiti objašnjenje.

3.2. ELEMENTARNA IMPLEMENTACIJA KLASE

Programski jezik C^{++} je hibridni jezik zahvaljujući činjenici da je izveden iz procedurnog jezika C i da je zadržao sve njegove elemente. Može se, štaviše, slobodno reći da se radi o jednom jeziku sa procedurnom komponentom pod nazivom C i objektnom komponentom koja se zove C^{++}, a sa punim nazivom C/C^{++}. Pored toga, neke novine uvedene u C^{++} retroaktivno su prenete i na C: na primer upotreba referenci za prenos argumenata po adresi, tzv. *inline* funkcije, rukovanje izuzecima pomoću vrednosti ili generičke funkcije. To, dakle, znači da u programskom jeziku C/C^{++} paralelno egzistiraju klase i tipovi podataka, te slobodne funkcije i funkcije-članice što je u velikoj meri uticalo na njegovu strukturu.

Kao i procedurni tipovi podataka, klasa se deklariše (definiše) posebnom naredbom, s tim što se kao značajna razlika pojavljuje potreba za definisanjem funkcija-članica. Funkcije-članice se po opštem obliku skoro i ne razlikuju od slo-

bodnih funkcija, sem što nisu nezavisne nego su deo definicije klase. Sastoje se, dakle, od zaglavlja sa nazivom, tipom i parametrima i tela (bloka) sa lokalnim promenljivim i opisom algoritma. Prilikom aktiviranja slanjem poruke, parametri bivaju zamenjeni argumentima po istim pravilima koja važe za sve potprograme. Podaci-članovi realizuju se na identičan način kao polja kod slogova (koji se u C/C^{++} iz nekog razloga nazivaju strukturama). Isto važi i za objekte-članove, ali uz napomenu da su oni, za razliku od polja, aktivni i mogu primati iste poruke kao i nezavisne instance klase kojoj pripadaju. I podaci-članovi i (naročito) objekti-članovi često zahtevaju inicijalizaciju zadavanjem početnih vrednosti, odnosno dovođenjem u početna stanja ako se radi o objektima-članovima. Imajući u vidu ovo dvojstvo - podaci i objekti članovi s jedne i funkcije-članice s druge strane - definicija klase sastoji se fizički iz dva dela: prvi deo sadrži opis podataka i objekata-članova kao i zaglavlja (prototipove) funkcija-članica[31] i realizuje se jednom naredbom. Drugi deo sadrži kompletan opis funkcija-članica, sa zaglavljem i blokom i nije deo naredbe za definisanje klase.

3.2.1. Implementacija klase u C^{++}

U programskom jeziku C^{++} klasa se ***definiše*** posebnom naredbom iz sintaksne familije koju sačinjavaju još naredbe *struct* i *union*. Opšti oblik naredbe je

```
class NazivKlase {
        deklaracije članova klase
};
```

gde je znak ";" obavezan. Naziv klase konvencionalno počinje velikim slovom, kao i svaka nova reč u imenu (*Point*, *AlarmClock*, *RandomAccessFile*). Programski jezik C^{++} tako je koncipiran da maksimalno motiviše programera da se pridržava principa skrivanja informacija, tako da je gornji oblik naredbe *class* praktično neupotrebljiv jer se *podrazumeva* da su članovi klase *zatvoreni*, a oni koji su otvoreni moraju se eksplicitno navesti kao takvi. Ovo se postiže deljenjem naredbe na dva segmenta: jedan je zatvoren i može se, ali ne mora posebno označiti, dok je drugi otvoren i mora se eksplicitno označiti. Segment koji je dostupan svim klijentima prepoznaje se po tome što počinje labelom

```
public:
```

iza koje slede deklaracije članova klase dostupnih svim klijentima. Dakle, stvarni opšti oblik naredbe *class* je

[31] C^{++} ima mogućnost da se umesto prototipa zada cela funkcija-članica (videti dalje).

```
class NazivKlase {
        deklaracije zatvorenih članova klase
public:
       deklaracije otvorenih članova klase
};
```

Zatvoreni članovi klase mogu se takođe označiti eksplicitno, korišćenjem labele

```
private:
```

tako da je alternativa gornjem obliku definicije

```
class NazivKlase {
private:
        deklaracije zatvorenih članova klase
public:
       deklaracije otvorenih članova klase
};
```

pri čemu je sada redosled navođenja dvaju segmenata proizvoljan. Na ovom mestu skrećemo pažnju na jedan potencijalni nesporazum: kada za neki član klase kažemo da je zatvoren, to nikako ne znači da nije dostupan nikom, jer bi to bila besmislica. Unutar klase ne važi nikakva zaštita (kontrola pristupa) tako da je stvarna interpretacija kontrole pristupa tipa "private"

- članovi klase kontrolisani nivoom "private" dostupni su samo metodama te klase (tj. mogu se neposredno koristiti u kodu tih metoda).

Dodajmo i to da C^{++} ima tri, a ne dva nivoa zaštite, te prema tome naredba *class* u opštem slučaju ima tri segmenta. Treći segment je poseban po tome što sadrži članove klase koji su dostupni samo nekim, specijalnim klijentima, tako da se može smatrati "poluotvorenim". Stvar je shvatanja da li će taj deo biti logički uvršten u interfejs ili neće. Isto tako, ugao gledanja može se promeniti tako da se smatra da klasa ima dva interfejsa: jedan je onaj kojim se komunicira sa običnim klijentima, a drugi onaj kojim se komunicira sa klijentima sa posebnim ovlašćenjima (i koji ima i treći segment). O trećem segmentu biće reči kasnije kada se budemo pozabavili vezom nasleđivanja. Inače, deklaracije članova klase imaju istu formu u svim segmentima.

Podaci-članovi klase deklarišu se potpuno isto kao polja kod struktura, dakle konstruktom

```
tip imePodatka;
```

i mogu da budu bilo kojeg standardnog ili programerski definisanog tipa. Imena podataka-članova, objekata-članova i metoda biraju se po konvenciji koja u poslednje vreme preovlađuje, najverovatnije pod uticajem jave. U svim navedenim slučajevima ime počinje malim slovom, a svaka nova reč u imenu velikim[32]. Ako više podataka-članova ima isti tip mogu se zadati jednom deklaracijom oblika *tip* ime_1, ime_2, ..., ime_n. Primeri deklaracija podataka-članova su

```
int a, b;
double t[100];
Visina h;
```

Objekti-članovi deklarišu se navođenjem naziva njihove klase i nazivom objekta-člana, tako da de facto nema razlike u sintaksi u odnosu na deklaracije podataka-članova. Ako su *v* iz klase sa nazivom *Vektor* i *q* iz klase *Kvadrat* objekti-članovi neke klase, njihove deklaracije su

```
Vektor v;
Kvadrat q;
```

Na novine se nailazi tek kod deklarisanja-definisanja *funkcija-članica*, što ne treba da čudi jer se klasa u sintaksnom smislu upravo po tome razlikuje od svih ostalih tipova podataka. Pre svega, u C++ obična funkcija-članica realizuje se standardno, tako što dobija osobine gotovo identične osobinama slobodnog potprograma (funkcije). Međutim, postoji i druga, posebna vrsta funkcija-članica koje se zovu ***inline*** (čita se "inlajn"[33]) funkcije. One se karakterišu time što pri prevođenju, na mesto njihovog poziva prevodilac umesto skoka na početak smešta *kompletan kod funkcije*. Time se, za račun utroška memorije, dobija na brzini rada jer nema ni skoka ni prenosa argumenata preko steka. *Inline* funkcije su po pravilu fizički kratke, tj. ne zauzimaju mnogo memorijskog prostora i u njima se ne mogu koristiti selekcije niti ciklusi.

Pođimo, prvo, od deklaracije *običnih funkcija-članica* (tj. onih koje nisu *inline*). U okviru naredbe *class* za ove funkcije-članice navodi se samo prototip, dok se kompletna definicija (realizacija) navodi posebno. Dakle, ako neka klasa sadrži funkciju-članicu sa nazivom "method" i dva parametra tipa int, njen prototip sadržan u naredbi *class* je *double method(int i, int j);* ili (češće)

```
double method(int, int);
```

[32] Ova notacija nosi (nezgrapan) naziv "camel notation" zbog vizuelne asocijacije: *myAge, inputRecord, setCrossPoint* itd.

[33] Pošto ne postoji opšteprihvaćen termin za inline funkcije zadržaćemo originalni, engleski naziv, jer je objektna terminologija i bez izmišljanja novih izraza dovoljno neusklađena.

Primer:

```
class ExampleClass  {
private:
.
.
.
public:
  double method(int, int);
.
.
.
};
```

Kompletna funkcija realizuje se fizički (ali ne i logički!) izvan naredbe *class* i to tako što se po strukturi uopšte ne razlikuje od slobodne funkcije osim u jednoj - ali bitnoj - pojedinosti. Razlika u definiciji funkcije-članice i slobodne funkcije proističe iz činjenice da funkcija-članica nije samostalna nego pripada klasi i to mora na neki način biti notirano. Pripadnost klasi navodi se u delu realizacije funkcije (tj. delu izvan naredbe *class*) tzv. *kvalifikovanjem* koje se ogleda u dodavanju konstrukta

NazivKlase::

neposredno ispred imena metode, pri čemu je NazivKlase identifikator klase matične za datu funkciju-članicu. Prema tome, ako navedena metoda *method* pripada klasi ExampleClass i njen prototip stoji u odgovarajućoj naredbi *class*, tada realizacija ima formu

```
double ExampleClass::method(int i, int j) {
      naredbe funkcije
}
```

Funkcije-članice koje su *inline* mogu se realizovati na dva načina: prvi (i češći, s obzirom na to da su kratke) je upisivanje *kompletne* definicije funkcije, sa zaglavljem i blokom, u naredbu *class*. U tom slučaju funkcija se automatski tretira kao *inline* funkcija. Drugi način je da se *inline* funkcija realizuje istovetno kao i obična, dakle putem prototipa, s tim što u zaglavlju teksta funkcije tipu prethodi službena reč "inline", dok se u naredbi *class* nalazi prototip bez ikakvih dodataka. Znači, ako želimo da funkcija *method* bude tipa *inline* tada ćemo je ili u celosti

uključiti u naredbu *class*, ili će se u toj naredbi naći samo prototip praćen definicijom

```
inline double ExampleClass::method(int i, int j) {
      naredbe funkcije
}
```

Bez obzira na vrstu, funkcije-članice predstavljaju integralni deo implementacije klase, te još jednom podsećamo

- da su funkcijama-članicama u svakom trenutku dostupni svi ostali članovi klase.

Obe vrste funkcija-članica (obične i *inline*) mogu se deklarisati kao tzv. ***konstantne funkcije-članice***. Njihova karakteristika je da ne mogu da menjaju stanje objekta (npr. ne mogu da zadaju ili promene vrednost podatka-člana) i da već u fazi prevođenja takav pokušaj rezultuje porukom o grešci. Svrha uvođenja konstantnih metoda dvostruka je:

1. Principijelni razlog: ako je neka funkcija po semantici takva da ne menja stanje objekta, to treba eksplicitno zabraniti. Naravno, ovo nije obavezno, jer programer prosto može da povede računa o tome da ne napravi takvu grešku.
2. Objekti (instance klase) mogu se definisati kao tzv. *konstantni objekti*, analogno zaključanim promenljivim. Ako je objekat definisan kao konstantan, njemu se mogu slati samo one poruke koje aktiviraju konstantne metode, što je uostalom i osnovna svrha postojanja konstantnih metoda.

Inače, dobra programerska praksa nalaže da se konstantne funkcije-članice eksplicitno deklarišu kao takve. Sintaksno, one se deklarišu tako što se u odgovarajućoj naredbi *class* iza liste formalnih parametara piše službena reč

const

bez obzira na to da li se radi o prototipu ili *inline* funkciji navedenoj u celosti. Ako pomenutu funkciju *method* želimo da deklarišemo kao konstantnu, tada se u odgovarajućoj naredbi *class* kao i u realizaciji funkcije piše

```
class ExampleClass {
.
.
.
public:
  double method(int, int) const;
 .
```

```
 .
 .
};
double ExampleClass::method(int i, int j) const {
      naredbe funkcije
}
```

Razmotrimo, u vidu primera, definiciju vrlo jednostavne klase *Point* koja predstavlja tačku u Dekartovom koordinatnom sistemu. Članovi klase su:

- Podaci-članovi *x* i *y* tipa *double* koji odgovaraju koordinatama i koji su, saobrazno principu skrivanja informacija, zatvoreni. Zapazimo da *x* i *y* imaju osobine atributa jer odgovaraju bitnim osobinama tačke.
- Funkcije-članice: *setPoint* za zadavanje koordinata, *getX* i *getY* za njihovo očitavanje i *distance* za određivanje rastojanja od koordinatnog početka. Neka prve tri budu *inline*, a četvrta neka bude obična.

Naredba *class* za zadavanje klase ima oblik

```
class Point {
private:
  double x, y;
public:
  void setPoint(double xx, double yy) {x= xx; y= yy;}
  double getX() const {return x;}
  double getY() const {return y;}
  double distance() const;
};
double Point::distance() const {
  return sqrt(x*x + y*y);
}
```

Uočimo da samo funkcija *setPoint* menja stanje, dok ostale to ne čine pa možemo da ih definišemo kao konstantne. Funkcija *setPoint* (zasad) služi i za inicijalizaciju što kod C^{++} nije praksa jer za tu svrhu postoje posebni mehanizmi nazvani konstruktorima o kojima će biti reči kasnije.

Instanciranje klase u C^{++} može da bude jednostavna, ali i relativno složena operacija. U najjednostavnijem obliku, sa kojim ćemo započeti, instanciranje klase sintaksno se poklapa sa definicijom procedurne promenljive. Dakle, opšti oblik najjednostavnije naredbe za instanciranje je

NazivKlase $objekat_1$, $objekat_2$, ..., $objekat_n$;

Ako, na primer, treba definisati celobrojne promenljive *k* i *m*, promenljivu *h* tipa *Height* definisanog u programu i objekte *p1* i *p2* klase *Point* odgovarajući segment programa ima sledeći izgled:

```
int k, m;
Height h;
Point p1, p2;
```

gde je očigledno da nema razlike, tj. da na osnovu samih definicija ne možemo odrediti da li se radi o promenljivim ili objektima. Iako se u ovoj formi to ne vidi, odmah treba napomenuti da je *instanciranje klase izvršna naredba* koja predstavlja aktiviranje posebne metode klase sa nazivom konstruktor. Ovo podvlačimo stoga što se prve dve definicije iz primera mogu, a ne moraju shvatiti kao aktivne (moderni pogled na programiranje i to preporučuje). Prema tome, *Point p1, p2;* treba tretirati kao aktiviranje metode za konstrukciju objekta, a ne kao njegovu pasivnu definiciju.

Pristup *podacima-članovima* (naravno, pod uslovom da su dostupni) sintaksno se izvodi isto kao pristup poljima sloga (strukture), tj. operacijom

```
nazivObjekta.nazivPodatka
```

Funkcije-članice aktiviraju se slično:

```
nazivObjekta.nazivFunkcije(argumenti)
```

što je ilustrovano sledećim primerima:

```
double r, t, dist;
Point p1, p2, p3;     // Konstruisanje objekata (instanciranje klase)
p1.setPoint(1.5, -2.8)     // Inicijalizacija p1
p2.setPoint(4, 2);
r= p1.distance();     // Aktiviranje funkcije distance (slanje poruke objektu p1)
p3= p2;     // Nad objektima je definisana operacija dodele!
t= p3.x;     // Sintaksno pravilno, ali je nedozvoljeno jer je x zatvoren
dist= sqrt(pow((p1.getX() - p2.getX())), 2) +
          pow((p1.getY() - p2.getY())), 2));     // Racunanje rastojanja p1 od p2
```

Zapazimo da sintaksno ispravna naredba t= p3.x nije dozvoljena samo zato što je x zatvoren podatak-član, tj. nalazi se u segmentu *private*. Ispravan oblik bio bi

t= p3.getX();

Dalje, vidimo da je nad objektima definisana operacija dodele (p3= p2). Ona se izvodi tako što se vrednosti svih podataka-članova u objektu koji se nalazi sa desne strane kopiraju u odgovarajuće podatke-članove objekta sa leve strane. Pritom, izraz "svi podaci-članovi" podrazumeva i podatke-članove objekta, ali i svih njihovih podobjekata. Napomenimo i to da objekti mogu biti upotrebljeni kao parametri odn. argumenti kako slobodnih funkcija, tako i metoda matične i drugih klasa.

Objektima-članovima pristupa se konstruktom

nazivObjekta.nazivObjektaČlana.α

gde α označava član podobjekta kojem se pristupa. Inače, α može da označava i pristup objektu na sledećem, nižem, nivou, njegovom podobjektu itd. Kao ilustraciju, formiraćemo klasu *Line* koja modeluje duž čije su krajnje tačke takođe objekti i to klase *Point*. Da bismo mogli da prikažemo pristup objektu-članu, tačke ćemo ostaviti otvorenim. Definicija klase *Line* izgleda ovako:

```
class Line {
public:
  Point a, b;     // Krajnje tacke koje su takodje objekti
  void setLine(double, double, double, double);     // Inicijalizacija duzi
  double length() const;     // Racunanje duzine
};
void Line::setLine(double x1, double y1, double x2, double y2) {
  a.setPoint(x1, y1);
  b.setPoint(x2, y2);
}
double Line::length() const
{
  return sqrt(pow(a.getX()-b.getX()), 2) +
              pow(a.getY()-b.getY()), 2));
}
```

U sledećim primerima prikazan je način pristupa objektima-članovima:

```
double x, y;
Line myLine;
myLine.setLine(1.5, 2.0, 5, 4.6);
x= myLine.a.getX();     // Pristup apscisi tacke a duzi
```

```
y= myLine.a.getY();     // Pristup ordinati tacke a duzi
```

3.2.2. Statički članovi klase

Prilikom razmatranja opštih osobina klase pomenuli smo da ona, pored podataka, objekata i funkcija nivoa instance, može da ima i članove koji su zajednički za sve. To su članovi koji su definisani na nivou klase, a ne pojedinačnih objekata. U programskom jeziku C^{++} (i u javi) poznati su pod nazivom *statički članovi klase*. Sve tri vrste članova klase mogu biti statičke: i podaci-članovi i objekti-članovi i funkcije-članice. Pristup statičkim članovima klase kontroliše se standardnim mehanizmima C^{++} (private, public itd.).

Statički članovi klase prepoznaju se po tome što u deklaraciji-definiciji imaju modifikator

static

koji se navodi ispred tipa. Posmatrajmo jedan primer klase sa statičkim članovima:

```
class ExampleClass  {
private:
  static int privStaticDat;
  int privDat;
.
.
.
public:
  static int publStaticDat;
  static OtherClass staticObj;
  static int getPrivStaticDat() { return privStaticDat; }
  static int getPublStaticDat() { return publStaticDat; }
  int nonStaticGetDat()  { return PrivStaticDat; }
.
.
.
};
int ExampleClass::privStaticDat = 1111;
int ExampleClass::publStaticDat = 2222;
OtherClass ExampleClass::staticObj;
```

Klasa ExampleClass sadrži sledeće statičke članove:

- Zatvoreni podatak-član privStaticDat
- Otvoreni podatak-član publStaticDat

- Otvoreni objekat-član staticObj iz klase OtherClass
- Statička metoda (funkcija-članica) getPrivStaticDat za očitavanje zatvorenog podatka-člana privStaticDat
- Statička metoda getPublStaticDat za očitavanje otvorenog podatka-člana publStaticDat.

Klasa ima i dva člana koji nisu statički, tj. koji su definisani na nivou instance klase:

- Zatvoreni (nestatički) podatak-član privDat
- Nestatička metoda nonStaticGetDat za očitavanje zatvorenog statičkog podatka-člana privStaticDat.

U primeru su sve statičke metode tipa *inline* što, naravno, nije obavezno. Statički podaci-članovi i objekti-članovi su naredbom *class* samo deklarisani, ali ne i definisani, zato što su delovi same klase ExampleClass, a ne pojedinačnih instanci. Stoga se moraju definisati (za objekte-članove kažemo *konstruisati*) posebnim naredbama sa inicijalizacijom ili bez nje. Pri definisanju kvalifikuju se konstruktom

NazivKlase::

Poslednje tri naredbe u definiciji klase ExampleClass služe upravo za tu svrhu. Pritom, podaci-članovi privStaticDat i publStaticDat su eksplicitno inicijalizovani vrednostima redom 1111 i 2222. Objekat-član staticObj konstruisan je poslednjom naredbom u navedenom segmentu i to na najjednostavniji način. Videćemo kasnije da za konstrukciju objekata postoje i drugi, složeniji i delotvorniji, postupci. Statičkim članovima klase pristupa se na dva načina:

- direktno i
- indirektno, preko instance klase.

Direktni pristup statičkim članovima moguć je zahvaljujući činjenici da su oni, i fizički, deo klase. Drugim rečima, njihov memorijski prostor deo je memorijskog prostora klase, a ne njenih instanci. Otvorene statičke metode aktiviraju se porukom koja ima oblik

NazivKlase::statičkaMetoda

Za naš primer to bi izgledalo ovako:

ExampleClass::getPublStaticDat()

Direktni pristup statičkim podacima i objektima članovima ostvaruje se analognim konstruktom. Na primer,

ExampleClass::publStaticDat = 1234;

```
ExampleClass::staticObj.methodFromOtherClass(a, b);
```

gde je methodFromOtherClass neka metoda iz klase OtherClass kojoj pripada statički objekat-član staticObj. Otvorenim statičkim podacima i objektima članovima može se direktno pristupiti i preko posebnih statičkih metoda. Primer je

```
x= ExampleClass::getPublStaticDat();
```

gde se promenljivoj x dodeljuje vrednost podatka-člana publStaticDat preko metode getPublStaticDat. Otvorenim podacima i objektima članovima pristupa se i preko slobodnih funkcija. Evo jedne slobodne funkcije za očitavanje publStaticDat:

```
int readPublStaticDat() {
  return ExampleClass::publStaticDat;
}
```

Direktni pristup zatvorenim podacima i objektima članovima moguće je realizovati samo putem statičkih metoda. Tako, podatak-član privStaticDat očitava se samo primenom statičke metode getPrivStaticDat. Primer:

```
y= ExampleClass::getPrivStaticDat() + 2;
```

Indirektni pristup članovima klase obavlja se iz instanci. Dakle, da bi se omogućio indirektan pristup mora biti kreiran objekat kroz koji se on vrši. Pritom, ako postoji više instanci klase, sasvim je svejedno koja će biti upotrebljena, a mogu se upotrebiti i više njih. Ako je kreiran objekat

```
ExampleClass myObject;
```

pristup članovima klase, kako otvorenim tako i zatvorenim, izvodi se putem objekta myObject. Evo par primera:

```
x= myObject.getPrivStaticDat();    // Pristup zatvorenom clanu
y= myObject.getPublStaticDat() + x - 1;    // Pristup otvorenom clanu
```

Za indirektan pristup može se primeniti i nestatička metoda jer je ona takođe član klase kao i svi drugi. Takva metoda je metoda nonStaticGetDat iz našeg primera. Ako želimo promenljivoj z pridružiti vrednost zatvorenog statičkog podatka člana privStaticDat preko metode nonStaticGetDat pisaćemo

```
z= myObject. nonStaticGetDat();    // Pristup preko nestaticke metode
```

pri čemu je metoda nonStaticGetDat nivoa objekta, a ne klase.

Prilikom realizovanja statičkih metoda treba voditi računa o tome da su one *zajedničke* za sve objekte date klase te, prema tome, ne mogu pristupati članovima nivoa objekta. U ExampleClass smo uključili podatak-član privDat koji nije statički, što znači da ga ima svaka instanca i da, u opštem slučaju, uzima različite vrednosti za različite instance. Upravo zato u klasu *ne bismo* mogli da uključimo statičku metodu za, recimo, očitavanje privDat:

```
static int statGetDat() { return privDat; }
```

Za ovaj podatak-član, metoda kojom se očitava vrednost mora biti nestatička uz aktiviranje isključivo preko instance klase.

Statički članovi klase pojavljuju se ređe od nestatičkih, što ne znači da ih treba izbegavati: jednostavno, priroda problema i realizacije je takva da su ovi članovi u manjini. Izuzetak su dve specijalne vrste metoda koje su nivoa klase, a koje poseduje, eksplicitno ili implicitno svaka klasa. Radi se o važnim metodama koje nose naziv konstruktori odn. destruktori i koje će biti obrađene u posebnim odeljcima.

4. INKAPSULACIJA I MODULARNOST

Zajedničko za inkapsulaciju i modularnost jeste to da se radi o sredstvima za organizovanje klasa. Inkapsulacijom se vrši neka vrsta unutrašnjeg organizovanja klase objedinjavanjem sadržaja (podaci-članovi, objekti-članovi i funkcije-članice) u softversku celinu, dok moduli služe za spoljašnje organizovanje klasa u celine višeg reda.

4.1. INKAPSULACIJA

Prilikom razmatranja objekta i klase ustanovili smo da stanje i ponašanje objekata date klase čine nedeljivu celinu, poput prostora-vremena u fizici ili hardvera-softvera u računarstvu. Reakcija na poruku - a poruka je vezana za ponašanje - u opštem slučaju podrazumeva promenu stanja; promena stanja, sa svoje strane, utiče na buduće ponašanje. Neka je *Stack* klasa koja predstavlja stek, sa osnovnim operacijama

- top (očitavanje vrha steka)
- pop (uklanjanje elementa sa vrha steka)
- push (upis elementa u stek)
- empty (provera da li je stek prazan).

Ako se instanci klase *Stack* pošalje poruka za upis elementa *push(a)* doćiće do promene stanja jer se na vrhu steka pojavljuje novi element *a*; promena stanja ima povratni efekat na ponašanje steka jer će operacije *top* i *pop* očitati, odn. ukloniti novododati element. Ako stek ima tačno jedan element, operacija *empty* daje vrednost 0 (false), ali po izvršenju operacije *pop* ta vrednost postaje 1 (true).

Jedinstvo stanja i ponašanja je, prema tome, *conditio sine qua non* za egzistenciju objekta. Ranije smo, takođe, ustanovili da je prostor stanja objekata date klase određen strukturom. To, konačno, znači da svaki objektno orijentisan jezik mora posedovati tehnička sredstva za *objedinjavanje strukture i ponašanja* u celinu. Mehanizam za ostvarivanje ovog cilja nosi naziv ***inkapsulacija*** (od latinske reči capsula = čaura, kutijica).

Objedinjavanje strukture i ponašanja nije jedini zadatak inkapsulacije. U prethodnom poglavlju detaljno smo diskutovali skrivanje informacija - princip koji,

doduše, nije teorijski apsolutno nezaobilazan, ali čije nepoštovanje vodi krajnje nekvalitetnom softveru. Drugim rečima, slobodno se može zaključiti da je kontrola pristupa članovima klase, izvedena iz principa skrivanja informacija, zahtev koji mora biti ispunjen već u definiciji klase. U navedenom primeru steka, princip skrivanja informacija podrazumevao bi takvu implementaciju da fizička struktura steka bude zaštićena od direktnog pristupa, tj. da se komunikacija klijenta sa stekom u celosti obavlja preko interfejsa sastavljenog od navedenih operacija. Promena fizičke strukture steka ne bi smela da se odrazi na formu interfejsa. *Obezbeđivanje kontrole pristupa* u cilju poštovanja principa skrivanja informacija drugi je osnovni zadatak inkapsulacije.

Imajući u vidu pomenute dve svrhe - objedinjavanje strukture i ponašanja, odnosno ostvarivanje kontrole pristupa - inkapsulaciju ćemo ***definisati*** kao

- sredstvo za objedinjavanje strukture i ponašanja (funkcionalnosti) u softversku celinu tako da se obezbedi poštovanje principa skrivanja informacija putem kontrole pristupa.

Drugim rečima, inkapsulirati klasu znači povezati u celinu interfejs i telo klase uz kontrolu pristupa za klijente raznih kategorija.

Striktno poštovanje principa skrivanja informacija u izvornom obliku podrazumeva da su nedostupni svi podaci-članovi i objekti-članovi, tako da se objektom rukuje samo putem metoda. Ova vrsta inkapsulacije nosi naziv *jaka inkapsulacija* (engl. "strong encapsulation"). U modernom objektnom programiranju jaka inkapsulacija je poželjna, ali nije obavezna.

U odnosu na mesto u tripletu domen problema - model - implementacija, inkapsulacija se delimično pojavljuje na nivou modela, jer se već kod modelovanja naznačuju članovi klase što, bar grubo, definišu strukturu i ponašanje objekata (npr. u dijagramima klasa UML daju se atributi i metode). Međutim, pravo mesto inkapsulacije jeste *nivo implementacije* i to stoga što sredstva za inkapsulaciju u određenoj meri zavise i od konkretnog programskog jezika.

U C^{++} osnovni zadatak inkapsulacije - objedinjavanje strukture i ponašanja - ostvaruje se u okviru deklaracije klase, odnosno naredbom *class*. Fizički izdvojeni članovi klase logički se povezuju sa ostalim putem kvalifikatora *NazivKlase::*, tako da zajedno sa naredbom *class* sačinjavaju potpunu deklaraciju klase. Prema tome, kada je u pitanju objedinjavanje strukture i ponašanja, sa stanovišta jezičkih sredstava, nema se šta novo dodati onom što je rečeno u prethodnom poglavlju. Još jednom,

- Objedinjavanje strukture i ponašanja objekata u okviru klase ostvaruje se preko celovitog skupa naredbi za njeno deklarisanje.

Nešto je drukčija situacija sa kontrolom pristupa koja je drugi osnovni zadatak inkapsulacije. Ovde programer ima mogućnost izbora koje članove klase će zat-

voriti, a koje ostaviti otvorenim. Jedno od jezičkih sredstava već smo naveli, a to je podela deklaracije klase na otvoreni deo sa labelom *public* i zatvoreni deo sa labelom *private*. Pored toga C^{++} ima još neke načine kontrole od kojih ćemo deo prikazati u nastavku, a deo ostaviti za kasnije.

Posmatrajmo klase *Point* i *Line* iz odeljka 3.2.1. Klasa *Point* je jako inkapsulirana jer se podaci-članovi *x* i *y* nalaze u segmentu labeliranom sa *private*, tako da se pristup ostvaruje isključivo metodama *setPoint*, *getX* i *getY*. Klasa *Line* nije jako inkapsulirana zato što su objekti-članovi *A* i *B* otvoreni stavljanjem u segment *public* što, odmah da podvučemo, nije greška nego prosto jedna od mogućnosti.

4.1.1. Inkapsulacija u C^{++}

Primera radi, učinićemo i klasu *Line* jako inkapsuliranom. Za tako nešto nisu potrebni nikakvi sintaksni dodaci: jednostavno, objekte-članove *A* i *B* treba zatvoriti (premestiti u segment *private*) i dodati posebne metode *getAx()*, *getAy()*, *getBx()* i *getBy()* za očitavanje koordinata temena Dakle, jako inkapsulirana klasa *Line* izgledala bi ovako:

```
class Line  {
private:
  Point A, B;
public:
  void setLine(double, double, double, double);
  double getAx() const {return A.getX();}
  double getAy() const {return A.getY();}
  double getBx() const {return B.getX();}
  double getBy() const {return B.getY();}
  double length() const;
};
void Line::setLine(double x1, double y1, double x2, double y2) {
  A.setPoint(x1, y1);
  B.setPoint(x2, y2);
}
double Line::length() const {
  return sqrt(pow((A.getX()-B.getX()),2)+
           pow((A.getY()-B.getY()),2));
}
```

Evo nekih primera primene jako inkapsulirane klase *Line*:

```
Line myLine; double x,s;
. . . . . .
myLine.setLine(0,1.5,3,3.8);     // Inicijalizacija
```

```
s= pow(myLine.length(),2);
x= myLine.A.getX();     // Nije dozvoljeno jer je A zatvoren objekat-clan
x= myLine.getAx();     // Korektno
```

Već na ovom, jednostavnom, primeru može se uočiti da jaka inkapsulacija ima za posledicu povećanje broja metoda, jer se sve potrebne metode iz zatvorenih polja moraju ponovo definisati u matičnoj klasi (u našem primeru to su metode *getAx(), getAy(), getBx()* i *getBy()*). O ovoj problematici videti detaljnije u odeljku koji se odnosi na Demetrin zakon, poglavlje 7.

4.1.2. Prijateljske (kooperativne) funkcije i klase

Prijateljske ili kooperativne funkcije i klase predstavljaju još jedan od načina za kontrolu pristupa klasi u C^{++}.

Prijateljske funkcije (kooperativne funkcije, funkcije-prijatelji) klase su funkcije kojima data klasa eksplicitno dozvoljava da pristupe svim njenim članovima, otvorenim i zatvorenim, tj. za koje kontrola pristupa ne važi. Mogu biti - i najčešće jesu - slobodne, ali isto tako mogu biti funkcije-članice druge klase.

Slično, ***prijateljske klase*** za datu klasu su one kojima ta klasa dozvoljava da pristupe svim njenim članovima.

Tehnički, svrha otvaranja klase za neke funkcije i klase je *brzina*, jer umesto da zatvorenim podacima-članovima i objektima-članovima pristupe preko odgovarajućih metoda, prijateljske funkcije i klase to čine direktno, kao da su i one članovi klase. Inače - to naročito treba podvući - prijateljske funkcije i klase *nisu članice klase* za koju su deklarisane kao prijateljske. Prijateljske funkcije se najčešće susreću kod tzv. preklapanja operatora (videti dalje).

Veoma je važno napomenuti da se funkcije odn. klase promovišu u prijateljske od strane osnovne klase, u okviru njene definicije. Drugim rečima, funkcija ili klasa ne mogu same sebe "proglasiti" prijateljskim za neke klase jer bi to predstavljalo obesmišljavanje svih pravila inkapsulacije. Opravdano bi se postavilo pitanje: "Čemu uopšte kontrola pristupa ako se ona može po volji narušiti sredstvima nezavisnim od klase?"

Kvalifikator za deklarisanje funkcije kao prijateljske je rezervisana reč

```
friend
```

koja se navodi u sklopu naredbe *class* zajedno sa prototipom funkcije. Mogu biti i *inline*. Ako je *inline* prijateljska funkcija u celosti uklopljena u naredbu *class*, kvalifikator *friend* navodi se u sklopu njenog zaglavlja.

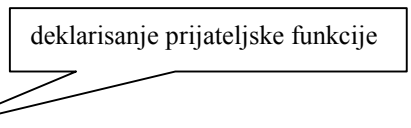
Recimo da želimo realizovati slobodnu funkciju *distanceBetween* koja računa rastojanje između dve tačke iz klase *Point*. Slobodna funkcija koja nije pri-

jateljska izgledala bi ovako:

```
double distanceBetween(Point p1, Point p2) {
  return sqrt( pow((p1.getX()-p2.getX()),2) + pow((p1.getY()-p2.getY()),2) );
}
```

Kako vidimo, za računanje rastojanja potrebno je posredovanje metoda *getX* i *getY* za očitavanje (zatvorenih) apscisa i ordinata tačaka, što usporava izvršavanje. Da bi se ubrzalo izvršavanje funkciju *distanceBetween* možemo realizovati kao prijateljsku funkciju klase *Point*. U tu svrhu definicija klase *Point* mora se preraditi tako da funkciji bude omogućen pristup podacima-članovima x i y:

```
class Point {
private:
  double x, y;
public:
  void setPoint ...
  double getX ...
  double getY ...
  double Distance ...
  friend double distanceBetween(Point, Point);
};

double distanceBetween(Point p1, Point p2) {
return sqrt(pow((p1.x-p2.x),2)+pow((p1.y-p2.y),2));
}
```



Uočimo da slobodna(!) funkcija *distanceBetween* direktno pristupa podacima-članovima x i y samo zato što joj je klasa *Point* to eksplicitno dozvolila.

Klasa se može, delimično ili potpuno, otvoriti i za drugu klasu. Klasa *P* se delimično otvara za klasu *Q* tako što se nekim metodama iz *Q* dozvoljava pristup svim članovima klase *P*. To se ostvaruje uključivanjem prototipova tih metoda u definiciju klase *P*, uz kvalifikatore *friend* i *Q::* gde drugi pokazuje da je prijateljska funkcija metoda klase *Q*, a ne slobodna funkcija. Klasa *P* se otvara za celu klasu *R* tako što se u definiciju klase *P* uvrštava konstrukt *friend class R*. U naredbi

```
class P {
. . . . . .
public:
  friend void Q::m();
  friend class R;
```

```
};
```

klasa P je otvorena za metodu m iz klase Q i za sve metode iz klase R.

Postoji prilična razlika između prijateljskih slobodnih funkcija s jedne i metoda s druge strane. Dok su metode i logički i fizički *deo* klase, prijateljske funkcije su *slobodne* funkcije koje su samo logički povezane sa klasom, čak i ako su izvedene *inline* u naredbi class. Štaviše, ista slobodna funkcija može biti kooperativna sa više klasa istovremeno ako ima formalne parametre iz različitih klasa i svaka od njih joj omogućava pristup svim delovima. Funkcija *distanceBetween* je logički čvrsto povezana sa klasom *Point* i verovatno ni sa jednom drugom. Stoga je za očekivanje da će se naći u istom modulu sa klasom *Point* ili da će, čak, biti izvedena *inline* u okviru naredbe class Point {...}:

```
class Point {
private:
  double x, y;
public:
  void setPoint ...
  double getX ...
  double getY ...
  double Distance ...
  friend double distanceBetween(Point p1, Point p2) {
    return sqrt(pow((p1.x-p2.x),2)+pow((p1.y-p2.y),2));
  }
};
```

Razlika između metoda i slobodnih prijateljskih funkcija date klase najbolje se uočava kod načina pozivanja. Metoda se aktivira porukom *nazivObjekta.nazivMetode(parametri)*, dok se slobodna kooperativna funkcija poziva kao i svaka druga funkcija. Da bismo ilustrovali razliku dopunićemo klasu *Point* metodom *distanceFrom* za određivanje rastojanja date tačke od druge tačke. Funkcija ima prototip

```
double distanceFrom(Point);
```

u naredbi class Point {...} i definiciju

```
double Point::distanceFrom(Point p) {
  return sqrt(pow((x-p.x),2)+pow((y-p.y),2));
}
```

Funkcije *distanceFrom* i *distanceBetween* pozivaju se na sledeći način:

```
Point a, b; double r, s;
. . . . . .
r= distanceBetween(a,b);
s= a.distanceFrom(b);
```

Očigledno, slobodna prijateljska funkcija *distanceBetween* ponaša se kao binarna operacija nad dva ravnopravna operanda, *a* i *b*, dok metoda *distanceFrom* ima prirodu unarne operacije nad objektom *a* sa parametrom *b*.

Delimično ili potpuno otvaranje klase za drugu klasu je pitanje koje se bitno razlikuje od korišćenja slobodnih prijateljskih funkcija. Radi se o tome da klase ne čine (ili bar ne u opštem slučaju) logički čvrsto povezanu celinu, te upotreba ove mogućnosti može da ugrozi princip skrivanja informacija i još neke bazične zakone inkapsulacije (na primer, pomenuti "Demetrin zakon" koji propisuje da se data klasa delimično ili potpuno otvara samo za klase koje su sa datom klasom u posebnoj vrsti veze nazvanoj "nasleđivanje", videti poglavlje 7). Dakle, klasu ćemo delimično ili potpuno otvoriti za drugu klasu samo u posebnim situacijama i uz precizno obrazloženje (kada je, recimo, brzina imperativ, a klase su logički povezane).

Bez obzira na to da li se neka klasa otvara za slobodne funkcije, metode ili druge klase, a u cilju očuvanja temeljnih principa vezanih za kontrolu pristupa, treba se pridržavati sledećeg pravila:

- Softverske komponente (slobodne funkcije i klase) za koje je data klasa potpuno ili delimično otvorena preko mehanizma *friend*, treba da se nalaze *u istom modulu* sa njom.

Ovo pravilo obezbeđuje da, ako dođe do modifikacije klase koja je otvorena za druge komponente i ako se, pritom, modifikuju i te komponente, sve izmene budu u okviru izvornog koda istog modula.

Recimo, klasa *Point* može se otvoriti za klasu *Line*, ali samo ako su u istom modulu. Kada to ne bi bio slučaj, izmena u klasi *Point* mogla bi da zahteva izmene u klasi *Line*, a da to programer previdi. Primera radi, navodimo kompletan kod modifikovanih klasa *Point* i *Line* zajedno sa slobodnom funkcijom *distanceBetween* koja je kooperativna sa klasom *Point*.

```
class Point {
private:
  double x, y;
public:
```

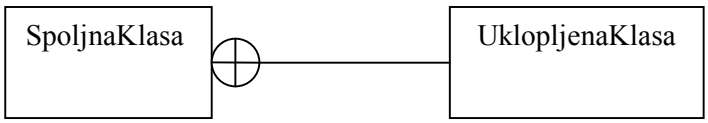
```
  void setPoint(double xx, double yy) {x= xx; y= yy;}
  double getX() const {return x;}
  double getY() const {return y;}
  double distance() const;
  friend class Line;     // Prijateljska klasa Line
  friend double distanceBetween(Point, Point);    // Prijateljska funkcija
};
double Point::distance() const {
  return sqrt(x*x + y*y);
}
double distanceBetween(Point p1, Point p2) {
  return sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2));  // Pristup podacima-clanovima
}

class Line  {
private:
  Point A, B;
public:
  void setLine(double, double, double, double);
  double getAx() const {return A.x;}  // Line ima pristup unutrasnjosti Point
  double getAy() const {return A.y;}
  double getBx() const {return B.x;}
  double getBy() const {return B.y;}
  double length() const;
};
void Line::setLine(double x1, double y1, double x2, double y2) {
  A.x= x1; A.y= y1;   // Line ima pristup unutrasnjosti Point
  B.x= x2; B.y= y2;
}
double Line::length() const {
  return sqrt(pow(A.x-B.x,2)+
              pow(A.y-B.y,2));    // Line ima pristup unutrasnjosti Point
}
```

4.1.3. Uklopljene klase i strukture

Programski jezik C^{++} dozvoljava da se u definiciji klase nađe definicija druge klase, što takođe predstavlja jednu vrstu inkapsulacije. Za klasu čija je definicija sadržana u definiciji druge klase kažemo da je uklopljena (ugneždena) u tu klasu. Sintaksno, to se izvodi jednostavno, uključivanjem naredbe *class* za uklopljenu klasu u naredbu *class* spoljašnje klase:

```
class SpoljnaKlasa {
  class UkloplјenaKlasa {
  .................
  };
.................
};
```

Pritom, uklopljena klasa može se uvrstiti u otvoreni ili zaštićeni deo spoljašnje klase. Zaštićeni delovi obe klase zatvoreni su jedan za drugi, tako da se po potrebi, moraju učiniti tzv. prijateljskim (videti glavu 6). Na slici 4.1 dat je UML prikaz uklopljene klase.



Slika 4.1

Uklopljene klase koriste se, kao deo sadržaja spoljne klase, u slučajevima kada treba zatvoriti ne samo neke objekte-članove nego i klase kojima pripadaju, tako da i objekat-član i njegova klasa budu nedostupni spolja. Klijent pristupa uklopljenoj klasi preko konstrukta *SpoljnaKlasa::UkloplјenaKlasa*.

Uklopljene strukture imaju istu svrhu kao i uklopljene klase: ako je zatvoreni podatak-član tipa strukture (sloga) i ako je i taj tip interni za klasu, zatvaraju se kako podatak-član tako i njegov tip. Primer:

```
class A {
private:
  struct InternaStruktura {     // Deklarisanje uklopljenog tipa "InternaStruktura"
    double x;
    int j;
  };
  InternaStruktura *pEl;
.............
};
```

Klasa sadrži zatvoreni podatak-član **pEl* koji je tipa pokazivača na interno deklarisanu strukturu sa nazivom *InternaStruktura*. Pošto se, po pretpostavci, ova struktura ne koristi nigde izvan klase *A*, i podatak-član *pEl* i njegov tip zatvoreni su, tj. inkapsulirani, u klasu *A*.

Tipičnu primenu uklopljene klase i strukture nalaze u objektnoj realizaciji dinamičkih struktura podataka (npr. liste ili stabla) i to kada je realizacija spregnuta, tj. bazirana na pokazivačima. U takvim slučajevima stvarni sadržaj svakog elementa sastoji se od korisnog informacionog sadržaja proširenog određenim brojem pokazivača. Pošto pokazivači imaju isključivo internu upotrebu, a shodno principu skrivanja informacija, oni ne treba da budu "vidljivi" izvan klase. Stoga se stvarni tip elementa (zajedno sa pokazivačima), definiše unutar klase što realizuje strukturu podataka, kao uklopljena klasa ili struktura, u zavisnosti od konkretnog slučaja. Primer primene uklopljene strukture nalazi se u odeljku 9.3.

4.2. MODULARNOST

Osobinu modularnosti imaju sistemi sastavljeni od visoko autonomnih delova koji zajednički ostvaruju projektovanu funkciju. Takvi autonomni delovi nose naziv *moduli* (od latinskog modulus = mera, merilo).

Na primer, module stereo uređaja čine tjuner, zvučnici, kasetofon, CD uređaj itd. Nastavni plan u školi ili na fakultetu može se realizovati modularno tako što se srodni pedmeti grupišu u module i biraju kao celina. Softverski razvojni sistemi kao što su "Turbo Pascal", "Turbo Vision", "Delphi", razni builder-i ili familija "Visual", nude veliki broj modula sa komponentama (tipovi, potprogrami, konstante, promenljive, klase) koje imaju zajednički domen (npr. matematičke funkcije, komunikacija sa štampačem, korisnički intefejs, grafika, sistemske funkcije itd.).

Može se diskutovati o tome da li je neki sistem modularan *per se*, tj. inherentno, ili je modularnost *pogled* na sistem kakav jeste. Srećom, u ovom kontekstu nema potrebe upuštati se u ovakva razmatranja, jer imamo u vidu softver kao tehnički sistem koji nije zadat nego se projektuje i realizuje. Jedan od načina za projektovanje-realizovanje je upravo primena modularizacije. Shodno tom, modul ćemo locirati na prelaz sa modelovanja na realizaciju i, naravno, unutar same realizacije.

Opšta definicija modula u tehnici, prema [76], glasi:

- "[...] 4. U tehnici samostalni deo neke mašine ili uređaja koji zajedno sa drugim delovima čini kvalitativno viši sistem i vrši višu funkciju od one koja odgovara zbiru pojedinačnih delova. Upotreba modula omogućuje takođe višestruku namenu pojedinog uređaja zamenom delova."

Od softverskog modula očekuje se da ima dve esencijalne osobine: osobinu *autonomnosti* i mogućnost *višekratne upotrebe* (engl. "reusability"), gde ovo drugo znači da usluge modula mogu koristiti različiti klijenti i to bez ikakvih intervencija na samom modulu. ***Softverski modul*** definiše se na sledeći način:

- osobinu *modularnosti* ima softverska komponenta koja se **(a)** realizuje, dakle projektuje, kodira, testira i modifikuje autonomno i **(b)** koja sadrži

skup softverskih servisa. Svaka takva komponenta nosi naziv *modul*.

Ovde moramo podvući da ovo nije jedina definicija modula. Naime, neki autori iz definicije izostavljaju tačku *b*, što bi značilo da i glavni program ima karakteristike modula. Ranije se osobina modularnosti pripisivala čak i slobodnim potprogramima zato što predstavljaju logičku celinu, iako nemaju osobinu autonomnosti. Ako ipak prihvatimo gornju definiciju, lako uočavamo da se softverski modul tretira kao analogon kutiji sa običnim alatom. Osobinu modularnosti imaju biblioteke u C, jedinice ("unit") u paskalu, takođe klase *i* paketi u javi.

Prema Mejeru, [82] modularnost softvera u pojmovnom smislu obuhvata dve karakteristike:

1. mogućnost *višekratne upotrebe* softvera u različitim programima;
2. *proširivost*, gde pod proširivošću podrazumevamo mogućnost dopunjavanja softverske komponente bez ikakvih izmena (ili bar sa minimalnim izmenama) osnovnog sadržaja.

MODULARNOST = VIŠEKRATNA UPOTREBA + PROŠIRIVOST

Modul ima karakteristike crne kutije i raspolaže interfejsom namenjenim za interakciju sa klijentom, ali i zatvorenim delom u koji se, kako navode Parnas i dr. u [75], smeštaju "oni detalji koji su podložni promenama", te tako predstavljaju "tajnu sistema", čime se omogućuju izmene i dopune modula sa minimalnim uticajem na druge module.

Softverski moduli pojavili su se šezdesetih godina kao biblioteke gotovih potprograma, poglavito fortranskih, koje nisu imale strukturu nego su predstavljale proste skupove potprograma istog domena primene (npr. potprogrami za numeričke proračune). Tek kasnije, kao jedan od rezultata inauguracije strukturiranog programiranja, moduli su dobili i strukturu sastavljenu od dela za spregu (interfejsa) i dela sa realizacijom (tela modula). Takođe otvorena je mogućnost uključivanja u modul, pored potprograma, i drugih komponenata: tipova, konstanata i promenljivih. Na taj način, moduli su postali osnovni elementi za izgradnju kompleksnih programa, preuzimajući tu ulogu od potprograma. Pored definicionih, modul ima još neke bitne osobine (Parnas, [74,75]):

- Struktura modula mora biti jednostavna da se može razumeti u potpunosti
- Modul mora da se realizuje tako da promene niti zavise od drugih modula, niti utiču na njih. Na primer, prelaz sa dvokomponentnog niza na slog u modulu koji realizuje tačku u Dekartovom koordinatnom sistemu ne sme zavisiti od drugih modula niti zahtevati promene u njima. Ista situacija je kod modula što realizuje red (queue) koji se može realizovati sekvencijalno ili spregnuto, ali to ne sme uticati na klijente.
- Vrlo verovatne promene u realizaciji modula ne smeju biti detektabilne od strane klijenta; preciznije, promene koje su za očekivanje ne smeju se

odražavati na interfejs modula. Manje verovatne promene mogu uticati na interfejs, ali samo kod malih i ređe korišćenih modula. Promene koje su sasvim malo verovatne mogu menjati interfejs često korišćenih modula.

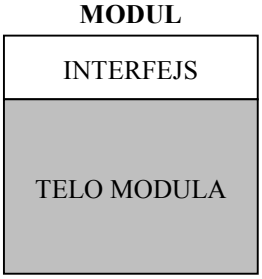
- Ozbiljnije promene u softverskom sistemu moraju se sastojati od skupa međusobno nezavisnih izmena u modulima.

Ove i slične osobine sintetizuju se u jednu:

- Modul mora imati jaku unutrašnju koheziju i slabu adheziju (respektivno jaku odn. slabu logičku povezanost sa drugim modulima).

Uzgred, ranije pomenuti princip skrivanja informacija koji *de facto* stoji iza svih pobrojanih zahteva, Parnas je formulisao baš u vezi sa modulima, a ne sa objektima, i to pre pojave objektnog programiranja.

U opštem slučaju, ***struktura modula*** nije naročito komplikovana. Modul se sastoji od dela za spregu sa klijentima, koji nosi naziv *interfejs* i zatvorenog dela sa neposrednom realizacijom koji ćemo nazvati *telo modula*, slika 4.2.

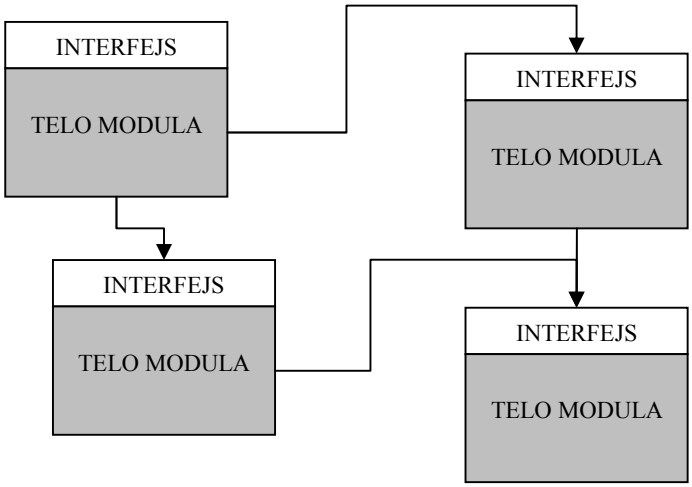


Slika 4.2

Interfejs modula, kao i svaki drugi interfejs, koristi se kao sprega modula sa klijentom i, u tom smislu, sadrži deklaracije-definicije svih dostupnih elemenata modula. U zavisnosti od programskog jezika, tu se mogu naći definicije tipova, klasa, zatim imenovanih konstanata, globalnih promenljivih i prototipova slobodnih funkcija iz modula. Telo modula sadrži realizaciju elemenata modula: funkcija i metoda iz interfejsa, kao i lokalnih funkcija, promenljivih i konstanata, pa i lokalnih tipova i klasa, ako je to potrebno. Interfejs i telo modula mogu činiti programsku celinu (tj. biti u istoj datoteci), ali ne moraju. Prvi slučaj je tipičan za paskal, a drugi za C/C^{++}. U svakom slučaju, programski sistem se realizuje kao sistem modula koji međusobno komuniciraju preko interfejsa, kao što je prikazano na slici 4.3.

Pomenimo i pitanje glavnog programa sa ulogom koordinatora u procesu obrade. Programi pisani na programskom jeziku C/C^{++} fizički se raspoređuju u nestrukturirane izvorne datoteke, tako da se "glavni program" sintaksno ne razlikuje od modula. Činjenica da se u izvornoj datoteci nalazi jedini primerak funkcije *main* sama po sebi nema uticaja na strukturu, jer je *main* funkcija kao i svaka druga. Izvorna datoteka sa funkcijom *main* razlikuje se od ostalih *po sadržaju*, stoga što se u

njoj nalazi ono što se kod nekih drugih modularnih programskih jezika nalazi u glavnom programu. Dakle, *namena* a ne sintaksa izvorne datoteke sa funkcijom *main* je to što diktira jednodelnu realizaciju. U daljem tekstu više se nećemo zadržavati na izvornoj datoteci sa ulogom glavnog programa - pažnju ćemo usmeriti na prave module (one koji zadovoljavaju usvojenu definiciju), dakle na one kod kojih je uočljiva granica interfejs - telo modula.



Slika 4.3

Interesantno je povući paralelu između klase i procedurnog modula (biblioteke u C odnosno jedinice u paskalu). Prvo, imaju istu globalnu strukturu sastavljenu iz interfejsa (otvorenog dela) i tela (zaštićenog dela) koje razdvaja lako uočljiva granica. Dalje, ulogu potprograma iz procedurnog modula, u klasi igra funkcija-članica klase. Procedurni modul može da ima sopstvene promenljive, kao što klasa ima podatke-članove. Klasa je u semantičkom pogledu visoko autonomna, a isto to važi i za procedurni modul. Konačno, u oba slučaja radi se o skupovima servisa namenjenim proizvoljnim klijentima. Očigledno, između modula kao takvog i klase postoji određeni odnos koji, u zavisnosti od programskog jezika, može imati tri pojavna oblika:

1. klasa *je* modul
2. klasa *pripada* modulu
3. i jedno i drugo

U prvom slučaju klasa *per se* ima osobine modula, dakle autonomna je i predstavlja skup servisa ovaploćenih u metodama. Uočimo da se ovde ne radi samo o pogledu na klasu nego, pre svega, o njenoj realizaciji na datom programskom jeziku čiji je zadatak da obezbedi modularnost klase. Ovaj zahtev formulisao je Mejer još osamdesetih godina prošlog veka, [1] izrazivši ga formulom

KLASA = MODUL

koja je po njemu nazvana *Mejerova jednakost*. Treba je shvatiti isključivo kvalitativno sa smislom "klasa je modul". Kasnije, sam Mejer je pooštrio interpretaciju tako što je dodao uslov da, osim klase, drugih vrsta modula u datom objektnom jeziku nema. Lako se uočava da Mejerova jednakost direktno proizlazi iz prvog od pet principa Alena Keja nabrojanih u odeljku 2.4: "sve je objekat". Iako je u pojmovnom smislu sasvim korektna, Mejerova jednakost, naročito u pooštrenom obliku, ima nedostataka u pogledu fizičke realizacije. Naime, veliki objektni sistemi zasnovani su na mnoštvu klasa. Ako je svaka klasa realizovana kao zaseban modul, dolazi do pojave razmrvljenosti (preterane granularnosti) objektnog sistema čija je direktna posledica slaba upravljivost. Čak i kada su logički srodne (na primer, klase *Trougao* i *Kvadrat*) klase se realizuju i memorišu potpuno nezavisno, pretvarajući tako objektni sistem u veliku, neuređenu kolekciju klasa koju je teško koristiti i još teže održavati. To podseća na početke procedurnog programiranja, kada se program razlagao direktno na potprograme, bez mogućnosti organizovanja potonjih u uređen skup. Tendencija razmrvljavanja tako koncipiranog softvera izazvala je potrebu za uvođenjem procedurnih modula kao sredstva za smanjenje granulacije i povećanje upravljivosti.

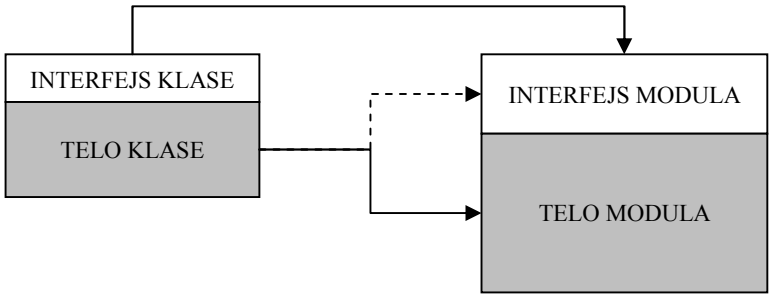
Problemi sa preteranom granularnošću, proistekli iz Mejerove jednakosti, rešavaju se tako što se programski jezik snabdeva posebnim sredstvima za realizaciju modula, nezavisnim od klase. Radi se o jedinicama u paskalu ili o bibliotekama C-a koje su u nepromenjenom obliku preuzete u objektne dijalekte. Umesto da bude modul, klasa se smešta u modul, što ćemo prikazati (kvalitativnom) formulom

KLASA $\in$ MODUL

U ovoj varijanti klasa više nije modul (nema osobinu autonomnosti), jer se u istom modulu mogu naći i druge klase, pa i slobodni potprogrami, imenovane konstante, konstantni objekti itd. Naravno, ukoliko baš želimo da očuvamo Mejerovu jednakost (ili bar da postignemo gotovo isti efekat), nema nikakve prepreke da svaku klasu smestimo u poseban, sopstveni modul koji nema drugog sadržaja. Naravno, u isti modul uvrštavaju se srodne klase da ne bi došlo do narušavanja njegove kohezije. Na primer, klase *Point* i *Line* mogu i treba da se nađu u istom modulu jer su logički čvrsto povezane. S druge strane, bilo bi pogrešno smestiti u isti modul npr. klasu *Semafor* i klasu *Matrica* jer odgovaraju stvarima između kojih je verovatno jedina sličnost to što su - stvari. Način smeštanja klase u modul uslovljen je njihovom strukturom, ali i sintaksom jezika-domaćina. Najkraće,

- interfejs klase uključuje se u interfejs modula, a telo klase u telo modula (slika 4.4).

Zapažamo da se interfejs klase obavezno u celosti nalazi u interfejsu modula, jer bi u suprotnom neki njegovi delovi bili nedostupni zbog zaštite unutar modula, a to se kosi sa svakom definicijom interfejsa klase. Nešto je drukčija situacija kod tela klase. U jezicima kao što su C++ i paskal, naredba za deklarisanje klase nalazi se u interfejsu modula. S druge strane, tom naredbom zadaju se i zatvoreni podaci-članovi i objekti-članovi klase, a kod C++ u naredbi za deklaraciju može se pojaviti i izvršni kod (*inline* metode). Usled toga, telo klase može se delom naći i u interfejsu modula a da, pritom, i dalje bude nedostupno klijentima, što se postiže dodatnim sredstvima poput kontrole pristupa unutar klase (*public*, *private* itd.).



Slika 4.4

Treća mogućnost jeste kombinacija prethodne dve: klasa je modul i klasa pripada modulu. Ovo je najbolje rešenje jer obezbeđuje i autonomnost klase i višekratnu upotrebu, a da se pritom izbegne granularizacija softvera. Primenjena je npr. u programskom jeziku java, a postiže se primenom dva nivoa modularizacije. Na nižem nivou, klasa se realizuje kao modul, a na višem se takvi moduli dalje objedinjavaju u tzv. pakete koji opet imaju osobine modula. S obzirom na to da C++ nema tu mogućnost, na ovom se nećemo duže zadržavati.

Dodajmo, na kraju, još nešto: bez obzira na to o kojem se od tri odnosa klasa-modul radi, obaveza izmeštanja klase iz glavnog programa ili njegovog ekvivalenta je na snazi i u vezi je kako sa višekratnom upotrebom tako i sa kontrolom pristupa članovima klase. Ako bi se kompletna klasa kojim slučajem našla u glavnom programu ili njegovom ekvivalentu, tada bi njen *izvorni kod* bio deo glavnog programa i to dostupan u celosti, te bi se prvo onemogućila višekratna upotreba i drugo svaka zaštita faktički bi postala besmislena. Samo ako se klasa uključi u poseban modul, kontrola pristupa dobija praktičan smisao jer se izvorni kod modula uvek smatra nedostupnim. Prema tome, može se reći da mehanizmi zaštite unutar klase služe za kontrolu pristupa za klijente-programe, dok modul obezbeđuje dodatnu zaštitu od klijenata-programera. Da rezimiramo: kontrola pristupa članovima klase ostvaruje se kombinacijom dvaju mehanizama:

1. Klasa se realizuje kao modul ili smešta u modul, tako da njen izvorni kod postaje nedostupan.

2. Dalje zatvaranje zaštićenih članova klase realizuje se sredstvima objektnog programiranja, tj. deljenjem klase na segmente "private", "public" itd.

4.2.1. Moduli i C++

Način realizacije modula u programskom jeziku C++ preuzet je, bez ikakvih izmena, iz procedurnog dela, tj. iz programskog jezika C. S obzirom na to, za realizaciju klase koristi se druga varijanta iz prethodnog odeljka, tj. varijanta u kojoj klasa pripada modulu. Modul se izvodi, pod nazivom ***biblioteka***, iz izvorne datoteke tako što se ona deli na dva dela:

1. ***Zaglavlje*** sa ulogom interfejsa modula koje, po konvenciji, ima ekstenziju *.H* ili *.HPP*. Zaglavlje je tekstuelna datoteka u kojoj se nalaze naredba *class* za deklarisanje klase, prototipovi slobodnih funkcija, konstantni objekti (tj. objekti-konstante) i sve ostalo što je u logičkoj vezi sa klasom (klasama, ako ih ima više). Očigledno, u sklopu zaglavlja nalazi se kompletan interfejs klase. Dobra praksa je da se zaglavlje snabde pretprocesorskom direktivom za uslovno prevođenje *#ifndef ... #endif*. Zaglavlje se ne prevodi autonomno. Uključuje se u klijent u izvornom (tekstuelnom) obliku putem direktive *#include*.
2. ***Telo modula*** čiji se izvorni kod upisuje u datoteku koja konvencionalno ima ekstenziju *.CPP*. Ova datoteka sadrži neposredne realizacije funkcija-članica čiji se prototipovi nalaze u naredbi *class* iz zaglavlja, zatim slobodne funkcije, kao i sve komponente koje su lokalne za modul. Prevodi se nezavisno i koristi se u prevedenom obliku. Način uključivanja u program zavisi od konkretnog razvojnog sistema, a obično je to preko tzv. "projekta".

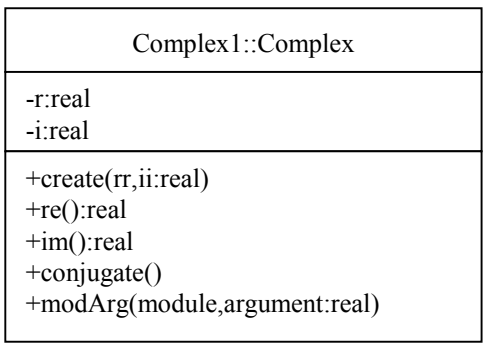
Odmah napominjemo da je ovakav razmeštaj klase u modul samo tipičan, ali ne i obavezan. S obzirom na labavu strukturu biblioteka u C/C++, jasno je da se i realizacije svih funkcija, po potrebi, mogu naći u zaglavlju, iako to, u opštem slučaju, nije dobro rešenje jer produžava proces kompilacije. Ipak, kod jedne vrste klasa, tzv. generičkih klasa, smeštanje kompletnog sadržaja klase u zaglavlje obavezno je. Druga stvar na koju bismo skrenuli pažnju jeste činjenica da je izdvajanje realizacija običnih metoda iz naredbe *class*, u stvari, posledica dvodelne strukture modula! Takođe, biblioteka u ulozi modula dozvoljava smeštanje u njega jedne jedine klase sa pratećim elementima u vidu slobodnih funkcija ili konstantnih objekata ili, alternativno, smeštanje više logički povezanih klasa u isti modul, a u cilju smanjenja granularnosti.

Konkretnu primenu modula za realizaciju klase u C++, kao i ostvarivanje potpune kontrole pristupa demonstriraćemo na jednostavnom primeru klase koja objedinjuje neke operacije nad kompleksnim brojevima. Klasa ima sledeće karakteristike i sadržaj:

- Naziv klase je *Complex*.

- Klasa se nalazi u modulu sa nazivom *Complex1*.
- Podaci-članovi su *r* (realni deo) i *i* (imaginarni deo). Zatvoreni su.
- Metoda *create* za kreiranje kompleksnog broja.
- Metode *re* i *im* za očitavanje realnog i imaginarnog dela kompleksnog broja.
- Metoda *conjugate* za konjugovanje.
- Metoda *modArg* za određivanje modula i argumenta.
- Slobodni potprogrami *add*, *subtract*, *multiply* i *divide* za redom sabiranje, oduzimanje, množenje i deljenje kompleksnih brojeva. Pri deljenju se ne proverava vrednost imenioca.
- Operacija dodele je eksplicitno dozvoljena.

Klasa *Complex* prikazana je na slici 4.5. Uočiti kvalifikator *Complex1::* kojim se u UML označava da klasa *Complex* pripada modulu *Complex1*.



Slika 4.5

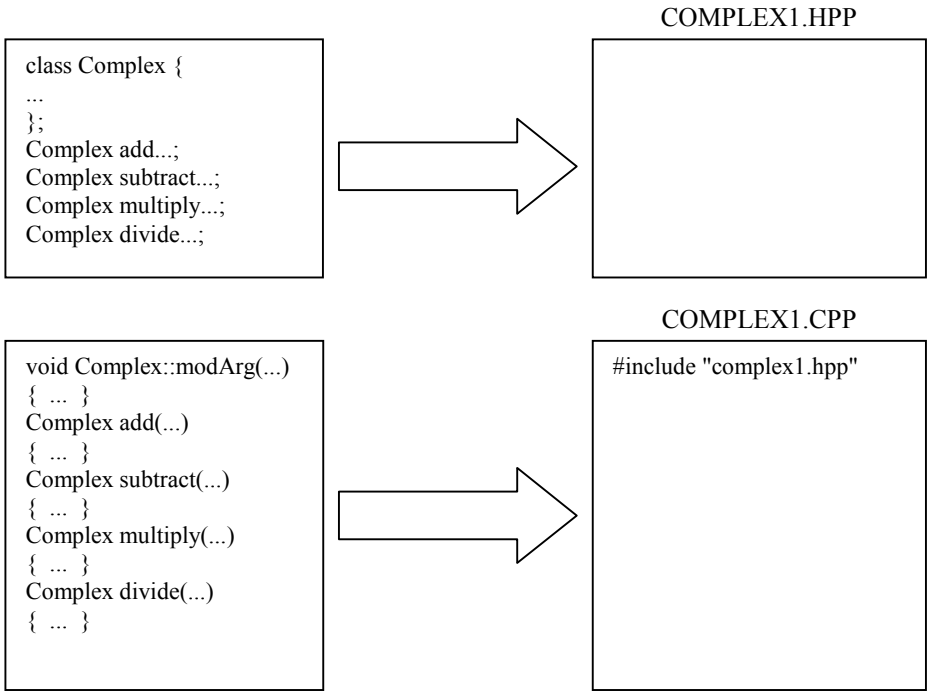
Pre no što pređemo na prikaz realizacije klase dužni smo dati jedno razjašnjenje, a u vezi sa slobodnim potprogramima *add*, *subtract*, *multiply* i *divide*. Odgovarajuće operacije mogli smo, čini se, implementirati kao metode klase *Complex*, a ne kao slobodne potprograme. Kažemo "čini se" jer se, u stvari, radi o različitim operacijama. Na primer, slobodni potprogram *add* modeluje operaciju sabiranja dva *ravnopravna* kompleksna operanda. Drukčija je situacija ako se opredelimo za metodu, jer se tada radi o *unarnoj operaciji nadodavanja* drugog operanda na prvi[34], pri čemu dolazi do promene stanja prvog operanda, a drugi operand igra ulogu parametra[35]. Isto važi i za operacije oduzimanja, množenja i deljenja.

Način razmeštanja klase *Complex* i pratećih funkcija *add*, *subtract*, *multiply* i *divide* u modul sa datotekama COMPLEX1.HPP i COMPLEX1.CPP prikazan

[34] Slobodna operacija sabiranja i operacija nadodavanja se u C-u realizuju kao različite operacije redom "+" i "+=".

[35] Razlika je ista kao kad kažemo "saberi x sa 1", odnosno "uvećaj x za 1". Posle prve operacije x zadržava raniju vrednost, a posle druge menja je na novu, za 1 veću.

je na slici 4.6. Pošto su metode *create*, *re*, *im* i *conjugate* kratke biće realizovane kao *inline* metode. Metodu *modArg* nećemo realizovati kao *inline*. U skladu sa stilom programiranja u C++, slobodne funkcije *add*, *subtract*, *multiply* i *divide* koje su logički čvrsto povezane sa klasom realizovaćemo kao prijateljske funkcije. Sledi kompletan sadržaj modula COMPLEX1.



Slika 4.6

```
/*****************************************************************
*******
                INTERFEJS MODULA COMPLEX1 (Kompleksni broj)

Sadrzaj:
 - Metoda create za kreiranje (inline)
 - Metode re i im za ocitavanje realnog i imaginarnog dela (inline)
 - Metoda conjugate za konjugovanje (inline)
 - Prototip metode modArg za nalazenje modula i argumenta
 - Prototipovi kooperativnih funkcija add, subtract, multiply, divide
   za sabiranje, oduzimanje, mnozenje i deljenje kompleksnih brojeva

Datoteka: COMPLEX1.HPP
*****************************************************************
```

```
******/

#ifndef COMPLEX1_HPP
  #define COMPLEX1_HPP

class Complex {
private:
  double r, i;
public:
  void create(double rr, double ii) { r = rr; i = ii; }
  double re() { return r; }
  double im() { return i; }
  void conjugate() { i = -i; }
  void modArg(double &, double &);
  friend Complex add(Complex, Complex);
  friend Complex subtract(Complex, Complex);
  friend Complex multiply(Complex, Complex);
  friend Complex divide(Complex, Complex);
};

#endif
```

```
/*****************************************************************
*******

                TELO MODULA COMPLEX1 (Kompleksni broj)

Sadrzaj:
 - realizacija metode modArg
 - realizacija kooperativnih funkcija add, subtract, multiply, divide
   za sabiranje, oduzimanje, mnozenje i deljenje kompleksnih brojeva

Datoteka: COMPLEX1.CPP
*****************************************************************/

#include <math.h>
#include "complex1.hpp"

void Complex::modArg(double &mod, double &arg) {
  mod= sqrt(r*r+i*i);
```

```
  arg= (r==0 && i==0) ? 0 : atan2(i,r);
}
Complex add(Complex z1, Complex z2) {
  Complex result;
  result.r= z1.r + z2.r; result.i= z1.i + z2.i;
  return result;
}
Complex subtract(Complex z1, Complex z2) {
  Complex result;
  result.r= z1.r - z2.r; result.i= z1.i - z2.i;
  return result;
}
Complex multiply(Complex z1, Complex z2) {
  Complex result;
  result.r= z1.r*z2.r - z1.i*z2.i;
  result.i= z1.r*z2.i + z1.i*z2.r;
  return result;
}
Complex divide(Complex z1, Complex z2) {
  Complex result; double temp = z2.r*z2.r + z2.i*z2.i;
  result.r= (z1.r*z2.r + z1.i*z2.i) / temp;
  result.i= (z1.i*z2.r - z1.r*z2.i) / temp;
  return result;
}
```

Primeri primene klase su:

```
#include <math.h>
#include "complex1.hpp"

Complex a, b, c, d;
double moda, arga;

a.create(1, 2);     // Kreiranje broja (1,2)
b.create(3, 4);
c= add(a, b);     // c = a + b
d= multiply(a, add(b,c));     // d = a * (b + c)
a.modArg(moda, arga);     // Odredjivanje modula i argumenta a
```

Zapazimo čudan oblik parametara metode *modArg*. Radi se o tzv. referencama, me-

hanizmu za prenos argumenata po adresi koji je uveden u C++ kao zamena za neposrednu primenu pokazivača. Detaljniji osvrt na reference daćemo u sledećem odeljku.

Ponuđeno rešenje, korektno iz aspekta sintakse, ipak nije naročito kvalitetno sa stanovišta praktične primene. Naime, klasa sa operacijama koje se mogu kombinovati u složene izraze, teško se koristi kada su operacije realizovane kao funkcije. Neka su date instance

Complex a, b, c, d, e, f, g;

Jednostavniji kompleksni izrazi poput a+b ili a+b*c bez osobitih problema realizuju se pomoću slobodnih funkcija *add* i *multiply* kao

add(a, b) odnosno add(a, multiply(b,c)).

Međutim, u samo malo složenijim slučajevima izrazi postaju programski komplikovani te, samim tim, podložni greškama. Primerice, matematička formula

g = a * (b+c-d) / (e+f)

programirala bi se izrazom

g= divide(multiply(subtract(add(b,c),d),a),add(e,f))

koji je očigledno komplikovan (sama formula nije!) i nudi obilje mogućnosti za grešku. Problem, svakako, nije nerešiv jer se izraz može računati po delovima uz korišćenje pomoćnih promenljivih ali rešenje i dalje ostaje ponešto nezgrapno. Programski jezik C++ daje mogućnost za daleko kvalitetniju soluciju upotrebom tzv. preklapanja operatora koje će biti prikazano u posebnom poglavlju.

4.2.1. Reference

Programski jezik C++ sadrži jednu novinu koja se odnosi na njegov proceduralni deo. Radi se o tipu koji se zove ***referenca***, ***alijas*** ili ***upućivač***. Referenca na neku programsku kategoriju (promenljiva, objekat) definiše se kao njen *alternativni naziv*, pa otud i termin "alijas". Prikazuje se tako što se iza osnovnog tipa navede simbol &. Dakle, konstrukt

int &

služi za definisanje reference na tip int. Tip reference je srodan pokazivačkom tipu utoliko što se i jedan i drugi realizuju kao *adrese*. Segment

```
double x = 1.5;
double &rx = x;
```

definiše realnu promenljivu *x* sa početnom vrednošću 1.5 i referencu *rx* koja upućuje na *x* tako što je njen stvarni sadržaj *adresa x*. Pristup promenljivoj *x* preko reference *rx* je indirektan, kao i preko pokazivača, ali ne zahteva operator dereferenciranja. Da bi se dohvatio sadržaj *x* dovoljno je napisati *rx* bez ikakvih sintaksnih dodataka. Jednom postavljena na neku adresu, referenca je više ne može menjati, u čemu se bitno razlikuje od pokazivača. Primeri:

```
double x = 1.5, &rx = x, y;
y= x;    // Promenljiva y dobija vrednost 1.5
y= rx;    // Isto!
x= -5.6;    // Sada je x i rx jednako -5.6
y= rx;    // Promenljiva y jednaka je -5.6
rx= 0;    // Sada je x i rx jednako 0
rx= &y;    // Greska! Nije dozvoljena izmena vrednosti same reference.
&rx= y;    // Takodje nije dozvoljeno
```

Očigledno, osnovna promenljiva *x* i referenca *rx* upotrebljavaju se simultano, kao da su sinonimi. Razlika između njih ipak postoji, a sastoji se u tome što se vrednost nalazi u memorijskom prostoru koji zauzima promenljiva *x*, dok referenca *rx* sadrži samo adresu tog memorijskog prostora. Po sadržaju, referenca se poklapa sa pokazivačem - oba tipa kao sopstvenu vrednost sadrže adresu. Ovde, kao što je pomenuto, postoji razlika i ogleda se u tome što se ta adresa u pokazivaču može menjati, a u referenci ne. Operacija dodele primenjena na referencu može da se odnosi samo na sadržaj lokacije čija se adresa nalazi u referenci, te su otud dve poslednje naredbe u gornjem segmentu neispravne.

Na osnovu izloženog čitalac će verovatno postaviti pitanje "Lepo, a šta se time dobija?". Biće u pravu jer glavna svrha reference u C^{++} nije unošenje konfuzije u program upotrebom dva ili čak više imena za istu promenljivu, nego nešto sasvim drugo. Radi se o eliminisanju nezgrapnog korišćenja pokazivača za prenos po adresi izlaznih parametara slobodnih funkcija i funkcija-članica klase. Standardna varijanta programskog jezika C predviđa da se izlazni parametri obavezno definišu kao pokazivači, da bi funkcija formirala njihovu vrednost na originalnim lokacijama. Neka funkcija *minMax* treba da generiše izlazni parametar *min* jednak min(x,y) i izlazni parametar *max* jednak max(x,y), gde su *x* i *y* parametri tipa *double*. Funkcija ima prototip

```
void minMax(double, double, double *, double *);
```

i realizaciju

```
void minMax(double x, double y, double *min, double *max) {
  *min= (x<y) ? x : y;
  *max= (x>y) ? x : y;
}
```

Uočimo obaveznu primenu operatora dereferenciranja * koja svakako ne olakšava programiranje, naročito kada se radi o pokazivačima na pokazivače i sličnim poslasticama. Još veću nelagodnost izaziva aktiviranje takvih funkcija jer zahteva eksplicitno slanje adrese. Ako je u pitanju skalarna izlazna promenljiva, to zahteva da se primeni adresni operator &[36]. Za naš primer, aktiviranje minMax izgledalo bi ovako:

```
double a, b, minab, maxab;
. . . . . .
minMax(a, b, &minab, &maxab);
```

Upotreba referenci za definisanje izlaznih parametara u potpunosti odstranjuje dereferenciranje u funkciji kao i primenu adresnih operatora pri njenom pozivu, ostavljajući kao posebnost samo konstrukt *tip &* u zaglavlju funkcije[37]. Ako u gornjem primeru zamenimo pokazivače referencama dobijamo novo rešenje:

```
void minMax(double, double, double &, double &);     // Prototip

void minMax(double x, double y, double &min, double &max) {
  min= (x<y) ? x : y;
  max= (x>y) ? x : y;
}
```

Pojednostavljuje se i poziv *minMax*:

```
minMax(a, b, minab, maxab);
```

Inače, na parametre tipa reference mogu se primeniti isti modifikatori kao i na druge parametre. Ako se koristi modifikator *const* pri pozivu se prosleđuje adresa argumenta, ali bez mogućnosti njegove izmene.

[36] Kome nije, bar u početku, zasmetao način poziva funkcije *scanf*?
[37] Paskal, na primer, od svoje prve varijante koristi slično, jednostavno, rešenje.

4.3. PRIMER: KLASA "SEMAFOR"

U ovom primeru detaljno ćemo obraditi softverski model semafora naznačen u poglavlju 2. Naš zadatak vezan je za tipičnu *apstrakciju entiteta* (videti prethodno poglavlje) jer je semafor realna stvar. Ovaj tip apstrakcije, inače, predstavlja najkorisniji i najbolji tip apstrakcije jer se modeluje klasni pojam za stvar koja se karakteriše *stanjima*. Postoji bitna razlika između klase entiteta SEMAFOR i klase entiteta KOMPLEKSNI BROJ iz prethodnog primera jer se kompleksni brojevi karakterišu *vrednostima*, zbog čega se klasa *Complex* može se tretirati kao objektna realizacija tipa podataka. Ono što, dakle, želimo je da se napravi programski (softverski) model klasnog pojma za stvar "automobilski semafor".

Pre nego što se pristupi izradi softvera, pa i u sasvim jednostavnim situacijama (koje se često pokazuju mnogo komplikovanijim nego što se to čini na prvi pogled), dakle *uvek* se analizira problem, što kod objektne paradigme znači da za buduću klasu treba definisati

- identitet (u ovom slučaju naziv klase)
- strukturu odnosno skup stanja i
- ponašanje odnosno interfejs klase.

Klasa će nositi naziv *Semaphore*. Strukturu i ponašanje objekata klase modelujemo ne izlazeći iz domena problema ili, preciznije, trudeći se u najvećoj mogućoj meri da izbegnemo bilo kakva razmišljanja o realizaciji. Činjenica da je reč o apstrakciji entiteta veoma olakšava ovakav pristup. Pitanja koja se postavljaju su tipa:

- U kojim stanjima može da se nađe semafor?
- Koje su operacije koje se mogu vršiti nad semaforom, uključujući i operacije što služe samo za očitavanje, tj. formiranje izlaza na osnovu stanja, a bez njegove promene?

Pretpostavićemo da stanja koja karakterišu semafor mogu biti[38]

- u kvaru
- isključen
- uključen sa crvenom bojom
- uključen sa žutom bojom
- uključen sa zelenom bojom
- uključen sa trepćućom žutom bojom.

Neka je *Status* skup od tri elementa:

$$\text{Status} = \{\text{sOUT}, \text{sOFF}, \text{sON}\}$$

[38] Napominje se da nemamo aspiracije na sveobuhvatno rešenje jer bismo morali uzeti u obzir različite tipove semafora (ima ih i sa trepćućom zelenom, na primer).

koji opisuju situacije redom "u kvaru", "isključen" i "uključen". Neka je, dalje,

Color = {cRED, cREDYELLOW, cYELLOW, cGREEN, cBLINK}

skup čiji elementi odgovaraju različitim bojama koje se mogu pojaviti na semaforu, gde cREDYELLOW označava uključenu crvenu i žutu boju istovremeno, a cBLINK trepćuću žutu.

Skupovi *Status* i *Color* određuju prostor stanja semafora koji obuhvata sva, pa i nedozvoljena, stanja. Taj prostor stanja je dat Dekartovim proizvodom skupova *Status* i *Color*, tj.

Status x Color

gde su neka stanja regularna (dozvoljena), npr.

(sON, cRED)

a neka nisu, npr.

(sOFF, cGREEN) ili (sOUT, cRED).

Za naš slučaj, problem predstavljaju upravo ova poslednja stanja iako se bez muke mogu programski zaobići tako što se u metode (dakle algoritamski deo) ugradi činjenica da, na primer, u stanjima sa sOFF ili sOUT boja može biti bilo koja, a programski zakloni tipa *if* sprečavaju da se ta boja očita. Drugim rečima, operacija očitavanja boje semafora može da se izvede samo ako stanje nije sOFF ili sOUT. Osnovni nedostatak ove varijante leži u činjenici da *to nije realna situacija* jer kada je semafor u kvaru ili je isključen na njemu nije uključeno ni jedno svetlo, tj. *nema nikakve boje*! Iza ovog nedostatka brzo se pomaljaju drugi problemi kao što su povećana verovatnoća greške u programiranju (ako se na nekom mestu zaboravi na proveru) i komplikovanije testiranje jer se posebno moraju predvideti slučajevi koji obuhvataju baš ovakva stanja. Poznato je, uopšte, da je softverski model tim bolji (pouzdaniji i jednostavniji za testiranje) što je bliži realnoj situaciji u domenu problema. Otuda i preporuka da se, tokom modelovanja apstrakcijom entiteta, ne izlazi iz domena problema, tj. da se izbegava prejudiciranje realizacije. U skladu sa svim rečenim, opredelićemo se za rešenje koje vernije odslikava stvarnu situaciju: kada je semafor isključen ili pokvaren *boje nema*. To, pak, znači da ćemo skup *Color* proširiti još jednom vrednošću boje, cNONE (nikakva), a koja je aktuelna u stanjima u kojima je status sOFF ili sOUT, tako da je sada

Color = {cNONE, cRED, cREDYELLOW, cYELLOW, cGREEN, cBLINK}.

Podaci-članovi klase *Semaphore* biće

$$\text{status} \in \text{Status} \quad \text{i} \quad \text{light} \in \text{Color}.$$

Razmotrimo, sada, kakve se operacije (dakle, buduće metode) mogu vršiti nad semaforom. Izbor metoda u biti nije jednostavan posao, - nedovoljno iskusni programeri skloni su da u klasu ugrade premnogo metoda (sve što im padne na pamet, među programerima poznato kao "featurism"), što otežava korišćenje, jer klijent mora te metode na *nauči*. Srećom, problem je mnogo manji kod apstrakcije entiteta jer je skup metoda u dobroj meri određen realnom situacijom, odn. nalazi se u domenu problema. Odlučićemo se za sledeće operacije:

- uključiti semafor
- isključiti semafor
- semafor se kvari
- semafor je popravljen
- ustanoviti (očitati) status semafora
- uključiti crvenu boju
- uključiti žutu boju
- uključiti zelenu boju
- prebaciti semafor u režim sa trepćućom žutom
- ustanoviti aktuelnu boju semafora (uključujući i "nikakvu" i trepćuću žutu).

I ovo na prvi pogled izgleda vrlo jednostavno, skoro trivijalno. Međutim, ne treba izgubiti iz vida da među operacijama ima i takvih koje su zavisne od tekućeg stanja ili su čak međusobno zavisne. Tako, na primer, semafor može da se pokvari bez obzira na trenutno stanje, osim ako je već pokvaren. Dalje, semafor može da se uključi samo ako je bio u stanju sa sOFF jer

- ako je bio u stanju sOUT treba ga prethodno popraviti, a
- ako je već bio u stanju sON to znači da je bio uključen, pa u domenu problema takva akcija ne može da se pojavi (da se uključen semafor uključuje).

Zapazimo da su prethodna dva slučaja suštinski različita sa modelske tačke gledišta. Naime, u prvom slučaju radi se o pokušaju da se izvede operacija u stanju u kojem nije izvodljiva (uključivanje pokvarenog semafora), dok se drugi slučaj odnosi na trivijalnu operaciju koja ne izaziva nikakve promene. Ni operacije promene boje nisu nezavisne. Recimo, jedina dozvoljena "boja" za sOUT i sOFF je cNONE; isto tako, posle boje cGREEN ne može se uključiti direktno cRED i sl. Načine promene boje rezimiraćemo ovako:

1. Za status sOUT i sOFF boja je cNONE.
2. Po ulasku u stanje sON automatski se uključuje cBLINK (trepćuća žuta).
3. U stanju sON moguće su boje cRED, cREDYELLOW, cYELLOW,

cGREEN i cBLINK. Boja cREDYELLOW uključuje se kada se u stanju sa cRED uključi žuta (cYELLOW).

4. Iz boje cBLINK može se preći u cRED ili u cGREEN.
5. Iz bilo koje od boja može se preći u cBLINK (osim, naravno, iz same cBLINK).
6. Iz cRED prelazi se u cYELLOWRED uključivanjem cYELLOW; iz cREDYELLOW prelazi se u cGREEN; iz cGREEN prelazi se u cYELLOW, a iz nje u cRED.
7. Zahtev za uključivanje već postojeće boje ne izaziva nikakvu promenu, tj. predstavlja trivijalnu operaciju.

Za navedene probleme ne možemo unapred sugerisati jednoznačna rešenja jer se ne zna kontekst u kojem će se klasa eksploatisati. Zato je najsigurniji način (u datom slučaju semafora, ali ne generalno!)

- ignorisati nedozvoljene i trivijalne operacije, ali ugraditi u metode formiranje izlazne informacije o tome da li je operacija izvršena ili nije.

Sledeći korak u rešavanju našeg zadatka je prevođenje operacija iz domena problema u programski model; drugim rečima treba ih pretvoriti u prototipove (zaglavlja) metoda. Jedno od (mnogih) mogućih rešenja je da se predvide sledeće metode:

- setStatus(Status s) - uključivanje, isključivanje, kvarenje i popravka semafora
- getStatus - ustanovljavanje (očitavanje) tekućeg statusa.
- setColor(Color c) - postavljanje boje
- getColor - očitavanje tekuće boje

pri čemu metode *setStatus* i *setColor* vraćaju 1 (true) ako su uspešno okončane, odnosno 0 (false) ako iz bilo kojeg razloga ne mogu da se izvedu. Za ovaj primer nije neophodno predvideti prevremeno okončanje programa pri pokušaju da se izvrši neregularna operacija jer njihova primena ne može uticati na tačnost rezultata. Treba još jednom napomenuti da ovakav pristup nije generalno primenljiv: pokušaj, recimo, upisa u pun, sekvencijalno realizovan stek mora izazvati momentalan prekid izvršenja (uz odgovarajuću poruku) jer su posledice nepredvidive, tipa "jump in blue".

Kao poslednje, ali nikako i najmanje važno, treba predvideti metodu za inicijalizaciju objekta. Ove metode (može ih biti i više u sklopu iste klase) su od velike važnosti jer postavljaju objekat u početno stanje. Neinicijalizovani objekti ponašaju se po pravilu nepredvidivo: u konkretnom slučaju, primena bilo koje od četiri navedene metode na neinicijalizovan objekat teorijski nije moguća, ali praktično, zbog C^{++}-ovskih specijaliteta, vodi u neizvesnost. Uopšte, treba se pridržavati principa

- Ako se za objekat može definisati početno stanje (stanja) tada se on obavezno inicijalizuje (najčešći slučaj kod apstrakcije entiteta). U ostalim slučajevima odluka zavisi od konkretne situacije.

