Saga o OOP
Dragan Masulovic
(u sklopu Vannastavnih aktivnosti iz programiranja, PMF-Novi Sad)

Uvod

U narednih nekoliko poruka (mozda iz u nekoliko navrata) desice se Saga o OOP. Ideja je da se na brzaka da pregled ideja koji vodi od niza bitova, do OOP. Nakon pregleda osnovnih ideja mozemo razviti diskusiju o specijalitetima kuce (Borland, JPI, ...)

Treba uociti da ovaj pregled ni priblizno nece biti hronoloski, jer (verovali ili ne), prvi jezik sa elementima OO je nastao u Svedskoj i zvao se Simula-67. U to vreme nije doziveo neku veliku srecu, ali je posluzio kao odskocna daska za Smalltalk-80 (Xerox, Palo Alto 1980), od koga je sve pocelo.

No, podjimo redom (dakle, ne-redom, ne-hronoloski).

Glava 1.

% U kojoj se postavlja *ono* pitanje. Pitanje od koga mnoge stvari
% stvari pocinju, koga niko ne voli, a na koga se mnogi nastavnici
% mrste, kao kada ih ucenici pitaju da im potanko objasne kako
% bebe dolaze na svet.

Sta je to tip podataka, a sta struktura? U cemu je razlika izmedju ta dva pojma?

```
+-------------------------------------------------------------------+
| Pre nego sto odgovori na pitanja budu isporuceni, jedna napomena: |
| ovde necete naci *stroge* i *formalne* definicije pojmova. Samo   |
| ideje. Za one *prave* matematicke definicije je potrebno dati     |
| prilicno neprijatan uvod u nekoliko zabacenih teorija. O tome     |
| drugi put...                                                      |
+-------------------------------------------------------------------+
```

Tip podataka je odredjen (konacnim) skupom vrednosti i nekim paketom operacija i relacija nad elementima tog skupa. Na primer, ({-32768, ..., -1, 0, 1, ..., 32767}, +, -, * div, mod, =, <) je tip podataka u narodu poznatiji kao celobrojni tip (ili INTEGER). To sto se u programiranju zove tip podataka jako lici na ono sto se u algebri i teoriji modela zove operacijsko-relacijska struktura, mada stanovite (i to ZNATNE) razlike postoje. O njima drugom prilikom.

Struktura podataka je konglomerat podataka. Ona se formira od drugih (jednostavnijih) struktura i od tipova. Jedine operacije koje se nad strukturama mogu obavljati su operacije selekcije. Posao operacija selekcije je da iz strukture izdvoje jedan jednostavniji podatak (jednu komponentu strukture), ili da nam omoguce pristup ka njoj.

Ovde treba razlikovati strukturu podataka od metoda struktuiranja podataka. Recimo, ``array'' je metod struktuiranja, a nakon deklaracije

```
var a : array [1 .. 100] of real;
```

``a'' postaje struktura. Svaki metod struktuiranja ima karakteristicne operacije selekcije. Recimo, ``array'' kao operaciju selekcije ima indeksiranje, a ``record'' kao operaciju selekcije ima operaciju koja se zove projektovanje (u stvari, to je samo odabir polja sloga).

U Pascal-u postoje tri eksplicitna metoda struktuiranja (array, record, record-case) i jedan implicitni (preko pokazivaca).

Razlika izmedju tipa i strukture je u tome sto je nad elementima tipa moguce vrsiti nekakve operacije i smestati ih u nekakve relacije, dok je nad strukturom moguce obavljati samo selekciju komponente.

Malko matematike (ali *jako* malko) na ovu temu ce se javiti u Dodatku 1.

Glava 2.

% U kojoj se ono sto se sada svima cini jasno, nacini nejasnim

I tek sto nam se sjasnilo sta je to tip, a sta struktura, pocinjemo malko da gledamo po programskim jezicima. I uvidjamo da granica izmedju ova dva pojma nije uopste jasna.

Na primer, u Pascal-u su stringovi implementirani kao pakovani nizovi karaktera, i tako predstavljaju strukturu. S druge strane, u BASIC-u su stringovi implementirani kao osnovni tip (jer se na stringovima u BASIC-u mogu obavljati razne operacije i relacije: nadovezivanje, uzimanje podstringa, ispitivanje jednakosti stringova...)

Prema ovome se cini da je stvar ostavljena onome ko dizjanira jezik. U jednom jeziku se nesto moze javiti kao tip, a u drugom jeziku to isto kao struktura. I cini se da smo na taj nacin pomirili antagonizme. Ali nismo!

Jer, sta se desava kada u Pascal-u napisete paket procedura koje rade sa stringovima? Recimo, kada se u nekom programu javi ovako nesto:

```
type STR = packed array [1 .. 100] of char;

procedure Concat(s1, s2 : STR; var res : STR); ...
procedure Substr(s : STR; from, to : integer; var res : STR); ...
function  Compare(s1, s2 : STR) : integer; ...
```

Nije li tako STR postao tip? Imamo skup vrednosti (CHAR^100) i operacije na elementima tog skupa (Concat, Substr, Compare). Sve sto treba da nesto bude tip ...

??

Glava 3.

% koja, nakon dramske pauze, daje odgovor na pitanje iz prethodne glave

Jeste. STR iz prethodne glave je postao tip. Dakle, STR je, po sebi, struktura, ali je u vrednim rukama programera postao tip.

Naravoucenije: svaku strukturu mozemo pretvoriti u tip ako je snabdemo operacijama koje rade (nesto korisno, ali to nije obavezno) sa njom.

Prethodna glava je demonstrirala klasican nacin da struktura postane tip: damo strukturi ime i snabdemo je operacijama. Dobijamo novi tip.

Glava 4.

% U kojoj se vidi zasto je vazno praviti nove tipove i kako cemo
% biti srecni kada to budemo radili u OO stilu

Zasto je vazno praviti nove tipove? Zbog toga sto je mnogo lakse raditi kada se apstrahuju nebitni detalji. (To su oni detalji koji nisu bitni u *tom* trenutku; svaki detalj je bitan, pre ili kasnije!)

Za uspesan rad u velikim timovima i na velikim projektima bitno pisati apstraktan kood. Zamislite projekt u kome je potrebno raditi nesto sa matricama. Najbolje je napraviti tip MATRIX (dakle, nekoj strukturi dati ime MATRIX i onda je snabdeti operacijama za rad sa matricama (sabiranje, mnozenje, invertovanje, ...)), pa kada dodje do resavanja konkretnog problema, samo pozivati gotove (i testirane) procedure.

Nije dobra politika pisati svaku operaciju iz pocetka i to bas tamo gde vam je zatrebala. To ne samo da uvecava kood, nego i povecava mogucnost da dodje do greske. Osim toga, cilj modernih programskih jezika je u tome da tekst programa sto vernije odslikava strukturu problema. Ako je potrebno u jednom trenutku uraditi neku vratolomiju sa matricama, idealno bi bilo da se moze napisati nesto ovako:

```
C := Inv(A) * Transpose(B) + Det(Y) * Adj(X)
```

Nesto nepozeljnije, ali jos uvek prihvatljivo je ovakvo parce kooda:

```
Inv(A, A1);
Transpose(B, B1);
MatMul(A1, B1, P);

Adj(X, X1);
ScalMul(Det(Y), Q);

MatAdd(P, Q, C);
```

No, najruzniji, najlosiji i najneprihvatljiviji nacin je onaj koji pocinje ovako:

```
for i := 1 to n do
    for j := 1 to n do
        (* rutina koja invertuje matricu A u A1 *);
for i := 1 to n do
    for j := 1 to n do
        B1[i,j] := B[j, i];
for i := 1 to n do
    for j := 1 to n do
        (* rutina koja mnozi A1 i B1 *);
(* itd *)
```

O ovom poslednjem ne treba ni pricati. Ono drugo moze u svakom programskom jeziku (osim starih *pravih* BASIC-a), a ono prvo samo u nekim jezicima. Izmedju ostalih, i u OO jezicima. Jasno je da je to izrazajna moc koju zelimo!

Ukoliko nemate mogucnost da koristite OO jezik, koristite bar ideje. Radite kao sto je demonstrirano u drugom primeru. Tako ce vasi programi biti razumljiviji, lakse ce se testirati, i programiranje ce teci mnogo prirodnije.

Glava 5.

% U kojoj se vidi sta je to programiranje na jednom visem nivou
% apstrakcije, i koja se zavrsava mantrom za ovu Sagu.

Opsta metodologija programiranja koja se moze okarakterisati kao *dooobra* metodologija se ukratko moze opisati ovako: treba dizajnirati odgovarajuce strukture podataka, napisati procedure za manipulaciju tim strukturama podataka, i kada dodje do kljucnog trenutka, samo se pozivaju gotove procedure koje rade posao.

Tako dolazimo do jednog viseg nivoa apstrakcije u programima. U trenutku kada je potrebno primeniti neke operacije na nekim strukturama, ne interesuje nas *kako* procedure rade, vec *sta* rade.

Primer. Pretpostavimo da imamo tip MATRIX i paket procedura koje barataju sa matricama. Kada kazemo:

```
var a, b : MATRIX;
begin
    NewMatrix(a); NewMatrix(b);
    ReadMatrix(a);
    Transpose(a, b);
    WriteMatrix(b);
    DisposeMatrix(a); DisposeMatrix(b)
end;
```

mi zelimo da posao bude obavljen! Uopste nas ne interesuje da li su matrice implementirane kao

```
type MATRIX = array [1 .. MaxX, 1 .. MaxY] of real;
```

ili su implementirane preko pokazivackih struktura

```
type MATRIX      = ^MatrixEntry;
     MatrixEntry =  record
                        i, j  : integer;
                        entry : real;
                        right, down : MATRIX
                    end;
```

(sto moze biti jako zgodno kada se radi sa retkim matricama).

Kako su moguce razne implementacije koje rade isti posao, to u trenutku upotrebe procedura nije bitna unutrasnja struktura, vec *ponasanje* procedura (koje je dato u specifikaciji paketa).

NIJE BITNA IMPLEMENTACIJA, VEC MANIFESTACIJE (TJ. OSOBINE) OPERACIJA DATOG TIPA PODATAKA.

Glava 6.

% U kojoj se javlja pojam apstraktnog tipa podataka. Tako smo jos
% jedan korak blize pojmu OO. Konvergencija je spora, ali neumitna.

Videli kako struktura podataka snabdevena odgovarajucim operacijama postaje tip podataka.

Apstraktni tip podataka je tip podataka ciju implementaciju ne znamo (primer: ne znamo da li su matrice predstavljene sa array ili preko pokazivaca), ali znamo kako se ponasaju operacije nad vrednostima tog tipa, tako da pisemo program koristeci samo osobine operacija.

Jasno je zasto se to zove tip podataka (zato sto je to upravo tip podataka). A dodat mu je atribut ``apstraktni'' zato sto korisnik ne poznaje konkretnu implementaciju. Korisniku su poznata samo apstraktna svojstva operacija nad tim tipom (drugim recima, poznato je sta koja operacija radi, ali ne i kako to radi). Dakle, apstrahovana je jedna dimenzija problema: implementacija tipa.
(NB: apstrahovati znaci izbaciti iz posmatranja ono sto u tom trenutku nije bitno).

Apstraktni tip podataka cemo oznacavati sa ADT (Abstract Data Type).

Glava 7.

