ZAVOD ZA AUTOMATIKU I PROCESNO RAČUNARSTVO
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
SVEUČILIŠTE U ZAGREBU

OO SIMSCRIPT

Dinko Korunić 0036355514

SEMINARSKI RAD IZ PREDMETA *MODELIRANJE I SIMULIRANJE*

Zagreb, 2005.

Sadržaj

1 Uvod

Tema ovog seminarskog rada je kratki i kritički osvrt na OO SIMSCRIPT, objektno orijentirani jezik u nastanku i njegove mogućnosti. Kao što je i definirano, cilj ovog seminara nije sintaksa samog jezika, koliko mogućnosti i njegove OO karakteristike.

U nastavku uvoda ćemo samo površno dotaknuti problematiku objektno orijentiranih simulacijskih jezika, budući da je za razumijevanje OO SIMSCRIPT-a prilično korisno znati i njegovu općenitu OO podlogu. Jasno je da će nam ta razrada upravo pomoći i za kasniju detaljnu analizu OO SIMSCRIPT-a, klasificirajući mogućnosti i karakteristike po unaprijed određenim kategorijama; stoga se uvodni dio može promatrati kao kraći repetitorij objektno orijentiranog programiranja i njegove problematike.

1.1 Rast softvera i hardvera

Tijekom zadnjih 50ak godina došlo je do iznimno brzog rasta mogućnosti računala - cijena računalnih komponenti je postala vrlo popularna, a brzina memorije i procesora se višestruko povećala. Razvoj računalnog hardvera je praktički pratio Mooreov zakon, gdje se gustoća pakiranja logike na silikonskoj jezgri grubo opisuje jednadžbom $y = 2^{(x - 1962)}$, pri čemu je x vrijeme u godinama.

Prateći isti taj trend, softverski i ini sustavi postaju sve složeniji i pisanje softvera "od početka" se dešava sve rjeđe. Razlozi su prvenstveno velika zasićenost tržišta gotovim komponentama (kakve god one bile, u vidu gotovog okruženja, funkcija, biblioteka i sl.), ali i vrlo veliko i najčešće neopravdano vrijeme potrebno za kompletnu reimplementaciju sustava. Budući da je i samo modeliranje usko povezano sa softverom, raste i kompleksnost modeliranja gdje sam proces modeliranja prestaje biti čisto izvršavanje računskih simulacijskih algoritama. Sama kompleksnost softvera zahtijeva i sve bolje metode reprezentiranja odnosno modeliranja složenih sustava i njihovih međuodnosa.

1.2 Proceduralno (funkcijsko) programiranje

Proceduralno programiranje kao takvo predstavlja najčešće prvi dodir s programiranjem ili modeliranjem. Simulacijski problemi se predočavaju procedurama, te bivaju reprezentirani ili nekim općenitim komponentama (redovi i sl.) ili u kodu s podatkovnim strukturama i kodom. No, upravo proceduralno programiranje zbog svoje jednostavnosti ima niz ozbiljnih problema. Između ostaloga, procedure od kojih se takav program sastoji odgovaraju različitim metodama i algoritmima, a ne komponentama iz stvarnog svijeta. Također, događaji koji se opisuju moraju biti prošireni kontekstom koji omogućava da se procedure mogu lako opisati. Nadalje, komunikacija koja se obavlja sa simulacijskom kodom je najčešće izvedena kroz globalnu razmjenu podataka ili funkcijske pozive, stoga su takvi mehanizmi potpuno otvoreni krajnjem korisniku.

Jasno je, time su i otvoreniji za krivo, odnosno od programera nepredviđeno korištenje. Također je jedan od problema i nadograđivanje jednom napravljenih funkcija i procedura - naime, u slučaju simulacije kakvog složenijeg modela, autor je prisiljen ili koristiti već gotove funkcije s ograničenim setom mogućnosti ili sam u potpunosti krenuti u reimplementaciju. Dakle, u klasičnom proceduralnom programiranju neki problem iz stvarnog svijeta se pokušava prikazati pomoću relativno malog broja unaprijed definiranih tipova podataka (cijelih i realnih brojeva, nizova znakova i vektora/polja).