Metoda za inicijalizaciju nosiće naziv

- init

i postavljaće semafor u stanje (sOFF, cNONE).

Da bismo jasnije prikazali promene stanja semafora poslužićemo se dijagramima stanja UML koji pregledno pokazuju stanja i način njihove promene. Opisali smo ih u poglavlju 2. Pre nego što nacrtamo dijagram stanja semafora, obratimo pažnju na još jednu pojedinost: naime neki od prelaza uslovljeni su ne samo tekućim stanjem i porukom, nego i prethodnim stanjem (u kojem slučaju se, uopšte, kaže da "proces ima memoriju"): naime, sekvenca poruka semaforu tipa

setColor(cRED); setColor(cYELLOW); setColor(cRED)

ne bi smela biti omogućena jer je u realnim okolnostima nema! Načina za prevazilaženje ovog problema ima na pretek: primerice, može se memorisati uvek i prethodna boja, ali bi to zahtevalo zahvate u praktično svim metodama; druga mogućnost (nju smo već primenili u poglavlju 2) je da se stanje u kojem je boja žuta razloži na dva:

(sON, cYELLOWRED) i (sON, cYELLOW)

i to tako da se u stanje (sON, cREDYELLOW) ulazi iz stanja (sON, cRED), a prelazi u stanje (sON, cGREEN). Slično, u stanje (sON, cYELLOW) ulazi se iz (sON, cGREEN), a prelazi u (sON, cRED). Sada su, dakle, definisane vrednosti oba podatka-člana, *status* i *light* skupovima redom

{sOUT, sOFF, sON} i
{cRED, cYELLOWRED, cYELLOW, cGREEN, cBLINK}.

Dijagram stanja predstavljen je na slici 4.7 na kraju ovog odeljka.

Poboljšano rešenje

Iako prihvatljivo, ponuđeno rešenje ima neke nedostatke u koje se mogu ubrojiti i sledeća dva:

a) Metoda *setStatus* je nepotrebno parametrizovana. Naime, pouzdanije je, razumljivije i bliže realnosti da je zamenimo neparametrizovanim metodama sa mnemoničkim identifikatorima i to:

- turnOn() umesto setStatus(sON)
- turnOff() umesto setStatus(sOFF) i
- fail() umesto setStatus(sOUT)

i da uvedemo još jednu metodu

- repair() kada je semafor popravljen posle kvara.

b) (važnije!) U realnoj situaciji redosled promene boja je uvek isti:

crvena - žuta - zelena - žuta - crvena - žuta - ...

te nema potrebe posebno zadavati boju kao parametar: dovoljno je predvideti metodu

- changeColor()

koja će, bez uplitanja programera, a na osnovu tekućeg stanja, sama odabrati odgovarajuću boju. Pored ove metode dodaćemo još i metode

- setBlink() umesto setColor(cBLINK)
- setRed() i
- setGreen() za prelaz iz trepćuće žute u crvenu odnosno zelenu.

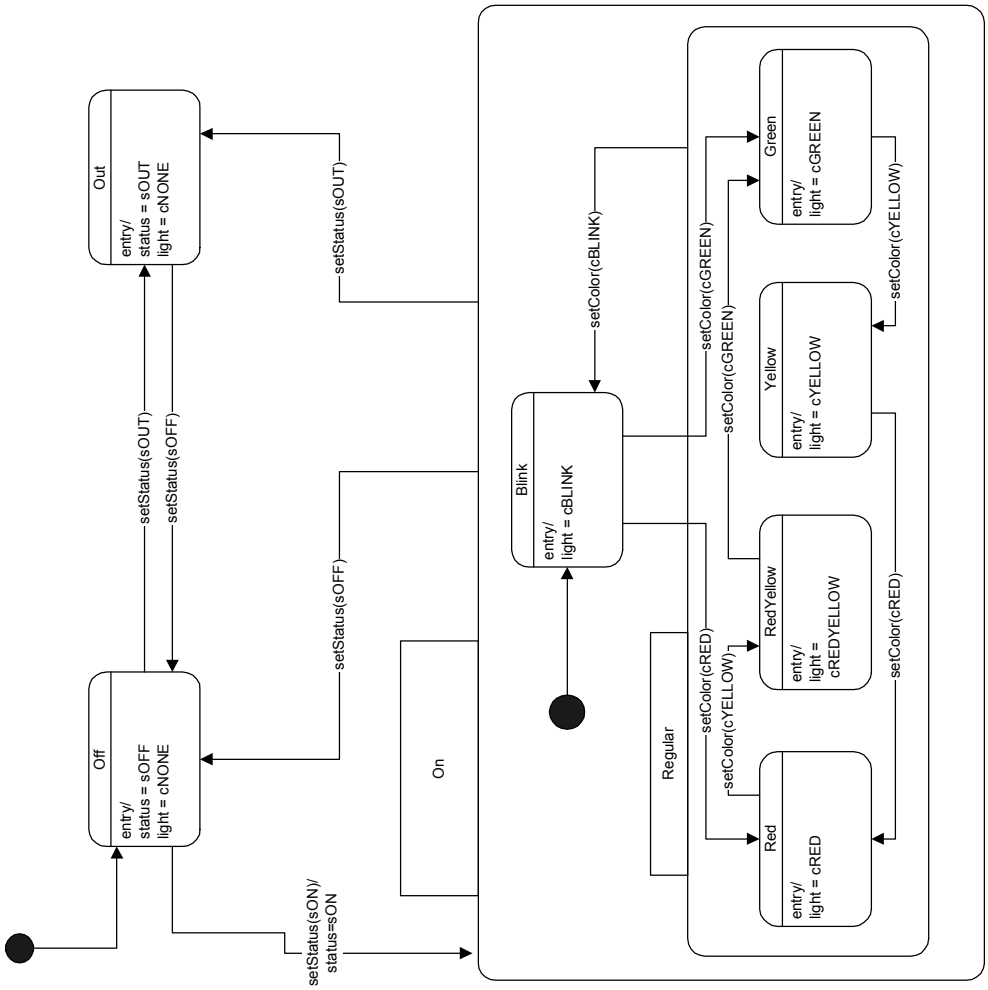
Dijagram stanja sada dobija oblik dat na slici 4.8 na kraju odeljka. Programski kod izgleda ovako:

```
/*****************************************************************
*****

                ZAGLAVLJE KLASE "SEMAPHORE"

              Metode:

              - Init za inicijalizaciju
              - turnOn za ukljucivanje
              - turnOff za iskljucivanje
              - fail za otkaz
              - repair za popravljen
              - setBlink za ukljucivanje trepcuce zute
              - setRed za prelaz sa trepcuce zute na crvenu
              - setGreen za prelaz sa trepcuce zute na zelenu
              - changeColor za promenu boje
              - getStatus (citanje statusa)
              - getColor (ocitavanje boje)


Naziv datoteke: SEMAFOR.HPP
*****************************************************************
****/
enum Status {sOUT, sOFF, sON};
enum Color {cNONE, cRED, cREDYELLOW, cYELLOW, cGREEN, cBLINK};

class Semaphore {
```

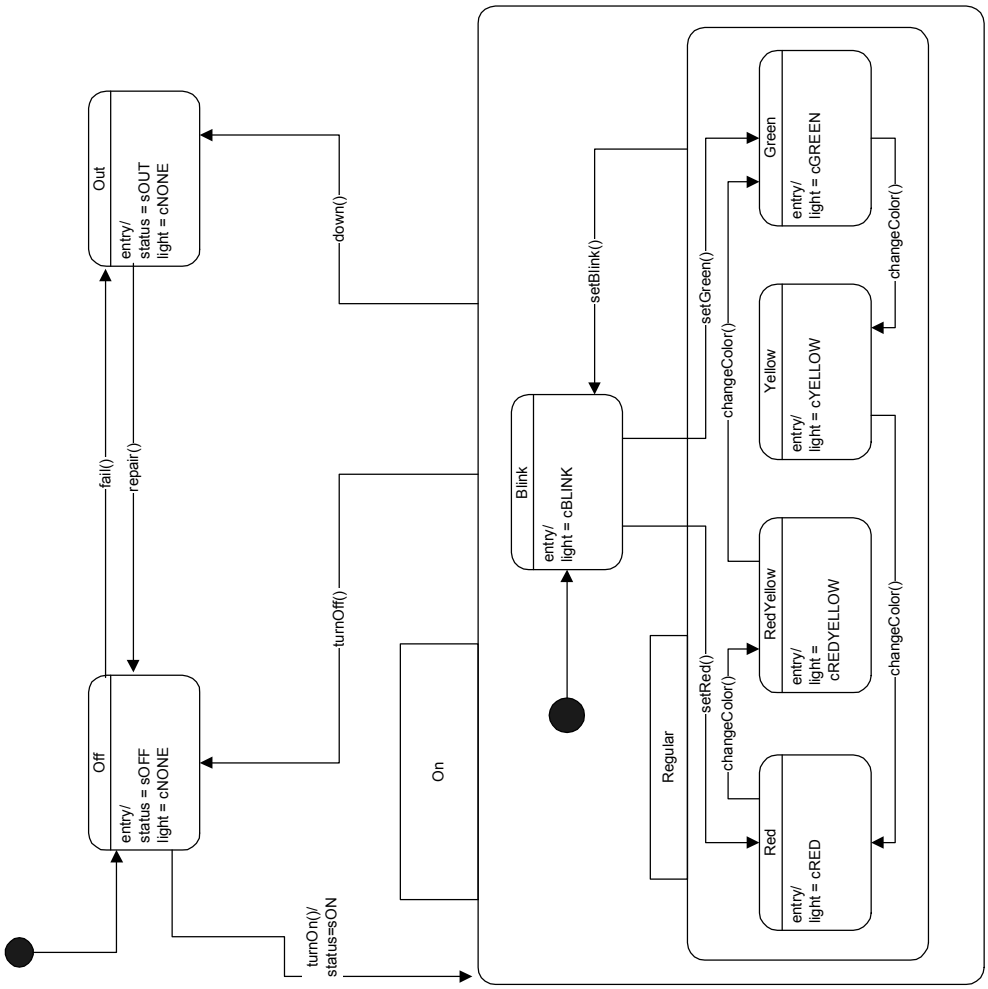
```
private:
    Status status;
    Color light;
public:
   void init() {status = sOFF; light = cNONE;}
   int turnOn();
   int turnOff();
   int fail();
   int repair();
   int setBlink();
   int setRed();
   int setGreen();
   int changeColor();
   Status getStatus() const {return status;}
   Color getColor() const {return light;}
};
```

```
/*****************************************************************

                    TELO KLASE "SEMAPHORE"


Naziv datoteke: SEMAFOR.CPP
*****************************************************************/

#include "semafor.hpp"

int Semaphore::turnOn() {
  if(status == sOFF) {status= sON; light= cBLINK; return 1;}
  else return 0;
};
int Semaphore::turnOff() {
  if(status==sON) {status= sOFF; light= cNONE; return 1;}
  else return 0;
};
int Semaphore::fail() {
  if(status != sOUT) {status= sOUT; light= cNONE; return 1;}
  else return 0;
};
int Semaphore::repair() {
```

```
  if(status == sOUT) {status= sOFF; light= cNONE; return 1;}
  else return 0;
};
int Semaphore::setBlink() {
  if((status != sON) || (light == cBLINK)) return 0;
  light= cBLINK;
  return 1;
}
int Semaphore::setRed() {
  if(light != cBLINK) return 0;
  light= cRED;
  return 1;
}
int Semaphore::setGreen() {
  if(light != cBLINK) return 0;
  light= cGREEN;
  return 1;
}
int Semaphore::changeColor() {
  if((status != sON) || (light == cBLINK)) return 0;
  switch(light) {
    case cRED: light = cREDYELLOW; break;
    case cREDYELLOW: light = cGREEN; break;
    case cYELLOW: light = cRED; break;
    case cGREEN: light = cYELLOW; break;
  };
  return 1;
}
```


Out
entry/
status = sOUT
light = cNONE
Off
entry/
status = sOFF
light = cNONE
setStatus(sOUT)
setStatus(sOFF)
setStatus(sOUT)
setStatus(sOFF)
setStatus(sON)/
status=sON
On
Blink
entry/
light = cBLINK
Regular
setColor(cBLINK)
setColor(cGREEN)
setColor(cGREEN)
setColor(cRED)
setColor(cYELLOW)
Green
entry/
light = cGREEN
Yellow
entry/
light = cYELLOW
RedYellow
entry/
light =
cREDYELLOW
Red
entry/
light = cRED
setColor(cYELLOW)
setColor(cRED)

Slika 4.7


Out
entry/
status = sOUT
light = cNONE
down()
fail()
repair()
Off
entry/
status = sOFF
light = cNONE
turnOff()
turnOn()/
status=sON
On
Blink
entry/
light = cBLINK
setBlink()
Regular
setGreen()
setRed()
changeColor()
changeColor()
changeColor()
changeColor()
Green
entry/
light = cGREEN
Yellow
entry/
light = cYELLOW
RedYellow
entry/
light =
cREDYELLOW
Red
entry/
light = cRED

Slika 4.8

5. KLASIFIKACIJA OPERACIJA. KONSTRUKTORI I DESTRUKTORI

Pre četrdesetak godina, u doba konsolidacije strukturirane metodologije, Hoar [7] ustvrdio je da "najvažniji aspekt podataka jeste način na koji se njima rukuje i opseg baznih operacija koje su na raspolaganju za tu svrhu".

Prelaskom na objektno programiranje ova tvrdnja je čak dobila na značaju budući da su operacije (tj. metode) integralni deo klase, zajednički za sve objekte. Izbor repertoara metoda predstavlja jedan od ključnih koraka prilikom definisanja klase, jer neposredno određuje ponašanje njenih instanci. Skup metoda, koji se bira donekle arbitrarno, ne sme biti preuzak zato što tada objekti nedovoljno verno modeluju ponašanje odgovarajućeg pojma. S druge strane, repertoar metoda ne sme da bude ni preterano širok, stoga što prevelik broj metoda otežava rukovanje objektima; programer se teško snalazi u mnoštvu metoda i nema jasan uvid u to koje su važne, a koje su tu jer ih je "lepo imati". Naravno, izbor metoda nije jednoznačan postupak, ali je daleko od potpuno arbitrarnog. U pomenutoj raspravi Hoar daje vrlo delotvornu preporuku za način odabira metoda koja, prevedena na jezik objektne metodologije, glasi:

- Bazni skup metoda mora biti dovoljno velik da obezbedi efikasnu realizaciju svake dodatne operacije potrebne programeru.

Ako malo pažljivije proučimo ovu preporuku primetićemo da je akcent stavljen na dve stavke: prva je zahtev da se svaka dodatna operacija može realizovati primenom baznih metoda, dakle bez učešća podataka-članova. Druga je da ta realizacija mora biti još i efikasna, naročito u pogledu brzine izvršavanja. To pak znači da same bazne metode moraju biti posebno efikasne iz čega Hoar izvlači još jedan zaključak:

- Metode uključene u klasu (tj. bazne metode) jako se oslanjaju na njenu strukturu (Hoar upotrebljava izraz "heavily depend").

Drugim rečima, metode koje pripadaju interfejsu klase u načelu ne samo što mogu nego i treba da direktno pristupaju njenim zaštićenim delovima.

Za definisanje repertoara metoda od velike koristi je poznavati klasifikaciju tipičnih operacija, dovoljno opštih da se mogu sresti kod većine konkretnih klasa.

Podvucimo odmah da se, s obzirom na praktično neograničen broj raznih operacija nad objektima, klasifikacija odnosi samo na tipične, a ne na sve moguće, metode. Metode klasifikujemo u 4 grupe. To su

1. konstruktori i inicijalizatori
2. destruktori i terminatori
3. akcesori (pristupnici, upitne funkcije) koji se dalje dele na selektore, indikatore i iteratore
4. modifikatori (komande),

pri čemu su neke od metoda specijalne, tj. zadate su u okviru jezika, a druge su obične, tj. pravi ih programer.

Zanimljivo je saznanje da ova klasifikacija, uz male izmene, potiče još iz ere strukturiranog programiranja, što dalje potkrepljuje tvrdnju da je većina bitnih karakteristika objektne metodologije uočena i opisana *pre* njene pojave. Činjenica da se objektna klasifikacija skoro i ne razlikuje od procedurne u stvari ne treba da čudi, jer su metode klase s jedne i procedurni operatori s druge strane samo dva modela iste kategorije - operacije. Stoga ima smisla govoriti o klasifikaciji ne samo metoda nego i operacija uopšte. Radi lakšeg razumevanja, naporedo sa razmatranjem metoda iz pojedinih grupa davaćemo i primere njihovih procedurnih analogona.

Konstruktori su metode koje imaju za glavni zadatak da učestvuju u kreiranju objekta ili promenljive, sa inicijalizacijom ili bez nje. Konstruktori spadaju u red pomalo neobičnih operacija. Naime, u doba kompozitne metodologije, naredbe programskih jezika opšte namene (tipičan je fortran) delile su se na dve potpuno različite grupe: tzv. opisne naredbe (neizvršne naredbe, deklarativne naredbe) i izvršne naredbe. Opisne naredbe su se tretirale kao neka vrsta kvazinaredbi jer se smatralo da, posmatrano sa nivoa višeg programskog jezika, one ne izazivaju nikakvo dejstvo. Zadatak im je bio da proslede neke deskriptivne podatke namenjene prevodiocu. Najautentičniji primer opisnih naredbi bile su deklaracije promenljivih, pri čemu je fortran išao tako daleko da za neke tipove promenljivih (celobrojne i realne) nije ni nalagao eksplicitno deklarisanje. Nasuprot opisnim stajale su izvršne naredbe - "prave" naredbe - koje su pokretale neku akciju vidljivu sa nivoa višeg programskog jezika. U izvršne naredbe ubrajane su naredbe dodele, zatim selekcije, ciklusi, skokovi i pozivi potprograma.

Softverska revolucija sedamdesetih promenila je tretman opisnih naredbi tako što je i njima priznat status operacija. Deklaracija promenljive, na primer, jeste operacija budući da izaziva dejstvo koje se ogleda u zauzimanju memorijskog prostora i dodeli identifikacije promenljivoj. Štaviše, pojavom pokazivača uvedena je mogućnost eksplicitnog zauzimanja memorijskog prostora *u toku izvršenja programa*, a putem standardnih potprograma *malloc*, *calloc* ili *realloc* u C-u, odnosno *new* u paskalu. Operaciona priroda kreiranja promenljive postala je očigledna. Razlika između "izvršnih" i "opisnih" naredbi izbrisana je, što je široko obznanjeno

u čuvenoj studiji "Structured Programming" gde je Hoar izvršio prvu modernu klasifikaciju operacija. Prvi put pojavili su se termini "konstruktor" i "selektor" sa istim značenjem koje imaju i danas. Paskal, čedo strukturirane metodologije, pravio je razliku između naredbi za kreiranje promenljivih i ostalih izvršnih naredbi samo utoliko što ih je smeštao u različite segmente programa, bez mogućnosti mešanja. Noviji programski jezici izbrisali su i tu razliku.

Za kreiranje promenljive u C-u koristi se konstrukt *tip ime* uz moguću inicijalizaciju. Promenljiva identifikovana pokazivačem zahteva naknadnu primenu neke od funkcija kao što su *malloc* ili *calloc*. Primeri:

```
int i, j=2;     // Kreiranje celobrojnih promenljivih i i j sa incijalizacijom j
double x, y;    // Kreiranje realnih promenljivih x i y
char *pC;                       // Kreiranje znakovne...
pC= malloc(sizeof(char));    //...promenljive identifikovane pokazivacem pC
```

Objektna metodologija otišla je korak dalje u izjednačavanju operacije kreiranja sa drugim operacijama tako što je dozvolila programeru da *utiče* na proces kreiranja instance klase putem posebnih metoda - konstruktora. Dakle, više se ne radi o promeni pogleda na naredbu za deklarisanje-definisanje, nego o uvođenju mogućnosti za njeno *programiranje*. Kreiranje objekta na programskom jeziku C^{++} obavezno uključuje aktiviranje konstruktora, tačnije jednog od konstruktora jer ih može biti više. Minimalni posao koji obavlja konstruktor jeste rezervisanje memorijskog prostora, a pored toga on može da izvede doslovno sve što je programer zamislio i ugradio u kod. Dakle, kada na C^{++} napišemo

```
Complex z;
```

to više nije obično definisanje objekta sa nazivom "z" nego *eksplicitno aktiviranje* konstruktora iz klase *Complex*.

U tesnoj vezi sa konstrukcijom objekta je njegova *inicijalizacija*. Pod inicijalizacijom (objekta) podrazumeva se njegovo dovođenje u neko početno stanje zadavanjem vrednosti podacima-članovima i inicijalizacijom objekata-članova. Inicijalizaciju može izvršiti konstruktor (najčešće), ali taj posao mogu da obave i obične metode, koje zovemo ***inicijalizatori***. Inicijalizaciju objekata u klasama *Point*, *Line*, *Complex* i *Semaphore* obavljaju inicijalizatori redom *setPoint*, *setLine*, *create* i *init*. O načinima inicijalizacije pomoću konstruktora biće reči kasnije u ovom poglavlju.

Sa ***destruktorima*** stvar stoji nešto drukčije. Naime, u zavisnosti od programskog jezika i konkretne situacije, destruktor može fizički da ukloni objekat iz memorije ili samo da ga dovede u neko završno stanje, oslobađanjem resursa koje objekat koristi (npr. zatvaranjem nekih tokova povezanih sa objektom ili dealokaci-

jom pokazivačkih članova). Prvi pravi destruktori bile su operacije dealokacije pokazivača. U C-u to je funkcija *free*, koja je u C^{++} zamenjena operatorom *delete*.

U C^{++}-u metoda-destruktor je nezaobilazna pošto je, po definiciji, sadrži svaka klasa. Aktivira se automatski ili - retko - eksplicitnim pozivom, a detaljnije će biti obrađena u narednom odeljku.

U uskoj vezi sa destruktorima su metode koje ćemo nazvati ***terminatorima***. To su obične metode, ali sa posebnim zadatkom prevođenja objekta u već pomenuto završno ili, tačnije, "zamrznuto" stanje, koje se često poklapa sa početnim stanjem. Na primer, ako objekat sadrži pokazivačke članove, terminatori vrše njihovu dealokaciju. Razlika između destruktora i terminatora je u tome što su terminatori obične metode. Klasa može sadržati uporedo i destruktore i terminatore pri čemu i destruktor obavlja posao terminiranja. To je, štaviše, obavezno ukoliko u klasi postoje pokazivači koje destruktor mora dealocirati.

U vezi sa destrukcijom objekata u dinamičkoj memoriji, interesantno je napomenuti da za tu aktivnost postoji i savršeniji mehanizam od destruktora. Radi se o posebnom pozadinskom (engl. "background") procesu sa nazivom *garbage collector* ("sakupljač smeća") koji se povremeno automatski uključuje oslobađajući svu memoriju koja je u tom trenutku van upotrebe. Ovaj način destrukcije primenjen je, na primer, u programskom jeziku java. C^{++} ga nema.

Akcesori ili pristupnici jesu metode koje obavljaju pristup članovima klase. Mogu da budu takvi da samo odaberu (markiraju) član klase i tada se nazivaju selektorima. Mogu da vrate kao rezultat vrednost formiranu na osnovu stanja objekta i tada nose naziv indikatori. Konačno, mogu da obezbede sistematski, linearan pristup svim elementima objektno realizovanih struktura podataka (operacija obilaska ili prolaza) u kojem slučaju se radi o iteratorima.

Operacije koje spadaju u grupu ***selektora*** ostvaruju pojedinačan pristup promenljivim, njihovim delovima, objektima ili njihovim članovima. U proceduralnom programiranju selektori su delovi programskog jezika. Najjednostavniji selektori su selektori kod skalarnih promenljivih: aktiviraju se prostim navođenjem imena promenljive. Dakle, ako je *skalpro* skalarna promenljiva nekog tipa, tada je njen selektor

skalpro

Ulogu selektora kod nizova ima indeksiranje. Ako napišemo

niz[i]

to se tretira kao izvršenje selekcije i-tog elementa niza *niz*. Kod slogova selektor je označen simbolom ".", tako da

s.p

označava selekciju polja p koje je deo sloga s. Konačno, selekcija kod pokazivačkih promenljivih ostvaruje se dereferenciranjem pokazivača

*pPok

Selektori u objektno orijentisanom programiranju služe prvenstveno za pristup članovima klase. Radi se o operaciji koja se u svim programskim jezicima označava tačkom. Ako je *ob* neki objekat sa podatkom-članom *p* i metodom *m*, tada primena selektora ima formu

ob.p odnosno ob.m(*argumenti*)

Pored ove vrste selektora treba izdvojiti još i operaciju indeksiranja, tipičnu za nizove, kojom se na osnovu zadatog indeksa pristupa nekom internom podatku-članu, najčešće elementu niza zatvorenog u objektu. U C^{++} ova operacija nije unapred zadata, ali se može realizovati mehanizmom preklapanja operatora (videti u sledećem poglavlju). U javi, na primer, nizovi su realizovani kao objekti, te je indeksiranje operacija koja je deo jezika.

Indikatori su karakteristični za objektno programiranje i predstavljaju, pre svega, obične metode. Indikatorom se očitava neki podatak-član ili objekat-član. U ovom drugom slučaju dejstvo indikatora poklapa se sa dejstvom odgovarajućeg selektora, s tim što je, za razliku od selektora ".", ovde u pitanju obična metoda. U klasi *Complex* indikatori su npr. *re* i *im*, u klasi *Point* metode *getX* i *getY*, a u klasi *Semaphore* metode *getColor* i *getStatus*. Inače, jedan od jednostavnih testova potpunosti repertoara metoda neke klase jeste provera da li se pomoću skupa indikatora može tačno utvrditi trenutno stanje objekta. Tako, na primer, metode *re* i *im* u klasi *Complex* zajedno daju informaciju o stanju odgovarajućeg objekta, kao i metode *getX* i *getY* u klasi *Point*. Prirodu indikatora imaju i metode koje izveštavaju o tome da li se objekat nalazi u nekom specijalnom stanju (na primer, metoda koja utvrđuje da li je stek prazan ili nije). Inače, selektori i indikatori su bliski po semantici i nose zajednički naziv ***akcesori*** (pristupnici). Pošto u imenu često imaju prefiks *get*, u programerskom žargonu akcesori nose naziv "geteri". Već pomenuta operacija indeksiranja ponaša se kao selektor kada se nalazi sa leve strane znaka dodeljivanja, a kao indikator kada učestvuje u izrazu.

Iteratori sačinjavaju dosta specifičnu grupu operacija koje se vezuju za složene tipove i klase. Iterator je operacija koja ostvaruje sistematski pristup svim delovima kompozitne promenljive. Tako, iterator za niz u svim procedurnim jezicima jeste brojački *for* ciklus. Iteratori objekata su obične metode, a tipičnu primenu nalaze u klasama što modeluju strukture podataka kao što su dinamički

niz, lista ili binarno stablo kod kojih je operacija pristupa svim elementima jedna od najčešće korišćenih. Inače, iteratori obično nisu pojedinačne metode već se realizuju kroz dve-tri metode: recimo, metoda za otpočinjanje iteracije, zatim metoda za prelazak na sledeću iteraciju i najzad metoda za proveru kraja iteriranja.

Modifikatori su operacije kojima se menja vrednost promenljivih odn. stanje objekta. Najpoznatiji modifikator je operacija dodele, a tu su i operacije ++, --, +=, -=, *= itd. u programskom jeziku C ili, na primer, procedure *Inc* i *Dec* u paskalu. U klasama, modifikatori su metode. Praktično, nema klase bez modifikatora budući da je u prirodi objekta da bude aktivan, tj. da menja stanja. I ne samo to: klase najčešće imaju više modifikatora, a njihov broj može biti impozantan. Tipični modifikatori su *conjugate* iz klase *Complex* ili *turnOff*, *turnOn*, *fail*, *repair*, *setBlink*, *setRed*, *setGreen* i *changeColor* iz klase *Semaphore*. Formalno, inicijalizatori i terminatori mogli bi se svrstati u modifikatore, ali je bolje povezati ih sa konstruktorima odn. destruktorima imajući u vidu da su rezultujuća stanja specifična. Zbog čestog prefiksa *set* nose kolokvijalni naziv "seteri".

5.1. KONSTRUKTORI

Pod ***kreiranjem (stvaranjem) objekta*** podrazumevamo postupak izgradnje jedne instance date klase. Kreiranje objekta je složena operacija koja obuhvata tri aktivnosti:

1. davanje identiteta objektu,
2. zauzimanje memorijskog prostora za objekat i
3. inicijalizaciju objekta.

Pritom, objektno orijentisani jezici koncipirani su tako da dodela identiteta objektu istovremeno obezbeđuje mehanizam pristupa njegovom memorijskom prostoru. Takođe, aktivnosti identifikovanja objekta, dodele memorije i inicijalizacije mogu se izvršiti odjednom (jednom naredbom) ili u koracima koji nisu obavezno sukcesivni. Postupak kreiranja zavisi kako od programskog jezika tako i od konkretne situacije. Na primer, dinamički objekti[39] u nekim programskim jezicima (C^{++}, java) kreiraju se u jednom ili dva koraka, dok se u paskalu to čak ni ne može izvesti jednom naredbom.

Specifičnu ulogu u kreiranju objekta igraju ***konstruktori***. Moramo odmah podvući da, pored načina korišćenja, i sama namena konstruktora jako zavisi od programskog jezika. U jezicima poput C^{++}, jave ili Object Pascal-a konstruktori zauzimaju centralno mesto u procesu kreiranja i predstavljaju jednu od najvažnijih vrsta metoda uopšte. U (Turbo) paskalu konstruktori imaju nešto užu oblast primene; štaviše, neki objekti mogu da se kreiraju i bez primene konstruktora te, za razliku od prethodno pomenutih jezika, paskal dozvoljava definisanje klasa bez

[39] Objekti koji se stvaraju i uništavaju tokom izvršenja programa, a pristupa im se preko pokazivača odn. adrese.

ikakvih konstruktora. Uopšte uzev, konstruktor može da

- kreira objekat u celosti,
- učestvuje u kreiranju objekta ili
- ne učestvuje u kreiranju objekta

a sve u zavisnosti od programskog jezika i konkretnog rešenja. Konstruktori imaju jednu osobinu koja ih odvaja od ostalih metoda. Naime,

- konstruktori su metode *nivoa klase*, a ne pojedinačnih objekata, jer se operacija kreiranja objekta vrši nad klasom[40]!

U većini objektno orijentisanih jezika (izuzetak je paskal) konstruktor kao metoda ni ne može da se primeni na objekat nego samo na klasu; preciznije, sintaksna fraza za aktiviranje konstruktora ne sadrži ime objekta nego ime klase. Međutim, i kada je konstruktor definisan na nivou klase on ipak ostaje metoda, što znači da se može, a najčešće i mora, programirati. Upravo u tome leži razlika između kreiranja promenljive koja nije instanca klase i kreiranja objekta. Promenljiva se definiše (deklariše) naredbama koje su deo programskog jezika i na koje programer nema upliv u smislu mogućnosti modifikovanja. Nasuprot tome, ako se objekat kreira putem konstruktora, programer na to ima uticaja jer je u mogućnosti da konstruktor, kao i svaku drugu metodu, programira.

5.1.1. Konstruktori u C^{++}

Imajući u vidu raznovrsnost uloga konstruktora u C^{++} nećemo pogrešiti ako kažemo da oni spadaju u najvažnije metode uopšte. No, pre nego što nabrojimo aktivnosti koje konstruktori obavljaju neposredno ili pak učestvuju u njima, izvršićemo klasifikaciju C^{++}-objekata saobrazno načinu kreiranja-uništavanja i ulozi u procesu obrade (prema [29] i [77]). Po tim kriterijumima objekti se dele na sledeće kategorije:

1. *Automatski objekti* koji se kreiraju svaki put kada se naiđe na njihovu definiciju i destruišu kada izađu iz opsega. Odgovaraju automatskim promenljivim iz procedurnog dela. Automatske objekte dalje delimo na imenovane i bezimene. *Imenovani objekti* su "obični" objekti kojima se rukuje preko imena. *Bezimeni objekti* dobijaju se eksplicitnim pozivom konstruktora, a ponašaju se kao rezultat poziva funkcije, tj. učestvuju u izrazima.
2. *Implicitni objekti* nastaju u toku računanja vrednosti nekog izraza koji sadrži objekte kao međurezultate. Shodno tome, brigu oko kreiranja i destrukcije implicitnih objekata vodi prevodilac, dok programer o njima ne mora da zna ništa, čak ni da postoje.
3. *Statički objekti* se formiraju jednom, na početku programa i destruišu se na njegovom završetku. Odgovaraju statičkim promenljivim.
4. *Dinamički objekti* su objekti koji se nalaze u dinamičkoj memoriji pro-

[40] Bilo bi nelogično poslati poruku "kreiraj se" objektu koji još ne postoji.

grama (dobro poznatom hipu, eng. "heap") i kojima se manipuliše posredstvom pokazivača. Odgovaraju pokazivačkim promenljivim.

5. *Objekti-članovi* već su više puta pominjani. Sadržani su u drugim objektima.

Posebnu vrstu objekata čine *konstantni objekti*. Odgovaraju imenovanim konstantama (ili, kako se još nazivaju, "zaključanim promenljivim") u procedurnom delu C/C^{++}. To su objekti koji ne mogu da promene početno stanje zadato pri inicijalizaciji.

Sada smo u mogućnosti da nabrojimo primene konstruktora u programskom jeziku C^{++}:

1. Kreiranje imenovanih objekata
2. Kreiranje bezimenih objekata
3. Kreiranje implicitnih objekata
4. Učestvovanje u kreiranju dinamičkih objekata (objekata na hipu)
5. Kreiranje objekata-članova
6. Kreiranje konstantnih objekata
7. Inicijalizacija objekata
8. Posredovanje u prenosu objekata kao argumenata u slobodne funkcije ili metode (tj. kreiranje kopija objekata na steku)
9. Eksplicitna ili implicitna konverzija tipa (sledeća glava)
10. Omogućavanje upotrebe tzv. virtuelnih metoda koje će biti obrađene u posebnom poglavlju.

Kako vidimo, konstruktori se pojavljuju u raznovrsnim ulogama, ali svi imaju jednu svrhu koja je zajednička: konstruisanje objekta sa inicijalizacijom ili bez nje. Za specijalne potrebe, poput posredovanja pri prenosu objekata kao argumenata (tačka 8) ili kreiranja implicitnih objekata (tačka 3), postoje posebni konstruktori koji, pored osnovne namene, obavljaju i te, specifične, zadatke. Jedan od takvih konstruktora služi za prenos objekta kao argumenta i ima posebno ime – konstruktor kopije (engl. "copy constructor"). Drugi takav konstruktor jeste tzv. podrazumevani konstruktor koji ima dodatnu mogućnost implicitnog pozivanja. Pre nego što razmotrimo konstruktore sa specijalnim osobinama, nabrojaćemo karakteristike koje imaju svi konstruktori:

- Konstruktori su metode, ali metode nivoa klase. Može ih biti više u istoj klasi.
- Po imenu se poklapaju sa klasom. Dakle, svi konstruktori u klasi *ExampleClass* nose isto ime: *ExampleClass*.
- Obično se kaže da konstruktori nemaju nikakav tip, čak ni void. Ovo, međutim, nije dovoljno precizno, jer bi navelo na pomisao da konstruktori ne vraćaju nikakav rezultat. Stvarno stanje nije takvo: svaki konstruktor *vraća rezultat* a to je upravo konstruisani objekat! Pošto je to uvek objekat matične klase, istog naziva kao i konstruktor, kratkoće radi tip se ne piše

nego se podrazumeva. Znači, konstruktor sa nazivom *ExampleClass* kao rezultat primene vraća objekat klase *ExampleClass*.

- Mogu imati proizvoljan broj parametara (0 ili više). Parametri mogu biti bilo kojeg tipa osim same klase. Dakle, u klasi *ExampleClass* parametar konstruktora ne može biti objekat iz te klase. Pokazivač ili referenca na samu klasu može biti parametar. U pomenutoj klasi, prema tome, *ExampleClass** ili *ExampleClass&* mogu da budu tipovi parametara konstruktora.
- Ako konstruktora ima više, moraju se razlikovati po broju ili tipu parametara. Minimalna razlika je tip jednog parametra.
- Konstruktor objekta i konstruktor kopije uvek postoje čak i ako se ne navedu, u kojem slučaju ih generiše prevodilac.

Pošto će konstruktor kopije biti obrađen u narednom odeljku, zajedno sa destruktorom koji mu je srodan, ovde ćemo razmotriti ostale podvrste konstruktora. Još jednom, ***konstruktor (objekta)*** je metoda koja služi za kreiranje instance klase uz inicijalizaciju ili bez nje. Može se pozvati eksplicitno ili automatski, pri čemu se automatsko aktiviranje vrši npr. prilikom kreiranja implicitnih objekata. Mogućnost automatskog aktiviranja konstruktora s jedne i postojanja više konstruktora s druge strane dovodi do pitanja koji se od konstruktora aktivira kada se to izvodi automatski? Odgovor je: pri automatskom aktiviranju poziva se jedan od specijalnih konstruktora, tzv. podrazumevani konstruktor. ***Podrazumevani konstruktor*** je, po definiciji, konstruktor koji nema parametre. Može da bude *ugrađen* ili *eksplicitno definisan* od strane programera. U prvom slučaju radi se o konstruktoru bez parametara koji nije zadat u definiciji klase, dok se drugi slučaj odnosi na konstruktor bez parametara koji programer piše i uključuje u klasu. U vezi sa podrazumevanim konstruktorima važi pravilo da ako klasa ima bar jedan eksplicitno definisan konstruktor, ugrađeni podrazumevani konstruktor više ne može da se aktivira! To znači da ako, pored drugih, programer želi i podrazumevani konstruktor, mora ga uključiti u klasu kao eksplicitni konstruktor bez parametara. Sve do sada prikazane klase (*Point*, *Line*, *Complex* i *Semaphore*) koriste ugrađene konstruktore, pošto na spisku eksplicitno zadatih metoda nema konstruktora. Kao i sve druge metode, konstruktor se može realizovati kao *inline* metoda.

Konstruktori u C^{++} najčešće preuzimaju i zadatak *inicijalizacije* objekta. Ne zaboravimo, objekat koji nije inicijalizovan nema definisano stanje te je, prema tome, neupotrebljiv. Ugrađeni konstruktor ne vrši inicijalizaciju tako da, želimo li da se objekat inicijalizuje i pri automatskoj konstrukciji, moramo u klasu eksplicitno uključiti podrazumevani konstruktor snabdeven naredbama za inicijalizaciju. Inače, dosta ustaljena praksa je

- u definiciju klase uključiti eksplicitno i podrazumevani konstruktor.

Eksplicitno zadat podrazumevani konstruktor bez parametara i sa praznim telom ima dejstvo identično dejstvu ugrađenog konstruktora. Razlika je u tome što takav

konstruktor ne biva, kao ugrađeni konstruktor, "prekriven" drugim konstruktorima, nego koegzistira sa njima. Rezimirajmo: neka je data klasa *ExampleClass*. Mogući su sledeći slučajevi:

1. Klasa nema eksplicitno zadatih konstruktora (slučaj svih do sada navedenih klasa). Tada se koristi ugrađeni podrazumevani konstruktor sa imenom *ExampleClass* i bez parametara.
2. Klasa ima bar jedan konstruktor. Ugrađeni podrazumevani konstruktor iz prethodne tačke više se ne može aktivirati. Ako programer želi podrazumevani konstruktor, mora ga sam dodati u klasu po imenom *ExampleClass*, bez parametara. Ukoliko taj konstruktor ima oblik *ExampleClass() {}* on se po dejstvu poklapa sa ugrađenim podrazumevanim konstruktorom. Ako pak ima oblik *ExampleClass() {... naredbe ...}* tada i dalje ima ulogu podrazumevanog konstruktora, ali se od ugrađenog razlikuje po tome što obavlja dodatne aktivnosti zadate naredbama u telu (to je obično inicijalizacija).

Realizaciju i primenu raznih konstruktora demonstriraćemo na primeru klase *Complex* iz prethodnog poglavlja. Rekli smo da, onako kako je realizovana, ova klasa raspolaže samo ugrađenim podrazumevanim konstruktorom. Kada napišemo

```
Complex z;
```

u stvari smo *aktivirali ugrađeni podrazumevani konstruktor* u čemu je, podvucimo još jednom, bitna razlika između konstrukcije objekta i definicije promenljive.

Ilustracije radi, dopunićemo klasu *Complex* sa nekoliko različitih konstruktora, tako da njena definicija dobije sledeći oblik[41]:

```
class Complex {
private:
  double r, i;
public:
  Complex() {}   // Podrazumevani konstruktor zadat eksplicitno
  Complex(double rl) {r= rl; i= 0;}
  Complex(double rl, double img) {r= rl; i= img;}
  Complex(Complex z) {r= z.r; i= z.i;}  /* Greska! Parametar z ne sme biti tipa
                                           Complex */
  Complex(Complex *);
  Complex(Complex &);
. . . . . .
};
```

[41] Konvencija, koje se pridržavaju praktično svi programeri, jeste da se konstruktori navode kao prvi među svim metodama.

```
Complex::Complex(Complex *pZ) {
  r= pZ→r; i= pZ→i;   // Zapaziti primenu operatora "→". z->r je isto sto i (*z).r
}
Complex::Complex(Complex &z) {
  r= z.r; i= z.i;
}
```

Pošto se svi navedeni konstruktori međusobno razlikuju po broju ili tipu parametara nema prepreka da koegzistiraju u klasi. Nekoliko primera konstruisanja objekata klase *Complex*:

```
Complex a;   // Aktivira se Complex(); a nije inicijalizovan
Complex b(1);   // Aktivira se Complex(double); b = (1,0)
Complex c(1.5,-0.8);   // Aktivira se Complex(double, double); c = (1.5,-0.8)
Complex d(&a), e(c);   // Aktivira se Complex(Complex *); d = a ...
                       // ... a zatim Complex(Complex &); e = c
```

Lako se uočava neuobičajen način za aktiviranje konstruktora, karakterističan po tom što se naziv metode (Complex) razdvaja od parametara nazivom objekta. Takođe, kada nema parametara ne piše se ništa, za razliku od drugih metoda bez parametara gde se navodi (). Ta razlika, međutim, samo je sintaksna i nije od značaja za tretman konstruktora kao metode. Inače, uvedena je iz dva razloga: prvo da bi podsećala na srodnu operaciju definisanja promenljive i drugo da bi se jednom naredbom moglo izvršiti višekratno pozivanje konstruktora.

Takođe, na primeru se može videti zašto nije dozvoljeno postojanje konstruktora sa parametrom tipa matične klase. Povod je taj što se ne bi mogao razlikovati poziv takvog konstruktora i konstruktora sa parametrom tipa reference (poslednji navedeni poziv u primeru *Complex e(c)* odnosio bi se na obe varijante).

Inicijalizacija objekta može se izvršiti i drukčije, pomoću operatora "=", slično inicijalizaciji promenljivih. Izvesna razlika ipak postoji i ogleda se u dva pravila:

1. Inicijalizacija za klase sa eksplicitno zadatim konstruktorom može biti samo oblika

   A a = izraz; (a je objekat klase A),

   gde u klasi mora postojati konstruktor koji prima *izraz* za argument, uključujući i mogućnost da *izraz* bude već konstruisan objekat iste klase.
2. Za niz objekata koristi se konstrukt {izraz$_1$, ..., izraz$_n$}.

Primeri:

```
Complex f = 3.5;   // Aktivira se Complex(double); f = (3.5,0)
```

```
Complex g = f;      // Aktivira se Complex(Complex &); g = f
Complex x[] = {1.1, -0.8, 6.6};   /*  Konstrukcija niza od tri objekta.
                                      Aktivira se Complex(double)   */
```

Konstruktor se može aktivirati i eksplicitnim pozivom oblika

$$\text{NazivKonstruktora}(p_1, ..., p_n)$$

gde su $p_1, ..., p_n$ argumenti. Objekat koji se generiše kao rezultat eksplicitnog poziva konstruktora nosi naziv *bezimeni objekat*. Još jednom, uočimo

- da se u slučaju eksplicitnog poziva, konstruktor ponaša kao slobodna funkcija koja za rezultat vraća instancu klase.

Dajemo i za ovo nekoliko primera:

```
Complex p, q, r;
Complex nizc[] = {Complex(1.6), -0.5, Complex(1.1,4.8)};
/*  nizc je troelementni niz sastava (1.6,0), (-0.5,0), (1.1,4.8)  */
. . . . . .
p= Complex(1, 2);   // Complex(1,2) je bezimeni objekat sa r=1, i=2
q= Complex(5);   // Complex(5) je bezimeni objekat sa r=5, i=0
r= add(Complex(2.3,4.5),Complex(-0.6));   // r je (1.7,4.5)
```

Za potrebe konstrukcije sa inicijalizacijom C^{++} predviđa još jedan oblik konstruktora sa nazivom ***konstruktor-inicijalizator***. Koristi se isključivo tada kada se, uporedo sa konstruisanjem, objekat i inicijalizuje. Konstruktor-inicijalizator ima posebnu sintaksnu formu, različitu od drugih konstruktora. Ova forma konstruktora ima segment za inicijalizaciju (tzv. inicijalizacioni blok) koji se upisuje neposredno iza liste parametara. Opšti oblik inicijalizacionog bloka je

$$: \alpha_1, ..., \alpha_n$$

gde svaki od simbola α_i označava sintaksnu jedinicu vida

* *nazivPodatkaČlana(vrednost)* ako se radi o podatku-članu ili
* *nazivObjektaČlana(parametri konstruktora)* za objekte-članove.

Inicijalizacija objekta-člana je, u stvari, poziv njegovog konstruktora, gde se po parametrima određuje koji će konstruktor biti aktiviran. Neka klasa *A* sadrži podatke-članove *x* i *y* tipa double i objekat-član *ob* klase *K* koja ima konstruktor *K(int)*. Tada bi konstruktor-inicijalizator u klasi *A* mogao da izgleda ovako:

A(double xx, double yy, int j): x(xx), y(yy), ob(j) {}

gde je *ob(j)*, u stvari, poziv konstruktora iz klase *K* sa parametrom *j*. Telo konstruktora-inicijalizatora nije obavezno prazno. Može da sadrži sve što sadrže tela drugih konstruktora. Na primer, gornji konstruktor može se realizovati tako što se deo inicijalizacije izvodi u inicijalizacionom bloku, a deo u telu konstruktora:

A(double xx, double yy, int j): ob(j) {x= xx; y= yy;}

5.1.2. Podrazumevane vrednosti parametara

Podrazumevane vrednosti parametara funkcija jesu još jedna od novina koje je uneo C^{++}, a koje imaju retroaktivno dejstvo i na procedurni deo. Dakle, ne odnose se isključivo na konstruktore nego na sve metode i na sve slobodne funkcije. Razmatramo ih u ovom poglavlju zato što se u objektnom programiarnju često koriste u sklopu konstruktora, a sa ciljem da se smanji broj različitih konstruktora. Podrazumevane vrednosti parametara su vrednosti koje uzimaju parametri ako pri pozivu nisu navedeni odgovarajući argumenti. Opšti oblik prototipa funkcije ili metode sa takvim parametrima je

tip nazivFunkcije($T_1\ x_1$, ..., $T_m\ x_m$, $P_1\ y_1=c_1$, ..., $P_n\ y_n=c_n$);

gde parametri $y_1, ..., y_n$ dobijaju podrazumevane vrednosti konstanata $c_1, ..., c_n$. Pri pozivu, prvonavedeni parametri $x_1, ..., x_m$ moraju biti zamenjeni argumentima jer nemaju podrazumevane vrednosti. Parametri $y_1, ..., y_n$ mogu pri pozivu biti zamenjeni argumentima ili, ako ovi nisu navedeni, dobiti podrazumevane vrednosti. Parametri y_i dobijaju vrednosti argumenata redom kojim su navedeni, a ostali dobijaju podrazumevane vrednosti. Posmatrajmo funkciju

double f(int a, double b, int i=0, int j=1);

Funkcija se može pozvati sa f(p,q,r,s) i tada će biti i=r, j=s; ako se pozove sa f(p,q,r) tada će biti j=1; konačno, ako se pozove sa f(p,q) biće i=0 i j=1. Zapazimo da kada se u listi formalnih parametara nađe jedan koji ima podrazumevanu vrednost, iz razumljivih razloga takvu vrednost moraju imati svi koji se fizički nalaze iza njega.

Za konstruktor klase *Complex*, upravo ova mogućnost stvara uslove za formulisanje jednostavnog i moćnog konstruktora koji ćemo koristiti u sledećem poglavlju. Konstruktor oblika

Complex(double rl=0, double img=0): r(rl), i(img) {} ... (5.1)

može se aktivirati na tri načina (prva tri konstruktora iz primera moraju se ukloniti):

- Bez ijednog parametra, kada se broj inicijalizuje na vrednost (0,0). Pri ovakvom pozivu, konstruktor se ponaša kao podrazumevani. Primer: Complex a;
- Sa jednim parametrom, npr. x, kada se broj inicijalizuje na vrednost (x,0). Primer: Complex b(1.5);
- Sa dva parametra, npr. x i y, kada se broj inicijalizuje na vrednost (x,y). Primer: Complex c(2.2, 3.3);

Dakle, naredba Complex a, b(5), c(1,3); kreiraće objekte a=(0,0), b=(5,0) i c=(1,3).

5.1.3. Konstantni objekti

Konstantni objekti su, po definiciji, objekti koji ne mogu da promene početno stanje, tj. imaju prirodu objekata-konstanti. Nemogućnost menjanja stanja podrazumeva, naravno, da moraju biti inicijalizovani već pri konstrukciji. Na konstantne objekte mogu biti primenjene *samo konstantne funkcije-članice*, što je uostalom i osnovna svrha postojanja tih funkcija. Konstantni objekti kreiraju se na isti način kao i imenovane konstante, tj. pomoću modifikatora pristupa *const*, s tim da primenjeni konstruktor mora izvršiti inicijalizaciju.

Primer konstantnog objekta u klasi *Complex* mogla bi da bude imaginarna jedinica koja se, pod imenom CIM, kreira naredbom

```
const Complex CIM(0,1);
```

Uzgred, ako se vratimo na primer 4.2.1 videćemo da ni jedna od funkcija-članica iz klase ne može da se primeni na CIM, prosto zato što među njima nema konstantnih. Da smo metode *re*, *im* ili *modArg* definisali kao konstantne to bi bilo izvodljivo, što nije od velike važnosti u datom primeru - mnogo je važnije to što konstantni objekat CIM sme da se nađe na mestu stvarnog parametra u funkcijama i metodama, što će se pokazati kao značajno kada u klasi *Complex* primenimo tzv. preklapanje operatora (sledeće poglavlje).

Primer 5.1. U svrhu primera, snabdećemo konstruktorima klase *Point* i *Line*. U prethodnim verzijama, za inicijalizaciju su bile korišćene obične metode, redom *setPoint* i *setLine*. Možemo ih zameniti konstruktorima, ali i zadržati u klasama sa ulogom modifikatora i koristiti za promene vrednosti podataka-članova. Prema tome, klase *Point* i *Line* dogradićemo uvrštavanjem konstruktora redom

```
Point(double xx, double yy): x(xx), y(yy) {}
Line(double x1, double y1, double x2, double y2): a(x1,y1), b(x2,y2) {}
```

čije je mesto u otvorenim (public) segmentima odgovarajućih klasa. Skrećemo

pažnju na primenu konstruktora *Point* za inicijalizaciju objekata-članova *a* i *b* u inicijalizacionom bloku konstruktora *Line*.

Uočimo, takođe, da uključivanjem ovih konstruktora u klase njihovi ugrađeni konstruktori prestaju da važe što znači da više ne možemo upotrebiti naredbe oblika npr.

```
Point pt; Line ln;
```

stoga što one predstavljaju poziv podrazumevanog konstruktora. Greška će biti signalizovana još tokom prevođenja. Da bi se izbegli eventualni problemi u slučajevima kada se implicitno (automatski) pozivaju podrazumevani konstruktori, preporučljivo je i njih uključiti u klase. U tu svrhu na raspolaganju su dva postupka:

1. Standardni postupak je da se u klasu eksplicitno uključi podrazumevani konstruktor, kao konstruktor bez parametara. U našem slučaju to znači da bi klase *Point* i *Line* sadržavale još po jedan konstruktor oblika redom Point() {} i Line() {}. Dodatna mogućnost je da se u podrazumevane konstruktore uključi inicijalizacija na neke standardne vrednosti.
2. Alternativni postupak bio bi upotreba podrazumevanih vrednosti parametara postojećih konstruktora. Ako prvobitne konstruktore modifikujemo u oblik

   ```
   Point(double xx=0, double yy=0): x(xx), y(yy) {}
   Line(double x1=0, double y1=0, double x2=0, double y2=0):
       A(x1,y1), B(x2,y2) {}
   ```

   oni se mogu aktivirati i bez parametara, te tako igrati ulogu podrazumevanih konstruktora.

Na kraju, ima smisla snabdeti klasu *Point* konstantnim objektom *P00* koji odgovara koordinatnom početku, jer se u domenu problema ta tačka upadljivo izdvaja od ostalih. Objekat formiramo naredbom

```
const Point P00(0,0);
```

koja bi se nalazila u zaglavlju modula sa klasom *Point*, zajedno sa naredbom *class*.

5.2. DESTRUKTORI I KONSTRUKTORI KOPIJE

Destrukcija objekta je operacija koja je približno inverzna operaciji kreiranja. Kažemo "približno" jer kreiranje obavezno podrazumeva zauzimanje memorije za objekat, dok postoje slučajevi kada destrukcija ne oslobađa zauzeti prostor. Operaciju izvode specijalne metode nazvane ***destruktorima***. Zajednička

karakteristika svih destruktora je

- da se posle destrukcije objekat *smatra* nedostupnim, tj. neupotrebljivim, nezavisno od toga da li je njegov memorijski prostor oslobođen ili nije.

Može se reći da, na neki način, destrukcija oduzima objektu identitet, jer čak i ako njegovo ime nastavlja da egzistira, ono je neupotrebljivo. Destrukciju ne treba izjednačavati sa operacijom terminiranja, tj. vraćanja objekta u neko neutralno stanje, stoga što potonja operacija ostavlja objekat u upotrebljivom stanju. Destrukcija obuhvata dve aktivnosti:

1. oduzimanje identiteta objektu
2. (eventualno) oslobađanje memorije dodeljene objektu.

Iako to nije tako očigledno kao kod konstruktora, i destruktori su metode *nivoa klase*. Utisak da nije tako jer se poruka "uništi se" šalje objektu - pogrešan je. Dovoljno je uočiti da po izvršenom destruisanju objekat ne menja stanje nego nestaje, a "nikakvo", "nepostojeće" stanje nije svojstveno objektu. Ono što se menja posle destruisanja objekta jeste broj trenutno postojećih objekata, a to je podatak tipičan za klasu!

Destruktori izvršavaju operaciju destrukcije bilo samostalno bilo u sadejstvu sa drugim rutinama (npr. pri destrukciji dinamičkih objekata). Inače, kao što se u nekim slučajevima umesto konstruktora zadovoljavamo uslugama inicijalizatora, tako i destruktore ponekad mogu da zamene terminatori.

5.2.1. Destruktori u C^{++}

Destrukcija objekata u C^{++} vrši se primenom posebne metode - destruktora koji ima više uloga:

1. neposredno vrši destrukciju, osim objekata koji se nalaze na steku i dinamičkih objekata[42]
2. učestvuje u destrukciji objekata na steku i dinamičkih objekata
3. vrši terminiranje.

Njegove opšte osobine su sledeće:

- Identifikuje se imenom klase ispred kojeg stoji znak ~ (tilda). Dakle, destruktor klase *ExampleClass* ima naziv *~ExampleClass* (interesantna stvar: ime destruktora u C^{++} nema standardnu osobinu identifikatora da se sastoji samo od slova, cifara i znaka "_").
- Destruktor nema parametara.
- Pri definisanju destruktora ne navodi se nikakav tip (jer je tip, u stvari, uvek *void*).
- Svaka klasa ima tačno jedan destruktor. Ukoliko se ne navede eksplicitno, koristi se ugrađeni destruktor koji generiše prevodilac. Ugrađeni destruktor, poput ugrađenog konstruktora, ima prazno telo.

[42] Kako se destruktor tačno ponaša zaiunteresovani čitalac naćiće u sledećem odeljku.

Napomenimo odmah da ne postoji prepreka da se u klasu, pored destruktora, uključe i obične metode za terminiranje posle čije primene objekat zadržava identitet. Primer takvog terminatora bila bi (obična) metoda za pražnjenje steka ili liste. Po izvršenju metode stek i lista su i dalje na raspolaganju[43].

Destruktori se mogu pozivati implicitno ili eksplicitno, a standardni način aktiviranja je implicitni poziv do kojeg dolazi *pri izlasku objekta iz dosega*. Eksplicitni poziv je izuzetak i ostvaruje se kao kod drugih metoda. Treba voditi samo računa o tome da znak ~ bude pravilno protumačen kao deo imena destruktora. Ako je *exampleMethod* metoda klase *ExampleClass*, a *exampleObject* objekat iz te klase, destruktor se eksplicitno poziva na sledeći način:

- this→~ExampleClass() za eksplicitni poziv destruktora unutar metode *exampleMethod* koja će destruisati objekat preko kojeg je pozvana
- exampleObject.~ExampleClass() za eksplicitnu destrukciju objekta *exampleObject* koji je lokalni za neku funkciju ili metodu.

Kako smo rekli, ugrađeni destruktor ima isto dejstvo kao eksplicitni destruktor sa praznim telom. Za pomenutu klasu, ugrađeni destruktor i eksplicitni destruktor oblika

```
~ExampleClass() {}
```

su funkcionalno ekvivalentni. Inače, u C++ je ustaljena praksa da se destruktor uključi u naredbu *class*, čak i ako je prazan te se ne razlikuje od ugrađenog. Prema tome, ako se pridržavamo pomenute konvencije, klasa *Complex* sadržala bi eksplicitno zadati destruktor

```
~Complex() {}
```

Osvrnimo se i na verovatno najvažnije pitanje vezano za destruktore: kada se destruktor *mora* programirati? Odgovor je jednostavan:

- Destruktor se mora realizovati za sve klase koje imaju dinamičke članove (pokazivače na podatke odn. pokazivače na objekte).

U takvim slučajevima, destruktor sadrži naredbe za dealokaciju svih dinamičkih članova, tj. igra i ulogu terminatora. Neophodnost realizacije takvog destruktora proističe iz činjenice da se destruktor praktično uvek poziva implicitno, izlaskom objekta iz dosega. U takvim slučajevima mora se osloboditi sva memorija koju objekat zauzima, uključujući i hip. Da bismo ilustrovali ovaj zahtev, pretpostavićemo da klasa *ExampleClass* ima dinamičkih članova, ali da koristi ugrađeni destruktor sa praznim telom. Kada objekat *exampleObject* koji pripada toj klasi izađe iz dosega, ugrađeni destruktor se automatski aktivira. Međutim, sve što će on ura-

[43] Ibid.

diti je oslobađanje memorije dodeljene objektu, uključujući samo pokazivače na dinamičke članove, dok će deo dinamičke memorije koji ti članovi zauzimaju ostati blokiran. To je razlog zbog kojeg klasa mora da se snabde destruktorom oblika

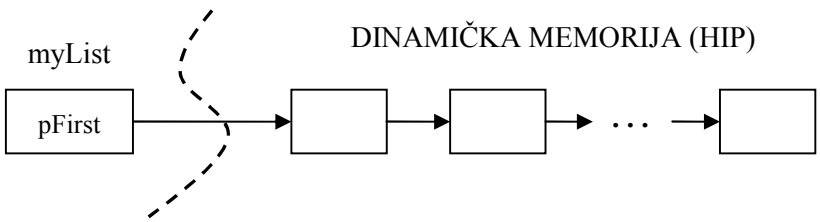
~ExampleClass() {... dealokacija dinamičkih članova ...}

tako da pri implicitnom pozivu destruktora bude oslobođena sva memorija dodeljena objektu, statička i dinamička.

Neka je *List* klasa koja predstavlja jednostruko spregnutu listu sa podatkom *data* nekog tipa *T*. Neka definicija klase bude ovakva:

```
class List {
private:
  struct Node {   // Uociti primenu uklopljene strukture Node
    T data;
    Node *pNext;
  };
  Node *pFirst;
public:
  List () {pFirst= 0;}
  ... ostale metode ...
};
```

Pretpostavka je da u naredbi *class* nema posebno realizovanog destruktora. Primetimo, takođe, da klasa ima jedan jedini podatak-član: pokazivač *pFirst* na prvi element liste. Sama lista u celosti se nalazi u dinamičkoj memoriji. Neka je *myList* instanca klase *List*. Njen opšti izgled prikazan je na slici 5.1.



Slika 5.1

U trenutku kada objekat *myList* izađe iz dosega, automatski se uključuje destruktor i to ugrađeni pošto klasa drugog nema. Međutim, ugrađeni destruktor funkcionalno je ekvivalentan destruktoru ~List () {}; sve što će on uraditi u procesu destrukcije jeste uništenje vrednosti *myList.pFirst* pokazivača na početak liste. Sama lista ostaće u dinamičkoj memoriji, ali bez mogućnosti pristupa jer je adresa prvog ele-

menta zagubljena. Radi se očigledno o klasičnom primeru curenja memorije. Da do ove neželjene pojave ne bi došlo prinuđeni smo da klasu snabdemo eksplicitnim destruktorom sa zadatkom da isprazni listu, tako da u trenutku uništavanja objekta njegov dinamički deo bude prazan. Takav destruktor imao bi globalni izgled

~List() {... pražnjenje liste ...}

Po izvršenju tela destruktora svi elementi liste iz dinamičke memorije uklonjeni su i važi pFirst=0, te je tako curenje memorije sprečeno.

Nažalost, uključivanje ovakvog destruktora u klasu (iznuđeno!) otvara na drugoj strani novi problem vezan za *prenos po vrednosti* liste kao stvarnog parametra slobodne funkcije ili metode. Neka je *f* slobodna funkcija tipa *FType* sa prototipom

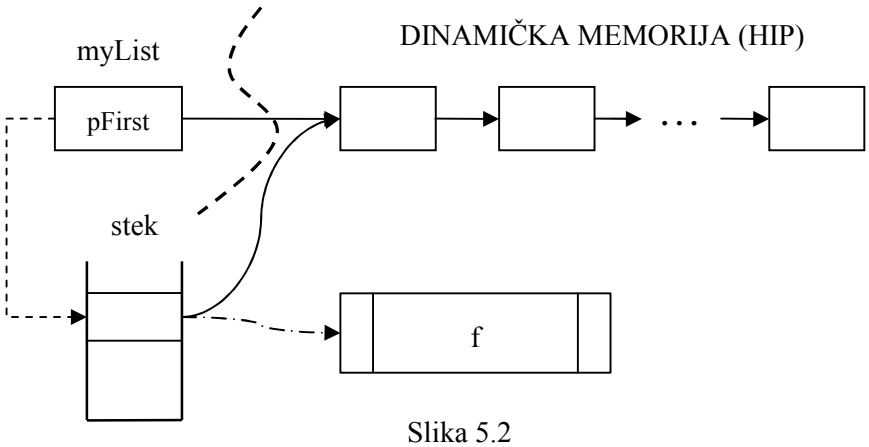
FType f(List list);

Prilikom poziva funkcije *f* sa stvarnim parametrom npr. *myList*

f(myList)

na steku će se naći kopija objekta *myList* koja se - ne zaboravimo - sastoji samo od kopije pokazivača *pFirst* jer je to jedini podatak-član u klasi (slika 5.2). Po završetku funkcije *f* biće izvršena automatska destrukcija te kopije jer se radi o parametru koji je izašao iz dosega. Destrukciju obavlja eksplicitni destruktor *~List* koji sadrži i pražnjenje liste. Upravo u tome i jeste problem: destruktor će uništiti kopiju ulaznog objekta na steku (pokazivača), ali će prethodno *izvršiti i operaciju pražnjenja liste na originalima u dinamičkoj memoriji*!

Da bi se predupredila katastrofa, u programski jezik morao je da bude uvršten mehanizam odbrane (dakle, opet iznuđeno rešenje). Radi se o ***konstruktorima kopije*** koje smo samo pomenuli u odeljku o konstruktorima. Naizgled neuobičajen raspored izlaganja, u kojem konstruktore kopije razmatramo u kontekstu destrukcije, uslovljen je baš time što su oni uvedeni kao zaštita od neželjenog dejstva destruktora, demonstriranog na primeru liste. Da bi odmah bilo jasno o čemu se radi, zadatak konstruktora kopije u datom primeru bio bi da napravi potpunu kopiju liste, a ne samo pokazivača na prvi element, tako da po završetku funkcije *f* destruktor ne uništava original nego kopiju liste. Vratićemo se na konstruktor kopije posle kratkog razmatranja dveju novouvedenih operacija u C^{++}, koje su potrebne za ilustrovanje njegovog funkcionisanja.



Slika 5.2

5.2.2. Operatori new i delete

Alokacija i dealokacija dinamičkih članova objekata pojavljuju se kao ključni problem u funkcionisanju destruktora. Stoga ćemo se, pre nego što detaljno objasnimo namenu i funkcionisanje konstruktora kopije, osvrnuti na jednu novinu u C^{++} koja je vezana za alokaciju i dealokaciju pokazivača. Naime, ove operacije su u programskom jeziku C realizovane preko standardnih funkcija *malloc*, *calloc*, *realloc* i *free*. Funkcije se ni po čemu ne razlikuju od drugih standardnih funkcija i, kao takve, nisu neposredno ugrađene u C nego se nalaze u njegovim bibliotekama. Imajući u vidu izuzetnu važnost pokazivača u C/C^{++}, kao i činjenicu da je C *operatorski jezik*, Straustrup je pomenute funkcije zamenio operatorima koji *jesu* deo programskog jezika C^{++} i ni po čemu se ne razlikuju od ostalih operatora.

Operator ***new*** koristi se za alokaciju pokazivača na osnovne tipove, nizove ili klase, sa inicijalizacijom ili bez nje. U svim slučajevima rezultat primene operatora *new* je pokazivač na zauzeti memorijski prostor. Za razliku od funkcija za alokaciju, *new* je tipizirani pokazivač (kod funkcija C-a tip je void*), tako da pri dodeli nije potrebna nikakva konverzija. U slučaju neuspešne dodele rezultat primene *new* je 0 (NULL)[44]. Tipični oblici su:

1. Oblik *new tip* gde *tip* predstavlja tip podatka na koji pokazuje pokazivač. Tip može biti osnovni ili programerski definisan tip. Sadržaj memorijskog prostora nije definisan.
2. Oblik *new tip(vrednost)*. U memoriju se upisuje je vrednost navedena u zagradama.
3. Oblik *new tip[dužina]*. Služi za odvajanje prostora za pokazivač na niz.

[44] Programeri u C^{++} radije upotrebljavaju vrednost 0 nego simboličku konstantu NULL.

4. Oblik *new poziv_konstruktora* za formiranje pokazivača na objekte. Poziv konstruktora može podrazumevati inicijalizaciju ako je konstruktor podešen za to.

Operator *new* može se primeniti na već definisan pokazivač, pomoću operatora dodele (kao i kod funkcija u C-u) ili u okviru definicije pokazivača, kao inicijalizacija. Dajemo nekoliko primera:

```
int *pJ, *pK;   // Definisanje pokazivača bez zauzimanja memorije
double *pX = new double;   // Definisanje pokazivača sa zauzimanjem memorije
double *pY = new double(1.5);  // Definisanje pokazivača sa inicijalizacijom
Complex *pZ, *pW(1.4,2.3)   // Definisanje pokazivača na objekte
int *pNiz[100]   // Definisanje niza pokazivača
. . . . . .
pJ= new int;   // Zauzimanje memorije za pJ bez inicijalizacije
*pJ= 10;
pK= new int(5);   // Zauzimanje memorije sa inicijalizacijom
pZ= new Complex(1,2);   // Zauzimanje memorije za objekat, sa inicijalizacijom
```

Operator ***delete*** ima samo dva oblika: *delete pokazivač* za sve pokazivače osim pokazivača na nizove i *delete[] pokazivač* za pokazivač na niz. Liberti [12] preporučuje da se po izvršenju operacije *delete* pokazivaču dodeli vrednost 0 jer nema garancije da će to biti urađeno automatski. Primeri:

```
delete pJ, pJ= 0;
delete pZ, pZ= 0;
delete[] pNiz; pNiz= 0;
```

Na kraju još jedna stvar, važna za objekte: pri izvršenju operacije *delete* nad pokazivačem na objekt, automatski se *izvršava destruktor*.

5.2.3. Konstruktori kopije

Konstruktori kopije posreduju pri prenosu *po vrednosti* objekata kao argumenata ili rezultata slobodne funkcije odn. metode. Osnovni zadatak im je da formiraju kopiju objekta - argumenta ili rezultata.

Konstruktor kopije uvek postoji, kao i konstruktor(i) objekta i destruktor. Dakle, ako se ne navede eksplicitno, pomenuti zadatak izrade kopije preuzima ugrađeni konstruktor kopije. U svakoj klasi postoji *tačno jedan* konstruktor kopije, ugrađeni ili eksplicitno zadat.

Već smo pomenuli da konstruktor kopije nije posebna vrsta metode, treća u nizu konstruktor objekta, destruktor, konstruktor kopije. Naprotiv, radi se samo o dodatnoj ulozi jednog od konstruktora objekta i to uvek onog koji za formalni pa-

rametar ima *referencu na objekat matične klase*. Prema tome, u klasi *ExampleClass* ulogu konstruktora kopije igra konstruktor

```
ExampleClass::ExampleClass(const ExampleClass &);
```

koji pritome ne gubi osobine konstruktora objekta. To će reći da ovaj konstruktor ima dvostruku namenu:

1. Ako se pozove eksplicitno ponaša se kao konstruktor objekta.
2. Ako se aktivira implicitno, u procesu prenosa objekta u funkciju ili iznošenja objekta kao rezultata, tada se ponaša kao konstruktor kopije.

Uočimo razliku u mehanizmu prenosa u funkciju ili iz nje promenljive s jedne i objekta s druge strane: u prvom slučaju parametar/rezultat smešta se na stek neposredno, dok prenos objekta zahteva posredstvo (tj. *izvršavanje*) konstruktora kopije.

Posmatrajmo opet klasu *Complex*. Iako u konkretnom slučaju klasa ne zahteva poseban konstruktor kopije (nema dinamičkih članova), uvrstićemo u nju, primera radi, konstruktor oblika

```
Complex(const Complex &z) {r= z.r; i= z.i;}
```

Pošto je parametar konstruktora kopije uvek čisto ulazni, praksa je da se on definiše kao konstantna referenca. Kao što smo rekli, ovaj konstruktor može se koristiti i kao konstruktor objekta i kao konstruktor kopije. Kao konstruktor objekta primenjuje se ovako:

```
Complex u(3,4);   // Konstruisanje objekta u pomocu konstruktora objekta
Complex w(u);   // Konstruisanje w=u pomocu gornjeg konstruktora
```

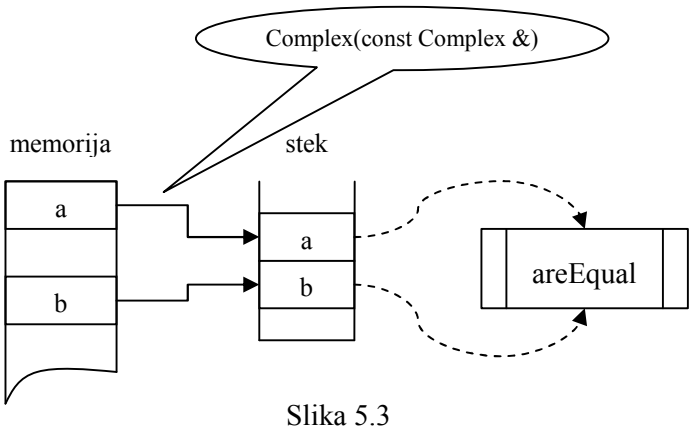
Razmotrimo sada drugu primenu ovog konstruktora - onu glavnu - a to je posredovanje pri prenosu parametara i rezultata. Pošto je to osnovna namena ovakvih konstruktora, uobičajeno je da ih uvek nazivamo konstruktorima kopije, čak i kada se primenjuju kao konstruktori objekta. Neka je potrebno napisati slobodnu funkciju *areEqual* za proveru da li su dva kompleksna broja jednaka ili nisu. Funkcija ima formu

```
int areEqual(Complex z1, Complex z2) {   // Uociti prenos z1 i z2 po vrednosti!
  return ( z1.re()==z2.re()) && (z1.im()==z2.im() );
}
```

gde se, kako vidimo, prenos z1 i z2 vrši po vrednosti. Ako se sada u nekoj naredbi pojavi poziv funkcije

... areEqual(a,b) ...

kopije vrednosti a i b na steku biće formirane izvršavanjem konstruktora kopije, eksplicitnog ili ugrađenog ako prvog nema. Šema prenosa prikazana je na slici 5.3.



Slika 5.3

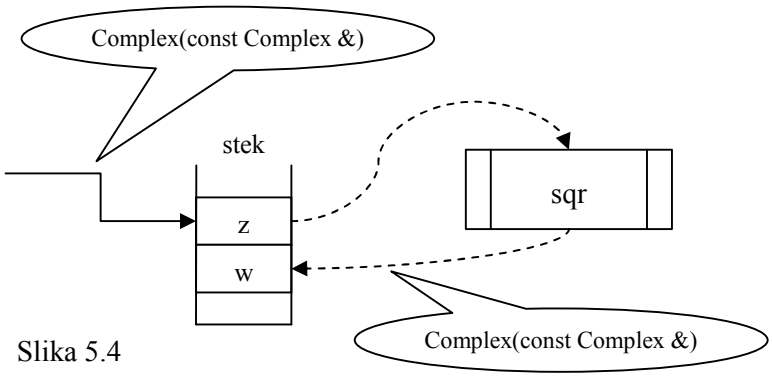
Isto se dešava i kada se rezultat neke funkcije/metode vraća po vrednosti. Neka je potrebno napraviti slobodnu funkciju *sqr(z)* za računanje kvadrata kompleksnog broja, gde je *z* objekat iz klase *Complex*. Jedno od rešenja je

```
Complex sqr(Complex z) {
  double zr=z.re(), zi=z.im();
  Complex w(zr*zr-zi*zi, 2*zr*zi);
  return w;
}
```

Pri aktiviranju funkcije *sqr* konstruktor kopije uključuje se dva puta: prvi put prilikom prenosa *z* u funkciju, a drugi put pri vraćanju rezultata *w*, kako je ilustrovano na slici 5.4.

Najvažnije pitanje vezano za konstruktor kopije u stvari je pitanje zašto on uopšte postoji? Zašto se prenos ne vrši automatski, kao kod promenljivih? Odgovor na ovo pitanje treba tražiti u načinu *prenosa pokazivača* kao argumenata, primenjenom još u programskom jeziku C. Naime, kada se kao parametar pojavi pokazivač, svako dejstvo nad dereferenciranim pokazivačem-parametrom rezultuje pristupom originalnom podatku, a ne njegovoj kopiji na steku (setimo se funkcije *scanf*!). Takav mehanizam prenosa po adresi, iako ne baš najelegantnije rešenje, nije pričinjavao naročite probleme u smislu nehotične promene vrednosti originala jer je pokazivač-parametar dosta upadljiv za vičnog programera. Situacija je nešto drukčija ako je parametar struktura (slog) koji sadrži pokazivač kao polje jer se on

ne vidi u prototipu funkcije. Ipak, ovakvi slučajevi dovoljno su retki da ne uzrokuju naročite poteškoće.



Slika 5.4

Sa objektima stvar stoji sasvim drukčije. Oni ne samo da mogu da imaju članove u dinamičkoj memoriji, nego to nije nikakva retkost. Pošto se u takvim slučajevima na steku nalaze pokazivači, tj. adrese originalnih podataka, itekako može doći do nehotične i neželjene promene vrednosti nekih od članova, čime se, takođe nehotično i neželjeno, *menja stanje* objekta, čak i kada se prenosi po vrednosti.

Iako potencijalno neugodan, problem je rešiv: sa objektima koji imaju dinamičke članove treba postupati oprezno. Postoji, međutim, još jedan problem koji se ne može rešiti pukim oprezom. Radi se o objektima sa dinamičkim članovima i njihovom odnosu sa destruktorom. Problem smo već opisali u okviru rasprave o destruktorima, a ilustrovali smo ga na primeru jednostruko spregnute liste sa slike 5.2, gde se vidi da destruktor uništava original liste kada se ona prenosi u funkciju po vrednosti. Ovde ćemo dati sasvim jednostavan ali programski precizno izveden primer koji pokazuje šta se dešava kada se objekat sa dinamičkim podatkom-članom prenosi u funkciju po vrednosti. Neka je *DynClass* klasa sa jednim jedinim podatkom-članom *pI* koji je pokazivač na podatak tipa *int*. Prvo, jasno je da klasa *mora* imati destruktor i to zbog dealokacije tog dinamičkog podatka-člana[45]. Definicija klase ima sledeći izgled:

```
class DynClass {
private:
  int *pI;
public:
  DynClass(int);   // Konstruktor (objekta)
~DynClass();   // Destruktor
```

[45] Nota bene: neophodnost uključivanja destruktora u klasu nema nikakve veze sa eventualnim prenosom objekta kao stvarnog parametra.

```
  int getI() const {return *pI;}
};
// Konstruktor koji alocira pokazivac pI i upisuje vrednost i na lokaciju
DynClass::DynClass(int i) {
  pI= new int(i);
}
// Destruktor koji dealocira pokazivač pI
DynClass::~DynClass() {
  delete pI, pI= 0;
}
```

Neka je potrebno realizovati jednostavnu slobodnu funkciju *s* sa parametrom klase *DynClass* i zadatkom da formira izlaz jednak petostrukoj vrednosti *pI. Funkcija ima sledeći izgled:

```
int s(DynClass objPar) {
  return 5*objPar.getI();
}
```

Naizgled, sve je u najboljem redu: funkcija *s* prima, posredstvom ugrađenog konstruktora kopije, kopiju *objPar*, računa traženu vrednost preko metode *getI* i vraća je klijentu. Međutim, ako u klijent uključimo samo dva uzastopna poziva *s*, recimo

```
DynClass dynObj(10);
int k1, k2;
. . . . . .
k1= s(dynObj);
k2= s(dynObj);
```

dogodiće se to da k1 dobije tačnu vrednost 50, a k2 neku sasvim proizvoljnu i to, nažalost, bez ikakve poruke o grešci. Razlog za ovu nepodopštinu je taj što se, posle završetka prvog poziva *s*, automatski uključuje destruktor koji preko *kopije* pokazivača pI sa steka uništava *originalnu* lokaciju *pI u dinamičkoj memoriji.

To, dakle, predstavlja glavni povod za uvođenje konstruktora kopije. Njegov posao u ovom slučaju je da napravi vernu kopiju objekta, uključujući i kopiju lokacije *pI. Novo rešenje izgleda ovako:

```
class DynClass {
private:
  int *pI;
public:
```

```
  DynClass(int);   // Konstruktor (objekta)
  DynClass(const DynClass &);   // Konstruktor kopije
~DynClass();   // Destruktor
  int getI() const {return *pI;}
};
// Konstruktor koji alocira pokazivac pI i upisuje vrednost "i" na lokaciju
DynClass::DynClass(int i) {
  pI= new int(i);
}
// Konstruktor kopije koji pravi kompletnu kopiju objekta
DynClass::DynClass(const DynClass &rhs)[46] {
  pI= new int;
  *pI= *(rhs.pI);
}
// Destruktor koji dealocira pokazivač pI
DynClass::~DynClass() {
  delete pI, pI= 0;
}
```

Sada više nema opasnosti od greške jer pri pozivu funkcije *s* konstruktor kopije pravi celovitu kopiju objekta. Po izlasku iz funkcije destruktor uništava tu kopiju, tako da original ostaje nepromenjen. Iz ovog primera jasno se vidi da je glavni razlog za uvođenje konstruktora kopije, u stvari, potencijalno *neželjeno dejstvo destruktora*.

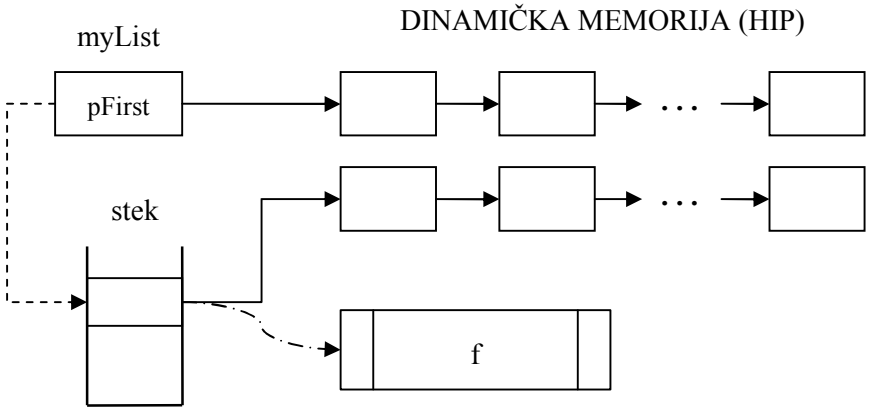
Vratimo se primeru jednostruko spregnute liste, da bismo i za njega prikazali opšte rešenje. Mehanizam koji će sprečiti da destruktor uništi originalnu listu i ovde je konstruktor kopije. Njega treba realizovati tako da izradi potpunu kopiju *cele liste*, a ne samo pokazivača na prvi element. Okvirni izgled takvog konstruktora kopije bio bi

```
List(const List &rlist) {... izrada kopije ulazne liste rlist ...}
```

Kopije koje sadrže repliku objekta sa svim njegovim članovima uključujući i delove u dinamičkoj memoriji nose naziv *duboke kopije*. Konstruktor kopije iz klase *DynClass* je dobar primer konstruktora koji pravi duboku kopiju stoga što je u kopiju uključena i lokacija *pI. S druge strane, pod *plitkim kopijama* podrazumevaju se kopije koje sadrže samo vrednosti članova iz memorijskog prostora objekta-originala (bez onih u dinamičkoj memoriji). Ugrađeni konstruktor kopije pravi isključivo plitku kopiju objekta. Poziv funkcije *f* sa slike 5.2 uz primenu konstruk-

[46] Parametar konstruktora kopije često nosi naziv *rhs* od engleskog "right hand side".

tora koji pravi duboku kopiju prikazan je na slici 5.5. Po završetku rada funkcije *f*, destruktor će uništiti kopiju liste, dok će original ostati nedirnut.



Slika 5.5

Na kraju, nekoliko reči i o tome kako se, u nekim slučajevima, može izbeći aktiviranje kako konstruktora kopije tako i destruktora. Pri prenosu po vrednosti malih objekata bez dinamičkih članova rad konstruktora kopije ne traje dugo niti kopije zauzimaju mnogo prostora na steku. U takvim slučajevima posredovanje pri prenosu prepušta se konstruktoru kopije. Međutim, ako su objekti veliki, vreme izrade kopije, kao i zauzeće memorijskog prostora mogu da pređu granicu tolerancije. Konstruktor kopije izbegava se usvajanjem reference na objekat (po potrebi konstantne) kao parametra, prosto zbog toga što se tada vrši prenos po adresi. Neka je *BigClass* klasa čiji objekti zauzimaju mnogo memorije i neka je *g* funkcija tipa *GType* za koju želimo da izbegnemo aktiviranje konstruktora kopije pri prenosu po vrednosti. Tada ćemo umesto prototipa

```
GType g(BigClass);
```

funkciju realizovati tako da ima prototip

```
GType g(BigClass &);   ili   GType g(const BigClass &);
```

Prenosom objekta putem reference istovremeno se *zaobilazi i destruktor*, tj. ne aktivira se pri završavanju rada funkcije, opet zato što je parametar adresa a ne objekat. Vratimo se primeru klase *DynClass* i funkciji za računanje petostruke vrednosti *pI. Ako tu funkciju realizujemo sa

```
int r(const DynClass &objPar) {
```

```
 return 5*objPar.getI();
}
```

bićemo u stanju da je pozivamo proizvoljan broj puta uz korektan rezultat, čak i ako u klasi nema posebnog konstruktora kopije. Razlog je to što se, pri napuštanju funkcije, destruktor ne uključuje.

Istim postupkom moguće je, ali ne uvek(!), zaobići konstruktor kopije i pri iznošenju rezultata. Ako se radi o nekoj funkciji *h* koja vraća kao rezultat objekat klase *BigClass* umesto prototipa

```
BigClass h(parametri);
```

funkciju ćemo realizovati tako da ima prototip

```
BigClass& h(parametri);
```

Ovde, međutim, treba biti oprezan. Naime, ako se rezultat formira u automatskom objektu, lokalnom za *h* (tzv. privremenom objektu), doćiće do greške. Da bismo razjasnili u čemu je problem navešćemo kostur funkcije h:

```
BigClass& h(parametri) {
 BigClass tempObj;
 . . . . . .
 return tempObj;
}
```

Ova funkcija neće raditi kako treba jer kao rezultat vraća referencu na lokalni objekat *tempObj* (tj. njegovu adresu), a on je u trenutku povratka iz *h* već izašao iz dosega! Dodajmo da je, nažalost, ovaj slučaj u praksi i najčešći.

5.2.4. Napomene u vezi sa destruktorom i konstruktorom kopije

Materijal koji će biti izložen u ovom odeljku treba da posluži isključivo za bolje razumevanje ponašanja destruktora i - naročito - konstruktora kopije. Nije neophodan za pravilno korišćenje ovih metoda. Štaviše, čitaocu se ne preporučuje da eksperimentiše činjenicama koje ćemo navesti, jer one same po sebi predstavljaju detalje realizacija i njihova zloupotreba znači direktno narušavanje principa skrivanja informacija.

Pođimo od destruktora. Podsetimo se: destruktor, *kao vrsta metode*, služi za uništavanje objekta. Međutim, sa destruktorom programskog jezika C^{++} stvar stoji nešto drukčije. Da bismo razjasnili ponašanje destruktora u C^{++} razmotrićemo, za početak, kako uopšte funkcioniše destrukcija. Objekat koji podleže destrukciji

može se nalaziti na tri mesta u memoriji:

- *Na steku*. Ali, tada destruktor samo učestvuje u uništavanju, jer se uništavanje objekta vrši pomeranjem stek-pointera.
- *Na hipu*. Ali, tada uništavanje vrši operacija *delete*, a destruktor opet samo učestvuje u tome. Drugim rečima operacija *delete* je destruktor, a ne metoda koja nosi to ime.
- *U statičkoj memoriji*.

Kako vidimo, u prva dva slučaja metoda C^{++} koja nosi naziv destruktor ne uništava objekat, iako u toj operaciji igra važnu ulogu. Najinteresantniji slučaj je treći, kada se objekat nalazi u statičkoj memoriji programa. No, tada se on ne može ni na koji način uništiti, baš kao ni ostale statičke promenljive. Sve što bi se moglo očekivati jeste da objekat postane nedostupan, tj. neupotrebljiv. Međutim, *ni to* nije moguće jer bi se tada pojavili ozbiljni problemi sa dosegom objekta. Posmatrajmo primer objekta *obK* neke klase *K*:

```
K obK;
obK.~K();  //eksplicitni poziv destruktora
obK. metoda();  //ovo ne bi trebalo da radi, ali radi!
```

Zašto radi poziv *obK.metoda()* kada bi objekat prethodnom naredbom trebalo da bude uništen ili bar nedostupan? Odgovor nije sasvim trivijalan, jer se radi o jednom od fundamentalnih pojmova u programiranju - dosegu (opsegu, oblasti važenja) objekta. Ako bi posle eksplicitnog poziva destruktora objekat trebalo da postane neupotrebljiv, to bi značilo da se doseg objekta posle eksplicitne primene destruktora u toku izvršenja programa menja i na taj način od statičke postaje dinamička kategorija. Još je očigledniji sledeći primer:

```
K obK;
if(p) obK.~K();  //opseg cas jeste cas nije skracen!
```

gde bi doseg objekta *obK* zavisio od vrednosti koju predikat *p* ima u trenutku izvršavanja naredbe *if*. To bi, konačno, značilo da bi se pre primene svake metode, *u toku izvršenja programa*, moralo proveriti da li je objekat preko kojeg se metoda aktivira trenutno dostupan ili nije. Za familiju jezika kojoj pripada C^{++} tako nešto je neoprostivo traćenje vremena (ne zaboravimo da se čak ni proboj opsega indeksa u C ne proverava). Očigledno, najbolje rešenje jeste da doseg ostane statčiki (dakle razrešava se u toku prevođenja) i da posle eksplicitne primene destruktora objekat ostane dostupan. Za slučaj izlaska iz opsega oblika

```
{
 K obK;
```

```
  . . .
}
obK.metoda();  //ovo ne moze!
```

poziv metode preko objekta koji je izašao iz opsega sprečava *prevodilac* koji ne dopušta da se takav program uopšte prevede. Iz ove kratke analize slede tri zaključka:

1. Pri eksplicitnom pozivu, destruktor u C^{++} ponaša se, ustvari, kao terminator!
2. Ako repertoar klase predviđa terminiranje, bolje je za tu svrhu napraviti običnu metodu-terminator i koristiti je umesto eksplicitnog poziva destruktora.
3. Destruktor koristiti implicitno, tj. prepustiti prevodiocu da obezbedi njegovo uključenje prilikom izlaska objekta iz opsega.