% Dva primera

Primer 1: Stek sa celobrojnim elementima kao apstraktni tip podataka
---

```
(* SPECIFIKACIJA *)

    Tip: StackOfInt

    Open(s) - inicijalizuje stek s; mora se pozvati pre prve
            upotrebe promenljive s
    Close(s) - deinicijalizuje stek s; mora se pozvati kada s vise
            nije potrebna i vise se nece koristiti
    Push(s, n, ok) - na stek s stavi broj n i vrati ok = true; ako to
            nije moguce, vrati ok = false
    Pop(s, n, ok) - sa steka s skine broj n i vrati ok = true; ako to
            nije moguce, vrati ok = false
    Full(s) - vrati true akko na s vise nema mesta
```

```
    Empty(s) - vrati true akko je s prazan

(* IMPLEMENTACIJA *)

type StackOfInt = ^StackEntry;
     StackEntry =  record
                       entry : integer;
                       link  : StackOfInt
                   end;

procedure Open(var s : StackOfInt);
begin
    s := nil
end (* Open *);

procedure Close(var s : StackOfInt);
var
    t : StackOfInt;
begin
    while s <> nil do begin
        t := s;
        s := s^.link;
        dispose(t)
    end
end (* Close *);

procedure Push(var s : StackOfInt; n : integer; var OK : Boolean);
var
    t : StackOfInt;
begin
    new(t);
    if t = nil then
        OK := false
    else begin
        t^.entry := n;
        t^.link  := s;
        s        := t;
        OK       := true
    end
end (* Push *);

procedure Pop(var s : StackOfInt; var n : integer; var OK : Boolean);
var
    t : StackOfInt;
begin
    if s = nil then
        OK := false
    else begin
        t  := s;
        s  := s^.link;
        n  := t^.entry;
        OK := true;
        dispose(t)
    end
end (* Pop *);

function Full(s : StackOfInt) : Boolean;
begin
    Full := false
    (*
     * ovde moze da ide i poziv neke sistemske rutine koja ce
     * proveriti da li ima dovoljno mesta za alokaciju
     *)
end (* Full *);

function Empty(s : StackOfInt) : Boolean;
begin
    Empty := s = nil
end (* Empty *);
```

Primer 2: Stek sa celobrojnim elementima kao apstraktni tip podataka

```
-------------------------------------------------------------------

    (* SPECIFIKACIJA *)

        Tip: StackOfInt

        Open(s) - inicijalizuje stek s; mora se pozvati pre prve
                upotrebe promenljive s
        Close(s) - deinicijalizuje stek s; mora se pozvati kada s vise
                nije potrebna i vise se nece koristiti
        Push(s, n, ok) - na stek s stavi broj n i vrati ok = true; ako to
                nije moguce, vrati ok = false
        Pop(s, n, ok) - sa steka s skine broj n i vrati ok = true; ako to
                nije moguce, vrati ok = false
        Full(s) - vrati true akko na s vise nema mesta
        Empty(s) - vrati true akko je s prazan

    (* IMPLEMENTACIJA *)

    const MaxStack = 100;
    type  StackOfInt = record
                            top : integer;
                            entry : array [1 .. MaxStack] of integer
                       end;

    procedure Open(var s : StackOfInt);
    begin
        s.top := 0
    end (* Open *);

    procedure Close(var s : StackOfInt);
    begin
        s.top := 0
    end (* Close *);

    procedure Push(var s : StackOfInt; n : integer; var OK : Boolean);
    begin
        if s.top = MaxStack then
            OK := false
        else begin
            s.top          := s.top + 1;
            s.entry[s.top] := n;
            OK             := true
        end
    end (* Push *);

    procedure Pop(var s : StackOfInt; var n : integer; var OK : Boolean);
    begin
        if s.top = 0 then
            OK := false
        else begin
            n     := s.entry[s.top];
            s.top := s.top - 1
        end
    end (* Pop *);

    function Full(s : StackOfInt) : Boolean;
    begin
        Full := s.top = MaxStack
    end (* Full *);

    function Empty(s : StackOfInt) : Boolean;
    begin
        Empty := s.top = 0
    end (* Empty *);



Glava 8.

% Koja opravdava prethodnu glavu i u kojoj se javlja jedna cudna
% rec -- izomorfizam
```

Koja je razlika izmedju primera 1 i 2? U implementaciji tipa. Korisnik ne vidi nikakvu razliku! Ako korisnik ima specifikaciju

```
        Tip: StackOfInt

        Open(s) - inicijalizuje stek s; mora se pozvati pre prve
                upotrebe promenljive s
        Close(s) - deinicijalizuje stek s; mora se pozvati kada s vise
                nije potrebna i vise se nece koristiti
        Push(s, n, ok) - na stek s stavi broj n i vrati ok = true; ako to
                nije moguce, vrati ok = false
        Pop(s, n, ok) - sa steka s skine broj n i vrati ok = true; ako to
                nije moguce, vrati ok = false
        Full(s) - vrati true akko na s vise nema mesta
        Empty(s) - vrati true akko je s prazan
```

potpuno mu je nebitno sta se desava iza toga. Njegov program koji koristi stek ce raditi savrseno bez obzira na konkretne procedure koje rade u pozadini. Stavise, ako je prvo koristio paket iz primera 1, a onda bio primoran da koristi paket iz primera 2, njegov program se nece promeniti ni za liniju!!!

Tajna moci ADT lezi u tome da su svi apstraktni tipovi koji imaju istu specifikaciju izomorfni (u algebarskom smislu). Naime, specifikacija se moze opisati identitetima (tj. jednacinama), a svaki tip (tj. algebarska struktura) koja zadovoljava te identitete je izomorfan sa svakim drugim takvim tipom.

Ideja iza izomorfizama je upravo nasa mantra: izomorfne strukture u sustini predstavljaju jednu istu stvar. Razlika je jedino u nacinu realizacije te jedne stvari!

Glava 9.

% Koja kazuje zasto je dobro raditi sa ADT

Upotreba ADT ima nekoliko prednosti nad obicnim (jako konkretnim) programiranjem.

Prvo, programer ne mora da razmislja o implementaciji jedne stvari dok implementira drugu. Tako su njegovi mozdani talasi fokusirani samo na jedno. Implementaciju pomocnih objekata slobodno moze da zaboravi i da ih koristi znajuci samo njihove apstraktne osobine.

Drugo, na taj nacin se jednostavnije radi timski. Svaki clan tima dobije posao da implementira jedan ADT. On to uradi, spakuje u modul i distribuira ostalima u timu samo specifikaciju svog ADT. Ako se ostali u timu strogo pridrzavaju specifikacije, implementator garantuje da ce to da radi. Takodje, ostali ne moraju da se opterecuju detaljima o implementaciji tudjeg ADT, jer imaju i svog posla preko glave.

Trece, kada onaj ko je implementirao ADT nadje bolje i brze algoritme da uradi svoj posao, slobodno moze da menja kood po svom modulu, da doteruje i ubrzava, sve dotle dok ne menja specifikaciju. Izmene koje on vrsi uopste ne uticu na ostale u timu (jedino mogu biti prijatno iznenadjeni kada vide da njihov program odjedared radi brze ;)

Glava 10.

% U kojoj se vidi kako se ADT lepo slaze sa modularnom strukturom
% jezika i zasto je dobro sakrivati ponesto; takodje, ovo je kraj
% segmenta o ADT.

ADT se jako lepo slaze sa modularnom strukturom jezika. Uglavnom se jedan ADT implementira kao jedan modul. Definicioni deo se razdeli potencijalnim korisnicima kao specifikacija tipa, a u implementacionom se tip realizuje.

Efikasna upotreba apstraktnih tipova podataka se bazira na koriscenju specifikacije, bez virenja u implementaciju tipa. Ako korisnik makar malko cacne po implementaciji, cela ideja propada.

Praksa je pokazala da kaogod sto je decu nemoguce odvici od cackanja nosa, tako je programere nemoguce odvici od cackanja po implementaciji tipa. Uvek se nadje neki mracni kutak svesti iz koga iskoci ideja:

A zasto bih ja ovu trivijalnu stvar radio pozivom procedure (i tako gubio vreme), kada je mnogo brze reci a.struct[x]^.left := nil; Sada cu ja to caskom, da niko ne vidi.

Zato je za pojam ADT neraskidivo vezan pojam sakrivanja informacije (information hiding). Programeru se da ime tipa i paket procedura. Implementacija tipa se SAKRIJE da je on ne vidi. Cak i ako pozeli da ceprka po strukturi, jezik treba to da mu zabrani. Modula-2 ima direktnu podrsku za implementiranje ADT. Mehanizam se zove Opaque Types. Evo primera steka na Moduli-2 koji je implementiran kao ADT:

```
DEFINITION MODULE StackOfInt;

    TYPE STACK;
    (* uocite kako je implementacija *fizicki* sakrivena od
     * korisnika ADT; Korisnik dobija samo .DEF datoteku, pa da
     * ga vidim hoce li uspeti da ceprka po mojim stekovima,
     * ili ce da programira k'o sav normalan svet!  ;] *)

    PROCEDURE Open(VAR s : STACK);
    PROCEDURE Close(VAR s : STACK);

    PROCEDURE Push(VAR s : STACK; n : INTEGER; VAR OK : BOOLEAN);
    PROCEDURE Pop(VAR s : STACK; VAR n : INTEGER; VAR OK : BOOLEAN);

    PROCEDURE Full(s : STACK) : BOOLEAN;
    PROCEDURE Empty(s : STACK) : BOOLEAN;

END (* DEFINITION *) StackOfInt.
```

Tip se u potpunost definise u implementacionom modulu, koji, po pretpostavci, nije dostupan korisniku (a i da jeste, komapjler ce mu zabraniti da uradi bilo sta sa promenljivom tipa STACK, osim da pozove neku od navedenih procedura).

Glava 11.

% Mali rezime i odgovor na pitanje zasto smo se toliko time bavili u
% prici o OOP

Do sada smo videli sta je tip podataka, sta je struktura podataka i kako se od strukture napravi tip.

Videli smo kako iz toga naraste ideja ADT i dve osnovne stvari koje se vezuju za ADT:

1. apstrahovanje (data abstraction)
2. skrivanje podataka (information hiding)

Takodje je napomenuto da se ADT jako lepo slaze sa modularnim jezicima. To ne znaci da ideje ADT ne mozete koristiti na nekom jeziku koji nema mehanizam modula i nema mehanizam za skrivanje (npr, Pascal). Pri radu sa takvim jezicima treba da pokazete vise samodiscipline.

Ideja ADT je beskonacno vazna. To je *osnovna* ideja OOP. Sve ostalo o cemu cemo pricati je vazno, ali je manje apstraktno.

Glava 12.

% U kojoj pocinje ispucavanje OO folklora: objekti i metodi.

Objekt je paket podataka koji ima svoj tajni zivot. Mozete ga zamisliti kao jednu skupinu surovih podataka i procedura kojoj je nadenuto ime (to sto skupina ima ime je jako vazno!).

Procedure koje su deo objekta se zovu metodi (to su metodi za menjanje stanja objekta), dok surovi podaci opisuju (tekuce) stanje objekta.

Na primer, opis objekta u nekom pseudojeziku moze da izgleda ovako:

```
var matrix :
    object
        entry : array of array of real;
        m, n  : cardinal;
    methods
        init(p, q : cardinal) is
        begin
            new(entry, p, q);
            m := p;
            n := q
        end init;

        free() is
        begin
            dispose(entry, m, n);
            m := 0;
            n := 0
        end free;

        read() is
            i, j : cardinal;
        begin
            if null(entry) then
                ERROR("Uninitialised matrix")
            else
                for i := 1 to m do
                    for j := 1 to n do
                        ReadReal(entry[i, j])
                end end
        end end read;

        write() is
            (* slicno kao read *)
        end write
    end object;
```

Ovaj objekt predstavlja jednu matricu koja moze da uradi cetiri stvari: da se inicijalizuje, da se samoubije, da se ucita i da se ispise.

Glava 13.

% Poruke i metodi

Poruka je nesto kao poziv procedure (za one koji su ucili logiku: poruka je term!). Ona nema znacenje dok se ne uputi nekom objektu. To je samo niz znakova.

Kada se objektu uputi poruka, on medju svojim metodima potrazi onaj koji ima isto ime i signaturu (signatura je struktura argumenata) kao sto se zahteva u poruci. Ako ne uspe, objekt odbaci poruku (i tako izazove gresku, ili nesto slicno).

Ako objekt uspe da nadje odgovarajuci metod medju metodima kojima raspolaze, primeni taj metod na sebi, sto moze da dovede do promene stanja objekta (izmene se vrednosti nekih promenljivih) i do upucivanja novih poruka novim objektima. (Uputiti poruku objektu znaci (u terminima logike) pokusati interpretirati odgovarajuci term.)

Objekt kome se upucuje poruka zove se primalac (receiver), sto je dosta logicno.

Na primer, ako imamo objekt matrix kao u prethodnom primeru:

```
    var matrix :
        object
            entry : array of array of real;
            m, n  : cardinal;
        methods
            init(p, q : cardinal) is begin ... end init;
            free() is begin ... end free;
            read() is ... begin ... end read;
            write() is ... begin ... end write
        end object;
```

tada

```
    matrix.init(9, 10)
```

predstavlja poruku init(9, 10) upucenu objektu matrix. Kako objekt poseduje metod koji se zove init i prima dva argumenta tipa cardinal, to ce objekt interpretirati poruku na odgovarajuci nacin, tj. aktivirace metod init sa argumentima 9 i 10. Aktivacija metoda init ce dovesti do promene stanja objekta: bice alociran prostor za elemente matrice i bice popunjene vrednosti za m i n.

Glava 14.

% Polimorfizam

Razliciti objekti mogu imati metode sa istim imenom. Ako uputimo poruku jednom objektu, on ce je interpretirati koristeci svoje metode. No, tu istu poruku drugi objekt moze interpretirati na sasvim drugi nacin, jer ce on koristiti svoje metode (metodi se zovu isto, ali rade sasvim drugacije stvari).

Ova osobina se zove polimorfizam.

Primer: posmatrajmo objekte TraficLight (koji modeluje semafor), i PrimeGen (koji je generator prostih brojeva):

```
    var TraficLight :
        object
            light : (red, red-yellow, yellow, green);
        methods
            init() is
            begin
                light := red
            end init;

            next() is
            begin
                case light of
                | red        :  light := red-yellow;
                | red-yellow :  light := green;
                | yellow     :  light := red;
                | green      :  light := yellow
                end
            end next;

            write() is
            begin
                case light of
                | red        :  WriteStr("red");
                | red-yellow :  WriteStr("red & yellow");
                | yellow     :  WriteStr("yellow");
                | green      :  WriteStr("green")
                end
            end write
        end object;

    var PrimeGen :
        object
            p : cardinal;
        methods
            init() is
            begin
```

```
            p := 3
        end init;

        next() is
        begin
            repeat
                p := p + 2
            until Prime(p)
        end next;

        write() is
        begin
            WriteCard(p)
        end write
    end object;
```

Uocimo da oba objekta imaju tri metoda: init, next, write. Kada kazemo

```
TraficLight.init; PrimeGen.init;
```

jednu istu poruku smo uputili i jednom i drugom objektu. Razlika je u tome sto ce poruka init u prvom pozivu aktivirati metod init objekta TraficLight, dok ce poruka init u drugom pozivu aktivirati metod init objekta PrimeGen. Zato, nakon

```
TraficLight.init;
TraficLight.next;
TraficLight.next;
TraficLight.next;
TraficLight.write;
```

dobijamo string "yellow" na ekranu, dok nakon

```
PrimeGen.init;
PrimeGen.next;
PrimeGen.next;
PrimeGen.next;
PrimeGen.write;
```

dobijamo broj 11 na ekranu. Ova dva objekta su na iste poruke reagovali razlicito, jer su ih interpretirali svako na svoj nacin. Jedna ista poruka upucena razlicitim objektima dovodi do razlicitog ponasanja.

Glava 15.

% Koja govori o jednom sinonimu -> encapsulation

Svaki objekt koji drzi do sebe mora svoju unutrasnjost drzati daleko od prljavih prstiju okolnih korisnika (ili, klijenata). Zato onaj deo objekta koji opisuje strukturu promenljivih koje cuvaju stanje objekta ne sme biti dostupan klijentima.

Ako neko zeli da proveri i/ili da promeni stanje objekta, mora to to uciniti upucivanjem poruke objektu. Na taj nacin objekt ima potpunu kontrolu nad samim sobom i moze da stiti svoj integritet (u krajnjem slucaju, metod koji proverava/menja stanje objekta moze na neki nacin proveriti ko je poslao poruku i, ukoliko posiljalac nije klijent od poverenja, moze odbiti da otkrije/izmeni svoje stanje).

To je u skladu sa opstom idejom o skrivanju podataka (information hiding) i zove se jos i encapsulation (ucaurenost). Podaci su ucaureni u objektu i do njih se ne moze doci lako.

Glava 16.

% U kojoj se izgradi najjednostavniji objektni model ---
% --- programiranje bazirano na objektima (object-based programming)

Kada imamo programski jezik koji podrzava objekte (kao nezavisne entitete), ucaurenost, mehanizam poruka i polimorfizam, dobijamo najjednostavniji oblik objektnog programiranja: programiranje bazirano na objektima (object-based programming).

U nekim slucajevima to je sasvim prihvatljiv model. Najpopularnija verzija object-based modela je actor model (objekti se zovu ``glumci'' [actors], a metodi kojima su naoruzani su njihov tekst [script]; programski jezik koji implementira ovu ideju se zove Act2).

Glava 17.

% Koja objasnjava pojam klase i pojam instance klase

Kao sto smo videli u modelu baziranom na objektima (object-based model), svaki objekt za sebe definise sopstveno ponasanje. Kada objekata iste vrste ima mnogo, to znaci da mnogo puta trenba uraditi jednu istu stvar. Zato se uvodi pojam klase. Klasa se moze shvatiti kao opis jedne vrste objekata. Klasa opisuje zajednicko ponasanje svih objekata koji pripadaju toj klasi.

Objekti koji pripadaju nekoj klasi zovu se instance te klase.

Klasa opisuje koja polja i koje metode ce imati instance te klase. Umesto da za svaki objekt posebno opisujemo strukturu polja i metoda, to opisemo u klasi, pa onda samo kazemo da je objekt O instanca klase. Prevodilac na osnovu toga zna kako izgleda struktura objekta.

Glava 18.

% U kojoj se daje kompletan primer jedne klase

PRIMER: klasa koja opisuje realne matrice

```
    class RealMatrix

    export  init, dim1, dim2, read, write, at, put,
            mulScal, add, sub, mul;
            (*
             * ovo su imena metoda koji cine specifikaciju ponasanja
             * instanci klase
             *)

    instance variables
        entry : array of array of real;
        m, n  : cardinal;

    instance methods
        (*
         * inicijalizuje objekt tako sto postavi format matrice i
         * aliocira prostor
         *)
        init(p, q : cardinal) is
        begin
            new(entry, p, q); m := p; n := q
        end init;

        (* prva i druga dimenzija matrice *)

        dim1() : cardinal is
        begin
            return m
        end dim1;

        dim2() : cardinal is
        begin
            return n
        end dim2;
```

```
(* ucitavanje i ispis matrice *)

read() is
    i, j : cardinal;
begin
    if null(entry) then ERROR("Uninitialised matrix")
    else
        for i := 1 to m do
            for j := 1 to n do
                ReadReal(entry[i, j])
        end end
end end read;

write() is
    i, j : cardinal;
begin
    if null(entry) then ERROR("Uninitialised matrix")
    else
        for i := 1 to m do
            for j := 1 to n do
                WriteReal(entry[i, j])
        end end
end end write;

(*
 * vrati element matrice koji se nalazi na datoj koordinati
 *)
at(p, q : cardinal) : real is
begin
    if(1 =< p =< m) and (1 =< q =< n) then return entry[p, q]
    else ERROR("Invalid index")
end end at;

(*
 * na datu koordinatu u matrici stavi datu vrednost
 *)
put(p, q : cardinal; val : real) is
begin
    if(1 =< p =< m) and (1 =< q =< n) then entry[p, q] := val
    else ERROR("Invalid index")
end end put;


(*
 * mnozenje matrice datim brojem
 *)
mulScal(x : real) : RealMatrix is
    i, j : cardinal;
    res  : RealMatrix;
begin
    res.init(m, n);
    for i := 1 to m do
        for j := i to n do
            res.put(i, j, x * entry[i,j])
    end end;
    return res
end mulScal;

(* zbir i razlika dve matrice *)

add(q : RealMatrix) : RealMatrix is
    i, j : cardinal;
    res  : RealMatrix;
begin
    if (m = q.dim1) and (n = q.dim2) then
        res.init(m, n);
        for i := 1 to m do
            for j := 1 to n do
                res.put(i, j, entry[i, j] + q.at(i, j))
        end end;
        return res
    else ERROR("Attempt to add incompatible matrices")
end end add;
```

```
        sub(q : RealMatrix) : RealMatrix is
            i, j : cardinal;
            res  : RealMatrix;
        begin
            if (m = q.dim1) and (n = q.dim2) then
                res.init(m, n);
                for i := 1 to m do
                    for j := 1 to n do
                        res.put(i, j, entry[i, j] - q.at(i, j))
                end end;
                return res
            else ERROR("Attempt to add incompatible matrices")
        end end sub;

        (*
         * proizvod dve matrice (receiver * q)
         *)
        mul(q : RealMatrix) : RealMatrix is
            res     : RealMatrix;
            r       : real;
            i, j, k : cardinal;
        begin
            if n = q.dim1 then
                res.init(m, q.dim2);
                for i := 1 to m do
                    for j := 1 to q.dim2 do
                        r := 0;
                        for k := 1 to n do
                            r := r + entry[i, k] * q.at(k, j)
                        end;
                        res.put(i, j, r)
                end end;
                return res
            else
                ERROR("Attempt to multiply incompatible matrices")
            end
        end mul;

    end RealMatrix.
```

Glava 19.

% Primer objekata koji pripadaju klasi RealMatrix

Posmatrajmo klasu RealMatrix iz prethodnog primera. Nakon deklaracija:

```
    var a : RealMatrix;
    var b : RealMatrix;
```

``a'' i ``b'' su instance klase RealMatrix. To znaci da oba imaju polja ``entry'', ``m'', i ``n'' (koja su privatna stvar) i imaju metode init, dim1, dim2, read, write, at, put, mulScal, add, sub, mul.

Evo i primera upotrebe objekata ``a'' i ``b'':

```
    begin
        a.init(10, 10); a.read;
        b := a.mul(a);              (* tj. b := a * a *)
        a := b.mulScal(2.0);        (* tj. a := 2 * b *)
        ...
        WriteReal(a.at(3,3));       (* ispise a[3,3] *)
        ...
    end.
```

Glava 20.

% Koja prica o jos nekim stvarima koje mogu da rade klase:
% class methods & class variables

Klasa opisuje strukturu i ponasanje svojih instanci. No, ima programskih jezika u kojima i klasa moze imati svoj privatni zivot. Drugim recima, i klasa moze imati svoje promenljive i svoje metode.

Promenljive koje pripadaju klasi (class variables) sluze za to da klasa drzi neke svoje informacije (recimo, koliko instanci te klase je do sada napravljeno, ...). Te informacije nisu nikome dostupne direktno. One se mogu korisitit/menjati iskljucivo putem poruka upucenih klasi. Poruke upucene klasi se interpretiraju na isti nacin kao poruke upucene objektu. Jedina razlika je u tome sto poruke upucene klasi traze metode iz skupa privatnih metoda klase (class methods).

Smalltalk, recimo, ima malo drugaciji pristup. Tu su promenljive koje pripadaju klasi u stvari globalne promenljive koje koriste instance te klase i to za medjusobnu komunikaciju. Promenljive koje pripadaju klasi u Smalltalk-u su dostupne samo instancama te klase i metodima klase.

Jos uvek ne postoji opsteprihvaceno misljenje da li klasa treba da ima pravo na intimni zivot ili ne; da li moze da ima svoju privatnost, ili treba samo da sluzi svojim instancama.

Meni se cini da je dobro kada i klase imaju svoju intimu.

Glava 21.