Rećićemo sada nešto i o konstruktoru kopije. U prethodnim izlaganjima naveli smo da se konstruktor kopije uključuje svaki put kada se vrši prenos objekta po vrednosti, kao i vraćanje objekta-rezultata funkcije. U novijim verzijama C^{++}

- više nema garancije da će se konstruktor kopije uključiti prilikom formiranja rezultata funkcije u slučajevima kada je rezultat objekat.

Razlog je *optimizacija*. Naime, kada kompajler "zna" koji lokalni objekat jeste jedinstveni rezultat funkcije neće ga formirati konstruktorom kopije nego će njegovu adresu sa steka proslediti kao rezultat. Ova mogućnost otvorena je redosledom izvođenja destrukcije. Lokalni objekat koji je jedinstveni rezultat funkcije prvo se putem adrese sa steka prosledi klijentu, pa se tek kada ga klijent preuzme vrši destrukcija. Posmatrajmo primer neke funkcije koja vraća objekat klase *K*:

```
K f(parametri) {
  K result;
  // formiranje lokalnog objekta result
  return result;
}
```

Neka smo funkciju pozvali sa

```
obK= f(argumenti);    //doseg lokalne promenljive result je ova naredba
```

Lanac aktivnosti je: 1) *result* se prenosi iz funkcije i dodeljuje obK 2) result se uklanja sa steka (uz eventualni poziv destruktora). Ovo, dalje, znači da je efektivni opseg lokalne promenljive *result* ne blok funkcije nego naredba poziva!

Navedeno, međutim ne znači da se ova činjenica sme koristiti. Naime, ako

kompajler ne zna koji objekat je reezultat on će uključiti konstruktor kopije! U funkciji

```
K g(parametri) {
  K res1, res2;
  //formiranje res1 i res 2
  if(...) return res1;
  return res2;
}
```

pri pozivu funkcije g konstruktor kopije će se ipak uključiti jer se ne zna unapred da li je rezultat *res1* ili *res2*!

Primer: klasa *Integer* predstavlja običan *int* broj predstavljen kao objekat. Metode i slobodne funkcije su samoobjašnjavajuće.

```
class Integer {
private:
  int dat;
public:
  Integer(int);
  Integer(const Integer&);
  void set(int);
  int get() const;
};
Integer::Integer(int i=0) {
  dat= i;
}
Integer::Integer(const Integer& rhs) {
  dat= rhs.dat;
}
void Integer::set(int i) {
  dat= i;
}
int Integer::get() const {
  return dat;
}

/*
  Pri formiranju rezultata, konstruktor kopije NECE SE UKLJUCITI.
  Posto je rezultat jedan jedini lokalni objekat, njega ce klijent
```

```
  preuzeti, pa ce se tek onda destruisati!
  Razlog: OPTIMIZACIJA
*/
Integer zbir(const Integer& i1, const Integer& i2) {
  Integer result;
  result.set(i1.get()+i2.get());
  return result;
}

/*
  Konstruktor kopije CE SE UKLJUCITI zato sto se objekat koji je rezultat
  ne zna unapred. U izlazni bafer upisace se ili i1 ili i2, a to mora da
  odradi konstruktor kopije!
*/
Integer max(const Integer& i1, const Integer& i2) {
  return (i1.get()>i2.get()) ? i1 : i2;
}
```

Zaključak koji izvlačimo iz dela izlaganja o konstruktoru kopije jednostavan je:

- činjenica da se konstruktor kopije uključuje po potrebi ne sme da utiče na promenu načina programiranja. Pošto se ne zna da li će konstruktor kopije biti uključivan ili neće, programer se i dalje ponaša kao dosad. Jedina izmena sastoji se u tome što konstruktor kopije ne sme da menja svoj parametar[47], što i jeste razlog za uvođenje obaveze da parametar konstruktora kopije bude *const*[48]. Inače, starije verzije kompajlera uključuju konstruktor kopije svaki put kada se formira rezultat.

5.3. PRIMER

U svojstvu primera sačinićemo modul sa klasom *List* koja predstavlja jednostruko spregnutu listu čiji su elementi generički pokazivači. Pre nego što prikažemo konkretno rešenje osvrnimo se na opšte osobine jednostruko spregnute liste kao strukture podataka. Jednostruko spregnuta lista (u daljem tekstu kratko: lista) je najfleksibilnija linearna struktura podataka i kao takva veoma široko primenjivana u programerskoj praksi. Lista je naizgled veoma jednostavna struktura: linearna je, dozvoljen je pristup svakom elementu kao i njegovo uklanjanje, a novi element može se dodati na svakom mestu. Lako se realizuje sprezanjem elemenata pomoću pokazivača. Ova jednostavnost je, međutim, samo prividna. Naime, ako se usre-

[47] I dosad je promena argumenta unutar konstruktora kopije bila vrlo egzotična i nepreporučljiva praksa!
[48] Na grešku će, inače, ukazati prevodilac.

dotočimo na primenu liste, ustanovićemo da je upotreba vrlo divergentna što bitno utiče na konkretnu realizaciju. Navedimo nekoliko zahteva koji se mogu postaviti pred listu, a koji presudno utiču na njenu realizaciju:

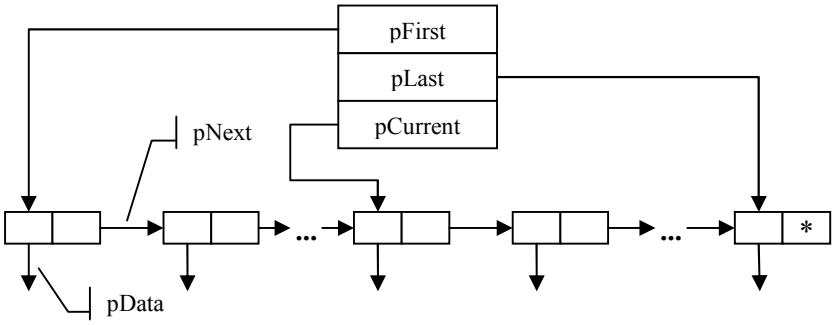
- Redosled elemenata u listi je proizvoljan. U tom slučaju zaglavlje liste (tzv. *deskriptor*) sadrži samo pokazivač na prvi element, a novi element se dodaje isključivo ispred prvog.
- Lista mora da čuva hronološki redosled elemenata. Sada deskriptor, pored pokazivača na prvi, mora sadržati i pokazivač na poslednji element. Dodavanje se vrši isključivo iza poslednjeg elementa.
- Lista je sortirana po nekom ključu. Operacija dodavanja mora biti izvedena tako da se prethodno ustanovi mesto na koje će biti upisan novi element. Mesto se određuje na osnovu vrednosti ključa novog elementa.
- Lista se intenzivno obrađuje redoslednim postupkom. Tada je treba snabdeti mehanizmom tzv. *navigacije*. Taj mehanizam uključuje postojanje tekućeg elementa koji je poznat u svakom trenutku, a čija se adresa (pokazivač) nalazi u deskriptoru. Operacija pristupa vrši se nad tekućim elementom, a operacije dodavanja i uklanjanja elementa zahtevaju promenu pokazivača na tekući element.
- Lista zahteva pristup po logičkoj poziciji. Sve operacije - čitanje, dodavanje, uklanjanje i modifikacija - imaju za argument i (logički) redni broj elementa. Pristup elementu podrazumeva pristup njegovim logičkim prethodnicima.

Nekoliko napomena: prvo, ovaj spisak ne sadrži sve moguće zahteve; drugo, oni se ne moraju međusobno isključivati; treće, lista se može realizovati tako da ispuni sve moguće zahteve, ali su tada operacije po pravilu spore. Jednom rečju, pre realizacije listu treba *projektovati* u skladu sa konkretnim uslovima upotrebe. U ovom primeru, pretpostavićemo da ambijent u kojem će se klasa *List* upotrebljavati podrazumeva sledeće:

1. Očekuje se često dodavanje elemenata na kraju liste. Zbog toga se u deskriptoru liste nalazi i pokazivač na njen poslednji element.
2. Očekuje se navigacija kroz listu. Zato postoji tekući element.
3. Ne očekuje se česta provera aktuelnog broja elemenata u listi (dužine liste). Stoga se broj elemenata ne nalazi u deskriptoru (njegovo ažuriranje usporavalo bi ostale operacije).

Struktura liste prikazana je na slici 5.6.

Elementi liste sastoje se od dva pokazivača: prvi je generički pokazivač *pData* koji predstavlja informacioni sadržaj elementa, a drugi je pokazivač *pNext* na sledeći element u listi. Sa slike vidimo da sam objekat klase *List* nije ništa drugo do zaglavlje (deskriptor) liste sa tri pokazivača na element liste: *pFirst* (pokazivač na prvi), *pLast* (pokazivač na poslednji) i *pCurrent* (pokazivač na tekući element).



Slika 5.6

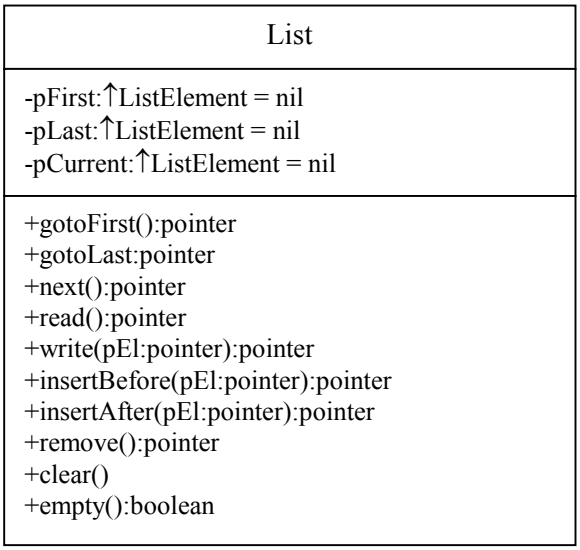
Na slici 5.7 prikazan je simbol klase *List*. Prvo što na slici primećujemo je to da nisu navedeni konstruktori i destruktor. Razlog za to je sadržan u jednom nedostatku UML. Naime, simbol klase treba da sadrži model odgovarajućeg tipa entiteta, nezavisan od buduće realizacije. To, međutim, nije uvek moguće sprovesti do kraja, a ovde se radi upravo o takvom slučaju. Naime, C^{++} *zahteva* da se konstruktori i destruktor po imenu poklapaju za klasom, dok paskal to izričito *zabranjuje*. U datoj situaciji, najbolje je konstruktore i destruktor dati tesktuelnim opisom, jer i inače programeri dobro znaju da oni moraju postojati (bar u C^{++}), te ih ne treba posebno podsećati na to.

Dajemo prvo sasvim kratak opis metoda:

1. *gotoFirst* pomera marker tekućeg elementa (tj. pokazivač *pCurrent*) na poziciju prvog elementa
2. *gotoLast* radi to isto, ali za poslednji element
3. *next* pomera marker tekućeg na sledeći element
4. *read* čita sadržaj *pData* tekućeg elementa
5. *write* upisuje novi sadržaj *pData* u tekući element
6. *insertBefore* dodaje novi element liste ispred tekućeg
7. *insertAfter* dodaje novi element iza tekućeg
8. *remove* uklanja tekući element
9. *clear* prazni listu
10. *empty* proverava da li je lista prazna.

Ako stanje liste definišemo kao stanje svih njenih elemenata, metode možemo razvrstati u sledeće kategorije:

- gotoFirst, gotoLast i next su selektori
- write, insertBefore, insertAfter i remove su modifikatori
- clear je terminator
- empty i read su indikatori.



Slika 5.7

Sada ćemo navesti nešto detaljniju specifikaciju navedenih metoda.

1. Konstruktori i destruktor

- Konstruktor List() postavlja sve atribute na 0 (NULL)
- Konstruktor kopije List(const List&) pravi duboku kopiju
- Destruktor ~List() uništava listu

2. Selektori

- void* gotoFirst() postavlja tekući na prvi i vraća njegovu adresu (pokazivač). Ako je lista prazna vraća NULL.
- void* gotoLast() radi analogan posao za poslednji element liste.
- void* next() postavlja tekući na sledeći. Ako je tekući element jednak NULL vraća vrednost NULL, a ako nije vraća adresu novog tekućeg. Važno je napomenuti da kada je *pCurrent=pLast* ova metoda može da se primeni još jednom i tada postaje *pCurrent=NULL*.
- void* read() vraća sadržaj polja *pData* tekućeg elementa. Ostavlja tekući nepromenjenim. Ako je lista prazna vraća NULL.

3. Modifikatori

- void* write(void *pEl): ako tekući nije NULL upisuje pEl u njegovo polje *pData*. Vraća adresu tekućeg (što znači da ako je pCurrent prazan vraća NULL)

- void* insertBefore(void *pEl): dodaje novi element čije je polje *pData* jednako parametru *pEl* ispred tekućeg. Ako je lista prazna upisuje novi element kao prvi. Ako je tekući jednak NULL ne izvršava se i takođe vraća NULL. Ako tekući nije bio prazan posle dodavanja postavlja vrednost *pCurrent*. Vraća adresu tekućeg.
- void* insertAfter(void *pEl): sve je isto, samo dodaje iza tekućeg.
- void* remove() uklanja tekući. Ako je lista prazna ili je tekući NULL ne izvršava se i vraća vrednost NULL. Novi tekući element je onaj *ispred* uklonjenog tekućeg, ali ako je uklonjen prvi po redu novi tekući postaje njegov sledbenik (tj. bivši drugi). Vraća adresu novog tekućeg.

4. Terminator

- void clear() briše (prazni) listu.

5. Indikator

- int empty() vraća 1 ako je lista prazna, odnosno 0 ako nije.

Klasa *List* smeštena je u modul sa istim nazivom. Modul se sastoji iz dva dela : zaglavlja LIST.HPP i tela modula sa nazivom LIST.CPP. Modul ima sledeći izgled:

```
/*****************************************************************
**

        ZAGLAVLJE MODULA SA KLASOM LISTE POKAZIVACA


Naziv datoteke: LIST.HPP
*****************************************************************
*/

#ifndef LIST_HPP
 #define LIST_HPP

class List {
private:
 struct ListElement {
   void *pData;
   ListElement *pnext;
 };
 ListElement *pFirst, *pLast, *pCurrent;
public:
```

```
// Konstruktori i destruktor
  List() {pFirst= pLast= pCurrent= 0;}    // Konstruktor
  ~List();                          // Destruktor
  List(const List &);                   // Konstruktor kopije
// Selektori
  void* gotoFirst() {return pFirst ? (pCurrent=pFirst) : 0;}
  void* gotoLast() {return pLast ? (pCurrent=pLast) : 0;}
  void* next() {return pCurrent ? pCurrent=pCurrent->pnext : 0;}
  // Modifikatori
  void* write(void *);
  void* insertBefore(void *);
  void* insertAfter(void *);
  void* remove();
// Terminator
  void  clear();
// Indikatori
  void* read() const {return pCurrent ? pCurrent->pData : 0;}
  int  empty() const {return pFirst==0;}
};

#endif
```

```
/*****************************************************************
**

            TELO MODULA SA KLASOM LISTE POKAZIVACA

Naziv datoteke: LIST.CPP
*****************************************************************
*/
#include "list.hpp"

// Destruktor
List::~List() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pnext; delete temp;}
  pLast= pCurrent= 0;
}
// Konstruktor (duboke) kopije
```

```
List::List(const List &rList) {
  ListElement *orig;
  if(rList.empty()) {pFirst= pLast= pCurrent= 0;}   // Prazna lista
  else {
    orig= rList.pFirst;                          // Postaviti prvi
    pCurrent= pFirst= new ListElement;
    pCurrent->pData= orig->pData;
    while(orig= orig->pnext) {
      pCurrent= pCurrent->pnext= new ListElement;
      pCurrent->pData= orig->pData; }
    (pLast=pCurrent)->pnext= 0;
    pCurrent= rList.pCurrent;
  }
}
// Upis novog sadrzaja u tekuci element
// Ako je lista prazna vraca 0
void* List::write(void *pEl) {
  if(pCurrent) pCurrent->pData= pEl;
  return pCurrent;
}
// Dodavanje ispred tekuceg elementa
// Novi element postaje tekuci
void* List::insertBefore(void *pEl) {
  ListElement *newEl, *prev, *next;
  newEl= new ListElement;
  newEl->pData= pEl;
  if(empty()) {newEl->pnext= 0;          // Lista prazna
               pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else if(pCurrent == pFirst) {          // Dodavanje ispred prvog
    newEl->pnext= pFirst;
    pCurrent= pFirst= newEl; }
  else {                                 // Ostalo
    next= pFirst;
    while(next != pCurrent) {prev= next; next= next->pnext;}
    newEl->pnext= next;
    pCurrent= prev->pnext= newEl; }
  return pCurrent;
}
// Dodavanje iza tekuceg elementa
```

```
// Novi element postaje tekuci
void* List::insertAfter(void *pEl) {
  ListElement *newEl;
  newEl= new ListElement;
  newEl->pData= pEl;
  if(empty()) {newEl->pnext= 0;          // Lista prazna
               pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else {                                  // Ostalo
    newEl->pnext= pCurrent->pnext;
    pCurrent->pnext= newEl;
    if(pCurrent == pLast) pLast= newEl;   // Dodat iza poslednjeg
    pCurrent= newEl; }
  return pCurrent;
}
// Uklanjanje tekuceg
// Novi tekuci postaje prethodni osim ako je uklonjen prvi, a
// tada novi tekuci postaje sledeci
void* List:: remove() {
  ListElement *prev, *next;
  if (empty()) return 0;               // Lista je prazna
  else if(pCurrent == 0) return 0;     // Tekuci prazan
  else {
    prev= next= pFirst;
    while(next != pCurrent) {prev= next; next= next->pnext;}
    if(next != pFirst) {               // Nije prvi
      prev->pnext= next->pnext;
      pCurrent= prev;
      if(next == pLast) pLast= prev; }  // Poslednji
    else {                             // Jeste prvi
      pCurrent= pFirst= pFirst->pnext;
      if(next == pLast) pLast= 0; }     // Bio jedini
    delete next;
    return pCurrent;
  }
}
// Ciscenje liste
void List::clear() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pnext; delete temp;}
```

```
  pLast= pCurrent= 0;
}
```

6. UVOD U POLIMORFIZAM. PREKLAPANJE OPERATORA. KONVERZIJA.

Polimorfizam (multiformnost), kao opšti pojam, predstavlja sposobnost nekih entiteta da uzmu različite oblike, u zavisnosti od konkretnog ambijenta. Konstatovan je, na primer, u biologiji kao sposobnost pojedinih vrsta živih bića da se pojavljuju u više različitih obličja ili varijeteta boja, ili u kristalografiji kao mogućnost materije istog hemijskog sastava da kristalizuje na više načina. Recimo, u zavisnosti od uslova kristalizacije čist ugljenik u prirodi se pojavljuje kao dijamant ili kao grafit pri čemu te dve forme osim hemijskog sastava nemaju ničeg zajedničkog.

U programiranju, polimorfizam je prvi put uočen kod funkcija, kao mogućnost rada nad širim spektrom tipova (Strachey, [80]). Konkretnije, polimorfna funkcija za argumente prihvata promenljive ili izraze različitih (mada nikad proizvoljnih) tipova i podešava izlaz prema tipu, a ne samo prema vrednosti argumenata. Tu osobinu imaju, na primer, paskalske funkcije *pred* i *succ* (prethodnik i sledbenik), koje za argumente prihvataju vrednosti celobrojnog ili znakovnog tipa, pa i enumeraciju, a zatim u skladu sa tipom argumenta kao rezultat generišu sledeću u nizu vrednosti. Primerice, succ(1) je 2, dok succ('a') daje 'b', iako se radi o istoj funkciji! Uočimo da su navedena tri tipa slična po tome što se za svaku vrednost mogu definisati prethodna i sledeća, tako da se *pred* i *succ* ponašaju uniformno bez obzira na tip argumenta. Polimorfizam, inače, ne insistira na istovetnom ponašanju: Buč [3] ga posmatra u nešto širem kontekstu veze između identifikatora i tipa, što bi u slučaju funkcije značilo da i različite funkcije ili operatori sa argumentima različitog tipa mogu imati isto *ime*. Na primer, operator "+" u paskalu može označavati sabiranje ako su operandi celobrojni ili realni, uniju za skupove ili spajanje (konkatenaciju) za stringove. Treba znati da se ovde radi o drukčijoj vrsti polimorfizma jer se simbol "+" dodeljuje trima *različitim* operacijama, dok npr. "pred" označavaja *istu* funkciju koja prepoznaje tip argumenta. Programske kategorije koje ispoljavaju polimorfizam su:

- promenljive odnosno instance klase,
- operatori i
- potprogrami odnosno metode,

a u nekim slučajevima to mogu biti i konstante.

Imajući u vidu do sada rečeno možemo usvojiti radnu ***definiciju*** polimorfizma kao sposobnosti navedenih kategorija da ispolje *kontekstno zavisno ponašanje*. C-ov operator * u numeričkom kontekstu znači množenje, a u pokazivačkom dereferenciranje. Paskalske funkcije *pred* i *succ* u kontekstu celobrojnog tipa generišu kao rezultat broj za jedan manji odn. veći od argumenta, dok u kontekstu znakovnog tipa generišu prethodni odn. sledeći znak u standardnom nizu znakova. Paskalski operator + u celobrojno-realnom kontekstu znači sabiranje, u skupovnom uniju itd.

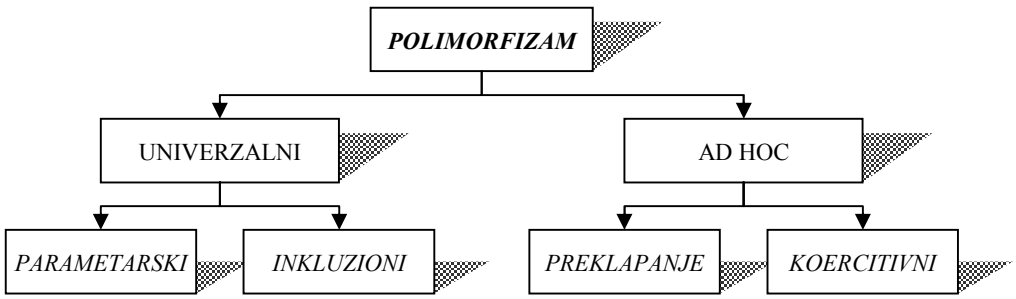
I sam polimorfizam pojavljuje se u različitim vidovima, što je verovatni uzrok nepostojanja jedinstvene opšte definicije. Dajemo nekoliko primera različitih vidova polimorfizma u programskom jeziku C/C^{++}. Uz primere navedeni su i nazivi tih vidova.

1. (Parametarski polimorfizam) Standardni tip strukture (sloga) u C/C^{++} sam po sebi predstavlja jednu vrstu polimorfizma stoga što tipovi pojedinih polja nisu zadati unapred nego ih programer definiše prilikom konkretizacije. Isto važi i za nizove čijim se elementima zadaje tip tek pri definisanju odgovarajuće promenljive. Potprogrami za ulaz-izlaz scanf i printf u C/C^{++} prihvataju, od slučaja do slučaja, različit broj argumenata.
2. (Inkluzioni polimorfizam) U C/C^{++} promenljiva tipa *char* može da se tretira kao znakovna ili celobrojna, u zavisnosti od konkretne naredbe, tako da se stiče (tačan) utisak da pripada i jednom i drugom tipu. U istom programskom jeziku 0 i 1 mogu se, po potrebi, tretirati i kao logičke konstante $\perp$ i T. Pod određenim, specifičnim, uslovima objekat *z* iz klase *K* može se pojaviti na mestu na kojem se očekuje objekat druge klase *L*.
3. (Preklapanje operatora) Isti simbol može označavati različite operacije: u C/C^{++} simbol & može biti referenca, adresni operator ili bit-konjunkcija; simbol * predstavlja množenje ili dereferenciranje pokazivača. Ovde treba podvući da su navedeni simboli isti samo na nivou programskog jezika, dok su same operacije (tj. interni kodovi odgovarajućih rutina) različite.
4. (Koercitivni polimorfizam) U polimorfizam ubrajamo i implicitnu (automatsku) i eksplicitnu konverziju tipa, naročito izraženu u C/C^{++} kao slabo tipiziranom jeziku. Ako su, recimo, *i* i *r* respektivno celobrojna i realna promenljiva, u izrazu *i*+*r* promenljiva *i* ponašaće se kao realna promenljiva sa razlomljenim delom jednakim nuli. U drugim delovima programa ista promenljiva ponašaće se onako kako je definisana - kao celobrojna. Konverzija tipa može biti i eksplicitna: ako u C/C^{++} napišemo *(double) a* konstrukt će biti tipa *double* bez obzira na osnovni tip promenljive *a*. Ovaj mehanizam poznat je pod nazivom *type cast*.

Navedene 4 grupe primera koncipirane su tako da istovremeno ilustruju i ***klasifi-***

kaciju polimorfizama prema Kardeliju i Vegneru (Cardelli, Wegner, [81]). Prema njihovoj klasifikaciji, polimorfizam ima dva osnovna vida, svaki sa po dve podvrste (slika 6.1).

Opšta karakteristika ***univerzalnog polimorfizma*** je ta da se *ista kategorija* (promenljiva, objekat, potprogram, metoda) ponaša različito u zavisnosti od uslova u kojima se referencira. Primeri označeni sa 1 i 2 odnose se na univerzalni polimorfizam.



Slika 6.1

Primerom 1 obuhvaćen je tzv. ***parametarski polimorfizam***. Na ovu vrstu polimorfizma ukazano je još u vreme formulisanja opšte teorije tipova podataka, kada je zapaženo da neki tipovi podataka mogu da se definišu tako da funkcionalno zavise od drugih tipova podataka koji se, sa svoje strane, pojavljuju kao slobodne promenljive. Takvi tipovi podataka dobili su naziv *generički* ili *parametrizovani* tipovi podataka. Razmotrimo, u svojstvu jednostavnog primera, generički tip steka sa imenom *Stack*. Lako se uočava da realizacija osnovnih operacija *top*, *pop*, *push* i *pmpty* ne zavisi od tipa elemenata steka. Ako su elementi steka uopštenog tipa T, tada je signatura osnovnih operacija

top(s: Stack): T
pop(s:Stack): Stack
push(s:Stack, e:T): Stack
empty(s:Stack): Logical

Sve četiri operacije moguće je do kraja definisati, pa i realizovati ako to dopušta programski jezik, a da se ne zada konkretni tip elemenata *T*. Na sličnu situaciju nailazimo kod generičkog tipa skupa jer osnovne skupovne operacije i relacije (unija, presek, inkluzija ...) ne zavise od tipa elemenata skupa. Isto je i u slučaju svih vrsta lista, stabala itd. Ono što je zajedničko za sve generičke tipove jeste činjenica da tipovi-parametri bivaju zadati prilikom *konkretizacije*, tj. pri definisanju promenljivih. Dakle ako sa *Stack[T]* označimo generički tip steka tada se (tek) pri

definisanju odgovarajuće promenljive mora obezbediti podatak o tome šta je tačno tip *T*. Sličnost odnosa tipa-parametra i konkretnog tipa s jedne i parametra-argumenta kod potprograma sa druge strane, očigledna je i uopšte nije slučajna.

Programski jezici koji ne poseduju generičke tipove (klase) kakav je, na primer, paskal zahtevaju pisanje posebnog koda za svaki konkretan tip (dakle, realizaciju steka znakova, steka pokazivača, steka celih brojeva itd.). Na takvim jezicima, umesto generičkog tipa *Stack[T]* sa mehanizmom zadavanja *T* kao *char, pointer* ili *integer*, moraju se realizovati posebni tipovi steka, npr. *CharStack* (stek znakova), *PtrStack* (stek pokazivača), *IntStack* (celobrojni stek), pri čemu je najveći deo koda istovetan za sve. Na sreću, situacija nije baš sasvim deprimirajuća: paskal pruža mogućnost za simuliranje generičkih tipova, no uz obavezno dopisivanje koda i ne tako jednostavno kao kod pravih generičkih tipova (videti poglavlje o generičkim klasama).

Parametarski polimorfizam u objektnom ambijentu ni po čemu se ne razlikuje od polimorfizma generičkih tipova podataka, osim po tome što govorimo o generičkim klasama i klasama-parametrima koje dobijaju konkretne oblike prilikom instanciranja. Štaviše, a kako će biti prikazano u posebnom poglavlju, programski jezik C^{++} omogućuje realizaciju generičkih tipova podataka posredstvom generičkih klasa (tzv. "templejta").

Konačno, parametarski polimorfizam kod potprograma ispoljava se kao mogućnost prepoznavanja tipa argumen(a)ta i, saobrazno tome, generisanja različitih odziva. Po pravilu, spektar tipova-parametara kao i eventualnih izlaza ograničen je, stoga što nad parametrom proizvoljnog tipa ne mogu da se vrše skoro nikakve operacije jer se ne zna unapred kakvim repertoarom taj tip raspolaže.

Sledeća vrsta polimorfizma, ilustrovana primerima u tački 2, nosi naziv ***inkluzioni polimorfizam***. Originalna, Strachey-eva, klasifikacija nije pravila razliku između parametarskog i inkluzionog polimorfizma. Razdvojili su ih Kardeli i Vegner uočivši da se u određenim situacijama promenljive i posebno objekti ponašaju kao da pripadaju istovremeno većem broju tipova, kao što je to slučaj sa znakovnim promenljivim u C/C^{++}. Kažemo "kao da pripadaju" jer se promenljivoj ili objektu pri definisanju dodeljuje - ipak - jedan jedini tip (klasa) koji je nepromenljiv i kojeg Mejer [1] naziva statičkim tipom. Promenljiva ili objekat sa statičkim tipom (klasom) *K* u posebnim uslovima može se *ponašati* kao da je član nekog drugog tipa (klase) *K′* zadržavajući pri tom osnovni, statički tip. Mejer naziva *K′* dinamičkim tipom. Ovo je moguće samo uz uslov da *K* i *K′* nisu proizvoljni i međusobno nezavisni, nego su povezani nekom vrstom relacije inkluzije, što u ovom slučaju znači da *K′* na neki način obuhvata *K*. Poznato je da znakovni tip u C/C^{++} pripada familiji celobrojnih tipova i upravo je to uzrok inkluzionom polimorfizmu znakovnih promenljivih.

Inkluzioni polimorfizam od posebnog je značaja baš u objektnom programiranju, gde se realizuje posebnim mehanizmom zvanim "nasleđivanje" kojim

ćemo se podrobno pozabaviti u posebnom poglavlju, jer se radi o jednom od najvažnijih aspekata objektne metodologije.

Ad hoc (lat. "u datu svrhu", "za datu potrebu") polimorfizam iz primera 3 i 4 predstavlja bitno različitu vrstu polimorfizma od univerzalnog, jer se različito ponašanje postiže ne samoprilagođavanjem koda nego korišćenjem različitih kodova u istu ili sličnu svrhu (Elmer, [27]). Ad hoc polimorfizam deli se na dve podvrste: preklapanje (primer 3) i koercitivni ili prinudni polimorfizam (primer 4).

Preklapanje (engl. *overloading*) operatora[49] i potprograma (metoda) nije ništa drugo do pridruživanje istog simbola različitim rutinama, kao što je to slučaj npr. sa operatorima * ili &. Prema tome, dva ili više operatora su preklopljeni ako imaju isti simbol uz različit kod, a isto važi i za potprograme odnosno metode. Očigledno, preklapanje je specifična vrsta polimorfizma jer se istom identifikatoru pridružuje *skup* rutina koje, svaka za sebe, imaju *jedan* nepromenljiv oblik. Takve rutine nazivaju se monomorfnim, tako da se za preklapanje kaže da je to polimorfizam predstavljen skupom monomorfizama. Napomenimo odmah da se u C-u preklapanje operatora može samo konstatovati, dok C^{++} pruža mogućnost njegove programske realizacije, što će biti detaljno izloženo u ovom poglavlju.

Preklapanje potprograma i metoda takođe nije novina: u paskalu, na primer, različiti moduli (tzv. "jedinice") mogu sadržati istoimene potprograme koji se eksplicitno pozivaju kvalifikovanjem pomoću imena jedinice. Isto tako, nema nikakvih prepreka da se u različitim klasama u C^{++} nađu funkcije-članice sa istim imenom.

U pogledu preklapanja imena funkcija, C^{++} uvodi jednu inovaciju, a to je mogućnost da se u *istom* modulu ili klasi nađu slobodne funkcije ili funkcije-članice jednakog imena. Da bi se takve funkcije ipak mogle prepoznati negde mora postojati razlika: ona se sastoji u tome da funkcije sa istim imenom moraju da se razlikuju po broju ili tipu formalnih parametara, pri čemu je minimalna razlika tip jednog parametra. Tip rezultata nije obuhvaćen ovim pravilom. Zapazimo da smo na takvu pojavu već nailazili i to kod konstruktora: svi konstruktori jedne klase nose isto ime - ime klase - ali se razlikuju po formalnim parametrima.

Jasno je, dakle, da se u okviru preklapanja pojavljuju dva slučaja između kojih postoji tanka linija razdvajanja. Jedan je preklapanje potprograma koji se nalaze u različitim, međusobno nezavisnim modulima ili klasama, gde se radi o nekoj vrsti homonimije imena. Drugi slučaj je preklapanje potprograma, metoda i operatora u istom modulu-klasi. Ove dve vrste preklapanja nose i nazive redom *semantičko* odnosno *sintaksno preklapanje* [82].

Bez obzira na to da li se radi o semantičkom ili sintaksnom perklapanju, treba imati na umu da nije dobro da operatori ili funkcije nose isto ime *slučajno*. U

[49] Neki autori za preklapanje upotrebljavaju doslovni prevod "preopterećenje operatora" koji deluje pomalo nezgrapno te neće biti upotrebljavan u ovom tekstu.

tom smislu, veoma je preporučljivo pridržavati se jednog pravila koje je od izuzetne važnosti pri opredeljivanju za upotrebu preklapanja. U pitanju je ***pravilo očuvanja semantike*** koje glasi:

- Preklopljeni potprogrami, metode i operatori treba da imaju istu ili, bar, sasvim sličnu semantiku.

Na primer, potprogram ili metoda sa nazivom *sum* treba da obavlja sumiranje ili objedinjavanje operanada; ako je naziv *store*, posao treba da bude nekakvo skladištenje; samo iz čiste zlobe programer će klijentu podmetnuti preklopljen operator oduzimanja označen sa "+". Primetimo da, na primer, preklopljeni paskalski operator "+" odgovara sabiranju, uniji ili konkatenaciji, a to su operacije čija je zajednička karakteristika objedinjavanje (brojeva, skupova, stringova). Istini za volju, u programskom jeziku C ima odstupanja od pravila očuvanja semantike: množenje i dereferenciranje pokazivača imaju isti simbol "*" i nikakvu međusobnu vezu, kao ni bit-konjunkcija "&" sa adresnim operatorom. Razlog za odstupanje je čisto tehničke prirode i posledica je mnoštva operatora (preko 40) s jedne i ograničenog broja simbola na tastaturi s druge strane.

Prenebregavanje pravila očuvanja semantike može biti prilično respektabilan izvor grešaka. Na primer, operator dodele "=" u C^{++} ima dvostruk zadatak: da kopira vrednost s desne strane u levu *i* da kao rezultat vrati vrednost desne strane. Programeri su toliko navikli na to da od svake verzije preklopljenog operatora "=" očekuju da izvrši oba zadatka. Primerice, ako preklopljeni operator ne vrati rezultat, to može izazvati nedoumicu, nervozu pa i grešku (tada se, recimo, ne može izvršiti višestruko dodeljivanje tipa x= y= z).

Poslednja vrsta polimorfizma je tipa *ad hoc* i zove se ***koercitivni (prinudni) polimorfizam***. Ovde se, u stvari, radi o poznatim pojmovima: implicitnoj i eksplicitnoj konverziji tipa u izrazima odn. pozivima potprograma ili metoda.

Implicitnu konverziju tipa imao je još fortran. Naime, ako u istom izrazu egzistiraju realna i celobrojna promenljiva tada će prilikom sračunavanja vrednost celobrojne promenljive biti prethodno konvertovana u realnu. Bez implicitne konverzije mešovitih izraza ne bi moglo biti. Ista takva konverzija vrši se i pri pozivu potprograma sa realnim parametrom i celobrojnim argumentom. Implicitna konverzija postoji i u paskalu jer - ne zaboravimo - moderne verzije paskala imaju familiju celobrojnih tipova (integer, shortint, longint, byte, word) i familiju realnih tipova (real, single, extended, double, comp), tako da može doći do svakovrsnog mešanja tipova u izrazima, bez obzira na to što je paskal poznat kao jako tipiziran jezik. Programski jezik C je paradigma slabo tipiziranog jezika, što znači da mu je jedna od osnovnih karakteristika izraženo mešanje tipova te intenzivna primena implicitne konverzije.

Pored implicitne, C poseduje i mogućnost eksplicitne konverzije tipa. Ako je φ neki izraz tada npr. (int) φ obavezno ima celobrojnu vrednost, nezavisno od tipa samog izraza φ. Inače, u objektnom ambijentu, eksplicitna konverzija tipa

koristi se u sadejstvu sa dinamičkim objektima, a u svrhu realizovanja inkluzionog polimorfizma.

U nastavku ćemo, kao prvo, razmotriti preklapanje, a zatim i koercitivni polimorfizam, dok će ostale vrste polimorfizma biti opisane u narednim poglavljima.

6.1. PREKLAPANJE FUNKCIJA U C^{++}

Preklapanje funkcija u C^{++} podrazumevamo mogućnost da dve rutine mogu da dobiju isto ime. Rutine mogu biti metode ili pak slobodne funkcije, što opet predstavlja retroaktivne promene u procedurnom delu, tj. u C-u. Preklapanje funkcija pojavljuje se u dva vida, [82]:

1. *Semantičko* preklapanje kod kojeg se metode sa istim imenom nalaze u različitim klasama (isto važi i za slobodne funkcije koje se nalaze u različitim modulima)
2. *Sintaksno* preklapanje gde se metode sa istim imenom nalaze u istoj klasi (slobodne funkcije sa istim imenom u istom modulu).

Prvi slučaj - ***semantičko preklapanje*** - sam je po sebi jasan, ne predstavlja novu informaciju niti zahteva inovacije u prevodiocu, što nikako ne znači da se sme prenebregnuti. Naprotiv, radi se o vrlo važnom vidu preklapanja i to iz dva razloga:

1. Prvi razlog jeste zakon očuvanja semantike koji nalaže da metode sa istom ulogom u različitim klasama[50] ne samo što mogu nego i *moraju* nositi isto ime. U suprotnom, objektni sistem, koji po pravilu obiluje metodama, dodatno bi se (i potpuno nepotrebno) komplikovao. Zamislimo sistem klasa u kojima se metoda za čitanje zove čas *citaj*, čas *ulaz*, pa *ucitati*, *citaj*, *read*, *input*... Problem koji i inače opterećuje objektni pristup - mnoštvo različitih metoda kojima programer manipuliše istovremeno - povećao bi se bez ikakvog povoda. O ovom treba posebno voditi računa jer klase koje se istovremeno koriste ne moraju biti istovremeno i razvijene, tako da je dobra praksa usvojiti određene konvencije na nivou tima i njih se strogo pridržavati.
2. Drugi razlog je čak i značajniji. Radi se o tome da se istoimene metode mogu naći u različitim klasama koje su, međutim, povezane najvažnijom vezom u objektnom programiranju - nasleđivanjem. Metode sa istom semantikom u takvim klasama *moraju* imati isto ime - čak čitav prototip - u cilju ostvarivanja inkluzionog polimorfizma koji, u zajednici sa pomenutim nasleđivanjem, čini fundament objektne paradigme.

Drugi vid preklapanja funkcija jeste ***sintaksno preklapanje***, tj. situacija u kojoj se istoimene funkcije nalaze u istoj klasu ili modulu. Mada to deluje neobično, ne radi se o narušavanju jedinstvenosti identifikatora nego o prostoj činjenici da je naziv

[50] dakle, sa istom semantikom

preklopljene funkcije samo *deo* njenog identifikatora, dok ostatak čine broj i tip argumenata. Postupci i pravila za preklapanje slobodnih funkcija i funkcija-članica klasa identični su. Neka su date dve funkcije sa prototipovima

$$T_0^a\ f(T_1^a, ..., T_m^a);$$
$$T_0^b\ f(T_1^b, ..., T_n^b);$$

Da bi se ove dve funkcije mogle preklopiti dovoljno je da bude $m \neq n$. Ukoliko je pak $m = n$, tada je potrebno da za najmanje jedan par odgovarajućih tipova parametara važi $T_i^a \neq T_i^b$, $i = 1, 2, ..., n$. Iz ovog pravila isključeni su tipovi samih funkcija T_0^a i T_0^b koji *nemaju* uticaja na njihovo raspoznavanje. Pravilo se lako proširuje na proizvoljan broj preklopljenih funkcija.

Sintaksno preklapanje funkcija ne spada u standardnu praksu većine programera koji su - ipak - navikli da potprograme identifikuju imenom. Stoga nije na odmet sugerisati dozu opreza, posebno zato što C^{++} postavlja dodatna ograničenja u pogledu konverzije tipova, a prilikom prenosa argumenata. Ovo nikako ne treba tumačiti kao preporuku za izbegavanje preklapanja - naprotiv, ako programer dobro poznaje pravila i limite, opasnosti od pravljenja grešaka nema ili je, bar, minimalna. Najvažnije pravilo, iako neformalno, jeste pomenuto pravilo očuvanja semantike koje, interpretirano u obrnutom smislu, nalaže da funkcije (metode) koje nemaju istu semantiku ne treba preklapati. Razmotrićemo, sada, nekoliko karakterističnih situacija na koje se nailazi pri upotrebi ovog mehanizma.

Prvi, najbezbedniji, slučaj je preklapanje funkcija koje imaju različit broj parametara, pri čemu ni jedan od njih nema podrazumevanu vrednost. Pošto podrazumevanih vrednosti nema, broj stvarnih parametara je taj koji identifikuje odabranu verziju funkcije. Funkcije

```
void oneOrTwoVars(double x) {...}
void oneOrTwoVars(double x, double y) {...}
```

mogu biti pozvane sa proizvoljnim standardnim tipovima argumenata jer se funkcija bira isključivo na osnovu njihovog broja koji je obavezno različit. Automatska konverzija tipa argumenata, ako je uopšte potrebna, vrši se u skladu sa pravilima programskog jezika C.

Identifikovanje funkcija brojem argumenata potpuno je bezbedno samo ako nema parametara sa podrazumevanim vrednostima, jer se u potonjem slučaju funkcija može pozvati i sa manjim brojem argumenata od deklarisanog broja parametara. Ukoliko ima podrazumevanih vrednosti može se dogoditi da se deo parametara koji nemaju podrazumevane vrednosti poklapa, što dovodi do greške. Posmatrajmo jedan par takvih funkcija:

```
void f(double x, double y=0.0), double z=0.0) {...}     // Verzija 1
void f(double x, int i=0) {...}     // Verzija 2
```

Neka su *a*, *b* i *c* promenljive tipa *double*, a *j* promenljiva tipa *int*. Pozivi

```
f(a,b,c);
f(a,j);
```

su regularni. U prvom se referencira verzija 1, a u drugom verzija 2. Međutim, referenca

```
f(a);
```

ne može biti razrešena zato što se ne zna tip drugog i eventualno trećeg parametra koji, u stvari, jedini identifikuju verziju.

Drugi slučaj je već znatno složeniji i podložniji poskliznućima. Radi se o situaciji kada je broj formalnih parametara u preklopljenim funkcijama isti, svakako bez podrazumevanih vrednosti. Pošto broj argumenata više ne može da posluži za izbor funkcije, primenjuje se poseban algoritam čiji precizan opis sledi. Neka su date funkcije

```
long intOrDouble(int i) {...}     // Verzija 1
short intOrDouble(double x) {...}     // Verzija 2
```

Još jednom napominjemo da tipovi samih funkcija (u našem primeru *long* i *short*) ne učestvuju u postupku identifikacije. Postupak za izbor verzije funkcije ima sledeći oblik:

1. Prvo se proverava tzv. "potpuno slaganje" parametara i argumenata što je, u ovom kontekstu, najmanje rizičan način poziva. Ako se *intOrDouble* pozove sa argumentima tipa *int* odnosno *double* smatra se da je slaganje potpuno i biće odabrane verzije 1 odnosno 2. Izraz "potpuno slaganje" je specifičan i podrazumeva da se celobrojna nula, *short* i *char* potpuno slažu sa tipom *int*, a tip *float* sa tipom *double* (obrnuto ne važi!). Dakle, ako funkciju *intOrDouble* pozovemo sa parametrom tipa npr. *char* aktiviraće se verzija 1 jer se *char* "potpuno slaže" sa *int*. Iz istih razloga parametar tipa *float* aktiviraće verziju 2.
2. U slučaju da nema potpunog slaganja u smislu gornje definicije ovog pojma, traži se slaganje korišćenjem implicitnih konverzija osnovnih tipova podataka. Po skromnom mišljenju ovog autora, pravilo je dosta nesigurno, ponajviše zbog toga što se ne primenjuju standardna pravila za konverziju. Prema zvaničnim pravilima, par preklopljenih funkcija

```
void intIntOrDoubleDouble(int i, int j) {...}     // Verzija 1
void intIntOrDoubleDouble(double x, double y) {...}     // Verzija 2
```

trebalo bi da može da se pozove sa

```
intIntOrDoubleDouble(a, k);     // Argument a je double, k je int
intIntOrDoubleDouble(k, a);
```

referencirajući u oba slučaja verziju 2. Nažalost, ustanovljeno je da egzistira bar jedan prevodilac koji ovo neće dozvoliti! I u mnogim drugim prilikama implicitna konverzija otkazuje usled nemogućnosti razrešenja referenci na preklopljene funkcije. Takvih situacija ima sasvim dovoljno da se oslanjanje na implicitnu konverziju može pretvoriti u dosta neizvestan poduhvat. Navedimo još nekoliko primera: neka su dati parovi preklopljenih funkcija

```
void intOrDouble(int i) {...}
void intOrDouble(double d) {...}

void longOrDouble(long g) {...}
void longOrDouble(double d) {...}

void intOrFloat(int i) {...}
void intOrFloat(float f) {...}
```

Sledeći pozivi generisaće poruke o greškama (ponovo, bar na jednom standardnom prevodiocu na kojem su proveravani):

```
int intg; long lng; double dbl; long double lngdbl;
. . . . . .
intOrDouble(lng);     // Slucaj A
intOrDouble(lngdbl);     // Slucaj B
longOrDouble(intg);     // Slucaj C
intOrFloat(dbl);     // Slucaj D
```

Nezavisno od toga da li je izvodljiva konverzija u složeniji tip (slučajevi A i C) ili se mora konvertovati u manje složen tip (slučajevi B i D), svi navedeni pozivi biće tretirani kao neodređeni, još u fazi pripreme programa. Stoga je za programera najbolje da se kloni ovakvih eksperimenata, a ukoliko je baš neophodno može uvek pribeći eksplicitnoj konverziji tipa argu-

menata. Pozivi

```
longOrDouble((long) intg);
longOrDouble((double) intg);
```

aktiviraće korektne verzije funkcije.

3. Kada nema ni potpunog slaganja niti implicitne konverzije, prevodilac će pokušati da uključi konverzije koje je definisao programer, a kako se takve konverzije realizuju videćemo u poslednjem delu ovog poglavlja.

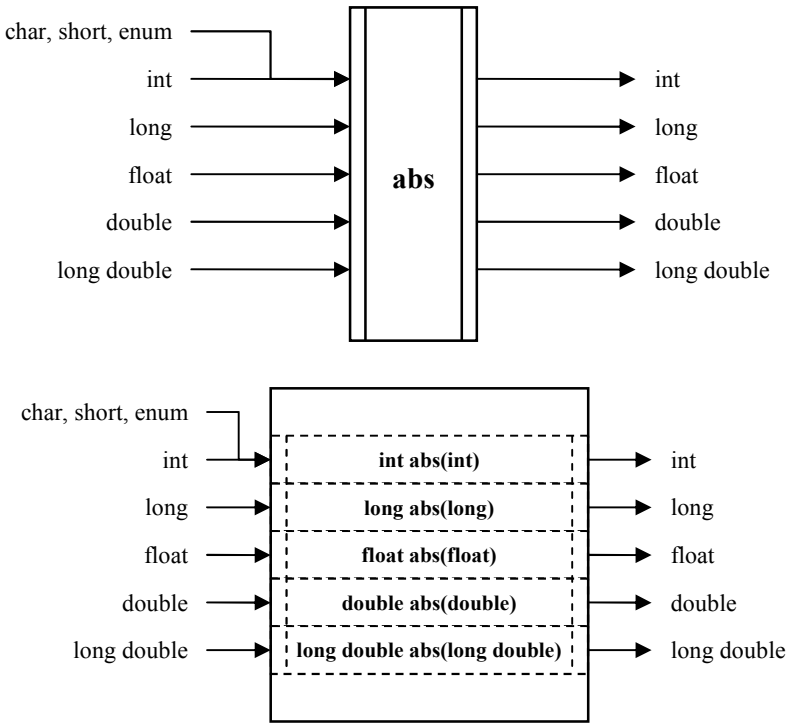
Primer 6.1. Način i smisao preklapanja funkcija ilustrovaćemo jednostavnim primerom realizacije slobodnih funkcija za određivanje apsolutne vrednosti, sa parametrima redom *int*, *long*, *float*, *double* i *long double*. Funkcije će nositi (isti) naziv *abs* i svaka će vraćati tip rezultata koji odgovara tipu parametra. Uočimo, pre svega, da je pravilo očuvanja semantike strogo poštovano jer svih pet verzija rade isti posao - određivanje apsolutne vrednosti, a razlikuju se samo po tipu argumenta. Za stvarne parametre tipa *int*, *short* ili *char* aktiviraće se prva verzija, za *long* druga, za *float* treća, za *double* četvrta i konačno za *long double* peta.

Pored pravila očuvanja semantike obezbeđeno je i sprovođenje principa skrivanja informacija tako što su funkcije smeštene u poseban modul sa interfejsom (zaglavljem) ABSVAL.HPP i telom modula ABSVAL.CPP.

```
/*****************************************************************
**

        ZAGLAVLJE MODULA SA PREKLOPLJENIM FUNKCIJAMA "ABS"

Naziv datoteke: ABSVAL.HPP
*****************************************************************
*/
int abs(int);
long abs(long);
float abs(float);
double abs(double);
long double abs(long double);
```

```
/*****************************************************************
**

        TELO MODULA SA PREKLOPLJENIM FUNKCIJAMA "abs"
```

```
Naziv datoteke: ABSVAL.CPP
*************************************************************
*/
#include "absval.hpp"

int abs(int intg) {return intg>=0 ? intg : -intg;}
long abs(long lng) {return lng>=0 ? lng : -lng;}
float abs(float flt) {return flt>=0.0 ? flt : -flt;}
double abs(double dbl) {return dbl>=0.0 ? dbl : -dbl;}
long double abs(long double lndb) {return lndb>=0.0 ? lndb : -lndb;}
```



Slika 6.2

Da bi se do kraja shvatili suština i pogodnosti preklapanja važno je da realizovani softver osmotrimo i iz ugla klijenta-programera. Iako funkcija ima ukupno pet i svaka je monomorfna, programer se prema njima odnosi kao prema *jednoj*, spolja polimorfnoj, funkciji sa nazivom *abs* i osobinom da je osetljiva na tip argumenta. To i jeste smisao ranije navedenog zaključka da je *ad hoc* polimorfizam tipa preklapanja, u stvari, skup monomorfizama.

Na slici 6.2 gore prikazan je pogled klijenta-programera na funkciju *abs*, dok se dole vidi stvarno stanje.

6.2. PREKLAPANJE OPERATORA U C^{++}

Preklapanje operatora, tj. pridruživanje istog simbola različitim operacijama, u osnovi se ne razlikuje od preklapanja funkcija, jednostavno zato što i programske funkcije i operatori imaju prirodu matematičkog preslikavanja. Uostalom, ni u matematici nije zabranjeno da se operator prikaže kao preslikavanje. Nema nikakve prepreke da umesto a+b napišemo +(a,b), samo što to nije uobičajeno jer je svrha simboličkog prikaza operatora da se olakša čitanje izraza. Neuporedivo je lakše pročitati npr. a*(b+c)-d nego -(*(a,+(b,c)),d). Preklapanje operatora nije nepoznato ni u matematici: simbol "-" može da označi promenu predznaka, oduzimanje brojeva, oduzimanje matrica ili diferenciju skupova. Znak "+" pridružuje se sabiranju, ali i apstraktnim operacijama, recimo u Bulovim algebrama.

Preklapanje operatora u C^{++} služi istoj svrsi - povećanju čitljivosti složenijih izraza (i stoga ne treba precenjivati njegov značaj!). I ne samo to: u C^{++} ova mogućnost uvedena je stoga da se programska funkcija koja ima osobine matematičkog operatora baš tako i prikaže. Ako je A*B proizvod dveju matrica u modelu, svakako je poželjno predstaviti ga tako i u programskoj realizaciji. Podvucimo odmah da se preklapanje operatora u C^{++} izvodi isključivo posredstvom objektnog programiranja. Ostvaruje se putem tzv. ***operatorskih funkcija*** koje se izdvajaju time što za identifikator imaju simbol (uračunavajući i neke od rezervisanih reči poput *new* i *delete*), kao i po drukčijem načinu pozivanja. Inače, one se ponašaju poput svih ostalih funkcije, tako da i same mogu biti preklopljene. Da bi se naznačilo da je neka funkcija, slobodna ili članica klase, operatorska potrebno je ispred simbola što predstavlja ime napisati službenu reč *operator*. Dakle, prototip operatorske funkcije sa simboličkim imenom α je

$$\text{tip operator } \alpha(p_1, ..., p_n);$$

gde su svi ostali elementi prototipa isti kao kod ostalih funkcija. Primerice, operatorska funkcija za sabiranje kompleksnih brojeva (klasa *Complex*) imala bi prototip

Complex operator +(const Complex&, const Complex&);

Kada smo rekli da je preklapanje operatora izvodljivo samo preko objekata imali smo na umu osnovno ograničenje:

- operatorska funkcija mora biti članica klase ili
- bar jedan od parametara mora da bude objekat.

Naravno, navedene stavke se ne isključuju. Drugim rečima, operatorske funkcije su

ili funkcije-članice neke klase ili su slobodne funkcije sa bar jednim parametrom koji je objekat. Slobodne operatorske funkcije mogu da budu - i po pravilu jesu - *prijateljske (kooperativne)*. Operatori u okviru standardnih (procedurnih) tipova ne mogu se preklapati, što neposredno proizlazi iz pomenutih uslova.

Osim u pogledu ovog ograničenja operatorske funkcije se ponašaju poput svih drugih. Funkcija "+" klase *Complex* mogla bi se aktivirati kao i ostale slobodne funkcije:

```
Complex a, b, c;
. . . . . .
a= operator +(b,c);     // Sluzbena rec operator je obavezna
```

što, naravno, nema nikakvog smisla niti opravdanja. Osnovna ideja pri uvođenju operatorskih funkcija je upravo izbegavanje takvog poziva stvaranjem uslova da se operatorske funkcije *aktiviraju sintaksnim konstruktima koji odgovaraju operatorima*. Prema tome, umesto da pomenutu operatorsku funkciju aktiviramo eksplicitnim pozivom kao gore, možemo (i moramo, ukoliko želimo da cela stvar ima smisla) koristiti tzv. ***infiksni poziv*** oblika

```
a= b+c;
```

Zapazimo - a to je važno zbog razumevanja načina funkcionisanja operatorskih funkcija - da su izrazi *operator +(b,c)* i *b+c* u svakom pogledu ekvivalentni. U stvari,

- osnovna karakteristika operatorskih funkcija jeste mogućnost infiksnog poziva (prefiksnog odn. postfiksnog za unarne operatore).

Već na osnovu do sada izloženog lako zaključujemo da se kroz mehanizam preklapanja operatora ostvaruje uloga klase kao sredstva za izradu tipova podataka, te da je u ovom kontekstu akcenat na objektu u domenu realizacije, a ne u domenu modela.

Tehnički, preklapanje operatora je *zamena funkcije-članice ili slobodne funkcije operatorskom funkcijom*. Pri tom, programski jezik C^{++} zahteva poštovanje nekih formalnih pravila od kojih navodimo najvažnija (više detalja može se naći u [77]):

1. U svrhu preklapanja mogu se koristiti samo postojeći operatori C^{++} (tj. + - % & = += new delete itd.). Ne mogu se uvoditi novi simboli za operatore, npr. ↑ ili ∩.
2. Ne mogu se preklopiti operatori

```
.
.*
::
```

?:
sizeof

3. Preklopljeni operatori zadržavaju prioritet i smer grupisanja. Na primer, sve operatorske funkcije označene simbolom > imaju prioritet 10 i smer grupisanja s leva u desno.

I još nešto: svaka klasa apriori sadrži operatore

=
& (adresni operator)
, (niz izraza)

koji se mogu preklopiti, što je naročito važno za operator "=" kada klasa sadrži dinamičke članove.

Podatak-član *this*

Pre nego što pređemo na detaljan opis postupaka za preklapanje operatora osvrnućemo se na jedan podatak-član koji igra važnu ulogu kod nekih tehnika preklapanja. Radi se o podatku-članu koji se ne navodi posebno jer ga sadrži svaka klasa i kreira se automatski prilikom instanciranja. Naziv mu je *this*, a po prirodi je pokazivač na instancu klase takav da, za matični objekat, podatak-član *this* sadrži njegovu adresu. Adresa se upisuje u *this* prilikom konstrukcije objekta, što stvara uslove da objekat pristupi sopstvenom memorijskom prostoru. Prema tome, ako definišemo klasu

```
class ExampleClass {
. . . . . .
};
```

i njenu instancu

```
ExampleClass inst;
```

tada će vrednost **this* u okviru svake metode, a primenjene na *inst*, dati vrednost dereferenciranog pokazivača na *inst*, a *this* njegovu adresu. Ovaj podatak-član nije dostupan prijateljskim funkcijama, jer se one ne mogu pozvati preko instance klase.

Postoji više načina za preklapanje operatora koji se primenjuju u različitim situacijama, zavisno od prirode operatora koji se preklapa (unaran, binaran, sa bočnim efektom ili bez njega). Dve osnovne tehnike za preklapanje operatora su

- primena operatorskih metoda i
- primena operatorskih slobodnih funkcija koje su, kako smo rekli, praktično uvek prijateljske (zbog brzine).

U mnogim slučajevima ove tehnike primenljive su alternativno, pa se onda postavlja problem inženjerskog odlučivanja o tome kada primeniti koju. Ako nema for-

malnih ograničenja samog C++, odgovor pruža pravilo očuvanja semantike iz kojeg direktno proizlazi preporuka[51]:

- operatore koji menjaju stanje operan(a)da treba preklopiti operatorskim metodama (uključujući i slučaj kada je rezultat *lvrednost*)
- operatore koji ne menjaju stanje ili vrednosti operan(a)da treba preklopiti prijateljskim funkcijama.

Na primer, preklopljeni operator dodele "=" po osnovnoj semantici menja stanje operanda sa leve strane, tako da se preklapa kao operatorska metoda koja menja stanje objekta-operanda sa leve strane i vraća kao rezultat operand s desne (tačnije, zajedničku vrednost operanada posle izvršenja operacije). Nasuprot tome, operator binarnog sabiranja ima dva ravnopravna operanda koji ne menjaju vrednost, pa je stoga pogodno primeniti operatorsku slobodnu funkciju. Opredeljivanje za tehniku preklapanja obrazložićemo podrobnije u sklopu razmatranja konkretnih slučajeva.

Sintaksa poziva preklopljene metode i preklopljene slobodne funkcije ista je, tako da se samo na osnovu oblika izraza što ih sadrži ne može zaključiti koja je tehnika u pitanju. Recimo, iz sintakse izraza *a*b* ne može se dešifrovati da li je "*" preklopljena metoda ili preklopljena slobodna funkcija. S druge pak strane, ove dve vrste operatorskih funkcija očigledno nisu iste i vrlo je važno razumeti njihovo (različito) ponašanje. Da bismo precizno predočili način rada jednih i drugih najbolje je analizovati standardnu formu poziva.

Neka je *op* unarni operator preklopljen operatorskom metodom u klasi *K* sa instancom *k*. Tada izraz *op k* nije ništa drugo do aktiviranje metode *op* konstruktom *k. operator op()*. Primerice, *-k* znači, u stvari, *k. operator -()*. Ako je *op* binarni operator, tada *k op* α predstavlja aktiviranje metode *op* sa parametrom α, tj. ekvivalentni oblik je *k. operator op(*α*)*. Recimo, *k= t* znači *k. operator =(t)*.

Kod slobodnih funkcija situacija je drukčija jer one nisu članovi klase čak ni kada su definisane kao prijateljske. Kada operator *op* zamenjujemo slobodnom funkcijom sa istim simbolom tada, za slučaj da je recimo binarna, izraz *x op y* ima interpretaciju *operator op(x,y)*. Dakle, ako je operator "+" preklopljen slobodnom funkcijom, izraz *x+y* ima značenje *operator +(x,y)*.

Kako smo rekli, C++ dozvoljava da se pozivi operatorskih funkcija tipa *k. operator =(t)* ili *operator +(x,y)* po želji mogu zadati i u takvom obliku, no ideja preklapanja operatora sastoji se upravo u izbegavanju toga.

U nastavku, opisaćemo postupke za preklapanje karakterističnih grupa operatora. Za ilustraciju koristićemo uglavnom klasu kompleksnih brojeva *Complex* jer je ona tipičan primer klase u kojoj treba iskoristiti ovu pogodnost C++.

6.2.1. Preklapanje operatora dodele =

Operator dodele = najvažniji je za preklapanje upravo zato što apriori

[51] Nikako pravilo, jer ima izuzetaka!

postoji u svakoj klasi. Može se preklopiti uvek, a *mora* se preklopiti ako objekat ima dinamičkih članova, da bi prilikom kopiranja bio prenet ceo objekat, a ne samo njegov statički deo. Naime, ako su *a* i *b* objekti sa dinamičkim članovima tada ugrađena operacija dodele oblika *a= b* prenosi, pored ostalog, samo sadržaj odgovarajućih pokazivača iz *b* u *a* (plitka kopija), što ima za posledicu da se delovi *a* i *b* u dinamičkoj memoriji poklapaju. Da bi se to izbeglo, tj. da bi memorijski prostori objekata bili razdvojeni, operator "=" mora biti preklopljen tako da se kopira ne sadržaj pokazivača nego memorije na koju oni pokazuju, tj. da se izvrši *duboko kopiranje*.

Kako je pomenuto, najvažnije pitanje za svako, pa i ovo, preklapanje jeste izbor tehnike. U načelu, na raspolaganju su dve varijante: operatorska metoda i operatorska slobodna funkcija. Naglasili smo, takođe, da operatori koji menjaju vrednost operanada treba da budu preklopljeni operatorskim metodama. U slučaju operatora dodele nema dileme, jer pravila C^{++} eksplicitno zahtevaju preklapanje operatorskom metodom. Bez obzira na to što alternative nema, analizovaćemo osobine ovog operatora, makar stoga da bismo razumeli zašto C^{++} propisuje preklapanje isključivo operatorskom metodom. Počnimo od glavnih osobina operatora dodele. To su:

1. Objekat sa leve strane *menja stanje* tako da ono bude identično stanju objekta koji je rezultat izračunavanja izraza sa desne.
2. Operacija kao rezultat *vraća stanje* (objektnog) izraza sa desne strane. Kako su posle izvršenja dodele stanja izvornog i odredišnog objekta identična, za rezultat se može iskoristiti i stanje objekta sa leve strane.
3. Rezultat operacije mora biti *lvrednost* ("lvalue") da bi se omogućilo korišćenje višestruke dodele oblika a= b= c.

Činjenica da operator dodele menja stanje objekta jasno ukazuje na razlog zbog kojeg C^{++} propisuje preklapanje dodele operatorskom metodom i to metodom koja ima prirodu modifikatora. Metoda treba da ima i osobine indikatora jer vraća stanje formirano na osnovu stanja izvornog, odnosno odredišnog objekta. Sledi da je tip operatorske metode za preklapanje "=" referenca na matičnu klasu.

Za klasu *Complex* prototip preklopljenog operatora dodele ima sledeći oblik:

```
Complex &operator =(const Complex &);
```

gde je rezultat referenca da bi kompletan izraz bio *lvrednost*. Parametar je takođe referenca u cilju zaobilaženja konstruktora kopije. Puna definicija operatorske funkcije izgleda ovako:

```
Complex& Complex::operator =(const Complex &z) {
  r= z.r; i= z.i;
```

```
return *this;
}
```

S obzirom na to da nema upravljačkih struktura, operator dodele za ovaj primer može se realizovati i *inline*. Povrh svega, dužni smo napomenuti da u konkretnom slučaju operator dodele nije bilo nužno preklopiti jer i osnovni operator dodele funkcioniše identično (klasa nema dinamičkih članova). Primeri:

```
Complex x, a, b, c(1,2);
. . . . . .
x= c;
. . . . . .
a= b= c;
```

Da bismo ispitali kako se ponaša preklopljeni operator dodele kada klasa ima dinamičkih članova posmatraćemo primer jednostruko spregnute liste sa nazivom *TList* i elementima tipa *T*. Prilikom preklapanja operatora dodele mora se voditi računa o sledećem:

- Da ne bi došlo do curenja memorije, pre početka kopiranja lista sa leve strane mora se isprazniti (tj. mora se eliminisati dinamički deo objekta).
- Vrednost koja se vraća je po svojoj prirodi deskriptor liste (tj. statički deo odgovarajućeg objekta)
- Posebno se mora obratiti pažnja na jednu potencijalnu neugodnost - dodelu liste samoj sebi naredbom oblika *lst= lst;* gde je *lst* instanca klase. Inače, naredba je sasvim regularna. Ona ne može funkcionisati normalno jer će pre kopiranja lista sa leve strane biti ispražnjena, a na desnoj strani nalazi se ista ta lista! Da bi se prevazišao ovaj problem - a on postoji kod svake klase sa dinamičkim članovima - neophodno je na samom početku proveriti da li su liste sa leve i desne strane identične i u takvom slučaju odmah terminirati.

Da bismo uprostili tekst preklopljene metode "=" pretpostavićemo da klasa *TList* raspolaže metodama *clear()* za pražnjenje liste i *copyFrom(const TList &)* za kopiranje sadržaja liste-parametra u datu listu. Operatorska metoda ima sledeću definiciju:

```
TList& TList:: operator =(const TList &rlst) {
 if(&rlst == this) return *this;     // Dodela liste samoj sebi
 else {
  clear();
  copyFrom(rlst);
  return *this;
```

```
}
```

6.2.2. Preklapanje ostalih operatora dodele

Ostali operatori dodele (+= -= *= itd.) preklapaju se na isti način kao i osnovni. Preklopljeni operator "+=" u klasi *Complex* izgledao bi ovako:

```
Complex& Complex::operator +=(const Complex &z) {
  r+= z.r; i+= z.i;
  return *this;
}
```

U okviru ovog primera iskoristićemo priliku da argumentujemo preporuku za izbor operatorske metode za preklapanje. Naime, operator += može se preklopiti i prijateljskom funkcijom sa prototipom

```
friend Complex operator +=(Complex &, Complex &);
```

i definicijom

```
Complex operator +=(Complex &levaStrana, const Complex &desnaStrana) {
  levaStrana.r+= desnaStrana.r; levaStrana.i+= desnaStrana.i;
  return levaStrana;
}
```

Ovde se modifikovani objekat vraća kao izlazni parametar *levaStrana*, a vrednost celokupnog izraza predstavlja vrednost funkcije. Ovo rešenje, iako dozvoljeno, manje je pogodno od prethodnog jer se primenom prijateljske funkcije ništa ne dobija u odnosu na primenu operatorske metode, dok nedostaci postoje. Glavna zamerka je principijelna i u vezi je (opet!) sa pravilom očuvanja semantike. Semantika osnovnog operatora "+=" je sledeća:

- primarni efekat izraza x+= y je uvećanje x za y
- sekundarni efekat je formiranje vrednosti celog izraza koja je jednaka x+y gde je x prethodna vrednost promenljive.

Preklapanjem operatora += prijateljskom funkcijom na način dat gore zadržava zahtevane osobine, ali uz malo odstupanje jer primarni efekat prijateljske funkcije - vraćanje vrednosti - odgovara sekundarnom efektu originalnog operatora, dok je osnovna namena operatora realizovana kao bočni efekat, što unosi nepotrebnu komplikaciju. Konačno, a možda i najvažnije, C^{++} postavlja dodatna ograničenja na preklapanje prijateljskim funkcijama (npr. "=" se ne može preklopiti prijateljskom funkcijom). Vođenje računa o izuzecima i ograničenjima ni u kom slučaju ne budi oduševljenje među programerima.

6.2.3. Preklapanje relacionih operatora

Relacioni operatori, bar u načelu, mogu se preklopiti na oba načina, tj. kao unarne parametrizovane operatorske metode ili kao binarne prijateljske funkcije. Odgovor na pitanje koju varijantu odabrati ponovo nudi pravilo očuvanja semantike. Neka je potrebno preklopiti operator "==" u klasi *Complex*. Osnovni operator "==" jeste binarni komutativni operator sa dva *ravnopravna* operanda. Pri tom, operator nema bočnih efekata. Ako bismo ga preklopili operatorskom metodom

```
int Complex::operator ==(const Complex &z) {
  return (r==z.r)&&(i==z.i);
}
```

tada bi izraz *a==b* imao stvarno značenje *a.operator ==(b)*, tj. *a* i *b* ne bi bili ravnopravni stoga što se operacija izvodi nad *a* sa *b* kao parametrom! Posebno, u odeljku posvećenom konverziji videćemo da je moguće obezbediti automatsku konverziju tipa *double* u klasu *Complex* što čini izvodljivom operaciju *z==d* (uz Complex z; double d;). Obrnuto nije moguće jer *d==z* znači *d. operator ==(z)*, a promenljiva *d* nije objekat.

Drugo rešenje, upotreba binarne prijateljske funkcije eliminiše ove probleme (ne uvodeći nove) jer se operandi u funkciji tretiraju ravnopravno. Preklopljena prijateljska funkcija ima prototip

```
friend int operator ==(const Complex &, const Complex &);
```

i definiciju

```
int operator ==(const Complex &z1, const Complex &z2) {
  return (z1.r==z2.r)&&(z1.i==z2.i);
}
```

6.2.4. Preklapanje binarnih aritmetičkih operatora

Za binarne aritmetičke operatore + - * / itd. važi isto što i za relacione: operandi su ravnopravni i nema promena stanja, tj. bočnih efekata. Zato se i za njihovo preklapanje koriste slobodne operatorske funkcije. Postoji, međutim, jedna bitno nova okolnost: operatori ne vraćaju neki od standardnih tipova, već opet objekat date klase (u našim primerima klase *Complex*). Ova činjenica je od presudne važnosti jer zahteva kreiranje novog, lokalnog objekta čije se stanje vraća kao rezultat. Postoje dve, u osnovi iste, tehnike:

- Korišćenje tzv. privremenog objekta
- Korišćenje tzv. bezimenog objekta.

Privremeni objekat nije ništa drugo do običan lokalni objekat preklopljene funkcije. Demonstriraćemo ovu tehniku na primeru kompleksnog sabiranja:

```
friend Complex operator +(const Complex &, const Complex &);     // Prototip

Complex operator +(const Complex &z1, const Complex &z2) {
  Complex w;
  w.r= z1.r+z2.r; w.i= z1.i+z2.i;
  return w;
}
```

U okviru preklopljene funkcije objekat *w* igra ulogu privremenog objekta. Skreće se pažnja na to da funkcija ne sme vraćati referencu na klasu *Complex* jer bi u tom slučaju bila prosleđena adresa objekta *w* koji je izašao iz dosega i izbrisan je sa steka!

Drugi način, upotreba bezimenog objekta, podrazumeva direktnu primenu konstruktora i to je sve u čemu se razlikuje od prve varijante. Uz isti prototip, definicija preklopljene operatorske funkcije je

```
Complex operator +(const Complex &z1, const Complex &z2) {
  return Complex(z1.r+z2.r, z1.i+z2.i);
}
```

6.2.5. Preklapanje unarnih aritmetičkih operatora

Unarni aritmetički operatori mogu se preklopiti se na oba načina: i prijateljskom funkcijom i metodom. Ako želimo da preklopimo operator promene predznaka (unarno "-") u klasi *Complex*, rešenje u vidu slobodne prijateljske funkcije izgledalo bi ovako:

```
friend Complex operator -(const Complex &);     // Prototip u naredbi class

Complex operator -(const Complex &z) {
  return Complex(-z.r,-z.i);
}
```

Interesantan je način poziva ove funkcije ilustrovan sledećim primerom:

```
Complex a, b = Complex(1,1);
. . . . . .
a= -b;
```

gde izraz *-b* ima interpretaciju *operator -(b)*. Uočimo da, zbog različitog broja operanada, unarni operator promene predznaka i binarni operator oduzimanja bez problema koegzistiraju u istoj klasi.

Preklapanje pomoću operatorske metode vrši se pomoću konstantne operatorske metode koja nema parametra, jer se izlaz generiše direktno iz stanja objekta na koji je metoda primenjena:

```
Complex operator -() const {
  return Complex(-r, -i);
}
```

Zapazimo da u ovom slučaju izraz *-b* iz primera značio *b.operator -()*.

Između navedena dva rešenja bitnih razlika u kvalitetu i nema no, s obzirom na to da originalni operatori ne utiču na operande, prijateljska operatorska funkcija može se smatrati konzistentnijim rešenjem.

6.2.6. Preklapanje operatora ++ i --

Unarni operatori "++" i "--" predstavljaju poseban slučaj. Pre svega, oni menjaju stanje objekta te se, saobrazno tome, preklapaju operatorskim metodama. Ovi operatori, međutim, imaju i jedinstvenu osobinu da se mogu pisati ispred operanda ili iza njega, sa različitim značenjem. U pitanju su, naravno, poznati prefiksni i postfiksni oblik. Preklapanje ćemo demonstrirati na primeru operatora "++" jer se potpuno ista tehnika primenjuje i na "--".

Prefiksni oblik preklapa se standardnim postupkom, posredstvom podatka-člana *this* za vraćanje promenjenog stanja. Neka je operator "++" u klasi *Complex* definisan tako da poveća za 1 i realni i imaginarni deo. Definicija prefiksnog operatora ima sledeći oblik (ilustracije radi, izvedena je *inline*):

```
const Complex &operator ++() {++r; ++i; return *this;}
```

gde je novina samo (neobavezni) modifikator *const*. O njegovoj nameni reći ćemo nešto više na kraju.

Preklapanje postfiksnog oblika nešto je složenije. Prvo, povratna vrednost odgovara stanju objekta *pre* primene operatora, što zahteva upotrebu lokalnog objekta. Postoji, međutim, jedan ozbiljniji problem: i prefiksna i postfiksna forma predstavljaju operatorske metode bez parametara, tako da imaju isti prototip i ne mogu se razaznati! Iz tog razloga autori programskog jezika bili su prinuđeni da naprave nešto što se ne može nazvati drugim imenom nego "zakrpa". Naime, postfiksni operator realizuje se kao operatorska metoda sa jednim parametrom tipa *int* koji de facto ne služi ničemu osim da omogući razlikovanje od prefiksnog oblika. Prilikom aktiviranja postfiksnog operatora parametar se ne navodi. Dakle, preklopl-

jeni postfiksni operator "++" u klasi *Complex* imao bi sledeću definiciju:

```
const Complex::Complex operator ++(int k) {
  Complex x(r,i);
  r++; i++;
  return x;
}
```

Primetimo da operator ne sme da vrati referencu na *Complex* jer je povratna vrednost *x* lokalni objekat. Upotreba prefiksnog i postfiksnog oblika je uobičajena:

```
Complex a, b(1,1);
. . . . . .
a= b++;     // Vrednosti: a=(1,1)  b=(2,2)
++b;     // Vrednost b=(3,3)
```

Ostaje još samo da se razjasni zašto su vrednosti operatorskih metoda definisane kao konstantni objekti. To, pre svega, nije obavezno i u mnogim situacijama ispuštanje modifikatora *const* ne uzrokuje probleme. Ipak, poteškoće mogu da se pojave u vezi sa primenom postfiksnog operatora, zato što on vraća privremeni objekat. Pretpostavimo da je postfiksni operator "++" realizovan bez modifikatora *const*, dakle sa prototipom

```
Complex operator ++(int);
```

Operator "++" je *lvrednost* tako da su na odgovarajući objekat primenljive metode-modifikatori. Konkretno, moguće je pisati

```
Complex z(1,1), one(1,0);
z++ = one;
```

Za očekivanje je da se rezultat formira tako što se prvo izračuna z++ (što daje kompleksni broj 2+2i), a zatim sve to prekrije vrednošću *one* koja je jednaka (1,0). Krajnji ishod trebalo bi da bude z = (1,0). Međutim, to se neće desiti jer je povratna vrednost postfiksnog operatora "++" privremeni objekat koji se formira na steku, te operacija dodeljivanja nema efekta. Rezultat će biti neočekivan: z = (2,2), a prevodilac (bar jedan) neće prijaviti grešku.

Ako programer hoće da se osigura od ovakvih i sličnih iznenađenja treba objekat-rezultat primene postfiksnog operatora da deklariše kao konstantan, tako da izrazi tipa *z++= one* ne mogu da se formiraju. Logično, ako se postfiksni operator realizuje kao konstantan to treba, uniformnosti radi, učiniti i sa prefiksnim.

6.2.7. Preklapanje operatora () [] ->

U programskom jeziku C/C^{++} - nota bene! - poziv funkcije tretira se kao n-arna operacija. Poziv funkcije oblika f(x) može se shvatiti kao operacija koja povezuje naziv funkcije *f* i promenljivu *x* u kojoj je sama operacija napisana oko drugog operanda. Nema stoga nikakvih prepreka da se dozvoli preklapanje i za operator "()". Programski jezik propisuje da se operator "()" preklapa isključivo operatorskom metodom sa opštim oblikom prototipa

$$T \text{ operator } () (T_1, ..., T_n)$$

gde je T tip (klasa) rezultata, a T_1, ..., T_n tipovi (klase) parametara.

Preklapanje ovog operatora daje mogućnost da se metoda aktivira izrazom koji sasvim podseća na poziv funkcije, tj.

$$p(x_1, ..., x_n)$$

s tim što *p* nije naziv funkcije nego *identifikator objekta* iz klase koja ima preklopljen operator "()". Dakle, ako je *a* instanca klase koja ima preklopljen operator "()" sa npr. jednim parametrom, tada se *a(z)* interpretira kao *a. operator ()(z)*.

Jedna od karakterističnih primena preklapanja operatora "()" je realizacija funkcija koje, pored argumenata, imaju više parametara sa vrednostima koje se retko menjaju. Tipičan slučaj je polinom

$$P_n(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$$

gde su n, a_0, ..., a_n parametri, a x promenljiva. Standardni način za realizaciju $P_n(x)$ je funkcija sa zaglavljem

```
double p(double x, int n, double a[]);
```

koja se poziva konstruktom

```
p(y, m, b)
```

Preklopljeni operator "()" nudi mogućnost za elegantniju realizaciju, "elegantniju" u smislu da više liči na matematičku notaciju. Treba definisati odgovarajuću klasu, recimo *Polinom*, sa podacima-članovima n, a_0, ..., a_n, dok promenljiva *x* ostaje kao parametar preklopljene operacije. Dajemo skraćeni prikaz realizacije klase:

```
class Polinom {
private:
```

```
  int n;
  double a[101];
public:
. . . . . .
double operator ()(double);
};

double Polinom::operator ()(double x) {
  double tmp = a[n];
  for(int i=n-1; i>=0; i--) tmp= x*tmp+a[i];
  return tmp;
}
```

Podrazumeva se, naravno, da klasa ima konstruktor(e) i metode za rukovanje vrednostima n, a[0], ..., a[n]. Ako se konstruiše instanca *p* klase *Polinom* i konstruktorom ili na neki drugi način postave vrednosti n, a[0], ..., a[n], tada se vrednost polinoma računa izrazima poput p(x) ili p(t+u*v) itd. Uočimo da, ako se klasa realizuje uz poštovanje inkapsulacije i kontrole pristupa, klijent-programer čak ni ne mora da zna da je *Polinom* klasa, a *p* njena instanca jer se navedeni izrazi sintaksno ne razlikuju od poziva slobodne funkcije.

Operator "[]" takođe se preklapa isključivo operatorskom metodom (takva su pravila C^{++}). Specifična semantika osnovnog operatora "[]" je *indeksiranje*, dakle pristup delovima složene promenljive ili pojedinačnom podatku-članu složenog objekta zadavanjem pozicije preko indeksa. U procedurnim programskim jezicima gde je indeksiranje operator vezan isključivo za nizove nije sasvim očigledno da ovaj operator ima dvostruku ulogu:

1. Indeksirani izraz može se pojaviti kao levi operand u operaciji dodele oblika npr. *x[i]= 5* gde je *x* niz. Drugim rečima, indeksirani izraz jeste *lvrednost*.
2. Indeksiranje kao operacija može se upotrebiti samo za očitavanje vrednosti odgovarajućeg elementa, kao npr. u izrazu *y= x[10].*

Posebnost ponašanja operatora indeksiranja može se uočiti u trivijalnom izrazu *x[i]= x[i].* Njegova stvarna struktura je

$$x\_element_i = x\_element_i\_vrednost$$

gde se lako uo;ava da iyray *x[i]* nema istu semantiku na levoj i desnoj strani operatora dodele.

U objektnom ambijentu ovo znači da preklapanje operatora indeksiranja mora obezbediti sve osobine akcesora (kako osobine selektora, tako i osobine indi-

katora). U prvom slučaju to znači da vrednost koju odgovarajuća operatorska metoda vraća mora biti *lvrednost*. U drugom pak slučaju podrazumeva se da metoda mora biti primenljiva i na konstantne objekte, jer se to očekuje od svakog indikatora. Da bi se ostvarila oba cilja, operator indeksiranja preklapa se *dva puta*.

Verzija operatora indeksiranja koja generiše *lvrednost* realizuje se tako što vraća kao rezultat *referencu*, tj. adresu. Ako je u nekoj klasi *K* preklopljen operator "[]", opšti oblik ove operatorske metode je

```
tip& K::operator [](parametri) {...}
```

gde je *tip* tip vrednosti koju vraća operatorska metoda.

Kada se pojavljuje u ulozi indikatora, preklopljeni operator mora biti primenljiv i na konstantne objekte. Da bi se to obezbedilo i da bi se istovremeno omogućilo dvostruko preklapanje druga verzija realizuje se kao *konstantna metoda*. Za naš opšti slučaj klase *K* druga verzija imala bi okvirni oblik

```
tip K::operator [](parametri) const {...}
```

Inače, ako su parametri dve verzije identični, kompajler će ipak uspeti da ih raspozna, jer rezervisana reč *const* takođe učestvuje u identifikaciji verzije. Konstantna verzija biće uključena kada se indeksiranje primenjuje na konstantan objekat (uključujući i konstantan objekat-argument neke funkcije!), a u ostalim slučajevima uključuje se prva verzija.

Kao kratku ilustraciju preklapanja operatora "[]" prikazaćemo kostur klase *String* što sadrži string dužine $n \leq 256$ znakova kojima se pristupa putem indeksa sa rasponom vrednosti od 0 do $n-1$.

Klasa *String* sadrži dva podatka-člana: aktuelnu dužinu stringa *n* i njegov sadržaj koji se nalazi u znakovnom nizu sa nazivom *text*. Skraćeni prikaz klase izgleda ovako:

```
#define N_MAX 256

class String {
private:
  int n;
  char text[N_MAX];
public:
. . . . . .
char& operator [](int);
char operator [](int) const;
};
```

```
char& String::operator [](int i) {return text[i];}   //verzija 1
char String::operator [](int i) const {return text[i];}   //verzija 2
```

Nekoliko primera primene:

```
void prikazStringa(const String& s) {
  cout << "Sadrzaj stringa:  ";
  for(int i=0; i<duzina(s); i++) cout << s[i] << " ";   // verzija 2
  cout << endl;
}
String s; char ch; int i, j;
. . . . . .
ch= s[j];   // verzija 1
s[5]= 'A';   // verzija 1
s[i]= s[j];   // verzija 1
prikazStringa(s);
```

Operator "->" preklapa se operatorskom metodom, iako se praktično uvek koristi u osnovnom obliku (tj. ne preklapa se). Rezultat koji vraća mora da bude neki od pokazivačkih tipova. Ako se uopšte preklapa, svrha bi mogla da bude sprovođenje nekih provera ispravnosti dereferenciranja.

6.2.8. Preklapanje operatora new i delete

Što se tiče preklapanja ova dva operatora zadovoljićemo se konstatacijom da se *new* i *delete* mogu preklopiti. U praksi se umesto preklapanja koristi kombinacija osnovnih operatora: *new* sa konstruktorom odnosno *delete* sa destruktorom koji se pri izvođenju *delete* nad dinamičkim objektom uključuje automatski. Čitaoce koji su zainteresivani za detalje preklapanja *new* i *delete* upućujemo na [77].

* * *

Završićemo ovaj odeljak kratkim osvrtom na najvažnije preporuke i propise vezane za praktične aspekte mehanizma preklapanja.

Pre svega, u praksi se pojavljuju dve gotovo disjunktne vrste klasa. Prvu čine klase u kojima dominiraju metode uz poneki preklopljeni operator (najčešće operator dodele "="). To su klase nastale kao apstrakcija entiteta, a tipičan primer je ranije opisana klasa sa modelom semafora. Nazvaćemo ih ***metodski orijentisanim*** klasama. Kod druge vrste klasa, koje ćemo zvati ***operatorski orijentisane*** klase, običnih metoda osim konstruktora i destruktora praktično nema, a zasnovane su na preklopljenim operatorima i slobodnim prijateljskim funkcijama. Radi se o

klasama koje nastaju kao realizacija tipa podataka, gde se objekti koriste u *domenu realizacije*, a kao mehanizam za implementiranje tipova podataka (primeri: klase *Complex* i *String*).

Pri korišćenju mehanizma preklapanja preporučljivo je pridržavati se sledećih načela:

1. Kao prvo i najvažnije, treba poštovati pravilo očuvanja semantike
2. Operatore treba preklopiti tada kada se očekuje intenzivna upotreba većeg broja operacija u istom izrazu, jer tada korišćenje operatora umesto metoda pozitivno utiče na čitljivost izraza. Ako su z, a, b i c članovi klase *Complex*, sigurno je lakše pročitati (i napisati!) z=a+b*c nego z=a.Add(b.Multiply(c)).
3. Klasa treba da bude ili metodski orijentisana (ako je po prirodi apstrakcija entiteta) ili pak operatorski orijentisana (ako predstavlja objektnu realizaciju tipa podataka). Mešovite klase nisu preporučljive.
4. Kao što smo videli, u nekim slučajevima preklapanje operatora zahteva kreiranje lokalnih objekata na steku. Ako su memorija i, naročito, vreme kritični faktori može se razmišljati o odustajanju od preklapanja.
5. Suviše često preklapanje operatora (zapaženo je da se to dopada početnicima) može da izazove konfuziju kod klijenta-programera koji dolazi u situaciju da se više ne snalazi šta u kojoj klasi znači neki operator. Pridržavanje pravila očuvanja semantike ublažava ovaj problem, ali ga ipak ne eliminiše.

6.3. KOERCITIVNI POLIMORFIZAM (KONVERZIJA TIPOVA)

Koercitivni (prinudni) polimorfizam nosi takav naziv jer se njegovom primenom promenljiva, objekat ili argument rutine primoravaju da privremeno promene ponašanje ne menjajući, pri tom, osnovni tip. Inače, koercitivni polimorfizam u procedurnim jezicima nije novost: javljao se u mešovitim izrazima još u fortranu, pod nazivom implicitna (automatska) konverzija tipa. S obzirom na to da je koercitivni polimorfizam i u modernijim programskim jezicima zadržao naziv implicitna odnosno eksplicitna konverzija tipa, u daljem tekstu koristićemo uglavnom ove termine.

Implicitna (automatska) konverzija standardnih tipova intenzivno se primenjuje u C/C^{++}. Sreće se u mešovitim izrazima kao i prilikom poziva potprograma. Neka su *i* i *r* celobrojna i realna promenljiva i neka je *p(t)* funkcija sa parametrom realnog tipa. Tada svi programski jezici dozvoljavaju mešovite izraze oblika npr. *i+r*, kao i poziv funkcije oblika *p(i)*. I u jednom i u drugom slučaju desiće se ista stvar: trenutna vrednost celobrojne promenljive *i* (ne i ona sama!) biće *in situ* konvertovana u realnu da bi moglo da se obavi odgovarajuće sračunavanje. Programski jezik C ide znatno dalje od toga dozvoljavajući sve mo-

guće konverzije između osnovnih tipova. Ono što ovde treba podvući jeste činjenica da do konverzije dolazi na licu mesta i samo pri sračunavanju izraza odnosno pozivu potprograma. Van tog konteksta promenljive zadržavaju osnovni tip (promenljiva *i*, dakle, ostaje i dalje celobrojna).

Druga vrsta konverzije tipa je *eksplicitna konverzija tipa*, poznata pod imenom *type cast* ("zaobilaženje tipa"), gde se konvertovanje vrednosti u zadati tip zahteva eksplicitno, posebnim konstruktom. Kod programskog jezika C, eksplicitna konverzija standardnih tipova obavlja se gotovo bez ikakvih ograničenja, uz primenu standardnih operatora konverzije. I kod eksplicitne konverzije važi isto pravilo kao i kod implicitne: vrši se samo na zahtevanom mestu a van njega nema uticaja.

Obe vrste konverzije postoje i u objektnom programiranju, gde je ponovo C^{++} znatno bogatiji mogućnostima od drugih programskih jezika.

6.3.1. Implicitna konverzija

Implicitna ili automatska konverzija u objektnom delu C/C^{++} vrši se *posredstvom konstruktora*, pri čemu se mogu konvertovati kako standardni tipovi tako i instance druge klase. Sve što je neophodno jeste uključivanje odgovarajućeg konstruktora u klasu u koju se vrši konverzija (tj. u odredišnu klasu).

Neka su dati klasa A i tip T koji nije klasa. Za obezbeđivanje implicitne konverzije iz tipa T u klasu A potrebno je u klasu A uključiti konstruktor sa parametrom tipa T:

```
class A {
. . . . . .
  A(T t) {...}
. . . . . .
};
```

Ako sada definišemo slobodnu funkciju ili metodu *f* sa formalnim parametrom iz klase *A*, tj.

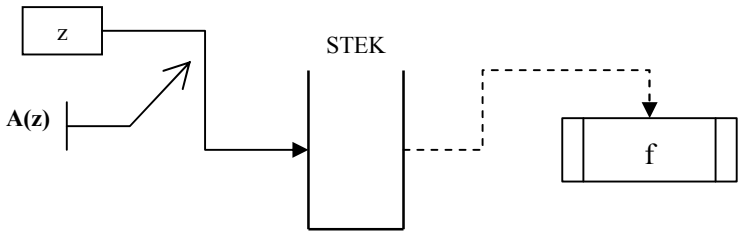
```
tip f(A a, ...) {...}
```

tada se ona može aktivirati i sa argumentom *z* koji je tipa *T*:

```
T z;
. . . . . .
f(z);
```

Prilikom aktiviranja f(z) prenos argumenta *z* vrši se posredstvom konstruktora, tako

što se pre početka rada funkcije izvrši konstruktor *A(z)* sa parametrom z produkujući tako rezultat koji je objekat klase *A* (slika 6.3). Za ovu vrstu konverzije kažemo da je implicitna ili automatska zato što se pri pozivu funkcije *f* konstruktor *A(z)* uključuje automatski, samo na osnovu konstatacije da je stvarni parametar *z* tipa T.



Slika 6.3

Primer 6.2. Jednostavni konstruktor klase *Complex*

```
Complex(double rl=0, double img=0): r(rl), i(img) {}
```

u zajednici sa preklopljenim operatorima pokazuje se kao moćno sredstvo za formulisanje kompleksnih izraza sa mogućnošću kombinovanja objekata klase sa svim baznim tipovima. Tako se, recimo, može pisati

```
Complex c = 5;     // Operator = je preklopljen.
                   // Celobrojna konstanta 5 se prvo konvertuje u tip double,
                   // a zatim konstruktorom u (5.0, 0.0)
. . . . . .
c= 10;             // Isto
```

Da bismo tačno razjasnili sadejstvo konverzije i preklopljenih operatora posmatraćemo sledeći segment:

```
Complex a, b(2,3);
. . . . . .
a= b+1;
```

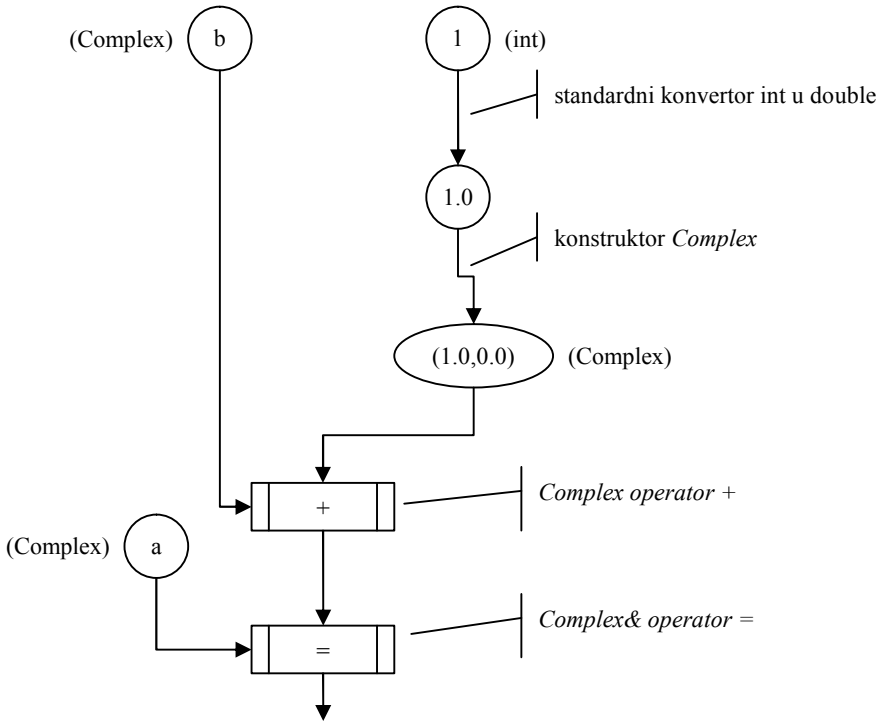
koji je - da podvučemo odmah - regularan. Pre svega, izraz

```
a= b+1
```

ima stvarno značenje

```
a.operator =(operator +(b,1))
```

tako da je sled događaja pri izvršenju naredbe *a= b+1;* kao na slici 6.4.



Slika 6.3

Prilikom aktiviranja operatorske funkcije "+" ustanovljava se da stvarni parametar nije objekat klase *Complex* i traži se konstruktor koji će izvršiti konverziju celobrojne jedinice u klasu *Complex*. Kako takvog nema, bira se postojeći konstruktor za konverziju iz tipa *double* u *Complex*, pošto se prethodno izvrši standardna konverzija celobrojne jedinice u realni oblik 1.0.

Implicitna konverzija izvodljiva je i kada su u pitanju dve klase. Neka su date klase *X* i *Y* i neka je potrebno obezbediti implicitnu konverziju iz *Y* u *X*. To se postiže uključivanjem odgovarajućeg konstruktora u odredišnu klasu *X*, s tim da njegov parametar pripada klasi *Y*, a može da bude objekat sa prenosom po vrednosti ili referenca sa prenosom po adresi (ali ne oba istovremeno) [52]. Dakle,

```
class X {
. . . . . .
  X(Y &y) {...}
```

[52] Standardna praksa je referenca, tj. prenos po adresi

```
. . . . . .
};
```

Neka je *m* slobodna funkcija (isto važi i za metodu) sa prototipom

tip m(X x);

Ako se funkcija *m* pozove sa argumentom *y* iz klase *Y*, tj . ako se napiše

m(y)

prilikom prenosa parametra *y* izvršiće se konverzija u klasu *X* posredstvom navedenog konstruktora, po istom postupku kao u slučaju konverzije tipa u klasu.

6.3.2. Eksplicitna konverzija

Eksplicitna konverzija iz nekog tipa ili klase u datu klasu obavlja se istim mehanizmom kao i implicitna: primenom konstruktora iz ciljne klase. Prema tome, ako je potrebno vršiti eksplicitnu konverziju iz tipa *T* odnosno klase *B* u klasu *A* ovu poslednju treba snabdeti konstruktorima čiji su prototipovi *A(T);* odnosno *A(B&);*. Eksplicitna konverzija promenljive *t* tipa *T* u klasu *A* dobija se konstruktom *A(t)*. Isto, ali za instancu *b* klase **B** ostvaruje se konstruktom *A(b)*. Uočimo da za eksplicitnu konverziju iz tipa-klase u datu klasu služi potpuno isti konstruktor kao i za implicitnu. U klasi *Complex* navedeni konstruktor primenjuje se za eksplicitnu konverziju iz tipa *double*. Primer:

```
Complex z; double d;
. . . . . .
z= Complex(d);
```

Eksplicitna konverzija iz date klase u tip realizuje se posebnim *operatorom konverzije* koji nema tip, a po imenu se poklapa sa ciljnim tipom. Konverzija iz klase *X* u tip *T* vrši se operatorom konverzije koji je član klase *X*:

```
class X {
. . . . . .
  operator T() {... konverzija ...}
. . . . . .
};
```

Operator konverzije

- nema tip

- po imenu se slaže sa tipom u koji se vrši konverzija
- nema parametre i
- vraća vrednost tipa T (zato se ni ne navodi tip operatora).

Sama konverzija izvodi se izrazom koji odgovara standardnim operatorima eksplicitne konverzije. Ako instanciramo klasu X naredbom *X a;* tada se eksplicitna konverzija u tip T dobija izrazom

(T) a

pri čemu je, iz nekog razloga, moguće pisati i T(a).

Neka je potrebno klasu *Complex* snabdeti (ne baš neophodnim) konvertorom u tip *double* i to tako da se pod konverzijom podrazumeva izdvajanje realnog dela. Tada konvertor ima oblik

```
class Complex {
. . . . . .
  operator double() {return r;}
. . . . . .
};
```

Iskoristićemo ovaj primer da ukažemo na moguću grešku koja nastaje usled nesaglasja obostrane konverzije između klase i tipa (tzv. "cirkularna konverzija"), koja se ogleda u potencijalnoj koliziji tipova-klasa u mešovitim izrazima. Pretpostavimo da u klasi *Complex* postoji navedeni konvertor u tip *double*, kao i više puta pominjani konstruktor koji igra ulogu konvertora iz tipa *double* u klasu *Complex*. Pretpostavimo, takođe, da klasa sadrži ranije opisanu operatorsku funkciju "+" za sabiranje kompleksnih brojeva. Tada se za izraz

z + 1.5

gde je *z* instanca klase *Complex*, ne može ustanoviti tip, jer je nejasno da li *z* treba gornjim konvertorom *operator double()* pretvoriti u *double* ili pak konstantu 1.5 konstruktorom *Complex* pretvoriti u odgovarajući objekat!

6.4. KOMPLETNA KLASA *COMPLEX* (PRIMER 6.3)

Objedinimo komponente klase *Complex* koje smo opisali u ovom poglavlju. Klasu smo dopunili, ilustracije radi, i konstantnim objektom CIM koji ima prirodu imaginarne jedinice, kao i kompleksnom eksponencijalnom funkcijom.

```
/*************************************************************
```

```
                INKAPSULIRANA KLASA SA KONSTRUKTORIMA,
              DESTRUKTORIMA, KOOPERATIVNIM FUNKCIJAMA,
                  PREKLOPLJENIM OPERATORIMA I
                   KOMPLEKSNIM KONSTANTAMA

        - Prijateljske funkcije:
           re  realni deo
           im  imaginarni deo
           conj  konjugovanje
           mod  odredjivanje modula
           arg  odredjivanje argumenta
           exp  eksponencijalna funkcija

        - Preklopljeni operatori:
           =  osnovna dodela
           +  unarno +
           -  unarno -
           +  kao operatorska funkcija
           -  kao operatorska funkcija
           *  kao operatorska funkcija
           /  kao operatorska funkcija
          +=  preko "this"
          ==  kao operatorska funkcija
 prefix  ++  preko "this", kao konstantan objekat
postfix  ++  preko privremenog objekta, kao konstantan objekat

        - Kompleksna konstanta CIM = (0,1)

 Sada je nivo apstrakcije toliki da se ne vidi cak ni da je u pi-
 tanju klasa. Znak dodeljivanja, funkcije i operacije spolja
 sasvim podsecaju na standardni tip podataka!

Datoteka: COMPLEX2.HPP
**************************************************************/

#ifndef COMPLEX2_HPP
  #define COMPLEX2_HPP

class Complex {
private:
```

```
  double r, i;
public:
// Konstruktor i destruktor
  Complex(double rl=0, double img=0): r(rl), i(img) {}
  ~Complex() {};
// Prijateljske funkcije
  friend double re(const Complex &z) {return z.r;}
  friend double im(const Complex &z) {return z.i;}
  friend Complex conj(const Complex &);
  friend double mod(const Complex &);
  friend double arg(const Complex &);
  friend Complex exp(const Complex &);
// Preklopljeni operatori
  Complex& operator =(const Complex &);
  friend Complex operator +(const Complex &);
  friend Complex operator -(const Complex &);
  friend Complex operator +(const Complex &, const Complex &);
  friend Complex operator -(const Complex &, const Complex &);
   friend Complex operator *(const Complex &, const Complex &);
  friend Complex operator /(const Complex &, const Complex &);
  Complex& operator +=(const Complex &);
  friend int operator ==(const Complex &, const Complex &);
  const Complex &operator ++() {r++; i++; return *this;}
  const Complex operator ++(int);
};

// Kompleksna konstanta (konstantni objekat)
extern const Complex CIM;     // imaginarna jedinica

#endif
```

```
/*************************************************************

                        TELO KLASE COMPLEX

Datoteka: COMPLEX2.CPP
*************************************************************/

#include "complex2.hpp"
#include <math.h>
```

```
Complex conj(const Complex &z) {
  return Complex(z.r,-z.i);
}
Complex exp(const Complex &z) {
  double temp = exp(z.r);
  return Complex(temp*cos(z.i),temp*sin(z.i));
}
// "=" mora da bude operatorska metoda
// vracanje vrednosti preko "this"
Complex& Complex::operator =(const Complex &z) {
  r= z.r; i= z.i;
  return *this;
}
Complex operator +(const Complex &z) {
  return z;
}
// Vracanje vrednosti preko bezimenog objekta
Complex operator -(const Complex &z) {
  return Complex(-z.r, -z.i);
}
// Vracanje vrednosti preko bezimenog objekta
Complex operator +(const Complex &z, const Complex &t) {
  return Complex(z.r+t.r, z.i+t.i);
}
Complex operator -(const Complex &z, const Complex &t) {
  return Complex(z.r-t.r, z.i-t.i);
}
Complex operator *(const Complex &z, const Complex &t) {
  return Complex(z.r*t.r-z.i*t.i, z.r*t.i+z.i*t.r);
}
Complex operator /(const Complex &z, const Complex &t) {
  double temp = t.r*t.r + t.i*t.i;
  return Complex((z.r*t.r+z.i*t.i)/temp, (z.i*t.r-z.r*t.i)/temp);
}
Complex& Complex::operator +=(const Complex &z) {
  r+= z.r; i+= z.i;
  return *this;
}
// Vracanje int vrednosti preko operatorske funkcije
```

```
int operator ==(const Complex &z, const Complex &t) {
  return (z.r==t.r) && (z.i==t.i);
}
// Postfiksni operator. Vracanje preko privremenog objekta
const Complex Complex::operator ++(int k) {
  Complex w(r,i);
  r++; i++;
  return w;
}
double mod(const Complex &z) {
  return sqrt(pow(z.r,2) + pow(z.i,2));
}
double arg(const Complex &z) {
  return (z.r == 0 && z.i == 0) ? 0 : atan2(z.i,z.r);
}
const Complex CIM(0,1);     // imaginarna jedinica
```

Zapazimo da, kako se radi o dosledno izvedenoj operatorskoj klasi, realizacija ne sadrži ni jednu običnu metodu - sve su zamenjene bilo preklopljenim operatorima bilo prijateljskim funkcijama. S obzirom na činjenicu da su kontrola pristupa i pravilo očuvanja semantike poštovani do kraja, klijent-programer će teško primetiti da *Complex* nije običan tip nego da je realizovan korišćenjem objektnog programiranja. Postojanje "konstante" CIM koja predstavlja imaginarnu jedinicu omogućuje definisanje "konstanti" kompleksnog "tipa". Na primer, kompleksni broj 1+2i u klasi *Complex* prikazujemo kao 1+2*CIM. Pomoću ovakvih "konstanti" može se izvršiti čak i inicijalizacija kompleksne "promenljive" *c* oblika npr.

```
Complex c = 2+3*CIM;
```

koja se postavlja na vrednost 2+3i. Evo još nekoliko primera upotrebe klase *Complex* kao realizacije tipa kompleksnog broja:

```
#include "complex2.hpp"
Complex a, b = 1.6+9.3*CIM, d = 5, x, y, z;     // a,x,y,z su jednaki (0,0)
double r, teta;
. . . . . .
a= 1+CIM;
a+= b;
x= ++a;
y= b++;
```

```
z= (x+y)*(b-d)+Exp(y);
r= mod(z);
teta= arg(z);
z= conj(z);
a= b= c= exp(1+CIM);
d= 10;
```

Sledeća slobodna funkcija služi za sumiranje *n* elemenata kompleksnog niza *cArray*:

```
Complex sum(Complex cArray[], int n) {
  Complex s; int j;
  for(s=0, j=0; j<n; j++) s+= cArray[j];
  return s;
}
```

Predlaže se čitaocu da objasni zašto se eksponencijalna funkcija može nazvati isto kao i standardna funkcija *exp* iz modula *math.h*.

6.5. VARIJABILNI STRING (PRIMER 6.4)

Još jedan primer operatorski orijentisane klase je klasa *DynString* koja realizuje string promenljive dužine. Da bi se omogućila promena dužine u toku izvršenja programa, deo objekta koji sadrži string smešta se u dinamičku memoriju (hip). Ovde dajemo nešto širi (mada ne i potpun) sadržaj ove klase. Čitalac može sam dopuniti klasu još nekim operatorima (npr. "!="). Konstruktor klase inicijalizuje string zadatom vrednošću ili, ako nje nema, formira prazan string. Dvostruko preklopljen operator "[]" služi za indeksiranje stringa, tj. za pristup pojedinačnim znakovima. Prva varijanta je referenca zato da bi imala osobine *lvrednosti*, tj. da bi indeksirani znak mogao da dobije vrednost. Druga varijanta je konstantna metoda za indeksiranje konstantnih stringova. Preklopljeni operator "+=" predstavlja nadodavanje stringa na kraj datog stringa. Operator "==' služi za poređenje dva stringa, "+" je konkatenacija, a prijateljska funkcija *len* izdaje aktuelnu dužinu stringa.

```
/*************************************************************
                    INTERFEJS KLASE DYNSTRING

Datoteka: DSTRING.HPP
*************************************************************/
#ifndef DSTRING_HPP
  #define DSTRING_HPP
```

```
class DynString {
private:
  int n;
  char *text;
public:
  DynString(const char[]="");     // Konstruktor
  DynString(const DynString&);    // Konstruktor kopije
  ~DynString() {delete[] text, text= 0;}   // Destruktor
  char& operator [](int);
  char operator [](int) const;
  DynString& operator =(const DynString&);
  DynString& operator +=(const DynString&);
  friend int operator ==(const DynString&, const DynString&);
  friend DynString operator +(const DynString&, const DynString&);
  friend int len(const DynString&);
};

#endif
```

```
/*************************************************************

                    TELO KLASE DYNSTRING

Datoteka: DSTRING.CPP
*************************************************************/
#include "dstring.hpp"

DynString::DynString(const char inpStr[]) {
  n= 0; while(inpStr[n]!='\0') n++;
  text=new char[n];
  for(int i=0; i<n; i++) text[i]= inpStr[i];
}
DynString::DynString(const DynString& rhs[53]) {
  n= rhs.n;
  text= new char[n];
  for(int i=0; i<n; i++) text[i]= rhs.text[i];
```

[53] Obratiti pažnju na to da parametar konstruktora kopije mora biti *konstantna* referenca. Stariji prevodioci tolerisaće i običnu referencu, dok noviji neće.

```
}
char& DynString::operator [](int i) {
  return this->text[i];
}
char DynString::operator [](int i) const {
  return this->text[i];
}
DynString& DynString::operator =(const DynString& rhs) {
  if(this==&rhs) return *this;
  delete[] text; n= rhs.n;
  text= new char[n];
  for(int i=0; i<n; i++) text[i]= rhs.text[i];
  return *this;
}
DynString& DynString::operator +=(const DynString& rhs) {
  int i; char *temptxt=text;
  text= new char[n+rhs.n];
  for(i=0; i<n; i++) text[i]= temptxt[i];
  for(i=0; i<rhs.n; i++) text[n+i]= rhs.text[i];
  n+= rhs.n;
  delete[] temptxt;
  return *this;
}
int operator ==(const DynString& s1, const DynString& s2) {
  if(s1.n != s2.n) return 0;
  for(int i=0; i<s1.n; i++) if(s1.text[i] != s2.text[i]) return 0;
  return 1;
}
DynString operator +(const DynString& s1, const DynString& s2) {
  DynString temp; int i,j=0;
  temp.n= s1.n + s2.n; temp.text= new char[temp.n];
  for(i=0; i<s1.n; i++) temp.text[i]= s1.text[i];
  while(i<temp.n) {temp.text[i]= s2.text[j++]; i++;}
  return temp;
}
int len(const DynString& s) {
  return s.n;
}
```

Nekoliko primera upotrebe klase *DynString* (koja, kao i *Complex* realizuje "tip podataka"):

```
#include "dstring.hpp"
String s1="efg", s2, s3;     // Vrednosti: s1 = "efg"   s2 = s3 = ""
char c;
int m;
. . . . . .
s2= "12345";
s1= s2;     // Vrednost s1 = "12345"
s1+= "ABC";     // Vrednost s1 = "12345ABC"
c= 'q';
s1[0]= 'p'; s1[2]= c; s[7]= s[0];     // Sada je s1 = "p2q45ABp"
s3= s1+s2;     // Vrednost s3 = "p2q45ABp12345"
s3+= "*";     // Vrednost s3 = "p2q45ABp12345*"
m= len(s3);     // Vrednost m = 14
```

7. VEZE IZMEĐU KLASA. NASLEĐIVANJE.

Drugi princip Alena Keja, naveden u odeljku 2.2, propisuje da je "program skup objekata koji zadaju poslove jedan drugom, putem slanja poruka". Ovom treba dodati i to da pomenuti skup objekata, u opštem slučaju, čine instance različitih klasa, tako da stoji činjenica da se objektni program gradi na bazi mnoštva različitih klasa. Osnova za ovu tvrdnju nije samo činjenica da je realni svet složen, sastavljen od mnoštva različitih, međusobno zavisnih entiteta nego, još više, da je naša *percepcija* sveta isto tako složena.

Već smo napomenuli da se softverski inženjer ne bavi realnim stvarima nego *pojmovima* o stvarima. On modeluje takve pojmove recimo klasama i objektima, ali se ne može zaustaviti na tome, jer se pojmovi nalaze u raznim ***odnosima***, privremenim i trajnim, pojedinačnim i opštim, koji takođe spadaju u opis modelovanog sistema. Kada odnosi između individualnih pojmova za stvar nisu sporadični i slučajni i kada se smatraju relevantnim za funkcionisanje sistema, prenose se na odgovarajuće klasne pojmove. Na primer, svaki voz (tj. individualni pojam voza) sastoji se od vagona i lokomotive (takođe individualni pojmovi), što daje za pravo da se između klasnih pojmova VOZ s jedne i VAGON i LOKOMOTIVA s druge strane, konstatuje odnos tipa "ima", "sastoji se", "sadrži" itd. odnosno, u obrnutom smislu, "pripada", "ulazi u sastav" ili "je deo". Isto tako, postoje odnosi koji su apriorno vezani za klase entiteta kao što je, recimo, odnos generalizacije (GRIZLI je MEDVED).

Pošto su klase objekata modeli klasnih pojmova za stvar, kao što su objekti modeli individualnih pojmova za stvar, neminovno je da se njihovi međusobni odnosi preslikavaju na modele, stvarajući tako ***veze*** između klasa objekata odnosno njihovih instanci. Odnosi između klasnih pojmova i, shodno tome, veze između odgovarajućih klasa objekata imaju najrazličitiju semantiku. Vrlo dobar primer objektnog sistema je grafički korisnički interfejs koji se sastoji od čitavog niza objekata (prozori, ekranski tasteri, meniji, dijalozi i još mnogo toga) koji pak nisu nezavisni nego se nalaze u raznovrsnim vezama: meniji pripadaju prozorima, ekranski tasteri i ulazne linije pripadaju dijalozima, dijalozi su posebne vrste prozora, konkretni prozori izvode se iz opšte šeme koja ima elemente zajedničke za sve prozore itd.

Odnosi između pojmova za stvari toliko su semantički raznoliki da je njihovo sistematsko modelovanje odgovarajućim vezama ravno čišćenju Augijevih štala, dakle zadatak težak ali izvodljiv. Ključ za rešenje je u pridevu "relevantan" koji, kao i u slučaju klase i objekta, ide uz odnos, a koji širom otvara vrata za uvođenje (sic!) apstrakcije kao sredstva za uprošćavanje modela.

Relevantni odnosi generišu veze koje, mada i dalje semantički raznovrsne, podležu klasifikaciji s tim da se, kao što ćemo videti, stavka "ostalo" ne može izbeći. Razni autori različito klasifikuju veze: Mejer [82] ustanovljava samo dve vrste veza, Martin [2] tri; u UML ima ih znatno više. Navedeni klasifikacioni sistemi, na sreću, nisu u koliziji te ćemo na ovom mestu pokušati da ih usaglasimo. Mejerov način klasifikacije je vrlo uopšten i obuhvata samo dve vrste veza. To su

- ***Klijentske veze*** u kojima klase-klijenti koriste usluge klasa-snabdevača i
- ***Nasleđivanje*** koje modeluje odnos generalizacija - specijalizacija.

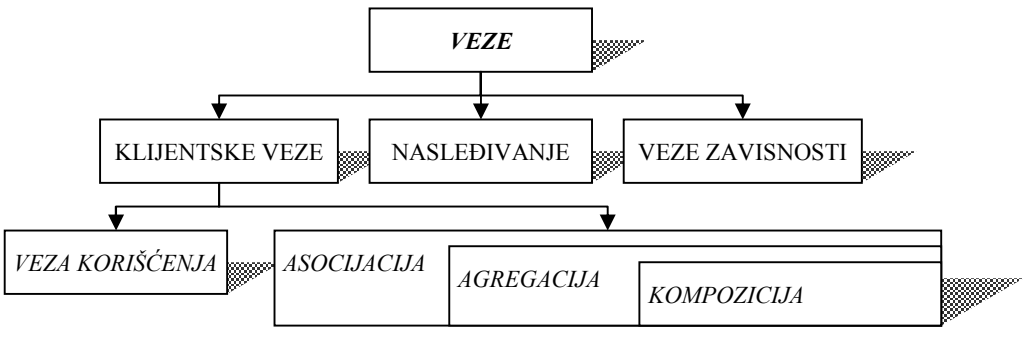
Da bi se klasifikacija zaokružila, klijentskim vezama i nasleđivanju dodajemo

- ***Veze zavisnosti***, preuzete iz UML, što predstavljaju skup prilično raznorodnih veza (videti u [13]), a čija je zajednička karakteristika da nisu ni klijentske niti veze nasleđivanja. Drugim rečima, veze zavisnosti u dobroj meri odgovaraju stavci "ostalo" u klasifikacionom sistemu.

Klijentske veze dalje delimo na

- ***asocijaciju***, vezu koja obuhvata širok spektar logičkih relacija između semantički nezavisnih klasa
- ***agregaciju***, vezu koja modeluje odnos celina - deo
- ***kompoziciju*** koja modeluje odnos posedovanja
- ***vezu korišćenja*** koja označava da neke operacije klijenta ne mogu biti izvedene bez posredstva snabdevača.

Klasifikacija veza prikazana je na slici 7.1



Slika 7.1

U nastavku ćemo razmotriti navedene tipove veza, s tim što ćemo vezu nasleđivanja ostaviti za kraj jer se proteže i na sledeće poglavlje.

7.1. KLIJENTSKE VEZE

Za klasu *A* reći ćemo da je klijent klase *B* ako klasa *B* eksplicitno učestvuje u definiciji sadržaja klase *A*. Klijentska veza između klase *A* i klase *B* podrazumeva da se *B* može pojaviti u sklopu definicije nekih objekata-članova klase *A*, zatim kao deo signature (zaglavlja, prototipa) njenih funkcija-članica ili pak i jedno i drugo.

Dalja podela klijentskih veza, iako opšteprihvaćena, nije sasvim uobičajena. Naime, između asocijacije, agregacije i kompozicije s jedne i veze korišćenja s druge strane, postoji jasna razlika: dok se prve tri izvode iz pojedinačnih odnosa između pojmova iste klase poslednja je implementacione prirode jer znači da su objekti korišćene klase parametri moteda klase koja koristi.

Zajednička karakteristika asocijacije, agregacije i kompozicije jeste to što se uspostavljaju na osnovu pojedinačnih veza između instanci klasa-učesnika. Potiču od istih takvih odnosa, tj. od odnosa između klasnih pojmova koji se izvode iz odnosa između odgovarajućih individualnih pojmova. Recimo, na konceptualnom nivou, pojam NASTAVNIK je u odnosu PREDAJE sa pojmom PREDMET (asocijacija) jer je to samo drukčija formulacija trivijalne tvrdnje "nastavnik predaje predmet". Klasni pojam VAGON je sa klasnim pojmom VOZ u odnosu PRIPADA (agregacija) jer vagoni ulaze u sastav voza. Pojam TAČKA i pojam DUŽ su u odnosu JE DEO (kompozicija) zato što su temena duži, koja su njen integralni deo, po prirodi tačke. U formalno-matematičkom smislu, agregacija je specijalan slučaj asocijacije, a kompozicija specijalan slučaj agregacije, što, međutim, ne znači da je asocijacija, kao opštiji slučaj, važnija od druge dve veze. Naime, semantika agregacije i naročito kompozicije je takva da su ove veze po važnosti u najmanju ruku ravnopravne sa asocijacijom. Štaviše, jedna vrsta kompozicije, i to ona koja povezuje objekat sa njegovim objektima-članovima ("podobjektima"), uz nasleđivanje, predstavlja najvažniji tip veza između klasa.

7.1.1. Asocijacija

Asocijacija je naziv za semantičku vezu između inače nezavisnih klasa (prema [3] i [27]) izvedenu iz veza individualnih instanci tih klasa. U ovoj (neformalnoj) definiciji naglasak je na sintagmama "semantička veza" i "nezavisne klase". Iz prve sledi da asocijacija nije ma kakva veza dobijena proizvoljnim grupisanjem instanci, već da ima značenje, oličeno u nazivu asocijacije. Sintagma "nezavisne klase" u ovom kontekstu interpretira se kao "nezavisne apstrakcije", tj. kao apstrakcije koje se mogu formulisati i razumeti kao logička celina. Tako, klase *Nastavnik* i *Predmet* povezuju se asocijacijom sa semantikom PREDAJE ili PREDAJE_GA ako se čita u suprotnom smeru. Inače, očigledno se radi o nezavisnim apstrakcijama jer se odgovarajući klasni pojmovi mogu opisati i razumeti nezavisno jedan od drugog.

Asocijacijom se mogu povezati i više od dve klase: na primer, klase

Nastavnik, *Predmet*, *NastavnaGrupa* i *Sala* povezujemo u kvaternarnu relaciju sa nazivom PREDAVANJE. Inače, daleko najčešći slučaj je asocijacija reda 2 koja povezuje dve klase.

I sama asocijacija može biti klasa sa sopstvenim sadržajem. Gornjoj asocijaciji možemo dodeliti atribut *Termin* i operacije *OčitajTermin* i *PromeniTermin*, te je modelovati kao klasu sa nazivom *Predavanje*.

Konačno, postoje i rekurzivne asocijacije koje vezuju klasu sa samom sobom.

U opštem slučaju asocijacija reda *n* predstavlja šemu veza čiji su primerci (pojave) uređene n-torke semantički ravnopravnih objekata iz klasa-učesnika u vezi. Neka su K_1, ..., K_n (ne obavezno različite) klase-učesnice u vezi tipa asocijacije. Tada vezi asocijacije pridružujemo matematičku relaciju

$$r_n(t) \subseteq O_1(t) \text{ x } ... \text{ x } O_n(t)$$

gde su $O_1(t)$, ... $O_n(t)$ skupovi instanci klasa redom K_1, ..., K_n u trenutku t. Veoma važnu osobinu asocijacije predstavljaju tzv. kardinaliteti učesnika u asocijaciji. ***Kardinalitet*** učesnika K_i u asocijaciji $r_n(t)$ jeste skup celih nenegativnih brojeva koji pokazuju mogući broj pojavljivanja različitih objekata iz skupa O_i u n-torkama u kojima su ostale komponente iste. Ako asocijaciju predstavimo kao višestruko preslikavanje

$$f_i: \ O_1 \text{ x} ... \text{x } O_{i-1} \text{ x } O_{i+1} \text{ x} ... \text{x } O_n \ \rightarrow \ \mathcal{P}(O_i)$$

gde je $\mathcal{P}(O_i)$ partitivni skup O_i tada kardinalitet klase K_i propisuje broj mogućih veličina skupova iz $f_i(O_1x...xO_{i-1}xO_{i+1}x...xO_n)$. Ako, na primer, u asocijaciji PREDAJE kardinalitet učesnika *Predmet* iznosi 1 do 3, to znači da nastavnik može da predaje jedan, dva ili tri predmeta. Često se pojavljuje potreba da kardinalitet zadamo rasponom od donje do gornje granice pri čemu je gornja granica proizvoljna, tj. znači "više". U navedenom primeru, kardinalitet učesnika *Predmet* možemo promeniti na "1 ili više" što znači da nastavnik može predavati jedan ili više predmeta.

Slika 7.2

U UML binarna asocijacija prikazuje se punom linijom koja povezuje klase-učesnice, slika 7.2. Veza se snabdeva podacima o nazivu asocijacije, smeru čitanja

naziva (crna strelica), kardinalitetima i tzv. ulogama klasa u asocijaciji[54]. Za oznaku kardinaliteta koristi se sintaksa paskalskog tipa skupa (izuzev simbole [i]) jer kardinalitet i jeste skup. Kardinaliteti se zadaju konkretnim brojevima, opsezima "od-do" ili kombinacijom. Ako je gornja granica oblika "više" upotrebljava se simbol *. Nekoliko primera kardinaliteta:

1	jedan
*	više
0..10	nula do deset
1..*	jedan ili više
1,2,10..20	jedan, dva ili deset do dvadeset.

Uloga klase u asocijaciji jeste tumačenje svrhe postojanja svakog pojedinačnog učesnika u asocijaciji. Naročito je važna kod rekurzivnih asocijacija jer kod njih je objekat klase *K* u ulozi α u vezi sa drugim objektom *iste klase* u ulozi β.

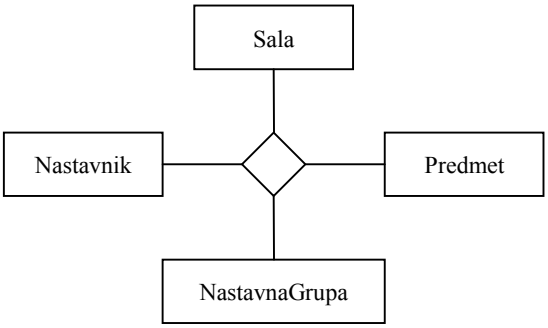
Na slici 7.3 prikazane su dve asocijacije: jedna povezuje klase *Nastavnik* i *Predmet* i čita se stilom "Nastavnik predaje predmet". Pri tom, dozvoljava se da nastavnik predaje više predmeta ili da (trenutno) ne predaje nijedan predmet; s druge strane propisuje se da svaki predmet mora imati bar jednog nastavnika, a ne može ih biti više od dva. Druga asocijacija povezuje instance klase *Predmet*. U jednom smeru, znači da položen ispit iz nekog predmeta može da uslovljava slušanje drugih, odnosno, u suprotnom, da je slušanje nekog predmeta uslovljeno položenim ispitima iz drugih.

Slika 7.3

Kardinalitet učesnika *Predmet* mogli smo predstaviti i oblikom 0..*, sa istim značenjem.

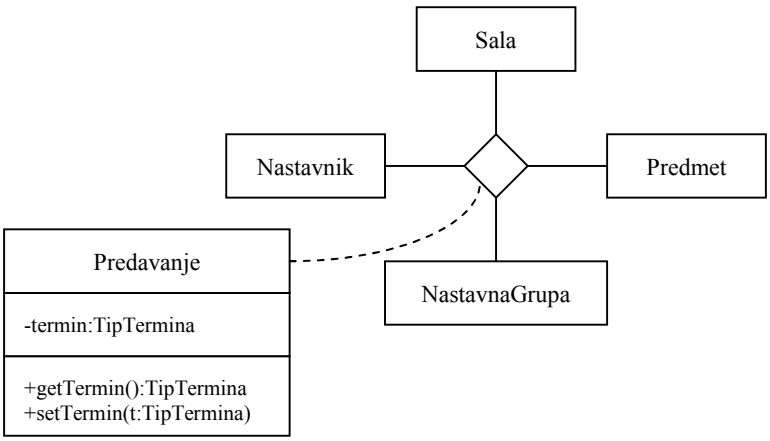
Asocijacija višeg reda prikazuje se rombom koji povezuje sve učesnike. Primer je dat na slici 7.4.

[54] To nije sve. O punom skupu oznaka videti npr. u [13].



Slika 7.4

U slučaju da asocijacija ima osobine klase, za simbol asocijacije vezuje se isprekidanom linijom simbol odgovarajuće klase-veze (slika 7.5).



Slika 7.5

Veza asocijacije u upotrebi je davno pre pojave objektnog programiranja i karakteristična je za projektovanje i realizaciju baza podataka, gde se realizacija veze prepušta sistemu za upravljanje bazom podataka. Jedan od zadataka takvog sistema je očuvanje tzv. ***referencijalnog integriteta*** koji je jedna od bitnih odlika svake asocijacije. Radi se o prostoj činjenici da uklanjanje objekta zahteva uklanjanje svih veza koje vode od njega ka drugim objektima i, naročito, od drugih objekata ka njemu. Ako nastavnik napusti školu potrebno je a) ukloniti odgovarajući objekat b) ukloniti sve veze ka predmetima koje predaje (ne i predmete!) i c) ukloniti veze koje vode od predmeta koje predaje ka nastavniku. U suprotnom, moglo bi da se dogodi da se iz nekog od objekata koji nisu uklonjeni pokuša pristup uklonjenom objektu (npr. spisak nastavnika koji predaju zadati predmet ne sme sadržati uklon-

jeni objekat klase *Nastavnik*). Inače, najveći problem predstavlja upravo stavka *c* jer je daleko jednostavnije locirati veze od objekta nego veze ka njemu.
Asocijacija se fizički *realizuje* na više načina:

1. Kao jednostruko spregnuta lista u okviru svakog učesnika. Lista sadrži adrese objekata sa kojima je dati objekat u vezi asocijacije. Ovakav način realizacije karakterističan je za binarne asocijacije sa kardinalitetom tipa "više".
2. Ako je veza binarna sa kardinalitetima m..n gde su m i n konstante, tada se veza realizuje kao statički niz adresa sa rasponom indeksa od m do n.
3. Asocijacija dva ili više učesnika može se realizovati i u vidu tabele u kojoj svaka vrsta sadrži identifikatore (npr. adrese ili ključeve) svih učesnika u jednoj pojavi veze. Ovaj način je istovremeno i najopštiji, jer funkcioniše za sve oblike asocijacije.

7.1.2. Agregacija

Formalno, agregacija je posebna vrsta asocijacije, sa semantikom "ima" (engl. HAS), "sadrži", "obuhvata", "je deo" (engl. PARTOF), "pripada" i sl. Suštinski, pak, agregacija modeluje odnos ***celina-deo*** koji je toliko važan i specifičan da se, bez obzira na formalizam, agregacija shvata kao posebna vrsta veze i tretira ravnopravno sa asocijacijom. U agregaciji celina ima poseban naziv: ***agregat***. Evo nekoliko primera agregacije:

- Nastavna grupa SASTOJI SE od studenata
- Vagon PRIPADA vozu
- Voz SADRŽI jednu ili dve lokomotive
- Muzički stub IMA zvučnike.

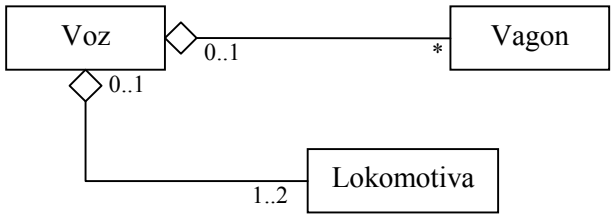
Katkad je teško ustanoviti da li je veza, naročito kada je opisana rečju "ima", agregacija ili asocijacija. Na primer, forma odnosa "vojnik ima oružje" sugeriše vezu agregacije kao model. To bi, međutim, značilo da je oružje deo vojnika što se, bar na prvi pogled, čini besmislenim. No, ako pojam "vojnik" shvatimo kao "naoružani čovek" pokazuje se da agregacija kao model ipak nije lišena smisla te da odnos možemo modelovati na oba načina - i agregacijom i asocijacijom.

Inače, u [83] je data kratka rasprava o tome zašto se agregacija odvaja od ostalih vrsta asocijacije. Kao najvažniji razlog podvlači se tzv. *propagacija* koja se često susreće upravo kod ovog tipa veze. Pod propagacijom podrazumeva se konsekutivna primena neke operacije na mrežu objekata i to počev od fiksiranog objekta (a to je agregat). Na primer, operacija kopiranja primenjuje se tako što počinje od agregata i sistematski se izvršava nad svim delovima (kako se kaže, propagira kroz njih).

Agregacija se u UML predstavlja punom linijom koja počinje belim rom-

bom na strani klase koja odgovara agregatu, slika 7.6. Ostali elementi su isti kao kod asocijacije, s tim što se naziv pojavljuje ređe (jer je uvek oblika "sadrži", "pripada" i sl.). Konkretan primer agregacije prikazan je na slici 7.7.

Slika 7.6



Slika 7.7

Agregacija se realizuje na isti način kao i asocijacija.

7.1.3. Kompozicija

Kompozicija jeste poseban slučaj agregacije, ali dovoljno poseban da se čak u UML označava nešto drukčije. Posmatrajmo, za početak, dva primera agregacije. Voz ima vagone, pri čemu su i voz i vagoni modelovani objektima. Isto tako, trougao ima temena gde su, ponovo, i temena i sam trougao objekti (teme pripada klasi npr. *Tačka*). Oba slučaja odgovaraju agregaciji gde uloge celine imaju voz odnosno trougao, a delova vagoni odnosno temena. Međutim, vagoni kao objekti postoje pre formiranja voza i mogu se uključivati u voz i isključivati iz njega. Takođe, destrukcija matičnog objekta klase voz ne podrazumeva destrukciju vagona niti obrnuto. S druge strane, uništavanje instance klase *Trougao* nužno izaziva destrukciju svih njegovih temena. Razlika očito postoji i ogleda se u tome što je uslov za egzistenciju temena postojanje odgovarajućeg trougla, što ne važi za voz i njegove vagone.

- ***Kompozicija*** je vrsta agregacije u kojoj delovi ne mogu egzistirati izvan celine.

U našem primeru odnos trougla i njegovih temena modeluje se vezom kompozicije. Semantika kompozicije je "poseduje" (engl. OWNS). Klasa koja predstavlja celinu nosi naziv ***vlasnik*** (owner), a klasa koja odgovara delu zove se ***komponenta***. Isti termini upotrebljavaju se i za klase i za pojedinačne objekte. Kao što smo rekli, definiciona osobina kompozicije jeste to što instanca klase-komponente ne može

postojati pre kreiranja vlasnika niti opstaje posle njegovog uništenja. Drugim rečima, životni vek (period između kreiranja i uništavanja) komponente sadržan je u životnom veku vlasnika. Na primer, između klase *Dokument* i klase *Pasus* postoji takva veza, jer pasus može da egzistira samo kao deo dokumenta. Isto važi i za klase *Pravougaonik* i *Duž* što se najlakše ustanovljava na osnovu činjenice da uništavanjem pravougaonika moramo uništiti i njegove stranice što pripadaju klasi *Duž*. Ova karakteristika poznata je pod nazivom egzistencijalna zavisnost, [83].

Neka su K_1 i K_2 klase i neka su $O_1(t)$ i $O_2(t)$ skupovi njihovih instanci u trenutku t. Neka je r(t) binarna relacija oblika

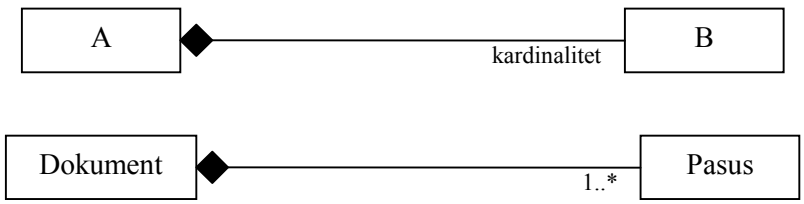
$$r(t) \subseteq O_1(t) \times O_2(t)$$

Ako za svaki uređeni par $(o_1,o_2) \in r(t)$ u svakom trenutku važi da je životni vek instance o_2 u celosti sadržan u životnom veku o_1 i da važi $(o,o_2) \in r(t)$ samo ako je $o=o_1$, tada ćemo r nazivati *relacijom parcijalne egzistencijalne zavisnosti*. Ako je $r(t) = O_1(t) \times O_2(t)$ tada se r zove relacija totalne egzistencijalne zavisnosti.

Za klasu K_2 kažemo da egzistencijalno zavisi od klase K_1 ako postoji odgovarajuća relacija parcijalne egzistencijalne zavisnosti. Uočimo da klasa može egzistencijalno zavisiti od više drugih klasa. Takođe, ako r(t) nije relacija totalne egzistencijalne zavisnosti postoje instance zavisne klase K_2 koje nisu u relaciji sa instancama klase K_1. Štaviše, iz (parcijalno) zavisne klase mogu se generisati i potpuno nezavisne instance.

Očigledno, svaka klasa-komponenta egzistencijalno zavisi od klase-vlasnika pri čemu zavisnost ne mora biti totalna. Tako, na primer, instanca klase *Tačka* može, u ulozi temena, biti komponenta objekta klase *Četvorougao*, druga instanca može biti komponenta objekta klase *MaterijalnaTačka* i treća egzistirati izolovano, kao tačka u Dekartovom koordinatnom sistemu.

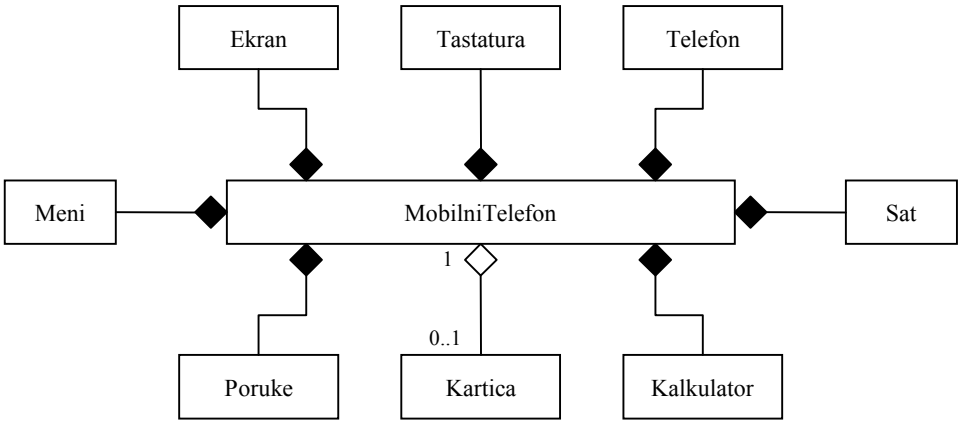
Simbol za kompoziciju u UML sličan je simbolu za čistu agregaciju (onu koja nije kompozicija). Razlika je u tome što je romb popunjen (slika 7.8).



Slika 7.8

Na slici 7.9 prikazan je još jedan primer primene kompozicije. Iskoristićemo ga da ukažemo na potencijalni problem pri izboru vrste veze u toku postupka modelovanja. Postavimo, naime, pitanje da li je veza između mobilnog telefona i

npr. ekrana zaista kompozicija, odnosno da li ekran može da egzistira izvan instance telefona? Odgovor bi bio i potvrdan i odričan, a ključ treba tražiti u *domenu problema* jer - ne zaboravimo - klasa *MobilniTelefon* je samo *model*. Ako akcije u domenu problema obuhvataju montiranje i razmontiranje telefona tada je logično modelovati vezu kao agregaciju. U suprotnom radi se o tipičnoj kompoziciji. Podsetimo se, ograničavanje na domen problema umesto na Univerzum upravo i služi tome da se izbegnu aporije ovakve vrste ("kakvu boju ima mleko u mraku?", "kakve je boje mrak?").



Slika 7.9

Inače, za agregaciju i kompoziciju postoje i alternativni termini, redom veza po referenci i veza po vrednosti. Po našem mišljenju izrazi nisu najsrećnije odabrani jer stvaraju impresiju da se agregacija realizuje putem reference (pokazivača), a kompozicija putem uključivanja cele komponente u memorijski prostor vlasnika. Pored toga što prejudicira realizaciju u fazi modelovanja, ovakav utisak jednostavno nije tačan jer se komponenta realizuje na oba načina. Štaviše, neki jezici (java, Object Pascal) ni nemaju drugu mogućnost za pristup objektima nego preko referenci odn. pokazivača.

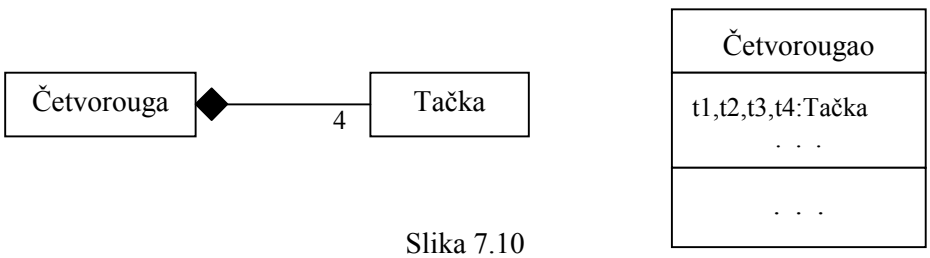
Kao i kod asocijacije i agregacije i kod kompozicije postoje posebni slučajevi. Podsetimo se, za svaku kompoziciju važi da je životni vek komponente sadržan u životnom veku vlasnika. To nikako ne znači da svaka komponenta ima obavezno samo jednog vlasnika. Posmatrajmo objekte iz klasa *Čitalac* i *Knjiga* u domenu bibliotečkog informacionog sistema. Kada čitalac pozajmi knjigu kreira se objekat klase *Pozajmica* koji povezuje čitaoca i knjigu. Pri tom, ovaj objekat jeste komponenta i klase *Čitalac* i klase *Knjiga*, jer ne može postojati ako ne postoji par objekata (čitalac, knjiga) iz odgovarajućih klasa. Iako nisu česte, ovakve kompozicije sa više vlasnika očigledno postoje.

Posebni slučajevi kompozicije su tzv. rep polje i podobjekat. Komponenta koja ima tačno jednog vlasnika nosi naziv *rep polje* ("representation field") i karakteristična je po tom što se izmena stanja rep polja može izvršiti samo preko vlasnika, što dalje znači da se i samo stanje objekta može promeniti isključivo dejstvom na taj objekat. Inače, kod opšte kompozicije to ne mora biti tako: ako komponenta ima dva vlasnika tada se iz jednog vlasnika može direktno promeniti njeno stanje menjajući indirektno i stanje drugog vlasnika.

Možda najznačajnija vrsta kompozicije jeste veza "biti podobjekat" koja postoji između objekta i njegovog objekta-člana[55]. Prvo, lako se uočava da između objekta-člana i njegovog matičnog objekta postoji egzistencijalna zavisnost. Primerice, objekti-članovi koji predstavljaju temena četvorougla egzistiraju samo dok postoji matični objekat - četvorougao. Sledstveno, svaki objekat-član je istovremeno i komponenta ili, bolje, veza "biti objekat član" jeste veza kompozicije. U ovom slučaju, međutim, važi i obrnuto: iz četvorougla se ne može ukloniti ni jedno teme, a da on ostane četvorougao! Iz toga, dalje, sledi da između objekta i njegovog podobjekta postoji *obostrana egzistencijalna zavisnost* i da se njihovi životni vekovi poklapaju. Prema tome,

- Objekat-član (podobjekat) jeste komponenta koja ima jednog jedinog vlasnika i čiji se životni vek poklapa sa životnim vekom vlasnika.

Navedena definiciona osobina objekta-člana utemeljena je na jednoj opštijoj karakteristici a to je da objekat-član u kognitivnom smislu predstavlja *integralni deo vlasnika*. Jezikom pojmova, objekat-član ili podobjekat jeste model *fragmenta* definisanog u odeljku 2.1.2 prilikom razmatranja konceptualnog pogleda na klasu i objekat.



Slika 7.10

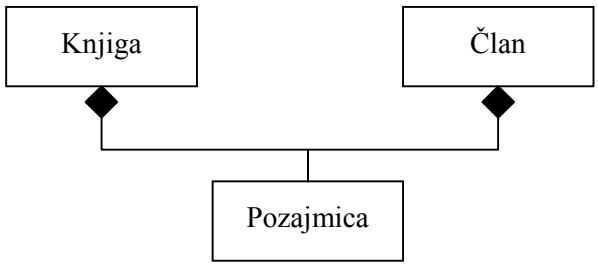
Veza podobjekta u UML može se prikazati na tri načina: zadavanjem konstantnog kardinaliteta na strani komponente (slika 7.10 levo) ili navođenjem podobjekta u opisu klase (slika 7.10 desno), a nije zabranjena ni kombinacija.

Razmotrimo, na kraju, i pitanje realizacije. Uopšte, komponenta se realizuje

1. putem pokazivača (u nekim jezicima poput jave tzv. reference) za sve vrste komponenata osim objekata-članova i
2. neposrednim uključivanjem u memorijski prostor vlasnika (ako to jezik dozvol-

[55] Drugi naziv za objekat-član jeste *podobjekat* [82]. Koristićemo oba termina.

java) ili opet putem pokazivača za objekte-članove.
Očigledno, povezivanje putem pokazivača moguće je u svim slučajevima. Pri tom, specijalne karakteristike kompozicije regulišu se metodama, a posebno konstruktorima i destruktorima. Kod komponente koja može imati više vlasnika mora se viditi računa o referencijalnom integritetu. Jedna takva situacija prikazana je na slici 7.11.



Slika 7.10

Kada član biblioteke vrati pozajmljenu knjigu neophodno je destruisati odgovarajuću instancu klase *Pozajmica*. To, međutim, zahteva poništavanje obe veze sa tom instancom: i vezu instance *Knjiga* kao i vezu instance *Član*. Cilj je očuvanje referencijalnog integriteta, tj. sprečavanje pristupa destruisanom objektu klase *Pozajmica* iz bilo kojeg (bivšeg) vlasnika.

Dalje, ako je u pitanju *rep* polje mora se obezbediti da se pokazivač na komponentu može naći samo u okviru vlasnika i da se destrukcija komponente mora izvršiti najkasnije destruktorom ako ne i nekom metodom.

Konačno, ako se radi o objektu-članu mora se obezbediti njegova alokacija u okviru konstruktora vlasnika i dealokacija unutar destruktora vlasnika.

Uključivanje podobjekta u memorijski prostor vlasnika najbolji je način njegove realizacije, kada to programski jezik omogućuje (u C^{++} može, dok, recimo, java i Object Pascal to ne dozvoljavaju). Uključivanjem podobjekta u memorijski prostor vlasnika automatski se rešava pitanje obostrane egzistencijalne zavisnosti jer se objekat-član konstruiše i destruiše isključivo zajedno sa vlasnikom.

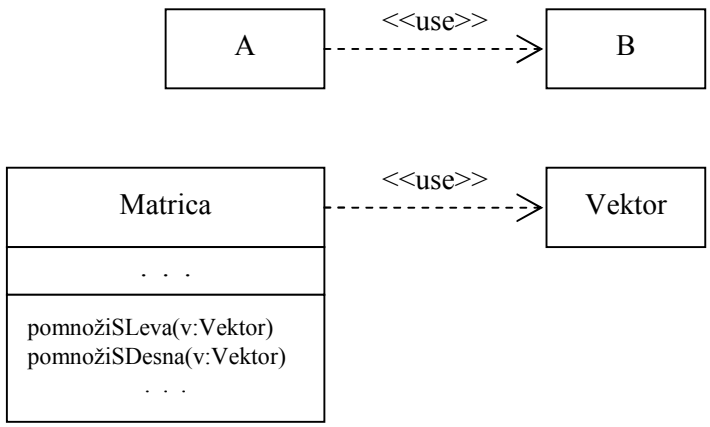
7.1.4. Veza korišćenja

Veza korišćenja (engl. USING, [3]) je logički najlabavija od klijentskih veza. Između klase *A* i klase *B* postoji veza korišćenja od *A* ka *B* ako bar jedan formalni parametar ili rezultat neke od metoda klase *A* pripada klasi *B*. Klasa *Matrica*, na primer, koristi klasu *Vektor* ako u njoj postoje metode *pomnožiSleva* ili *pomnožiSdesna* čiji su formalni parametri vektori. Tipične karakteristike veze korišćenja su:

- Veza nema posebnu semantiku. Preciznije, sve veze korišćenja imaju semantiku "koristi usluge".
- Konkretizacija veze obično je kratkotrajna i svodi se na izvršenje odgovarajuće metode.

I još nešto: klasa *A* i klasa *B* mogu istovremeno biti i u vezi asocijacije i u vezi korišćenja. Na primer, klasa *Nastavnik* jeste u vezi asocijacije sa klasom *Predmet*. Ako pak u nju uključimo metodu *dodajPredmet(p:Predmet)* (što je za očekivanje) time formiramo i vezu korišćenja. U načelu, možemo prikazati obe veze ili se zadržati samo na vezi asocijacije jer potonja obezbeđuje postojanje obe klase što je, u stvari, svrha uvođenja veze korišćenja.

U UML veza korišćenja prikazuje se isprekidanom linijom usmerenom od klijenta ka snabdevaču. Pošto sličnih veza ima još (bar je tako predviđeno u UML) i sve se prikazuju istim simbolom, veza korišćenja snabdeva se stereotipom (službenom rečju) <<use>>, slika 7.12.



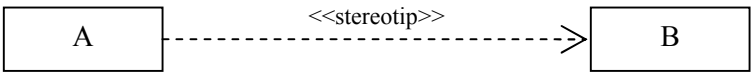
Slika 7.12

7.2. VEZE ZAVISNOSTI

Veze zavisnosti čine šaroliku grupu veza koje bi se najtačnije definisale iskazom "nisu ni klijentske veze niti nasleđivanje" ili pak "ostale veze". Zajednički im je samo simbol[56] u UML koji se poklapa sa simbolom veze korišćenja, slika 7.13.[57]

[56] Za veze zavisnosti se kaže da imaju za zajedničku karakteristiku i to što zavisni element modela ne može postojati bez prisustva nezavisnog, ali to važi i za kompoziciju.

[57] U stvari, UML tretira vezu korišćenja kao jednu od veza zavisnosti, ali zbog posebnih karakteristika veze korišćenja pridržavaćemo se Mejerove sistematizacije.



Slika 7.13

Bliže su određene stereotipom. Nekoliko primera, [13, 86]:

- Veza kooperativnosti; dozvola da jedan element pristupi sadržaju drugog bez obzira na zaštitu (stereotip <<friend>>)
- Veza između operacije i njenog parametra (stereotip <<parameter>>)
- Veza između specifikacije i realizacije u kojoj je zavisni element realizacija (implementacija) nezavisnog (stereotip <<realize>>)
- Veza između pošiljaoca i primaoca poruke (stereotip <<send>>)
- Veza instanciranja u kojoj je zavisni element instanca nezavisnog (stereotip <<instanceOf>>). Ne odnosi se isključivo na objekte, jer klasa može da bude instanca metaklase, tj. klase klasa.
- Veza pozivanja; metoda jedne klase poziva metodu druge klase (stereotip <<call>>)

7.3. NASLEĐIVANJE

Neko je duhovito primetio da je razvoj tehnologije u dobroj meri posledica čovekove lenjosti. Programeri se u tom pogledu nimalo ne razlikuju od ostalog sveta - naprotiv![58] Ideal programera je da softver nipošto ne razvijaju *ab ovo* nego da taj posao obavljaju prilagođavanjem, doterivanjem i uklapanjem već napravljenog. Ako raspolaže paketom za manipulisanje matricama kojem nedostaje samo inverzija, programer će na svaki način gledati da tu operaciju nekako *doda* već postojećem softveru[59] izbegavajući tako ponovno pisanje napisanog. Uostalom, i sami viši programski jezici imaju sličnu namenu: umesto da se piše niz asemblerskih instrukcija formira se jedna naredba višeg programskog jezika.

Višekratna upotreba (ponovno korišćenje, engl. "reuse") u smislu pukog korišćenja gotovih modula, nije dovoljna za kvalitetno programiranje. Praksa, po pravilu, zahteva određena podešavanja: izmene algoritama, dodavanje algoritama i podataka, pa čak i modifikaciju podataka, a sve to nosi zajednički naziv *proširivost* (videti odeljak 4.2). Mehanizam proširivanja koji, u sadejstvu sa višekratnom upotrebom, obezbeđuje takav, savršeniji vid upotrebe softvera nosi naziv ***nasleđivanje*** (engl. "inheritance").

- Pod nasleđivanjem podrazumevamo preuzimanje sadržaja nekih klasa u

[58] Neverne Tome treba da pregledaju programersku literaturu i prebroje pojave reči "lakše" koja se po pravilu odnosi na napor programera i nikako na korisnika.
[59] Pošto *i za nju* nađe gotovu verziju.

datu klasu, uz mogućnost modifikacije preuzetih članova i dodavanja novih.

Klasa (može ih biti i više) od koje se preuzima sadržaj zove se *predak*, *osnovna klasa*, *natklasa* ili *klasa-davalac*[60]. Klasa koja prima sadržaj nosi naziv *potomak*, *izvedena klasa*, *potklasa* ili *klasa-primalac*. Odmah skrećemo pažnju na izuzetno važnu karakteristiku nasleđivanja, a to je da

- nasleđivanje ne podrazumeva *nikakve izmene u kodu predaka*.

Štaviše, vrlo je preporučljivo postulirati da, pri realizaciji nasleđivanja, izvorni kod predaka uopšte nije na raspolaganju. Najvažnije ***osobine nasleđivanja*** su:

1. Mogućnost *modifikacije* preuzetog sadržaja. U načelu, mogu se menjati svi članovi pretka: i podaci i objekti i funkcije. U praksi, gotovo isključivo modifikuju se otvorene funkcije-članice, tj. *metode*. Modifikacija metode zove se ***redefinisanje*** (engl. "overriding") i izvodi se tako što se metoda iznova definiše (tj. napiše pod istim imenom) u sklopu potomka. Podvucimo da se prilikom redefinisanja metode mora strogo voditi računa o pravilu očuvanja semantike koje, u datom slučaju, propisuje da se može menjati samo algoritam - svrha metode ostaje ista. Odstupanje od ovog pravila uvek ima nepredvidive posledice!
2. Mogućnost proširivanja, dodavanjem novog sadržaja - u načelu, i podataka i objekata i funkcija. Osobina ***proširivosti*** jeste, uz mogućnost višekratne upotrebe, deo definicije pojma modularnosti (videti odeljak 4.2).
3. Kao matematička relacija, nasleđivanje ima osobinu *tranzitivnosti* sa značenjem: ako klasa *C* nasleđuje klasu *B* i klasa *B* nasleđuje klasu *A* tada klasa *C* nasleđuje klasu *A*. U nekim slučajevima potrebno je terminološki razlikovati neposrednog pretka (potomka) od ostalih, pa neposrednog pretka zovemo roditelj (roditeljska klasa), a neposrednog potomka naslednik.
4. Nasleđivanje može biti ***jednostruko*** (preuzimanje osobina od jedne roditeljske klase) ili ***višestruko*** (ako ih ima više), pri čemu je razlika veća nego što se to na prvi pogled može učiniti. Mnogi programski jezici (paskal, Object Pascal, pa i java) nemaju višestruko nasleđivanje. C^{++} ga ima. S obzirom na suštinske razlike između jednostrukog i višestrukog nasleđivanja, ovo poslednje razmotrićemo u posebnom poglavlju. U nastavku, koncentrisaćemo se samo na jednostruko nasleđivanje, a višestruko nasleđivanje razmatraćemo u posebnom odeljku.
5. Naslednik se formira ili uvođenjem ograničenja ili dodavanjem semantike ili, naravno, na oba načina. Ako na klasu *Pravougaonik* postavimo ograničenje da su sve stranice iste dužine, mehanizmom nasleđivanja dobijamo klasu *Kvadrat*. S druge strane, ako klasi *Pas* proširimo semantiku do-

[60] U pitanju je prevod, te se mora tolerisati neslaganje u rodu.

davanjem podataka o rasi, nagradama, broju pedigrea i sl., nasleđivanjem dobijamo klasu *RasniPas*.

6. Po definiciji, svaka klasa nasleđuje samu sebe (tj. nasleđivanje kao relacija ima osobinu *refleksivnosti*). To takođe znači da se sve osobine klase mogu smatrati nasleđenim, bilo od pretka bilo od same sebe.

Vratimo se definiciji nasleđivanja. Udubimo li se malo, zapazićemo da ono, za razliku od ostalih veza, ima dva, podjednako važna, aspekta: tehnološki (proširivost) i konceptualni (odnos generalizacije), slika 7.14.

Slika 7.14

Tehnološki aspekt nasleđivanja

Prvo, tu je ***tehnološki aspekt*** koji kvalifikuje nasleđivanje kao *sredstvo za realizaciju proširivosti* kao komponente modularnosti softvera. Osobina tranzitivnosti daje podlogu za stvaranje hijerarhije klasa povezanih nasleđivanjem, takve da se zajedničke osobine klasa na istom hijerarhijskom nivou izdvajaju u posebnu klasu koja postaje njhov zajednički predak. Umesto da klase-naslednice implementiraju zajedničke osobine svaka za sebe one ih prosto preuzimaju od opšteg pretka. Na primer, svi (ekranski) prozori imaju okvir, naslov, koordinate i mogućnost pomeranja, te je pogodno formirati klasu koja poseduje te osobine realizovane putem njenih članova. Sve klase-naslednice preuzimaju ih direktno, u gotovoj formi i samo dodaju sopstvene članove. Na taj način iz pomenute klase možemo formirati klasu prozora sa diskretnom i kontinualnom promenom veličine, iz nje klasu prozora sa menijem i statusnom linijom itd.

U poslednje vreme pojavila se potreba za uvođenjem dve podvrste nasleđivanja. Jedna je *nasleđivanje interfejsa*, a druga *nasleđivanje implementacije*. Prva podvrsta podrazumeva preuzimanje interfejsa osnovne klase uz dodavanje implementacije. Za ostvarivanje nasleđivanja interfejsa neophodno je da odgovarajući jezik poseduje poseban oblik klasa, tzv. apstraktne klase kojima nedostaje deo ili čak cela implementacija metoda (videti sledeću glavu). Nasleđivanje interfejsa posebno je interesantno u distribuiranim sistemima kada se za implementaciju koriste različiti jezici, dok je interfejs izveden na jednom, zajedničkom jeziku. Nasleđivanje implementacije jeste suprotan pristup: naslednik preuzima telo klase od pretka dopunjujući ga sopstvenim interfejsom. Ostvaruje se primenom sredstava

zaštite (*private*, *public*), i to tako što se preuzeti interfejs pretka[61] skriva od klijenta.

Konceptualni aspekt nasleđivanja

Drugi, možda i važniji, aspekt nasleđivanja jeste konceptualni aspekt. Ovde se ima u vidu priroda nasleđivanja kao veze koja modeluje odnos generalizacija-specijalizacija. Naime, prihvatanjem sadržaja pretka, naslednik faktički preuzima njegovu strukturu i, naročito, ponašanje[62], tako da se faktički ponaša kao predak sa dodatim specifičnostima. Drugim rečima, naslednik je specijalizovan predak odnosno, obrnuto, predak je generalizacija naslednika (i svih ostalih potomaka). Ako je klasa *Kvadrat* izvedena iz klase *Pravougaonik* tada se svaka njena instanca ponaša i kao instanca klase *Pravougaonik*, uz određene specifičnosti poput npr. mogućnosti upisivanja kruga koje kod pretka nema. Nasleđivanje kao veza ima semantiku "je", "jeste" (engl. ISA od "is a ..."). Na primer, *NeutronskaZvezda* je *Zvezda*, *PravougliTrougao* je *Trougao*, *Automobil* je *DrumskoVozilo*, *SaveDialog* je *Dialog* itd.

Pokazaćemo sada da se do bîti nasleđivanja dolazi upravo sledeći njegov konceptualni aspekt. Naime, definicija nasleđivanja koju smo naveli tehnički je orijentisana, jer pokazuje kako se ono realizuje, ne ulazeći posebno u suštinu ove fundamentalne veze. Osobine nasleđivanja - mogućnost proširivanja sadržaja, redefinisanje, tranzitivnost, višestrukost i refleksivnost - tretirane su kao preskriptivne, bez traganja za nekim dubljim smislom. U nastavku ćemo pokazati da se sve navedene osobine nasleđivanja, uključujući čak i blisku vezu sa klasičnom formom definicije kao takve, pojavljuju kao posledica pojmovne (konceptualne) prirode klase i objekta.

Pre svega, u svakom suvislom misaonom sistemu pojmovi ne mogu egzistirati izolovano - oni se nalaze u najrazličitijim *odnosima*. Odnos koji je od interesa za naša razmatranja jeste odnos tzv. subordinacije. Podsetimo se, pojmovi se odlikuju sadržajem što predstavlja skup njihovih relevantnih osobina i opsegom koji takođe predstavlja skup njegovih vrsnih pojmova (pojmova prvog nižeg reda). Neka su ta dva skupa označeni sa *sadržaj(X)* i *opseg(X)* gde je *X* pojam. Sledi ***definicija subordinacije***:

- Neka su *A* i *B* dva pojma. Za pojam *B* kaže se da je subordiniran pojmu *A*, u oznaci *B sub A*, ako važi

$$\text{sadržaj(A)} \subseteq \text{sadržaj(B)}$$
$$\text{opseg(B)} \subseteq \text{opseg(A)}.$$

Lako uočavamo da su odnos subordinacije i obrnuti odnos ***superordinacije*** samo drugi nazivi za redom specijalizaciju i generalizaciju. Štaviše, termini "generalizacija" i "specijalizacija" suvišni su, a morali smo ih navesti jer su u širokoj upot-

[61] Ne zaboravimo da naslednik *nolens-volens* mora da preuzme i interfejs pretka.
[62] Naravno, pod uslovom da se poštuje pravilo očuvanja semantike.

rebi. Na primer, pojam BUDILNIK subordiniran je pojmu SAT, zato što svaki budilnik sadrži sve što sadrži svaki sat, ali i alarmni mehanizam kojeg neki satovi nemaju. S druge strane, u opseg sata spadaju svi budilnici, ali i oni satovi koji nemaju alarmni mehanizam. Pojam TIGAR subordiniran je pojmu MAČKA jer svaki tigar ima sve osobine mačke i neke koje nema svaka mačka, a postoje i mačke koje nisu tigrovi.

Upoređujući relaciju subordinacije sa nasleđivanjem lako uočavamo da je paralela potpuna: klase koje su vezane relacijom nasleđivanja modeluju pojmove koji su u odnosu subordinacije. Izvedena klasa obuhvata sve članove bazne klase prosto zato što odgovarajući subordinirani pojam obuhvata kompletan sadržaj superordiniranog pojma. Istovremeno, pored date izvedene klase postoje i druge klase koje se mogu izvesti iz posmatrane natklase. Ove sasvim očigledne činjenice vode ka ***konceptualnoj definiciji nasleđivanja***:

- Nasleđivanje je veza između klasa koja modeluje odnos subordinacije između odgovarajućih pojmova.

U nastavku, razmotrićemo navedene (preskriptivne) osobine nasleđivanja i pokazati da one ili direktno proističu ili jesu posledica odnosa subordinacije.

U svetlu ovakve definicije postaje očigledno da izvedena klasa ne samo da može nego i *mora* da preuzme sadržaj bazne klase, a *može* i da ga proširi. To sledi iz proste činjenice da je sadržaj izvedene klase nadskup sadržaja bazne.

Refleksivnost i tranzitivnost relacije subordinacije neposredno slede iz istih osobina skupovne inkluzije. Pošto za ma kakve skupove X, Y i Z važi $X \subseteq X$ kao i $(Z \subseteq Y) \wedge (Y \subseteq X) \Rightarrow (Z \subseteq X)$, na osnovu definicije subordinacije pojmova sledi

$$\text{A sub A}$$
$$(\text{C sub B}) \wedge (\text{B sub A}) \Rightarrow (\text{C sub A})$$

gde su A, B i C (klasni) pojmovi.

Višestruko nasleđivanje zasniva se na činjenici da odgovarajući odnos može da postoji i između klasnih pojmova. Recimo, klasni pojam RADIO-SAT ima u sadržaj inkorporine sadržaje kako pojma RADIO, tako i pojma SAT.

Mogućnost (tačnije *neophodnost*) redefinisanja metoda ima neznatno drukčije izvorište nego prethodne stavke, jer je vezana za realizaciju klase. Naime, metoda preuzeta od bazne klase mora da zadrži semantiku osnovne verzije[63], ali česta je situacija da njena programska implementacija mora biti promenjena da bi se uskladila sa izvedenom klasom. O tome će biti još dosta reči u narednim odeljcima ovog poglavlja.

Najzad, podudarnost nasleđivanja sa arsitotelovskom formom definicije

[63] Opet posledica konceptualne definicije! Ako bi metoda promenila semantiku, to bi značilo da izvedena klasa *nije* preuzela verziju iz bazne klase.

nije slučajna: izvedena klasa odgovara pojmu što se definiše osloncem na superordinirani pojam, a ovom odgovara bazna klasa!

Nasleđivanje i klasična struktura definicije

Povezivanje nasleđivanja (izvođenja) sa odnosom među pojmovima na konceptualnom nivou, otvara još jedno pitanje: relaciju nasleđivanja prema definiciji kao takvoj. Konceptualna definicija definicije glasi [102]:

- definicija jeste iskaz kojim se nedvosmisleno određuje sadržaj jednog pojma.

Klasična struktura definicije (nazvana i aristotelovskom, videti poglavlje 2.1) predviđa da se *definiendum* (pojam što se definiše) određuje *definiensom* (ono čime se definiše) koji ima opšti oblik

definiens = genus proximum + differentia specifica

gde je *genus proximum* prvi viši rod pojmova (tzv. najbliži rod) u odnosu na *definiendum*, a *differentia specifica* deo koji opisuje razlike između *definiendum*-a i drugih pojmova iz najbližeg višeg roda. Kada, na primer, kažemo "ravnostrani trougao je trougao čije su sve stranice iste dužine" tada je

a) definiendum "ravnostrani trougao"
b) genus proximum "trougao", a
c) differentia specifica "imati stranice iste dužine".

Drugim rečima, ravnostrani trougao je trougao, ali se od ostalih trouglova razlikuje po tome što ima stranice jednakih dužina. Vidimo da je pojam ravnostranog trougla *izveden* iz pojma trougla *obogaćivanjem semantike* osobinom jednakosti stranica. Dalje nije ni potrebno ići: uporedimo li ovu kratku raspravu sa opštom raspravom o nasleđivanju dolazimo do zaključka da

- neposredno izvođenje klase B iz klase A u potpunosti odgovara postupku definisanja pojma P_B kojem odgovara klasa B iz genus-a proximum-a P_A kojem odgovara klasa A!

Ono što se u klasičnoj definiciji zove *genus proximum* u objektnoj metodologiji nosi naziv *(neposredni) predak*, a *differentia specifica* ogleda se u dodatim osobinama naslednika (koje predak nema). Uočimo i to da u načelu ne postoje prepreke da broj klasa koje igraju ulogu genus-a proximum-a može biti i veći od 1.

Očigledno, a uprkos skromnom nazivu, nasleđivanje spada u fundamentalne kategorije i daleko nadilazi ne samo objektnu metodologiju nego i računarstvo uopšte!

Kontrola pristupa

U poglavlju 4 detaljno smo razmotrili pitanje ***kontrole pristupa*** sadržaju klase. Konstatovali smo da princip skrivanja informacija zahteva da klasa, u odnosu

na bilo kakvog klijenta, ima bar dva dela: nedostupni (private) i otvoreni (public). Šta je, međutim, sa klasom-naslednicom? Da li je ona "bilo kakav klijent"? Odgovor je negativan: klasa-naslednica ne samo da nije "ma kakav" klijent nego uopšte nije klijent pretka jer ne koristi njegove usluge nego *preuzima* strukturu i ponašanje. To, uostalom, jasno sledi iz činjenice da nasleđivanje po Mejerovoj klasifikaciji nije klijentska veza i da se ne izvodi iz odnosa pojedinačnih objekata nego uspostavlja direktno između klasa. Shodno tome, logično je preispitati i način kontrole pristupa sadržaju klase od strane njenih potomaka, a posebno (neposrednih) naslednika. S obzirom na to da naslednici preuzimaju semantiku od pretka, oni jesu privilegovani u odnosu na druge klase i to mora da se odrazi na kontrolu pristupa. Naime, neki delovi klase (naročito podaci-članovi) koji su zatvoreni za "spoljni svet" ne treba da budu zatvoreni za naslednike *zato što pripadaju i naslednicima kao njihov integralni deo*. Nije stoga iznenađujuće što noviji objektni jezici imaju najmanje tri nivoa kontrole pristupa (Turbo Pascal nažalost ne, java čak četiri):

1. Nivo ***private*** zatvoren za sve klase osim matične
2. Nivo ***public*** otvoren za sve klase
3. Nivo zaštite u kojem se nalazi deo sadržaja koji su zatvoreni za sve osim za potomke. Ovaj nivo zaštite nosi naziv ***protected***.

Dakle, metode naslednika mogu pristupiti ne samo delu koji je otvoren (public) nego i nivou *protected* rezervisanom za privilegovane klase. Ukazujemo odmah na čestu grešku koja se pravi usled mešanja pojma klase i njene instance. Radi se o sledećem:

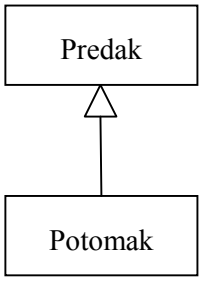
- sadržaj klase zaštićen nivoom *protected* dostupan je *metodama potomaka*, ali ne i njihovim instancama!

Ovde nema ničeg posebnog, jer isto važi i za deo *private*: taj deo dostupan je metodama matične klase i samo njima. Dakle, da rezimiramo:

- Segment *private* dostupan je samo metodama matične (tj. date) klase
- Segment *protected* dostupan je metodama matične klase i njenih potomaka
- Segment *public* dostupan je svim klasama, kao i klijentima koji nisu klase (glavni program ako ga ima, slobodni potprogrami).

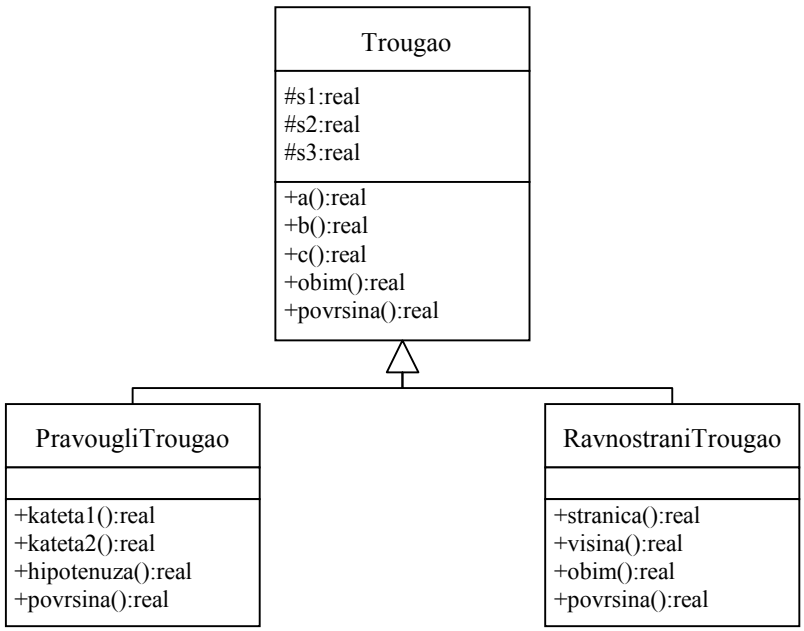
U UML, nasleđivanje se prikazuje strelicom sa trouglastim vrhom okrenutim od naslednika ka neposrednom pretku (precima), slika 7.15. Zapazimo da je strelica okrenuta od naslednika ka pretku jer se veza čita "nasleđuje".[64]

[64] Dužni smo napomenuti da za ovo ima i teorijskih razloga, vezanih za odnos tip-podtip, u koje se nećemo upuštati.



Slika 7.15

Primer 7.1. Na slici 7.16 prikazana je jednostavna hijerarhija nasleđivanja koja obuhvata predak *Trougao* i naslednike *PravougliTrougao* i *RavnostraniTrougao*.



Slika 7.16

Osnovna klasa *Trougao* ima sledeći sadržaj:

1. Podaci-članovi *s1*, *s2* i *s3* koji čuvaju vrednosti dužina stranica. Imaju nivo zaštite *protected*, tj. može im se pristupiti iz metoda klasa *Trougao*, *PravougliTrougao* i *RavnostraniTrougao* (i njihovih potomaka).
2. Metode *a()*, *b()* i *c()* za očitavanje dužina stranica.
3. Metode *obim()* i *povrsina()* za izračunavanje odgovarajućih veličina.

Klasu *PravougliTrougao* čine:

1. Članovi *s1*, *s2*, *s3*, *a()*, *b()*, *c()* i *obim()* koji su nasleđeni od pretka i zato nisu navedeni u odgovarajućem simbolu.
2. Metode *kateta1()*, *kateta2()* i *hipotenuza()* koje su dodate u klasu (nema ih u klasi *Trougao*). U osnovi rade isti posao kao i *a()*, *b()* odnosno *c()* respektivno, a uvedene su zbog toga što je uobičajeno da stranice pravouglog trougla imaju posebne nazive.
3. Metoda *povrsina()* je redefinisana istoimena metoda iz pretka.

Od svih karakteristika klase *PravougliTrougao* najinteresantnija je redefinisana metoda *povrsina()*. Pre svega, zbog pravila očuvanja semantike ona mora raditi isto što i odgovarajuća metoda iz pretka, dakle računati površinu trougla. Zašto onda uopšte postoji kada metoda *povrsina()* klase *Trougao* može da posluži u istu svrhu? Odgovor je važan, i glasi: redefinisana metoda obavlja isti posao kao osnovna, ali na drugi način, saobrazno osobinama naslednika. U našem slučaju metoda *povrsina()* klase *Trougao* koristi Heronov obrazac, dok je formula za računanje površine u klasi *PravougliTrougao* - poluproizvod dveju kateta.

Klasu *RavnostraniTrougao* sačinjavaju:

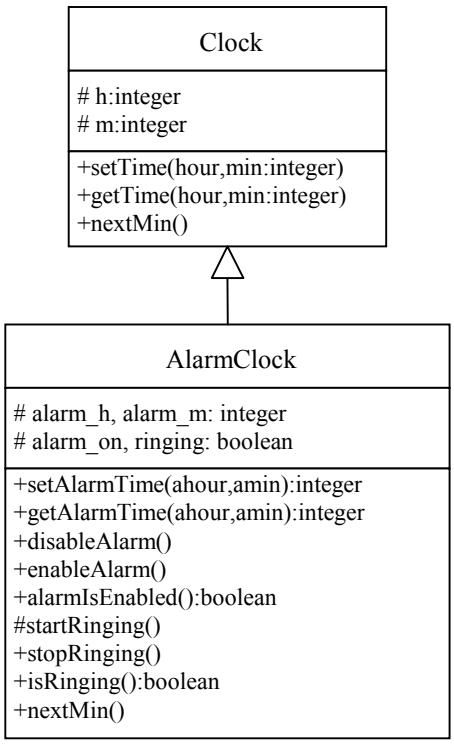
1. Članovi *s1*, *s2*, *s3*, *a()*, *b()* i *c()* gde se, kao i kod prvog naslednika, metode *a()*, *b()* i *c()* ne koriste neposredno, ali su važne zbog inkluzionog polimorfizma (sledeće poglavlje).
2. Dodata metoda *stranica()* sa istom motivacijom kao kod metoda *kateta1()*, *kateta2()* i *hipotenuza()* klase *PravougliTrougao*.
3. Metode *obim()* i *povrsina()* redefinišu se jer se kod ravnostranog trougla računaju na poseban način.
4. Dodata metoda *visina()* koja u pretku ne postoji jer proizvoljan trougao nema jedinstvenu visinu.

S obzirom na česte nedoumice, podvućićemo još jednom dejstvo zaštite:

- Podacima-članovima *s1*, *s2* i *s3* sa zaštitom tipa *protected* mogu pristupiti (1) metode klase *Trougao* (2) metode klasa *PravougliTrougao* i *RavnostraniTrougao*. Podacima-članovima *s1*, *s2* i *s3* ne mogu pristupati ni instance triju klasa niti njihovi klijenti.
- Metode klasa (sve su otvorene) mogu koristiti svi klijenti. Štaviše, instanca klase-naslednice može, ako je to potrebno, koristiti originalnu verziju redefinisane metode pretka (npr. *povrsina()*). Eksplicitni poziv originalne verzije postiže se kvalifikovanjem, u skladu sa sintaksom upotrebljenog jezika.

- Instanca klase *Trougao* može aktivirati metode *a()*, *b()*, *c()*, *obim()* i *povrsina()*.
- Instanca klase *PravougliTrougao* može aktivirati nasleđene metode *a()*, *b()*, *c()* i *obim()*, kao i sopstvene metode.
- Instanca klase *RavnostraniTrougao* može pozvati nasleđene metode *a()*, *b()* i *c()*, kao i sopstvene metode.
- Instanca klase *Trougao* ne može aktivirati metode *kateta1()*, *kateta2()*, *hipotenuza()*, *stranica()* i *visina()*.

Primer 7.2. Drugi primer nije matematički. Formiraćemo klasu *Clock* što predstavlja model sata i njenu naslednicu, klasu *AlarmClock* koja modeluje sat snabdeven alarmom, tj. budilnik. Hijerarhija dveju klasa prikazana je na slici 7.17.



Slika 7.17

Za razliku od prethodnog primera, ovde naslednik ima znatno više dodatne semantike ugrađene kako kroz nove metode tako i kroz dodatne podatke-članove. Sledi sadržaj klasa (za klasu *AlarmClock* data su objašnjenja samo za sopstvene, ne i za nasleđene metode).

Klasa *Clock*:

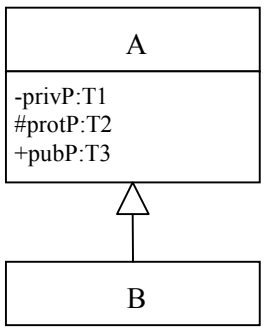
1. Podaci-članovi *h* i *m* koji su zaštićeni nivoom *protected*. Predstavljaju sate odnosno minute (sekunde nisu uključene zbog jednostavnosti)
2. Metode *setTime()* i *getTime()* za postavljanje odnosno očitavanje vremena
3. Metoda *nextMin()* za pomeranje sata za 1 minut napred.

Klasa *AlarmClock*:

1. Podaci-članovi *alarm_h* i *alarm_m* za podešavanje vremena aktiviranja alarma
2. Podatak-član *alarm_on* za indikaciju da li je alarm uključen
3. Podatak-član *ringing* koji pokazuje da li je uključen signal alarma
4. Metode za postavljanje i očitavanje vremena alarma, *setAlarmTime()* i *getAlarmTime()*
5. Metode *disableAlarm()* i *enableAlarm()* za uključivanje-isključivanje alarma (odgovaraju dugmetu ili tasteru na budilniku)
6. Metoda *alarmIsEnabled()* za proveru da li je alarm uključen
7. Metode *startRinging()* i *stopRinging()* za otpočinjanje odnosno zaustavljanje alarma ("zvonjave")
8. Metoda *isRinging()* za proveru da li je uključen signal alarma
9. Redefinisana metoda *nextMin()* koja, pored pomeranja za minut napred, proverava da li se vreme poklapa sa otpočinjanjem alarma. Ako se poklapa i ako je alarm uključen metodom *enableAlarm()* tada se aktivira signal alarma.

7.3.1. Nasleđivanje i C++

Neka su date klase *A* i *B* takve da klasa *B* nasleđuje klasu *A*, slika 7.18.



Slika 7.18

Sintaksa nasleđivanja u C++ ima sledeći oblik:

```
class B: μA {
... sadržaj klase B ...
};
```

gde je μ oznaka zaštite nasleđivanja. Na mestu simbola μ može stajati jedan od sledećih ***modifikatora zaštite***:

public (najčešće)
protected
private
(ništa) što je isto kao private

Modifikator zaštite μ određuje rezultantni nivo zaštite članova klase B. Kažemo "rezultantni" jer se stvarni nivo zaštite dobija kao rezultanta tog modifikatora i odgovarajuće labele u okviru opisa pretka. Sama klasa pak ima *tri*, a ne dva nivoa zaštite kako smo smatrali do sada. To su segmenti labelirani redom sa *private:*, *public:* i *protected:*. Na primer,

```
class A {
private:
  T1 privP;
protected:
  T2 protP;
public:
  T3 pubP;
};
```

Segmenti *private* i *public* poznati su odranije: pripadnici prvog zatvoreni su za sve osim za sopstvene metode klase *A*; članovi iz segmenta *public* otvoreni su za sve. Članovi klase iz segmenta *protected* dostupni su metodama iz klase *A* ali i njenih naslednika, dok su zatvoreni za sve ostale klijente.