% U kojoj se izgradjuje novi stepenik u hijerarhiji objektnih
% modela --- programiranje bazirano na klasama (class-based
% programming)

Ako neki programski jezik podrzava klase, objekte kao instance klasa (dakle, ne mogu se opisati objekti sami za sebe, vec se opisuju klase, a objekti se javljaju kao instance klasa), ucaurenost, mehanizam poruka i polimorfizam, dobijamo novi model objektnog programiranja: programiranje bazirano na klasama (class-based programming).

Nestrpljivog citaoca obavestavamo da ce Saga vec u narednom paketu poruka potpuno iskonvergirati ka objektno-orijentisanom modelu. Iskreno se nadamo da nestrpljenje nije zgasnulo zar u srcima s pocetka.

Glava 22.

% U kojoj se vidi kako M2 podrzava object-based programiranje
% (doduse, sa malim zakasnjenjem)

Jedna od lepota M2 je to sto M2 podrzava object-based programiranje. Ideja je da se objekti implementiraju kao bibliotecki moduli: za svaki objekt po jedan bibliotecki modul.

U definicionom delu se nalazi paket procedura koje opisuju ponasanje objekta (metodi), a u implementacionom delu se implementiraju struktura objekta i metodi (koji su navedeni u definicionom delu).

Na primer, objekt koji opisuje stampac moze da bude implementiran u M2 ovako:

```
    DEFINITION MODULE Printer;

    FROM FontSupport     IMPORT Font;
    FROM GraphicsSupport IMPORT Graphics;

        TYPE FontHandle = CARDINAL;
        TYPE MODE       = (draftMode, lqMode, graphMode, myfontMode);

        PROCEDURE Open();
        PROCEDURE Close();
```

```
        PROCEDURE SetModeTo(mode : MODE);
        PROCEDURE ThisMode() : MODE;

        PROCEDURE PrintText(text : ARRAY OF CHAR);
        PROCEDURE PrintGraphics(graph : Graphics);
        PROCEDURE NewLine();
        PROCEDURE LineFeed();
        PROCEDURE FormFeed();

        PROCEDURE Status() : INTEGER;

        PROCEDURE DownloadFont(font : Font) : FontHandle;
        PROCEDURE SelectFont(handle : FontHandle);

END Printer.
```

Njega koristimo u drugim modulima ovako:

```
    IMPORT Printer;
```

pa kada kazemo

```
    Printer.Open;
    Printer.PrintText("Hello, World!");
    Printer.FormFeed;
    Printer.Close;
```

to mozemo shvatiti kao ``proceduralno'' programiranje (tj. kao niz poziva procedura), ali i kao ``objektno'' programiranje (objektu Printer je upuceno nekoliko poruka na koje ce on adekvatno reagovati).

Uocimo da su svi elementi objektno-baziranog programiranja podrzani (apstrakcija, ucaurenost, podrzka objektima kao nezavisnim entitetima).

Glava 23.

% Koja sadrzi jos jednu pricu ispricanu sa zakasnjenjem:
% Zasto dva bajta nisu uvek dva bajta
% (sto je trebalo biti ispricano nakon glave o ADT)

Mnogi inzenjeri (pa i neki nasi studenti), a narocito C programeri, smatraju da su dva bajta uvek dva bajta. Na taj nacin oni uopste ne razlikuju (do na epsilon) tipove, recimo, INTEGER i ARRAY [0 .. 1] OF CHAR. Njima je to isto: dva bajta.

DVA BAJTA NISU UVEK DVA BAJTA! Oni se razlikuju po tome koja im je uloga dodeljena! Dva bajta rezervisana za neki INTEGER imaju sasvim drugaciju ulogu od dva bajta rezervisana za neki ARRAY [0 .. 1] OF CHAR. Zato su oni beskonacno razliciti i niposto ih ne smemo mesati.

Analogija iz zivota bi, priblizno, isla ovako: skola i bolnica mogu imati istu zapreminu (kao zgrade), ali to niposto ne znaci da je to jedno te isto! Razlikuju se po funkciji!

Glava 24.

% U kojoj Saga pocinje malko da lici na pravu sapunsku operu, jer
% cemo poceti da pricamo o nasledjivanju. Srecna okolnost je to
% sto u Sagi niko ne mora da umre kako bi moglo da dodje do
% nasledjivanja.

Osnovna odlika objektno-orijentisanih sistema je nasledjivanje. Da se ne bismo upustali u veliko filozofiranje i puno zvucnih fraza, evo prvo jednog primera. Posmatrajmo klasu RealMatrix (od ranije):

```
    class RealMatrix
    export  init, dim1, dim2, read, write, at, put,
            mulScal, add, sub, mul;
```

```
    instance variables
        entry : array of array of real;
        m, n  : cardinal;

    instance methods
        init(p, q : cardinal)            is ...;
        dim1() : cardinal                is ...;
        dim2() : cardinal                is ...;
        read()                           is ...;
        write()                          is ...;
        at(p, q : cardinal) : real       is ...;
        put(p, q : cardinal; val : real) is ...;
        mulScal(x : real) : RealMatrix   is ...;
        add(q : RealMatrix) : RealMatrix is ...;
        sub(q : RealMatrix) : RealMatrix is ...;
        mul(q : RealMatrix) : RealMatrix is ...;
    end RealMatrix.
```

Zeleli bismo da naravimo klasu SquareRealMatrix koja ce predstavljati kvadratne realne matrice.

Jedan nacin je da uzmemo klasu RealMatrix i da je malo preradimo. Na taj nacin dobijamo dve klase kod kojih je 90% kooda identicno. Gubimo vreme i mesto na disku.

Zar ne bi bilo lepo kada bismo mogli da kazemo da je klasa SquareRealMatrix do na epsilon jednaka sa klasom RealMatrix, pa da dopisemo samo razliku?

O, da. Ne samo da bi bilo lepo, nego je to i moguce! Mehanizam koji nam to omogucuje zove se nasledjivanje. Napravicemo zeljenu klasu i, umesto da prekucavamo kood iz klase RealMatrix, reci cemo da je nova klasa podklasa klase RealMatrix. Tada ce ona naslediti sve osobine klase RealMatrix, a mi cemo samo dopisati nove (specificne) metode i redefinisati neke stare. Recimo, ovako:

```
    class SquareRealMatrix

    inherits RealMatrix;
        (* ova deklaracija znaci da klasa SqareRealaMatrix nasledjuje
         * sve osobine klase RealMatrix *)

    export init, det, inv;
        (* sve metode koje je klasa RealMatrix nudila okruzenju nova
         * klasa nudi po automatizmu; jedino sto nova klasa jos treba
         * da izveze su novi metodi ``det'' i ``inv'', i redefinisani
         * metod ``init'' *)

    (* nova klasa preuzima implementaciju od stare klase i,
     * ukoliko je to potrebno, dodaje neke specificne. U ovom slucaju,
     * nema potrebe da se dodaju nove promenljive *)

    instance methods

        init(p : cardinal) is
        begin
            self.^init(p, p)
        end init;
            (* klasa SquareRealMatrix ima novu proceduru za inicijali-
             * zaciju koja je karakteristicna za nju. Ona radi tako
             * sto objektu koji je primio poruku init(p) posalje
             * metod init(p, p) iz nadklase. (self je promenljiva u
             * koju se smesti primalac poruke; poruka oblika
             * o.^m(a) znaci da objektu o treba uputiti poruku m(a),
             * ali implementaciju potraziti tek u njegovoj nadklasi.
             * Na ovaj nacin smo u podklasi redefinisali proceduru
             * iz nadklase *)

        det() : real is
        begin
            (* procedura koja racuna determinantu matrice *)
        end det;
```

```
        inv() : SquareRealMatrix is
        begin
            (* procedura koja invertuje matricu *)
        end inv;

    end SquareRealMatrix.
```

Glava 25.

% Jos o nasledjivanju, kao i jedan popularan termin: code reuse

Nasledjivanje je relacija medju klasama. Ono omogucuje da se definicija i implementacija jedne klase bazira na definiciji i implementaciji neke vec postojece klase. Na taj ncin se stedi vreme, novac, ali i mesto na disku.

Prilikom nasledjivanja podklasa nasledjuje i metode i strukturu objekta od svoje nadklase. Cinjenica da podklasa nasledjuje metode znaci da njih ne treba pisati ponovo. Treba dopisati samo nove stvari, kao i one koje se razlikuju. Ova osobina se zove ``code reuse'' (jer podklasa koristi kood iz nadklase).

Pri nasledjivanju, situacija iz nadklase moze malo da se izmeni, tako da mozemo razlikovati tri ``tradicionalne'' upotrebe nasledjivanja:

- Specijalizacija. To je situacija u kojoj podklasa doda neke nove instance variables i eventualno jos neke metode. Primer:

```
class PlaneFigure
instance variables
    centerX, centerY : cardinal;
    name             : string;
    colour           : Colours;
instance methods
    setCenter(x, y : cardinal)     is ...;
    setName(newName : string)      is ...;
    setColour(newColour : Colours) is ...;
    move(dx, dy : cardinal)        is ...;
end PlaneFigure.

class Circle
inherits PlaneFigure;
instance variables
    radius : cardinal;
instance methods
    draw() is ...;
    wipe() is ...;
end Circle.
```

  Klasa Circle je iz klase PlaneFigure nasledila strukturu objekta (tj. promenljive centerX, centerY, name, colour) i sve metode, i tome jos dodala promenljivu radius i dva nova metoda. Uocimo da klasa Circle nista nije menjala od onoga sto je nasledila.

- Prosirenje funkcionalnosti klase. To je situacija u kojoj podklasa ne dodaje nove instance variables, vec samo doda nove metode. Primer:

```
class Menu
instance variables
    menuBar : list of SubMenu;
instance methods
    create()                   is ...;
    remove()                   is ...;
    display()                  is ...;
    clear()                    is ...;
    getSellection() : cardinal is ...;
end Menu.

class WeirdMenu
```

```
            inherits Menu
            instance methods
                displayEvery(milliseconds : longcard) is ...;
                displayRandomly()                     is ...;
                refuseSellection() : cardinal         is ...;
                monkeyWithUser()                      is ...;
            end WeirdMenu.
```

- Modifikacija. To je situacija u kojoj podklasa redefinise neke (ili sve) metode koje je nasledila iz nadklase. Primer smo videli kod klase SquareRealMatrix gde je mnogo metoda preuzeto iz nadklase, ali je jedan (metod init) redefinisan.

Glava 26.

% Vrste nasledjivanja

Nasledjivanje moze biti jednostruko i visestruko. Jednostruko nasledjivanje je ono kod koga klasa moze da nasledi osobine najvise jedne klase. Kod visestrukog nasledjivanja klasa moze da nasledi osobine nekoliko klasa.

Primeri koje smo do sada videli su bazirani na modelu jednostrukog nasledjivanja. Primere visestrukog nasledjivanja cemo videti malo kasnije. Razlog za to je relativna slozenost modela sa visestrukim nasledjivanjem. Ima dosta tehnikalija koje treba pomenuti da bi se dobila potpuna slika.

Jos uvek se vodi diskusija o tome da li je dovoljno da programski jezik podrzava jednostruko nasledjivanje, ili je mnogo bolje da podrzava visestruko. I jedan i drugi model imaju i prednosti i mane.

Glava 27.

% U kojoj se klase uredjuju u hijerarhiju i u kojoj se uvodi
% jedna oznaka.

Kao sto smo pomenuli, nasledjivanje je relacija medju klasama. Ona uredjuje klase u stablo (u sistemima sa jednostrukim nasledjivanjem; kod sistema sa visestrukim nasledjivanjem, dobija se aciklican digraf), dakle, hijerarhijsku strukturu.

Ako je klasa C1 podklasa klase C0, onda cemo to oznaciti sa C0 > C1, ili C1 < C0. Podklasa moze biti direktna ili indirektna. Nezavisno od toga koristimo oznaku ``<''. Na primer, ako je C < C2 < C1 < C0 niz klasa koje su jedna drugoj direktne podklase, slobodno mozemo reci da je C < C0 jer je C doista podklasa od C0, mada ne direktna.

Glava 28.

% U kojoj se prica o tome kako se tumaci poruka upucena objektu
% u OO sistemima sa jednostrukim nasledjivanjem.

Neka je X instanca klase C i neka je C < C2 < C1 < C0. Ako objektu X uputimo poruku m(a):

```
X.m(a)
```

potrebno je naci implementaciju odgovarajuceg metoda. Prvo se pregleda klasa C. Ako u njoj postoji metod sa imenom ``m'', onda se poruka m(a) protumaci kao poziv metoda ``m'' iz klase C.

No, ako u klasi C ne postoji metod sa imenom ``m'', objekt nece odbaciti poruku, vec ce proveriti da li u nekoj od nadklasa postoji odgovarajuci metod. Potraga za metodom se nastavlja u njegovoj nadklasi (to je, u ovom primeru, klasa C2). Ako u klasi C2 postoji

metod koji se zove ``m'', onda je sve OK. Poruka X.m(a) se protumaci kao primena metoda ``m'' iz klase C2 na objekt X.

U suprotnom se proverava da li u klasi C1 postoji metod ``m'', itd... Potraga se nastavlja sve dok se ne pronadje odgovarajuci metod, ili dok ne stignemo do vrha hijerarhije. Ako nigde nema zeljenog nam metoda, on se odbacuje (uz odgovarajuce posledice).

Glava 29.

% Jos jedan pomodan termin: assignment compatibility

Jedna od lepih (ali i zapetljanijih) stvari vezanih za nasledjivanje je da se promenljivoj koja je deklarisana kao promenljiva klase C1 moze dodeliti objekt bilo koje klase C2 za koju je C2 < C1.

Na primer, ako imamo da je

```
    var a : PlaneFigure; (* videti rethodne poruke *)
    var b : Circle;
    var m : RealMatrix;
```

onda je dodela

```
    a := b
```

ispravna, jer je Circle podklasa klase PlaneFigure, dok dodela

```
    a := m
```

nije ispravna, jer RealMatrix nije podklasa klase PlaneFigure. Ovo se strucno kaze da je Circle ``assignment compatible'' sa PlaneFigure, ali da RealMatrix to nije. Takodje, kazemo da je ``b'' ``assignment compatible'' sa ``a'', ali da ``m'' to nije.

Glava 30.

% Koja je malko surova jer opisuje kako se metodi vezuju. Srecom,
% to ih ne boli. Odgovarajuci pomodni termin je dinamicko
% vezivanje (dynamic binding). A pricacemo i o statickom vezivanju.

Kada imamo promenljivu koja je deklarisana kao promenljiva klase C1, njoj mozemo dodeliti objekt bilo koje klase C2 za koju je C2 < C1. To ima interesantne posledice na tumacenje poruke koja je upucena promenljivoj klase C1.

Tumacenje poruka se jos zove i ``vezvanje metoda'' jer se odgovarajuci metod ``veze'' za poruku (binding). Dinamicko vezivanje metoda znaci da se metodi vezuju za poruke u toku izvrsavanja programa, a ne u toku kompilacije. Razlog je krajnje jednostavan: u toku kompilacije niko ziv ne zna kojoj klasi ce pripadati objekt kome se upucuje poruka. Zato se nikada ne zna odakle treba poceti potragu za onim pravim metodom.

Evo primera. Neka je data klasa PlaneFigure (kao ranije) i njene podklase Circle, Rectagle, Triangle:

```
    class PlaneFigure
    instance variables
        centerX, centerY : cardinal;
        name             : string;
        colour           : Colours;
    instance methods
        setCenter(x, y : cardinal)     is ...;
        setName(newName : string)      is ...;
        setColour(newColour : Colours) is ...;
        move(dx, dy : cardinal)        is ...;
    end PlaneFigure.

    class Circle
    inherits PlaneFigure;
```

```
    instance variables
        radius : cardinal;
    instance methods
        draw() is ...;
        wipe() is ...;
    end Circle.

    class Rectangle
    inherits PlaneFigure;
    instance variables
        width, height : cardinal;
    instance methods
        draw() is ...;
        wipe() is ...;
    end Rectangle.

    class Triangle
    inherits PlaneFigure;
    instance variables
        side1, side2, side3 : cardinal;
    instance methods
        draw() is ...;
        wipe() is ...;
    end Triangle.
```

Posmatrajmo ovakve deklaracije:

```
    var pf : PlaneFigure;
    var c  : Circle;
    var r  : Rectangle;
    var t  : Triangle;
```

Pretpostavimo da smo inicijalizovali promenljive ``c'', ``r'' i ``t''. Sada je ``c'' neki krug, ``r'' je neki pravougaonik, a ``t'' je neki trougao. Ako kazemo

```
     c.draw
```

jasno je da ce biti pozvan metod ``draw'' iz klase Circle. No sta se desava ako kazemo

```
    pf := c;
    pf.draw;  ?
```

Ideja dinamickog vezivanja je u tome da run-time sistem proveri sta se nalazi u promenljivoj ``pf''. Tada, kada sazna kojoj podklasi klase PlaneFigure pripada sadrzaj promenljive ``pf'', pocinje da trazi metod u odgovarajucoj klasi. Dakle, nakon prethodne sekvence opet ce biti aktiviran metod ``draw'' iz klase Circle. Slicno, naredna sekvenca poziva metode ``draw'' iz sve tri podklase klase PlaneFigure, redom:

```
    pf := c; pf.draw;   (* pozove metod iz klase Circle    *)
    pf := r; pf.draw;   (* pozove metod iz klase Rectangle *)
    pf := t; pf.draw;   (* pozove metod iz klase Triangle  *)
```

Jasno je da se u toku kompilacije nikako ne moze odrediti sta ce biti sadrzaj promenljive ``pf'', pa tako do samog kraja ostaje neizvesno u kojoj klasi treba poceti potragu za metodom ``draw''.

S druge strane, sistem koji podrzava staticko vezivanje bi jednostavno potrazio metod ``draw'' u klasi PlaneFigure. Kako tamo nema metoda sa imenom ``draw'' i kako PlaneFigure nema nadklasu, sistem sa statickim vezivanjem bi (neopravdano) prijavio gresku.

Glava 31.

```
% U kojoj se daje primer za dinamicko vezivanje metoda:
% heterogeni nizovi
```

Dinamicko vezivanje je jako zgodna stvar. Sada cemo demonstrirati jednu interesantnu primenu. Pozmatrajmo niz ravanskih figura:

```
    var fig : array [1 .. Max] of PlaneFigure;
    var N   : cardinal; (* broj figura u nizu *)
```

Prica o assignment compatibility nam daje da element niza ``fig'' moze biti objekt bilo koje od klasa: Circle, Rectangle, Triangle. Pretpostavimo da smo na neki nacin popunili elemente niza ``fig'' raznim krugovima, pravougaonicima i/ili trouglovima. To se, na primer, moze uciniti ovako:

```
    N := 0;
    repeat
        inc(N);
        WhichFigureToRead(kind);
        case kind of
        |   rect : ReadRectangle(fig[N]);
        |   circ : ReadCircle(fig[N]);
        |   tri  : ReadTriangle(fig[N]);
        else
            PANIC
        end
    until NoMoreDesireToEnterFigures;
```

Ako treba iscrtati svih N figura, ovo je najelegantniji nacin da se to ucini:

```
    for i := 1 to N do
        fig[i].draw
    end;
```

Zbog price o dinamickom vezivanju, run-time sistem prvo proveri sta se nalazi u promenljivoj fig[i], pa onda odluci od koje klase da pocne pretrazivanje. Na taj nacin ce za svaku figuru biti pozvan metod ``draw'' bas iz njene klase (na trouglove ce biti primenjen metod ``draw'' iz klase Triangle, na krugove metod iz klase Circle, a na pravougaonike metod iz klase Rectangle).

Ako treba pomeriti celu konstelaciju figura za vektor (DX, DY), to se moze uciniti ovako:

```
    for i := 1 to N do
        fig[i].move(DX, DY)
    end;
```

Uocimo da je ``move'' metod iz klase PlaneFigure. On nije implementiran ni u jednoj od podklasa koje koristimo, ali nasledjivanje obezbedjuje da on ipak bude pronadjen i primenjen kako treba. Uocimo da se i u ovom slucaju primenjuje dinamicko vezivanje metoda! Drugim recima, tek run-time sistem odredjuje tumacenje poruke ``move(DX, DY)''. Razlog je krajnje jednostavan: u toku kompilacije ne mozemo biti sigurni da metod ``move'' nije redefinisan u nekoj od podklasa! U ovom primeru metod ``move'' nije bio redefinisan ni u jednoj podklasi, pa je pozvan onaj zajednicki iz klase PlaneFigure.

Da smo kojim slucajem, recimo, u klasi Rectangle redefinisali metod ``move'', na sve pravougaonike u listi bi bio primenjen njihov metod ``move'', dok bi na ostale bio primenjen zajednicki metod ``move'' (tj. onaj iz klase PlaneFigure).

Glava 32.

% Eto, jos ni stota poruka, a vec cemo definisati pojam
% objektno-orijentisanog programirnja ;]

Ako neki programski jezik podrzava klase, objekte kao instance klasa (dakle, ne mogu se opisati objekti sami za sebe, vec se opisuju klase, a objekti se javljaju kao instance klasa), ucaurenost, nasledjivanje, mehanizam poruka, polimorfizam i dinamicko vezivanje, to je objektno-orijentisani jezik!

Nasledjivanje moze biti i jednostruko i visestruko. To ne utice

na objektnu orijentisanost jezika. Vazno je da jezik podrzava neki model nasledjivanja.

Takodje, neki objektni jezici podrzavaju jedino staticko vezivanje metoda. To ne znaci da oni nisu objektni jezici, vec samo da imaju manju izrazajnu moc.

Ovim je zavrsen osnovni deo Sage o OO jezicima i varijetetima cija je namena bila da ustanovi definicije. U nastavku Sage ce se govoriti o nekim naprednim tehnikama koje jesu bitne za OO, ali nisu ono sto definise OO. Stvari o kojima cemo govoriti uglavnom spadaju u tehnike implementacije OO jezika, ili su posledica implementacije.

Saga o OOP, Drugi deo : Implementacije

Dragan Masulovic
(u sklopu Vannastavnih aktivnosti iz programiranja, PMF-Novi Sad)

Uvod

Prvi deo Sage je uveo osnovne pojmove OOP i demonstrirao ih na primerima. Sada cemo malo pricati o tehnikama implementacije OO programskih jezika i svim dosetkama i zackoljicama koje je potrebno imati da bi se od ideje moglo preci na nesto sto je upotrebljivo. Tu spadaju konstruktori i destruktori, genericke klase, virtuelni metodi ... Dakle, sve ono o cemu ljudi glasno govore kada zele da se spontano procuje u drustvu da su oni magovi OOP-a.

Treba sve vreme imati na umu da ono o cemu ce biti reci u ovom delu Sage _nije_ono_sto_karakterise_OOP_, vec je samo posledica odluka u dizajniranju OO jezika i implementacije njihovih prevodilaca.

Pri kraju Sage cemo reci i nekoliko reci o visestrukom nasledjivanju, njegovim prednostima i problemima koji se javljaju prilikom njegove implementacije.

Glava 1.

% U kojoj se vidi da se bez mnogih stvari moze

Neki OO jezici se prevode, a neki se interpretiraju. Neki podrzavaju strong typing i imaju staticku proveru tipova, a neki su tzv. typeless jezici, tj. jezici kod kojih se provera tipova vrsi u toku izvrsenja programa (run-time).