Prema primeru na slici 7.18, klasa *B* nasleđuje klasu *A*, preuzimajući tako podatke-članove *privP*, *protP* i *pubP*. Rezultantni nivo zaštite preuzetih članova *u klasi B* određen je kako nivoom zaštite u pretku *A* tako i modifikatorom μ. Pravila koja regulišu nivo zaštite su sledeća:

1. Članovi koji u pretku imaju nivo zaštite *private* nedostupni su metodama naslednika nezavisno od oblika modifikatora μ. U našem primeru član klase *A* za nazivom *privP* nedostupan je metodama klase *B*.
2. Za članove pretka koji imaju zaštitu *protected* odnosno *private* efektivni nivo zaštite određen je sledećom šemom:

		zaštita u pretku		
		public	*protected*	*private*
modifik μ	*public*	public	protected	- - -
	protected	protected	protected	- - -
	private	private	private	- - -

Slika 7.19

Prema tome, ako je modifikator μ oblika *public* tada preuzeti članovi zadržavaju i u nasledniku nivo zaštite iz pretka. Ako je μ oblika *protected* zaštita se pomera ka *protected* (u našem primeru članovi *pubP* i *protP* u klasi *B* postaju *protected*). Konačno, ako je μ tipa *private* svi preuzeti članovi nivoa *public* i *protected* (tj. *pubP* i *protP* u klasi *A*) dobijaju u nasledniku efektivni nivo zaštite *private*.

U praksi, modifikator μ je najčešće oblika *public* koji ne menja zaštitu iz pretka (osim, naravno, članova *private* iz pretka koji su nedostupni metodama naslednika). Posebno skrećemo pažnju na to da ako se na mestu μ ne navede ništa (tj. class B: A {...};) podrazumeva se ne najfrekventniji oblik *public* nego *private*(!) čime se programer motiviše da razmišlja o pitanju zaštite.

Ostaje još da se razjasni kada se nasleđivanje izvodi kao *private* odnosno *protected*. Modifikator *private* koristi se u situacijama kada se od pretka preuzima samo implementacija, uz nov interfejs, tj. kada se radi o nasleđivanju implementacije. Naime, ako je μ oblika *private* kompletan interfejs pretka postaje dostupan samo metodama naslednika, što zahteva izradu potpuno novog interfejsa naslednika. Ako je nasleđivanje izvedeno kao *protected* tada naslednik zadržava interfejs pretka ali samo za sopstvene potomke, dok se za klijente mora sačiniti nov.

Redefinisanje metoda iz pretka vrši se jednostavno: metoda se ponovo realizuje u nasledniku, pod istim imenom. Pri tom, tip metode i parametri ne moraju se poklapati[65]. Treba zapaziti da se redefinisana metoda koja nema iste parametre u

[65] Napomena: videćemo kasnije da za tzv. "virtuelne metode" to ne važi. Kod njih se ime *i parametri* moraju poklapati (tip ne).

nasledniku ne smatra preklopljenom: metoda pretka sa istim imenom "prekriva" se metodom naslednika bez obzira na to da li imaju jednake parametre ili ne.

Ako to zaštita dozvoljava, metode date klase mogu aktivirati metode tačno određenog pretka, neposrednog ili daljeg. To se postiže *kvalifikovanjem poziva* imenom pretka praćenim dvoznakom ::. Neka klasa A iz našeg primera sadrži metodu *metM()* koja je redefinisana u klasi *B*. To znači da se izrazom *metM()* u klasi *B* aktivira redefinisana verzija metode *metM()*. Želimo li da iz metode klase *B* ipak aktiviramo originalnu verziju *metM()* iz klase *A*, neophodno je kvalifikovati poziv tako da dobije oblik

A::metM().

Mogućnost kvalifikovanja verzije pri pozivu proteže se na ma koji predak, a ne samo na roditeljsku klasu.

U svojstvu prvog primera realizovaćemo i diskutovati hijerarhiju klasa trouglova iz primera 7.1. Osnovna klasa *Trougao* ima sledeću realizaciju:

```
class Trougao {
protected:
  double s1, s2, s3;
public:
  Trougao(double x, double y, double z): s1(x),s2(y),s3(z) {}
  ~Trougao() {}
  double a() const {return s1;}
  double b() const {return s2;}
  double c() const {return s3;}
  double obim() const {return s1+s2+s3;}
  double povrsina() const;
};

double Trougao::povrsina() {
  double p = 0.5*(s1+s2+s3);
  return sqrt(p*(p-s1)*(p-s2)*(p-s3));
}
```

U klasi nema ničeg posebno novog izuzev segmenta koji je labeliran sa *protected:* što znači da su podaci-članovi s1, s2 i s3 dostupni metodama (ne i klijentima!) klase *Trougao* i njenih potomaka, naravno u zavisnosti of modifikatora zaštite μ. Klasa *PravougliTrougao* ima sledeći izgled:

```
class PravougliTrougao {
public:
  PravougliTrougao(double, double);
  ~PravougliTrougao() {}
  double kateta1() const {return s1;}
  double kateta2() const {return s2;}
  double hipotenuza() const {return s3;}
  double povrsina() const {return 0.5*s1*s2;}
};

PravougliTrougao::PravougliTrougao (double kat1, double kat2):
                    Trougao(kat1,kat2,sqrt(kat1*kat1+kat2*kat2)) {}
```

Kako vidimo, metode *kateta1()*, *kateta2()*, *hipotenuza()* i povrsina() imaju slobodan pristup podacima-članovima *s1*, *s2* i *s3*, koji se preuzimaju iz klase *Trougao* jer su u toj klasi zaštićeni nivoom *protected*. Metode *a()*, *b()*, *c()* i *obim()* prenose se u klasu *PravougliTrougao* i otvorene su za upotrebu kao da su definisane u njoj. Postavlja se pitanje da li je metode *a()*, *b()* i *c()* moguće na neki način "sakriti" jer se u klasi *PravougliTrougao* pojavljuju pod drugim, uobičajenijim imenima? Odgovor glasi: moguće je i to redefinisanjem u segmentu *private* klase *PravougliTrougao*. Odmah, međutim, dodajemo da tako nešto može da prouzrokuje probleme sa inkluzionim polimorfizmom, pa je bolje ostaviti ih otvorenim jer ne smetaju.

Metoda *povrsina()* klase *PravougliTrougao* redefinisana je tako da, u skladu sa pravilom očuvanja semantike, obavlja isti posao (računa površinu), ali na drukčiji način, primeren pravouglom trouglu.

Klasa *RavnostraniTrougao* ima sledeću realizaciju:

```
class RavnostraniTrougao {
public:
  RavnostraniTrougao(double s): Trougao(s,s,s) {}
  ~RavnostraniTrougao() {}
  double stranica() const {return s1;}
  double visina() const {return 0.866025403*s1;}
  double obim() const {return 3*s1;}
  double povrsina() const {return 0.433012701*s1;}
};
```

Ovde su redefinisane i metoda *obim()* i metoda *povrsina()*, jer se računaju na drugi način. Dodate su nove metode *stranica()* i *visina()* kojih u pretku nema (u opštem

slučaju trougao ima tri različite stranice i tri visine).

Konstruisanje objekta u uslovima kada ima nasleđivanja vrši se u etapama, po principu "svaki objekat vodi brigu sam o sebi". Redosled poziva konstruktora objekta *b* klase *B* čiji je neposredni predak klasa *A* je sledeći:

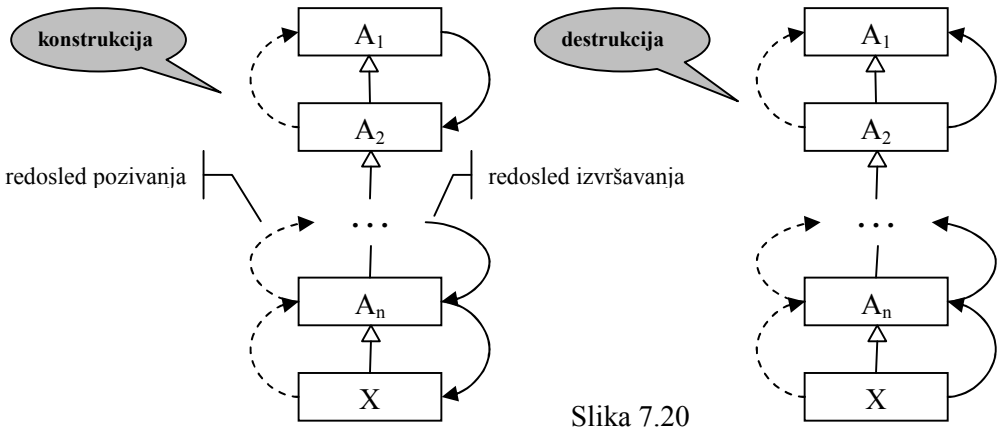
1. Poziv konstruktora klase A (pretka) čiji je zadatak da konstruiše deo objekta b koji je preuzet od klase A. U klasama *PravougliTrougao* i *RavnostraniTrougao* prvo se poziva konstruktor pretka *Trougao*. Ako u nekom konstruktoru naslednika nema eksplicitnog poziva konstruktora pretka tada se poziva podrazumevani konstruktor, pri čemu se mora voditi računa da on postoji(!). Inače, podaci-članovi preuzeti od pretka po pravilu se inicijalizuju inicijalizatorom pretka da bi se poštovalo pravilo inkapsulacije. Na primer, alternativna realizacija konstruktora klase *RavnostraniTrougao* oblika

   RavnostraniTrougao(double s): s1(s), s2(s), s3(s) {}

   ili

   RavnostraniTrougao(double s) {s1= s2= s3= s;}

   koja je moguća, ne smatra se dobrim rešenjem jer nepotrebno zadire u strukturu klase *Trougao*. Naročito se izbegava inicijalizacija članova koji pripadaju udaljenim precima.
2. U drugom koraku konstruišu se objekti-članovi, eksplicitno ili korišćenjem njihovih podrazumevanih konstruktora.
3. Konstrukcija se završava delom koji pripada matičnoj klasi.



Slika 7.20

Postupak konstruisanja je rekurzivan, tj. primenjuje se po istoj šemi i na pretke i na objekte-članove. Posmatrajmo klasu *X* čiji su preci klase $A_1, \ldots, A_n$ gde je A_n roditeljska klasa klase *X*, a klasa A_{i+1} nasleđuje klasu A_i za i=1, ..., n-1, slika 7.20. Rekurzivnost postupka znači sledeće: kada se pozove konstruktor klase *X* on poziva

konstruktor klase A_n i to pre nego što otpočne sa svojim zadatkom. Isto važi i za sve ostale konstruktore[66], tako da je redosled *pozivanja* konstruktora X, A_n,...,A_1, a redosled *izvršavanja* obrnut, kao što je prikazano na slici 7.20 levo. Na taj način obezbeđeno je nešto što važi u svim objektnim jezicima: deo objekta iz klase X koji je nasleđen od klase A_i konstruiše se konstruktorom iz klase A_i, za i=1,...,n, a sopstveni konstruktor X konstruiše samo onaj deo objekta koji je formiran u klasi X.

Destrukcija objekta vrši se u suprotnom redosledu od njegove konstrukcije, u skladu sa istim principom "objekat vodi brigu o sebi". Do razlike u odnosu na redosled konstruisanja dolazi zbog toga što se prilikom rekurzivne destrukcije destruktor roditeljske klase poziva posle destruktora naslednika. Dakle, prvo se pozove i izvrši destruktor ~X, zatim destruktor ~A_n, pa destruktor ~A_{n-1} itd. sve do destruktora ~A_1, slika 7.20 desno. Vidimo da i za destruktore važi isti princip kao i za konstruktore: deo objekta iz klase X koji je nasleđen od klase A_i destruiše se destruktorom iz klase A_i, za i=1,...,n, a sopstveni destruktor X destruiše samo onaj deo objekta koji je konstruisan u klasi X.

Za nasleđivanje uopšte, važe još neka pravila koja su vrlo logična. Prvo, operator dodele (koji je uvek definisan) ne prenosi se u potklasu. Ovo je razumljivo s obzirom na to da operator može da bude preklopljen u natklasi na način koji ne odgovara potklasi. Kooperativnost se takođe ne nasleđuje, saobrazno pravilu da svaka klasa sama reguliše sopstvenu zaštitu. Ako je neka slobodna funkcija ili klasa prijateljska za natklasu to ne sme imati uticaja na potklasu koja, po potrebi, može "obnoviti prijateljstvo", ali samo eksplicitno, upotrebom modifikatora *friend* u sopstvenom sadržaju.

Primer 7.3.

Da se ne bismo zadržali samo na matematičkom primeru realizovaćemo i klase *Clock* i *AlarmClock* čiji je dijagram dat u primeru 7.3. Zapazimo da izvedena klasa *AlarmClock* ima znatno više dodatih članova nego u primeru sa trouglom. Takođe, dodati članovi u mnogo većoj meri obogaćuju semantiku pretka, jer modeluju osobinu budilnika da se može podesiti da u određeno vreme generiše alarm. Konačno, uspostavljanje veze nasleđivanja između dve klase potpuno je u skladu sa mogućom (aristotelovskom) definicijom budilnika kao "sata koji može da generiše alarm". Sledi izvedba odgovarajućih modula.

```
/************************************************************

                    KLASA CLOCK (SAT)
```

[66] jer je to bilo obezbeđeno prilikom realizacije klasa A_n, ..., A_1.

```
Naziv datoteke: CLOCK.HPP
*****************************************************************/
#ifndef CLOCK_HPP
  #define CLOCK_HPP

class Clock {
protected:
  int h,m;
public:
  Clock(): h(0),m(0) {}
  ~Clock() {}
  void SetTime(int hour,int min) {h= hour; m= min;}
  void GetTime(int &hour,int &min) const {hour= h; min= m;}
  void NextMin();
};

void Clock::NextMin()
{
  if((m=(m+1)%60)==0) h= (h+1)%24;
}

#endif
```

```
/*****************************************************************

                    KLASA ALARMCLOCK (BUDILNIK)

Naziv datoteke: ALCLOCK.HPP
*****************************************************************/

#ifndef ALCLOCK_HPP
  #define ALCLOCK_HPP
#include "clock.hpp"

class AlarmClock: public Clock {
protected:
  int alarm_h,alarm_m;                  // Dodati podaci-clanovi
  int alarm_on,ringing;                 // Dodati podaci-clanovi
  void startRinging() {ringing= 1;}             // Dodata metoda
public:
```

```
  AlarmClock(): alarm_h(0),alarm_m(0),alarm_on(0),ringing(0) {}
  ~AlarmClock() {}
  void setAlarmTime(int ahour,int amin)
    {alarm_h= ahour; alarm_m= amin;}            // Dodata metoda
  void getAlarmTime(int &ahour,int &amin) const   // Dodata metoda
    {ahour= alarm_h; amin= alarm_m;}            // Dodata metoda
  void disableAlarm() {alarm_on= 0;}             // Dodata metoda
  void enableAlarm() {alarm_on= 1;}              // Dodata metoda
  int alarmIsEnabled() const {return alarm_on;}   // Dodata metoda
  void stopRinging() {ringing= 0; alarm_on= 0;}   // Dodata metoda
  int isRinging() const {return ringing;}        // Dodata metoda
  void nextMin();                        // Redefinisana metoda
};

void AlarmClock::nextMin()
{
  if((m=(m+1)%60)==0) h= (h+1)%24;
  if(alarm_on&&(h==alarm_h)&&(m==alarm_m)) StartRinging();
}

#endif
```

Kratak primer primene:

```
AlarmClock alClock;
. . . . . .
alClock.setTime(10,59);     // SetTime je nasledjena metoda
alClock.setAlarmTime(11,0)     // Postavljanje vremena alarma
alClock.enableAlarm();     // Ukljucivanje alarma
alClock.nextMin();     // Pomeranje minut napred. NextMin je redefinisana
```

Metoda *nextMin()* je redefinisana, tako da se pri pomeranju napred proverava da li se novo vreme poklapa sa vremenom alarma. Ako se poklapa i ako je alarm uključen (omogućen metodom *enableAlarm()*) aktivira se signal alarma. Gornji primer je podešen upravo na takvo stanje, pa će na kraju segmenta važiti alClock.isRinging() = 1. Na kraju, jedna napomena: iz razloga koji će biti objašnjeni u poglavlju o virtuelnim funkcijama, objekti klase *AlarmClock* ne smeju se koristiti posredstvom pokazivača.

7.4. DEMETRIN ZAKON

Demetrin zakon (engl. Law of Demeter, skraćeno LoD) bavi se pitanjima inkapsulacije i kontrole pristupa, posebno pri *razmeni poruka* u sistemu objekata povezanih klijentskim vezama i vezama nasleđivanja. Formulisan je u okviru projekta "Demeter" na Northeastern University u Bostonu, SAD [84, 85]. Naziv je sasvim sigurno dobio po starogrčkoj boginji plodnosti i zemljoradnje, Demetri čije se ime na engleskom piše "Demeter"[67].

Zakon ima snagu preporuke, tj. softver se može realizovati i bez njegove primene što, uostalom, neki autori (npr. Mejer) i zagovaraju. Demetrin zakon neposredno propisuje *kojim objektima dati objekat sme da šalje poruke*, a da se pri tom poštuje tzv. "princip minimalnog znanja" (engl. "principle of least knowledge"). Sa svoje strane, ovaj princip zahteva da se objektom manipuliše uz minimum poznavanja njegove interne strukture i internih osobina i nije ništa drugo do jedna od formi principa skrivanja informacija.

U originalnom obliku pojavljuje se u dva vida: kao tzv. Slabi Demetrin zakon ("Weak Law of Demeter") i kao Strogi Demetrin zakon ("Strong Law of Demeter"), u zavisnosti od tretmana nasleđenih veza sa drugim objektima. U prvom slučaju smatra se da su nasleđene veze deo klase, a u drugom da nisu. ***Strogi Demetrin zakon*** glasi:

- *Metoda M* objekta O iz klase C može aktivirati metode sledećih klasa: (1) sopstvene (tj. karakterisane sa *this*) (2) svojih parametara (3) svojih lokalnih objekata i (4) objekata sa kojima je u neposrednoj klijentskoj vezi.

Uočimo, dakle, da Strogi Demetrin zakon ne dozvoljava aktiviranje metoda objekata koje je data klasa preuzela nasleđivanjem. ***Slabi Demetrin zakon*** pored navedenog dozvoljava i pristup (tačnije, slanje poruka) nasleđenim objektima.

Od datuma formulisanja Demetrinog zakona prošlo je petnaestak godina. U toku tog vremena količina znanja o objektnom programiranju enormno se povećala (između ostalog, izgrađena je objektna *metodologija*), što nije moglo proći bez reperkusija na oblik samog zakona. Buč [3] - doduše neformalno - uvodi i treći vid Zakona koji bismo mogli okarakterisati kao "srednji nivo":

- *Metoda M* objekta O iz klase C može aktivirati metode sledećih klasa: (1) sopstvene (2) svojih parametara (3) svojih lokalnih objekata (4) objekata sa kojima je u neposrednoj klijentskoj vezi i (4) svojih neposrednih natklasa (roditeljskih klasa)[68].

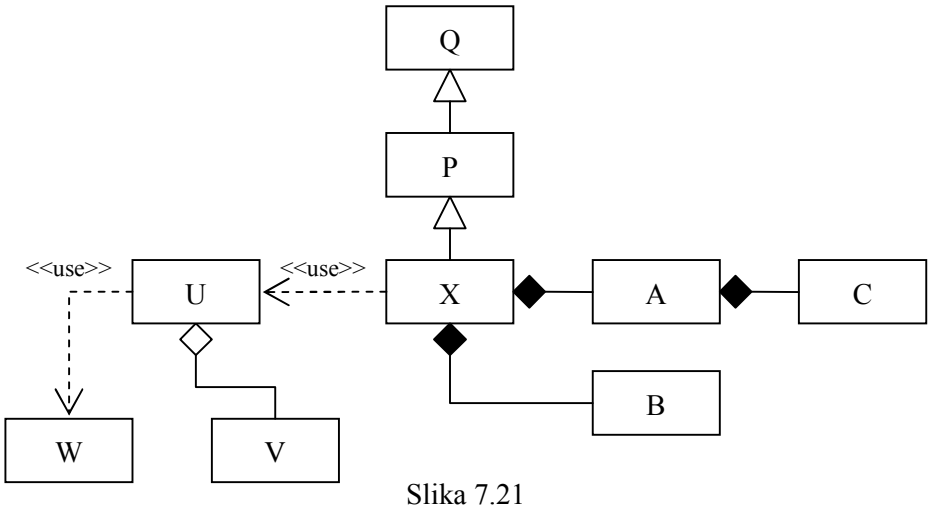
Skup objekata kojima se može poslati poruka a da se ne naruši Demetrin zakon srednjeg nivoa lako se prepoznaje na UML dijagramu klasa. To su objekti čije su klase direktno povezane sa matičnom klasom datog objekta klijentskim vezama i

[67] Dakle, teško da se radi o "Demeterovom zakonu" kako se često prevodi.

[68] Napomena: pod "slanjem poruke neposrednom pretku" podrazumeva se, u stvari, korišćenje originalne verzije redefinisane metode pretka.

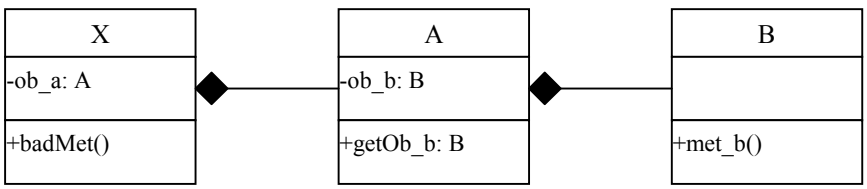
nasleđivanjem, s tim da se kod veza korišćenja i nasleđivanja poštuje smer (*od* datog objekta), a kod agregacije - kompozicije *ka* klasi sa ulogom dela (dati objekat je celina).

Prema Demetrinom zakonu srednjeg nivoa (njim ćemo se baviti u nastavku), objekat *x* klase *X* može slati poruke objektima klasa *A*, *B*, *P* i *U*, a ne može slati poruke objektima iz klasa *C*, *Q*, *V* i *W*.



Slika 7.21

Geneza Demetrinog zakona zasnovana je na zabrani korišćenja konstrukata oblika *y.m1().m2()* gde je *y* objekat, *m1* metoda kojom se pristupa njegovom objektu-članu, a *m2* metoda klase objekta-člana. Naime, da je na mestu *m1* identifikator objekta-člana z, tj. kad bi konstrukt imao oblik *y.z.m2()* probijanje zaštite bilo bi očigledno. Međutim, ako je *m1* metoda kojom se samo pristupa objektu-članu *z*, narušavanje inkapsulacije više nije upadljivo, ali postoji. Posmatrajmo primer dat na slici 7.22.

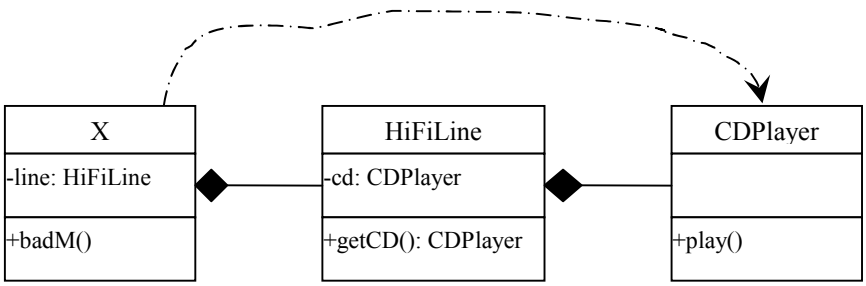


Slika 7.22

Pretpostavićemo da metoda *getOb_b* kao vrednost vraća objekat *ob_b* klase *B*, koji je član klase *A*. Neka, dalje, klasa *X* sadrži metodu *badMet* u kojoj se nalazi poziv

ob_a.getOb_b().met_b().

Naizgled, sa inkapsulacijom i zaštitom sve je u redu: objekti-članovi *ob_a* i *ob_b* su zatvoreni i klasama se pristupa samo putem interfejsa, dakle bez ulaženja u njihovu internu strukturu. A da li je zaista tako? Odgovor je *ne* jer metoda *getOb_b* vraća kao vrednost zatvoreni objekat član *ob_b* da bi se zatim na *zatvoreni objekat-član* primenila metoda *met_b*. Iako prikriven, proboj inkapsulacije ipak postoji! Stvarni uzrok nije zaobilaženje interfejsa (toga nema) nego nemoć zaštite da spreči zaobilazni ulazak u strukturu putem metode *getOb_b*. U svrhu potpune zaštite neophodni su dodatni mehanizmi, a oni su obuhvaćeni upravo Demetrinim zakonom. Posmatrajmo jedan primer. Na slici 7.23 prikazane su klase *HiFiLine* i *CDPlayer* koje predstavljaju sasvim redukovane modele stereo linije i ugrađenog CD plejera. Pretpostavimo da se model proširuje novom klasom *X* (nije važno šta je ona konkretno) koja za komponentu ima primerak klase *HiFiLine*.



Slika 7.23

Neka su metode *badM* i *getCD* realizovane na sledeći način:

```
void X::badM() {line.getCD().play();}
CDPlayer& HiFiLine::getCD() {return cd;}
```

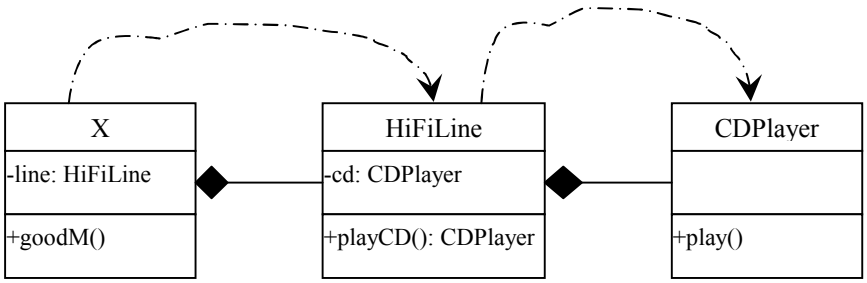
Do narušavanja inkapsulacije dolazi u metodi *badM* klase *X* što se jasno vidi i na crtežu gde je nepravilnom linijom prikazano obraćanje klase *X* klasi *CDPlayer* koja sa njom nije u neposrednoj vezi. Međutim, stvarni problem nije ni u klasi *X* niti u njenoj metodi *badM*. Nalazi se u klasi *HiFiLine* koja je formalno jako inkapsulirana (objekat-član *cd* je zatvoren), ali suštinski nije jer se objekat-član *cd* u celosti dohvata metodom *getCD* i nad njim se mogu vršiti sve operacije predviđene klasom *CDPlayer*. Na taj način, klasa *HiFiLine* efektivno se ponaša kao da je *cd* otvoren za sve klijente (public). Navešćemo četiri načina za realizaciju ovog i sličnih objektnih sistema.

Rešenje 1. Bazirano je na opštem, originalnom rešenju iz [85]. Osnovna ideja je "ispraviti grešku tamo gde je nastala", tj. modifikovati ne klasu *X* gde se problem

ispoljava nego klasu *HiFiLine* gde je problem nastao efektivnim otvaranjem objekta-člana *cd*. Rešenje funkcioniše za proizvoljnu dubinu poziva tipa *y.m1().m2(). ... .mk()*, a mehanizam se zasniva na takvoj izvedbi metoda da poruke propagiraju od polaznog objekta (u našem primeru objekat klase *X*) do odredišnog (klasa *CDPlayer*) ne preskačući intermedijarne objekte (klasa *HiFiLine*). Pri tom, svaki čvor na putu poruke sam odlučuje kojem neposrednom susedu će poruka biti prosleđena.

Da bi se ovaj postupak implementirao, u našem primeru neophodno je ukloniti spornu metodu *getCD* iz klase *HiFiLine* i zameniti je metodom koja se obraća klasi *CDPlayer* pobuđujući njenu metodu *play*, slika 7.24. Sada metode imaju sledeći izgled:

```
void  X::goodM() {line.playCD();}
void HiFiLine::playCD() {cd.play();}
```

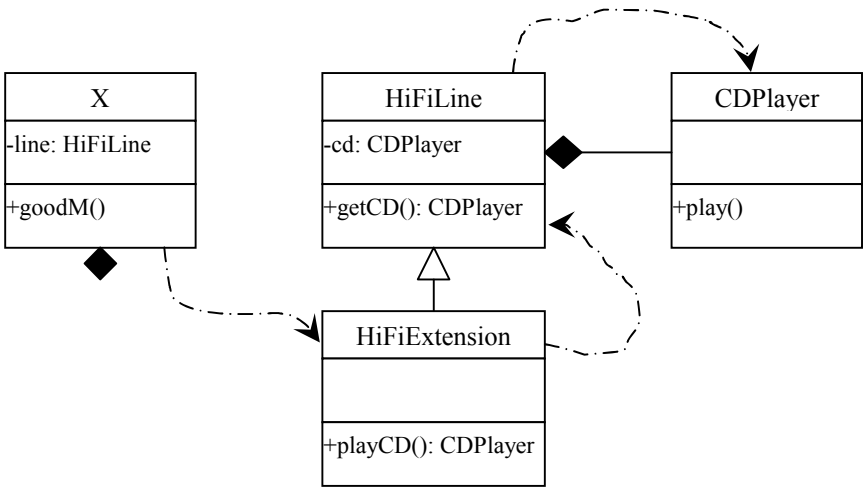


Slika 7.24

Kako vidimo, klasa *X* obraća se isključivo susedu, klasi *HiFiLine*, porukom *playCD* koju ova prosleđuje klasi *CDPlayer* porukom *play*. Prednost ovog rešenja leži u činjenici da je temeljni princip skrivanja informacija proširen i na put kojim propagira poruka, a koji postaje nepoznat pošiljaocu i može se menjati na nivou svakog čvora bez učešća ostalih čvorova. Ovakav način programiranja dobio je čak i naziv: *adaptivno programiranje*. Broj veza se ne menja, pa time ni složenost sistema. Izmene u nekoj klasi dotiču samo njene susede. Nedostatak je povećavanje broja metoda što u datom primeru nije očigledno. Međutim, kada bi u klasi *CDPlayer* postojala i metoda *stop* (što je za očekivanje) tada bi se klasa *HiFiLine* morala proširiti i metodom npr. *stopCD* za pobuđivanje metode *stop*. Još jedan nedostatak je to što grešku konstatovanu u jednoj klasi (*X*) ispravljamo menjanjem druge klase (*HiFiLine*).

Rešenje 2. U slučaju da se klasa *HiFiLine* ne želi ili ne može modifikovati, pozivamo u pomoć nasleđivanje. Formiraćemo klasu *HiFiExtension* koja nasleđuje

klasu *HiFiLine* i u njen interfejs uključiti sve one metode kojim bismo proširivali klasu *HiFiLine* u rešenju 1. Izmenjeni dijagram klasa prikazan je na slici 7.25.



Slika 7.25

Ostaje da se vidi šta raditi sa metodom *getCD* koju takođe nasleđuje klasa *HiFiExtension*. Za ovo ima više rešenja:

1. Preuzeti metodu u interfejs, ali je ne koristiti
2. Poslati metodu "u karantin", tj. redefinisati je tako da ne radi ništa i nalazi se u delu *protected* ili *private* klase *HiFiExtension*.
3. Nasleđivanje izvesti tako da bude tipa *protected* ili *private* i ponovo napisati interfejs klase *HiFiExtension*.

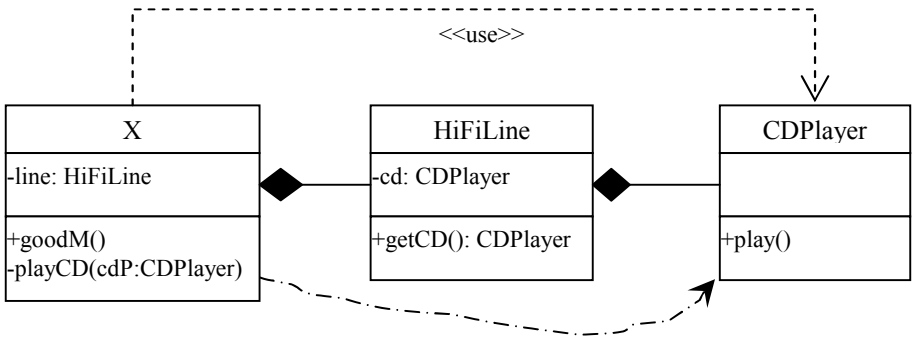
Ilustracije radi, prikazaćemo varijantu 2. Realizacija ima sledeći oblik:

```
class HiFiExtension: public HiFiLine {
protected:
  CDPlayer& getCD() {}
public:
  void playCD() {HiFiLine::getCD().play();}
};
class X {
private:
  HiFiExtension line;
public:
  void goodM() {line.playCD();}
```

Prednost ovog rešenja u odnosu na prethodno je to da se (gotova) klasa *HiFiLine* ne

menja. Međutim, ovo rešenje ima i potencijalno ozbiljan nedostatak koji se sastoji u tom što metoda *getCD* faktički nije preuzeta, osim kod varijante 1. Ovo je u koliziji sa prirodom nasleđivanja kao modela odnosa subordinacije koji ne predviđa odbacivanje dela sadržaja natklase. Tehnički, ovaj nedostatak pokazao bi se pri primeni inkluzionog polimorfizma (videti dalje). S druge strane, varijanta 1 ostavlja i metodu *getCD* na raspolaganju klijentu što smo, na neki način, želeli upravo da izbegnemo.

Rešenje 3. Eliminisati obraćanje klasi koja nije neposredni sused tako što je učinimo susedom primenom veze korišćenja. Rešenje se razlikuje od prethodnih po tome što se podešava *samo nova klasa* (našem slučaju klasa *X*), dok ostale ostaju nepromenjene. U klasu *X*, umesto jedne, uključuju se dve metode: modifikovana sporna metoda *goodM* i metoda *playCD sa parametrom klase CDPlayer*, gde *playCD* može po potrebi biti i zatvorena. Na taj način uspostavlja se veza korišćenja između klasa *X* i *CDPlayer*, tako da one postaju susedi u dijagramu klasa, slika 7.26.



Slika 7.26

Sada klasa *X* ima ovakav izgled:

```
class X {
private:
  HiFiLine line;
  void playcd(CDPlayer &cdP) {cdP.Play();}
public:
  void GoodM() {playcd(line.GetCD());}
};
```

Prednost ovog rešenja očigledno je u tome što se klase *HiFiLine* i *CDPlayer* ne modifikuju. S druge strane, uvođenjem veze korišćenja (naročito ako ih ima više sličnih ovoj) povećava se kompleksnost objektnog sistema.

Rešenje 4. Ovo rešenje zagovara B. Mejer koji na primedbu o obraćanju nesusednoj klasi porukom tipa *y.m1().m2(). ... .mk()* odgovara "pa šta onda?", ne nalazeći da takva solucija unosi neke posebne probleme. Drugim rečima, a nasuprot prethodnim postupcima, ovo rešenje jednostavno prelazi preko činjenice da je Demetrin zakon narušen. U svetlu Mejerovih ideja, naš primer bismo "modifikovali" tako što bismo promenili naziv metode *badM* u *goodM* i to je sve. Prednost je besumnje u tome što nije potrebno dopisivati nikakav softver samo da bismo poštovali Demetrin zakon. Nedostatak je postojanje "nevidljivih" veza koje formalno nisu ni klijentske niti spadaju u nasleđivanje i kojih nema na dijagramu klasa. Nepoželjna logička kohezija između separatnih klasa povećava se. Ako u našem primeru klasa *X* poseduje konstrukt *getCD().Play()* tada ona *de facto* postaje neka vrsta klijenta klase *CDPlayer*, samo što se to na dijagramu ne vidi. Ako u klasi-receptoru ovakve poruke (klasa *CDPlayer*) dođe do izmene interfejsa (sa čim se uvek mora računati) znatno je teže ustanoviti u kojim još klasama treba da bude izvršena modifikacija.

Sve u svemu, smatramo da poštovanje ili nepoštovanje Demetrinog zakona ne treba da bude generalno opredeljenje "po svaku cenu". Odluka o tome zavisi od situacije, a uslovi za njeno donošenje determinisani su razmatranjem navedenih prednosti i nedostataka u svakom konkretnom slučaju.

8. SLOŽENIJI ASPEKTI NASLEĐIVANJA. VIŠESTRUKO NASLEĐIVANJE

U ovom poglavlju pozabavićemo se složenijim - u stvari najbitnijim - aspektima nasleđivanja: pre svega vezom nasleđivanja sa inkluzionim polimorfizmom koja je toliko čvrsta da se ova dva pojma u stvari prožimaju. Potom ćemo razmotriti opštiji, ali ne toliko važan, pojam višestrukog nasleđivanja.

8.1. NASLEĐIVANJE I POLIMORFIZAM

Inkluzioni polimorfizam je karakteristika koja se vezuje za promenljive odnosno objekte. U procedurnim jezicima poput C ispoljava se tako što se promenljiva ponaša kao da pripada većem broju tipova. Na primer, znakovna promenljiva ili promenljiva tipa enumeracije uključuju se po potrebi u celobrojne izraze u kojima se ponašaju kao da su celobrojne. Takođe, može im se dodeliti celobrojna vrednost i mogu se prikazati u celobrojnom formatu. Uočimo da inkluzioni polimorfizam ovakve vrste postoji samo zahvaljujući tome što su i znakovni tip i enumeracija u C-u članovi familije celobrojnih tipova. Paskal na primer, kao strogo tipiziran jezik, sličnu mogućnost nema zato što su znakovni tip odnosno enumeracija tipovi za sebe i nemaju nikakve veze sa celobrojnim tipom.

U objektnom ambijentu, na inkluzioni polimorfizam nailazimo u kontekstu operacije koju ćemo nazvati ***pridruživanje***. Pod pridruživanjem (u objektnom ambijentu) podrazumevaćemo

1. operaciju dodele, gde se objekat koji je desni operand dodeljuje objektu sa leve strane operatora dodele;
2. operaciju prenosa argumenata u slobodni potprogram ili metodu kod koje objekat-argument zamenjuje objekat-parametar;
3. prosleđivanje rezultata funkcije (slobodne ili članice) kada je rezultat objekat.

Pošto nema nikakve razlike u pogledu polimorfnog ponašanja, sve tri operacije označićemo simbolom "$\leftarrow$". Dakle, kada napišemo

$$a \leftarrow b$$

to će formalno značiti

- izraz a= b
- zamenu a sa b prilikom poziva nekog potprograma-metode u kojoj je *a* ime parametar u definiciji funkcije, a *b* argument pri pozivu
- objekat *b* formiran u funkciji vraća se kao rezultat (gde *a* formalno označava rezultat)

Za sve vrste pridruživanja, u objektnom programiranju važi isto ***pravilo pridruživanja*** koje glasi:

- Objekat izvedene klase može se pridružiti objektu bazne klase, dok obrnuto ne važi[69].

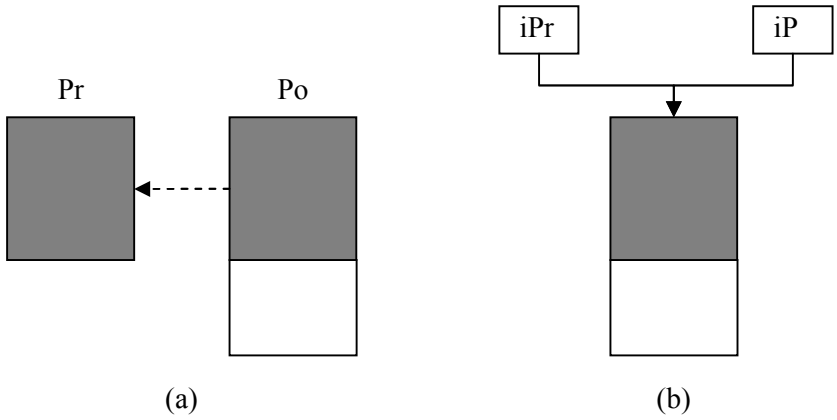
Pre svega, tehnički razlog za zabranu pridruživanja potomka pretku vrlo je jasan: potomak ima sve podatke-članove i objekte-članove koje ima i predak, ali i neke sopstvene. Ako bi se dozvolilo pridruživanje pretka potomku članovi potomka kojih nema u pretku ostali bi nedefinisani. Mnogo je, međutim, važniji suštinski razlog zabrane do kojeg dolazimo analizom operacije pridruživanja. Naime, pridruživanje oblika $a \leftarrow b$ odgovara situaciji u kojoj se sadržaj pojma P_b kojem odgovara objekat *b* prenosi na pojam P_a kojem odgovara objekat *a*. Kako je pojam P_b subordiniran pojmu P_a, on obuhvata kompletan sadržaj pojma P_a (i još ponešto), tako da se taj zajednički deo sadržaja prenosi na P_a. U obrnutoj situaciji, $b \leftarrow a$, pošto je sadržaj pojma P_a uži od sadržaja pojma P_b, po izvršenom pridruživanjau P_b nije definisan do kraja, jer deo sadržaja kojeg nema u P_a ostaje bez vrednosti. Kada se izvodi pridruživanje objekata oblika *sat*←*budilnik* objekat *budilnik* prenosi na objekat *sat* osobine koje su zajedničke. Pridruživanje oblika *budilnik*←*sat* nije dozvoljeno jer se ovom operacijom ne mogu preneti osobine vezane za alarmni mehanizam koje objekat *sat* jednostavno ne poseduje.

Prilikom izvršavanja operacije pridruživanja pojavljuju se dve vrlo različite situacije. Neka su *Pr* i *Po* redom (obične) instance klasa *Predak* i *Potomak*. Neka su *iPr* i *iPo* indirekcije na instance redom *Predak* i *Potomak*, pri čemu se pod ***indirekcijom*** podrazumevaju pokazivač ili referenca. Operacija pridruživanja ponaša se različito za instance *Pr* i *Po* odnosno indirekcije *iPr* i *iPo*. Naime, operacijom nad instancama

$$Pr \leftarrow Po$$

[69] Naravno, klasa može biti neposredni potomak, ali i potomak na ma kojem nivou, sve zbog tranzitivnosti nasleđivanja!

zajednički podaci-članovi kopiraju se iz memorijskog prostora potomka *Po* na odgovarajuće lokacije u memorijskom prostoru pretka *Pr*, (varijanta *(a)* na slici 8.1).



Slika 8.1

Sasvim je drukčiji slučaj pridruživanja nad indirekcijama. Pre svega, indirekcija uvek podrazumeva kao vrednost ne instancu klase nego *adresu* instance klase, što je naročito uočljivo kod pokazivača. Pridruživanjem indirekcije oblika

$$iPr \leftarrow iPo$$

u *iPr* se prenosi adresa sadržana u *iPo*, tako da po izvršenom pridruživanju obe indirekcije, *iPr* i *iPo*, sadrže *adresu istog objekta*, slika 8.1, varijanta *(b)*.

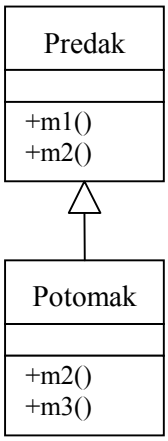
Pridruživanje instanci $Pr \leftarrow Po$ ne donosi ništa novo: kopiraju se zajednički delovi objekata, pri čemu obe instance zadržavaju originalne karakteristike, što je naročito važno za objekat *Pr* koji, dakle, zadržava sve osobine instance klase *Predak*. Donekle sličan način pridruživanja sreće se i u procedurnim programskim jezicima: ako je *r* promenljiva realnog tipa, a *j* celobrojna promenljiva tada se *j* može pridružiti promenljivoj *r*, ali tako da se vrednost *j* prilikom prenosa konvertuje u realni tip. Obe promenljive u potpunosti zadržavaju ponašanje svojstveno svom originalnom tipu. Kao slabo tipiziran jezik, C/C^{++} ima daleko širi repertoar sličnih pridruživanja, ali se uvek dešava ista stvar: interna konverzija vrednosti uz zadržavanje ranijih osobina. Razlika između objektne i procedurne dodele ipak postoji: kada se instanca potomka dodeljuje instanci pretka smatra se da nema konverzije, nego da deluje tzv. ***zakon supstitucije*** (formulisala ga je Barbara Liskov) koji, u objektnoj interpretaciji, glasi:

- Ako za svaki objekat *n* klase *N* postoji objekat *p* klase *P* takav da proizvol-

jni program definisan nad *P* ne menja ponašanje kada se *p* zameni sa *n*, tada je *N* potklasa (potomak) klase *P*.

Ovde, naravno, "zameniti *p* sa *n*" znači izvršiti pridruživanje *p*←*n*. Očigledno, zakon supstitucije veoma potencira značaj (polimorfnog) pridruživanja, jer se može shvatiti čak i kao definicija nasleđivanja.

Šta se, međutim, dešava sa indirekcijom? Za razliku od prethodnog slučaja, ovde postoji *samo jedan objekat* (i to potomak) čiju adresu posle pridruživanja sadrže obe indirekcije, *iPr* i *iPo*. Neka su klase *Predak* i *Potomak* povezane nasleđivanjem kao na slici 8.2 (činjenica da je nasleđivanje neposredno nema uticaja).



Slika 8.2

Uočimo da je klasa *Potomak* preuzela metodu *m1*, redefinisala metodu *m2* i ima sopstvenu metodu *m3*. Postavlja se pitanje šta se dešava sa ovim metodama posle pridruživanja *iPr*←*iPo*? Na prvi pogled čini se logičnim da, posle pridruživanja, indirekcija na predak menja ponašanje tako da se *iPr* počinje ponašati kao indirekcija na potomak, što ona tehnički i jeste. Ma kako privlačno delovao, zaključak je *netačan*! U svrhu potvrde ovog stava analizovaćemo šta se u objektnim programskim jezicima uopšte dešava sa gornjim metodama, a posebno posle pridruživanja indirekcije *iPr*←*iPo*.

Neka je data klasa *SingleInt* koja sadrži jedan jedini podatak-član *content* tipa *int*. Ona raspolaže konstruktorom koji postavlja vrednost *content* na 0, metodom *setContent(int i)* za postavljanje vrednosti *content* i metodom *getContent()* za očitavanje te vrednosti:

```
class SingleInt {
protected:
  int content;
```

```
public:
  SingleInt() {content= 0;}
  void setContent(int i) {content= i;}
  int getContent() {return content;}
};
```

Iz ove klase izvešćemo klasu *SingleIntWithBackup* koja ima još jedno polje *backup* koje u početnom stanju ima vrednost 0, a u svim ostalim bivšu vrednost polja *content*. Dakle, pre svake promene *content* metodom *setContent* mora se obezbediti da se zatečena vrednost prethodno prepiše u polje *backup*, što znači da se metoda *setContent* mora redefinisati. Klasa ima još jednu metodu *getBackup()* za očitavanje polja *backup*, tj. za očitavanje pretposlednje vrednosti polja *content*:

```
class SingleIntWithBackup: public SingleInt {
public:
  SingleIntWith Backup() {backup= 0;}
  void setContent(int i) {
    backup= content;
    content= i;
  }
  int getBackup() {return backup;}
};
```

Dakle, u ulozi metoda *m1*, *m2* i *m3* sa slike 8.1 pojavljuju se redom metode *getContent*, *setContent* i *getBackup*. Uočimo da u svakom trenutku za objekat klase *SingleIntWithBackup* mora da važi (*backup*=0)∨(*backup* ima prethodnu vrednost polja *content*). Prva tvrdnja obezbeđena je konstruktorom klase koji implicitno poziva podrazumevani konstruktor klase *SingleInt* i na taj način postavlja backup=content=0. Drugi deo obezbeđuje refinisana verzija metode *setContent*. Neka su *c* i *b* instance klasa *SingleInt* i *SingleIntWithBackup* respektivno, a *pC* i *pB* indirekcije (u ovom slučaju pokazivači) na objekte ovih dveju klasa:

```
SingleInt c, *pC;
SingleIntWithBackup b, *pB;  //b.content = b.backup = 0
b.setContent(1);  //b.content = 1   b.backup = 0
c= b;
```

Operacijom *b.setContent(1)* u polje *backup* objekta *b* prepisuje se iz polja *content* vrednost 0, a zatim u polje *content* upisuje nova vrednost 1. Dodela *c= b* ne unosi nikakvu zabunu: vrednost polja *content* kopira se u odgovarajuće polje objekta *c*, tako da je po završenoj operaciji *c.content* jednako 1. Stanje objekta *c* potpuno je

definisano i on se u daljem ponaša kao objekat klase *SingleInt*. Posmatrajmo međutim sledeći segment

```
pB= new SingleIntWithBackup();  //backup = content = 0
pB→setContent(1);  //backup = 0, content = 1
pC= pB;
pC→setContent(2);  //trebalo bi da bude backup = 1, content = 2
```

Prvom naredbom formira se na hipu objekat klase *SingleIntWithBackup* čija su polja *content* i *backup* jednaka 0. Zatim se operacijom *pB→setContent(1)* u polje *backup* upisuje 0, a u polje *content* vrednost 1. Po izvršenoj operaciji *pC= pB* i pokazivač *pC* pokazuje na isti objekat. Najzad, *nad istim objektom* ali preko pokazivača *pC* vrši se operacija *setContent(2)*. Pitanje koje se postavlja jeste: hoće li se izvršiti verzija iz bazne klase zato što je pokazivač *pC* tako definisan ili iz izvedene, jer taj pokazivač faktički pokazuje na objekat izvedene klase? Odgovor je nedvosmislen: mora da se izvrši verzija iz izvedene klase! Naime, ako bi se izvršila verzija *setContent* iz bazne klase *SingleInt*, došlo bi do narušavanja integriteta objekta jer ta verzija ne sadrži, niti može sadržati, deo kojim se kopira vrednost *content* u polje *backup*. Pritom, "narušavanje integriteta" znači da stanje objekta prestaje da zadovoljava predikat naveden gore i taj objekat postaje neupotrebljiv. Ako bi se, dakle, izvršila verzija *setContent* iz pretka, kasnijim izvođenjem operacije *pB→getBackup()* dobili bismo pogrešan rezultat (0 umesto 1)!

U svetlu opšteg primera sa slike 8.1, to bi značilo da posle izvršenja dodele indirekcija *iPr←iPo*, u operaciji *iPr.m2()* mora biti aktivirana verzija redefinisane metode *m2* iz klase *Potomak*, a ne verzije iz klase *Predak*.

Iz ovih razmatranja izvodimo opšti zakon koji ćemo nazvati ***principom očuvanja integriteta objekta***:

- verzija neke metode koja se aktivira pri pozivu određena je isključivo objektom preko kojeg je pozvana, bez obzira na to da li se objektu pristupa preko imena ili indirekcijom.

Lako uočavamo da je ovaj zakon potpuno opšti i da važi u svim slučajevima: i kada se radi o objektu i kada se radi o indirekciji (pokazivaču ili referenci na objekat). Za nače primere: bez obzira na to što su pokazivači definisani kao pokazivači na baznu klasu, oni faktički pokazuju na objekat izvedene klase te će, u skladu sa navedenim zakonom, biti aktivirane verzije metoda *setContent* odn. *m2* iz izvedenih klasa *SingleIntWithBackup* odn. *Potomak*. U vezi s ovim, već sada moramo skrenuti pažnju na jednu izuzetno važnu stvar koja se tiče programskog jezika C^{++}:

- Programski jezik C^{++} ne obezbeđuje automatizam u izboru verzije redefinisane metode!!

Ovim se želi posebno skrenuti pažnja da, mimo onog što teorija i naš primer jasno nalažu, u programskom jeziku C^{++} operacija *pC→setContent()* nažalost aktivira

nekorektnu verziju *setContent* iz pretka - klase *SingleInt*. Da bi se ovaj problem rešio i pozvala korektna verzija iz potomka, C++ zahteva uvođenje nove vrste metoda, tzv. *virtuelnih metoda* o kojima će biti reči u posebnom odeljku ovog poglavlja.

Sa dodatim metodama, i uopšte dodatim članovima, stvar stoji drukčije: njima se ne može pristupiti ni preko objekta bazne klase niti preko indirekcije. Drugim rečima, po izvršenim operacijama *pC= pB* odnosno *iPr←iPo*, nisu dozvoljene operacije *pC→getBackup()* odn. *iPr.m3()*, bez obzira na činjenicu da objekti na koje pokazuju indirekcije te operacije omogućuju!

Sve ovo - aktiviranje redefinisanih verzija kod indirekcije s jedne, ali i zabrana korišćenja dodatih metoda s druge strane - vodi sledećem zaključku:

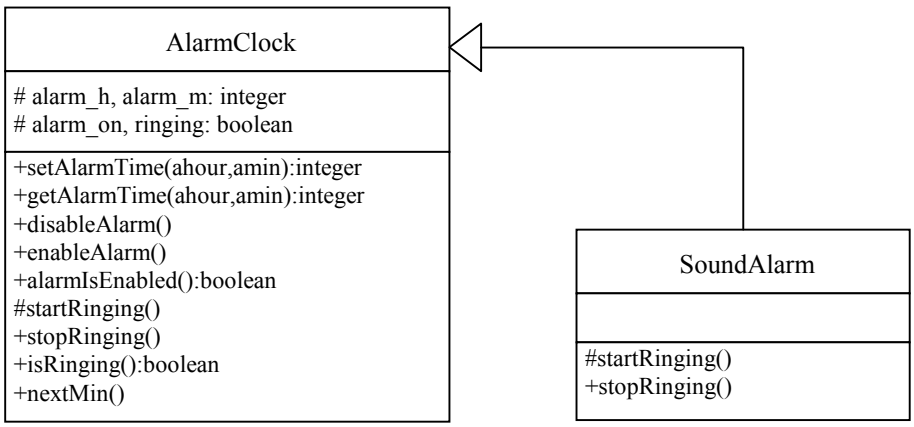
- Posle pridruživanja iPr←iPo, indirekcija iPr delimično menja ponašanje, i to tako što koristi redefinisane metode iz potomka, ali se *ne* ponaša kao potomak jer ne može da aktivira metode kojih u pretku nema.

Dodajmo da, pošto indirekcija iPr ipak sadrži adresu potomka, postoji način da se ona prinudi na ponašanje potomka, ali je za tu svrhu neophodno upotrebiti koercitivni polimorfizam.

Zanimljivo je uočiti da se upravo izvedena pravila vezana za inkluzioni polimorfizam pridruživanja indirekcije mogu objasniti koristeći konceptualnu definiciju objekta. Prvo, kako bismo objasnili pridruživanje indirekcije? Odgovor se nameće: pošto se kod ove vrste pridruživanja radi o samo jednom objektu sa dva pristupa (preko indirekcija na predak odn. potomak), jasno je da na konceptualnom nivou to znači formiranje *dva pogleda* na *isti* objekat. Na primer, budilnik možemo posmatrati i koristiti kao takav, ali i kao običan sat. U drugom slučaju, jednostavno ingorišemo alarmni mehanizam. Neophodnost poštovanja principa očuvanja integriteta objekta takođe ima jednostavno objašnjenje: ma kako ga posmatrali, kao budilnik ili kao sat, uređaj ostaje budilnik i tako se mora i ponašati!

Primer 8.1. Formirajmo klasu *SoundAlarm* kao naslednicu klase *AlarmClock* iz primera 7.2 redefinisanjem metoda *startRinging* i *stopRinging* tako da, umesto prostog postavljanja u stanje aktivnog alarma, zaista generišu odnosno prekinu zvuk (realizacija na C++ biće data u posebnom primeru). Deo dijagrama klasa koji povezuje *AlarmClock* i *SoundAlarm* prikazan je na slici 8.3. Podrazumeva se, dakle, da redefinisana metoda *startRinging* otpočinje generisanje zvuka, a *stopRinging* taj zvuk prekida.

Treba napomenuti da se metoda *startRinging* aktivira posredno, iz metode *nextMin*, u trenutku kada se dostigne vremenski trenutak predviđen za otpočinjanje alarma (videti primer 7.2).



Slika 8.3

Neka su *iAlCl* i *iSndAl* redom indirekcije na instance *AlarmClock* i *SoundAlarm*. U "normalnim" uslovima poruka *iAlCl.nextMin()* neće proizvesti nikakav zvuk jer to verzija *startRinging* iz pretka ne predviđa. Međutim, ako se izvrši pridruživanje

$$iAlCl \leftarrow iSndAl$$

tada će poruka *iAlcl.nextMin()* generisati zvuk kada dođe do aktiviranja alarma, zato što se, prema principu očuvanja integriteta objekta, iz *nextMin* poziva redefinisana metoda *startRinging*. Dakle, indirekcija *iAlCl* pridruživanjem menja ponašanje, ali je ta promena ograničena samo na korišćenje redefinisanih metoda.

Način na koji se posle pridruživanja ponašaju kako instanca klase tako i indirekcija opisuje se ***pravilom obraćanja*** članu klase koje glasi:

- objekat se obraća onoj verziji člana[70] koja je najbliža njegovoj klasi u lancu nasleđivanja.

Dakle, ako je neki član definisan u matičnoj klasi, odgovarajući objekat se obraća toj verziji. Ako je preuzet od neposrednog pretka tada se objekat obraća verziji iz pretka. Ako, pak, člana nema ni u neposrednom pretku traži se u pretku drugog nivoa itd. Primenjeno na indirekciju, pravilo propisuje obraćanje verziji člana objekta čiju adresu sadrži indirekcija, a ne verziji klase kojoj indirekcija pripada. Očigledno, ovo pravilo neposredno proističe iz principa očuvanja integriteta.

Sada ćemo nešto detaljnije razmotriti pridruživanje u vidu *operacije dodele* u programskom jeziku C^{++}. Ono se ostvaruje direktno nad instancama klasa ili pak

[70] Teorijski, ovo se odnosi na sve članove klase, a praktično samo na metode jer redefinisanje podataka-članova ili objekata-članova nije preporučljivo, a najčešće nije ni dozvoljeno.

putem pokazivača. Neka su *pr* i *po* instance klasa *Predak* odn. *Potomak*, a *pPr* i *pPo* pokazivači na objekte klasa *Predak* odn. *Potomak* sa slike 8.2. Operacija dodele može da uzme tri karakteristična oblika:

1. Dodela instance *po* instanci *pr*
2. Dodela dereferenciranog pokazivača *pPo* dereferenciranom pokazivaču *pPr*
3. Dodela vrednosti pokazivača *pPo* pokazivaču *pPr*.

Ponašanje operatora dodele ilustrovaćemo programskim segmentom koji je redukovan samo na deo neophodan za prikaz (prema slici 8.2).

```
class Predak {
. . . . . .
public:
  void m1() {...}
  void m2() {...}
};
```

```
class Potomak: public Predak {
. . . . . .
public:
  void m2() {...}     //Redefinisana metoda m2
  void m3() {...}     //Nova metoda
};
```

```
.....
Predak pr, *pPr;
Potomak po,*pPo;

pr= po;
pr.m1();    //Aktivira se m1 iz pretka
pr.m2();    //Aktivira se m2 iz pretka
pr.m3();    //Greska! Metoda m3 ne postoji u pretku

*pPr= *pPo;     // I
pPr→m1();       // S
pPr→m2();       // T
pPr→m3();       // O

pPr= pPo;
pPr→m1();    //Aktivira se m1 iz pretka
pPr→m2();    //Aktivira se m2 iz pretka
pPr→m3();    //Greska! Metoda m3 ne postoji u pretku
```

Kako se vidi, ponašanje dodele u oba programska jezika identično je i odgovara opštem obrascu, osim u jednom - važnom! - slučaju. Pošto ulogu indirekcije igraju pokazivači, posle dodele *pPr= pPo*, prema pravilima obraćanja, trebalo bi da se pozivom metode *m2* aktivira verzija *m2* iz potomka. Pokreće se, međutim, verzija *m2* iz pretka, a razlog je konstrukcija prevodioca. Dakle, rezimiramo:

1. Dodela potomka pretku u C^{++} dozvoljena je i za objekte i za pokazivače
2. Po izvršenoj dodeli predak uopšte ne menja ponašanje!

Na potpuno istu situaciju nailazimo i kod prenosa parametara. Ako je f metoda ili slobodna funkcija sa objektom, referencom ili pokazivačem kao parametrom, tada se f može pozvati sa argumentom koji je instanca potomka, ali se unutar funkcije parametar u celosti ponaša kao instanca pretka.

8.2. VIRTUELNE METODE

Virtuelne metode jesu posebna vrsta metoda, posebna po načinu pozivanja koji se potpuno razlikuje od poziva "običnih" metoda ili slobodnih potprograma. Omogućuju primenu pravila obraćanja članovima, ali treba odmah naglasiti da nisu uvedene samo zbog toga nego i zbog još jednog problema koji bi veoma ograničio bezbednu primenu nasleđivanja i na taj način praktično obesmislio celu objektnu metodologiju.

Primer 8.3. Da bismo ilustrovali neophodnost uvođenja virtuelnih metoda, realizovaćemo klasu *SoundAlarm* iz primera 8.1, slika 8.3. Klasa *SoundAlarm* je naslednica klase *AlarmClock* koja se razlikuje po tome što pored postavljanja podatka-člana *ringing* na vrednost *true* još i aktivira zvučnik. Da bi se to postiglo, treba redefinisati metodu *startRinging*, kao i metodu *stopRinging* čiji je novi zadatak da prekine zvuk. Uočimo vrlo važnu pojedinost: metodu *startRinging* poziva druga metoda, *nextMin*, u trenutku kada se vreme izjednači sa postavljenim vremenom alarma.

```
/*************************************************************

          KLASA SOUNDALARM (BUDILNIK SA ZVUKOM)

Naziv datoteke: SOUNDAL.HPP
*************************************************************/
#ifndef SOUNDAL_HPP
  #define SOUNDAL_HPP
#include <conio.h>
#include <dos.h>
```

```
#include "alclock.hpp"

class SoundAlarm: public AlarmClock {
protected:
  void startRinging();
public:
  SoundAlarm() {}
  ~SoundAlarm() {}
  void stopRinging();
};
void SoundAlarm::startRinging() {
  ringing= 1;
  while(!kbhit()) {nosound(); delay(200); sound(1000); delay(200);}
  if(getch()==0) getch();
}
void SoundAlarm::stopRinging() {
  ringing= 0; alarm_on= 0;
  nosound();
}
#endif
```

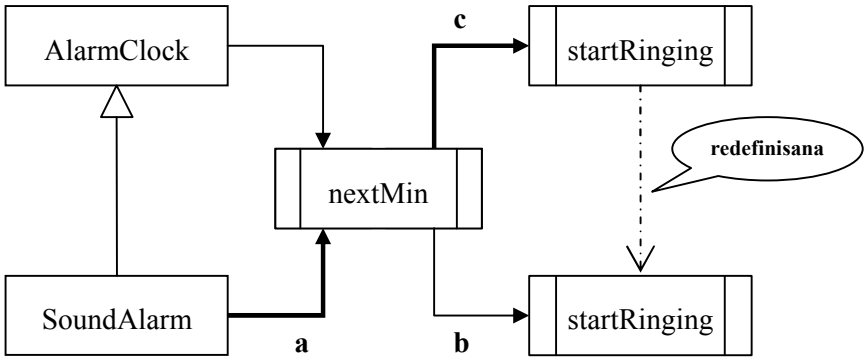
Funkcije *kbhit()* (detekcija pritiska na taster), *sound(f)* (proizvodnja zvuka frekvencije f), *nosound* (prestanak zvuka), *delay(m)* (pauza u trajanju *m* milisekundi) i getch() (prihvat jednog znaka sa tastature) nalaze se u nestandardnim bibliotekama *conio.h* i *dos.h*.

Naizgled, sve je u najboljem redu: u trenutku alarma treba da se aktivira redefinisana metoda *startRinging* iz *SoundAlarm*, proizvodeći tako zvuk. Odgovarajući primer primene jeste:

```
SoundAlarm sndClock;
sndClock.setTime(10,59);     //Postavljanje sata na 10,59
sndClock.setAlarmTime(11,0);     //Postavljanje alarma na 11,00
sndClock.enableAlarm();     //Omogućavanje alarma
sndClock.nextMin();     //Sat bi trebalo da "zvoni", ali zvuka NEMA!
```

Ako obratimo pažnju na odgovarajuće komentare, primetićemo da pokretanje metode *nextMin* što poziva redefinisanu metodu *startRinging* ni u jednom slučaju ne izaziva željeni efekat! Problem nije nimalo naivan jer proističe iz samog načina rada prevodioca odn. linkera. Uočimo, prvo, da metoda *nextMin* koja poziva redefinisanu metodu *startRinging* pripada pretku, klasi *AlarmClock*, i da je u fazi for-

miranja naslednika *SoundAlarm* ona već prevedena i uključena u matičnu klasu. Na mestu poziva *startRinging* u prevedenom kodu metode *nextMin* nalazi se instrukcija skoka na početak *startRinging* iz pretka. Drukčije i ne može biti jer u vreme stvaranja pretka potomak još ni ne postoji i u opštem slučaju ne mora se nikada ni pojaviti. Pošto se metoda *nextMin* prenosi bez ikakve izmene u potklasu, ona će u svakoj potklasi aktivirati isključivo verziju *startRinging* iz klase *AlarmClock*. Situacija je prikazana na slici 8.4 gde se vidi kako se, umesto željenog poziva linijom *a*-*b*, poziv uvek odvija linijom *a*-*c*.

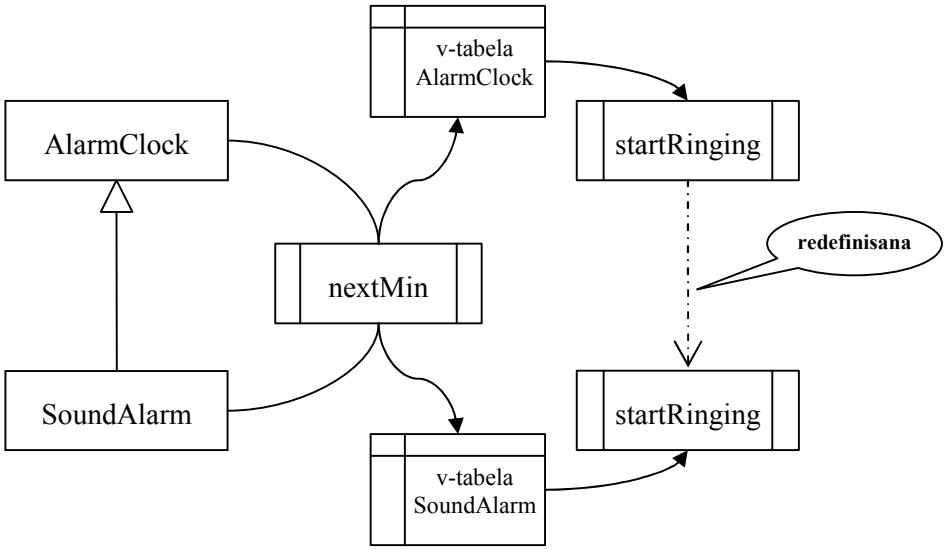


Slika 8.4

Očigledno, do greške dolazi ne zbog propusta proizvođača razvojnog sistema nego zbog neodgovarajućeg bazičnog postupka prevođenja i povezivanja. Ukoliko ovaj problem ne bi bio otklonjen nikada ne bismo bili sigurni da li će biti aktivirana pravilna verzija neke od redefinisanih metoda, što bi mehanizam nasleđivanja učinilo krajnje nesigurnim. Da bi se došlo do rešenja *morale* su biti uvedene posebne metode, nazvane ***virtuelnim metodama***, koje se odlikuju drukčijim mehanizmom poziva. Da bi se podvukla ta razlika, metode opisane do sada ubuduće ćemo nazivati ***nevirtuelnim metodama***.

Virtuelne metode razlikuju se od nevirtuelnih po tome što se na mestu poziva u prevedenom kodu ne nalazi direktan skok na njihov početak. Umesto toga, *na nivou klase* formira se posebna tabela koja, pored još nekih podataka, sadrži adrese svih virtuelnih metoda koje postoje u klasi (ako postoje). Prilikom poziva virtuelne metode prethodno se iz te tabele čita njena adresa i tek na bazi te adrese izvršava se instrukcija skoka. Pritom, svaki objekat sadrži adresu tabele u sopstvenom memorijskom prostoru, tako da se na bazi sadržaja datog *objekta* a ne njegove klase određuje verzija pozvane virtuelne metode, što je u skladu sa principom očuvanja integriteta objekta. Ove tabele nose naziv *v-tabele*.

U našem primeru, metode *startRinging* i *stopRinging* bile bi realizovane kao virtuelne, tako da ispravljena šema obraćanja ima oblik prikazan na slici 8.5.



Slika 8.5

Kao što se vidi, verzija metode *startRinging* koja će biti pozvana *zavisi od objekta* iz kojeg se aktivira metoda *nextMin* jer se u njemu nalazi adresa odgovarajuće tabele. Ako je u pitanju instanca pretka u njoj se nalazi adresa v-tabele pretka, dok instanca potomka sadrži drugu adresu tabele - one koja se vezuje za potomak. Kada se objektu *sndClock* klase *SoundAlarm* pošalje poruka *nextMin* ova metoda pobuđuje metodu *startRinging* tako što njenu adresu čita iz v-tabele, da bi se tek potom izveo skok na početak. Kako objekat *sndClock* pripada klasi *SoundAlarm*, u njegovom memorijskom prostoru stoji adresa v-tabele ove klase a ne klase *AlarmClock* u okviru koje je definisana metoda *nextMin* To znači da će se aktivirati ispravna verzija virtuelne metode *startRinging*, tj. ona iz klase *SoundAlarm*.

Zapazimo jednu veoma važnu činjenicu: da bi mehanizam virtuelnih metoda funkcionisao, one moraju biti proglašene za virtuelne *već u pretku*. Shodno tome, moramo se pridržavati sledećeg pravila:

- Prilikom formiranja neke klase, metode za koje se očekuje da budu redefinisane treba proglasiti za virtuelne.

Uvođenje virtuelnih metoda ima za nuspojavu stvaranje uslova za pravu primenu inkluzionog polimorfizma. Vratimo se na primer sa slike 8.2. Videli smo da u opštem slučaju polimorfna dodela pokazivača, oblika *pPr= pPo* ne izaziva željeni efekat pri pozivu redefinisane metode *m2* jer se ipak poziva verzija iz pretka. Šta se, međutim, dešava ako je *m2* virtuelna metoda? U tom slučaju objekat na kojeg pokazuje *pPo* koristi v-tabelu potomka. Kada se izvrši dodela, pokazivač *pPr* pokazuje na isti taj objekat, što znači da će porukom *pPr→m2()* biti aktivirana verzija *m2* iz potomka. Time se ostvaruje polimorfizam indirekcije definisan u te-

orijskom izlaganju na početku odeljka, tj. poštuje se princip očivanja integriteta te, shodno tome, i pravilo obraćanja članovim.

Neka je *pOb* pokazivač na objekat i *m* virtuelna metoda. Tada se samo na osnovu poruke *pOb→m3()* ne može unapred (tj. u toku prevođenja) znati kojoj se verziji metode *m* obraća pokazivač. Razlog je polimorfna dodela: ako u datom trenutku *pOb* pokazuje na objekat sopstvene klase biće aktivirana verzija *m* iz te klase; ako, međutim, *pOb* pokazuje na objekat iz nekog od potomaka biće aktivirana verzija koja važi u tom potomku. Sve izgleda tako kao da se odluka o povezivanju metode donosi u trenutku poziva, dakle ne u fazi prevođenja nego u fazi izvršavanja. Ovaj mehanizam povezivanja, koji je u stvari *posledica* postojanja virtuelnih metoda (pravi uzrok smo videli!) nosi naziv ***dinamičko ili kasno povezivanje*** (engl. "late binding" ili "dynamic dispatch").

S obzirom na potencijalne nesporazume oko uloge virtuelnih metoda dajemo kratak rezime dosadašnjih izlaganja:

1. Virtuelne metode *morale* su da budu uvedene da bi se nasleđivanje snabdelo mogućnošću bezbednog redefinisanja.
2. Dinamičko povezivanje se pojavilo kao posledica postojanja virtuelnih metoda i omogućilo sprovođenje principa očuvanja integriteta objekta.

Pošto je ovakav način povezivanja neizbežan, naoko čudno deluje to što u nekim jezicima nema virtuelnih metoda. Razlog je, u stvari, vrlo jednostavan: u tim jezicima virtuelnih metoda nema jer su sve metode virtuelne (ako se drukčije ne naredi)! Programski jezici paskal i C^{++} zadržali su nevirtuelne metode kao osnovnu vrstu zbog veće brzine pozivanja jer se ne zahteva prethodni pristup v-tabelama.

8.2.1. Realizacija virtuelnih metoda u C^{++}

Rukovanje virtuelnim metodama u C^{++} krajnje je jednostavno: sve što treba uraditi jeste deklarisati metodu kao virtuelnu dodavanjem rezervisane reči *virtual* u zaglavlju ispred tipa metode. Dakle, za metodu *method* tipa *T* sa listom parametara *param_list* prototip je

```
virtual T method(param_list);
```

pri čemu važe dva formalna pravila:

1. Metoda se proglašava virtuelnom tačno jedanput, u klasi u kojoj se prvi put pojavljuje kao takva. U svim potomcima metoda automatski zadržava ovu osobinu.
2. Redefinisana virtuelna metoda mora imati isti prototip kao originalna.

Ovako deklarisane virtuelne metode koriste se potpuno isto kao i nevirtuelne, tako da se iz same poruke ne može utvrditi da li je pozvana metoda nevirtuelna ili je virtuelna.

Prema tome, da bi klasa *SoundAlarm* funkcionisala kako treba, neophodno je metode *startRinging* i *stopRinging* proglasiti za virtuelne i to u pretku, tj. klasi *AlarmClock* koja, na taj način, dobija sledeći oblik:

```
/*****************************************************************

                    KLASA ALARMCLOCK (BUDILNIK)

Naziv datoteke: ALCLOCK.HPP
*****************************************************************/

#ifndef ALCLOCK_HPP
  #define ALCLOCK_HPP
#include "clock.hpp"

class AlarmClock: public Clock {
protected:
  int alarm_h,alarm_m;
  int alarm_on,ringing;
  virtual void startRinging() {ringing= 1;}
public:
  AlarmClock(): alarm_h(0),alarm_m(0),alarm_on(0),ringing(0) {}
  ~AlarmClock() {}
  void setAlarmTime(int ahour,int amin) {
    alarm_h= ahour; alarm_m= amin;
  }
  void getAlarmTime(int &ahour,int &amin) const {
    ahour= alarm_h; amin= alarm_m;
  }
  void disableAlarm() {alarm_on= 0;}
  void enableAlarm() {alarm_on= 1;}
  int alarmIsEnabled() const {return alarm_on;}
  virtual void stopRinging() {
    ringing= 0; alarm_on= 0;
  }
  int isRinging() const {return ringing;}
  void nextMin();
};
void AlarmClock::nextMin() {
  if((m=(m+1)%60)==0) h= (h+1)%24;
```

```
  if(alarm_on&&(h==alarm_h)&&(m==alarm_m)) startRinging();
}

#endif
```

Uz ovako definisanog pretka klasa *SoundAlarm* funkcioniše pravilno u obliku u kojem je data, jer njene metode *startRinging* i *stopRinging* automatski postaju virtuelne.

Da bi virtuelne metode mogle da se dohvate, u memorijskom prostoru objekta mora se nalaziti adresa v-tabele njegove klase. Stoga se postavlja pitanje šta je to, koja rutina, što tu adresu upisuje u objekat? Odgovor je: konstruktor i to bez posebne intervencije programera, tj. automatski. Sam konstruktor, naravno, nije virtuelan jer za tako nešto nema potrebe.

Nasuprot tome, destruktor može da bude virtuelan i čak se tako nešto i preporučuje da bi se postupak dealokacije dinamičkih objekata obavio korektno (videti odeljak o dinamičkim objektima).

Još nekoliko reči o inkluzionom polimorfizmu i polimorfizmu uopšte, a u vezi sa virtuelnim metodama. Pre svega, da bi se pobudile virtuelne metode određene dinamičkim povezivanjem, nužno je objektima pristupiti preko *pokazivača* ili pak putem *reference*. Vratimo se na jednostavnu hijerarhiju trouglova iz primera 7.1. Preformulišimo korenu klasu *Trougao* tako da metode *obim* i *povrsina* postanu virtuelne:

```
class Trougao {
protected:
  double s1, s2, s3;
public:
  Trougao(double x, double y, double z): s1(x), s2(y), s3(z) {}
  ~Trougao() {}
  double a() const {return s1;}
  double b() const {return s2;}
  double c() const {return s3;}
  virtual double obim() const {return s1+s2+s3;}
  virtual double povrsina() const;
};
double Trougao::povrsina() const {
  double p = 0.5*(s1+s2+s3);
  return sqrt(p*(p-s1)*(p-s2)*(p-s3));
}
```

Iz osobine virtuelnosti metoda obim() i povrsina() sledi da se pomoću pokazivača na klasu *Trougao* može putem dodele pristupati i objektima iz naslednika, *PravougliTrougao* i *RavnostraniTrougao*, pobuđujući njihove verzije tih metoda:

```
double p , v;
Trougao tr(3,5,6), *pTr;
PravougliTrougao pravTr(4,5);
RavnostraniTrougao ravTr(10);
. . . . . .
pTr= &tr;
p= pTr→povrsina();     //Aktivira se metoda iz klase Trougao
pTr= &pravTr;
p= pTr→povrsina();     //Aktivira se metoda iz klase PravougliTrougao
pTr= &ravTr;
p= pTr→povrsina();     //Aktivira se metoda iz klase RavnostraniTrougao
v= pTr→visina();     //Greska! Metoda Visina ne postoji u klasi Trougao
```

Pažnju privlači poslednji primer gde se putem pokazivača na osnovnu klasu *Trougao* pokušava pobuditi metoda visina(), što je i na početku ovog izlaganja, u teorijskom delu, označeno kao greška. S druge strane, kada napišemo pTr= &ravTr pokazivač pTr sadrži adresu objekta klase *RavnostraniTrougao* koja ima metodu visina(). Logično je zapitati se da li ipak postoji način da se pTr prinudi na ponašanje koje u potpunosti odgovara ponašanju objekta čiju adresu sadrži? Odgovor je potvrdan, a mehanizam je uključivanje koercitivnog polimorfizma preko eksplicitne konverzije (type cast). Sve što je potrebno jeste eksplicitno konvertovati pokazivač pTr u pokazivač na klasu *RavnostraniTrougao*. Dakle, poruka

```
((RavnostraniTrougao *)pTr)→visina()
```

je regularna i aktiviraće metodu visina() jer se izraz ((RavnostraniTrougao *)pTr) ponaša kao pokazivač na instancu klase *RavnostraniTrougao*. Ova mogućnost deluje privlačno, ali je ne treba precenjivati. Naime, eksplicitna konverzija je deo programa, što znači da u trenutku ispisivanja izraza programer već zna da treba pristupiti objektu klase *RavnostraniTrougao*. Jednostavniji i logičniji način je da se to ne obavi preko pokazivača na predak pTr nego direktno, putem odgovarajuće instance ili pokazivača na matičnu klasu.

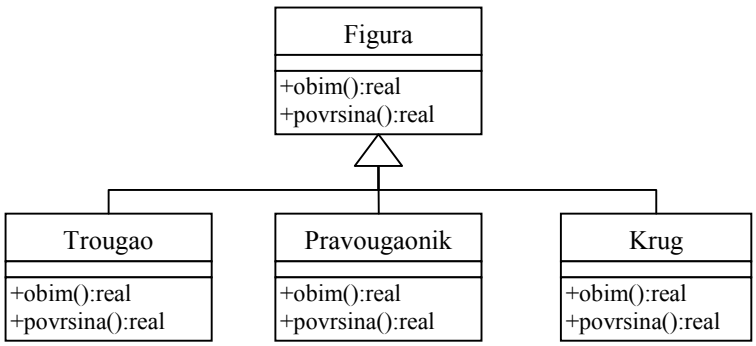
Na kraju, nešto i o polimorfnom prenosu argumenata u slobodnu funkciju ili metodu. Da bi se parametar iz klase pretka mogao zameniti argumentom iz potomka, a da se poštuje pravilo obraćanja članovima, parametar treba da bude referenca ili pokazivač, a redefinisane metode treba da budu virtuelne. Ako, dakle, definišemo funkciju (slobodnu ili metodu) sa prototipom

```
T f(Predak &);
```

ona može biti pozvana sa *f(po)* gde je *po* instanca potomka. Pritom, za virtuelne metode biće aktivirane verzije iz klase potomka.

8.3. APSTRAKTNE KLASE

Vratimo se još jednom na hijerarhiju tri klase trouglova iz primera 7.1 gde su, sada, metode *obim* i *povrsina* virtuelne. Ako hijerarhiju analizujemo sa dna ka vrhu, uočićemo da se predak, klasa *Trougao*, dobija odbacivanjem nekih osobina naslednika (metode *kateta1*, *kateta2*, *hipotenuza*, *stranica* i *visina*) i zadržavanjem onih koje su zajedničke. Zajedničke su metode *obim* i *povrsina* jer se mogu definisati za svaki trougao iako njihova realizacija (tj. način izračunavanja) nije ista u svim klasama. Uključimo sada u hijerarhiju još i klase *Pravougaonik* i *Krug*, tako da se za njih, zajedno sa klasom *Trougao*, može definisati zajednički predak *Figura*, slika 8.6.



Slika 8.6

Zapazimo važnu stvar: obim i površinu imaju i trougao i pravougaonik i krug, ali se način izračunavanja toliko razlikuje da se ne može obuhvatiti jednom skupnom formulom. Takođe, svaka druga zatvorena figura u ravni ima kako obim tako i površinu. Prema tome, potpuno je logično da se klasa *Figura* snabde metodama za izračunavanje obima i površine.

Primer drugi: klase *AutomobilskiSemafor*, *PesackiSemafor* i *UslovniSemafor* imaju zajedničke osobine:

- da se mogu uključiti i isključiti
- da mogu menjati boju koja se može očitati.

I ovde ima smisla formirati zajedničkog pretka *Semafor* koji zadržava pomenute osobine.

U oba slučaja, međutim, nailazimo na nepremostivu prepreku: za zajed-

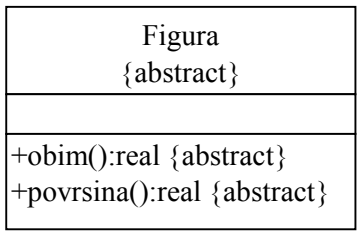
ničke metode znamo šta rade, ali ne i kako! Za obim i površinu proizvoljne figure ne postoji zajednička formula, kao što različite vrste semafora nemaju iste boje niti ih na isti način menjaju. Šta, dakle, znamo o metodama *obim* i *povrsina* u klasi *Figura*, odnosno o metodama *promenitiBoju* i *ocitatiBoju* u klasi *Semafor*? Znamo tip, naziv i listu parametara tojest znamo im *prototip*. Konkretna pak realizacija nije poznata.

- Metode koje imaju samo prototip (zaglavlje) nose naziv ***apstraktne metode***. Klasa koja ima bar jednu apstraktnu metodu nosi naziv ***apstraktna*** ili ***nepotpuna klasa***.

Jasno je da apstraktna klasa *ne može biti instancirana* jer se odgovarajućem objektu ne može poslati poruka koja, eksplicitno ili implicitno, sadrži poziv apstraktne metode. To pak ima za posledicu da potomak apstraktne klase koji može da se instancira, tj. da bude operativan, mora sadržati redefinisane apstraktne metode (sve). Pošto je redefinisanje apstraktne metode semantički specifičan postupak (ne može se redefinisati nešto što nije definisano), Mejer za ovu aktivnost uvodi poseban termin - ***operacionalizacija***. Inače, postupak je isti: metoda se realizuje u celini. Apstraktne metode su uvek virtuelne jer se *moraju* operacionalizovati. Potpunosti radi, dodajemo još nešto: kod apstraktnih metoda - formalno - poznat je tačan oblik zaglavlja (prototipa) i to bi bilo sve. Suštinski, međutim, o njima znamo mnogo više jer se već pri uvođenju definiše njihova namena, tj. semantika. Dakle, kod apstraktnih metoda poznato je *šta* rade, a *kak*o to obavljaju ostavlja se za operacionalizaciju.

Postojanje apstraktnih metoda otvara mogućnost za konstruisanje zaokruženih sistema klasa zasnovanih na hijerarhijskoj strukturi određenoj nasleđivanjem kao relacijom. U najvećem broju slučajeva ta hijerarhija ima strukturu stabla sa korenom klasom koja je apstraktna jer sadrži zajedničke osobine potomaka, bez obzira na to da li se mogu realizovati već u korenu ili ne[71]. Razvojni sistemi za objektno programiranje bazirani su upravo na takvim strukturama.

Apstraktna metoda u UML zadaje se dodavanjem fraze *{abstract}* iza prototipa u oznaci odgovarajuće klase. Sama klasa snabdeva se istim konstruktom koji se navodi u zaglavlju, slika 8.7.



Slika 8.7

[71] Na primer, u apstraktnoj klasi *Semafor* neke metode (uključivanje, isključivanje) mogu se realizovati već u korenoj, apstraktnoj klasi, a neke poput promene boje ne.

8.3.1. Apstraktne klase u C++

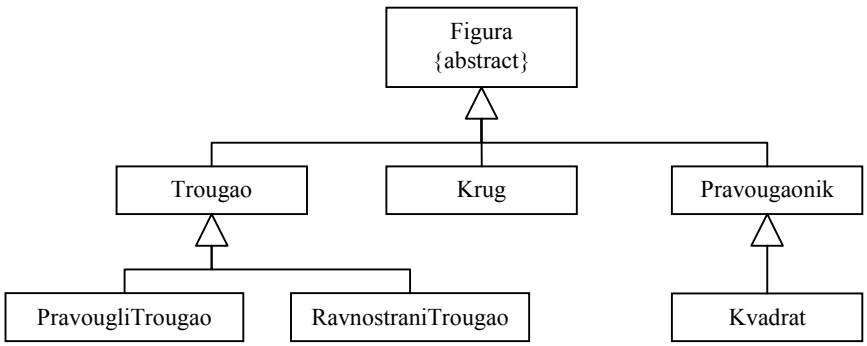
Apstraktna metoda u C++ ima prototip

virtual tip naziv(parametri) = **0**;

gde "izjednačavanje" metode sa nulom znači da je ona apstraktna i da prevodilac ne treba da očekuje kompletnu definiciju. Ideja verovatno potiče iz odlike funkcija u C-u da im naziv ima prirodu pokazivača na početak. Zahtev da apstraktna metoda bude virtuelna sadržan je već u sintaksi.

Sama pak apstraktna klasa ne nosi posebnu oznaku: dovoljno je da sadrži jednu apstraktnu metodu pa da bude automatski tretirana kao apstraktna.

Primer 8.1. U svojstvu primera realizovaćemo hijerarhiju nekoliko klasa što predstavljaju zatvorene geometrijske figure u ravni. Dijagram klasa prikazan je na slici 8.8. Klasa *Figura* koja se nalazi u korenu, apstraktna je zato što sadrži apstraktne metode *obim* i *povrsina*.