Sve tehnike o kojima cemo govoriti u ovoj tacki su karakteristicne za jezike koji imaju staticku proveru tipova i koji se prevode. OO jezici koji su typeless i interpretiraju se nemaju potrebu za tolikim i takvim komplikacijama. Ovim se samo jos jednom potvrdjuje da prica koja sledi _nije_ono_sto_karakterise_OOP_, vec je samo posledica implementacije prevodilaca za strong-typed OO jezike.

To ne znaci da su typeless OO jezici koji se interpretiraju ``manje OO''. Oni su u istoj meri objektno orijentisani, s tom razlikom da mnogo vise provera ostavljaju run-time sistemu. Time se dobija elegantniji jezik, ali se malo gubi na efikasnosti.

Glava 2.

% Problem sa smestanjem objekata

Posmatrajmo klasu PlaneFigure, poznatu nam jos iz prvog dela Sage:

```
    class PlaneFigure;
    export setCenter, setName, setColour, move;
    instance variables
        centerX, centerY : cardinal;
```

```
        name             : string;
        colour           : Colours;
    instance methods
        setCenter(x, y : cardinal)     is ...;
        setName(newName : string)      is ...;
        setColour(newColour : Colours) is ...;
        move(dx, dy : cardinal)        is ...;
    end PlaneFigure.
```

Ako pretpostavimo da je tip cardinal velik 2 bajta, tip string velik 256 bajta i tip Colours velik jos 2 bajta, za pamcenje instance klase PlaneFigure nam je potrebno 262 bajta.

Posmatrajmo jos i klasu Circle, koja je podklasa klase PlaneFigure:

```
    class Circle;
    inherits PlaneFigure;
    export draw, wipe;
    instance variables
        radius : cardinal;
    instance methods
        draw() is ...;
        wipe() is ...;
    end Circle.
```

Ona od klase PlaneFigure nasledjuje sve instance variables, i dodaje jos jednu -> radius. Zato je za pamcenje instance klase Circle potrebno 262 + 2 = 264 bajta.

Neka se negde u programu javlja ovakva deklaracija:

```
    var pf : PlaneFigure;
```

Recite, koliko bajtova treba alocirati za promenljivu pf? Da li 262 (zato sto je to osnovna velicina) ili 264 (zato sto prica o assignment compatibility daje da se promenljivoj pf uvek moze dodeliti promenljiva tipa Circle)? A sta ako negde u buducnosti napravimo podklasu klase PlaneFigure cija instanca zahteva 12012 bajtova za smestanje? Da li odmah treba alocirati najveci moguci prostor, pa neka zvrji prazan ako ga ne upotrebimo?

Glava 3.

% Resenje problema: reference semantics

Skoro svi OO programski jezici ovu dilemu razrese na najjednostavniji moguci nacin: instanca klase se tretira kao pokazivac na nesto. Zato ce deklaracija

```
    var pf : PlaneFigure;
```

navesti prevodilac da za promenljivu pf alocira samo 4 bajta (koliko je, recimo, potrebno za pokazivac), dok se ono pravo, tj. ono na sta pf pokazuje, alocira naknadno, u toku rada programa.

Na ovaj nacin se obezbedjuje uniformnost pri dodeljivanju: kada se promenljivoj pf dodeljuje instanca klase Circle, recimo, tada se prebacuju samo pokazivaci, cija velicina je prilicno konstantna na fiksiranoj platformi.

Za ovakav pogled na objektivnu realnost kaze se da implementira reference semantics. Ima nekih jezika u kojima se to pokusava srediti na drugi nacin, ali se stvari prilicno zapetljaju. Svi popularni jezici koriste reference semantics. Ako je jezik jos i dobro dizajniran, to se uopste ne vidi, ili se tek naslucuje.

Glava 4.

% Problemi sa resenjem problema; i odmah resenje: konstruktori,

% destruktori i kloniranje

Postoji jedna od posledica Marfijevog zakona (Beckhapov zakon)
koja kaze da je

    lepota * pamet = const.

Zato jako pametno resenje uglavnom nije i jako lepo. To vazi i za
reference semantics: mada resava jedan problem, reference semantics
sa sobom donosi nove.

Prvi veliki problem je u tome da prevodilac alocira samo pokazivac
na objekt. To znaci da se prostor za sadrzaj mora alocirati naknadno!
Tako dolazimo do pojma konstruktora. Konstruktor je nesto (procedura,
metod, specijalna oznaka ili slicno) sto alocira prostor za instancu
date klase. On se mora pozivati eksplicitno.

Na primer, u Smalltalk-u su konstruktori realizovani kao metodi klase.
Svaka klasa ima class method new koji alocira prostor za instancu.
Na primer, ako u promenljivu c zelimo da smestimo instancu klase
Circle, to radimo tako sto klasi Circle uputimo poruku new. Ona napravi
novu instancu i vrati pokazivac:

```
| c |
 ...
c := Circle new.
```

U C++ se to radi pozivom posebnog metoda (koji ima isto ime kao i klasa,
pa tako znamo da je to konstruktor).
U Oberonu(2) se to radi pozivom standardne procedure NEW:

```
VAR c : Circle.Instance;
 ...
NEW(c);
```

a u Eiffel-u koristeci specijalnu sintaksnu konstrukciju (eventualno uz
neku inicijalizacionu poruku):

```
x : SOMETHING;
y : SOMETHING_ELSE;

!!x;
!!y.createProcedure(...);
```

Druga vrsta problema su destruktori. Destruktor sluzi da pocisti
djubre, tj. instance koje su odsluzile svojoj svrsi, pa memoriju
koju su zapremale treba osloboditi i reciklirati. U najvecem broju
kulturnih OO jezika recikliranje radi run-time sistem (u obliku
garbage collector-a). Tako rade Smalltalk, Oberon(2), Eiffel i mnogi
drugi. No, C++ zahteva da se i destruktori pozivaju eksplicitno.
Dakle, u C++ programer ne samo da mora da brine o kreiranju
objekata, vec i o njihovom unistavanju.

Treca vrsta problema je dodeljivanje. Ako su c i d promenljive,
one sadrze pokazivace na sadrzaj objekta, pa dodela

```
c := d;
```

samo prebaci pokazivac. Zato se kod svih OO jezika koji realizuju
reference semantics (a to su skoro svi OO jezici) mora voditi
racuna o dva nivoa dodele:
    (o) prebacivanje pokazivaca
    (o) pravljenje kopije objekta

Dodela oblika c := d u skoro svim jezicima predstavlja samo prebacivanje
pokazivaca. Pravljenje kopije objekta se mora eksplicitno zahtevati.
Npr, u Smalltalku tome sluze metodi shallowCopy i deepCopy,
u Eiffelu tome sluze metodi copy i clone, dok u Oberonu(2) i C++-u
programer mora sam da pise metode koji prave kopiju objekta.

Glava 5.

% Problem sa upucivanjem poruka

Skoro svi moderni OO jezici koji se prevode su dizajnirani tako da podrzavaju strong typing i staticku proveru tipova.
To znaci da se provere tipova vrse u toku kompilacije.

S jedne strane, to je dobro zato sto se jos u ranim fazama otkrivaju mnogi bugovi. S druge strane, staticka provera tipova se malko posvadja sa dinamickim vezivanjem metoda. Evo primera.

Posmatrajmo ponovo klasu PlaneFigure:

```
class PlaneFigure;
export setCenter, setName, setColour, move;
instance variables
    centerX, centerY : cardinal;
    name             : string;
    colour           : Colours;
instance methods
    setCenter(x, y : cardinal)     is ...;
    setName(newName : string)      is ...;
    setColour(newColour : Colours) is ...;
    move(dx, dy : cardinal)        is ...;
end PlaneFigure.
```

i njene podklase Circle, Rectagle, Triangle:

```
class Circle;
inherits PlaneFigure;
export draw, wipe;
instance variables
    radius : cardinal;
instance methods
    draw() is ...;
    wipe() is ...;
end Circle.

class Rectangle;
inherits PlaneFigure;
export draw, wipe;
instance variables
    width, height : cardinal;
instance methods
    draw() is ...;
    wipe() is ...;
end Rectangle.

class Triangle;
inherits PlaneFigure;
export draw, wipe;
instance variables
    side1, side2, side3 : cardinal;
instance methods
    draw() is ...;
    wipe() is ...;
end Triangle.
```

Uzmimo jos da je fig niz ravanskih figura deklarisan ovako:

```
var fig : array [1 .. MaxN] of PlaneFigure;
```

i da su u fig[1] do fig[N] nekako smesteni razni krugovi, pravougaonici i trouglovi. Vec smo konstatovali da je najelegantniji nacin da se iscrta svih N figura ovaj:

```
for i := 1 to N do
    fig[i].draw
end;
```

No, to su nase zelje. Pogledajmo kako se prevodilac ponasa kada pocne da prevodi ove tri linije. for je OK, tu nema sta da se filozofira (pod pretpostavkom da su i i N promenljive deklarisane na odgovarajuci nacin). Prevodilac pocinje da prevodi

drugu liniju. Pogleda promenljivu fig[i], primeti da je ona instanca klase PlaneFigure i da joj se upucuje poruka draw. Prevodilac bi sada zeleo da obavi type checking, pa razgleda malo po definiciji klase PlaneFigure. No, tamo nema metoda koji bi odgovarao poruci draw! Compilation error.

Dakle, strong typing i staticka provera tipova nam kvare zelje. Izgleda da se staticka provera tipova i dinamicko vezivanje metoda ne trpe. A zbog lepote obe ideje zelimo da zadrzimo i jedno i drugo. Sta cemo sad?

Glava 6.

% Resenje problema: virtuelni metodi

Resenje problema je dosetka koja se zove virtuelni metodi. Klasu PlaneFigure cemo definisati ovako:

```
class PlaneFigure;
export setCenter, setName, setColour, move, draw, wipe;
instance variables
    centerX, centerY : cardinal;
    name             : string;
    colour           : Colours;
instance methods
    setCenter(x, y : cardinal)     is ... begin ... end setCentrer;
    setName(newName : string)      is ... begin ... end setName;
    setColour(newColour : Colours) is ... begin ... end setColour;
    move(dx, dy : cardinal)        is ... begin ... end move;

    draw()                         is virtual;
    wipe()                         is virtual;
end PlaneFigure.
```

Deklaracija virtual predstavlja obecanje prevodiocu. Ona znaci da ce metodi draw i wipe biti definisani negde u nekoj podklasi, ali da cemo se na njih pozivati pre nego sto budu implementirani.

virtual je nesto slicno sa forward u Pascalu. Kao sto forward znaci da ce odgovarajuc procedura biti definisana kasnije, ali da je moramo koristiti pre nego sto bude definisana, tako i virtual obecava prevodiocu da cemo odgovarajuci metod definisati negde u podklasi klase, ali da ipak moramo da ga koristimo pre toga.

Ako je klasa PlaneFigure definisana na ovaj nacin, prevodjenje do sada problematicnog parceta kooda

```
for i := 1 to N do
    fig[i].draw
end;
```

tece bez problema. Kada shvati da je fig[i] instanca klase PlaneFigure i da joj je upucena poruka draw, prevodilac pronjuska po definiciji klase PlaneFigure, vidi da tamo postoji metod draw, proveri tipove (u ovom slucaju nema sta da proverava zato sto metod nema argumenata), i uradi sve ostalo sto treba da uradi. Cinjenica da je metod draw virutelan ga eventualno moze naterati da generise i neki dodatni kood koji ce u toku izvrsenja programa proveravati da li je programer ispunio obecanje i obezbedio metod draw u svim podklasama koje se koriste.

Glava 7.

% U kojoj se vidi kada se koriste virtuelni metodi i sta nama
% prevodilac obecava

Virtuelni metodi se javljaju u onim situacijama u kojima se ono sto metod treba da uradi implementira na potpuno razlicite nacine u razlicitim podklasama te klase.

Kao primer moze da posluzi klasa PlaneFigure i njene podklase

Circle, Rectangle i Triangle. Krug, pravougaonik i trougao se crtaju na potpuno razlicite nacine. Nema sanse da se u klasi PlaneFigure obezbedi metod koji bi bio u stanju da nacrta sve sto moze da bude ravanska figura. Zato se u klasi PlaneFigure samo oznaci da postoje metodi (draw i wipe) koje moraju imati sve podklase klase PlaneFigure i da ce oni biti implementirani tek tamo. Zasto? Zato sto se svaka vrsta ravanskih figura crta na svoj nacin. Najvise sto se moze ocekivati od klase PlaneFigure je to da nagovesti potrebu za metodima. Ona nista ne moze da kaze o njihovoj implementaciji.

Kada se prevodi program napisan na nekom OO jeziku, a koji ima virtuelne metode, staticka provera tipova utvrdjuje da li su ispunjena sva obecanja koja je programer dao. Dakle, prevodilac proverava da li postoji bar jedna implementacija za svaki virtuelni metod koji se javlja u programu. Naravno, moze ih postojati vise. Dinamicko vezivanje metoda odredjuje koju implementaciju virtuelnog metoda treba shvatiti kao interpretaciju poruke.

Glava 8.

% Apstraktne klase i njihove podklase; ujedno i kraj price o
% virtuelnim metodima

Klasa koja sadrzi virtuelne metode se zove apstraktna klasa. Klasa koja nema virtuelnih metoda se zove konkretna klasa. Podklasa apstraktne klase ne mora ponuditi implementaciju za virtuelne metode. Stavise, podklasa apstraktne klase moze dodati i nove virtuelne metode. Bitno je samo to da za svaku apstraktnu klasu postoji njena podklasa koja je konkretna. Time se obezbedjuje da svaki virtuelni metod bude nekad i negde implementiran.

Takodje, moguce je da podklasa neke konkretne klase bude apstraktna. Za nju tada vazi sve sto vazi i za ostale apstraktne klase.

Glava 9.

% Jos jedan korak dublje u apstraktno: genericki kood i genericke
% klase

Posmatrajmo klasu Stack cije instance su stekovi. Za stek nije bitno sta se na njega stavlja. Mi znamo da je stek nesto na cega mozemo staviti nesto drugo i to kasnije sa njega skinuti. S druge strane, kada implementiramo stek, moramo eksplicitno navesti sta je to sto se na stek stavlja. Na primer, u Moduli-2 se stek na koga se stavljaju celi brojevi opisuje ovako:

```
    TYPE IntStack      = POINTER TO IntStackEntry;
         IntStackEntry = RECORD
                             info : INTEGER;
                             link : IntStack
                         END;
```

dok se stek na koga se stavljaju realni brojevi opisuje ovako:

```
    TYPE RealStack      = POINTER TO RealStackEntry;
         RealStackEntry = RECORD
                              info : REAL;
                              link : RealStack
                          END;
```

Opis je skoro isti. Stavise, operacije za manipulaciju ovim stekovima su jos slicnije, bas zato sto stek kao struktura podataka savrseno ne zavisi od onoga sto se na njega stavlja. Bilo bi jako lepo kada bi nam jezik pruzao mogucnost da opisemo stek nezavisno od toga sta se na njega stavlja, pa da od te jedinstvene specifikacije kasnije nekako napravimo konkretnije stvari (kao sto su stek celih brojeva i stek realnih brojeva).

Mehanizam koga mnogi jezici eksploatisu da bi ovo ostvarili se

zove ``genericke klase'' (ili ``genericki moduli''). Evo primera klase Stack u pseudojeziku:

```
    class Stack[class T];
    export init, push, pop, full, empty;
    instance variables
        s   : array of T;
        max : cardinal;
        top : cardinal;
    insance methods
        init(M : cardinal) is
        begin
            new(s, M);
            max := M; top := 0
        end init;

        push(x : T) is
        begin
            if top > max then
                ERROR("Stack is full")
            else
                s[top] := x;
                inc(top)
        end end push;

        pop(var x : T) is
        begin
            if top = 0 then
                ERROR("Stack is empty")
            else
                x := s[top];
                dec(top)
        end end pop;

        full() : Boolean is
        begin
            return top > max
        end full;

        empty() : Boolean is
        begin
            return top = 0
        end empty;

    end Stack.
```

Genericka klasa predstavlja semu po kojoj se mogu praviti razne klase. To je nacin da se odjednom opise beskonacno mnogo klasa. Za svaku vrednost parametra T dobija se jedna nova klasa.

Primeri upotrebe klase Stack izgledaju ovako:

```
    var IntStack  : Stack[integer];
        RealStack : Stack[real];
```

Sada su IntStack i RealStack instance dve razlicite klase (s tim da su te dve klase nastale ``konkretizacijom'' jedne iste genericke klase). Proces u kome se od genericke klase pravi konkretna klasa se zove instanciranje genericke klase. Parametar genericke klase je najcesce neka druga klasa i/ili neka konstanta.

Glava 10.

% Koja zucne nesto o ogranicenim generickim parametrima

Za neke genericke klase je bitno da klasa koja je genericki parametar ima neka specijalna svojstva. Na primer, kada pravimo klasu BinSearchTree, bitno je da elementi koji se smestaju u cvorove stabla budu takvi da se mogu uporedjivati. Dobro je da se sva takva posebna svojstva posebno naglase u deklaraciji genericke klase.

Evo primera. Neka je data apstraktna klasa LinOrder koja opisuje linearno uredjene elemente:

```
    class LinOrder;
    export lt, le, gt, ge;
    instance methods
        lt (y : LinOrder) : Boolean is virtual;

        le (y : LinOrder) : Boolean is
        begin
            return self.lt(y) or (self = y)
        end le;

        gt (y : LinOrder) : Boolean is
        begin
            return y.lt(self)
        end gt;

        ge (y : LinOrder) : Boolean is
        begin
            return self.gt(y) or (self = y)
        end ge;

    end LinOrder.
```

Klasa BinSearchTree tada moze da se opise ovako:

```
    class BinSearchTree[class T(inherits LinOrder)];
    export new, insert;
    instance variables
        info        : T;
        left, right : BinSearchTree[T];
    instance methods
        new(x : T) is
        begin
            info := x;
            left := NIL; right := NIL
        end new;

        insert(x : T) is
        begin
            if null(self) then
                ERROR("Argh!")
            elseif x.lt(info) then
                if null(left) then
                    left.new(x)
                else
                    left.insert(x)
                end
            elseif null(right) then
                right.new(x)
            else
                right.insert(x)
        end end insert;
    end BinSearchTree.
```

Posebnom deklaracijom koju smo naveli u zagradi iza generickog argumenta naglasavamo da se kao argument T moze pojaviti bilo koja klasa koja je podklasa klase LinOrder. To nam treba zato sto elemente koje smestamo u cvorove stabla moramo porediti.

Glava 11.

% Koja kazuje da se mogu praviti poklase genericke klase;
% raj price o generickim klasama

Zivot se i dalje komplikuje. Jedna od lepih komplikacija je to da se mogu praviti podklase genericke klase. U takvim slucajevima i podklasa mora biti genericka. No, tu treba uociti jednu suptilnost. U ovom primeru:

```
    class WeirdStack;
    inherits Stack[integer];
    ...
    end WeirdStack.
```

klasa WeirdStack nije podklasa genericke klase. Klasa WeirdStack je podklasa obicne klase, s tim da je ta obicna klasa nastala instanciranjem genericke klase. Evo, sada, primera klase koja je doista podklasa genericke klase:

```
class AnotherWeirdStack[T];
inherits Stack[T];
...
end AnotherWeirdStack.
```

Ova klasa je genericka i nasledjuje genericku klasu. Tako, mozemo praviti citave mreze generickih klasa i na taj nacin pisati veoma apstraktan kood.

Glava 12.

% U kojoj se vidi sta je to visestruko nasledjivanje

Visestruko nasledjivanje je mehanizam u kome jedna klasa moze nastati nasledjivanjem osobina nekoliko klasa. Evo primera:

```
class Colour;
export setColour, colourOf;
instance variables
    col : cardinal;
instance methods
    setColour(c : cardinal) is ...;
    colourOf() : cardinal   is ...;
end Colour.

class BwPlaneFigure;
export setCenter, setName, move;
instance variables
    centerX, centerY : cardinal;
    name             : string;
instance methods
    setCenter(x, y : cardinal) is ...;
    setName(newName : string)  is ...;
    move(dx, dy : cardinal)    is ...;
end BwPlaneFigure.

class ColouredCircle;
inherits Colour, BwPlaneFigure;
export draw, wipe;
instance variables
    radius : cardinal;
instance methods
    draw() is ...;
    wipe() is ...;
end ColouredCircle.
```

Klasa ColouredCircle je klasa koja je nasledila kood od dve klase: Colour i BwPlaneFigure. Ona nasledjuje promenljive i metode od obe klase, i uz put doda jos koju promenljivu i metod. Ako je cc deklarisana ovako

```
var cc : ColouredCircle;
```

onda je ovo validan niz poruka:

```
cc.setColour(7);
cc.setName("Crveni krug");
cc.setCenter(10,10);
cc.move(-17,6);
...
if cc.colourOf() = 13 then ... end;
```

Instanca klase ColouredCircle ima osobine i jedne i druge klase.

Glava 13.

% Problemi... (za promenu)

Problemi nastaju kada klase od kojih nasa klasa nasledjuje osobine imaju metode sa istim imenom. Na primer ovako:

```
class Col;
export set, get;
instance variables
    col : cardinal;
instance methods
    set(c : cardinal) is ...;
    get() : cardinal  is ...;
end Col.

class BwPF;
export set, setName, move;
instance variables
    centerX, centerY : cardinal;
    name             : string;
instance methods
    set(x, y : cardinal)      is ...;
    setName(newName : string) is ...;
    move(dx, dy : cardinal)   is ...;
end BwPF.

class ColCirc;
inherits Col, BwPF;
export draw, wipe;
instance variables
    radius : cardinal;
instance methods
    draw() is ...;
    wipe() is ...;
end ColCirc.
```

Prilikom prevodjenja klase ColCirc prevodilac uocava da klase Col i BwPF imaju metod sa istim imenom: set. Compilation error: Name clash.

To se cesto desava zato sto se neke operacije, mada nad razlicitim vrstama podataka, najbolje opisuju istim imenom

Glava 14.

% I opet resenje: promena imena

Najjednostavniji nacin da se razrese probelmi sa istoimenim metodima je da se uvede mehanizam promene imena. Ideja je u tome da se u sklopu inherits deklaracije omoguci da se nekim metodima promeni ime samo za potrebe tekuce klase. I to radi fino. Resenje problema ColCirc bi moglo da izgleda ovako:

```
class ColCirc;
inherits Col(rename set as setCol),
         BwPF(rename set as setPos);
export draw, wipe;
instance variables
    radius : cardinal;
instance methods
    draw() is ...;
    wipe() is ...;
end ColCirc.
```

Kada pisemo kood za klasu ColCirc jedan metod ima ime setCol, a drugi setPos i time je konflikt razresen. Treba imati u vidu da je ova promena imena, kao sto je vec receno, lokalna. Imena setCol i setPos vaze samo u klasi ColCirc. Ostatak univerzuma te metode vidi pod starim imenom i, ako zatreba, moze da im (opet lokalno) promeni imena.

Glava 15.

% U kojoj se pojavljuje jedan suptilniji problem: ponovljeno
% nasledjivanje (repeated inheritance)

Posmatrajmo klasu A. Neka su B1 i B2 njeni naslednici i neka je klasa C naslednik klasa B1 i B2:

```
class A;
...
end A.

class B1;           class B2;
inherits A;         inherits A;
...                 ...
end B1.             end B2.

class C;
inherits B1, B2;
...
end C.
```

Klasa A obezbedjuje neke promenljive (instance variables) i metode. Svaka instanca klase B1 ima svoju kopiju promenljivih iz klase A, dok svaka instanca klase B2 ima svoju kopiju. Da li instance klase C treba da imaju dve kopije promenljivih koje poticu iz klase A ili samo jednu? Drugim recima, koliko puta ce klasa C naslediti klasu A? Da li dva puta ili jednom?