Slika 8.8

Realizacija na C++ ima sledeći oblik:

```
/*****************************************************************
        APSTRAKTNA KLASA SA GEOMETRIJSKIM FIGURAMA
Naziv datoteke: FIGURA.HPP
*****************************************************************/
#ifndef FIGURA_HPP
  #define FIGURA_HPP
class Figura  {
public:
  Figura() {}
  virtual ~Figura() {}
  virtual double obim() const = 0;
```

```
  virtual double povrsina() const = 0;
};
#endif
```

```
/*****************************************************************
                          KLASA TROUGAO
Naziv datoteke: TROUGAO.HPP
*****************************************************************/
#ifndef TROUGAO_HPP
  #define TROUGAO_HPP
#include <math.h>
class Trougao: public Figura {
protected:
  double s1, s2, s3;
public:
  Trougao(double x, double y, double z): s1(x), s2(y), s3(z) {}
  virtual ~Trougao() {}
  double a() const {return s1;}
  double b() const {return s2;}
  double c() const {return s3;}
  double obim() const {return s1+s2+s3;}
  double povrsina() const;
};
inline double Trougao::povrsina() const
{
  double p = 0.5*(s1+s2+s3);
  return sqrt(p*(p-s1)*(p-s2)*(p-s3));
}

#endif
```

```
/*****************************************************************
              KLASA PRAVOUGLI TROUGAO
Naziv datoteke: PRTROUG.HPP
*****************************************************************/
#ifndef PRTROUG_HPP
  #define PRTROUG_HPP
#include "trougao.hpp"
class PravougliTrougao: public Trougao {
public:
```

```
  PravougliTrougao(double kat1, double kat2):
   Trougao(kat1,kat2,sqrt(kat1*kat1+kat2*kat2)) {};
  virtual ~PravougliTrougao() {}
  double kateta1() const {return s1;}
  double kateta2() const {return s2;}
  double hipotenuza() const {return s3;}
  double povrsina() const {return 0.5*s1*s2;}
};
#endif
```

```
/*****************************************************************
                    RAVNOSTRANI TROUGAO
Naziv datoteke: RAVTROUG.HPP
*****************************************************************/
#ifndef RAVTROUG_HPP
  #define RAVTROUG_HPP
#include "trougao.hpp"
class RavnostraniTrougao: public Trougao {
public:
  RavnostraniTrougao(double s): Trougao(s,s,s) {}
  virtual ~RavnostraniTrougao() {}
  double stranica() const {return s1;}
  double visina() const {return 0.866025403*s1;}
  double obim() const {return 3*s1;}
  double povrsina() const {return 0.433012701*s1*s1;};
};
#endif
```

```
/*****************************************************************
              DATOTEKA SA KLASOM PRAVOUGAONIK
Naziv datoteke: PRAVOUG.HPP
*****************************************************************/
#ifndef PRAVOUG_HPP
  #define PRAVOUG_HPP
#include "figura.hpp"
class Pravougaonik : public Figura  {
protected:
  double s1, s2;
public:
  Pravougaonik(double x, double y): s1(x), s2(y) {}
```

```
  virtual ~Pravougaonik() {}
  double a() const {return s1;}
  double b() const {return s2;}
  double obim() const {return 2*(s1+s2);}
  double povrsina() const {return s1*s2;}
};
#endif
```

```
/*****************************************************************
              DATOTEKA SA KLASOM KVADRAT
Naziv datoteke: KVADRAT.HPP
*****************************************************************/
#ifndef KVADRAT_HPP
  #define KVADRAT_HPP
#include "pravoug.hpp"
class Kvadrat : public Pravougaonik  {
public:
  Kvadrat(double x): Pravougaonik(x,x) {}
  virtual ~Kvadrat() {}
  double stranica() const {return s1;}
  double obim() const {return 4*s1;}
  double povrsina() const {return s1*s1;}
};
#endif
```

```
/*****************************************************************
              DATOTEKA SA KLASOM KRUG
Naziv datoteke: KRUG.HPP
*****************************************************************/
#ifndef KRUG_HPP
  #define KRUG_HPP
#include <math.h>
#include "figura.hpp"
static const double PI=3.1415926535;
class Krug : public Figura  {
protected:
  double r;
public:
  Krug(double x): r(x) {}
  virtual ~Krug() {}
```

```
  double poluprecnik() const {return r;}
  double obim() const {return 2*r*PI;}
  double povrsina() const {return r*r*PI;}
};
#endif
```

Podsećamo da, prema pravilima jezika, u C++ nije dozvoljeno instanciranje apstraktne klase, tj. naredba oblika *Figura f;* rezultovaće porukom o grešci. Ono što je moguće jeste definisanje *pokazivača* na instance apstraktne klase, a sve zbog polimorfne dodele. Dajemo nekoliko primera primene, sa naglaskom na polimorfno korišćenje pokazivača:

```
double a, b, c, d;
Figura f;     //Greska! Apstraktna klasa ne moze se instancirati
Figura *pF;
Krug kr(10);
Kvadrat kv(5);
PravougliTrougao pravTr(5,6), *pPrav;
. . . . . . .
pFig= &Kr;
a= pFig→obim();
pFig= &kv;
b= pFig→povrsina();
pPrav= &pravTr;
pFig= pPrav;
c= pFig→obim();
d= ((PravougliTrougao *)pFig)→hipotenuza();
```

Na bazi definisane hijerarhije može se napraviti slobodna funkcija f sa formalnim parametrom klase *Figura* koja za stvarni parametar može da prihvati instancu bilo koje klase - naslednice iz hijerarhije. Uslovi su da formalni parametar bude referenca na klasu *Figura* i da funkcija koristi isključivo metode obim() i povrsina() jer su samo one definisane u korenoj klasi *Figura*. Kostur takve funkcije ima sledeću formu:

```
tip f(Figura &rF, ostali parametri)
{
  /* koriste se rF.obim() i rF.povrsina() */
}
```

Funkcija gornjeg oblika može se aktivirati bilo kojim od poziva f(kr,...), f(kv,...), f(pravTr,...), f(*pPrav,..) itd.

8.4. DINAMIČKI OBJEKTI

Pod dinamičkim objektima podrazumevamo objekte u dinamičkoj memoriji kojima se pristupa ne preko imena nego preko adrese. Razmatranje ove vrste objekata izdvojili smo u poseban odeljak zbog značaja koji imaju u *domenu realizacije*. Kada je u pitanju realizacija, u profesionalnom objektnom programiranju dinamički objekti igraju dominantnu ulogu jer se svi ili skoro svi objekti nalaze u dinamičkoj memoriji. Štaviše, neki od objektnih sistema poput jave ili Delphi-ja i nemaju drugih objekata osim dinamičkih. Opšti uslovi u kojima se dinamički objekti mogu koristiti su:

1. Pristup preko adrese (indirekcija)
2. Poseban postupak za konstrukciju koji podrazumeva istovremenu alokaciju dinamičke memorije
3. Poseban postupak za destrukciju zbog potrebe simultane dealokacije.

Dinamičkim objektima rukuje se kao i dinamičkim promenljivim: putem pokazivača odn. reference u jezicima koji nemaju pokazivače (npr. java). Notu složenosti unose aktivnosti konstrukcije i destrukcije pošto zahtevaju simultanu alokaciju i dealokaciju dinamičke memorije, što ima uticaja na odgovarajuća jezička sredstva.

Inače, dinamičke objekte ne treba mešati sa objektima koji imaju dinamičke članove jer se radi o ortogonalnim (nezavisnim) karakteristikama: dinamički objekat može i ne mora sadržati dinamičke članove, a isto važi i za obične (statičke) objekte.

8.4.1. Dinamički objekti u C^{++}

Dinamičkim objektima u C^{++} rukuje se pomoću pokazivača. Kompletna procedura kreiranja takvog objekta u C^{++} obavlja se operacijom *new*, ali uz istovremenu primenu nekog od konstruktora. Neka je *pK* pokazivač na objekat klase *K* koja ima konstruktor *K(parametri)* gde je *parametri* oznaka liste parametara. Tada je naredba za kreiranje dinamičkog objekta oblika

pK= new K(*argumenti*);

gde *argumenti* predstavlja listu argumenata. Na primer, dinamički objekat klase *Krug* iz primera 8.4 kreira se na sledeći način:

```
Krug *pKr;
. . . . . .
```

```
pKr= new Krug(10);
```

Kako izraz *new Krug(10)* vraća kao vrednost pokazivač na objekat klase *Krug*, dozvoljena je dodela pokazivaču na predak:

```
Figura *pFig;
. . . . . .
pFig= new Krug(10);     // ili
pFig= new Kvadrat(20);
```

Dealokacija dinamičkog objekta izvršava se standardnom formom operacije *delete*. Za opšti primer klase *K* sa početka odeljka to je prosta naredba

```
delete pK;
```

Sam postupak destrukcije ipak nije tako jednostavan. Prvo, objekat može imati dinamičkih članova koji takođe moraju biti oslobođeni tokom dealokacije. Da bi se to postiglo, operacija *delete* izvedena je tako da automatski aktivira destruktor. Zadatak programera je da u destruktor ugradi odgovarajući kod za dealokaciju svih dinamičkih članova. Međutim, to nije sve. Zbog polimorfne dodele ne može se unapred znati na kakav objekat pokazuje pokazivač - na objekat matične klase ili na neku od instanci potomaka. Posle polimorfne dodele

```
pFig= new Kvadrat(20);
```

pokazivač pFig pokazuje na instancu klase *Kvadrat*, što znači da se u naredbi

```
delete pFig;
```

očekuje implicitni poziv ne destruktora iz klase *Figura*, nego destruktora klase *Kvadrat*. U suprotnom, dealokacija ne bi bila izvedena pravilno, jer predak i potomak u opštem slučaju ne zauzimaju jednak memorijski prostor. Implicitni poziv destruktora pretka za pokazivač koji trenutno pokazuje na potomak bi stoga izazvao curenje memorije. Sve u svemu, da bi destrukcija dinamičkih objekata bila izvršena korektno treba se pridržavati sledećeg pravila:

- Ako se očekuje korišćenje dinamičkih objekta neke klase tada njen destruktor mora da bude virtuelan!

8.5. VIŠESTRUKO NASLEĐIVANJE

Po definiciji, ***višestruko nasleđivanje*** (engl. "multiple inheritance") jeste nasleđivanje dve ili više klasa, tj. preuzimanje sadržaja više klasa uz sve moguć-

nosti podešavanja koje postoje kod jednostrukog nasleđivanja. Formalno, dakle, višestruko nasleđivanje predstavlja uopštenje jednostrukog, ostvareno povećavanjem broja neposrednih predaka na proizvoljnu vrednost. Suštinski pak, višestruko nasleđivanje unosi važne novine od kojih su neke dobre, a neke i nisu, tako da nećemo pogrešiti ako kažemo da ova odlika spada u red najkontroverznijih u objektnoj metodologiji. Mejer [82], na primer, tvrdi da je višestruko nasleđivanje jedan od ključnih koncepata objektnog pristupa. Buč [3] je mnogo uzdržaniji kada kaže da je "višestruko nasleđivanje kao padobran: ne treba vam sve vreme, ali kada zatreba nije ga loše imati pri ruci". Razilaženje u pogledima na višestruko nasleđivanje vidi se i u samim objektnim jezicima: C^{++} ga ima, a paskal i java ne. Smatramo da nema svrhe upuštati se u besplodnu *pro et contra* diskusiju o tome da li je višestruko nasleđivanje *per se* dobro ili nije, jer se to uvek završava neodređenom frazom "istina je uvek negde na sredini". Umesto toga, usredsredićemo se na analizu njegovih osobina tako da čitalac može da stekne sliku o tome *kada i kako* se višestruko nasleđivanje upotrebljava za izradu kvalitetnog softvera.

Ono što višestruko nasleđivanje pruža, vidi se već iz same definicije: mogućnost kombinovanja sadržaja, tj. osobina više klasa, uz eventualne izmene i dopune. S druge strane, važno je poznavati limite ovog mehanizma da bi programer znao sa kakvim se problemima može suočiti prilikom primene. Na ograničenja višestrukog nasleđivanja nailazimo i u domenu modela i u domenu realizacije.

Prilikom ***modelovanja*** sukobljavamo se sa konceptualnim nedostatkom višestrukog nasleđivanja koji se ogleda u činjenici da se na ovu vrstu odnosa retko nailazi na konceptualnom nivou: prosto, retki su primeri pojmova koji imaju više superordiniranih pojmova. Inače, literatura obiluje primerima višestrukog nasleđivanja koji nemaju utemeljenje u domenu problema. Mejer [82] navodi bizaran primer višestrukog nasleđivanja: klasa *Vozač* modelovana kao potklasa klasa *Čovek* i *Automobil*. Dakle, čovek sa nogama, rukama, volanom, točkovima itd! U drugoj jednoj knjizi nailazimo na ništa manje apsurdan primer klase *Pegaz* kao višestruke naslednice klasa *Konj* i *Ptica*. Na stranu to što Pegaz nije klasa nego jedinka; ako se čak i složimo da je modelujemo kao klasu sa jednom jedinom instancom ostaju druge nesuvislosti: Pegaz ima kljun, perje, dlaku, kopita, kandže ... U stvarnosti, sve što Pegaz ima od odlika ptica jeste - krila. Ako pokušamo da ga definišemo (aristotelovskom) definicijom lako dolazimo do zaključka da je Pegaz konj s krilima te, shodno tome, klasa *Pegaz* jeste jednostruka naslednica klase *Konj* sa dodatom osobinom da ima krila. U svetlu ove kratke analize možemo naslutiti osnovnu konceptualnu manjkavost višestrukog nasleđivanja: struktura klasične definicije za *genus proximum* prihvata (jedan) najbliži viši rod, dok višestruko nasleđivanje predviđa više takvih rodova povezanih konjunkcijom (*Pegaz* je *Konj* **i** *Ptica*). To ipak ne znači da na konceptualnom nivou nema primera višestrukog nasleđivanja. Primerice, uređaj RADIO_SAT, sastavljen od radio aparata i sata najvernije se opisuje višestrukim nasleđivanjem klasa RADIO i SAT, jer se u svakom

trenutku ponaša i kao radio i kao sat. Klasa *IOFile* (ulazno-izlazna datoteka) takođe je dobar primer višestruke naslednice klasa *InputFile* (ulazna datoteka) i *OutputFile* (izlazna datoteka) pošto poseduje osobine oba pretka. Isto važi i za klasu *Konzola* koja je višestruki naslednik klasa *Tastatura* i *Ekran*. Iz ovih primera može se izvući jedna od preporuka za modelovanje višestrukim nasleđivanjem:

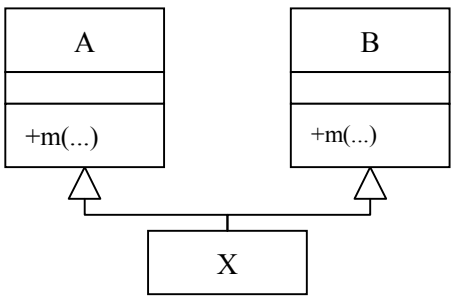
- višestruki naslednik se u svakom trenutku ponaša kao unija svih neposrednih predaka.

Imajući u vidu ovo pravilo lako je zaključiti zašto su primeri na početku izlaganja pogrešni: vozač se ni u jednom trenutku ne ponaša kao automobil; sličnost ponašanja Pegaza sa pticom svodi se samo na to da ima krila[72] i ništa više.

Još jedno "pravilo tri prsta" vezuje se za preuzimanje sadržaja predaka. Ako je prilikom definisanja naslednika nephodno *odbaciti* neke osobine (naročito podatke-članove), tada valja razmisliti o tome da li relacija višestrukog nasleđivanja zaista postoji. Naravno, ovo važi i za jednostruko nasleđivanje, ali je tu greška mnogo uočljivija nego kod višestrukog, te je pravilo korisnije u poslednjem slučaju. Na primerima klasa *RadioSat* i *Konzola* izgrađujemo još jedno praktično pravilo:

- Bezbedan slučaj višestrukog nasleđivanja je nasleđivanje semantički disjunktnih klasa (*Radio, Sat* odnosno *Tastatura, Ekran*), naravno pod uslovom da odgovarajući odnos ima smisla u domenu problema (radio-sat i konzola postoje!)

Višestruko nasleđivanje stvara određene probleme i u domenu ***realizacije***. Ključni je problem poklapanja naziva članova klasa, tzv. *kolizije imena*. Neka je klasa *X* višestruka naslednica klasa *A* i *B*. Neka u klasama *A* i *B* postoje metode[73] sa istim imenom, *m*. Pitanje koje se postavlja jeste koju od metoda, ili možda obe, preuzima klasa *X*, metodu *m* iz *A* ili *m* iz *B*?



Slika 8.9

Odgovor na ovo pitanje ne može se dati zaključivanjem na bazi "zdravog razuma".

[72] Pri tome, ptice nisu jedine životinje sa krilima!
[73] Važi ne samo za metode nego i za druge članove klase.

Klase *A* i *B* su, kao preci, potpuno ravnopravne i nijednoj od njih ne sme se dati prioritet. Generalno, za razrešavanje kolizije imena postoje dva načina:

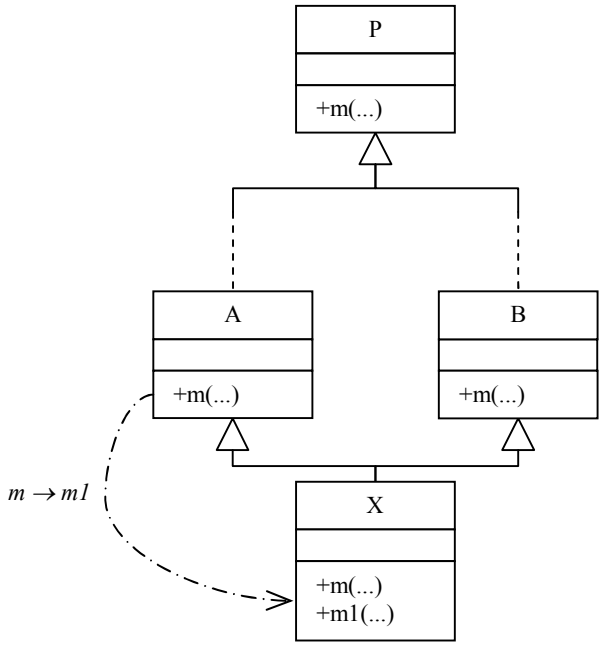
1. *Kvalifikovanje* pristupa gde se, prilikom pristupa, član kvalifikuje tačnim nazivom klase na koju se odnosi. U slučaju da su navedene klase realizovane na C^{++} pristup verzijama metode *m* redom iz *A* i *B* ostvarivao bi se kao *A::m(...)* odnosno *B::m(...)*. Napominjemo de se ovim pristupom otvara i mogućnost redefinisanja metode *m* tako da redefinisana metoda u *X* aktivira tačno određenu metodu *m* iz predaka ili nijednu od njih. Ovu vrstu mehanizma koristi C^{++}.
2. Drugi način je *preimenovanje*, gde jedna od metoda unutar mehanizma nasleđivanja menja ime bez promene realizacije. U našem primeru to bi značilo da je prilikom definisanja klase *X* neophodno navesti novo ime, npr. *m1*, bilo za *m* iz *A* bilo za *m* iz *B*, tako da klasa *X* jednu od metoda koristi pod originalnim, a drugu pod promenjenim imenom.

Nažalost, kolizija imena nije jedina poteškoća vezana za višestruko nasleđivanje. Drugi problem jeste tzv. ***ponovljeno nasleđivanje***. Naime, može se dogoditi da klase *A* i *B* sa slike 8.9 sadrže metode sa istim imenom *m* zato što imaju istog pretka, npr. *P*, u kojem je ta metoda definisana i samo preneta u klase *A* i *B*, slika 8.10.

U slučaju ponovljenog nasleđivanja klase *A* i *B* nemaju različite metode sa istim imenom *m*, nego se radi o dve *kopije* iste metode nasleđene iz zajedničkog pretka, klase *P*. Preimenovanje u ovom slučaju nema nikakvog smisla jer rezultuje pojavom dveju identičnih metoda sa različitim imenima, *m* i *m1*. Problem se još i zaoštrava ako se uzme u obzir da ovo važi i za ostale vrste članova klase - podatke-članove i objekte-članove, tako da (nažalost) važi

- Ponovljenim nasleđivanjem se više puta preuzima sadržaj iste klase (klase *P* na slici 8.10)!

Doda li se svemu tome činjenica da višestruko nasleđeni član može biti redefinisan u nekom od lanaca nasleđivanja (tada opet imamo različite verzije!), postaje prilično jasno zašto ima zamerki na mehanizam višestrukog nasleđivanja.



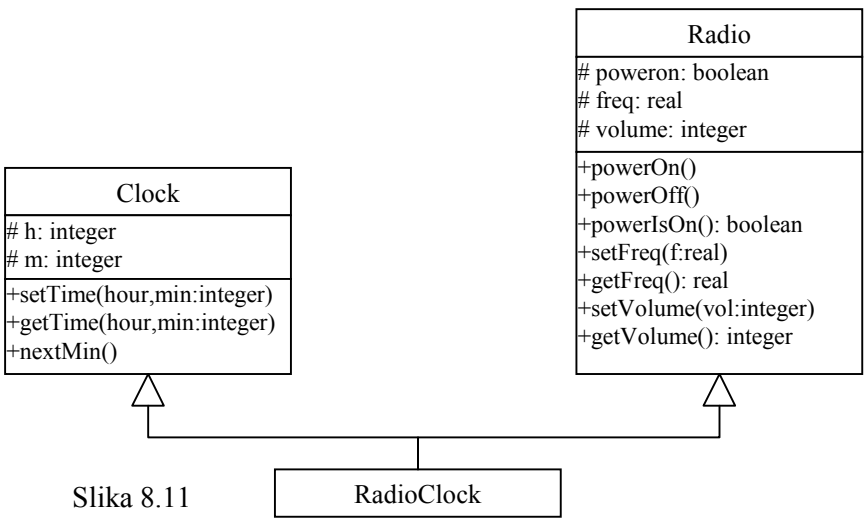
Slika 8.10

Višestruko nasleđivanje pokatkad se dovodi u vezu sa kompozicijom kao alternativom za postizanje istog cilja - objedinjavanja osobina više klasa. Pritom, zagovornici prve varijante (preimenovanja) to eksplicitno odbacuju dokazujući da su u pitanju dva sasvim različita odnosa: opšte - pojedinačno odnosno celina - deo, u čemu su sasvim u pravu. S druge strane, jezici koji ne podržavaju višestruko nasleđivanje (npr. paskal ili java) nude upravo kompoziciju kao sredstvo za aproksimaciju višestrukog nasleđivanja: klasu *X*, višestruku naslednicu klasa *A* i *B*, realizuju tako što se u klasu *X* uključuju dva objekta-člana, jedan iz klase *A*, a drugi iz klase *B*. Nikakvih nesporazuma ovde, u stvari, nema. U jezicima koji ne podržavaju višestruko nasleđivanje ono se realizuje putem kompozicije, tj. ne radi se o zameni višestrukog nasleđivanja kompozicijom nego o njegovoj aproksimativnoj *realizaciji* putem objekata-članova. Mada iznuđeno rešenje, realizacija višestrukog nasleđivanja preko kompozicije ima jednu prednost - unapred eliminiše koliziju imena i ponovljeno nasleđivanje. Nedostatak ovog pristupa pre svega je u tome što se veza nasleđivanja iz modela realizuje vezom celina - deo koja je u osnovi sasvim različita. Pored toga, nema mogućnosti ni za upotrebu polimorfizma. Konačno (i najneugodnije), ako se odlučimo za jaku inkapsulaciju (tj. ako zatvorimo objekte-članove) tada se broj metoda u nasledniku znatno povećava jer se praktično za svaku metodu svakog pretka mora realizovati posebna metoda u interfejsu višestrukog naslednika. Kada se sve odmeri, sledi zaključak da višestruko nasleđivanje realizujemo kompozicijom samo ako korišćeni programski jezik nema sopstvena sredstva.

Primer 8.5. Kao ilustraciju bezbednog višestrukog nasleđivanja formiraćemo klasu *RadioClock* što predstavlja model uređaja sa satom (klasa *Clock*) i radijom (klasa *Radio*). Smatra se da delovi koji modeluju sat odnosno radio nisu u međusobnoj interakciji pa tako primena višestrukog nasleđivanja ne dovodi do kolizije imena niti do ponovljenog nasleđivanja. Uočimo da se klasa *RadioClock* u svakom trenutku ponaša i kao član klase *Clock* i kao član klase *Radio* što i jeste motivacija za upotrebu višestrukog nasleđivanja. Klasa *Clock* opisana je u primeru 7.3. Klasa *Radio* sadrži sledeće članove:

- podatak-član *poweron* koji ima vrednost *true* ako je radio uključen, a inače *false*
- podatak-član *freq* koji ima vrednost frekvencije
- podatak-član *volume* sa vrednošću intenziteta zvuka izraženog kao procenat maksimuma
- funkcije-članice *powerOn* i *powerOff* za uključivanje-isključivanje radija
- funkcija-članica *powerIsOn* za proveru da li je radio uključen
- funkcije-članice *setFreq* i *getFreq* za zadavanje i očitavanje frekvencije
- funkcije-članice *setVolume* i *getVolume* za zadavanje i očitavanje jačine zvuka.

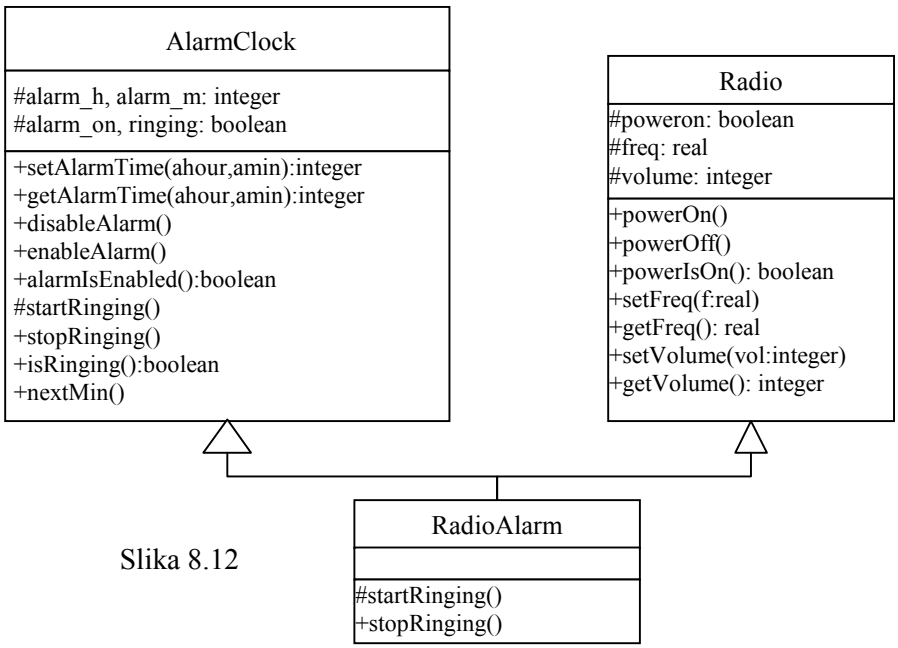
Klasa *RadioClock* nema sopstvenog sadržaja, već samo članove preuzete od predaka, što nije od značaja za primer. Dijagram klasa prikazan je na slici 8.11.



Slika 8.11

Primer 8.6. U prethodnom primeru klasa *RadioClock* ima dva potpuno disjunktna pretka, tako da, osim konstruktora i destruktora, nema sopstvenog sadržaja. U ovom primeru napravićemo klasu *RadioAlarm* koja podseća na prethodnu, ali čiji čiji delovi preuzeti od predaka imaju određenu interakciju. Klasa je zamišljena kao

kombinacija radija i budilnika (klasa *AlarmClock* iz primera 7.2) gde se u trenutku aktiviranja alarma uključuje radio. Klasa se izvodi iz klasa *AlarmClock* i *Radio* prema slici 8.12.



Slika 8.12

Da bi se obezbedilo uključivanje radija u trenutku aktiviranja alarma, mora se redefinisati (virtuelna) metoda *startRinging* koju poziva *nextMin* i to tako da se aktivira metoda *PowerOn* iz dela klase koji potiče od pretka *Radio*. Redefiniše se i metoda *stopRinging* tako da aktivira *powerOff* iz istog dela klase.

8.5.1. Višestruko nasleđivanje u C^{++}

Opšti oblik definicije klase *X* koja je višestruka naslednica klasa *A1*, *A2*, ..., *An* izgleda ovako:

$$\text{class X: } \mu_1 A1, \mu_2\ A2, ..., \mu_n An \{ ... \}$$

gde su μ_1, μ_2, ...,μ_n oznake modifikatora zaštite nasleđivanja koji, *svaki ponaosob*, mogu biti oblika *public*, *protected* ili *private*. Kako vidimo, zaštita se vezuje za svaki predak posebno tako da modifikatori μ_1, μ_2, ...,μ_n nisu međusobno uslovljeni. Imajući u vidu svrhu zaštite, to znači da su moguće razne kombinacije: nasleđivanje interfejsa od nekih klasa i istovremeno nasleđivanje realizacije od drugih; nasleđivanje kompletnog sadržaja od više predaka (kao u primeru 8.6); postavljanje zaštite nivoa *protected* na neke od klasa itd.

Kao primer, realizovaćemo klasu *RadioClock* opisanu u primeru 8.5. Realizacija klase *AlarmClock* nalazi se u primeru 7.3. Klasa *Radio* smeštena je u modul RADIO.HPP sa sledećim sadržajem:

```
/*************************************************************
                        KLASA RADIO
Naziv datoteke: RADIO.HPP
*************************************************************/
#ifndef RADIO_HPP
  #define RADIO_HPP
class Radio {
protected:
  int poweron;
  double freq;
  int volume;
public:
  Radio(): poweron(0),freq(100),volume(50) {}
  virtual ~Radio() {}
  void powerOn() {poweron= 1;}
  void powerOff() {poweron= 0;}
  int powerIsOn() const {return poweron;}
  void setFreq(double f) {freq= f;}
  double getFreq() const {return frq;}
  void setVolume(int vol) {volume= vol;}
  int getVolume() const {return volume;}
};
#endif
```

Klasa *RadioClock* je višestruka naslednica klasa *Clock* i *Radio* i nema sopstvenih osobina, te je programski kod vrlo jednostavan:

```
/*************************************************************
                    KLASA "RADIOCLOCK"
Naziv datoteke: RADCLOCK.HPP
*************************************************************/
#ifndef RADCLOCK_HPP
  #define RADCLOCK_HPP
#include "clock.hpp"
#include "radio.hpp"
class RadioClock: public Clock, public Radio {
```

```
};
#endif
```

Instance klase *RadioClock* imaju istovremeno sadržaj i jednog i drugog pretka, tako da su sve naredbe u sledećem primeru legalne:

```
RadioClock rdc;
. . . . . .
rdc.setTime(12,0);     //Postavljanje vremena na 12,00
rdc.setFreq(100;       //Postavljanje frekvencije na 100
rdc.setVolume(50);   //Postavljanje jacine zvuka na 50%
rdc.nextMin();         // Pomeranje sata na sledeci minut
rdc.powerOn();         //Ukljucivanje radija
```

Primer 8.7. Kao drugi primer realizovaćemo klasu *RadioAlarm* iz primera 8.6. Polazeći od gotovih klasa *AlarmClock* i *Radio*, novu klasu *RadioAlarm* realizujemo sasvim jednostavno: sve što treba uraditi jeste redefinisanje metoda *StartRinging* i *StopRinging* tako da posle poziva odgovarajućih metoda iz pretka *AlarmClock* aktiviraju još metode *powerOn* odnosno *powerOff*. Klasa *RadioAlarm* ima sledeći izgled:

```
/*****************************************************************
                        KLASA RADIOALARM
Naziv datoteke: RADIOAL.HPP
*****************************************************************/
#ifndef RADIOAL_HPP
  #define RADIOAL_HPP
#include "alclock.hpp"
#include "radio.hpp"
class RadioAlarm: public AlarmClock, public Radio {
protected:
  void startRinging();
public:
  void stopRinging();
};
void RadioAlarm::StartRinging() {
  AlarmClock::startRinging();
  powerOn();
}
void RadioAlarm::StopRinging() {
```

```
  AlarmClock::stopRinging();
  powerOff();
}
#endif
```

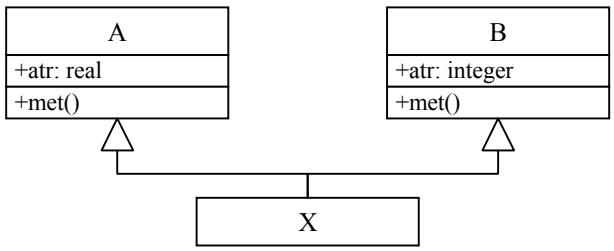
Primer primene:

```
RadioAlarm ral;
. . . . . .
ral.setFreq(100);          //nasledjeno od klase Radio
ral.setVol(50);            //nasledjeno od klase Radio
ral.setTime(10,l59);       //nasledjeno od klase Clock preko klase AlarmClock
ral.setAlarm(11,0);        //nasledjeno od klase AlarmClock
ral.enableAlarm();         //nasledjeno od klase AlarmClock
ral.nextMin();             //ukljucuje se radio; sada je ral.powerIsOn() = 1
```

Rećićemo dešto i o problemima kolizije imena, kao i o ponovljenom nasleđivanju u C^{++}. Programski jezik C^{++} nema mogućnost preimenovanja, tako da se ***kolizija imena*** razrešava kvalifikovanjem pristupa istoimenim članovima i to konstruktom

objekat.::naziv_pretka.član

Posmatrajmo dijagram klasa na slici 8.13.



Slika 8.13

U precima *A* i *B* nalaze se po jedan atribut i metoda istog imena (tipovi atributa i metoda mogu se ali ne moraju slagati). Realizacija klase *X* ima oblik

```
class X: public A, public B {
. . . . . .
};
```

Način korišćenja objekta klase *X* u nekom klijentu ilustruje se sledećim kodom:

```
X obX; double temp;
. . . . . .
temp= obX.atr;        //Greska! Ne zna se da li je atr iz A ili iz B
temp= obX.A::atr;     //Pristup atr preuzetom iz A
temp= obX.B::atr;     //Pristup atr preuzetom iz B
obX.met();            //Greska! Koja verzija met?
obX.A::met();         //Poziv met iz A
obX.B::met();         //Poziv met iz B
```

Iako direktno preimenovanje nije podržano, ono se bez poteškoća postiže redefinisanjem, mada rešenje nije naročito elegantno. Ono što treba uraditi, recimo za metodu met(), jeste dvostruko redefinisanje:

```
void X::metA() {A::met();}
void X::metB() {B::met();}
```

s tim što se broj metoda povećava jer i preuzete verzije met() ostaju u klasi *X* iako se ne koriste. Naravno, ukoliko se želi preuzeti samo jedna od dve metode (npr. *met* iz *A*) tada je najjednostavnije uvesti u *X* istoimenu metodu u kojoj će biti pozvana odabrana metoda:

```
void X::met() {A::met();}
```

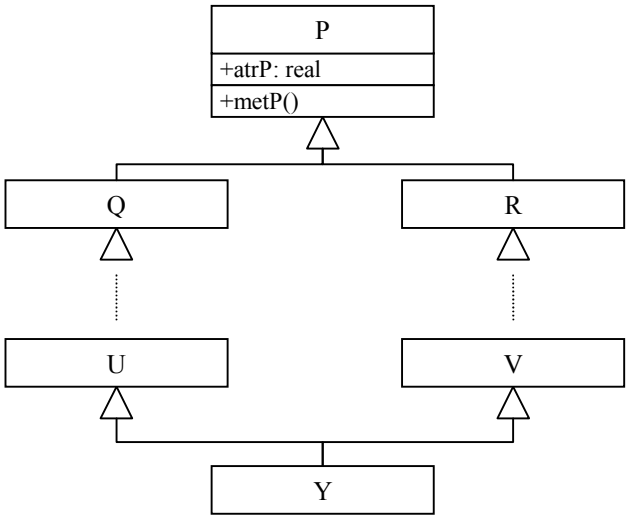
Problem ***ponovljenog nasleđivanja*** rešava se posebnom vrstom nasleđivanja - tzv. ***virtuelnim nasleđivanjem***. Posmatrajmo sistem sa ponovljenim nasleđivanjem prikazan na slici 8.14. Atribut *atrP* i metoda *metP* stižu u klasu *Y* u duplikatu: jedanput preko klasa *Q* i *V*, a drugi put preko *R* i *V*. Dakle, u klasi *Y* postoje po dva primerka *atrP* i *metP* što neće biti signalizirano ako nema pokušaja korišćenja. Problem se rešava tako što se neposredni naslednici *P*, klase *Q* i *R* vezuju za klasu virtuelnim nasleđivanjem:

```
class Q: virtual public P {...};
class R: virtual public P {...};
```

Ako sada klasu *Y* definišemo kao naslednicu *U* i *V*

```
class Y: public U, public V {...};
```

biće preneta samo po jedna kopija *atrP* i *metP*, te tako dvostrukog prenošenja više nema.



Slika 8.14

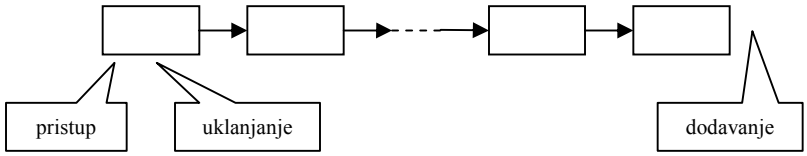
Napominjemo da ovo rešenje ima i jedan, potencijalno važan, nedostatak. Naime, da bi problemi ponovljenog nasleđivanja u podsistemu *U* - *V* - *Y* bili predupređeni, o tome se mora voditi računa već prilikom definisanja dela *P* - *Q* - *R* kada klase *U*, *V* i *Y* još ne postoje. Problem donekle ublažava činjenica da se ponovljeno nasleđivanje pojavljuje u komplikovanijim objektnim hijerarhijama koje se obavezno projektuju pa se unapred zna (npr. sa dijagrama klasa) da ponovljenog nasleđivanja ima.

8.5.2. Višestruko nasleđivanje implementacije

Višestruko nasleđivanje ima još jednu primenu u domenu realizacije kada se pojavljuje potreba za sledećim:

1. Ista klasa na nivou modela treba da ima više različitih oblika na nivou realizacije.
2. Način korišćenja klase (tj. skup metoda) mora biti isti za sve varijante realizacije.
3. Bar jedna varijanta izvodi se iz neke "treće" klase.

Da bismo razjasnili o čemu se radi prećićemo odmah na primer. Razmotrićemo jednu od najpoznatijih i najvažnijih struktura podataka, tzv. *red* ili *red čekanja* (eng. "queue"). U pitanju je linearna struktura u kojoj se pristupa prvom po redu elementu, uklanja se prvi, a dodaje se iza poslednjeg, slika 8.15.



Slika 8.15

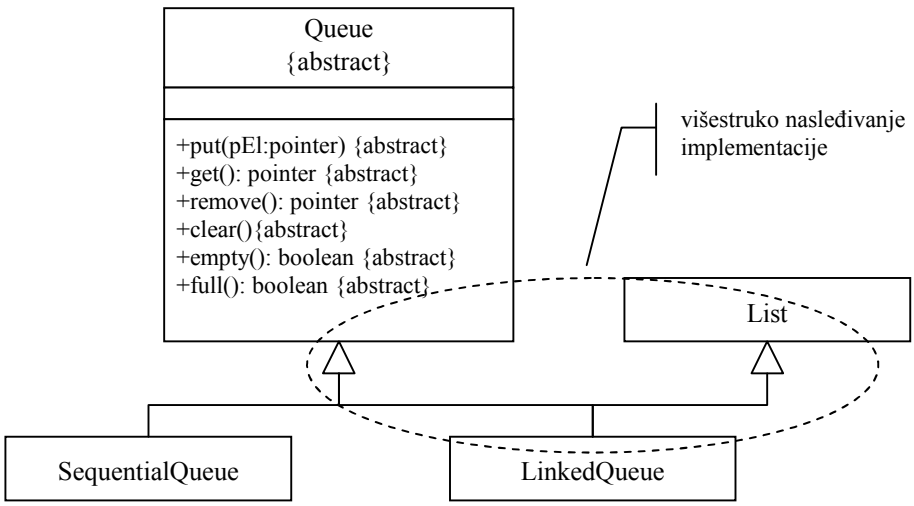
Red je dobio naziv red čekanja zbog jedinstvene osobine da se ranije uklanja onaj element koji je ranije i ušao u red. Ova osobina nosi naziv FIFO od First In First Out. Redovi se koriste kod prenosa podataka, upravljanja tastaturom i u diskretnoj računarskoj simulaciji, u stvari svugde gde je neophodna njihova FIFO osobina.

Red može da se realizuje na dva načina: spregnuto i sekvencijalno. Kod spregnute realizacije elementi reda nalaze se na hipu i spregnuti su pokazivačima tako da čine fizičku listu. Sekvencijalna realizacija podrazumeva implementaciju reda putem statičkog niza ograničenog kapaciteta u koji se elementi smeštaju sukcesivno. Prednost spregnute realizacije je u tome što je memorijski prostor uvek dovoljno veliki, tako da se red ne može prepuniti. Sekvencijalna realizacija podrazumeva ograničen memorijski prostor, pa prema tome i ograničen kapacitet reda, ali se zato operacije izvršavaju znatno brže. Odgovarajuće klase nazvaćemo redom *LinkedQueue* i *SequentialQueue* i one će za elemente imati generičke pokazivače. Bez obzira na način realizacije, operacije nad redom su iste:

- operacija *put* za dodavanje elementa u red
- operacija pristupa *get* kojom se očitava element na početku reda
- operacija *remove* za uklanjanje elementa iz reda
- operacija *clear* za pražnjenje reda (uklanjanje svih njegovih elemenata)
- operacija *empty* za proveru da li je red prazan
- operacija *full* za proveru da li je memorijski prostor reda popunjen (kod spregnute realizacije ova operacija uvek vraća vrednost *false* odn. 0 jer se smatra da se takav red ne može prepuniti).

Da bi se ispunio zahtev 1 obe realizacije moraju imati identične nazive operacija, što bi se lako postiglo: jednostavno, snabdeli bismo klase istoimenim operacijama. Međutim, takva realizacija ne ispunjava zahtev 2: na primer, ne može se napraviti neka funkcija *f* čiji je parametar red, a da se kao argument može poslati bilo spregnuta bilo sekvencijalna varijanta. Rešenje je da se formira apstraktna klasa, recimo *Queue* sa napred navedenim (apstraktnim) metodama, i da se iz nje izvedu klase *LinkedQueue* i *SequentialQueue*, te da se mehanizmom inkluzionog polimorfizma ispuni i zahtev 2. Pomenuta funkcija *f* imala bi za parametar tip *Queue&* što znači da bi kao argumente primila objekte obeju varijanti realizacije. Za razmatranja vezana za višestruko nasleđivanje najinteresantniji je zahtev 3. Ovde se kao "treća

strana" pojavljuje klasa sa jednostruko spregnutom listom jer ona već sadrži sve operacije potrebne za funkcionisanje spregnutog reda - ali i više! Naime, od onoga što sadrži klasa sa jednostruko spregnutom listom, za red su potrebne samo neke, dok ostale operacije vezane za listu (npr. čitanje proizvoljnog elementa ili upis na proizvoljno mesto) ne samo što nisu neophodne nego ne smeju da budu izvodljive. Očigledno, rešenje je da klasa *LinkedList* mora javno (public) naslediti apstraktnu klasu da bi se izvršila operacionalizacija apstraktnih metoda, i istovremeno mora naslediti klasu jednostruko spregnute liste (već smo je napravili pod imenom *List* u glavi 5), s tim što će nasleđivanje biti *private*, tj. nasleđivanje implementacije. Na slici 8.16 prikazan je dijagram klasa za našu mini hijerarhiju.

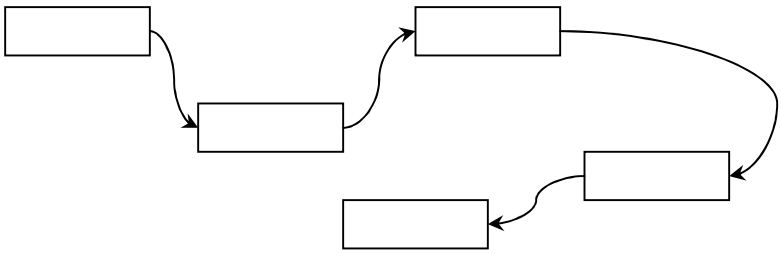


Slika 8.16

Metode apstraktne klase *Queue* treba da funkcionišu na sledeći način:

- metoda *void put(void* pEl)* upisuje element *pEl* koji je tipa generičkog pokazivača na kraj reda;
- metoda *void* get()* vraća sadržaj elementa sa početka reda; sadržaj elementa je generički (*void*) pokazivač;
- metoda *void* remove()* uklanja element sa početka reda i vraća uklonjeni element kao rezultat;
- metoda *void clear()* prazni red;
- metoda *int empty()* vraća 1 ako je red prazan, a 0 ako nije;
- metoda *int full()* vraća 1 ako je red popunjen, a 0 ako nije; kako je kapacitet spregnute varijante proizvoljan, za taj slučaj rezultat je uvek 0, dok kod sekvencijalne varijante *full* može vratiti i rezultat 1.

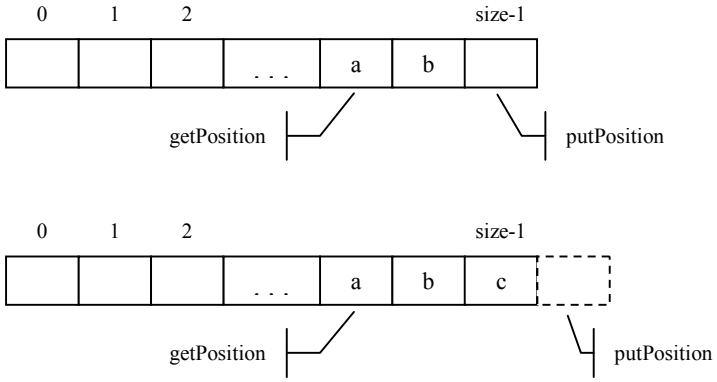
Prvi način realizacije koji ćemo razmotriti jeste *spregnuta realizacija* zbog koje je, uostalom, i sastavljen ovaj primer. Spregnuta realizacija smišljena je u svrhu izbegavanja problema prepunjenosti memorijskog prostora reda i to tako što se elementi reda smeštaju na hip i međusobno povezuju pokazivačima. Veza se uspostavlja tako što se u dati element reda, pored njegovog redovnog sadržaja, upisuje i pokazivač na sledeći element u redu, tako da lokacije na kojima se nalaze elementi više nisu od interesa jer se veza ostvaruje softverski, slika 8.17.



Slika 8.17

Ovako realizovan red ima prednost nad sekvencijalnim utoliko što se ne može prepuniti jer se memorijski prostor ne zauzima unapred i smatra se, s obzirom na kapacitet operativne memorije, da mesta za novi element uvek ima. Nedostatak je u tome što su u ovakvom redu operacije dodavanja i uklanjanja nešto sporije. Kako je najavljeno, sprezanje elemenata reda za potrebe klase *LinkedQueue* obavlja se *private* nasleđivanjem klase *List* koja nudi sav potreban repertoar operacija za tu svrhu. Ono što treba uraditi jeste operacionalizovanje apstraktnih metoda klase *Queue* pomoću preuzetih metoda klase *List*.

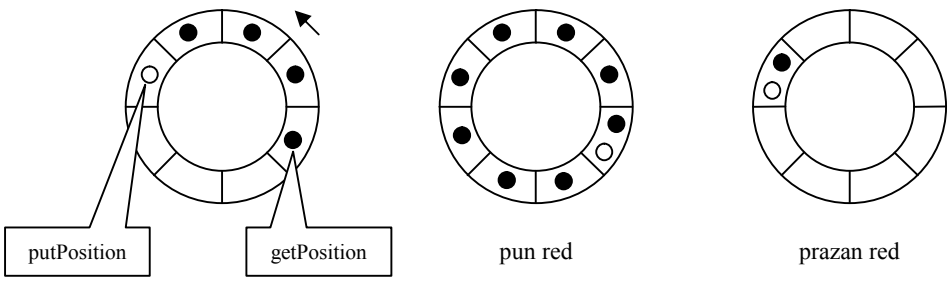
Drugi način realizacije, klasa *SequentialQueue*, koja predstavlja sekvencijalno realizovan red čekanja nije u neposrednoj vezi sa nasleđivanjem implementacije, ali ćemo je ipak realizovati u celosti da bi primer bio kompletan. "Sekvencijalno realizovan" red znači da se elementi upisuju u sukcesivne memorijske lokacije, u okviru unapred zauzetog segmenta memorije, zadatog kapaciteta koji će za naš primer biti 256 elemenata tipa *void**. To znači da se ovakav red programski realizuje kao niz. Elementi se ređaju od nižih indeksa ka višim, tako da se očitava i briše prvi zauzeti element, a uklanja poslednji. Neka je dužina niza *size*. Ova varijanta ima jednu manjkavost koja bi, ako je ne uklonimo, bila dovoljna da se odustane od sekvencijalne realizacije. Radi se o pojavi tzv. "lažne prepunjenosti" reda. Posmatrajmo stanje reda prikazano na slici 8.18 gore.



Slika 8.18

Oznaka *getPosition* odnosi se na indeks prvog elementa, a *putPosition* na mesto iza poslednjeg elementa koje je predviđeno za upis pri izvođenju operacije dodavanja. Kako se vidi na istoj slici, na donjem crtežu, već posle jednog dodavanja elementa npr. *c*, poslednja raspoloživa lokacija u nizu je zauzeta, te bi novi pokušaj dodavanja rezultovao porukom o prepunjenosti memorijskog prostora! To, očigledno, nije tačno jer na drugom kraju reda ima mesta, samo se ona ne mogu popunjavati elementarnim algoritmom za dodavanje. Da bi se ovaj problem rešio, red se realizuje cirkularno (kružno) tako da se, kada se stigne do kraja memorijskog prostora, sa dodavanjem nastavlja na početku (naravno, ako ima mesta). Cirkularno rešenje prikazano je na slici 8.19 levo.

Nažalost, ovo nije dovoljno jer se pojavljuje novi problem: (stvarno) popunjen red ne može se razlikovati od praznog reda jer u oba slučaja važi *getPosition = putPosition*. Ova situacija prikazana je na slici 8.19 u sredini i desno.

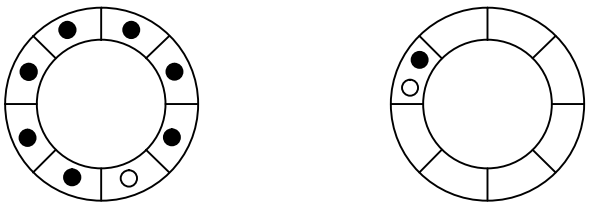


cirkularni red

Slika 8.19

Za ovaj problem ima više rešenja, a ovde ćemo opisati jedno, u svoje vreme prilično originalno. Ideja se sastoji u tome da se žrtvuje jedna lokacija u koju nije

dozvoljen upis, što znači da je memorijski prostor reda za 1 lokaciju veći od stvarnog kapaciteta (dakle jednak 257). Na taj način detektuje se razlika između praznog i punog reda: dok je kod praznog reda i dalje *getPosition* = *putPosition*, pun red prepoznaje se po tome što su pozicije *getPosition* i *putPosition* susedne (u cirkularnom smislu), slika 8.20. Inače, sekvencijalni red smešta se u interni niz *elements* čija je dužina za 1 veća od kapaciteta CAPACITY koji se, sa svoje strane, realizuje kao konstantno statičko polje sa vrednošću 256. Interno polje *size* predstavlja stvarnu veličinu memorijskog prostora, za 1 veću od kapaciteta (tj. jednako je 257).



pun red prazan red

Slika 8.20

Sledi kompletan kod apstraktne klase *Queue* i klasa *LinkedQueue* i *SequentialQueue*. Klase se nalaze u biblioteci QUEUES sa zaglavljem QUEUES.HPP i telom QUEUES.CPP.

```
/*****************************************************************

            ZAGLAVLJE BIBLIOTEKE SA REDOVIMA

Naziv datoteke: QUEUES.HPP
*****************************************************************/
#ifndef QUEUES_HPP
  #define QUEUES_HPP

#include "list.hpp"

class Queue {
public:
  virtual void put(void *) = 0;
  virtual void* get() = 0;
  virtual void* remove() = 0;
  virtual void clear() = 0;
  virtual int empty() = 0;
  virtual int full() = 0;
```

```
};

class LinkedQueue: public Queue, private List {
public:
  virtual ~LinkedQueue(){};
  void put(void *);
  void* get();
  void* remove();
  void clear();
  int empty();
  int full();
};

class SequentialQueue: public Queue {
private:
  static const int CAPACITY=256;
  int size, getPosition, putPosition;
  void* elements[CAPACITY+1];
public:
  SequentialQueue() {
    size= CAPACITY+1;
    getPosition= putPosition= 0;
  }
  virtual ~SequentialQueue() {}
  void put(void *);
  void* get();
  void* remove();
  void clear();
  int empty();
  int full();
};

#endif
```

```
/*****************************************************************

                    TELO BIBLIOTEKE SA REDOVIMA

Naziv datoteke: QUEUES.CPP
*****************************************************************/
#include "queues.hpp"
```

```
// SPREGNUTO REALIZOVAN RED

void LinkedQueue::put(void* pEl) {
  gotoLast();    //metoda klase List, kao i ostale sto slede
  insertAfter(pEl);
}
void* LinkedQueue::get() {
  gotoFirst();
  return read();
}
void* LinkedQueue::remove() {
  gotoFirst();
  return List::remove();
}
void LinkedQueue::clear() {
  List::clear();
  }
int LinkedQueue::empty() {
  return List::empty();
}
int LinkedQueue::full() {
  return 0;
}


// SEKVENCIJALNO REALIZOVAN RED

void SequentialQueue::put(void* pEl) {
  elements[putPosition]= pEl;
  putPosition= (putPosition+1)%size;
}
void* SequentialQueue::get() {
  return elements[getPosition];
}
void* SequentialQueue::remove() {
  void* removed = elements[getPosition];
  getPosition= (getPosition+1)%size;
  return removed;
}
void SequentialQueue::clear() {
```

```
  getPosition= putPosition;
}
int SequentialQueue::empty() {
  return getPosition==putPosition;
}
int SequentialQueue::full() {
  return getPosition==(putPosition+1)%size;
}
```

Prodiskutujmo, ukratko, šta se postiže organizacijom klasa sa slike 8.16. Prvo, uključivanjem metoda sa identičnim prototipovima u klase *LinkedQueue* i *SequentialQueue* ostvaruje se uniformnost ponašanja redova bez obzira na njihovu realizaciju. Dva reda *q1* i *q2* definisani sa

LinkedQueue q1; SequentialQueue q2;

koriste se na identičan način upotrebom metoda *get*, *put*, *remove* itd. čiji načini poziva ne zavise od realizacije. Recimo, naredbama

```
q1.remove();
q2.remove();
```

postiže se isti efekat - uklanjanje elementa sa početka reda. Sve što programer treba da zna (a to ga i interesuje!) jeste činjenica da se operacije nad *q1* izvršavaju nešto sporije, ali da kod *q1* ne može doći do prepunjenosti. Obrnuto, *q2* radi brže, ali ima ograničen kapacitet, u ovom slučaju 256.

Dalje, postojanje zajedničke (apstraktne) natklase *Queue* u ovom slučaju omogućuje punu primenu inkluzionog polimorfizma. Ako, recimo, realizujemo neku funkciju (slobodnu ili metodu) sa prototipom

tip f(Queue&, *ostali parametri*)

možemo biti sigurni da će se ona potpuno jednako ponašati bez obzira na to da li joj je prosleđen argument *q1* ili *q2*. Ukoliko se za pristup redovima koriste pokazivači, tada pokazivačem *Queue *pQ* koji je definisan kao pokazivač na apstraktnu klasu *Queue* možemo manipulisati redom nezavisno od njegove fizičke realizacije. Na primer, ponašanje sekvence oblika

```
pQ→put(x);
pQ→put(y);
pT= pQ→remove();
```

sasvim je nezavisno od toga da li *pQ* pokazuje na objekat klase *LinkedQueue* ili na objekat klase *SequentialQueue*!

Višestruko nasleđivanje implementacije primenjeno je na realizaciju klase *LinkedQueue*. Svrha je svakako da se olakša njena realizacija: umesto da se spregnuta realizacija izvede ispočetka, odgovarajući mehanizmi u vidu metoda preuzimaju se iz gotove klase *List*. S obzirom na to da klasa *LinkedQueue* već nasleđuje klasu *Queue*, klasa *List* mora se preuzeti višestrukim nasleđivanjem. Konačno, klasa *List* ima mnoštvo metoda koje ne spadaju u ponašanje reda i ne smeju se koristiti u klijentskom softveru, te je zato njeno nasleđivanje zaštićeno nivoom *private*. Uzgred, ako bismo želeli da metode klase *List* budu dostupne potklasama klase *LinkedQueue* i to bi bilo izvodljivo: sve što bi trebalo uraditi jeste izvesti nasleđivanje kao *protected*.

9. GENERIČKE KLASE I POTPROGRAMI

Osobina tzv. generičnosti sreće se kod posebnih vrsta klasa-tipova s jedne i potprograma s druge strane. Generička klasa (tip podataka) i generički potprogram karakterišu se time što u definiciji sadrže bar jednu klasu-tip kao slobodnu promenljivu, tj. parametar. Generičnost se pojavljuje i kod klasa i kod tipova podataka i to u identičnoj formi. Takođe, slobodne promenljive (parametri) u oba slučaja mogu biti i klase i tipovi. Stoga ćemo u nastavku govoriti o ***generičkim parametrima*** imajući u vidu da se sve što kažemo odnosi i na tipove podataka koji su parametri i na klase-parametre.

Generičnost, inače, nije nikakva novost: generički tipovi naziru se već u fortranu, a eksplicitno, kao generički, pojavljuju se u originalnom paskalu pod imenom strukturiranih (izvedenih) tipova. Primerice, u svim procedurnim jezicima tip niza predstavlja jedan od najvažnijih izvedenih tipova. Udubimo li se, međutim, u njegovu definiciju zapazićemo da niz kao tip *per se* ne postoji, već se može govoriti samo o realnim nizovima, celobrojnim nizovima, znakovnim nizovima itd., dakle o familiji međusobno veoma sličnih tipova. Tip niza, kao takav, očigledno ima prirodu *šeme* što objedinjava osobine praktično proizvoljnog broja konkretnih nizova sa različitim opsezima i tipovima elemenata. Da bismo definisali promenljivu tipa niza neophodno je u odgovarajućem programu izvršiti konkretizaciju zadavanjem opsega i tipa elemenata. Drukčije rečeno, tip niza je generički tip parametrizovan tipom elemenata, a u paskalu čak i tipom indeksa. Generički tip niza ugrađen je u programske jezike, no tu se generičnost ne iscrpljuje. Već neki procedurni jezici (npr. ada) pružaju programeru mogućnost definisanja sopstvenih generičkih tipova.

Posmatrajmo klasu stek. Algoritmi osnovnih operacija nad stekom, *top*, *pop*, *push* i *empty*, ni na koji način ne zavise od tipa njegovih elemenata[74], tj. isti su za stek znakova, celobrojni stek, adresni stek itd. Zato je vrlo pogodno posedovati mehanizam za realizaciju objedinjujuće klase sa nazivom npr. *GenStack<T>* gde je *T* generički parametar koji se pridružuje elementima steka, a čiji se oblik zadaje tek prilikom ***konkretizacije*** u klijentu. Da bi se izvršila konkretizacija, u klijentu treba

[74] kao što ni operacija indeksiranja ne zavisi od tipa elemenata niza

napisati nešto poput *GenStack<char>* gde sada tip *char* na svakom mestu zamenjuje tip *T* iz definicije generičke klase i predstavlja ***generički argument***. U proceduralnom programskom jeziku definicija *int x[100]* je konkretizacija generičkog tipa niza gde je tip elementa konkretizovan sa *int*. Lako se uočava analogija između poziva funkcije u kojem argumenti zamenjuju parametre i konkretizacije generičkog tipa (ili klase) gde se generički parametri zamenjuju generičkim argumentima.

Još upadljiviji - zbog složenosti - jeste primer jednostruko spregnute liste. Naime, sve liste poseduju jednu ili više operacija pristupa, dodavanja i uklanjanja elementa, kao i niz drugih, složenijih operacija. Bez generičnosti bili bismo prinuđeni da svaku konkretnu listu realizujemo iznova, bez obzira na to što se kodovi razlikuju neznatno, samo na mestima gde se navodi tip elementa. Umesto toga, daleko je zgodnije definisati klasu *List<T>* sa generičkim parametrom *T* koji se pridružuje elementima liste. Ceo posao oko izrade liste elemenata zadatog tipa svodi se sada na konkretizaciju pridruživanjem konkretnog tipa generičkom parametru *T*.

Kao sve do sada razmatrane karakteristike objektne metodologije, generičnost proističe iz konceptualne prirode klase - štaviše na vrlo upadljiv način. Naime, ***generičke pojmove*** srećemo na svakom koraku: pojam skupa, na primer, jeste tipičan generički pojam jer je parametrizovan tipom elemenata. Skup brojeva, skup ljudi i skup stolica jesu samo konkretizacije generičkog pojma skupa dobijene zamenom generičkog parametra "element" generičkim argumentima redom "broj", "čovek" i "stolica". Generički pojmovi su i spisak (nečega), kolekcija (nečega), tovar (nečega) i sl.

Primarna *tehnička svrha* generičnosti očigledno je *višekratna upotreba*. Budući da to važi i za nasleđivanje, odmah napominjemo da ovo sredstvo ni u kom slučaju nije protivstavljeno nasleđivanju. Naprotiv, generičnost i nasleđivanje su ortogonalne osobine i međusobno se dopunjuju u pokušaju postizanja najboljih uslova za višekratnu upotrebu. Nasleđivanje obezbeđuje mogućnost podešavanja, dok generičnost proširuje domen na više klasa odn. tipova. Na primer, iz generičke klase *Queue<T>* što predstavlja red čekanja sa elementima tipa *T* nasleđivanjem izvodimo klase *LinkedQueue<T>* ili *SequentialQueue<T>* (spregnuto i sekvencijalno realizovani redovi sa proizvoljnim tipom elemenata *T*), a zatim konkretizacijom recimo *SequentialQueue<char>* - sekvencijalno realizovan red čiji su elementi znakovi.

Generičnost ne podrazumeva samo tipove-klase kao parametre (tj. slobodne promenljive u definiciji), već se proteže i na konstante. Tip niza u C-u definisan sa

```
TipElementa  nazivNiza[BROJ_ELEMENATA];
```

parametrizovan je kako tipom elemenata tako i njihovim brojem koji je po prirodi

konstanta. Kada napišemo

typedef char nizZnakova[100];

izvršili smo jednu konkretizaciju generičkog tipa niza gde je *nizZnakova* naziv konkretnog tipa niza, *char* konkretizovani tip elemenata, a 100 konkretizovan broj elemenata.

Druga kategorija koja može da poseduje osobinu generičnosti jesu potprogrami. Generički potprogram ima generičke parametre kao deo definicije sopstvenih parametara ili izlaza. Potprogram na nekom uopštenom jeziku oblika

f(a:T, b:U, n:integer): T;

ima parametre *a* i *b* čiji su tipovi parametrizovani, kao i celobrojni parametar *n*. Tip izlaza potprograma takođe je parametrizovan i, kako vidimo, poklapa se sa tipom argumenta *a*. Standardni način za korišćenje generičkih potprograma je prethodna konkretizacija zamenom generičkih parametara generičkim argumentima npr.

f(char,real,integer): char;

gde se tip *T* konkretizuje sa *char*, a tip *U* sa *real*.

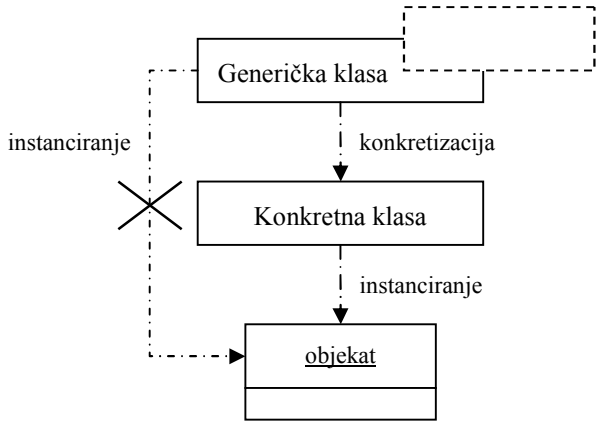
Već se može primetiti - a u nastavku će biti još uočljivije - da u pogledu generičnosti postoje sličnosti, ali i neke razlike između generičkih klasa i generičkih potprograma. Osnovna sličnost ogleda se u tome što je u oba slučaja generičnost oblik *parametarskog polimorfizma* (videti poglavlje 7). Razlika je, dakako, u tome što su generički potprogrami usmereni prvenstveno ka algoritmu, dok su generičke klase (i tipovi) znatno šira kategorija jer obuhvataju kako algoritme tako i podatke. Na ovo skrećemo pažnju samo zato što postoje određene razlike u sredstvima za realizaciju generičnosti klasa-tipova s jedne, odnosno potprograma s druge strane.

9.1. GENERIČKE KLASE

Generičke klase karakterišu se dvema opštim osobinama:

1. U definiciji eksplicitno sadrže druge klase-tipove kao generičke parametre. Ako je *GenClass* generička klasa parametrizovana klasama-tipovima T_1, ..., T_n označićemo je, nezavisno od programskog jezika, sa *GenClass*$<T_1,...,T_n>$.
2. Generičke klase ne mogu se direktno instancirati u smislu kreiranja nekakvog "generičkog" objekta. Da bi se izvršilo instanciranje, generička klasa mora se *konkretizovati*, tj. pretvoriti u konkretnu klasu dodeljivanjem generičkih argumenata svim generičkim parametrima, slika 9.1. Dobijena

konkretna klasa može imati posebno ime (naredbama poput *typedef* iz C^{++}) ili biti predstavljena nazivom generičke klase i stvarnim oblicima parametara, npr. *GenClass*<S_1, ..., S_n> gde su S_1, ..., S_n konkretni tipovi, tj. generički argumenti.



Slika 9.1

Druga osobina zahteva nešto više pažnje. Tvrdnju da se generička klasa ne može instancirati treba shvatiti uslovno, uz pretpostavku da instanciranje znači isključivo kreiranje *objekta* iz klase. Ako instanciranje shvatimo šire, kao kreiranje primerka klase ma šta on bio, lako zapažamo da se generička klasa itekako može instancirati, samo rezultat nije objekat nego *konkretna klasa*! Samo da bi se izbegli nesporazumi, umesto termina "instanciranje", kod generičke klase koristićemo upravo uvedeni naziv "konkretizacija". Dakle, konkretizacijom se dobija obična klasa koja se ponaša kao i svaka druga, posebno zato što se konkretizovani generički parametri više ne mogu menjati, tj. predstavljaju invarijantu u životnom veku konkretizovane generičke klase. To navodi na ideju da se generičke klase parametrizuju i konstantama koje po izvršenoj konkretizaciji takođe zadržavaju vrednost. Na primer, klasa *Niz* čiji su elementi tipa *T* a njihov (nepromenljiv) broj iznosi *B*, može se parametrizovati kao *Niz*<*T,B*>, naravno uz odgovarajuću sintaksu programskog jezika koja treba da obezbedi razlikovanje generičkog parametra koji je klasa-tip od generičkog parametra koji je po prirodi konstanta. Razmatranje konstanti-parametara ostavićemo za kasnije.

Odnos generičke i konkretne klase može se posmatrati iz dva ugla. Prvo, generička klasa može se shvatiti kao preslikavanje. Neka je Θ skup svih konkretnih tipova-klasa. Tada je konkretna klasa element skupa Θ, dok je generička klasa sa n parametara preslikavanje skupa Θ^n u skup Θ koje za konkretne vrednosti parametara daje za rezultat konkretnu klasu, kao što neka funkcija f(x,y) za konkretne vrednosti argumenata x i y generiše rezultat. Saobrazno tome, konkretna klasa

predstavlja rezultat primene preslikavanja $G<S_1,...,S_n>$ gde je G odgovarajuća generička klasa, a $S_1,...,S_n$ jesu konkretne klase-tipovi iz skupa Θ^n. Na primer, *List<integer>* (konkretna klasa) predstavlja primenu preslikavanja *List<T>* (generička klasa) u kojem je parametar *T* zamenjen stvarnim oblikom *integer*.[75]

Drugi pogled je shvatanje konkretizacije kao veze između generičke klase i konkretne klase izvedene iz nje. Neka je Γ skup generičkih klasa, a Θ kao i ranije skup konkretnih klasa. Tada se između skupova Γ i Θ uspostavlja relacija, tzv. ***veza konkretizacije***, koja povezuje generičke i konkretne klase. Veza se kvalifikuje stvarnim oblicima parametara $<S_1,...,S_n>$. Tako, konkretnu klasu *IntList* možemo povezati sa generičkom klasom *List<T>* vezom konkretizacije koja je kvalifikovana tipom *integer* što zamenjuje parametar *T*. Kao što ćemo videti, oba aspekta konkretizacije obuhvaćena su UML-om.

U generičkoj klasi $GenClass<T_1,...,T_n>$ generički parametri mogu se koristiti za definisanje tipova njenih podataka-članova, zatim klasa objekata-članova i konačno u prototipovima (zaglavljima) funkcija-članica. Otuda je logično pitanje kakve se *operacije* mogu izvoditi nad članovima čiji tip nije unapred poznat? Mejer [82] izdvaja nekoliko operacija koje su primenljive na sve tipove:

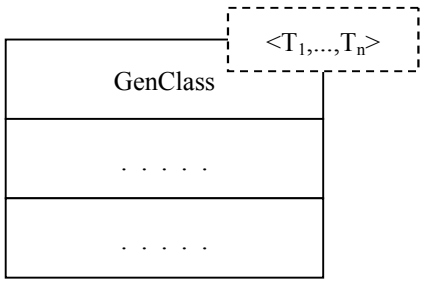
1. operacije pridruživanja (tj. dodele kao i zamene parametara argumentima)
2. operacije provere jednakosti odn. različitosti.
3. Mejerovim operacijama dodaćemo i operaciju *sizeof* primenljivu na klasu, tip i njihove instance, a koja računa njihovu veličinu, npr. u bajtovima.

Navedeni spisak ne iscrpljuje sve operacije što se mogu pojaviti u definiciji generičke klase. Prosto, to su operacije koje se uvek mogu konkretizovati (ili bi bar trebalo da je tako[76]). Neki programski jezici (npr. upravo C^{++}) dozvoljavaju da se u generičkoj klasi mogu da navedu proizvoljne operacije predviđene programskim jezikom, samo što nema garancije da će one biti uspešno povezane prilikom konkretizovanja klase. Neka su *a* i *b* dva podatka-člana generičke klase $G<T>$ i neka su oba tipa *T*. Ako se u nekoj od metoda klase *G* pojavi izraz oblika *a+b* konkretizacija će uspeti samo ako je operacija + definisana u tipu *T* - u suprotnom biće prosto prijavljena greška. Inače, ovu mogućnost većina objektnih jezika nema.

Generička klasa sa nazivom *GenClass* i generičkim parametrima $T_1,...,T_n$ u UML prikazuje se kao na slici 9.2.

[75] Inače, možemo otići i korak dalje dozvoljavajući da se generička klasa konkretizuje delimično, tako što neki tipovi-parametri dobiju konkretan oblik, a neki ne. U tom slučaju Θ treba shvatiti kao uniju skupova generičkih i konkretnih klasa.

[76] Ovde se ima u vidu model. U nekim konkretnim programskim jezicima nad nizom se ne može vršiti direktna dodela niti se dva niza mogu direktno porediti. To je, međutim, stvar realizacije.

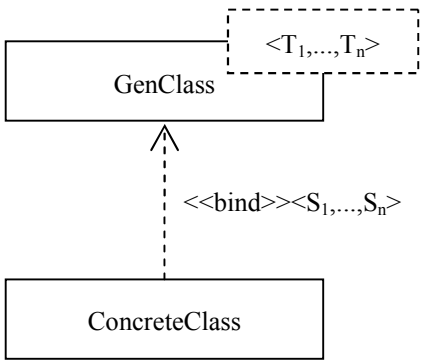


Slika 9.2

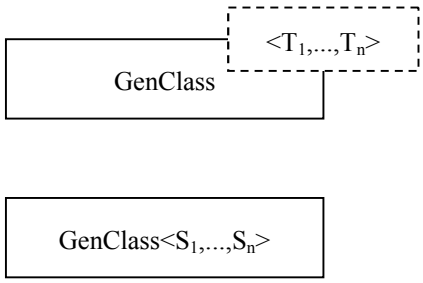
Kao što smo napomenuli, konkretizacija se može shvatiti kao operacija nad generičkom klasom ili pak kao veza između generičke i konkretne klase. UML predviđa posebnu simboliku za oba aspekta. Na slikama 9.3 i 9.4 prikazana su oba načina za prikaz konkretizacije.

Na slici 9.3 konkretizacija se tretira kao operacija nad generičkom klasom *GenClass*, koja se izvodi tako što se generički parametri $T_1,...,T_n$ zamenjuju generičkim argumentima $S_1,...,S_n$. Konkretizovana klasa obavezno nosi isti naziv kao i generička. Puno ime joj je *GenClass*$<S_1,...,S_n>$ jer se smatra rezultatom preslikavanja *GenClass* sa parametrima $S_1,...,S_n$.

Konkretizacija sa slike 9.4 shvata se kao veza između generičke klase *GenClass*$<T_1,...,T_n>$ i konkretne klase *ConcreteClass*. Veza je snabdevena stereotipom <<bind>> i stvarnim tipovima $<S_1,...,S_n>$ koji zamenjuju parametre $<T_1,...,T_n>$. Puno ime konkretne klase je *ConcreteClass*, jer se *GenClass* i *ConcreteClass* tretiraju kao posebne klase povezane vezom konkretizacije.



Slika 9.3



Slika 9.4

9.1.1. Generičke klase u C++

Sredstvo za realizaciju generičnosti, kako klasa tako i potprograma (tj. slobodnih funkcija), u C++ nosi naziv ***šablon*** ili ***templejt*** (engl. template). Izrazi "generička klasa" i "šablon klase" uglavnom se upotrebljavaju kao sinonimi, mada razlika postoji jer je pojam generičke klase opšti, dok šablon klase predstavlja sredstvo za izradu ovakvih klasa u programskom jeziku C++.

U sintaksnom smislu, šablon klase dobija se proširenjem naredbe *class*. Prvo, ispred službene reči *class* dodaje se prefiks sa opštim oblikom

template <class T_1, ..., class T_n, tip_1 K_1, ..., tip_m K_m>

gde su T_1, ..., T_n oznake klasa i-ili tipova generičkih parametara, a K_1, ..., K_m oznake konstanata-parametara čiji su tipovi respektivno tip_1, ..., tip_m. Redosled navođenja parametara nije obavezno navedeni; mogu se prvo napisati konstante pa tipovi, a mogu se i izmešati. Konstante-parametri, naravno, ne moraju postojati. Rezervisana reč *class* upotrebljava se i za klase i za tipove parametre[77]. S obzirom na slobodan format programa, fraza *template* najčešće se ne piše ispred nego iznad *class*.

Neka je *GenClass* generička klasa parametrizovana tipovima-klasama *G* i *H* i celobrojnom konstantom *C*. Tada odgovarajući šablon klase ima oblik

```
template <class G, class H, int C>
class GenClass {
. . . . . .
// G, H i C upotrebljavaju se kao da su poznati!
. . . . . .
```

[77] Inače, novi standard C++ dozvoljava da se uporedo sa rečju *class* upotrebljava i reč *typename* koja je sinonim, tako da se klase-parametri mogu označiti sa *class*, a tipovi-parametri sa *typename*.

```
};
```

Metode koje su fizički izvan naredbe *class* takođe se modifikuju pomoću šablona, dodavanjem istog prefiksa *template*, ali i navođenjem parametara u okviru kvalifikatora metode. Neka klasa *GenClass* ima metodu *met* tipa npr. *void*[78] koja nije *inline*. Tada njena definicija izgleda ovako:

```
template <class G, class H, int C>
void GenClass<G,H,C>::met(...) {
. . . . . .
}
```

Uopšte, samo ime generičke klase nije identifikator nego samo njegov deo, jer je isto za generičku klasu ali i za sve konkretne klase stvorene iz nje. Pun naziv uvek sadrži pored imena i parametre ako se radi o generičkoj klasi ili argumente kada je u pitanju neka od konkretnih klasa. Prema tome, ne *GenClass*, ali da *GenClass<G,H,C>* za generičku ili *GenClass<double,MyClass,1000>* za konkretnu klasu.

Generičke klase - sa izuzetkom postojanja parametara - ne razlikuju se od drugih klasa: imaju konstruktore i destruktor, mogu nasleđivati i biti nasleđivane, mogu biti apstraktne, imati preklopljene operatore itd. Suštinska razlika između generičke i konkretne klase, kako je napomenuto u opštem delu, leži u činjenici da se generička klasa ne može instancirati. Ona se mora konkretizovati posebnom naredbom *typedef* ili u okviru kreiranja objekta. Dajemo nekoliko primera kreiranja objekata iz generičke klase:

```
GenClass<double,ConcreteClass,500> x;      //Instanciranje sa konkretizacijom
GenClass<int,int,2000> y,z;                //Takođe
typedef GenClass<char,char,100> MyClass;  //Konkretizacija bez instanciranja
MyClass a, b;                              //Instanciranje konkretizovane klase
GenClass t;                                //NE! GenClass nije identifikator
GenClass<G,H,C> t;                         //NE! Klasa se mora konkretizovati
```

Valja napomenuti da sve navedene naredbe osim poslednje dve, po potrebi mogu egzistirati u istom programu. Posebno, instanciranje klase i ovde je poziv (nekog od) konstruktora. Konačno - a kako se vidi iz primera - klasa se može instancirati naporedo sa konkretizacijom, ali i konkretizovati naredbom *typedef*, a zatim nezavisno instancirati koristeći novouvedenu konkretnu klasu (*MyClass* u primeru).

Prilikom prevođenja, generička klasa mora da se prevodi zajedno sa klijen-

[78] Podvucimo da i tip same metode može da bude parametrizovan.

tom jer se tek tada može konkretizovati, odnosno tada se može izvršiti tzv. razrešenje imena. Stoga se generička klasa smešta u celosti u zaglavlje biblioteke u kojem treba da se nađe[79]. Pored toga, mora se voditi računa i o tome da neki članovi generičke klase ne zavise od generičkih parametara (ne koriste ih), tako da je redosled razrešavanja

1. razrešiti imena članova koji ne zavise od generičkih parametara
2. razrešiti imena onih članova koji zavise od generičkih parametara.

Kao prvi primer, izgradićemo generičku klasu *Vektor<E,B>* što predstavlja vektor sa elementima tipa *E* čiji je maksimalan broj dat konstantom-parametrom *B*. Pored konstruktora i destruktora klasa ima sledeće članove:

- aktuelni broj elemenata *n*
- niz *E el[B]* dužine *B* sa elementima tipa *E*, što odgovaraju elementima objekta-vektora
- funkcije *zadatiBrojEl* i *brojEl* za zadavanje i očitavanje aktuelnog broja elemenata
- dvostruko preklopljen operator indeksiranja []; operator je dvostruko preklopljen jer može da bude kako konstantan izraz tako i *lvrednost*
- preklopljeni operator dodele
- preklopljeni operatori provere jednakosti i nejednakosti dva vektora.

Prilikom realizacije klase, parametri *E* i *B* koriste se poput svih drugih tipova i konstanata. Štaviše, nad članovima tipa *E* mogu se izvršavati sve operacije, bez obzira na to što se ne zna da li će ih konkretizovani tip sadržati ili neće[80]. Ovo je moguće zahvaljujući načinu na koji razvojni sistem funkcioniše. Naime, sistem prvo uključuje tekstuelnu datoteku sa izvornim kodom šablona (npr. VEKTOR.HPP) u tekst klijenta, pa tek onda prelazi na prevođenje. Već smo zaključili da kompletan izvorni kod generičke klase mora biti sadržan u zaglavlju (nema tela modula).
Tekst modula (zaglavlja) VEKTOR.HPP sa generičkom klasom *Vektor* ima sledeći izgled:

```
/*****************************************************************
                    MODUL SA KLASOM GENERICKOG VEKTORA
Naziv datoteke: VEKTOR.HPP
*****************************************************************/
#ifndef VEKTOR_HPP
  #define VEKTOR_HPP

template <class E, int B>
```

[79] U novim verzijama C++ postoje i druge mogućnosti (videti [88]).
[80] Ako se ispostavi da ih ne sadrži, u pitanju je najobičnija greška koja se lako detektuje još pri kompilaciji.

```
class Vektor {
private:
  int n;
  E el[B];
public:
  Vektor() {n= 0;}
  ~Vektor() {n= 0;}
  void zadatiBrojEl(int br) {n= br;}
  int brojEl() const {return n;}
  E operator [](int i) const {return el[i];}
  E& operator [](int i) {return el[i];}
  Vektor<E,B>& operator =(const Vektor<E,B>);
  int operator ==(const Vektor<E,B>&);
  int operator !=(const Vektor<E,B>&);
};
template <class E,int B>
Vektor<E,B>& Vektor<E,B>::operator =(const Vektor<E,B> rhs) {
  for(n=0;n<rhs.BrojEl();n++) el[n]= rhs[n];
  return *this;
}
template <class E,int B>
int Vektor<E,B>::operator ==(const Vektor<E,B> &rhs) {
 if(n!=rhs.n) return 0;
 for(int i=0;i<n;i++) if(el[i]!=rhs.el[i]) return 0;
 return 1;
}
template <class E,int B>
int Vektor<E,B>::operator !=(const Vektor<E,B> &rhs) {
 if(n!=rhs.n) return 1;
 for(int i=0;i<n;i++) if(el[i]!=rhs.el[i]) return 1;
 return 0;
}

#endif
```

U principu, klasu *Vektor* možemo koristiti na dva neznatno različita načina: instanciranjem sa konkretizacijom ili prethodnom konkretizacijom.

U prvom slučaju, objekat konkretizovane klase *Vektor* neposredno se instancira preko imena generičke klase i konkretizovanih parametara. Na primer,

```
Vektor<int,100> iVek;
```

kreiraće vektor *iVek* sa celobrojnim elementima i dužinom 100. Isti krajnji efekat postižemo prethodnom konkretizacijom:

```
typedef Vektor<int,100> IntVektor;
```

i posebnim instanciranjem konkretne klase *IntVektor*

```
IntVektor iVek;
```

Drugi način je nešto fleksibilniji jer jednom uvedena, klasa *IntVektor* koristi se kao i svaka druga konkretna klasa.

9.2. GENERIČKI I ADAPTIVNI POTPROGRAMI

Prema osnovnoj definiciji (Strachey), potprogrami koji su parametarski polimorfni funkcionišu nad širim domenom i kodomenom tipova. Drukčije rečeno, neki potprogram $p(a_1:T_1,...,a_n:T_n):T_0$ izvršava se za više konkretnih oblika $T_0, ..., T_n$. S obzirom na način konkretizovanja tipova $T_0, ...,T_n$ razlikujemo dve vrste polimorfnih potprograma: generičke i adaptivne. ***Generički potprogrami*** pre upotrebe ili pri samom instanciranju nalažu eksplicitno zadavanje konkretnih oblika svim generičkim parametrima. ***Adaptivni potprogrami*** nešto su drukčiji. Takav potprogram bez posebne konkretizacije prepoznaje *tipove* argumenata i u skladu s tim podešava ponašanje, čak do nivoa trase algoritma. Adaptivni potprogram u paskalu je recimo funkcija-prethodnik *pred* koja za celobrojni argument daje kao rezultat za 1 manji ceo broj, za znakovni daje prethodni znak, a za enumeraciju prethodnu konstantu u kolacionom nizu.

Generički i adaptivni potprogrami jesu slični, ali se ne podudaraju po ponašanju. U C^{++} realizuju se istim sredstvom (šablonom), ali se različito koriste. Paskal, na primer, uopšte nema generičkih potprograma, dok poseduje čak prilično moćna sredstva za realizaciju adaptivnih.

9.2.1. Generičke i adaptivne funkcije u C^{++}

Obe vrste polimorfnih funkcija - i generičke i adaptivne - realizuju se istim mehanizmom: šablonom. Opšti oblik definicije polimorfne funkcije (generičke i adaptivne) jeste

```
template <class T1,...,class Tn, tip1 K1, ..., tipm Km>
t0 imeFunkcije(t1 p1, ..., tk pk) {
. . . . . .
```

}

gde na mestu nekih ili svih parametara t_1, ..., t_k stoje parametri T_1, ..., T_n. Pritom, važi pravilo da se svaki od generičkih parametara T_1, ..., T_n mora naći bar jedanput u skupu $\{t_1, ..., t_k\}$, a da među parametrima funkcije može biti i negeneričkih (računajući i tip same funkcije). Generičke konstante $K_1,...,K_m$ imaju iste osobine kao i kod generičkih klasa. Ukoliko želimo, novije verzije C^{++} i kod funkcija omogućavaju upotrebu sinonima *typename* umesto *class*.

Do razilaženja između generičkih i adaptivnih funkcija dolazi prilikom korišćenja. Generičke funkcije moraju se prethodno konkretizovati navođenjem *prototipa* u kojem su svi generički parametri zamenjeni konkretnim oblicima, tj. generičkim argumentima. Neka je *max* funkcija za određivanje veće od dve vrednosti, *a* i *b*, obe tipa *G*. Šablon funkcije ima sledeći izgled:

```
template <class G>
G max(G a, G b) {
  return (a>b) ?  a  :  b;
}
```

Šablon *max* se postavlja u ulogu generičke funkcije na dva načina, pri čemu - a nažalost - nema garancije da će oba načina biti moguća kod svakog prevodioca.

Prvi način je da se funkcija se u klijentu konkretizuje navođenjem *prototipa* u kojem se generički parametar *G* zamenjuje generičkim argumentom, na primer,

```
double max(double, double);
```

Posle ove naredbe *max* se ponaša kao *double* funkcija sa dva *double* parametra, ni po čemu različita od drugih funkcija, pa tako i podložna automatskoj konverziji tipova. Inače, ovaj postupak konkretizacije generičke funkcije zove se i generisanje funkcije iz šablona.

Drugi način jeste eksplicitna konkretizacija svih generičkih parametara koja se izvodi tako što se *pri pozivu* iza imena funkcije navedu generički argumenti umetnuti između simbola < i >. Za naš primer funkcije *max*, to bi mogao biti izraz oblika

```
x = max<double>(a,b)
```

gde <*double*> iza imena funkcije znači da će generički parametar *G* iz šablona, na svim mestima biti zamenjen generičkim argumentom *double*. Tip argumenata *a* i *b* ovde nije od značaja jer će biti konvertovan u *double*. Drugim rečima, pozivi

funkcije *max* oblika

max<double>(1,2) i max<double>(1.0,2.0)

rezultovaće vrednošću 1.0 tipa *double*. Rezultat tipa *double* ispostaviće i poziv

max<double>('u','v')

a sve zbog eksplicitnog oblika generičkog argumenta s jedne i konverzije tipova (*int* odn. *char* u *double*), s druge strane.

Adaptivne funkcije razlikuju se od generičkih po tome što se ne konkretizuju nego koriste direktno (uz rizik greške u tipu). Istu funkciju *max* možemo koristiti bez konkretizacije:

```
int i,j,k; double x,y,z;
. . . . . .
k= max(i,j);     //max automatski generiše izlaz tipa int
z= max(x,y);   //max automatski generiše izlaz tipa double
```

Funkcije prethodnik i sledbenik (*pred* i *succ* iz paskala) mogu se realizovati u C++ kao adaptivne funkcije:

```
template <class T>
T pred(T x) {return x-1;}

template <class T>
T succ(T x) {return x+1;}
```

Tako, *pred('B')* vratiće vrednost *'A'*, *succ(5)* biće jednako *6*, *succ(3.5)* daće *4.5* itd. Funkciju *abs* koju smo u glavi 6 realizovali preklapanjem pet funkcija, jednostavnije realizujemo kao jednu adaptivnu funkciju:

```
template <typename G>
G Abs(G arg) {
  return (arg>=0) ? arg : -arg;
}
```

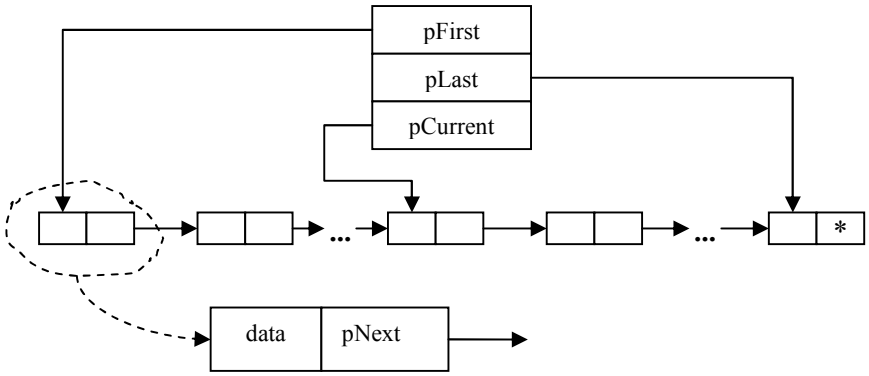
gde smo umesto *class* upotrebili *typename* samo radi ilustracije. Ma kakav bio parametar *arg*, funkcija vraća njegovu apsolutnu vrednost istog tipa. Naravno, može se desiti da funkciju aktiviramo sa argumentom na koji nije primenljiva operacija upoređenja >= ili operacija promene predznaka. U takvim slučajevima radi se, na-

ravno, o grešci koja će biti prijavljena, tako da to ne predstavlja naročit problem.

Vidimo da se adaptivne funkcije u C^{++} odlikuju mogućnošću podešavanja tipa izlaza prema nekom od tipova argumenata. Drugih mogućnosti za uticanje na ponašanje funkcije tipom stvarnog argumenta (npr. za izbor trase izvršenja) u većini aktuelnih verzija C^{++} nema[81].

9.3. PRIMER: GENERIČKA LISTA SA JEDNIM ITERATOROM

Generičke klase najviše se upotrebljavaju za izradu tzv. ***kontejnerskih klasa*** koje nisu ništa drugo do strukture podataka: niz, stek, red, razne vrste stabala itd. Tipična kontejnerska klasa jeste jednostruko spregnuta lista sa elementima proizvoljnog tipa, koju ćemo razmotriti u ovom primeru. Lista je prikazana na slici 9.6.



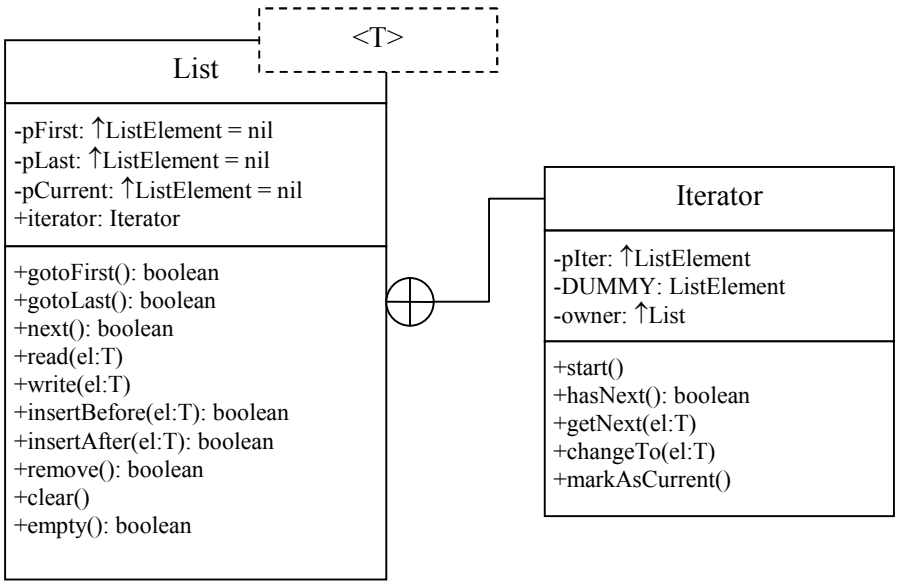
Slika 9.6

Već po slici lako zaključujemo da je ovaj oblik liste formalno gotovo identičan sa primerom liste iz odeljka 5.3, gde smo razmatrali jednostruko spregnutu listu čiji su elementi *void* pokazivači. Operacije će biti gotovo identične, ali između ova dva primera postoji bitna razlika: elementi liste iz odeljka 5.3. imaju konkretan sadržaj - *void* pokazivače. Generička lista koju ćemo realizovati bitno se razlikuje od prethodne po tome što su njeni elementi proizvoljnog tipa *T* koji je generički parametar te, shodno tome, lista ima daleko širu oblast primene. Listu ćemo realizovati tako da sadržaj elementa može biti bilo šta: podatak baznog tipa, niz (uključujući i string), slog, proizvoljni izvedeni tip napravljen pomoću *typedef* ili objekat.

Elementi liste sastoje se od dva polja: prvo je polje *data* (parametrizovanog) tipa *T* koje predstavlja sadržaj elementa, a drugo je pokazivač *pNext* na sledeći element u listi. Objekat klase *List* je, u stvari, zaglavlje (deskriptor) liste sa tri pokazivača na element liste: *pFirst* (pokazivač na prvi), *pLast* (pokazivač na

[81] U novim verzijama C^{++} postoji i takva mogućnost, korišćenjem novouvedene funkcije *typeid*, videti [88].

poslednji) i *pCurrent* (pokazivač na tekući element). Na slici 9.7 prikazan je simbol klase *List*. Objekat-član *iter* deo je mehanizma iteratora koji ćemo, ilustracije radi, uvesti i koji će biti opisan na posebnom mestu.



Slika 9.7

Dajemo prvo kratku specifikaciju metoda. Metode nose ista imena kao i kod liste pokazivača i praktično se ne razlikuju od metoda liste pokazivača: razlika nastaje na mestima gde je tip elementa *void* * zamenjen generičkim parametrom *T*. Iz posebnih razloga koji će biti navedeni uz opis, metode za očitavanje sadržaja elementa *read* i za uklanjanje *remove* morale su da promene prototip. Metode klase *List* (bez iteratora) jesu sledeće:

- Konstruktor *List()* postavlja sve atribute na 0 (NULL) i konstruiše objekat-član *iter* koji omogućuje rad iteratoru; konstrukcija *iter* zahteva slanje adrese objekta vlasnika.
- Konstruktor kopije *List(List&)* pravi duboku kopiju; takođe, konstruiše objekatćčlan *iter* na isti način kao i osnovni konstruktor.
- Destruktor *~List()* uništava listu.
- *int gotoFirst()* postavlja tekući na prvi i vraća vrednost 1 ako je operacija uspešna. Ako je lista prazna vraća vrednost 0.
- *int gotoLast()* radi analogan posao za poslednji element liste.
- *int next()* postavlja tekući na sledeći. Ako je tekući element jednak NULL vraća vrednost 0, a ako nije vraća vrednost 1. Kada je *pCurrent==pLast* ova metoda može da se primeni još jednom i tada postaje *pCurrent* ==

NULL.

- *int read(T &el)* preko parametra *el* vraća sadržaj polja *data* tekućeg elementa. Ostavlja tekući nepromenjenim. Vraća vrednost 1 ako je operacija uspešna, inače 0. Zapazimo da je metoda izmenjena u odnosu na istu u listi pokazivača iz odeljka 5.3 jer ova poslednja vraća sadržaj elementa kao rezultat. Do izmene je došlo zbog toga što je verzija koju razmatramo generička i treba da omogući da sadržaj elemenata budu i nizovi odn. stringovi koji se ne mogu vraćati kao rezultat. Konačno, formiranje izlazne vrednosti *el* obavlja se pomoću standardne funkcije *memcpy* koja, bez ikakve provere, kopira element liste u izlazni argument *el* na principu bajt-po-bajt.
- *int write(const T &el)*: ako tekući nije NULL upisuje *el* u njegovo polje *data* i vraća vrednost 1. Ako je *pCurrent* prazan vraća vrednost 0. Metoda *write* upisuje novi sadržaj u element liste istim mehanizmom koji koristi i *read*, dakle kopiranjem bajt po bajt putem funkcije *memcpy*.
- *int insertBefore(const T &el)*: dodaje novi element čije je polje *data* jednako *el* ispred tekućeg. Ako je lista prazna upisuje novi element kao prvi. Ako je tekući jednak NULL ne izvršava se i vraća vrednost 0. Ako tekući nije bio prazan posle dodavanja postavlja vrednost *pCurrent* i vraća vrednost 1.
- *int insertAfter(const T &el)*: sve je isto, samo dodaje iza tekućeg.
- *int remove()* uklanja tekući. Ako je lista prazna ili je tekući NULL ne izvršava se i vraća vrednost 0. Novi tekući element je onaj *ispred* uklonjenog tekućeg, ali ako je uklonjen prvi po redu novi tekući postaje njegov sledbenik (tj. bivši drugi). Vraća 1 ako je operacija uspešna. Za razliku od istoimene metode kod liste pokazivača, ova metoda ne vraća sadržaj obrisanog elementa kao rezultat, jer taj sadržaj može biti i niz odn. string.
- *void clear()* briše (prazni) listu.
- *int empty()* vraća 1 ako je lista prazna, odnosno 0 ako nije.

Napomene:

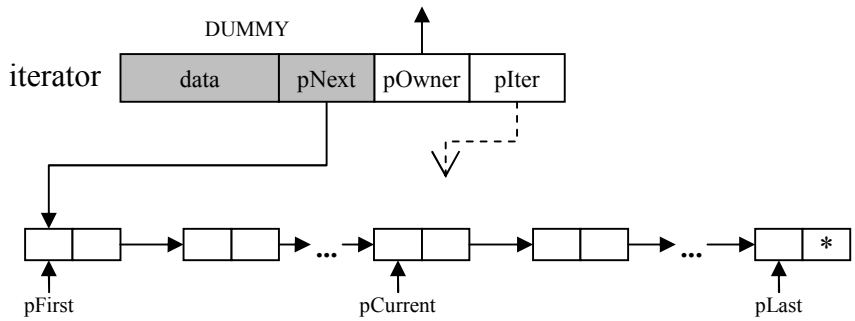
1. Kompletna generička klasa mora biti smeštena u zaglavlje da bi je prevodilac imao na raspolaganju prilikom konkretizacije.
2. Još jednom: za kopiranje sadržaja elementa umesto dodele koristi se standardna funkcija *memcpy* iz biblioteke <mem.h> da bi i nizovi (i stringovi!) mogli da predstavljaju informacioni sadržaj elementa.

Iterator

Vežbe radi, ali i zato što to ima praktičnog smisla, klasu *List* snabdećemo i iteratorom. Iterator je operacija uvedena u poglavlju 5 kao operacija za sistematski pristup svakom elementu kontejnerske klase. Iterator je po pravilu složena oper-

acija koja se realizuje kao posebna klasa sa više metoda kojima se otpočinje iteriranje, prelazi na sledeći element i proverava kraj. Pored ove tri osnovne, u iterator se mogu ugraditi i metode za obradu elemenata u toku pristupa: npr. čitanje ili izmenu njihovog sadržaja. Čitanje sadržaja obično se kombinuje sa prelaskom na sledeći element. Iterator ćemo realizovati kao *uklopljenu klasu* klase *List* (videti odeljak 4.3.1 u glavi 4), jer je iterator u našem primeru deo liste. U složenijim objektnim hijerarhijama, iteratori se realizuju i kao javne klase jer se mogu primeniti na više različitih kontejnerskih klasa, kao što su dinamički niz, lista ili binarno stablo.

Unutrašnja klasa *Iterator* ima tri podatka člana: jedan je tipa *ListElement* sa nazivom DUMMY koji igra ulogu praznog početnog elementa za prolaz kroz listu. Polje *data* ovog elementa nema definisanu vrednost, a polje *pNext* pokazuje na prvi element liste. Pokazivač *pIter* u toku iteriranja pokazuje na element koiem se pristupa. Treće polje jeste polje *owner* koje sadrži adresu vlasnika (koji je objekat klase *List*) da bi se preko njega omogućio pristup elementima liste. Rad iteratora koncipiran je tako da je proces iteriranja odvojen od ostalih operacija nad listom i ne menja njenu strukturu. To konkretno znači da se tekući element liste definisan pokazivačem *pCurrent* u toku iteriranja ne menja. Ako u toku iteriranja ne upotrebimo posebnu metodu *changeTo* za izmenu sadržaja elementa, lista će po završenom prolazu kroz listu biti u identičnom stanju kao i pre nje. Šema dela liste koji se odnosi na iterator prikazana je na slici 9.8.



Slika 9.8

Metode unutrašnje klase *Iterator* nisu komplikovane. U osnovi, da bi se izvelo iteriranje treba obezbediti samo samo tri metode: metodu koja postavlja iterator u početno stanje, metoda za proveru kraja celog procesa i metoda za otpočinjanje sledeće po redu iteracije. Ove tri metode predstavljaju minimum, a mi ćemo u klasu uključiti još neke, specifične:

- metoda *void start()* kojom se startuje novi proces za iteraciju i to tako što se polje *pNext* polja DUMMY postavlja na vrednost *pFirst* objekta klase *List* koji je vlasnik iteratora; takođe pokazivač *pIter* preko kojeg se vrši

iteriranje, postavlja se na polje DUMMY, da bi prelazak na sledeći element bio, u stvari, pristup prvom elementu liste;

- logička metoda *int hasNext()* koja vraća vrednost 1 ako iza elementa koji je trenutno adresiran sa *pIter* postoji sledeći element u listi, a u suprotnom 0; provera se vrši na osnovu vrednosti *pNext* poslednjeg elementa kojem je pristupljeno: ako je pokazivač *pNext* tog elementa NULL znači da *pIter* pokazuje na poslednji element liste i da je iteriranje završeno;
- metoda *void getNext(T& el)* kojom se prelazi na sledeći element liste i očitava vrednost njegovog polja *data*; metoda funkcioniše na isti način kao i metoda *read* liste, tj. ne vraća vrednost kao rezultat nego preko izlaznog parametra *el* da bi se omogućilo da sadržaj elementa liste bude niz ili string;
- metoda *void changeTo(T& el)* kojom se, u toku iteracije, može promeniti sadržaj elementa liste na koji trenutno pokazuje pokazivač *pIter*; iako je to izvodljivo i metodom *write* liste, iterator je koncipiran tako da se u toku iteracije ne koriste originalne metode liste;
- metoda *void markAsCurrent()* kojom se element na koji trenutno pokazuje pokazivač *pIter* proglašava za tekući element liste jer, ne zaboravimo, iteriranje kod liste ne menja tekući element; po završetku iteriranja, ako je upotrebljena metoda *markAsCurrent*, tekući element liste se promenio na markirani, a ako *markAsCurrent* nije upotrebljena tekući element je isti onaj koji je bio i pre iteriranja.

Iteratoru objekta klase *List* pristupa se preko objekta-člana *iterator* koji je otvoren. Alternativno rešenje bilo bi zatvoriti ga uz uvodjenje posebne metode za pristup toim objektu-članu, međutim to bi samo nepotrebno produžilo proces iteriranja jer bi takva metoda morala biti pozivana svaki put kada se izvodi neka iteratorska operacija. Polje *iterator* povezuje se sa samom listom preko svog dela *pOwner* koje sadrži adresu objekta-liste koja je vlasnik iteratora. Veza se uspostavlja pri konstrukciji objekta liste gde se adresa liste putem polja *this* šalje konstruktoru unutrašnje klase *Iterator*.

Klasa *List* smeštena je u modul sa nazivom GENLIST.HPP. Prema napomeni 1, klasa *List* mora se u celosti nalaziti u zaglavlju, tako da telo modula *List* ne postoji. Modul ima sledeći izgled:

```
/*****************************************************************

          MODUL SA KLASOM GENERICKE LISTE

Naziv datoteke: GENLIST.HPP
*****************************************************************/
#ifndef GENLIST_HPP
```

```
 #define GENLIST_HPP

#include <mem.h>

template <class T>
class List {
private:
  struct ListElement {
    T data;
    ListElement *pNext;
  };
// Uklopljena klasa iteratora
  class Iterator {
  private:
    ListElement *pIter,DUMMY;
    List *pOwner;
  public:
    Iterator(List *owner) {
      pOwner= owner;
    }
    void start() {
      DUMMY.pNext= pOwner->pFirst;
      pIter= &DUMMY;
    }
    int hasNext() {
      return (pIter->pNext!=0);
    }
    void getNext(T& el) {
      pIter= pIter->pNext;
      memcpy(&el,&(pIter->data),sizeof(T));
    }
    void changeTo(T& el) {
      memcpy(&(pIter->data),&el,sizeof(T));
    }
    void markAsCurrent() {
      pOwner->pCurrent= pIter;
    }
  };
  ListElement *pFirst, *pLast, *pCurrent;
public:
  Iterator iterator;
```

```
  List(): iter(this) {pFirst= pLast= pCurrent= 0;}
  ~List();
  List(const List&);
  int gotoFirst();
  int gotoLast();
  int next();
  int read(T&);
  int write(const T&);
  int insertBefore(const T&);
  int insertAfter(const T&);
  int remove();
  void clear();
  int empty() const {return pFirst==0;}
 };

template <class T>
List<T>::~List() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pNext; delete temp;}
  pLast= pCurrent= 0;
}

template <class T>
List<T>::List(const List<T> &rList): iter(this) {
  ListElement *orig;
  if(rList.empty()) {pFirst= pLast= pCurrent= 0;}   // Prazna lista
  else {
    orig= rList.pFirst;                          // Postaviti prvi
    pCurrent= pFirst= new ListElement;
    memcpy(&(pCurrent->data),&(orig->data),sizeof(T));
    while(orig= orig->pNext) {
      pCurrent= pCurrent->pNext= new ListElement;
      memcpy(&(pCurrent->data),&(orig->data),sizeof(T)); }
    (pLast=pCurrent)->pNext= 0;
    pCurrent= rList.pCurrent;
  }
}

template <class T>
int List<T>::gotoFirst() {
  if(pFirst) {pCurrent= pFirst; return 1;}
```

```
  else return 0;
}

template <class T>
int List<T>::gotoLast() {
  if(pLast) {pCurrent= pLast; return 1;}
  else return 0;
}

template <class T>
int List<T>::next() {
  if(pCurrent) pCurrent= pCurrent->pNext;
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>::read(T &el) {
  if(pCurrent) {
    memcpy(&el,&(pCurrent->data),sizeof(T));
    return 1;}
  else return 0;
}

template <class T>
int List<T>::write(const T &el) {
  if(pCurrent) {
    memcpy(&(pCurrent->data),&el,sizeof(T));
    return 1;}
  else return 0;
}

template <class T>
int List<T>::insertBefore(const T &el) {
  ListElement *newEl, *prev, *next;
  newEl= new ListElement;
  memcpy(&(newEl->data),&el,sizeof(T));
  if(empty()) {newEl->pNext= 0;          // Lista prazna
               pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else if(pCurrent == pFirst) {          // Dodavanje ispred prvog
    newEl->pNext= pFirst;
```

```
   pCurrent= pFirst= newEl; }
  else {                          // Ostalo
   next= pFirst;
   while(next != pCurrent) {prev= next; next= next->pNext;}
   newEl->pNext= next;
   pCurrent= prev->pNext= newEl; }
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>::insertAfter(const T &el) {
  ListElement *newEl;
  newEl= new ListElement;
  memcpy(&(newEl->data),&el,sizeof(T));
  if(empty()) {newEl->pNext= 0;          // Lista prazna
              pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else {                          // Ostalo
   newEl->pNext= pCurrent->pNext;
   pCurrent->pNext= newEl;
   if(pCurrent == pLast) pLast= newEl;   // Dodat iza poslednjeg
   pCurrent= newEl; }
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>:: remove() {
  ListElement *prev, *next;
  if (empty()) return 0;              // Lista je prazna
  else if(pCurrent == 0) return 0;      // Tekuci prazan
  else {
   prev= next= pFirst;
   while(next != pCurrent) {prev= next; next= next->pNext;}
   if(next != pFirst) {               // Nije prvi
     prev->pNext= next->pNext;
     pCurrent= prev;
     if(next == pLast) pLast= prev; }  // Poslednji
   else {                          // Jeste prvi
     pCurrent= pFirst= pFirst->pNext;
     if(next == pLast) pLast= 0; }     // Bio jedini
   delete next;
```

```
   return 1;
  }
}

template <class T>
void List<T>::clear() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pNext; delete temp;}
  pLast= pCurrent= 0;
}

#endif
```

Primer primene. Generička lista može se konkretizovati bilo kojim tipom elemenata, uključujući i nizove (zato se kopiranje izvodi funkcijom *memcpy*). Kao ilustraciju primene klase kreiraćemo dve liste, jednu sa elementima tipa sloga, a drugu sa elementima tipa stringa (tj. niza):

```
#include "genlist.hpp"
typedef struct {int p1; char p2[20];} RecType;
typedef char StrType[50];
typedef List<RecType> RecList;     //Konkretizacija liste u listu slogova
typedef List<StrType> StrList;     //Konkretizacija liste u listu stringova
RecList rlst1;     //Konstruisanje liste slogova rlst1
StrList slst1;     //Konstruisanje liste stringova slst1
List<RecType> rlst2;     //Direktno kreiranje liste slogova rlst2
List<StrType> slst2;     //Direktno kreiranje liste stringova slst2
```

Lista se može učiniti i polimorfnom strukturom u smislu da može da primi elemente međusobno različite po tipu. Potrebno i dovoljno je elemente realizovati kao generičke pokazivače:

```
typedef List<void *> PolymorficList;
```

U praksi, češći je slučaj da elementi liste ne budu potpuno proizvoljnog tipa, već tipa pokazivača na korenu klasu hijerarhije klasa. Primer je lista geometrijskih figura u koju se mogu upisivati i obrađivati virtuelnim metodama objekti svih klasa iz hijerarhije. Takva lista imala bi konkretizaciju:

```
typedef List<Figura *> ListaFigura;
```

Prikazaćemo i nekoliko primera iteriranja kroz listu. Opšta struktura iterativnog procesa za neku listu *lst* ima sledeći oblik:

```
lst.iterator.start();   //startovanje iteratora
while(lst.iterator.hasNext()) {  //provera ima li jos elemenata
  lst.iterator.getNext(el);   //citanje sledeceg elementa
  // obraditi element el
}
```

Neka, na primer, lista *intList* sadrži kao elemente celobrojne konstante. Lista se definiše sa

```
List<int> intList;
```

Sledeća iteracija odrediće zbir *sum* vrednosti svih elemenata liste:

```
int sum=0, i;
intList.iterator.start();
while(intList.iterator.hasNext()) {
  intList.iterator.getNext(i);
  sum+= i;
}
```

Iteracija koja sledi povećaće vrednost svakog elementa za 1:

```
intList.iterator.start();
while(intList.iterator.hasNext()) {
  intList.iterator.getNext(i);
  intList.iterator.changeTo(++i);
}
```

Neka je potrebno u listi pronaći prvi po redu element sa vrednošću 100 i učiniti ga tekućim elementom liste, a ako ga nema ostaviti tekući element nepromenjenim. To se postiže sledećim segmentom:

```
intList.iterator.start();
while(intList.iterator.hasNext()) {
  intList.iterator.getNext(i);
  if(i==100) {intList.iterator.markAsCurrent(); break;}
}
intList.insertAfter(200);
```

Ako u listi postoji element sa sadržajem 100, poslednjom naredbom u sekvenci element 200 biće dodat iza njega, jer je u tom trenutku taj element (ili prvi od takvih ako ih ima više) tekući. Ako takvog elementa nema, 200 će biti ubačen iza elementa koji je bio tekući i pre izvođenja iteracije.

U pogledu korišćenja iteratora postoje i neka, sasvim logična, ograničenja. Naime, unutar segmenta za iteriranje sama lista jeste dostupna i nad njom se mogu izvoditi sve operacije. Međutim, uklanjanje elementa liste operacijom *remove* ili pražnjenje liste operacijom *clear* izazvali bi katastrofalne posledice jer bi se moglo desiti da se obriše baš element koji je trenutno u fazi obrade u iteratoru ili, još gore, da se u fazi iteriranja obriše cela lista. Dodavanje elemenata nije kritično, ali može da rezultuje greškom: ako se, recimo, tokom izračunavanja sume svih elemenata *intList* ubaci novi element u deo liste koji je iterator već prošao, on neće biti uračunat u zbir. Zbog toga, treba se držati propisa da se unutar iteriranja koriste samo metode iteratora, što je i bio razlog za uključivanje metode *changeTo* kojom se, umesto *write*, menja vrednost aktivnog elementa liste.

Na kraju, dajemo u vidu nekakvog test-programa jedan potpuni primer korišćenja liste gde je uključen i iterator. Lista je definisana tako da za sadržaj elemenata ima znakove, tj. tip *char*. Zapazimo da je, pored posebnih segmenata za testiranje iteratora, i funkcija *show* za prikazivanje sadržaja liste takođe realizovana korišćenjem iteratora.

```
/*****************************************************************

    DEMONSTRACIONI PROGRAM ZA KLASU "GENLIST" (genericka lista)

                a) element liste je znak
                b) prikazuje se i iterator!


Naziv datoteke: LISTTST.CPP
*****************************************************************/
#include <iostream.h>
#include "genlist.hpp"

//Funkcija za prikaz sadrzaja liste
template <class T>
void show(List<T> list)
{
  T first,last,current,datum;
  if(list.empty()) cout << "List is empty." << endl;
```

```
  else {
    list.read(current);
    cout << "List contents: ";
    list.iterator.start();
    while(list.iterator.hasNext()){
      list.iterator.getNext(datum);
      cout << "  " << datum;
    }
    list.gotoFirst(); list.read(first);
    list.gotoLast(); list.read(last);
    cout << "\nFirst: " << first
         << "  Last: " << last
         << "  Current: " << current
         << endl;
  }
}

int main() {
  List<char> lst;   //konkretizacija liste znakova iz generic ke liste

  cout << "Lista je konstruisana." << endl;
  show(lst);

  cout << "\nUpisan element a" << endl;
  lst.insertAfter('a');
  show(lst);

  cout << "\nObrisan element a" << endl;
  lst.remove();
  show(lst);

  cout << "\nUpisan element a" << endl;
  cout << "ISPRED njega dodat b" << endl;
  cout << "IZA njega dodat c" << endl;
  lst.insertAfter('a');
  lst.insertBefore('b');
  lst.gotoLast();
  lst.insertAfter('c');
  show(lst);

  cout << "\nLista obrisana." << endl;
```

```
lst.clear();
show(lst);

cout << "\nU listu upisani a i e." << endl;
lst.gotoFirst();
lst.insertAfter('a');
lst.insertAfter('e');
show(lst);

cout << "\nIzmedju njih dodat b" << endl;
lst.insertBefore('b');
show(lst);

cout << "\nIza tekuceg dodat d" << endl;
lst.insertAfter('d');
show(lst);

cout << "\nIspred tekuceg dodat c" << endl;
lst.insertBefore('c');
show(lst);

cout << "\nObrisan tekuci" << endl;
lst.remove();
show(lst);

cout << "\nObrisan prvi" << endl;
lst.gotoFirst();
lst.remove();
show(lst);

cout << "\nObrisan poslednji" << endl;
lst.gotoLast();
lst.remove();
show(lst);

cout << "\nOpet obrisan poslednji" << endl;
lst.gotoLast();
lst.remove();
show(lst);

cout << "\nObrisan tekuci" << endl;
```

```
lst.remove();
show(lst);


cout << "\n\nDEMONSTRACIJA ITERATORA" << endl;

cout << "\nUpisani elementi a b c d e" << endl;
lst.clear();
lst.insertAfter('a');
lst.insertAfter('b');
lst.insertAfter('c');
lst.insertAfter('d');
lst.insertAfter('e');
show(lst);

cout << "\nIteratorom se trazi element d i postavlja kao tekuci" << endl;
char el;
lst.iterator.start();
while(lst.iterator.hasNext()) {
  lst.iterator.getNext(el);
  if(el=='d') {
    lst.iterator.markAsCurrent();
    cout << "Element d je NADJEN i ucinjen tekucim" << endl;
    break;
  }
}
show(lst);

cout << "\nIteratorom se trazi element x i postavlja kao tekuci" << endl;
int nadjen = 0;
lst.iterator.start();
while(lst.iterator.hasNext()) {
  lst.iterator.getNext(el);
  if(el=='x') {
    lst.iterator.markAsCurrent();
    nadjen= 1;
    break;
  }
}
if(nadjen) cout << "Element x je NADJEN i ucinjen tekucim" << endl;
else cout << "Element x NIJE NADJEN" << endl;
```

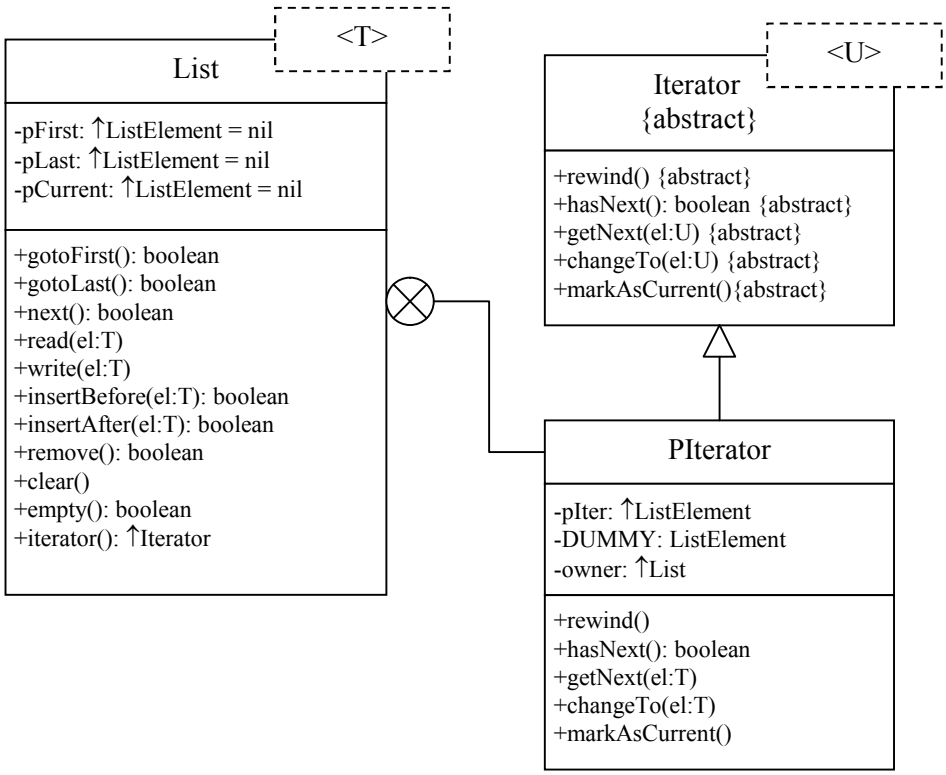
```
  show(lst);

  cout << "\nIteratorom se svaki element zamenjuje sledecim preko changeTo"
     << endl;
  lst.iterator.start();
  while(lst.iterator.hasNext()) {
    lst.iterator.getNext(el);
    lst.iterator.changeTo(++el);
  }
  show(lst);

  return 0;
}
```

9.4. PRIMER: GENERIČKA LISTA SA VIŠE ITERATORA

Prethodni primer liste sa ugrađenim iteratorom uglavnom bi zadovoljio osnovne kriterijume u pogledu njene upotrebe. Ono što se takvom realizacijom ne može postići jeste mogućnost da se za istu listu koristi više iteratora, ako treba i simultano. Ova činjenica teško da bi se mogla nazvati ozbiljnim nedostatkom, pa ćemo više u svrhu ilustrovanja generičkih klasa nego zbog neke preke potrebe, napraviti generičku listu koja može da ima više aktivnih iteratora istovremeno. Glavna svrha primera jeste da prikažemo mogućnost nasleđivanja generičkih klasa i upotrebu apstraktnih generičkih klasa.

Da bi se obezbedila mogućnost istovremenog korišćenja više iteratora, rešenje treba tražiti u realizaciji svakog iteratora kao dinamičkog objekta koji se formira na hipu i uništava kada više nije potreban[82]. To, dakle, znači da će iterator biti objekat uklopljene klase sa iteratorom koji se konstruiše posebnom funkcijom i smešta na hip odakle se uklanja standardnom operacijom *delete*. Međutim, uklopljena klasa iteratora je zatvorena, tako da se odgovarajućim dinamičkim objektom ne može rukovati metodama klase iteratora. Jedno od mogućih rešenja jeste dodati u sistem apstraktnu klasu koja je dostupna klijentima i koja sadrži apstraktne metode iteratora. Uklopljenu klasu iteratora zatim treba učiniti potklasom generičke apstraktne klase u kojoj se vrši operacionalizacija apstraktne. S obzirom na to da je apstraktna klasa ta kojoj klijenti pristupaju, ona će preuzeti naziv *Iterator*, dok ćemo unutrašnju klasu, nedostupnu za klijente, preimenovati u *PIterator*. Dijagram ovog malog sistema klasa prikazan je na slici 9.9.

[82] Inače, ovakvo rešenje nudi programski jezik java.



Slika 9.9

U samom modelu iteratora učinjene su dve izmene: klasa *List* izgubila je polje *iterator* kao nepotrebno jer se iterator formira novouvedenom metodom *iterator()* koja konstruiše na hipu instancu iteratora i kao rezultat vraća pokazivač na taj dinamički objekat. Pošto, u tom slučaju, iterator ne treba startovati (to je ugrađeno u metodu *iterator()*), klasu *Iterator* snabdeli smo novom metodom *void rewind()* koja vraća iterator u početno stanje, a da bi se izbegla potreba da se za svaku iteraciju mora na hipu konstruisati nov iterator. Umesto toga, ako postoji potreba, već postojeći iterator vraćamo u početno stanje metodom *rewind* koja je zamenila metodu *start()*, tako da se isti iterator može upotrebiti proizvoljan broj puta. Modul sa klasama *List* i *Iterator* sada ima sledeći izgled:

```
/*************************************************************

        ZAGLAVLJE MODULA SA KLASOM GENERICKE LISTE
```

```
Naziv datoteke: GENLIST.HPP
*************************************************************/
#ifndef GENLIST_HPP
  #define GENLIST_HPP

#include <mem.h>

//Apstraktna klasa iteratora
template <class U>
class Iterator {
public:
  virtual void rewind() = 0;
  virtual int hasNext() = 0;
  virtual void getNext(U& el) = 0;
  virtual void changeTo(U& el) = 0;
  virtual void markAsCurrent() = 0;
};

//Klasa sa listom
template <class T>
class List {
private:
  struct ListElement {
    T data;
    ListElement *pNext;
  };
// Uklopljena (zatvorena) klasa iteratora
  class PIterator: public Iterator<T> {
  private:
    ListElement *pIter,DUMMY;
    List *pOwner;
  public:
    PIterator(List *owner) {
      pOwner= owner;
      DUMMY.pNext= pOwner->pFirst;
      pIter= &DUMMY;
    }
    virtual ~PIterator() {}
    void rewind() {
      DUMMY.pNext= pOwner->pFirst;
      pIter= &DUMMY;
```

```
    }
    int hasNext() {
      return (pIter->pNext!=0);
    }
    void getNext(T& el) {
      pIter= pIter->pNext;
      memcpy(&el,&(pIter->data),sizeof(T));
    }
    void changeTo(T& el) {
      memcpy(&(pIter->data),&el,sizeof(T));
    }
    void markAsCurrent() {
      pOwner->pCurrent= pIter;
    }
  };
  ListElement *pFirst, *pLast, *pCurrent;
public:
  List() {pFirst= pLast= pCurrent= 0;}
  ~List();
  List(const List&);
  int gotoFirst();
  int gotoLast();
  int next();
  int read(T&);
  int write(const T&);
  int insertBefore(const T&);
  int insertAfter(const T&);
  int remove();
  void clear();
  int empty() const {return pFirst==0;}
  PIterator* iterator(){return new PIterator(this);}
};


template <class T>
List<T>::~List() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pNext; delete temp;}
  pLast= pCurrent= 0;
}
```

```
template <class T>
List<T>::List(const List<T> &rList) {
  ListElement *orig;
  if(rList.empty()) {pFirst= pLast= pCurrent= 0;}   // Prazna lista
  else {
    orig= rList.pFirst;                          // Postaviti prvi
    pCurrent= pFirst= new ListElement;
    memcpy(&(pCurrent->data),&(orig->data),sizeof(T));
    while(orig= orig->pNext) {
      pCurrent= pCurrent->pNext= new ListElement;
      memcpy(&(pCurrent->data),&(orig->data),sizeof(T)); }
    (pLast=pCurrent)->pNext= 0;
    pCurrent= rList.pCurrent;
  }
}

template <class T>
int List<T>::gotoFirst() {
  if(pFirst) {pCurrent= pFirst; return 1;}
  else return 0;
}

template <class T>
int List<T>::gotoLast() {
  if(pLast) {pCurrent= pLast; return 1;}
  else return 0;
}

template <class T>
int List<T>::next() {
  if(pCurrent) pCurrent= pCurrent->pNext;
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>::read(T &el) {
  if(pCurrent) {
    memcpy(&el,&(pCurrent->data),sizeof(T));
    return 1;}
  else return 0;
}
```

```
template <class T>
int List<T>::write(const T &el) {
  if(pCurrent) {
    memcpy(&(pCurrent->data),&el,sizeof(T));
    return 1;}
  else return 0;
}

template <class T>
int List<T>::insertBefore(const T &el) {
  ListElement *newEl, *prev, *next;
  newEl= new ListElement;
  memcpy(&(newEl->data),&el,sizeof(T));
  if(empty()) {newEl->pNext= 0;          // Lista prazna
              pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else if(pCurrent == pFirst) {          // Dodavanje ispred prvog
    newEl->pNext= pFirst;
    pCurrent= pFirst= newEl; }
  else {                                 // Ostalo
    next= pFirst;
    while(next != pCurrent) {prev= next; next= next->pNext;}
    newEl->pNext= next;
    pCurrent= prev->pNext= newEl; }
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>::insertAfter(const T &el) {
  ListElement *newEl;

  newEl= new ListElement;
  memcpy(&(newEl->data),&el,sizeof(T));

  if(empty()) {newEl->pNext= 0;          // Lista prazna
              pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else {                                 // Ostalo
    newEl->pNext= pCurrent->pNext;
    pCurrent->pNext= newEl;
```

```
   if(pCurrent == pLast) pLast= newEl;   // Dodat iza poslednjeg
   pCurrent= newEl; }
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>:: remove() {
  ListElement *prev, *next;
  if (empty()) return 0;              // Lista je prazna
  else if(pCurrent == 0) return 0;    // Tekuci prazan
  else {
    prev= next= pFirst;
    while(next != pCurrent) {prev= next; next= next->pNext;}
    if(next != pFirst) {              // Nije prvi
      prev->pNext= next->pNext;
      pCurrent= prev;
      if(next == pLast) pLast= prev; }  // Poslednji
    else {                            // Jeste prvi
      pCurrent= pFirst= pFirst->pNext;
      if(next == pLast) pLast= 0; }     // Bio jedini
    delete next;
    return 1;
  }
}

template <class T>
void List<T>::clear() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pNext; delete temp;}
  pLast= pCurrent= 0;
}

#endif
```

Primer primene sadržan je u sldećem jednostavnom programu:

```
/*****************************************************************

       DEMONSTRACIONI PROGRAM ZA GENERICKU LISTU

Naziv datoteke: LISTTST.CPP
```

```
**************************************************************/

#include <iostream>
#include "genlist.hpp"
using namespace std;


template <class T>
void show(List<T> list)
{
 T first,last,current,datum;

 if(list.empty()) cout << "List is empty." << endl;
 else {
  list.read(current);
  cout << "List contents: ";
  Iterator<T> *iter = list.iterator();
  while(iter->hasNext()){
   iter->getNext(datum);
   cout << "  " << datum;
  }
  delete iter;
  list.gotoFirst(); list.read(first);
  list.gotoLast(); list.read(last);
  cout << "\nFirst: " << first
       << "  Last: " << last
       << "  Current: " << current
       << endl;
 }
}


int main() {

 List<char> lst;

 cout << "DEMONSTRACIJA ITERATORA" << endl;

 cout << "\nUpisani elementi a b c d e" << endl;
 lst.clear();
 lst.insertAfter('a');
```

```
lst.insertAfter('b');
lst.insertAfter('c');
lst.insertAfter('d');
lst.insertAfter('e');
show(lst);

cout << "\nIteratorom se trazi element d i postavlja kao tekuci" << endl;
char el;
Iterator<char> *iter1 = lst.iterator();
while(iter1->hasNext()) {
  iter1->getNext(el);
  if(el=='d') {
    iter1->markAsCurrent();
    cout << "Element d je NADJEN i ucinjen tekucim" << endl;
    break;
  }
}
show(lst);

cout << "\nIstim iteratorom se trazi element x i postavlja kao tekuci" << endl;
int nadjen = 0;
iter1->rewind();
while(iter1->hasNext()) {
  iter1->getNext(el);
  if(el=='x') {
    iter1->markAsCurrent();
    nadjen= 1;
    break;
  }
}
delete iter1;
if(nadjen) cout << "Element x je NADJEN i ucinjen tekucim" << endl;
else cout << "Element x NIJE NADJEN" << endl;
show(lst);

cout << "\nDrugim iteratorom se svaki element zamenjuje sledecim "
          "preko changeTo" << endl;
Iterator<char> *iter2 = lst.iterator();
while(iter2->hasNext()) {
  iter2->getNext(el);
  iter2->changeTo(++el);
```

```
 }
 delete iter2;
 show(lst);

 return 0;
}
```

Verzija 2

Klasa *List* sa više iteratora može se realizovati na još jedan način koji ćemo pokazati da bismo kroz ovu, drugu, verziju ilustrovali još neke mogućnosti i rešenja, kako u domenu generičkih, tako i u domenu uklopljenih klasa. Konkretno, te mogućnosti i rešenja jesu:

- komunikacija između spoljne klase i zatvorenog dela uklopljene klase
- način pristupa klijenta uklopljenoj klasi
- tzv. "privatni konstruktor"
- instanciranje generičke klase u funkcijama.

Osnovna ideja za novo rešenje klase *List* jeste da se, umesto uvođenja apstraktne klase, uklopljena klasa iteratora jednostavno *otvori*, tj. učini *public*. Ponovo će nositi ime *Iterator*. Pošto je u tom slučaju pristup iteratoru moguć direktno, nema potrebe za posebnom apstraktnom klasom.

Dakle, prvo što treba uraditi jeste premeštanje klase *Iterator* iz segmenta *private* klase *List* u segment *public*. Za početak, to zahteva promenu realizacije klase *List* dodavanjem metode:

```
Iterator* iterator() {return new Iterator(this);}
```

jer se uklopljena klasa sada zove *Iterator*. Činjenica da je uklopljena klasa *Iterator* sada otvorena, odmah otvara jedan tehnološki problem. Naime, u tom slučaju ona se može instancirati i direktno preko sopstvenog konstruktora, što je nepotrebno (a možda i rizično) jer u tu svrhu služi upravo metoda *iterator()*! Tehnološko rešenje jeste zatvoriti konstruktor klase *Iterator* u segment *private* tako da postane nedostupan za klijenta koji je upućen isključivo na metodu *iterator()* klase *List*. Međutim, odnos spoljašnje i uklopljene klase je takav da spoljna klasa u odnosu na uklopljenu nema nikakvih privilegija u pogledu pristupa, te gornja verzija metode *iterator()* ne bi funkcionisala zato što metoda pripada klasi *List*[83], a unutar nje postoji poziv zatvorenog konstruktora *Iterator(this)*. Novi problem se u opštem - pa i ovom – slučaju rešava tako što se spoljna klasa *List* u uklopljenoj klasi *Iterator* pro-

[83] Inače, konstruktori koji su nedostupni klijentu i način njihovog korišćenja nose zajednički naziv ***privatni konstruktori***.

glašava za prijateljsku ("friend") klasu. Dakle, u klasi *Iterator* pojaviće se iskaz

friend class List;

Kompletan tekst biblioteke sa novom verzijom generičke liste ima sledeći izgled:

```
/*****************************************************************

        ZAGLAVLJE MODULA SA KLASOM GENERICKE LISTE


Naziv datoteke: GENLIST.HPP
*****************************************************************/
#ifndef GENLIST_HPP
  #define GENLIST_HPP

#include <mem.h>


template <class T>
class List {
private:
  struct ListElement {
    T data;
    ListElement *pNext;
  };
  ListElement *pFirst, *pLast, *pCurrent;
public:
// Uklopljena klasa iteratora
  class Iterator {
  private:
    ListElement *pIter,DUMMY;
    List *pOwner;
    Iterator(List *owner) {
      pOwner= owner;
      DUMMY.pNext= pOwner->pFirst;
      pIter= &DUMMY;
    }
  public:
    friend class List;
    void rewind() {
```

```
    DUMMY.pNext= pOwner->pFirst;
    pIter= &DUMMY;
   }
   int hasNext() {
    return (pIter->pNext!=0);
   }
   void getNext(T& el) {
    pIter= pIter->pNext;
    memcpy(&el,&(pIter->data),sizeof(T));
   }
   void changeTo(T& el) {
    memcpy(&(pIter->data),&el,sizeof(T));
   }
   void markAsCurrent() {
    pOwner->pCurrent= pIter;
   }
  };
  List() {pFirst= pLast= pCurrent= 0;}
  ~List();
  List(const List&);
  int gotoFirst();
  int gotoLast();
  int next();
  int read(T&);
  int write(const T&);
  int insertBefore(const T&);
  int insertAfter(const T&);
  int remove();
  void clear();
  int empty() const {return pFirst==0;}
  Iterator* iterator(){return new Iterator(this);}
};


template <class T>
List<T>::~List() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pNext; delete temp;}
  pLast= pCurrent= 0;
}
```

```
template <class T>
List<T>::List(const List<T> &rList) {
  ListElement *orig;
  if(rList.empty()) {pFirst= pLast= pCurrent= 0;}   // Prazna lista
  else {
    orig= rList.pFirst;                          // Postaviti prvi
    pCurrent= pFirst= new ListElement;
    memcpy(&(pCurrent->data),&(orig->data),sizeof(T));
    while(orig= orig->pNext) {
      pCurrent= pCurrent->pNext= new ListElement;
      memcpy(&(pCurrent->data),&(orig->data),sizeof(T)); }
    (pLast=pCurrent)->pNext= 0;
    pCurrent= rList.pCurrent;
  }
}

template <class T>
int List<T>::gotoFirst() {
  if(pFirst) {pCurrent= pFirst; return 1;}
  else return 0;
}

template <class T>
int List<T>::gotoLast() {
  if(pLast) {pCurrent= pLast; return 1;}
  else return 0;
}

template <class T>
int List<T>::next() {
  if(pCurrent) pCurrent= pCurrent->pNext;
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>::read(T &el) {
  if(pCurrent) {
    memcpy(&el,&(pCurrent->data),sizeof(T));
    return 1;}
  else return 0;
}
```

```
template <class T>
int List<T>::write(const T &el) {
  if(pCurrent) {
    memcpy(&(pCurrent->data),&el,sizeof(T));
    return 1;}
  else return 0;
}

template <class T>
int List<T>::insertBefore(const T &el) {
  ListElement *newEl, *prev, *next;
  newEl= new ListElement;
  memcpy(&(newEl->data),&el,sizeof(T));
  if(empty()) {newEl->pNext= 0;          // Lista prazna
              pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else if(pCurrent == pFirst) {          // Dodavanje ispred prvog
    newEl->pNext= pFirst;
    pCurrent= pFirst= newEl; }
  else {                                 // Ostalo
    next= pFirst;
    while(next != pCurrent) {prev= next; next= next->pNext;}
    newEl->pNext= next;
    pCurrent= prev->pNext= newEl; }
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>::insertAfter(const T &el) {
  ListElement *newEl;

  newEl= new ListElement;
  memcpy(&(newEl->data),&el,sizeof(T));

  if(empty()) {newEl->pNext= 0;          // Lista prazna
              pFirst= pLast= pCurrent= newEl;}
  else if(pCurrent == 0) delete newEl;    // Tekuci prazan
  else {                                 // Ostalo
    newEl->pNext= pCurrent->pNext;
    pCurrent->pNext= newEl;
```

```
   if(pCurrent == pLast) pLast= newEl;   // Dodat iza poslednjeg
   pCurrent= newEl; }
  return pCurrent? 1 : 0;
}

template <class T>
int List<T>:: remove() {
  ListElement *prev, *next;
  if (empty()) return 0;              // Lista je prazna
  else if(pCurrent == 0) return 0;    // Tekuci prazan
  else {
    prev= next= pFirst;
    while(next != pCurrent) {prev= next; next= next->pNext;}
    if(next != pFirst) {              // Nije prvi
      prev->pNext= next->pNext;
      pCurrent= prev;
      if(next == pLast) pLast= prev; }  // Poslednji
    else {                            // Jeste prvi
      pCurrent= pFirst= pFirst->pNext;
      if(next == pLast) pLast= 0; }     // Bio jedini
    delete next;
    return 1;
  }
}

template <class T>
void List<T>::clear() {
  ListElement *temp;
  while(pFirst) {temp= pFirst; pFirst= pFirst->pNext; delete temp;}
  pLast= pCurrent= 0;
}

#endif
```

Ostaje još da se reši pitanje instanciranja iteratora, jer se način korišćenja njegovih metoda ne razlikuje od onog u prethodnoj verziji. U odeljku o uklopljenim klasama glave 6 pomenuli smo da se uklopljena klasa referencira iz proizvoljnog klijenta samo preko spoljne klase operacijom *SpoljnaKlasa::UklopljenaKlasa*. Dakle, ako je *lst* instanca klase *List<char>*, kreiranje jednog njenog iteratora *iter* izgleda ovako:

List<char>::Iterator *iter = lst.iterator();

Sada se iterator *iter* koristi kao i u prethodnoj verziji operacijama tipa *iter→getNext()*, *iter→hasNext()* itd. Naravno, i ovde treba voditi računa da se iterator uništi operacijom *delete* u trenutku kada više nije potreban.

Na kraju, osvrnućemo se na još jedan potencijalni problem pri upotrebi generičkih klasa uopšte, pa i u ovom slučaju. Neka je potrebno napraviti funkciju *void show(List<T> list)* za prikazivanje sadržaja generičke liste na ekranu. Pošto ona treba da posluži za prikaz generičke liste, konkretan oblik *List<T>* unutar te funkcije još nije poznat. Pošto je za pokretanje iteratora potrebno njegovo kreiranje konstrukcijom prikazanom gore, trebalo bi da napišemo

List<T>::Iterator *iter = lst.iterator(); // greska!

što nije dozvoljeno u C++ jer prevodilac ne može da prepozna da li se radi o tipu uklopljene klase ili nečem drugom (npr. obraćanju statičkom polju). Ovaj problem nije moguće rešiti postojećim sredstvima, te je u C++ uvedena već pominjana službena reč ***typename*** koja se navodi ispred svakog tipa što tek treba da bude definisan. "Svaki put" znači i to da može biti upotrebljena u iskazu *template* uporedo sa rečju *class*. Korektan oblik kreiranja iteratora jeste

typename List<T>::Iterator *iter = list.iterator(); // korektno!

Sledi jednostavan primer primene ove verzije liste:

```
/*************************************************************

   DEMONSTRACIONI PROGRAM ZA KLASU "GENLIST" (genericka lista)

Naziv datoteke LISTTST.CPP
*************************************************************/

#include <iostream>
#include "genlist.hpp"
using namespace std;


template <class T>
void show(List<T> list)
{
  T first,last,current,datum;
```

```
  if(list.empty()) cout << "List is empty." << endl;
  else {
    list.read(current);
    cout << "List contents: ";
    typename List<T>::Iterator *iter = list.iterator();
    while(iter->hasNext()){
      iter->getNext(datum);
      cout << "  " << datum;
    }
    delete iter;
    list.gotoFirst(); list.read(first);
    list.gotoLast(); list.read(last);
    cout << "\nFirst: " << first
         << "  Last: " << last
         << "  Current: " << current
         << endl;
  }
}

int main()
{

  List<char> lst;

  cout << "DEMONSTRACIJA ITERATORA" << endl;

  cout << "\nUpisani elementi a b c d e" << endl;
  lst.clear();
  lst.insertAfter('a');
  lst.insertAfter('b');
  lst.insertAfter('c');
  lst.insertAfter('d');
  lst.insertAfter('e');
  show(lst);

  cout << "\nIteratorom se trazi element d i postavlja kao tekuci" << endl;
  char el;
  List<char>::Iterator *iter1 = lst.iterator();
  while(iter1->hasNext()) {
    iter1->getNext(el);
```

```
  if(el=='d') {
    iter1->markAsCurrent();
    cout << "Element d je NADJEN i ucinjen tekucim" << endl;
    break;
  }
 }
 show(lst);

 cout << "\nIstim iteratorom se trazi element x i postavlja kao tekuci" << endl;
 int nadjen = 0;
 iter1->rewind();
 while(iter1->hasNext()) {
  iter1->getNext(el);
  if(el=='x') {
    iter1->markAsCurrent();
    nadjen= 1;
    break;
  }
 }
 delete iter1;
 if(nadjen) cout << "Element x je NADJEN i ucinjen tekucim" << endl;
 else cout << "Element x NIJE NADJEN" << endl;
 show(lst);

 cout << "\nDrugim iteratorom se svaki element zamenjuje sledecim "
       "preko changeTo" << endl;
 List<char>::Iterator *iter2 = lst.iterator();
 while(iter2->hasNext()) {
  iter2->getNext(el);
  iter2->changeTo(++el);
 }
 delete iter2;
 show(lst);

 return 0;
}
```

9.5. NAPOMENE U VEZI S NASLEĐIVANJEM

U novom standardu C^{++} izvršene su neke izmene u vezi s nasleđivanjem, a

u kontekstu generičnosti. Navodimo ih jer se bez njih nasleđivanje generičkih klasa ne može realizovati. Radi se, naime, o pristupu nasleđenom sadržaju klase, a nova pravila su sledeća:

1. Nasleđenim poljima iz klase *Predak<T>*, u potklasi *Potomak<T>* može se pristupati samo eksplicitnim kvalifikovanjem iskazom *Predak<T>::*
2. Metodama bazne klase koje u prototipu nemaju generički parametar takođe se mora pristupati kvalifikovanjem sa *Predak<T>::*

Situacija je ilustrovana sledećim opštim primerom (poseban primer biće dat u sledećem poglavlju za slučaj generičkog reda):

```
template <class T>
class Predak {
protected:
  T a;
  int b;
public:
  . . . . .
  T m1() {...}
  int m2() {...}
};

template <class T>
class Potomak: public Predak<T> {
  // polju a mora se pristupati sa Predak<T>::a
  // polju b mora se pristupati sa Predak<T>::b
  // metodi m1 moze se pristupati sa m1() ili Predak<T>::m1()
  // metodi m2 mora se pristupati sa Predak<T>::m2()
};
```

10. KOREKTNOST I PREVENCIJA OTKAZA

10.1. KOREKTNOST METODE I KLASE

Od svakog proizvoda, pa tako i programskog sistema, očekuje se da "dobro radi", da je "kvalitetan". Kriterijumi kvaliteta softvera dobro su razrađeni, a jedan od primarnih jeste njegova korektnost. Korektnost softvera je dobro definisan pojam: programska jedinica (program, klasa, potprogram, metoda) je korektna ako se ponaša u skladu sa specifikacijom. Neka je X skup ulaza, a Y skup izlaza shvaćenih u najširem smislu. Specifikacija S neke programske jedinice jeste relacija između skupova X i Y koja povezuje ulaze sa izlazima, tj.

$$S \subseteq X \times Y$$

gde se domen specifikacije označava sa dom(S). Programska jedinica P u opštem slučaju takođe je relacija između skupova X i Y koja najčešće ima prirodu funkcije. Po definiciji domen programa obuhvata samo one ulaze za koje se program završava ili, kako se to kaže, *terminira*. Za programsku jedinicu P kažemo da je ***korektna*** u odnosu na specifikaciju S ako važi

$$dom(P \cap S) = dom(S)$$

što znači da za svaki ulaz program terminira i generiše izlaz predviđen specifikacijom.

Uočimo da je korektnost relativan pojam. Nijedan program nije *a priori* ni korektan ni nekorektan. Trivijalan program koji određuje vrednost $z = a+b$ može biti korektan ako tako nešto piše u specifikaciji ili nekorektan ako specifikacija predviđa da treba odrediti recimo $z = a-b$.

Korektnost realizacije klase razmatra se na dva nivoa: na nivou pojedinačnih metoda i na nivou cele klase.

10.1.1. Korektnost potprograma i metoda

Kao i svaki drugi element softverskog sistema potprogram je korektan ako je usklađen sa specifikacijom. Ovo pak direktno otvara pitanje formulisanja specifikacije koja mora biti tačna, potpuna i razumljiva [89]. Postoji čitav niz sredstava i modela za zadavanje specifikacija, a među popularnije spada korišćenje tzv. formula totalne i parcijalne korektnosti [31, 39, 48, 55, 89, 90, 94, 96, 100].

Formule totalne i parcijalne korektnosti zasnovane su na matematičkoj logici, preciznije na predikatskom računu. Mogu se primeniti na delove naredbi, naredbe, potprograme (metode) i čitave programe, koje ćemo obuhvatiti jednim imenom: sintaksne jedinice. Neka je S sintaksna jedinica. Obe formule kazuju kakva je semantika S, tj. kakav se rezultat ostvaruje izvršavanjem sintaksne jedinice, a to se postiže zadavanjem tzv. ***preduslova*** i ***postuslova***. Preduslov i postuslov (engl. "precondition" i "postcondition") su po prirodi *predikati* koji u različitim (apstraktnim) stanjima nekog programa imaju vrednosti T i $\perp$, što znači da i jedan i drugi definišu određene *skupove stanja*. Za datu sintaksnu jedinicu S preduslov određuje stanje u kojem otpočinje izvršavanje S, a postuslov stanje u kojem se S završava. ***Formula totalne korektnosti*** je predikat koji se prikazuje sa

$$\{P\}S\{Q\}$$

gde su P i Q redom preduslov i postuslov. Interpretacija ove formule glasi

- Ako sintaksna jedinica S otpočinje u stanju koje zadovoljava preduslov P tada (1) S terminira (završava se) i (2) stanje po završetku S zadovoljava predikat Q.

Druga formula, ***formula parcijalne korektnosti***, podseća na prethodnu s tim što je terminiranje deo antecedente, a ne konsekvente. Formula parcijalne korektnosti obično se prikazuje sa

$$P\{S\}Q$$

uz interpetaciju

- Ako sintaksna jedinica S otpočinje u stanju koje zadovoljava preduslov P i ako sigurno terminira, tada završno stanje zadovoljava predikat Q.

U čast tvorca, C.A.R. Hoare-a, izrazi gornjeg oblika zovu se i Hoarovi tripleti (trojke). Inače, formalna definicija Hoarovih tripleta i neke njihove važne osobine mogu se naći u [31] i [93]. Uočimo odmah da su obe formule predikati, te sledstveno mogu biti tačne, ali i netačne. Pitanje terminiranja po kojem se razlikuju

dve formule je od velike važnosti u teoriji programiranja, dok se u praksi o njemu vodi nešto manje računa (Hehner, [95]). Formule totalne odnosno parcijalne korektnosti omogućuju formalizaciju pojma korektnosti uopšte:

- Potprogram *S* je totalno korektan u odnosu na specifikaciju formulisanu uređenim parom (P,Q) preduslova i postuslova ako je predikat {P}S{Q} tačan. Ista tvrdnja važi i za metodu.
- Potprogram je parcijalno korektan ako zadovoljava isti uslov, ali uz korišćenje formule parcijalne korektnosti P{S}Q.

Pošto je formula totalne korektnosti nešto više tretirana u literaturi, u nastavku ćemo se zadržati samo na njoj. Kako smo rekli, primenljiva je na sve nivoe složenosti, od dela naredbe, pa do čitavog programa. Na primer, formula

$$\{x=1\}\ x:=x+1\ \{x'=2\}$$

znači da sintaksna jedinica dodele prevodi svako stanje u kojem je vrednost promenljive x bila 1 u stanje u kojem x ima vrednost 2. Oznake x i x' su uobičajene za vrednost promenljive x pre odnosno posle izvršavanja sintaksne jedinice. Za istu sintaksnu jedinicu može se definisati proizvoljan broj Hoarovih tripleta, tačnih i netačnih. Sledeći tripleti su tačni:

$$\{x=1\}\ x:=x+1\ \{x'>1\}$$
$$\{x=1\}\ x:=x+1\ \{x'\neq 0\}$$
$$\{x=2\}\ x:=x+1\ \{x'=3\}$$

a sledeći nisu:

$$\{x=1\}\ x:=x+1\ \{x'=1\}$$
$$\{x=1\}\ x:=x+1\ \{x'<0\}$$
$$\{x=2\}\ x:=x+1\ \{x'=2\}$$

Pitanjima jednoznačnog definisanja preduslova odn. postuslova bavili su se Dajkstra i Flojd. Više o ovoj tematici može se naći u [31] ili [96].

Ono što je od interesa za naša razmatranja jeste korišćenje Hoarovih tripleta u svrhu definisanja semantike potprograma i naročito metoda. Neka je *f* potprogram ili metoda neke klase. Neka su P i Q predikati koji predstavljaju redom preduslov i postuslov. Ako uređeni par (P,Q) prihvatimo kao formalizaciju specifikacije tada je *f* korektna u odnosu na specifikaciju (P,Q) ako je predikat {P}f{Q} tačan. Neka je *sort* funkcija koja služi za sortiranje realnog niza *s* sa $n\geq 2$ elemenata u rastućem poretku. Tada se korektnost *sort* istražuje na Hoarovom tripletu

$$\{(s \text{ je realan niz dužine } n) \wedge (n \geq 2)\} \text{ sort}(s,n) \{s'_i \leq s'_{i+1}, i=1, ..., n-1\}$$

Ako je triplet tačan kažemo da je *sort* korektan u odnosu na specifikaciju datu preduslovom i postuslovom.

Kao što vidimo, preduslov i postuslov mogu da se zadaju strogo, korišćenjem matematičke notacije, ali i manje formalno, upotrebom govornog jezika. Sve što je potrebno jeste ispunjenje zahteva u pogledu tačnosti, potpunosti i razumljivosti.

Inače, Hoarovi tripleti imaju primenu i izvan razmatranja korektnosti. Jedan od vrlo često korišćenih načina *komentarisanja* potprograma obuhvata uključivanje preduslova i postuslova. Komentar se zadaje na samom početku potprograma (preciznije, iznad zaglavlja) i u najužoj formi obuhvata opis posla koji potprogram obavlja, zatim preduslov i konačno postuslov.

10.1.2. Korektnost klase

U dosadašnjem izlaganju koncentrisali smo se na opšta pitanja korektnosti koja su vezana za sve sintaksne jedinice tipa potprograma. Razmatranje korektnosti metoda umnogome se poklapa sa razmatranjem korektnosti (slobodnih) potprograma, jer i same metode imaju mnoge osobine potprograma. Razlika naravno postoji i proističe iz nesamostalnosti metoda, što ne može a da nema uticaja i na definisanje odgovarajućih Hoarovih tripleta. Naime, preduslov odn. postuslov metode na neki način moraju zavisiti i od stanja objekta kojem se putem aktiviranja šalje poruka, čime se korektnost metoda direktno povezuje sa korektnošću klase. S druge strane, klasa nije prost skup metoda te se ni njena korektnost ne može svesti samo na nekakav "zbir" korektnosti pojedinačnih metoda. Uopšte uzev, formalna korektnost svake metode ponaosob ne garantuje korektnost ponašanja klase u celini. Posmatrajmo, kao primer, klasu *Trougao* i njenu metodu *Povrsina()*. S obzirom na to da metoda nema parametara moglo bi se zaključiti da je njen preduslov P≡T, tj. da u svakom stanju objekta klase *Trougao* metoda izdaje tačan rezultat. Ovo je naravno pogrešno jer P≡T obuhvata i stanja u kojima su stranice trougla, recimo, negativne!

Razrešenje ovog problema ne treba tražiti u definisanju oblika preduslova nego na drugoj strani: u propisivanju *dozvoljenih* stanja objekta klase. Drugim rečima, treba poći od pitanja "može li se dozvoliti da objekat klase *Trougao* bude u stanju u kojem su dužine stranica negativne?". Iako je odgovor svakako "ne može", pitanje nije baš sasvim retoričko jer treba razjasniti i zašto ne može. Nijedan objekat klase *Trougao* ne sme biti u pomenutom stanju *jer takav pojam ne postoji*, a objekat je - ne zaboravimo - model pojma.

Entiteti koje u domenu problema uključujemo u zajedničku klasu entiteta imaju iste bitne osobine - one na osnovu kojih su uvrštene u klasu. Svi individualni pojmovi klasnog pojma TROUGAO imaju po tri stranice. To, međutim, nije sve jer

između osobina entiteta postoje neke relacije koje se moraju očuvati u svakom od individualnih pojmova. U klasi TROUGAO to je dobro poznata karakteristika da zbir dužina ma koje dve stranice mora biti veći od dužine treće. Kada se to prenese na nivo modela sledi da svi objekti date klase imaju iste članove, ali i da održavaju neke konstantne relacije između njih. Ako su *s1*, *s2* i *s3* podaci-članovi klase *Trougao* koji predstavljaju dužine stranica, tada za svaki objekat u svakom njegovom stanju mora važiti

$$(s1<s2+s3)\wedge(s2<s1+s3)\wedge(s3<s1+s2).$$

Obratimo pažnju na to da je ovo ograničenje suštinsko, jer objekat klase *Trougao* ne može uopšte postojati ako ne ispunjava navedeni uslov! Slična ograničenja postoje za svaku klasu objekata jer su deo definicije odgovarajuće klase entiteta i nose naziv *invarijante klase*. Neformalno, invarijanta klase jeste dakle skup tvrdnji koje važe za svaki objekat u svakom stanju. Zapazimo odmah da invarijanta važi u svakom stanju ali ne i u svakom trenutku, pošto u prelaznom režimu može da se naruši. Naravno, po izvršenom prelazu invarijanta se obavezno ponovo uspostavlja, ako je uopšte bila narušavana.

Iako je i sam po sebi koncept invarijante prilično jasan, pokušaćemo da ga formalizujemo, prvenstveno zato što se pokazuje da je u pitanju jedna od fundamentalnih karakteristika svake klase. Pre svega, umesto pojma tvrdnje upotrebićemo pojam predikata definisanog na prostoru stanja klase. Pošto svi predikati koji ulaze u invarijantu moraju biti u svakom stanju tačni i samu invarijantu preciznije definišemo ne kao skup nego kao konjunkciju predikata. Prema tome, ako uvedemo oznake I_1, I_2 i I_3 za predikate redom:

$$I_1: s1<s2+s3$$
$$I_2: s2<s1+s3$$
$$I_3: s3<s1+s2$$

tada je invarijanta klase *Trougao* oblika $I_1\wedge I_2\wedge I_3$. Dalje, svaki objekat klase, kada nije u prelaznom režimu, nalazi se u nekom od stanja iz prostora stanja. Shodno tome,

- ***invarijanta klase*** je predikat koji ima vrednost T u svakom stanju svakog objekta te klase.

Prema ovoj definiciji invarijanta klase nije jedinstvena. Već na primeru klase *Trougao* zapažamo da je $I_1\wedge I_2\wedge I_3$ invarijanta, ali su invarijante i predikati I_1, I_2 i I_3 kao i konjunkcije $I_1\wedge I_2$, zatim $I_1\wedge I_3$ te $I_2\wedge I_3$. Predikat $I_0\equiv T$ je u tom smislu invarijanta svake klase. Mada to u praksi ne stvara probleme, definisaćemo još jedan oblik invarijante koji je jedinstven i koji ima izuzetnu teorijsku važnost. Radi se o tzv. *strogoj invarijanti klase*. U uvodnom razmatranju pomenuli smo da se objekat klase

sme naći samo u dozvoljenim stanjima, što odmah znači da postoje i ona druga - nedozvoljena. Nedozvoljena stanja za objekte klase *Trougao* su stanja u kojima nije tačan predikat $I_1 \wedge I_2 \wedge I_3$. Sledstveno, prostor stanja date klase *K*, u oznaci U_K, jeste skup svih *mogućih* stanja, dozvoljenih i nedozvoljenih, pri čemu objekat ulazi u nedozvoljeno stanje - greškom. Za klasu *Trougao* sa realnim podacima-članovima *s1*, *s2* i *s3* prostor stanja jeste skup R^3 gde je R skup realnih konstanti. Predikat I_K je ***stroga invarijanta*** klase *K* pod sledećim uslovima:

1. I_K je invarijanta klase *K*.
2. Ako je *I* invarijanta klase *K* različita od I_K tada važi $I_K \Rightarrow I$, pri čemu implikacija ima značenje $(\forall u \in U_K)\ [I_K(u) \Rightarrow I(u)]$, tj. važi na domenu interpretacije koji predstavlja kompletan prostor stanja klase.

Prvo, stroga invarijanta uvek postoji zato što uvek postoji invarijanta $I \equiv T$. Ako su I_1 i I_2 invarijante i ako nije tačno ni $I_1 \Rightarrow I_2$ niti $I_2 \Rightarrow I_1$ tada je $I_1 \wedge I_2$ stroga invarijanta. Dakle, stroga invarijanta uvek postoji i jedinstvena je do nivoa ekvivalencije, tj. sve stroge invarijante su međusobno ekvivalentne. I ne samo to: stroga invarijanta seže najdublje u samu definiciju klase. U tom smislu pokazaćemo da ona jednoznačno određuje skup stanja klase, kao skup svih stanja koje instance klase smeju da zauzmu. Neka je S skup svih dozvoljenih stanja i neka je I stroga invarijanta neke klase. Neka je, dalje, U prostor stanja klase za koji važi $U \supseteq S$. Na osnovu definicije (svake) invarijante sigurno je $I(s)=T$ za svako stanje $s \in S$. Neka je *u* stanje takvo da važi $I(u)=T$ i $u \notin S$. Za ovo stanje mora postojati neki predikat Z po kojem se ono razlikuje od svih stanja iz skupa S, tj. $Z(u)=T$ i $Z(s)=\perp$ za svako stanje s iz S. Kako je $\neg Z(s)=T$ za sva stanja iz S, sledi da je $\neg Z$ invarijanta klase te, po definiciji, $I \Rightarrow \neg Z$ na celom prostoru stanja U, pa i u stanju u. Međutim, ako važi $I(u) \Rightarrow \neg Z(u)$ tada ne može biti tačno $I(u) \wedge Z(u)$, tj. stanje poput *u* ne može postojati u skupu U. Ovim smo pokazali fundamentalnu osobinu stroge invarijante:

- Stroga invarijanta I_K klase *K* sa prostorom stanja U_K u celosti definiše skup dozvoljenih stanja, tj. $S_K = \{s|\ (s \in U_K) \wedge I_K(s)\}$. Štaviše, skup dozvoljenih stanja može se definisati kao skup svih stanja iz prostora stanja U_K koja zadovoljavaju strogu invarijantu klase.

Očigledno, invarijanta klase, a naročito stroga invarijanta, odražava njene suštinske, definicione osobine. Kao takva ona mora biti u vezi sa njenom definicijom, a preko definicije i sa relacijom nasleđivanja. Nije teško uočiti da se

- invarijanta klase može predstaviti kao konjunkcija invarijanti svih njenih predaka i dela koji je specifičan za samu klasu.

Invarijanta I_{PT} klase *PravougliTrougao* koja nasleđuje klasu *Trougao* može se predstaviti kao

$$I_{PT} = (I_1 \wedge I_2 \wedge I_3) \wedge (s3^2 = s1^2 + s2^2)$$

gde su *s1*, *s2* i *s3* dužine stranica, a $I_1 \wedge I_2 \wedge I_3$ je ranije uspostavljena invarijanta klase *Trougao*.

Opskrbljeni konceptom invarijante klase možemo se vratiti na razmatranje korektnosti metoda. Kako smo videli, invarijanta klase mora biti zadovoljena u svakom stanju objekta. S druge strane, promene stanja izvode se isključivo posredstvom metoda, što neposredno vodi ka novim zaključcima o preduslovima i postuslovima metoda. Ako za trenutak ostavimo po strani metode koje kreiraju odnosno uništavaju objekte, sve ostale metode iz interfejsa klase moraju održavati njenu invarijantu. To znači da one otpočinju i isto tako završavaju u stanjima koja zadovoljavaju invarijantu klase. Izraženo jezikom Hoarovih tripleta to znači da je invarijanta klase nezaobilazni deo kako preduslova tako i postuslova svake metode osim metoda za kreiranje i uništavanje. Neka je *m* metoda klase *K* sa punim preduslovom PR_m odnosno postuslovom PO_m. Ako je I_K stroga invarijanta klase tada preduslov i postuslov nužno imaju oblik

$$PR_m = P_m \wedge I_K$$
$$PO_m = Q_m \wedge I_K$$

gde su P_m i Q_m preduslov i postuslov kojima rukujemo, dok se invarijanta klase I_K podrazumeva kao obavezni deo. Kada, dakle, za metodu *m* formiramo Hoarov triplet $\{P_m\}m\{Q_m\}$, u stvari podrazumevamo

$$\{P_m \wedge I_K\}\ m\ \{Q_m \wedge I_K\}.$$

Metode za kreiranje i uništavanje - u daljem konstruktori i destruktori - razlikuju se od ostalih metoda po tome što konstruktor dovodi još nepostojeći objekat u početno stanje, a destruktor ga uništava, posle čega nema objekta, pa ni njegovog stanja. Ovo se vrlo jasno opaža upravo prilikom definisanja Hoarovih tripleta. Ako je *c* neki konstruktor sa preduslovom P_c i postuslovom Q_c, a *d* destruktor klase *K* (preduslov P_d i postuslov Q_d) tada važi

$$\{P_c\}\ c\ \{Q_c \wedge I_K\}$$
$$\{P_d \wedge I_K\}\ d\ \{Q_d\}$$

gde su P_c i Q_d predikati čiji domen nije prostor stanja klase.

Druga posebnost odnosi se na funkcije-članice klase koje nisu metode (tj. one koje nisu *public*). S obzirom na to da se one ne mogu pobuditi spolja sasvim je moguće da se angažuju samo u toku prelaznog režima, tokom promene stanja objekta. Iz toga neposredno sledi zaključak da zatvorene funkcije članice mogu i ne moraju da održavaju invarijantu klase.

Sada možemo pristupiti definiciji ***korektnosti klase*** u celini. Klasa *K* sa

strogom invarijantom I_K je korektna ako zadovoljava sledeće uslove:

1. Za svaki konstruktor *c* (ili inicijalizator ako nema konstruktora) važi $\{P_c\}$ c(argumenti) $\{Q_c \wedge I_K\}$.
2. Za destruktor *d* važi $\{P_d \wedge I_K\}$ d $\{Q_d\}$.
3. Za svaku drugu metodu *m* (funkciju-članicu koja nije zatvorena) važi $\{P_m \wedge I_K\}$ m(argumenti) $\{Q_m \wedge I_K\}$.

Umesto primera pokazaćemo sada kako i suptilna greška u promišljanju invarijante klase neposredno uzrokuje probleme pri realizaciji klase. Posmatraćemo opet klasu koja modeluje trougao. Za ovaj primer odlučujemo se baš stoga što se radi o entitetu koji je dobro definisan i poznat svima, pa ipak ...

Vratimo se na hijerarhiju trouglova iz primera 7.1 i uočimo klase *Trougao* i *PravougliTrougao*. Pretpostavimo da klasa *Trougao*, pored ranije navedenih metoda, poseduje i metode za promenu vrednosti stranica *PromeniA(x:real)*, *PromeniB(y:real)* i *PromeniC(z:real)* koje menjaju vrednosti podataka-članova redom *s1*, *s2* i *s3*. Metode deluju nevino i verovatno bi ih neki programeri uključili u klasu. Da bi bilo koja od njih bila primenljiva treba da očuva invarijantu klase. To znači da npr. vrednost *x* u *PromeniA* mora biti takva da obezbedi tačnost predikata

$$(x<s2+s3)\wedge(s2<x+s3)\wedge(s3<x+s2).$$

Usled postojanja veze nasleđivanja, navedene metode pojaviće se i u klasi *PravougliTrougao*, pri čemu metoda *PromeniC* u tom slučaju menja vrednost hipotenuze. Šta se, međutim, dešava kada porukom obPTr.PromeniC(h) u objektu *obPTr* promenimo vrednost hipotenuze na *h*, preko *nasleđene* metode *PromeniC*? Odgovor glasi: dolazi do haosa. Naime, da bi se očuvala invarijanta klase *PravougliTrougao* koja, pored nasleđene invarijante, sadrži i predikat $s1^2=s2^2+s3^2$ ta metoda mora se redefinisati - ali kako kada vrednost hipotenuze ne određuje jednoznačno vrednosti obe katete? Moguće rešenje bilo bi pomnožiti u okviru redefinisane metode *PromeniC* vrednosti s1 i s2 faktorom h/s3 uspostavljajući tako invarijantu. Ovo ipak nema mnogo smisla jer se ne radi o postupku uobičajenom u matematici - nema nikakvog posebnog razloga za izdvajanje linearne promene vrednosti kateta od neke druge koja takođe čuva invarijantu (uzgred, takvih "drugih" načina promene ima beskonačno mnogo).

Potražimo zato drugi izlaz: *preispitati uključivanje navedenih metoda u klasu*. Iako na prvi pogled deluje čudno, ovo rešenje ne samo što ima smisla, nego je i jedino koje je ispravno. Razlog je jednostavan: ako nekom trouglu kao matematičkom entitetu izmenimo dužine stranica to više nije isti trougao! Prisetimo se elementarne geometrije: postoje različiti trouglovi, slični trouglovi ili podudarni trouglovi, ali ne i promenljivi trouglovi. Dakle, stvarna stroga invarijanta klase *Trougao* nije samo napred navedeni predikat $I_1 \wedge I_2 \wedge I_3$ nego

$$(I_1 \wedge I_2 \wedge I_3) \wedge (s1, s2, s3 = const)$$

gde drugi deo označava da, jednom zadate, dužine stranica više ne mogu menjati vrednost, te stoga metodama *PromeniA*, *PromeniB* i *PromeniC* u klasi nema mesta. Jednostavno, kada se trouglu promene dužine stranica to više nije isti trougao.

Inače, ako je iz nekog razloga baš neophodno formirati klasu trouglova sa promenljivim stranicama to se može uraditi, no tada se definiše posebna klasa, npr. *PromenljiviTrougao*, čiji se naslednici - ako ih ima - snabdevaju specifičnim pravilima za promenu dužina stranica.

10.2. PREVENCIJA OTKAZA. RUKOVANJE IZUZECIMA

Svaki, pa i sasvim jednostavan sistem sadrži defekte. Softver, kao potencijalno visoko kompleksan sistem, toliko je podložan greškama da je sasvim opravdano pretpostaviti da ih svaki programski sistem ima. Praksa samo potvrđuje ovaj stav: analizom velikog broja projekata ustanovljeno je da složeni softverski sistemi sadrže 1-2% naredbi sa greškama (Lipajev, [33]) što je, imajući u vidu njihovu veličinu, itekako respektabilan iznos.

Kada su u pitanju greške, kvalitetan program mora da ispuni dva zahteva. Prvo, treba da bude *korektan*, tj. da odgovara specifikaciji, o čemu smo govorili u prethodnom odeljku. Drugo, mora biti *robustan* što znači otporan na neregularne ulazne podatke. Ova dva zahteva nisu međusobno uslovljeni. Korektan program, dakle napisan u skladu sa specifikacijom, nije obavezno zaštićen od unosa neregularnih (tzv. invalidnih) ulaznih podataka. Može čak da reaguje nepredvidivo ili, kako se u engleskom žargonu kaže, da izvede "jump in blue". S druge strane, u robustnom programu nema takvih poslastica, ali ni garancije da će izlaz biti tačan. Mejer [82, 97] pojmove korektnosti i robustnosti objedinjava u ***pouzdanost***, mada neki autori pouzdanost poistovećuju samo sa robustnošću.

U programerskom žargonu uobičajeno je da se svaka anomalija naziva greškom, često i "bagom" (od engleskog bug = buba, zbog insekta koji je svojevremeno izazvao kratak spoj u računaru). Stručnjaci za ovu oblast računarstva pak upotrebljavaju znatno širu taksonomiju [73, 92] koju ćemo ukratko izložiti. Prvo, u toku procesa realizacije programer može da pogreši u donošenju odluke o upisu nekog kôda u program. Ova vrsta mentalne aktivnosti nosi naziv *greška* (engl. "error") u užem smislu reči. Rezultat greške pojavljuje se u programu kao *defekt* ("fault"). Ako se u toku izvršavanja programa naiđe na defekt, dolazi do *otkaza* ("failure"). Otkaz može da se manifestuje, ali ne mora. Eksplicitna, vidljiva manifestacija otkaza nosi naziv *incident*. Inače, otkaz može da se manifestuje na više načina:

- Ne izaziva incident, što je krajnje nepoželjno jer pogrešan rezultat generisan programom prolazi neopaženo.

- Uzrokuje nepredviđeni prekid programa što takođe nije naročito privlačno, ali je bolje od prethodnog jer korisnik ipak dobija informaciju o tome da sa programom nešto nije u redu.
- Izaziva predviđeni ("havarijski") prekid programa. Ovo je nešto malo povoljnije od prethodnog, jer korisnik dobija informaciju o prekidu programa *i* uzroku prekida.
- Program se privremeno zaustavlja, generiše se poruka o otkazu i korisniku nudi repertoar aktivnosti iz kojeg se bira način reakcije.

Inače, otkaz programa može izazvati i zaustavljanje ("pad") operativnog sistema, što nije nimalo prijatno. Međutim, u takvim situacijama ostaje otvoreno pitanje da li je otkaz operativnog sistema rezultat defekta u programu ili - što je verovatnije - propusta u samom operativnom sistemu.

U objektnom ambijentu, postoje dva tipična slučaja pojave defekta: jedan je defekt u metodi neke klase; drugi se pak odnosi na defekt u klijentu, gde se korektna metoda poziva u neregularnim uslovima (npr. pokušaj čitanja iz prazne liste). Iako se u oba slučaja radi o pogrešno unetom kodu, postoji razlika u tretmanu. Defektna naredba u samoj metodi (ako nije poziv druge metode) sa velikom verovatnoćom poklapa se sa mestom otkaza ili mu je bar blizu. Najčešće se traži pokušajem izazivanja incidenta u uslovima neizvesnosti, jer se ne zna da li defekt uopšte postoji i posebno gde je lociran. Defekt u klijentu se ne nalazi na mestu otkaza. Primer je upravo pokušaj očitavanja prazne liste gde je (defektna) naredba očitavanja u klijentu, dok je mesto manifestovanja otkaza u (korektnoj) metodi klase-servera. Otkrivanje defekta u klijentu može se olakšati ako je pozvana metoda snabdevena zaštitom koja će sprečiti otkaz usled pogrešnog poziva *i o tome izvestiti*.

Postoje tri načina za predupređivanje otkaza, od kojih su prva dva karakteristična za sve procedurne jezike. Radi se o sledećim postupcima:

Metoda 1. Upravljanje putem prekida programa, gde u slučaju otkaza potprogram u kojem je nastao otkaz nameće potpuni prekid izvršavanja programa preko standardnih rutina (npr. *abort* ili *exit* u C/C^{++} ili *Halt* u paskalu).

Metoda 2. Upravljanje preko izlaza potprograma, gde se prosleđivanje koda ishoda klijentu vrši putem izlazne vrednosti potprograma.

Metoda 3. Primena posebnog mehanizma, tzv. rukovanja izuzecima, pod uslovom da ga programski jezik poseduje. Stariji programski jezici (paskal, starije varijante C) nemaju ugrađeno rukovanje izuzecima.

Zajednička karakteristika sve tri metode jeste ***preventivnost***, tj. sprečavanje izvršavanja naredbi koje izazivaju otkaz. Prve dve metode primenljive su u praktično svim imperativnim programskim jezicima, dok je treća novijeg datuma.

Prvi tip upravljanja otkazima upotrebljava se u slučajevima havarije programa, kada je jedina suvisla reakcija njegovo momentalno prekidanje. Karakterističan je po tome što se reakcija na otkaz nalazi takođe u potprogramu, a ne u kli-

jentu. Takođe, diskutabilno je u kojoj meri ova metoda stvarno služi za prevenciju otkaza jer se ona očigledno svodi na sprečavanje jednog otkaza izazivanjem drugog.

Drugi način za obradu otkaza - prosleđivanje klijentu koda (šifre) otkaza - ne zahteva od programskog jezika nikakve specijalne odlike, te je zato na raspolaganju u svakom imperativnom jeziku uključujući i najstariji - fortran. Izvodi se tako što povratna vrednost rizičnog potprograma više nije traženi rezultat, nego predstavlja kod (šifru) ishoda koja se vraća klijentu na obradu, dok se osnovni izlaz prosleđuje putem izlaznog parametra.

Rukovanje izuzecima (engl. "exception handling") je najbolji način za obradu iole složenijih otkaza. Neformalno, izuzetak je događaj koji se dešava kao posledica izvršavanja defektnog koda. Pošto isto važi i za otkaz, može se steći pogrešan utisak da su u pitanju sinonimi. Radi se, međutim, o sasvim različitim vrstama događaja i to iz tri razloga: (1) svrha izuzetka jeste upravo *sprečavanje pojave otkaza*; (2) otkaz je neželjeni događaj, dok se izuzetak izaziva *hotimice*, ugrađivanjem posebnih naredbi i (3) izuzetak se uvek manifestuje. Izuzetak se ne može poistovetiti ni sa incidentom jer ni do incidenta ne dolazi namerno, tako da o izuzetku možemo govoriti kao o kvazi-incidentu. Moguća ***definicija izuzetka*** bila bi:

- Izuzetak (engl. "exception") je *namerno* izazvan *događaj* čija je svrha predupređivanje otkaza i koji ostavlja program u zatečenom stanju (*status quo ante*). Obavezno je propraćen prijavom nastanka.

Mejer [82] daje možda najčistiju sliku odnosa otkaza i izuzetka. On polazi od *ishoda* izvršavanja rutine koji može da bude *uspešan* ako rutina završi u stanju predviđenom specifikacijskim postuslovom, odnosno *neuspešan* u suprotnom. I otkaz i izuzetak izazivaju neuspešan ishod rutine, ali je razlika u tome što je izuzetak kontrolisan i što sam po sebi ne menja stanje programa.

Postupak rukovanja izuzecima podrazumeva način programiranja koji će obezbediti detekciju barem očekivanih izuzetaka; drugim rečima, *izaziva se incident*[84] i klijent se izveštava o njegovom nastanku i uzroku. Osnovna ideja preuzeta je iz metode 2, s tim da kod izuzetka ne pripada ni listi argumenata niti je izlaz potprograma. Prenosi se i prihvata kao poseban, nevidljiv podatak ili objekat, čime se otklanja osnovni nedostatak metode 2 - upotreba izlaza potprograma u svrhu za koju nije predviđen. Ukoliko je potrebno, izuzetak može da izazove i prekid programa, tako da rukovanje izuzecima obuhvata obe prethodne metode. Metoda je zasnovana na sledećim *principima*:

- Izuzetak generisan nekom porukom detektuje se na mestu nastanka, a obrađuje na mestu odakle je poruka poslata.
- Izuzetak je (kontrolisana) anomalija te, shodno tome, zahteva poseban

[84] Tačnije, izaziva se kvazi-incident jer do otkaza ne dolazi.

tretman.

Kako vidimo, upravljanje otkazima prekidom programa ne poštuje prvi princip, jer se prekid nalazi u potprogramu u kojem je nastao incident, dok upravljanje putem izlaza potprograma nije u skladu sa drugim principom, jer poruku o incidentu prosleđuje istim kanalima kojima prosleđuje i regularne rezultate.

10.2.1. Prevencija otkaza u C^{++} prekidom programa i izlazom funkcije

Postupke ćemo ilustrovati na standardnom primeru najjednostavnije strukture podataka - steka. Posmatraćemo celobrojni stek kapaciteta zadatog simboličkom konstantom CAPACITY. Stek ima uobičajenih pet metoda: *top* za očitavanje, *pop* za uklanjanje, *push* za dodavanje i *empty* odn. *full* za utvrđivanje da li je stek redom prazan ili pun. Realizovan je sekvencijalno, preko niza *s[CAPACITY]* u koji se smeštaju elementi. Podatak-član *t* predstavlja indeks aktuelnog vrha steka. Originalni kod (bez generisanja izuzetaka) ima sledeći izgled:

```
// OSNOVNI KOD KLASE SEKVENCIJALNOG STEKA
#define CAPACITY 100
class Stack  {
private:
  int t;
  int s[CAPACITY];
public:
  Stack() {t = -1;}
  int empty() const {return t < 0;}
  int full() const {return t == CAPACITY-1;}
  int top() const;
  void pop();
  void push(int el);
};
int Stack::top() const {return s[t];}
void Stack::pop() {t--;}
void Stack::push(int el) {s[++t] = el;}
```

Pre svega, do otkaza može doći u slučajevima očitavanja praznog steka (funkcija *top*), uklanjanja iz praznog steka (funkcija *pop*) i dodavanja u pun stek (funkcija *push*). Ovim otkazima odgovaraju, u stvari, samo dva događaja: jedan nastaje u slučaju primene *top* i *pop* na prazan stek i rezultuje otkazom tipa "stack underflow", dok dodavanje elementa u pun stek generiše događaj poznat pod nazivom "stack overflow".

Upravljanje putem prekida programa. Postupak se svodi na eksplicitno ili implic-

itno aktiviranje standardne funkcije *abort* koja izaziva prekid programa, a treba ga primeniti samo u havarijskim situacijama, kada program ne sme da se nastavi ni pod kojim uslovima. U tu svrhu C/C^{++} ima makrodirektivu *assert* koja se nalazi u biblioteci ASSERT.H, sa opštim oblikom

assert(predikat);

Makrodirektiva je, u stvari, vrlo jednostavna. Ona se razvija u naredbu

if(!predikat) {poruka o grešci; abort();}

koja u slučaju da predikat nije tačan prekida program uz odgovarajuću poruku. Primenom ove makrodirektive dobijamo sledeći kod za tri metode steka:

```
// UPRAVLJANJE OTKAZIMA PREKO PREKIDA PROGRAMA
int Stack::top() const {
  assert(t >= 0);
  return s[t];
}
void Stack::pop() {
  assert(t >= 0);
  t--;
}
void Stack::push(int el) {
  assert(top < CAPACITY-1);
  s[++top] = el;
}
```

Ozbiljan nedostatak ove metode leži u činjenici da je rukovanje otkazom ostavljeno funkciji u kojoj je do otkaza i došlo (tačnije u kojoj *bi* došlo do otkaza ako se program ne prekine). Upravljanje se više ne vraća klijentu, a krajnji ishod je nešto što korisnike ne čini preterano srećnim: prekid programa. Zato se ovaj postupak koristi isključivo u situacijama kada je originalni otkaz havarijski, tj. kada više nema svrhe nastavljati sa obradom.

Upravljanje otkazima preko izlaza funkcije. Ideja se sastoji u tome da *sve* (rizične) funkcije vraćaju rezultat i da taj rezultat ukazuje na ishod obrade. Najjednostavnije rešenje je binarni izlaz gde jedna vrednost znači uspeh, a druga neuspeh. Ako proanaliziramo biblioteke C/C^{++} primetićemo da mnoge standardne funkcije koriste ovaj postupak: na primer, funkcija za otvaranje datoteka *fopen* vraća kao rezultat NULL ako otvaranje nije uspelo odnosno pokazivač različit od NULL u slučaju regularnog završetka; funkcija za zatvaranje datoteke *fclose* sa celobrojnim izlazom

vraća 0 ako je datoteka uspešno zatvorena i vrednost različitu od 0 u slučaju neuspeha.