Ima opravdanja i za jedno i za drugo! Ima situacija kada je potrebno klasu A naslediti jednom, a ima situacija kada je potrebno klasu A naslediti dva puta. Evo primera:

```
class Address;
...
instance variables
    street : string;
    no     : cardinal;
    city   : string;
    zip    : cardinal;
...
end Address.

class Home;
inherits Address;
...
end Home.

class Business;
inherits Address;
...
end Business.

class HomeBusiness;
inherits Home, Business;
...
end HomeBusiness.
```

Ako poretpostavimo da klasa Address opisuje podatke o adresi (postanskoj) nekog objekta, ako klasa Home opisuje kucu u kojoj neko stanuje, klasa Business opisuje posao kojim se neko bavi, a klasa HomeBusiness opisuje posao kojim se neko bavi kod svoje kuce, onda klasa HomeBusiness treba samo jednom da nasledi klasu Address. Razlog je jednostavan: obe stvari se desavaju na istoj adresi!

Pogledajmo, sada, drugi primer:

```
class Colour; ... end Colour.

class Jacket; inherits Colour; ... end Jacket.
class Trousers; inherits Colour; ... end Trousers.
class Shirt; inherits Colour; ... end Shirt.
class Underwear; inherits Colour; ... end Underwear.
class Shoes; inherits Colour; ... end Shoes.
```

```
    class Outfit;
    inherits Jacket, Trousers, Shirt, Underwear, Shoes;
    ...
    end Outfit.
```

Kako svaki deo odece ima svoju boju, i kako sve boje mogu biti razlicite, klasa Outfit treba da nasledi klasu Colour pet puta.

Ako jedna klasa nasledjuje drugu klasu vise od jednom, takav oblik nasledjivanja se zove ponovljeno nasledjivanje (repeated inheritance). Jasno je da se takav oblik nasledjivanja moze javiti samo kod OO jezika sa visestrukim nasledjivanjem.

Glava 16.

% Koja govori o tome kako razliciti jezici resavaju problem
% ponovljenog nasledjivanja

Razni jezici resavaju problem ponovljenog nasledjivanja na razne nacine. Neki jezici jednostavno zanemaruju ovaj problem i opredele se iskljucivo za jednu ili drugu varijantu kod ponovljenog nasledjivanja: ili se svaka klasa nasledi tacno jednom, ili se svaka klasa nasledi onoliko puta koliko se puta njene podklase pojave i inherits listi.

To su inferiorna resenja. Bolji jezici to rese tako sto omoguce oba mehanizma nasledjivanja, pa programer odabere verziju koja mu odgovara. Kao sto smo videli, oba pogleda su potrebna.

Ovakav pristup ima Eiffel. On omogucuje da se klasa nasledi jednom ili vise puta, a to postize kroz suptilan mehanizam promene imena (pravilo je jednostavno: ako se imena metoda iz klase preimenuju, onda se klasa nasledjuje nekoliko puta; ako se pak ne preimenuju vec se zadrze iskljucivo originalna imena, onda se klasa nasledi samo jednom).

Glava 17.

% Koja govori o cistim i hibridnim OO jezicima

OO jezika ima raznih vrsta. Neke do njih smo pomenuli jos na pocetku. No, ima i grugih podela. Jedna od znacajnijih je podela na *ciste* i *hibridne* OO jezike. OO jezik je cist ako se svi podati opisuju u terminima klasa i sve akcije se iskazuju u obliku upucivanja poruka. Na primer, u cistim jezicima se sve akcije, pa i akcija

```
    a := b.nameOf
```

shvata kao upucivanje poruke objektu (u ovom slucaju, objektu a upucujemo poruku := b.nameOf). Cisti OO jezici su, na primer, Smalltalk i Eiffel.

Hibridni jezici su mesavine OO i nekog drugog stila, recimo imperativnog. Kod hibridnih jezika se klase, objekti i poruke mesaju sa tipovima, strukturama i naredbama. Hibridni (imperativni) jezici podrzavaju bar dve vrste akcija: slanje poruke i poziv procedure. Tako, u velikom broju hibridnih (imperativnih) jezika se akcija

```
    a := b.nameOf
```

shvata kao poziv ``procedure'' := sa argumentima b.nameOf i a. Naravno, kao hibridni jezici se, pored imperativnih OO jezika, mogu javiti logicki OO jezici, funkcionalni OO jezici, ...

Zbog toga sto podrzavaju mesavinu pogleda na svet, hibridni jezici mogu biti veoma komplikovani. Njihova semantika je najcesce formulisana u obliku niza slozenih pravila.

Za hibridne OO jezike je karakteristicno jos i to da se klase implementiraju kao posebna vrsta tipova podataka. Tako, jedan

modul moze sadrzati opis nekoliko klasa. Kod cistih OO jezika to uglavnom nije tako. Tu klase sluze i kao mehanizam uvodjenja novih vrsta podataka i kao mehanizam razbijanja programa na module.

Glava 18.

% Koja predstavlja komentar i zakljucak

OO ideja je prilicno stara (1967), ali tek u poslednje vreme postaje popularna. Moglo bi se cak reci da se desilo i nesto nepozeljno: ne samo da je postala priznata kao lepa i korisna ideja, vec je postala moda. OO je danas *in*.

To nije dobro. Ima problema za koje nema potrebe koristiti OO. Ima problema koji se mogu mnogo brze i jednostavnije resiti klasicnim metodama. S druge strane, projektovanje velikih softverskih sistema moze biti mnogo lakse uz OO.

OO ne treba gurati tamo gde mu nije mesto. Na primer, OO sistemi nose odgovarajuci overhead bar zbog dve stvari: dinamickog vezivanja metoda i kupljenja djubreta (garbage collection). Zato oni uglavnom nisu pogodni za real-time aplikacije. OO je korisna i dobra metodogija, ali nikako ne i najbolja. Sve to treba imati u vidu kada se bira alat pri ulasku u veliki projekt.

Ovo je kraj Sage o OO. Cilj je bio da se uvedu i ilustruju osnovni pojmovi i da se pokazu neke tehnike implementacije OO jezika. Saga nikako nije potpun vodic kroz oblast, niti je definitivna definicija osnovnih pojmova. OO programiranje je jos uvek u eksperimentalnoj fazi, i ko zna koji mehanizmi ce za 10 godina biti prihvaceni kao dobri.

Saga je trebalo da da osnovne informacije o oblasti i tako olaksa samostalan rad i dalje ucenje. Nadam se da je u tome bar delom uspela.

Saga o OOP, Dodatak: Cemu OOP
Zoran Budimac
(u sklopu Vannastavnih aktivnosti iz programiranja, PMF-Novi Sad)

Glava D1.

Da je kriza sa softverom, to je svakome jasno. Pa i laicima koji su imali nesrecu da ih iko ikad informatizuje. Te ne radi, te ne radi onako kako treba, te ovo te ono. Pa su se ljudi poceli vremenom pitati zasto im recimo TV (uglavnom) radi i zasto se zgrada u kojoj stanuju vrlo retko rusi, a taj prokleti program nikako da proradi kao sto treba...

Ako pretpostavimo da je to tako zato sto se vecina stvari programira ispocetka, (sa obaveznim gledanjem u source, koji je cesto i neko drugi pravio, pa se tako neke greske ponavljaju, a bogami se naprave i nove...) ako je dakle problem sto se vrlo retko koriste *postojece i istestirane* stvari, onda je resenje napraviti module ("cipove") koji bi tacno odgovarali unapred dogovorenim specifikacijama i koji bi se umetali na odgovarajuca mesta u programima. Ako se bas na trzistu "cipova" ne moze pronaci onaj koji nam tereba, onda ne bi bilo zgoreg napraviti mehanizam koji ce nam omoguciti da iskoristimo one stvari koje nam iz "cipa" odgovaraju, te da neke malko doradimo ili izmenimo. Dorada i izmena bi se morali napraviti tako da se nikako ne *interferiraju* sa onim sto vec postoji u "cipu" te da samo koriste "funkcionalnost" onoga sto tamo postoji, nadogradjivanjem.

Glava D2.

Ostarivanjem tog ideala bi se programeri podelili u 4 grupe, prema interesovanjima i kvalitetima...

Prva grupa bi negde u "silikonskim" dolinama u sterilnim prostorijama i sa rukavicama na rukama pravila "cipove". "Cipovi" bi odgovarali specifikacijama koje su ranije dogovorene na "planetarnom" nivou i nosili bi neke oznake (SPC/14m, na primer). Tu bi morali raditi oni koji su postigli "unutrasnju harmoniju" ili oni koji imaju dobre veze...

Negde malo iza njih na pokretnoj traci (a u malo lagodnijoj atmosferi), bi bili "kontrolori" koji bi proveravali da li "cipovi" odgovaraju specifikacijama. Skartovi bi bili odbacivani, a kreatori skartova bi dobijali otkaz... Ovde bi mogli da rade propali asistenti...

Treca grupa programera bi se okupljala u fabrikama korisnickih programa. Takve fabrike bi od fabrika "cipova" redovno dobijale kataloge najnovijih cipova sa oznkama i specifikacijama. Fabrike korisnickih programa bi od postojecih "cipova" sklapale sopstvene programe koji treba da rese neki konkretan problem. Ako je fabrika dobra, ona bi od "cipova" mogla da napravi tzv. "kartice" za koje moze da ocekuje masovniju prodaju. Ako odgovarajuci "cip" ne postoji na trzistu, fabrika bi odabrala najpriblizniji "cip", nasledila njegove dobre osobine i malo ih prilagodila, pri tome ne menjajuci unutrasnjost "cipa" (jer za tako nesto ni nema mogucnosti).
Takva operacija bi se nazivala "lemljenjem". Ovde bi radili programeri sa "harmonijom", ali bez dobrih veza koje bi ih ubacile u "silikonske doline".

Cetvrta grupa programera bi bila poznata pod nazivom "majstori". Njih biste zvali kad vam program odjednom vise ne radi, a do juce je radio. "Majstor" bi (posto je prethodno trgn'o rakijicu) pokusao da popravi program. U zavisnosti od njegovih sposobnosti, on bi mogao da odnese "karticu" u servis i da vam posle donese drugu (trazio je recimo novu, bolju karticu od fabrike). Mogao bi da vam zameni cip koji mu je sumnjiv, sa nekim novim koji se nedavno pojavio na trzistu. Ako je malko vispreniji, mogao bi mozda da zameni cip svojim sopstvenim, koga je mukotrpno pravio nocu, uz svecu (mada bi takve slucajeve zabranjivalo udruzenje "silikonskih programera"). Mogao bi i da prizna da on to ne zna jer su to sve neki novi cipovi, o kojima on ne zna puno i da vam preporuci drugog majstora....

Majstor bi takodje morao da poznaje kataloge "cipova" i njihove specifikacije, mada ne bas sve - na primer samo one koje standardno upotrebljava fabrika "kartica" za koje se on specijalizovao. Ako bi majstor bio privatnik, on bi morao imati dobru vezu u Nemackoj koji bi mu nabavljao nove rezervne delove...

Glava D3.

Najblize opisanom idealu (standardnim cipovima koji se ne mogu mejati nego samo uglavljivati u za to predvidjena mesta) je objektno-orijentisano projektovanje, programiranje, baze podataka ...

I sta sad ne valja?

Ne valja sto se u tim cipovima ponovo moze naci neistestirano "djubre", koje ce jednom, bas kad ne bude trebalo, da "crkne".

Ne valja sto (bar sudeci po g-dji Mili Mitic) se nista nalik na opste prihvacene cipove sa oznakama jos nije pojavilo (bar iz oblasti OO projektovanja). Dakle, nema opstijih biblioteka, nego svako sam koristi svoju biblioteku, za sta mu (objektivno govoreci), OOP ne ni treba...

Ali prethodna dva "nevaljanja" nemaju nikakve veze sa OOP kao idejom, nego sa nesavrsenocu coveka...