Ako je iz bilo kojih razloga neophodno upoznati klijenta sa preciznijim opisom uzroka otkaza, umesto binarnog *uspeh-neuspeh* izlaza koristi se polivalentan izlaz sa vrednostima koje predstavljaju kod ishoda, najčešće tipa enumeracije. Svaka od vrednosti enumeracije odgovara po jednoj varijanti izlaza, uključujući i uspešan završetak. Demonstriraćemo ovaj postupak na primeru steka. Neka je tip ishoda dat enumeracijom

```
enum OutcomeType {OK, UNDERFLOW, OVERFLOW};
```

Sada tri rizične funkcije dobijaju sledeći oblik:

```
// UPRAVLJANJE OTKAZIMA PREKO IZLAZA FUNKCIJE
OutcomeType Stack::top(int &val) const {
  OutcomeType outc;
  if(t<0) outc= UNDERFLOW;
  else {
  val= s[t];
  outc= OK;
  }
  return outc;
}     // Napomena: treba promeniti i prototip!

OutcomeType Stack::pop() {
  OutcomeType outc;
  if(t<0) outc= UNDERFLOW;
  else {
    t--;
    outc= OK;
  }
  return outc;
}

OutcomeType Stack::push(int el) {
  OutcomeType outc;
  if(top==CAPACITY-1) outc= OVERFLOW;
  else {
    s[++t] = el;
    outc= OK;
```

```
  }
  return outc;
}
```

U vezi sa ovim pristupom treba podvući dve stvari, jednu prednost i jedan nedostatak:

- U slučaju da dođe do otkaza, odluka o akciji koju treba preduzeti *prepušta se klijentu*. Zadatak funkcije je samo taj da izvesti o otkazu, uz eventualnu informaciju o njegovom uzroku.
- Osnovni - ne baš mali! - nedostatak metode jeste nemogućnost uobičajene realizacije funkcija koje vraćaju vrednost, a uzrok leži u tome što je osnovni izlaz sada rezervisan za kod ishoda obrade. Najbolja ilustracija je baš metoda *top* koja po svojoj prirodi pripada upravo funkcijama koje generišu izlaz (nije *void*). Nju smo morali da modifikujemo tako da se osnovni i najvažniji rezultat, vrednost vrha steka, više ne predaje kao izlaz funkcije, nego kao njen novouvedeni izlazni argument *val*. To ima presudnog uticaja na način korišćenja funkcije jer se ona više ne može uključivati u izraze nego se mora pozivati posebno, uz dodatni zahtev da klijent poseduje promenljivu koja prihvata vrednost *val*. Funkcija se sada aktivira ovako:

  ```
  Stack s; int x;
  . . . . .
  if(s.top(x)!=OK)   {Postupak u slucaju otkaza}
  else  {normalan nastavak; x sadrzi vrednost vrha steka}
  ```

  Nedostatak ponekad može da se ublaži. To su slučajevi kada među izlaznim vrednostima postoje neke (obično jedna) koje same po sebi signalizuju otkaz. Recimo, ako funkcija vraća pokazivač takva vrednost bila bi 0 odn. NULL (primer je funkcija je *fopen*). U ovim - ipak specijalnim - uslovima nije neophodno uključiti osnovni izlaz među argumente jer pomenuta specijalna vrednost služi kao detekcija neuspeha. S druge strane, postupak je potpuno neprimenljiv na operatorske funkcije. Inače, korišćenje izlaza funkcije kao detektora ishoda dovoljno je velik nedostatak da se ovaj metod koristi samo tamo gde ne postoji poseban mehanizam za rukovanje izuzecima.

10.2.2. Rukovanje izuzecima u C^{++}

Treća metoda - korišćenje mehanizma rukovanja izuzecima najbolja je, jer eliminiše potrebu rezervisanja izlaza za indikaciju izuzetka uz istovremeno prepuštanje reakcije klijentu. Pored toga, po potrebi omogućuje i prekid programa. Glavni nedostaci ove metode jesu: (1) nema je svaki programski jezik (opšti nedostatak koji treba, u stvari, pripisati programskom jeziku) i (2) rukovanje izuzecima u C^{++} razlikuje se od sistema do sistema, i pored postojanja standarda.

Ovaj drugi nedostatak nalaže da se dobro prouči dokumentacija razvojnog sistema i utvrdi način rukovanja izuzecima.

U uvodnom delu ovog odeljka naveli smo dva principa na kojima počiva mehanizam rukovanja izuzecima. Prvi princip diktira dvodelnu arhitekturu podrške. Jedan deo nalazi se u rizičnoj rutini, tj. u funkciji u kojoj se izaziva incident (to bi bile metode *top*, *pop* i *push* u našem primeru). Svrha mu je da, kada se steknu uslovi, generiše poruku o izuzetku ili, kako se to kaže, da prijavi izuzetak klijentu *pre* nego što dođe do otkaza. Drugi deo pripada klijentu i ima zadatak da prihvati prijavu izuzetka, te da pokrene odgovarajuću akciju.

Drugi princip utiče na neposrednu realizaciju kanala kojim se predaje i prima prijava o izuzetku. Naime, izuzetak se prijavljuje prosleđivanjem bilo vrednosti bilo posebno kreiranog objekta. U oba slučaja radi se *de facto* o izlazima iz funkcije, ali izlazima koji se suštinski razlikuju od regularnih - onih radi kojih funkcija i postoji. Zato se poruke o nastanku izuzetaka ne smeju mešati sa regularnim rezultatima, tj. ne treba da budu realizovane kao izlaz ili parametri funkcije. Usled toga, mehanizam rukovanja izuzecima predviđa da se vrednosti odn. objekti što predstavljaju prijavu izuzetka u funkciji-predajniku pojavljuju kao dodatni, implicitni, "nevidljivi" izlazni argumenti kojih nema u njenom prototipu. Naravno, i deo za prijem prijave u klijentu, mora biti izveden tako da može da prihvati implicitne vrednosti i objekte. Klasični programski jezici nemaju rukovanje izuzecima iz prostog razloga što navedeni mehanizam može biti implementiran samo uvođenjem novih naredbi i konstrukata.

Prijavljivanje izuzetaka vrši se novom naredbom *throw* koja se pojavljuje u dve varijante. Prva je

throw *izraz*;

gde izračunavanje vrednosti izraza rezultuje kodom izuzetka. Kodovi izuzetaka u ovom slučaju pripadaju nekom od standardnih tipova ili, češće, imaju tip enumeracije. U praksi, *izraz* je najčešće sasvim jednostavan: predstavlja sam kod izuzetka i ništa više. Prijavu izuzetka naredbom *throw* demonstriraćemo na metodama klase *Stack*. Neka kodovi izuzetaka pripadaju enumeraciji

enum ErrorType {UNDERFLOW=1, OVERFLOW)[85];

Rizične metode klase sada poprimaju sledeći oblik (naredba *class* se ne menja):

// **Slučaj A: prijava izuzetaka preko izraza**

[85] Već smo rekli da se rukovanje izuzecima može neznatno razlikovati u različitim razvojnim sistemima. Sistem na kojem je testirano rukovanje rezerviše vrednost 0 za posebne slučajeve, te je dobro prvoj konstanti UNDERFLOW dodeliti vrednost 1 što će funkcionisati uvek.

```
int Stack::top() const
{
 if(top<0) throw UNDERFLOW;
 return s[top];
}
void Stack::pop()
{
 if(top<0) throw UNDERFLOW;
 top--;
}
void Stack::push(int el)
{
 if(top==CAPACITY-1) throw OVERFLOW;
 s[++top] = el;
}
```

U sve tri funkcije afirmativna vrednost uslova u naredbama *if* momentalno prekida njihovo izvršavanje i vraća kontrolu klijentu uz implicitnu predaju vrednosti UNDERFLOW (*top*, *pop*) odnosno OVERFLOW (*push*).

Glavne karakteristike prijavljivanja vrednosti kao koda izuzetka jesu (1) poruka je pasivna, tj. predstavlja samo informaciju da se izuzetak desio, sa podatkom o uzroku (2) mehanizam je primenljiv i u procedurnom C-u.

Za slučaj da se klijentu može sugerisati (ne i nametnuti!) akcija koju može da preduzme, izuzetak se prijavljuje preko objekta[86]. Objekat pripada grupi koju smo nazvali bezimenim objektima što, u ovom slučaju, znači da se konstruiše u sklopu naredbe *throw*. Opšti oblik druge varijante naredbe *throw*, kada se šalje objekat, je

throw *poziv_konstruktora*;

gde se konstruktor poziva imenom klase kojoj pripadaju objekti-poruke i eventualnim argumentima. U slučaju aktiviranja konstruktora bez argumenata, a zbog potencijalnih nesporazuma, sintaksa poziva konstruktora neznatno je izmenjena u

naziv_klase()

tj. iza naziva klase navodi se par zagrada. I ovaj način prijavljivanja izuzetka demonstriraćemo na primeru steka. U tu svrhu formiraćemo dve klase, *Ex_Underflow*

[86] Još jedan razlog za postojanje i vrednosti i objekata za rukovanje izuzecima jeste taj što su vrednosti primenljive i u procedurnom delu, tj. u C-u.

i *Ex_Overflow*, koje generišu objekte što odgovaraju porukama UNDERFLOW i OVERFLOW. Da bi upotreba objekata imala opravdanje, u obe klase uključićemo metode koje treba aktivirati u klijentu kada se primi odgovarajuća prijava izuzetka. Metode će imati isti prototip *void action()*. Konkretan oblik tela metoda nije od interesa, te ga nećemo posebno navoditi.

```
// Slučaj B: posebne klase za svaki izuzetak
class Ex_Underflow {
public:
  void action() {...}
};
class Ex_Underflow {
public:
  void action() {...}
};
```

Sada metode *top*, *pop* i *push* dobijaju sledeći oblik:

```
int Stack::top() const {
  if(t<0) throw Ex_Underflow();    // zapaziti poseban nacin poziva konstruktora
  return s[t];
}

void Stack::pop() {
  if(t<0) throw Ex_Underflow();
  t--;
}

void Stack::push(int el) {
  if(t==CAPACITY-1) throw Ex_Overflow();
  s[++t] = el;
}
```

Mogućnosti ima još. Umesto dve klase izuzetaka možemo formirati jednu, snabdevenu podatkom-članom što označava vrstu izuzetka, sa konstruktorom koji u taj podatak-član upisuje kod izuzetka.

```
// Slučaj C: zajednicka klasa za srodne izuzetke
//sa identifikatorima vrste izuzetka
enum ErrorType {UNDERFLOW=1, OVERFLOW);
```

```
class StackException {
private:
  ErrorType exeptioncode;
public:
  StackException(ErrorType code) {exceptioncode= code}
  ErrorType getExCode() {return exeptioncode;)
  void action() {...}
};
. . . . . .
int Stack::top() const {
  if(top<0) throw StackException(UNDERFLOW);
  return s[t];
}

void Stack::pop() {
  if(t<0) throw StackException(UNDERFLOW);
  t--;
}

void Stack::push(int el) {
  if(t==CAPACITY-1) throw StackException(OVERFLOW);
  s[++t] = el;
}
```

Poslednji među tipičnim primerima organizacije mehanizma predaje poruke o izuzetku bazira sa na primeni nasleđivanja. Naime, način prijema prijave izuzetka tako je koncipiran da rukovalac može da prihvati i pravilno interpretira kako dati objekat tako i objekte iz klasa-naslednica njegove klase (videti dalje). U našem primeru formiraćemo apstraktnu klasu *StackExceptions* koja sadrži virtuelnu apstraktnu metodu *action* i dve klase naslednice sa konkretnim oblicima metode *action*, jedna za stanje *underflow* i druga za stanje *overflow*.

```
// Slučaj D: primena nasledjivanja
class StackExceptions {
public:
  virtual void Action() = 0;
};
class StackUnderflow: public StackExceptions {
public:
  void action() {...}
```

```
};

class StackOverflow: public StackExceptions {
public:
  void action() {...}
};
. . . . . .
int Stack::top() const {
  if(t<0) throw StackUnderflow();
  return s[t];
}

void Stack::pop() {
  if(t<0) throw StackUnderflow();
  t--;
}

void Stack::push(int el) {
  if(t==CAPACITY-1) throw StackOverflow();
  s[++t] = el;
}
```

U dosadašnjim primerima sve varijante metoda mogle su da prijave sve izuzetke. Ako želimo da u nekoj funkciji ograničimo vrste izuzetaka koji mogu biti prijavljeni koristićemo *filter* - konstrukt koji propisuje tipove-klase izuzetaka što ih data funkcija može prijaviti. Filter je oblika

$$\text{throw}(K_1, ..., K_n)$$

gde su $K_1, ..., K_n$ identifikatori klasa i-ili tipova izuzetaka koje data funkcija sme prijaviti. Filter se navodi u sklopu zaglavlja funkcije odmah iza liste formalnih parametara. Funkcija

```
void f(...) throw(int, Ex_Class, char *) {
. . . . . .
}
```

može prijavljivati samo izuzetke predstavljene celobrojnim vrednostima, objektima klase *Ex_Class* ili stringovima. Ako preradimo poslednju varijantu metodu *push* tako da prototip dobije oblik

```
void Stack::push(int el) throw(StackOverflow);
```

ona neće moći da prijavi nijedan drugi izuzetak osim objekta klase *StackOverflow*.

Prihvatanje i obrada izuzetaka vrši se, kao i prijavljivanje, posebnom naredbom uvedenom takođe zbog potrebe da se kod izuzetka prenese implicitno. Naredba nosi naziv *try* i ima sledeći opšti oblik (sa mnoštvom varijanata):

```
try {
  naredbe
}
catch(kvaziparametar1) {
  naredbe
}
catch(kvaziparametar2) {
  naredbe
}
. . . . . .
catch(kvaziparametarn) {
  naredbe
}
```

Naredba se sastoji od dva jasno razdvojena dela: prvi se nalazi između službene reči *try* i prve pojave reči *catch*, dok se drugi sastoji od niza segmenata *catch*.

U bloku vezanom uz *try* nalaze se naredbe programskog jezika C++, među kojima bar jedna može da izazove prijavljivanje izuzetka (u suprotnom cela stvar gubi smisao). Unutar ovog bloka mogu se nalaziti i druge naredbe *try* koje se ponašaju kao uklopljene. Svrha ovog dela je da prihvati prijavu izuzetka i da je prosledi odgovarajućoj rutini koja će tu prijavu obraditi.

U drugom delu naredbe nalaze se rutine za prihvatanje i obradu izuzetaka koje nose naziv *rukovaoci izuzecima* (engl. "exception handlers"). Rukovaoci izuzecima podsećaju na *void* funkcije sa jednim argumentom, ali se od njih razlikuju po tome što nisu samostalni i što jedan jedini parametar koji imaju može imati oblik drukčiji od pravih parametara funkcije (zato su i nazvani kvaziparametrima). Posmatrajmo opšti oblik jednog rukovaoca izuzecima:

```
catch(kvaziparametar) {
  naredbe
}
```

Pre svega, rukovalac izuzecima *prepoznaje tip-klasu* izuzetka prosleđenog iz prvog

dela naredbe *try*. Drugim rečima, rukovaoci izuzecima u jednoj naredbi *try* ponašaju se otprilike kao preklopljene funkcije (svaka sa imenom *catch*) gde se korektna verzija bira na osnovu tipa-klase izuzetka prosleđenog iz prvog dela i poštujući *redosled* kojim su rukovaoci navedeni. Kvaziparametar može da ima tri oblika:

1. catch(*tip*) gde *tip* predstavlja naziv tipa ili klase izuzetaka. Kada se u prvom delu pojavi izuzetak opisan tipom *tip* obrada se prekida i prelazi se na ovaj rukovalac. Na primer, rukovalac *catch(int)* "uhvatiće" sve izuzetke kodirane celobrojnim vrednostima.
2. catch(*tip par*) gde *tip* ima istu interpretaciju, a *par* predstavlja pravi parametar; ovaj oblik ponaša se kao i prethodni, ali uz mogućnost da se vrednost parametra iskoristi u bloku sa naredbama rukovaoca. Na primer, *catch(ExceptionClass ex)* prihvata objekte iz klase *ExceptionClass* i pod imenom *ex* koristi ih u svom bloku sa naredbama.
3. catch(...) gde je troznak ... deo sintakse. Rukovaoci ovog oblika nose naziv *univerzalni rukovaoci* jer prihvataju sve vrste izuzetaka bez obzira na tip-klasu. Ako u istoj naredbi *try* postoji više rukovalaca uključujući i univerzalni, tada se univerzalni rukovalac navodi poslednji. Razlog je to što se prilikom traženja adekvatnog rukovaoca poštuje redosled njihovog navođenja, te univerzalni rukovalac pokriva sve koji bi se našli iza njega, tako da oni nikada ne bi bili aktivirani.

Cela naredba *try* ***izvršava se*** tako što se izvršavaju naredbe sadržane u bloku uz *try*. Ako pritom ne dođe do izuzetka program se nastavlja naredbom iza *try*. U suprotnom, obrada bloka se prekida i na osnovu vrste izuzetka traži odgovarajući rukovalac, redosledom kojim su rukovaoci navedeni. Ako se nađe, rukovalac se izvršava i prelazi se na naredbu iza naredbe *try*. Ako se rukovalac ne nađe i ako nema univerzalnog rukovaoca, program se prekida pozivom funkcije *abort*.

Način prenosa vrednosti odnosno objekta iz prvog dela naredbe *try* u locirani rukovalac podseća na prenos argumenata kod običnih funkcija. Razmotrićemo način prenosa na primeru objekta jer je ilustrativniji. Neka je *void f()* funkcija koja naredbom *throw* može da prijavi izuzetak i to tako što formira bezimeni objekat klase *ExceptionClass*:

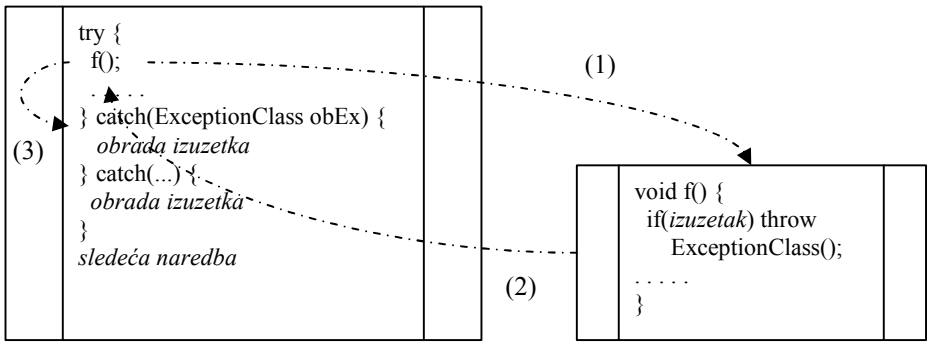
```
class ExceptionClass {
. . . . . .
};
void f(void) {
  if(uslov_za_izuzetak) throw ExceptionClass();
. . . . . .
}
```

Neka *u klijentu* funkcije *f* postoji naredba *try* oblika

```
try {
  f();
. . . . . .
}
catch(ExceptionClass obEx) {
  obrada izuzetka iz ExceptionClass uz korišćenje objekta obEx
}
catch(...) {
  obrada svih ostalih izuzetaka (ako ih ima)
}
```

Ukoliko tokom izvršavanja *f* dođe do izuzetka, funkcija će kreirati bezimeni objekat pozivom konstruktora *ExceptionClass()* i proslediti ga klijentu. Pošto je rad funkcije *f* prekinut u delu *try*, naredbe koje slede poziv *f* ignorišu se i prelazi se na rukovaoce izuzecima. Kako među njima postoji odgovarajući rukovalac (prvi po redu) on se aktivira, pri čemu se rutini prosleđuje *kopija* objekta ExceptionClass() jer je parametar *catch* predviđen za prenos po vrednosti. Po završetku obrade rukovaoca program se nastavlja naredbom koja sledi naredbu *try*, naravno ako rukovalac ne prekine rad pozivom *abort* ili *exit*. Ako je objekat koji se prenosi u rukovalac velik, kao i u drugim slučajevima prenos se može obaviti po referenci, tj. u rukovalac se može preneti adresa objekta ExceptionClass(). Da bi se to postiglo, argument *obEx* treba definisati kao referencu:

```
catch(ExceptionClass &obEx) {
  obrada izuzetka iz ExceptionClass uz korišćenje objekta obEx
}
```



Slika 10.1

Prijavljivanje i prijem izuzetka prikazani su na slici 10.1. Kako vidimo, mehanizam

throw - *try* za obradu izuzetaka u C^{++} pruža širok spektar mogućnosti primene. Bez namere da ih prodiskutujemo sve, izdvojićemo karakteristične slučajeve obrade izuzetaka, vezane za navedene primere prijavljivanja izuzetaka (slučajevi A, B, C i D navedeni gore).

Kada se izuzeci prijavljuju kao izrazi (tj. vrednosti) najpogodnije je grupisati ih u tipove srodnih. Za slučaj A logično je da se formira jedan tip koji sadrži sve izuzetke vezane za rad sa stekom, kao što je uostalom već i urađeno enumeracijom *ErrorType*. Pošto rukovalac izuzecima *catch* raspoznaje samo tipove moramo napraviti jedan jedini rukovalac koji prima tip *ErrorType*, a na osnovu prosleđene vrednosti naredbom *switch* bira akciju. Jednostavnosti radi, sve akcije predstavićemo slobodnom funkcijom *poruka(const char*)* koja argument - string prikazuje na ekranu kao poruku. Evo primera prihvatanja izuzetaka za slučaj A:

```
try  {
  naredbe koje koriste stek
 }
catch(ErrorType err) {
   switch(err) {
     case UNDERFLOW:
       poruka("*** STACK UNDERFLOW");
       break;
     case OVERFLOW:
       poruka("*** STACK OVERFLOW");
       break;
   }
 }
catch(...) {
 poruka("*** UNKNOWN EXCEPTION");
}
```

Ukoliko se u toku obrade bloka naredbe *try* pojavi neki izuzetak, kontrola se momentalno prenosi na blok rukovalaca. Ako je kod izuzetka tipa *ErrorType* aktivira se prvi rukovalac koji na osnovu vrednosti argumenta *err* (vrednost je UNDERFLOW ili OVERFLOW) bira poruku što će biti poslata. Ako se pojavi neki drugi izuzetak poziva se univerzalni rukovalac koji će generisati poruku o nepoznatom izuzetku. Još jednom napominjemo da redosled navođenja rukovalaca nije proizvoljan jer se koristi za njihov izbor. Ako bismo promenili redosled navođenja gornja dva rukovaoca tada bi reakcija na svaki izuzetak bila poruka *"*** UNKNOWN EXCEPTION"* jer univerzalni rukovalac prihvata sve izuzetke.

Sledeći primer jeste realizacija rukovalaca izuzecima koji odgovaraju slučaju B. Podsetimo se, u ovom slučaju izuzeci pripadaju različitim klasama

Ex_Underflow i *Ex_Overflow*. Pošto se objekti koji se šalju kao prijave nastanka izuzetaka razlikuju po klasi, za svaku klasu izrađuje se poseban rukovalac.

```
try  {
  naredbe koje koriste stek
  }
catch(Ex_Underflow &under) {
   under.action();
}
catch(Ex_Overflow &over) {
   over.action();
}
```

U zavisnosti od toga kojoj od dve predviđene vrste pripada "uhvaćeni" izuzetak, aktivira se odgovarajući rukovalac. U svakom od njih izvršava se akcija koja je takođe deo primljenog objekta u čemu i jeste suština korišćenja objekata za prijavljivanje izuzetaka. Zapazimo primenu reference kao tipa parametra rukovaoca. Univerzalni rukovalac namerno nije naveden da bismo pokazali da on nije neophodan. Ako bi se u ovom primeru iz ma kojih razloga pojavio nepredviđeni izuzetak, program bi se zaustavio kao rezultat tzv. propagacije izuzetaka koja će biti objašnjena u nastavku.

Sledeći primer je varijanta prethodnog koja podrazumeva primenu jednog rukovaoca za sve izuzetke vezane za stek (slučaj C). Da bi rukovalac mogao da prepozna izuzetak moramo preraditi klase izuzetaka tako što ih prvo spajamo u jednu, zatim snabdevamo identifikatorom izuzetka. Konačno, metoda *action* treba da sadrži skretnicu *switch* koja će na osnovu identifikatora izuzetka odabrati vrstu reakcije na izuzetak. Segment za obradu izuzetaka imao bi u tom slučaju sledeći oblik:

```
try  {
  naredbe koje koriste stek
  }
catch(StackException &ex) {
   ex.action();
   // Umesto gornje naredbe moze se metodom getCode procitati
   // kod izuzetka i u skladu s tim programirati reakcija na izuzetak
}
```

Nešto elegantnije rešenje je primena nasleđivanja (slučaj D). Rad rukovalaca izuzecima baziran je na inkluzionom polimorfizmu, jer rukovalac sa parametrom klase pretka može da prihvati kao stvarne oblike instance svih potomaka. Ovde

treba voditi računa o tome da ako među rukovaocima postoje posebni primerci za predak i potomke tada rukovaoci za potomke moraju biti navedeni pre rukovaoca za predak da ovaj poslednji ne bi prepoznao potomak i aktivirao se. U našem primeru toga nema jer je predak apstraktna klasa.

```
try  {
  naredbe koje koriste stek
  }
catch(StackExceptions &ex) {
   ex.action();
}
```

Pošto je metoda *action* virtuelna, sigurni smo da će u zavisnosti od toga kojem od potomaka *StackUnderflow* ili *StackOverflow* pripada objekat *ex* biti aktivirana korektna verzija.

Na kraju, ostaje da se razmotri još i pitanje tzv. ***propagacije izuzetaka***. Naime, poziv neke funkcije koja generiše izuzetak može da bude (i često jeste) samo poslednji korak u lancu sukcesivnih poziva funkcija (slobodnih ili članica). Neka je f_n funkcija koja generiše izuzetak *e* tipa ili klase *Exception*. Neka je u nekom trenutku ona bila aktivirana od strane funkcije f_{n-1}, koja je, sa svoje strane, aktivirana u funkciji f_{n-2} itd. sve do neke funkcije f_1, slika 10.2. Svaka od funkcija u lancu poziva, recimo f_k, može da reaguje na pristizanje izuzetka iz pozvane funkcije (npr. f_{k+1}) na tri načina:

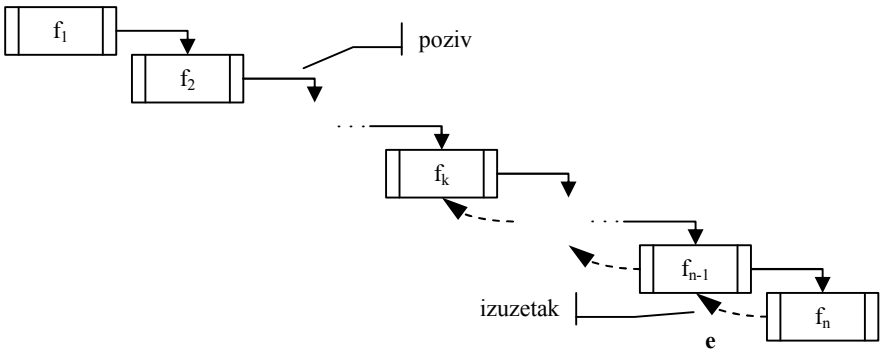
1. Može da prosledi izuzetak prethodnoj u lancu poziva (tj. f_{k-1}) bez ikakve reakcije sa svoje strane. Ovo se postiže tako što se u funkciju f_k ne ugrađuje rukovalac *catch* koji može da obradi izuzetak *e*. Inače, ako ni jedna od funkcija koje perthode f_k nema rukovalac za *e* izuzetak će biti prosleđivan sukcesivno sve dok ne stigne do prve funkcije f_1 (a to je *main*) i program će biti prekinut.
2. Funkcija f_k može da prihvati i obradi izuzetak. Da bi se to ostvarilo ona mora posedovati naredbu *try* sa rukovaocem *catch* koji prepoznaje klasu ili tip izuzetka *e*. Struktura rukovaoca ima opšti oblik

   ```
   catch(T ex) {
     obrada izuzetka
   }
   ```

   gde je *T*≡*Exception* ako je *Exception* tip, odnosno *T* je klasa *Exception* ili njena natklasa ako je *e* objekat. Naravno, rukovalac može biti i univerzalan sa zaglavljem *catch(...)*. U ovom slučaju propagacija *e* zaustavlja se.
3. Funkcija f_k može da prihvati izuzetak, obradi ga i prosledi dalje funkciji f_{k-1}, što na neki način predstavlja kombinaciju prethodnih postupaka. Struktura rukovaoca je sada

```
catch(T ex) {
  obrada izuzetka
  throw ex;
}
```

Lako se uočava da je sistem prenosa i obrade izuzetaka prilagođen principu "izuzetak mora biti detektovan".



Slika 10.2

10.2.3. Primer: klasa String

Programski jezik C/C++ ne raspolaže posebnim tipom stringa. Za realizaciju ovog tipa koristi se znakovni niz koji se razlikuje od ostalih nizova po postojanju string-konstanata. Rešenje je najverovatnije preuzeto iz originalnog paskala gde se susrećemo sa identičnom situacijom. U međuvremenu zapaženo je da je string poseban, i to važan tip te je odgovarajuće rešenje uneto u paskal. Programski jezik C nije u tom smislu pretrpeo nikakve izmene. U okviru ovog primera napravićemo klasu *String* koja služi za realizaciju tipa stringa. Okvirna specifikacija zahteva bila bi sledeća:

* Maksimalna dužina stringa zadaje se tek pri instanciranju, što znači da će klasa biti parametrizovana maksimalnom dužinom stringa, u oznaci C.
* Klasa treba da bude operatorski orijentisana.
* Klasa sadrži standardne string-operacije: dodelu =, određivanje dužine (funkcija *length*), pristup indeksiranjem [], proveru jednakosti == odnosno nejednakosti != dva stringa, konkatenaciju (spajanje) + i nadodavanje += stringa na dati.
* Operacije klase treba da budu kompatibilne sa string-konstantama.
* Sve operacije koje mogu da izazovu otkaz treba da budu zaštićene. Takve operacije su indeksiranje indeksom manjim od 0 ili većim od aktuelne (ne maksimalne!) dužine stringa, kao i spajanje odn. nadodavanje stringova sa rezultatom dužim od maksimalne dužine stringa. Vidimo da se pojavljuju dve vrste

izuzetaka: indeksiranje izvan opsega (kod izuzetka biće STR_RANGE) i prekoračenje memorijskog prostora dodeljenog stringu (izuzetak STR_OVERFLOW).

* Pošto rukovanje izuzecima zahteva vreme, treba da postoji mehanizam za uključivanje odnosno isključivanje zaštite. Ovo je realizovano simboličkom konstantom STR_EXCEPTIONS_ON koju treba definisati direktivom #define ako želimo zaštitu. U suprotnom zaštita se ne uključuje.

Klasa *String* nalazi se u modulu MYSTRING.HPP i ima sledeći oblik:

```
/*****************************************************************
 ZAGLAVLJE SA KLASOM GENERICKOG STRINGA SA IZUZECIMA

Naziv datoteke: MYSTRING.HPP
*****************************************************************/
#include <string.h>

enum StrError {STR_RANGE=1,STR_OVERFLOW};

template <int C>
class String {
private:
  int n;
  char s[C];
public:
  String(const char str[]="");
  ~String(){n= 0;}
  char operator [](int i) const;
  char& operator [](int i);
  String<C>& operator =(const String<C>&);
  template<int U> friend int operator ==(const String<U>&,const String<U>&);
  template<int U> friend int operator !=(const String<U>&,const String<U>&);
   template<int U>
     friend String<U> operator +(const String<U>&,const String<U>&);
  String<C>& String<C>::operator +=(const String<C>&);
  template<int U> friend int length(const String<U>&);
};

template <int C>
String<C>::String(const char str[]) {
#ifdef STR_EXCEPTIONS_ON
  if(strlen(str)>C) throw STR_OVERFLOW;
```

```
#endif
  n= 0;
  while(str[n]!='\0') {s[n]= str[n]; n++;}
}

template <int C>
char String<C>::operator [](int i) const {
#ifdef STR_EXCEPTIONS_ON
  if((i<0)||(i>=n)) throw STR_RANGE;
#endif
  return s[i];
}

template <int C>
char& String<C>::operator [](int i) {
#ifdef STR_EXCEPTIONS_ON
  if((i<0)||(i>=n)) throw STR_RANGE;
#endif
  return s[i];
}

template <int C>
String<C>& String<C>::operator =(const String<C>& rhs)
{
  for(n=0;n<rhs.n;n++) s[n]= rhs.s[n];
  return *this;
}

template <int U>
int operator ==(const String<U>& s1,const String<U>& s2) {
 if(s1.n!=s2.n) return 0;
 for(int i=0;i<s1.n;i++) if(s1[i]!=s2.s[i]) return 0;
 return 1;
}

template <int U>
int operator !=(const String<U>& s1,const String<U>& s2) {
 if(s1.n!=s2.n) return 1;
 for(int i=0;i<s1.n;i++) if(s1[i]!=s2.s[i]) return 1;
 return 0;
}
```

```
template <int U>
String<U> operator +(const String<U>& s1,const String<U>& s2) {
#ifdef STR_EXCEPTIONS_ON
  if(s1.n+s2.n>=U) throw STR_OVERFLOW;
#endif
  String<U> result;
  result= s1;
  for(int i=0; i<s2.n; i++) result.s[result.n++]= s2.s[i];
  return result;
}

template <int C>
String<C>& String<C>::operator +=(const String<C>& rhs) {
#ifdef STR_EXCEPTIONS_ON
  if(n+rhs.n>=C) throw STR_OVERFLOW;
#endif
  for(int i=0;i<rhs.n;i++) s[n++]= rhs.s[i];
  return *this;
}

template <int U>
int length(const String<U>& str) {
  return str.n;
}
```

Prvo što treba uočiti jeste da se operatorske metode ne razlikuju od običnih, što znači da mogu prijavljivati izuzetke na identičan način. Druga stvar koju uočavamo je način na koji se realizuju generičke operatorske funkcije *== != +*, kao i funkcija *length* koje su sve realizovane kao prijateljske. Pošto su slobodne, one imaju sopstveni šablon, sa posebnim generičkim parametrom *U*[87] koji se ne sme poklapati sa generičkim parametrom klase *C*.

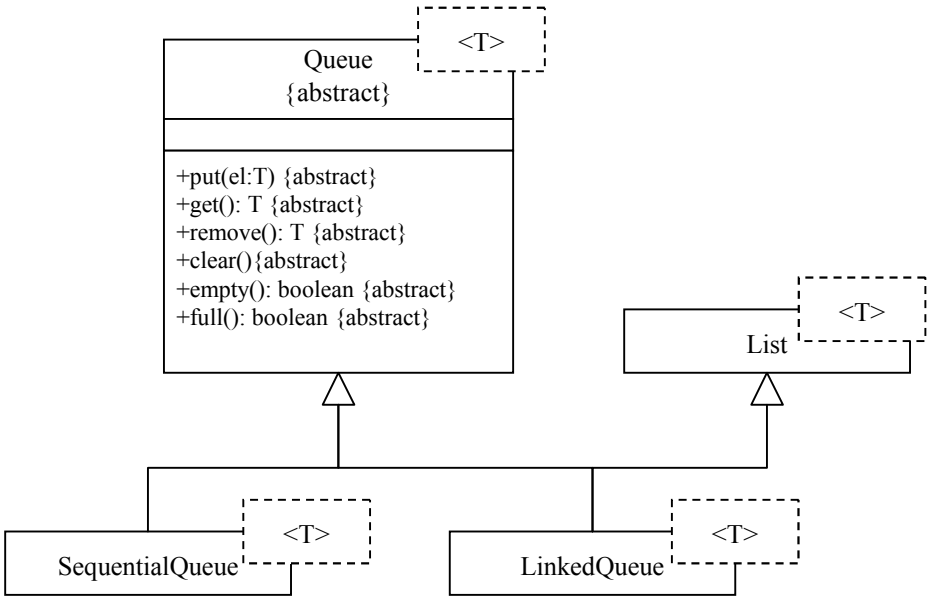
Na kraju, mehanizam za (ne)uključivanje zaštite primenjen u ovom primeru može se proširiti na bilo kakve, a ne samo generičke klase. Pri tom treba voditi računa o tome da direktiva *#ifdef* deluje samo na zaglavlje. Dakle, u slučaju da modul sa klasom ima i telo, sve funkcije sa naredbom *throw* moraju biti u zaglavlju.

[87] Naravno, svaka od njih mogla je imati sopstveni, posebno označen, generički parametar U, V, W itd., ali za primenu toga nije bilo razloga.

Opšta (ne i jedina moguća) šema upravljanja izuzecima klase *String* u klijentu izgleda ovako:

```
#define STR_EXCEPTIONS_ON
#include "mystring.hpp"
. . . . . .
try {
  naredbe koje koriste klasu String
}
catch(ErrorType er) {
  switch(er) {
  case STR_RANGE: /* postupak u slucaju prekoracenja opsega indeksa */
  case STR_OVERFLOW: /* postupak u slucaju prekoracenja memorijskog prostora */
}
```

10.2.4. Primer: generički red sa izuzecima



Slika 10.3

U odeljku 8.5.2. razradili smo malu hijerarhiju sa dve vrste redova: spregnutim i sekvencijalnim. Pritom, elementi redova bili su generički pokazivači. U ovom primeru realizovaćemo istu hijerarhiju, ali sa generičkim redovima čije su

kritične funkcije zaštićene mehanizmom za generisanje izuzetaka. Hijerarhija je prikazana na slici 10.3. Dijagram klasa potpuno odgovara onom sa slike 8.16, osim činjenice da su elementi reda tipa *T* koji je generički parametar. Na ovom primeru ilustrovaćemo apstraktne generičke klase (klasa *Queue*) i višestruko nasleđivanje generičkih klasa (klasa *LinkedQueue* koja nasleđuje *Queue* i *List*, a sve su generičke), kao i još jednom ukazati na novine koje su u generičke klase unete u novoj, standardnoj verziji C^{++}. Uočimo i to da je spregnuta verzija reda izvedena iz generičke liste razrađene u primerima 9.3 i 9.4.

```
/*************************************************************

                    ZAGLAVLJE BIBLIOTEKE
            SA APSTRAKTNIM GENERICKIM REDOM

Naziv datoteke: QUEUE.HPP
*************************************************************/
#ifndef QUEUE_HPP
  #define QUEUE_HPP

enum QError {UNDERFLOW=1, OVERFLOW};

template <class T>
class Queue {
public:
  virtual void put(T el) = 0;
  virtual T get() = 0;
  virtual T remove() = 0;
  virtual void clear() = 0;
  virtual int empty() const = 0;
  virtual int full() const = 0;
};

#endif
```

```
/*************************************************************

                    ZAGLAVLJE BIBLIOTEKE
            SA SPREGNUTIM GENERICKIM REDOM

Naziv datoteke: LINKEDQUEUE.HPP
*************************************************************/
```

```
#ifndef LINKEDQUEUE_HPP
  #define LINKEDQUEUE_HPP
#include "queue.hpp"
#include "genlist.hpp"

template <class T>
class LinkedQueue: public Queue<T>, protected List<T> {
public:
  virtual ~LinkedQueue(){};
  void put(T el);
  T get();
  T remove();
  void clear();
  int empty() const {return List<T>::empty();}
  int full() const {return 0;}
};

template <class T>
void LinkedQueue<T>::put(T el) {
  List<T>::gotoLast();
  insertAfter(el);
}

template <class T>
T LinkedQueue<T>::get() {
  if(empty()) throw UNDERFLOW;
  T temp;
  List<T>::gotoFirst();
  read(temp);
  return temp;
}

template <class T>
T LinkedQueue<T>::remove() {
  if(empty()) throw UNDERFLOW;
  T temp;
  List<T>::gotoFirst();
  List<T>::read(temp);
  List<T>::remove();
  return temp;
}
```

```
template <class T>
void LinkedQueue<T>::clear() {
  List<T>::clear();
}

#endif
```

```
/*************************************************************

                    ZAGLAVLJE BIBLIOTEKE
              SA SEKVENCIJALNIM GENERICKIM REDOM

Naziv datoteke: SequentialQueue.HPP
*************************************************************/
#ifndef SEQUENTIALQUEUE_HPP
  #define SEQUENTIALQUEUE_HPP
#include "queue.hpp"

template <class T,int CAPACITY>
class SequentialQueue: public Queue<T> {
protected:
  int size, getPosition, putPosition;
  T elements[CAPACITY+1];
public:
  SequentialQueue() {
    size= CAPACITY+1;
    getPosition= putPosition= 0;
  }
  virtual ~SequentialQueue() {}
  void put(T el);
  T get();
  T remove();
  void clear();
  int empty() const;
  int full() const;
};

template <class T,int CAPACITY>
void SequentialQueue<T,CAPACITY>::put(T el) {
  if(full()) throw OVERFLOW;
```

```
  elements[putPosition]= el;
  putPosition= (putPosition+1)%size;
}

template <class T,int CAPACITY>
T SequentialQueue<T,CAPACITY>::get() {
  if(empty()) throw UNDERFLOW;
  return elements[getPosition];
}

template <class T,int CAPACITY>
T SequentialQueue<T,CAPACITY>::remove() {
  if(empty()) throw UNDERFLOW;
  T removed = elements[getPosition];
  getPosition= (getPosition+1)%size;
  return removed;
}

template <class T,int CAPACITY>
void SequentialQueue<T,CAPACITY>::clear() {
  getPosition= putPosition;
}

template <class T,int CAPACITY>
int SequentialQueue<T,CAPACITY>::empty() const {
  return getPosition==putPosition;
}

template <class T,int CAPACITY>
int SequentialQueue<T,CAPACITY>::full() const {
  return getPosition==(putPosition+1)%size;
}

#endif
```

10.3. PROJEKTOVANJE PO UGOVORU

U dosadašnjim razmatranjima utvrdili smo *šta* su otkazi i *kako* se mogu sprečiti. Ostaje uvek neugodno pitanje: *kada i u kojoj meri* vršiti prevenciju. Odgo-

vor na ovakva pitanja nikada nije jednoznačan te i ovom prilikom moramo poći od *pro et contra*. O prednosti zaštite ne treba posebno govoriti - sastoji se u povećanju pouzdanosti na mikronivou klijent-server. Osnovni nedostaci jesu pre svega utrošak vremena potrebnog za izvršavanje naredbi što spadaju u zaštitni mehanizam, a zatim i posložnjavanje koda kako klijenta tako i servera[88]. Prema tome, odluka o uključivanju prevencije donosi se na osnovu funkcije cilja: ako je vreme kritičan resurs tada se prevencija minimizuje, što pri izradi klijentskog softvera zahteva povećanu opreznost, tačnije duže trajanje kodiranja i posebno testiranja. S druge strane, interaktivni softverski sistem širokog dijapazona primene po pravilu zahteva obimne mere zaštite, stoga što se u ulozi klijenta pojavljuje korisnik. Najbolju potvrdu tačnosti ovakvog načina razmišljanja pružaju dva klasična procedurna jezika: C i paskal. Poznato je da osnovnu funkciju cilja programskog jezika C čini vreme. Da bi se ona ostvarila žrtvovana je čak i provera prekoračenja opsega kod indeksiranja nizova, tako da pokušaj pristupa elementu izvan memorijskog prostora niza rezultuje otkazom koji može ali ne mora izazvati incident. S druge strane, paskal čija je glavna karakteristika čitljivost i razumljivost tako nešto ne dozvoljava.

Pouzdanost obuhvata dve ortogonalne komponente - korektnost i robustnost. Dakle, da bi se jedan sistem klijent - server smatrao pouzdanim mora zadovoljiti uslove vezane kako za korektnost tako i za robustnost. Posmatrajmo neku rutinu (potprogram, metodu) *r* koja se pojavljuje u ulozi servera. Do otkaza u *r* može doći iz dva razloga: defektnog koda u *r* što, po definiciji, znači da nije zadovoljena specifikacija, tj. da rutina nije korektna. S druge strane, korektna rutina može otkazati zbog defekta u pozivu koji se nalazi u klijentu. Prema tome, odgovornost za nastanak otkaza je podeljena - "krivac" može biti klijent jednako kao i server.

Uzme li se u obzir višestrukost faktora koji utiču na pojavu otkaza (defekt u klijentu, defekt u serveru), kao i različite osnovne funkcije cilja (brzina, čitljivost), postaje jasno da se projektovanje sistema zaštite zasniva na inženjerskom odlučivanju, a na bazi određenih *procena*, kao što su:

- Procena uticaja zaštite na trajanje izvršenja.
- Procena verovatnoće greške sa idejom da egzotične greške uglavnom nisu kandidati za prevenciju otkaza.
- Procena klijen(a)ta. Ako se u ulozi klijenta pojavljuje softverski element koji realizuje programer verovatno nije isplativo ugrađivati zaštitu od grubih omaški. S druge strane, kada je klijent korisnik, iskustvo je pokazalo da "ako postoji ma i najmanja verovatnoća za neku nesuvislu akciju ona će, ranije ili kasnije, sigurno biti izvedena".
- Procena štete nastale zbog otkaza.

  Da bismo bili u stanju da procenimo obim prevencije otkaza, najbolje je

[88] Povećanje složenosti može da smanji pouzdanost na *makronivou*. Zato smo se u prethodnom stavu ograničili na mikronivo klijent-server.

prvo analizirati ekstremna rešenja, a to su slučajevi kada se ne ugrađuje nikakva zaštita, odnosno kada se vrši kompletna provera svih dubioznih situacija. Rutina će biti potpuno nezaštićena pre svega kada je pojava otkaza krajnje neverovatna[89], kao i kada se realizuje u procesu elementarne obuke. Potpuna zaštita, ako tako nešto uopšte postoji, kao pristup nije ništa bolje rešenje od prethodnog. Prvo, zbog redundantnih provera trajanje izvršenja se povećava, a zatim naredbe za proveru posložnjavaju kod. Zamislimo rutinu pisanu na paskalu ili C/C^{++} u kojoj se pri svakom celobrojnom sabiranju, oduzimanju ili množenju proverava da li je prekoračen opseg celobrojnog tipa! Vidimo, dakle, da ni jedno od razmatranih rešenja nema kvalifikaciju da postane *stil programiranja*, što vrlo jasno vodi ka - inače dobro poznatom - pravilu

- Odluka o nivou zaštite donosi se za svaku rutinu posebno.

U pokušaju do omogući, ili bar olakša, utvrđivanje nivoa zaštite od otkaza u datoj rutini, Mejer je razvio poseban pristup za tretiranje pouzdanosti, nazvan ***projektovanje po ugovoru*** [1, 82]. Polazi se od već ustanovljene činjenice da otkazi u rutini mogu nastati kao posledica defekta u kodu same rutine, ali i usled neadekvatnog poziva iz klijenta. Stoga nije logično da se prevencija prepusti u celosti bilo kojem od ova dva elementa. Naprotiv, odgovornost za sprečavanje otkaza treba da bude *podeljena*, te da jedan deo preuzme sama rutina, a drugi deo da pripadne klijentu. Pri tom, Mejer je uspeo da precizno odredi granicu između domena odgovornosti. Kao sredstvo za to služi model Hoarovih tripleta sa preduslovom i postuslovom kao ključnim pojmovima. Podsetimo se, Hoarov triplet je predikat {P} S {Q} gde su P i Q takođe predikati, a S sintaksna jedinica (u našem slučaju posmatrana rutina r). Tačnost {P} r {Q} znači: ako je pre izvršenja rutine *r* predikat P tačan, rutina će sigurno terminirati u stanju u kojem je predikat Q tačan.

U metodi projektovanja po ugovoru preduslov P i postuslov Q određuju se na osnovu utvrđenih domena odgovornosti klijen(a)ta i rutine. Projektovanje po ugovoru zasnovano je na nekoliko sasvim jednostavnih pravila:

1. Za tačnost preduslova P odgovoran je samo i isključivo klijent.
2. Tačnost postuslova Q u celosti obezbeđuje rutina - server.
3. Preduslov i postuslov biraju se tako da triplet {P} r {Q} bude tačan.
4. S obzirom na kvalitativnu razliku između softverskog klijenta i korisnika, metoda se primenjuje samo na odnos softver - softver.

Dakle, tačnost tripleta {P} r {Q} obezbeđuju klijent i server zajedno. Na osnovu tačaka 1 i 2 zaključujemo da u rutini *r* ne sme biti nikakve provere ispunjenosti preduslova P, niti u klijentu provere postuslova Q, čime ne samo da je povučena granica odgovornosti nego su eliminisane i sve redundantne provere.

Ostaje još pitanje oblika predikata P i Q. U tripletu {P} r {Q} predikati P i

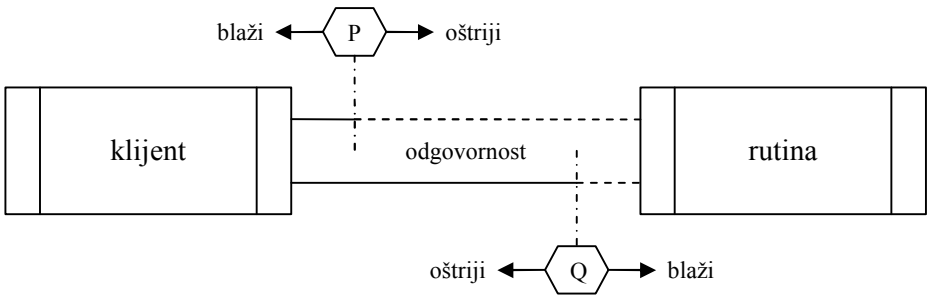
[89] Pažljivo smo izbegli formulaciju "ne može da dođe do otkaza" jer u programiranju takva tvrdnja nema opravdanja.

Q nemaju jedinstvenu formu, tj. triplet može biti tačan za razne oblike parova (P,Q). Neka je *f* funkcija koja sigurno računa vrednost ln x. Tada su sva sledeća tvrđenja tačna:

$$\{x>0\}\ f\ \{f=\ln x\}$$
$$\{x>1\}\ f\ \{f=\ln x\}$$
$$\{x=1\}\ f\ \{f=0\}$$

Između ovih i sličnih tvrđenja zanimaju nas, naravno, samo ona koja za postuslov imaju predikat f=ln x. Iako prva dva tripleta zadovoljavaju ovaj uslov oni očigledno nisu jednaki. Za preduslov x>1, kaže se da je oštriji od preduslova x>0 stoga što iz x>1 sigurno sledi x>0. Uopšte, za preduslov P_1 kažemo da je oštriji od preduslova P_2 ako važi $P_1 \Rightarrow P_2$ pri čemu P_1 i P_2 nisu ekvivalentni. Isto važi i za postuslove.

Mejerova metoda dobila je naziv po načinu na koji se utvrđuju oblici preduslova i postuslova. Naime, preduslov i postuslov utvrđuju se u sklopu svojevrsnog "ugovora" između klijen(a)ta i rutine-servera u kojem se klijent obavezuje da pri pozivu rutine obezbedi tačnost preduslova, dok se server sa svoje strane obavezuje da u tom slučaju obezbedi tačnost postuslova. Lako se zapaža da oštriji preduslov povećava odgovornost (i obaveze) klijenta oslobađajući rutinu od nekih provera, dok blaži preduslov teret provera prebacuje na server. Za postuslov važi obrnuto, slika 10.4.



Slika 10.4

Prilikom utvrđivanja oblika predikata koji čine preduslov i postuslov neophodno je voditi računa i o principu skrivanja informacija. Posmatrajmo primer osnovnog steka (onog bez zaštite) u prethodnom odeljku. U osnovnoj verziji ne predviđa se nikakva provera unutar metoda *top*, *pop* i *push* što nije greška nego prosto znači da je klijent odgovoran za stanje steka pre aktiviranja ovih metoda. Metoda *pop*, na primer, može da se aktivira samo ako stek nije prazan. Odgovarajući preduslov možemo izraziti na dva načina: (1) kao $empty=\perp$ ili (2) kao $t \geq 0$. S obzirom na to da

preduslov treba da ispuni klijent kojem je, prema principu skrivanja informacija, unutrašnjost klase nedostupna, sledi da je samo prvi način legalan, tj.

- prilikom formiranja preduslova smeju se koristiti samo članovi klase koji pripadaju njenom interfejsu.

Što se tiče postuslova ovo nije obavezno, a često ni moguće, jer se ispunjenjem postuslova bavi sama klasa. Kada to okolnosti dozvoljavaju poželjno je primeniti isto pravilo i na postuslov.

Vratimo se na primer funkcije *f* za računanje logaritma. Ako se kao preduslov "ugovori" predikat x>0 tada u rutini ne sme biti takve provere, a na klijentu je da ispuni navedeni uslov ili da snosi posledice nepoštovanja "ugovora". Ako se pak za ugovoreni preduslov prihvati predikat $P \equiv T$[90] tada rutina na sebe mora da preuzme odgovornost oko provere vrednosti x i, recimo, generiše izuzetak ako x nije veće od 0. Ni postuslov se ne podrazumeva sam po sebi. Zapazimo da u poslednjem slučaju ($P \equiv T$) za postuslov ne može da se proglasi $f=\ln(x)$[91], nego ($f=\ln(x)$) ili (f nije definisana). Isto tako, vrednost logaritma ne može da se računa drukčije nego numeričkim metodama koje proizvode približan rezultat. Strogo gledano, postuslov Q: $f=\ln(x)$ u opštem slučaju ne može da se ostvari zbog neizbežnih nepreciznosti proračuna. Realan postuslov mora sadržati i neku meru tačnosti rezultata, recimo gornju granicu apsolutne ili relativne numeričke greške. Dakle, realan postuslov mogao bi da bude recimo Q: ($f \approx \ln x$) *i* ($\Delta < 10^{-6}$) gde je Δ gornja granica apsolutne numeričke greške.

Videli smo da (ugovoreni) preduslov kao takav može imati različite oblike, strože ili manje stroge. U postupku izbora forme preduslova arhimedovsku tačku predstavlja zabrana provere preduslova unutar servera. Pokazaćemo na jednom primeru kako ova polazna tačka direktno utiče na oblik kako preduslova, tako i postuslova. Neka je server funkcija *invert(A,B,n)* koji invertuje kvadratnu matricu *A* reda *n* i rezultat vraća preko izlazne matrice *B*. Ako pođemo od semantike potprograma dobijamo "prirodan", "logičan" oblik odgovarajućeg Hoarovog tripleta:

$$\{\det(A) \neq 0\}\ \text{invert}(A,B,n)\ \{B=A^{-1}\}$$

Metoda projektovanja po ugovoru nalaže da je obezbeđivanje nesingularnosti matrice *A* zadatak koji pripada klijentu. U najgorem slučaju, kada se sadržaj matrice *A* ne može predvideti, klijent bi morao da *proveri* da li je *A* nesingularna. To, međutim, zahteva relativno obimne proračune pre poziva *invert*, s tim da se dobar deo tih proračuna, implicitno ili eksplicitno, *ponavlja* u funkciji, u procesu određivanja inverzne matrice. U ovom slučaju kao bolje rešenje nameće se prepuštanje provere nesingularnosti serveru *invert* koji će putem izuzetka obavestiti klijenta o nemo-

[90] Relacija $P \equiv T$ interpretira se sa "Predikat P u svim stanjima ima vrednost T".
[91] Može, ali tada nema garancije da je {P}f{Q} tačno!

gućnosti uspešnog završetka. No sada se javlja problem: ako se provera nesingularnosti vrši u serveru, tada ona ne može biti deo preduslova. Dakle, da bismo ovaj slučaj uskladili sa metodom projektovanja po ugovoru, formulu totalne korektnosti servera *invert* preinačujemo u

$$\{\tau\}\ \text{invert}(A,B,n)\ \{(B=A^{-1})\vee(\det(A)=0)\}$$

gde je τ predikat koji u svim stanjima ima vrednost T ($\tau\equiv T$). Interpretacija ove formule je "bez obzira na početno stanje funkcija *invert* se završava prevodeći sistem u stanje u kojem je $B=A^{-1}$ ili signalizira da je $\det(A)=0$" (ovo poslednje efektivno znači neuspeh). Ako uporedimo dva vida Hoarovog tripleta za funkciju *invert* uočavamo da oni nisu ekvivalentni: predikat $\det(A)\neq 0$ iz preduslova u prvom slučaju premešta se u postuslov u drugom i to kao sopstvena negacija, a sve usled dosledne primene Mejerovog metoda.

Na kraju jedna napomena u vezi sa tačnošću predikata {P} S {Q}. Poznato je (Hayden 1979.) da se korektnost proizvoljne, unapred zadate, sintaksne jedinice ne može dokazati. Takođe se dobro zna da ni u praksi uglavnom nema garancije da je realna sintaksna jedinica potpuno pouzdana. Sledstveno tome, u matematičkom smislu neformalan oblik ovog pravila bio bi da se P i Q biraju tako da tvrdnja {P} r {Q} bude *što tačnija*. Formalno, to bi značilo zamenu predikatskog računa nekim od modela kontinualne logike (jedan je opisan u [91]) ili prelazak na fazi logiku.

DODATAK

U ovom dodatku razmotrićemo još neka sredstva programskog jezika C^{++}, koja nisu deo objektne metodologije kao takva, ali su specifična za programski jezik C^{++}.

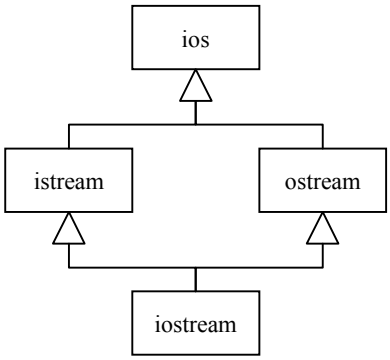
D1. ELEMENTARNI ULAZ-IZLAZ U C^{++}

Nijedan imperativni programski jezik ne može se zamisliti bez mogućnosti razmene podataka sa okolinom, bilo putem konzole bilo posredstvom datotečnog podsistema. Programski jezik C/C^{++} ima čak dva ulazno-izlazna podsistema: jedan je onaj koji je pripada njegovoj procedurnoj komponenti (tj. C-u), dok je drugi inkorporiran u objektni deo kao posebna hijerarhija klasa namenjenih elementarnom ulazu-izlazu, kao i korišćenju datoteka. Dakle, ulaz-izlaz u C/C^{++} realizovan je kao skup funkcija odnosno hijerarhija gotovih klasa i kao takav nije integralni deo jezika.

Ulanzo-izlazni podsistem C^{++} tretira datoteke i konzolu poput programskog jezika C, kao niz bajtova čija je interpretacija prepuštena programu. Stoga se i u C^{++} govori o ***tokovima*** (engl. "stream") koji se ne vezuju samo za periferijske memorije nego i za konzolu i to u načelu na isti način. Tokovi u objektnom delu C^{++} zasnovani su na klasama koje se izvode iz dve glavne, opisane u zaglavlju <*iostream.h*>. To su klase

- *istream* za unos podataka u program i
- *ostream* za iznos podataka iz programa.

Klase *istream* i *ostream* izvode se iz zajedničkog pretka *ios*, a imaju zajedničkog potomka, klasu *iostream* što predstavlja ulazno-izlazne tokove. Deo hijerarhije klasa za razmenu podataka sa okolinom prikazan je na slici D1.



Slika D1

Pošto se ovde radi o tematici koja nije neposredno vezana za objektno programiranje, a sama po sebi nije posebno komplikovana, u ovom dodatku ćemo se osvrnuti samo na elementarni ulaz-izlaz, najviše zbog toga da bi čitalac bio u stanju da izrađuje test programe za sopstvene klase.

Prilikom startovanja programa automatski se formiraju četiri standardna toka koji obezbeđuju razmenu podataka sa konzolom, tj. kombinacijom tastatura-ekran. Pomenućemo tri:

istream cin	- ulaz sa tastature
ostream cout	- izlaz na ekran
ostream cerr	- izlaz na ekran namenjen porukama.

Sva tri toka podležu redirekciji. U nastavku obradićemo nekoliko osnovnih sredstava za razmenu podataka preko ovih tokova i to uz primenu konverzije. Drugim rečima, razmotrićemo C^{++} ekvivalente funkcija *scanf* i *printf* (koje su, podvucimo, kao takve primenljive i u C^{++}). Za prenos sa konverzijom predviđene su dve operatorske slobodne funkcije (dakle, operatori) i to:

- operator >> za ulaz (tok *cin*; igra ulogu funkcije *scanf*)
- operator << za izlaz (tokovi *cout* i *cerr*; igra ulogu funkcije *printf*).

Operatori su binarni, pri čemu je levi operand neki od pomenutih tokova. Preklopljeni su za sve standardne tipove podataka, uključujući i stringove (tip char*) koji se pojavljuju kao desni operand. Način njihove primene je respektivno

cin << *lvrednost*
cout << *izraz*
cerr << *izraz*

gde se na ulazu pored *lvrednosti* može pojaviti i string. Na podatke se implicitno primenjuju standardne konverzione specifikacije jezika C.

Operatori >> i << podešeni su tako da se izbegne eksplicitno navođenje konverzionih specifikacija tipa %d, %f itd., mada analogna sredstva postoje. Pored toga, operatori vraćaju tip odgovarajuće klase, što omogućuje njihovu višestruku uzastopnu primenu. Posebno, operator ulaza >> ne zahteva primenu adresnog operatora &. Evo nekih primera (imena promenljivih odabrana su tako da ukažu na njihov tip):

cin >> intPod	odgovara	scanf("%d", &intPod)
cin >> intPod >> charPod	odgovara	scanf("%d%c",&intPod,&charPod)
cout << floatPod	odgovara	printf("%f", floatPod)

Na izlazu se mogu koristiti - i intenzivno se koriste - konstantni stringovi u ulozi pratećeg teksta. Ako je vrednost promenljive intPod npr. 1 tada će izvršenje naredbe

```
cout << "Vrednost intPod: " << intPod;
```

rezultovati pojavom teksta

```
Vrednost intPod: 1
```

Kako je pomenuto, posebne specifikacije za konverziju ne moraju se navoditi, što ne znači da ne postoji mogućnost uređivanja teksta koja je naročito potrebna pri izlazu. Osnovno sredstvo za ovu svrhu jeste metoda *setf* iz klase *ios* sa dva oblika od kojih navodimo jednostavniji:

long ios::setf(long *maska*)

gde je *maska* niz bitova što predstavljaju indikatore vezane za konverziju. Metoda *setf* dejstvuje nešto drukčije od konverzionih specifikacija jer se poziva autonomno postavljajući indikatore na vrednosti koje oni zadržavaju sve do eksplicitne promene. Metoda vraća zatečenu vrednost maske.

Maska se formira bit-disjunkcijom posebnih konstanata definisanih u klasi *ios*. Svaka od njih postavlja indikator za odgovarajuću konverziju, a karakteristične su sledeće:

ios::left	ravnanje podataka uz levu ivicu izlaznog polja

ios::right	ravnanje podataka uz desnu ivicu izlaznog polja
ios::dec	decimalna konverzija
ios::oct	oktalna konverzija
ios::hex	heksadecimalna konverzija
ios::showbase	prikaz oznake baze (0 za oktalni, 0x za heksadecimalni sistem)
ios::showpoint	obavezno prikazivanje decimalne tačke pri izlazu realnih konstanata
ios::fixed	prikaz realnih konstanata u decimalnom obliku
ios::scientific	prikaz realne konstante u eksponencijalnom obliku

Na primer, posle poziva

cout.setf(ios::hex | ios::showbase);

svi celobrojni izlazi biće predstavljeni u heksadecimalnoj formi sa vodećim znakovima 0x. Pored metode *setf* od interesa je i inverzna operacija

long ios::unsetf(long *maska*)

koja resetuje sve indikatore setovane u zadatoj *masci*. Naredba

cout.unsetf(ios::hex | ios::showbase);

primenjena negde posle gornjeg poziva *setf*, vratiće stanje na ono koje je važilo pre poziva *setf*. Metoda

int ios::width(int *širina*)

postavlja širinu (broj pozicija) izlaznog polja na vrednost zadatu argumentom. Konačno, metoda

int ios::precision(int *tačnost*)

služi za zadavanje broja decimala odn. značajnih cifara pri izlazu realnih vrednosti.

Podešavanje formata ulaza-izlaza može se obaviti i u okviru samog izvršavanja odgovarajuće operacije, korišćenjem tzv. *manipulatora* [77] koji se pojavljuju kao operandi operatora >> i <<. Neki od njih su

dec	decimalna konverzija
oct	oktalna konverzija

hex	heksadecimalna konverzija
endl	prelazak u novi red.

Poslednji manipulator *endl* najčešće se i koristi jer predstavlja ekvivalent za '\n' iz formatnog stringa funkcije *printf*. Inače, i sam znak \n može se koristiti kao deo (najčešće konstantnog) stringa na izlazu. Naredba

```
cout << hex << intPod << endl;
```

prikazaće vrednost intPod u heksadecimalnom obliku i po završenom izlazu preneti kursor u novi red.

Na kraju, operatori >> i (naročito) << mogu se dalje preklapati u okviru klasa kao, uostalom, i druge operatorske funkcije. Kao ilustraciju navodimo preklopljeni operator izlaza u okviru više puta pominjane operatorske klase *Complex*. Konciznosti radi, prikazaćemo samo onaj deo klase koji se odnosi neposredno na preklopljeni operator:

```
class Complex {
. . . . . .
friend ostream& operator <<(ostream&, const Complex&);
};

ostream& operator <<(ostream& outstr, const Complex z) {
  outstr << '(' << z.r << ',' << z.i << ')';
  return outstr;
}
```

Primer. Izvršenje segmenta

```
Complex a(1,2);
cout.setf(ios::fixed | ios::showpoint); cout.precision(1);
cout << "Vrednost a: " << a << endl;
```

rezultovaće izlazom oblika

```
Vrednost a: (1.0,2.0)
```

D2. PROSTOR IMENA

U velikim aplikacijama sa nizom raznovrsnih modula razvijenih od strane brojnih timova može se desiti i dešava se da se isti identifikator veže za sasvim različite objekte, funkcije, globalne promenljive, klase itd., dakle za entitete koji međusobno ne moraju imati nikakve veze. Ova pojava, sinonimija nesaglasnih entiteta, nazvana *zagađenje prostora imena* [88], krajnje je nepoželjna jer može otežati postupak integrisanja velike aplikacije, pošto zahteva izmene identifikatora u različitim modulima i višekratnu kompilaciju. Da bi se ovaj, nimalo naivan problem rešio ili bar ublažio u novu, standardnu verziju C^{++} ugrađeno je rukovanje tzv. ***prostorom imena*** (engl. "namespace"). U opštem slučaju, prostor imena jeste skup nesusednih segmenata izvornog koda objedinjen zajedničkim imenom. Osnovna njegova osobina je to što su identifikatori definisani u istom prostoru imena unikatni, dok se u različitim prostorima mogu pojaviti i duplikati. Odmah napominjemo da identifikatori definisani u jednom prostoru imena svakako mogu biti referencirani iz drugog prostora imena, ali na poseban način, korišćenjem već poznatog mehanizma kvalifikovanja. U svetlu ovog novog pojma, sve što je do sada izloženo u ovoj knjizi, a vezano za C^{++}, odnosi se na jedan jedini prostor imena, koji nosi naziv *globalni prostor imena*. Ono što ćemo pokazati u nastavku jeste mogućnost da se definišu posebni, dodatni prostori imena koji se razlikuju od do sada korišćenog globalnog. Unutar svakog prostora imena definiše se i koristi sve što smo i do sada definisali: klase, objekti, slobodne funkcije, globalne promenljive itd. Segmenti koji ulaze u isti prostor imena čak ne moraju biti ni u istoj datoteci (!). Prostor imena definiše se sintaksnom konstrukcijom

```
namespace naziv_prostora {
  sadržaj: klase, funkcije, promenljive itd.
}
```

gde je *namespace* službena reč, a *naziv_prostora* odabrani identifikator prostora imena. Za primer, obradićemo jednostavan sistem klasa sastavljen od klasa *Point* i *Line* koje predstavljaju respektivno tačku i duž u Dekartovom koordinatnom sistemu. Inače, neke verzije ovih klasa smo već razmatrali u poglavlju 4, a primer je odabran zbog toga što je sam po sebi jasan i ne zahteva nikakva posebna objašnjenja. Sistem klasa reorganizovaćemo u odnosu na ranije verzije tako što ćemo definicije metoda klasa i *Point* i *Line* smestiti u jedinstvenu biblioteku POINTLINE.CPP i za telo biblioteke vezati dva zaglavlja: jedno se nalazi u datoteci POINT.HPP i sadrži definiciju klase *Point*; drugo koje sadrži definiciju klase *Line* smešteno je u datoteku LINE.HPP. Sistem klasa *Point* i *Line* izdvojićemo u zaseban prostor imena sa nazivom *points_and_lines*, vodeći pritom računa o tome da se u ovaj prostor uključe ne samo definicije dveju klasa iz

zaglavlja nego i definicije metoda i slobodnih funkcija koje se nalaze u telu biblioteke POINTLINE.CPP.

```
/*****************************************************************
**

                        ZAGLAVLJE KLASE POINT

Naziv datoteke: POINT.HPP
*****************************************************************
*/
#ifndef POINT_HPP
  #define POINT_HPP

namespace points_and_lines {

class Point {
friend class Line;
private:
  double x, y;
public:
  Point(double xx, double yy);
  double getX() const;
  double getY() const;
  double distance() const;
  friend double distanceBetween(const Point& p1,const Point& p2);
};

}

#endif
```

```
/*****************************************************************
**

                    ZAGLAVLJE KLASE LINE

Naziv datoteke: LINE.HPP
*****************************************************************
*/
```

```
#ifndef LINE_HPP
  #define LINE_HPP
#include "point.hpp"

namespace points_and_lines {

class Line  {
private:
  Point a, b;
public:
  Line(double, double, double, double);
  Line(const Point&, const Point&);
  double getAx() const;
  double getAy() const;
  double getBx() const;
  double getBy() const;
  double length() const;
};

}

#endif
```

```
/*****************************************************************
**

              TELO BIBLIOTEKE SA KLASAMA POINT I LINE

Naziv datoteke: POINTLINE.CPP
*****************************************************************
*/
#include "point.hpp"
#include "line.hpp"
#include <math.h>

namespace points_and_lines {

// METODE KLASE POINT
Point::Point(double xx, double yy) {
  x= xx;
  y= yy;
```

```
}
double Point::getX() const {
  return x;
}
double Point::getY() const {
  return y;
}
double Point::distance() const {
  return sqrt(x*x + y*y);
}
double distanceBetween(const Point& p1,const Point& p2) {
  return sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2));
}


// METODE KLASE LINE
Line::Line(double x1,double y1,double x2,double y2): a(x1,y1),b(x2,y2) {}
Line::Line(const Point& p1,const Point& p2): a(p1),b(p2) {}
double Line::getAx() const {
  return a.x;
}
double Line::getAy() const {
  return a.y;
}
double Line::getBx() const {
  return b.x;
}
double Line::getBy() const {
  return b.y;
}
double Line::length() const {
  return sqrt(pow((a.getX()-b.getX()),2)+
              pow((a.getY()-b.getY()),2));
}

}
```

Neka se klijent klase (npr. neki test-program *main*) nalazi u globalnom prostoru imena, onom istom koji je korišćen u svim dosadašnjim razmatranjima u ovoj knjizi. To pak znači da pristup identifikatorima korišćenim u prostoru imena *points_and_lines* kojem pripadaju *Point* i *Line* više nije moguć bez dodatnih

sredstava. Tako, na primer, u klijentu nije moguće napisati

```
Point p1(1,2), p2(3,4);    // Greska! Point nije u globalnom prostoru imena
```

zato što identifikator *Point* ne pripada globalnom prostoru imena, nego prostoru *points_and_lines*. Da bi se identifikator *ime* iz prostora imena sa nazivom *prostor_imena* mogao koristiti u nekom drugom prostoru, neophodno je upotrebiti njegov potpuni oblik koji se dobija ***kvalifikovanjem*** pomoću naziva prostora imena. Dakle, potpuni oblik identifikatora *ime* iz prostora imena *prostor_imena* koji se koristi izvan tog prostora jeste

prostor_imena::ime

Prema tome, potpuni identifikator klase *Point* iz prostora imena *points_and_lines* ima oblik *points_and_lines::Point*, te gornja naredba treba da dobije oblik

```
points_and_lines::Point p1(1,2), p2(3,4);   // korektno
```

U skladu s tim, sve klase, funkcije, metode itd. koje smo do sada koristili, a koje pripadaju globalnom prostoru imena, implicitno su kvalifikovane kao

::ime

jer globalni prostor imena nema poseban naziv.

Kvalifikovanje nazivom prostora imena jeste osnovni način za potpunu identifikaciju. Pošto nazivi prostora imena mogu biti dugački i često korišćeni, sintaksa C^{++} obogaćena je sredstvima za skraćivanje. Jedan način koji se koristi kod dugih naziva prostora imena[92] jesu tzv. ***alijasi***. Alijas je samo dodatni naziv za prostor imena koji važi samo u klijentu i koji može da bude kraći. Dobija se naredbom

namespace alijas = naziv originalnog prostora imena;

U oblasti dejstva ove naredbe u programu u kojem se nalazi, umesto originalnog imena koristi se (kraći) alijas. Primer:

```
namespace PAL = points_and_lines;
. . . . .
PAL::Point p1(1,2), p2(3,4);
```

[92] Nazivi prostora imena u praksi i jesu dugi, jer oni moraju biti unikatni!

gde je *PAL* alijas (tj. sinonim) naziva *points_and_lines*.

Pored direktnog kvalifikovanja, a u svrhu skraćivanja koda, u sredstva za rukovanje prostorima imena uključena je još jedna mogućnost: naredba *using*. Svrha ove naredbe jeste da se njome u klijentu, a na jednom jedinom mestu, navede kvalifikator kojim se kvalifikuju sve pojave istog identifikatora ili čak svi identifikatori datog prostora imena. Prvi oblik je

using naziv_prostora::ime;

kojim se, u okviru dometa naredbe *using*, svaka pojava identifikatora *ime* automatski kvalifikuje sa *naziv_prostora::*. Dakle, ako navedemo

using points_and_lines::Point;

u daljem tekstu možemo koristiti identifikator *Point* bez ikakvog kvalifikovanja jer se kvalifikator *points_and_lines::* uz naziv *Point* podrazumeva. Ako isto želimo i za ime *Line*, moramo pisati

using points_and_lines::Line;

Konačno, jednom naredbom oblika

using namespace naziv_prostora;

omogućujemo da se *svi* identifikatori iz navedenog prostora imena u klijentu koriste bez kvalifikovanja. Naredbom

using namespace points_and_lines;

omogućujemo ne samo korišćenje nekvalifikovanih imena *Point* i *Line* nego i ma kojeg drugog dostupnog imena u prostoru imena *points_and_lines*. Evo jednog jednostavnog test-programa:

```
/*****************************************************************
**

                    ILUSTRACIJA NAREDBE "USING"

*****************************************************************
*/
```

```
#include "point.hpp"
#include "line.hpp"
#include <iostream>

using namespace std;
//using namespace points_and_lines;  //prva varijanta: zajednicka deklaracija
using points_and_lines::Point;  //druga varijanta: posebne deklaracije
using points_and_lines::Line;   //druga varijanta: posebne deklaracije

int main()
{

  Point p1(1,2), p2(3,4);

  cout << "Instancirane tacke (1,2) i (3,4)" << endl;
  cout << "Apscisa prve tacke: " << p1.getX() << "    "
      << "Ordinata prve tacke: " << p1.getY() << endl;
  cout << "Apscisa druge tacke: " << p2.getX() << "    "
      << "Ordinata druge tacke: " << p2.getY()<< endl;
  cout << "Rastojanje izmedju tacaka: " << distanceBetween(p1,p2) << endl;


  Line myLine(p1,p2);
  cout << "\n\n\nInstancirana duz (1,2) (3,4)" << endl;
  cout << "Apscisa prve tacke: " << myLine.getAx() << "    "
      << "Ordinata prve tacke: " << myLine.getAy() << endl;
  cout << "Apscisa druge tacke: " << myLine.getBx() << "    "
      << "Ordinata druge tacke: " << myLine.getBy()<< endl;
  cout << "Duzina: " << myLine.length() << endl;

  return 0;
}
```

U osenčenom delu date su dve varijante: automatsko kvalifikovanje kompletnog sadržaja points_and_lines (prva varijanta, data pod komentarom) ili kvalifikovanje samo sadržaja klasa *Point* i *Line*.

U primeru se vidi još nešto: upotreba naredbe *using* za otvaranje prostora imena sa nazivom *std*. Radi se o prostoru imena koji je deo nove verzije C^{++}, a kojem, između ostalog, pripadaju i objekti *cin* i *cout* definisani u novoj verziji biblioteke čije se zaglavlje više ne zove *iostream.h*, nego *iostream*. Imena *cin*, *cout* i *endl* uključena su u prostor imena *std* te, shodno tome, zahtevaju kvalifikovanje bilo pri

svakom korišćenju, bilo upotrebom *using*. Dakle, više se ne može koristiti naredba oblika npr.

```
cout << "Ovo je tekst" << endl;   // greska
```

nego

```
std::cout << "Ovo je tekst" << std::endl;
```

ili

```
using namespace std;
. . . . .
cout << "Ovo je tekst" << endl;
```

I na kraju: prostor imena može biti definisan unutar drugog prostora imena i tada se ponaša kao i svaka druga uklopljena struktura. Ako se, na primer, u unutrašnjem prostoru imena *unutrasnji* nalazi neki identifikator *indX* koji postoji i u spoljašnjem *spoljasnji*, tada za pristup *indX* iz spoljašnjeg prostora imena treba kvalifikovanje oblika *spoljasnji::indX*. Ako pak identifikator *indX* iz unutrašnjeg prostora treba dosegnuti iz segmenta izvan spoljašnjeg prostora, mora se koristiti dvostruko kvalifikovanje *spoljasnji::unutrasnji::indX*.

LITERATURA

[1] Meyer B.: Object-Oriented Software Construction, Prentice Hall, 1988
[2] Martin R.: Designing Object-Oriented C++ Applications Using the Booch Method, Prentice Hall, 1995
[3] Booch G.: *Object-Oriented Analysis and Design*, second edition, Addison-Wesley, 1994
[4] *A Brief History of Computing*, http://ox.compsoc.net/~swhite/timeline.html
[5] Hotomski P., Malbaški D.: *Matematička logika i principi programiranja*, Univerzitet u Novom Sadu, 2000.
[6] De Marco T.: Structured Analysis and System Specification, Prentice-Hall, 1979
[7] Dahl O.-J., Dijkstra E.W., C.A.R. Hoare: *Structured Programming*, Academic Press, 1972
[8] Uspenskij V.A., Semenov A.L.: *Teorija algoritmov: osnovnye otkrytija i priloženija*, Nauka, Moskva, 1987
[9] Marković M.: *Filozofski osnovi nauke*, Izabrana dela, Tom I, BIGZ, Genes_S štampa, Prosveta, SKZ, 1994.
[10] Wetherbe J.C.: *Systems Analysis & Design*, West Publishing Company
[11] Pappas C., Murray W.: *C/C++ Vodič za programere*, Mikro knjiga, 1996.
[12] Liberty J.: *C++ za 21 dan*, Kompjuter biblioteka, 1999.
[13] Stanojević I., Surla D.: *UML - Uvod u objedinjeni jezik modeliranja*, Grupa za informacione tehnologije, Novi Sad, 1999.
[14] Rumbaugh J., *OMT: The Object Model*, JOOP, January, 1995
[15] Rumbaugh J., *OMT: The Dynamic Model*, JOOP, February, 1995
[16] Rumbaugh J., *OMT: The Functional Model*, JOOP, March-April, 1995
[17] Rumbaugh J., *OMT: The Development Process*, JOOP, May, 1995
[18] Rumbaugh J., *OMT: What is a Method?*, JOOP, 1995
[19] Rumbaugh J., *OMT: Qualified Name*, JOOP, May, 1995
[20] Rumbaugh J., OMT: Going with the Flow: Flow Graphs in their Various Manifestations, JOOP, June, 1995
[21] Rumbaugh J., OMT: What's in a Name? A Qualified Answer, JOOP, 1995
[22] Rumbaugh J., OMT: Taking Things in Context: Using Composites to Build

Model, JOOP, November-December, 1995
[23] Ericsson H., Penker M.: *UML Toolkit*, John Wiley and Sons, 1997
[24] Jacobson I.: Object-Oriented Software Engineering, Addison-Wesley, 1994
[25] O'Brien S.: *Turbo Pascal 6*, Mikro knjiga, Beograd, 1991.
[26] O'Brien S., Nameroff S.: *Turbo Pascal 7*, Osborne McGraw-Hill, 1993
[27] Ellmer E.: *Object-Orientation - an Overview*, University of Viena, www.ifs.univie.ac.at/ISOO/overview.html
[28] Wirth N.: Algorithms+Data Structures=Programs, Prentice-Hall, 1976
[29] Stroustrup B.: *Programski jezik C++*, Mikro knjiga, Beograd, 1991.
[30] Eckel B.: *Thinking in Java*, Mind View Inc., 1997
[31] Dijkstra E.W.: *A Discipline of Programming*, Prentice-Hall, 1976
[32] Wirth N.: *A Plea for Lean Software*, Computer, February 1995
[33] Lipaev V.V.: *Kačestvo programmnogo obespečenija*, Finansy i statistika, Moskva, 1983
[34] *Enciklopedija kibernetiki*, Akademija nauk Ukrainy, 1974
[35] Brooks F.: *The Mythical Man-Month*, Addison-Wesley, 1975
[36] Obradović D., Malbaški D.: *Osnovne strukture podataka*, Univerzitet u Novom Sadu, 1995.
[37] Marciniak J., ed.: *Encyclopaedia of Software Engineering*, John Wiley and Sons, 1994
[38] Webster Illustrated Contemporary Dictionary, Encyclopedic Edition, 1987
[39] Ivetić D., Malbaški D.: *An Analysis of Streams as Automata*, Proc. 4th Balcan Conference On Operational Research, Thessaloniki, Greece, 1997
[40] Malbaški D., Obradović D.: *Informacione tehnologije - pojam i sadržaj*, izlaganje po pozivu, simpozijum "Informacione tehnologije i primena", Novi Sad, 1995.
[41] Ivetić D., Malbaški D.: *A Dichotomous Software Life Cycle Model*, Journal of Applied Systems Studies, Methodologies and Applications for Systems Approaches (JASS), Volume 2, Number 2, July, 2001
[42] Ivetić D., Malbaški D.: Paralelizmi u životnom ciklusu softvera", XLII Konferencija ETRAN, Vrnjačka Banja, 1998.
[43] Ivetić D., Malbaški D., Obradović D.: A Formal Description of the HCI-SE Coupling, 5th Balcan Conference on Operations Research, Banja Luka, 2000
[44] Myers B., Rosson M.B.: *Survey on User Interface Programming*, CHI'92 Conference Proceedings on Human Factors in Computing Systems, ACM Press, 1992
[45] Denning P., Gries D. et al.: *Computing as a Discipline*, Special report, Computer, February 1989
[46] Couger J., Knapp R.: *System Analysis Techniques*, John Wiley and Sons, 1974

[47] Dujmović J.J.: *Programski jezici i metode programiranja*, Naučna knjiga, Beograd, 1990.
[48] Kuročkin V.M. (ed.): *Semantika jazykov programmirovanija*, Mir, Moskva, 1980
[49] Meek B., Heath P. (ed.): *Guide to Good Programming Practice*, John Wiley and Sons, 1980
[50] Malbaški D., Obradović D.: *Objektna tehnologija: suština, stanje i perspektive*, uvodni referat, IV Međunarodno savetovanje o dostignućima elektro i mašinske industrije, Banja Luka, 2001.
[51] Shaw M.: *Comparing Architectural Design Styles*, IEEE Software, November 1995
[52] Monroe M. et al.: Architectural Styles, Design Patterns and Objects, IEEE Software, January 1997
[53] Siegel S.: *Object Oriented Software Testing*, John Wiley and Sons, 1996
[54] Binder R.: Modal Testing Strategies for OO Software, Computer, November 1996
[55] D. Ivetić, D. Malbaški: *Primena strima u formalnoj specifikaciji ponašanja programa*, XXXVIII Konferencija ETAN, Niš 1994
[56] Simon H.: *The Sciences of the Arificial*, Cambridge MA, The MIT Press, 1982
[57] *Filozofijski rečnik*, red. V. Filipović, Nakladni zavod Matice hrvatske, Zagreb, 1984.
[58] Malbaški D.: *Objekti i objektno programiranje kroz programske jezike C++ i paskal*, monografija, Fakultet tehničkih nauka, Novi Sad, 2006.
[59] Prešić S.: *Elementi matematičke logike*, Matematička biblioteka, Beograd, 1968.
[60] Kalman R., Falb P., Arbib M.: *Topics in Mathematical System Theory*, McGraw-Hill, 1969 (prevod na ruski Mir, Moskva, 1979)
[61] Smith M., Tockey S.: *An Integrated Approach to Software Requirements Definition Using Objects*, Scattle WA: Boeing Commercial Airplane Support Division, 1988 (prema [3])
[62] Cox B.: Object Oriented Programming: An Evolutionary Approach, Reading MA, Addison - Wesley, 1986
[63] The Oxford Companion to Philosophy, Oxford University Press, 1995
[64] Lerner A.J.: *Principi kibernetike*, Tehnička knjiga, Beograd, 1970.
[65] Rosen G.: *Abstract Objects*, Stanford Encyclopedia of Philosophy, http://plato.stanford.edu, 2001
[66] Swoyer C.: *Properties*, Stanford Encyclopedia of Philosophy, http://plato.stanford.edu, 1999, 2000
[67] Andrianov A.N., Byčkov S.P., Horošilov A.I.: *Programmirovanie na jazyke simula-67,* Nauka, Moskva, 1985

[68] Malbaški D., Obradović D.: *O terminološkim problemima u oblasti informacionih tehnologija*, Naučni skup o standardizaciji terminologije, SANU, Beograd, 1996.
[69] Ten H.: *Filozofija umetnosti*, izdanje I.Đ.Đurđevića, Beograd 1921., fototipsko izdanje, "Dereta", Beograd
[70] Shaw M.: *Abstraction Techniques in Modern Programming*, IEEE Software, vol. 1(4), Oct. 1984
[71] Berzins V., Gray M., Naumann D.: *Abstraction-Based Software Development*, Comm. of the ACM, vol. 29 (5), May 1986
[72] Malbaski D., Obradovic D.: *On Some Basic Concepts in Object Orientation*, 6th Balcan Conference on Operational Research, Thessaloniki, 2002
[73] Malbaški D.: *Odabrana poglavlja metoda programiranja*, Univerzitet u Novom Sadu, 2001.
[74] Parnas D.: On the Criteria to be Used in Decomposing Systems into Modules, Comm. of the ACM, Dec. 1972
[75] Parnas D., Clements P., Weiss D.: *The Modular Structure of Complex Systems*, IEEE Trans. on Software Engineering, Vol. SE-11, March 1985
[76] Opća enciklopedija Jugoslavenskog leksikografskog zavoda, Zagreb 1979.
[77] Kraus L.: *Programski jezik C^{++} sa rešenim zadacima*, Mikro knjiga, Beograd, 1994.
[78] Amadi M., Cardelli L.: *A Theory of Objects*, Springer-Verlag, New York, 1996
[79] Zarić M..: *Definicije polimorfizma*, seminarski rad na poslediplomskim studijama, Fakultet tehničkih nauka, Novi Sad, 2003.
[80] Strachey C.: *Fundamental Concepts in Programming Languages*, Lecture Notes for International Summer School in Computer Programming, Copenhagen, 1967
[81] Cardelli L., Wegner P.: On Understanding Types, Data Abstraction and Polymorphism, ACM Computer Surveys, 1985
[82] Meyer B.: *Objektno orijentisano konstruisanje softvera*, prevod II izdanja, CET, Beograd, 2003.
[83] Snoeck M., Dedene G.: Existence Dependency: The Key to Semantic Integrity Between Structural and Behavioral Aspects of Object Types, IEEE Trans. on Software Engineering, vol. 24, no. 4, April 1998
[84] Lieberherr K.J., Holland I.: *Formulations of the Law of Demeter*, Technical Report Demeter 2, Northeastern University, Boston, 1988
[85] Lieberherr K.J., Holland I., Riel A.: *Object-Oriented Programming: An Objective Sense of Style*, OOPSLA'88 Proceedings, 1988
[86] Rumbaugh J., Jacobson I., Booch G.: *The Unified Modeling Language Manual*, Addison-Wesley, 1999
[87] Naughton P., Schildt H.: *Java: The Complete Reference*, Mc-Graw Hill,

1997
[88] Lippman S., Lajoie J.: *C++ Primer*, Addison-Wesley, 2000, prevod na srpski CET Computer Trade and Equipment, Beograd
[89] Agafonov V.N.: *Specifikacija programm: ponjatiynye sredstva i ih organizacija*, Nauka, Sibirskoe otdelenie, Novosibirsk, 1987
[90] Malbaški D.: *Semantika programskih jezika*, int. rep. no. 021-21/17, Fakultet tehničkih nauka, Novi Sad, 2003.
[91] Malbaški D.: *Bulovska aproksimacija fazi iskaznih formula*, int. rep. no. 1/2002, Tehnički fakultet "Mihailo Pupin", Zrenjanin, 2002.
[92] Jorgensen P.C.: Software Testing - A Craftman's Approach, CRC Press, 1995
[93] Malbaški D.: *Programski račun*, int. rep. no. 021-21/85, Fakultet tehničkih nauka, Novi Sad, 2004.
[94] Malbaški D., Ivetić D.: *Some Notes on the Formal Definition of Streams*, YUJOR, Vol.6, No. 2, 1996.
[95] Hehner E.C.: Specifications, Programs and Total Correctness, University of Toronto, 1998
[96] Gries D.: *The Science of Programming*, Springer Verlag, 1981, prevod na ruski, "Mir", Moskva, 1984
[97] Building bug-free O-O software: An Introduction to Design By Contract, http://eiffel.archive.com
[98] Larousse
[99] 1130 Scientific Subroutine Package, IBM, 1968
[100] Markoski B., Malbaški D., Hotomski P.: *Verifying Program Correctness By Resolution Method*, ETAI 2000 V National Conference with International Participatio, Ohrid, 2000.
[101] ISO/IEC 2382 - 17.02.5
[102] Petrović G.: *Logika*, Školska knjiga, Zagreb 1981.
[103] Kupusinac A.: *Invarijanta klase u objektno orijentisanom programiranju*, magistarski rad, Fakultet tehničkih nauka, Novi Sad, 2007.
[104] Stanford Encyclopedia of Philosophy, http://plato.stanford.edu

INDEKS

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

Z