Osim klasičnog postoji i proceduralno strukturirano programiranje koje se zasniva na promatranju programa kao niza potprograma, odnosno programskih odsječaka. Specifično svaki od tih potprograma ima svoj naziv, ulazne i izlazne vrijednosti i lokalne podatke, a predstavlja logičku cjelinu unutar cjelokupnog programa. Postignuće prema klasičnom nestrukturiranom programiranju je vrlo hijerarhijska struktura, izdvajanje logičkih cjelina unutar programa u potprograme te striktno odvajanje podataka od njihove obrade. No, upravo je i ovo zadnje problem ako treba u istu strukturu voditi različite podatke, budući da im treba prvo ispitati tipove i onda pozvati odgovarajući potprogram.

1.3 Objektno orijentirani pristup

Danas se sve više spominje i u praksi koristi objektno orijentirano programiranje, polako gušeći strukturirano programiranje. No, teško je definirati točnu i nedvosmislenu definiciju objektnog pristupa, budući da svaki programer prakticira isto na različiti način. Posve je jasno da je objektno orijentirani pristup svojstvo kojeg može imati svega jedan dio sustava, a da sam sustav u cijelosti ne mora biti objektno orijentiran.

Vrijedi nekoliko općenitih osnovnih principa koje su još Stroustrup i Madsen postavili:

- dizajn može biti objektno orijentiran, bez obzira što rezultirajući program ne mora biti,
- program može biti objektno orijentiran, čak i ako jezik u kojem je napisan nije,
- objektno orijentirani program može biti napisan u bilo kojem jeziku, ali jezik ne može imati svojstvo objektne orijentiranosti osim ako jezik upravo omogućava objektno orijentirane programe.

Objektno orijentirane programe, nasuprot proceduralnim, pojednostavljeno možemo promatrati kroz njihovo izvršavanje u vidu fizikalnog modela koji simulira ponašanje promatranog sustava iz stvarnog ili imaginarnog svijeta. **Klase** (nasuprot ograničenim tipovima podataka iz proceduralnog pristupa) su tipovi koje programer definira pri modeliranju sustava, a riječ je o vrsti prototipa za svoje **instance** (objekte-primjerke nastale iz klase). Klase kao takve specificiraju i podatke ali i ponašanje svih objekata koji nastaju iz njih. Specifično, klasa može imati dva dijela, a to su **atributi** koji opisuju što čini klasu te **metode** koji opisuju

što klasa čini. Različite instance iste klase imaju iste atribute i metode, ali različite vrijednosti atributa.

1.4 Glavne karakteristike objektno orijentiranog programiranja

Tijekom dizajna zastarjeli koncepti podataka i procedura trebaju biti posve zamijenjeni novim konceptima **aktivnosti**, **komunikacije** i **nasljeđivanja**. Da bi programer mogao napisati objektno orijentirani program, nužno je da ima u jeziku implementiranu podršku za mehanizam klasa s nasljeđivanjem, kao i mehanizam koji omogućava pozive funkcija u objektima da ovise o stvarnom tipu objekta (jasno, u slučajevima kad je stvarni tip nepoznat u trenutku kompiliranja). Naravno, objektno orijentirano programiranje ne isključuje funkcionalno programiranje kad je ono prirodnije - funkcije, tipovi i vrijednosti se i dalje koriste da izraze mjerljive osobine objekata.

Pobrojimo dakle najvažnije pojmove OO paradigme:

- **identitet** - svaki objekt ima svoj identitet u stvarnosti koji ga razlikuje od okoline, ima svoje karakteristike (atribute) i ponašanje (metode),
- **klasifikacijska apstrakcija** - definiraju se klase objekata kao skupovi objekata od kojih svaki objekt ima različit identitet, a slične karakteristike i ponašanje,
- **učahurivanje** - objekti bivaju promatrani izvana, te je moguće vidjeti samo vanjske osobine objekta, ali ne i unutrašnje osobine; objekt kao takav je nemoguće otvoriti (takav koncept ne smije postojati u jeziku), saznati unutrašnjost ili mu pak promijeniti stanje; dakle skrivaju se informacije o unutrašnjoj strukturi objekta,
- **nasljeđivanje** - podređenoj klasi (izvedenoj iz jedne ili više nadređenih) se dodaju atributi i operacije svih nadređenih, tretirajući podklase tako da imaju svoje vlastito ponašanje, ali nasljeđuju atribute i operacije klase,
- **asocijacija** - moguća je pripadnost jednog objekta kao elementa u drugom objektu,
- **komunikacija** - objekti mogu razmjenjivati informacije, no poruka kao takva mora biti prenesena od pošiljatelja kao standardni zahtjev, ali bez ikakvih dodatnih naznaka što točno treba primatelj učiniti: primatelj kao takav mora odlučiti i obraditi takvu poruku na pravilan način,
- **polimorfizam** - omogućava adaptiranje, odnosno atributi mogu biti dijeljeni unutar neke grupe, no pojedinci grupe mogu te atribute tumačiti na sebi svojstveni način; time je dozvoljena individualnost i mogućnost da se s individualnim ponašanjem nadvlada zajedničko ponašanje.

Odmah možemo i primijetiti da zaštita pristupa (enkapsulacija) donosi poboljšanja u odnosu na proceduralni pristup:

- omogućuje uvođenje ograničenja na stanje objekta,
- osigurava jednostavnije sučelje prema korisniku,
- odvaja sučelje od implementacije.

Stoga možemo ukratko izložiti koje su glavne karakteristike koje očekujemo u jednom OO jeziku:

- objekti:
  - autonomne jedinke,
  - nema im direktnog pristupa,
  - komuniciraju isključivo kroz poruke, bez pretpostavki o implementaciji,
- organizacija objekata:
  - nasljeđivanje koje omogućava stvaranje hijerarhije klasificiranih objekata,
  - objekti ne postoje radi dijeljenja koda, već su prirodni problemu,
  - apstraktni čvorovi u hijerarhiji moraju postojati radi realnijeg modela,
  - mogućnost individualnosti objekta,
  - dinamičko određivanje (tijekom izvršavanja) ispravnih odgovora na poruke,
- programi kao modeli:
  - programi modeliraju razvoj nekog unaprijed definiranog sustava,
  - mijenjanje stanja objekata utiče na stanje cjelokupnog sustava,
  - objekti djeluju konkurentno,
  - podrška za ne-objektno orijentirane tehnike u slučaju kad je to potrebno.

Dakle, na te karakteristike ćemo upravo obratiti pažnju u OO SIMSCRIPT-u.

2 Objektno orijentirani SIMSCRIPT

Jezik OO SIMSCRIPT je zamišljen kao nadskup jezika SIMSCRIPT II.5. Prema autorima, nove mogućnosti dijele relativno sličnu sintaksu s originalnim jezikom i omogućavaju kompatibilnost, tvoreći tako koherentnu cjelinu. U sljedećem pregledu nećemo se sadržavati na specifičnoj sintaksi i naredbama, već ćemo analizirati mogućnosti i karakteristike jezika.

2.1 Uvod u karakteristike jezika

Prije nego prijeđemo na detalje, možemo ukratko sažeti idućih par poglavlja i izreći sljedeće činjenice o jeziku OO SIMSCRIPT:

- njegova struktura je modularna (kod razbijen na module, sustave, zaglavlja i potprograme), samim time i u duhu objektnog programiranja,
- prisutne su klase kao prototipovi objekata (dakle dinamički tipovi podataka), a pojedina klasa može sadržavati atribute, metode i grupe,
- atributi i metode mogu biti pridijeljeni individualnim klasama, čime se mogu ostvarivati statičke varijable,
- omogućeno je kako jednostavno tako i višestruko nasljeđivanje između klasa, a dozvoljeno je i preklapanje naslijeđenih metoda vlastitim implementacijama,
- enkapsulacija je moguća: prisutno je standardno definiranje javnih i privatnih osobina pojedinog objekta ili klase, pri čemu se mogu formirati sučelja prema korisniku i definirati dozvole pristupa,
- prisutno je i područje djelovanja simbola - oni mogu biti lokalni određenoj klasi ili pak globalni, ovisno o tipu definicije,
- postoji par tipova metoda jednog objekta - mogu biti funkcije (pozivaju se implicitno), a mogu biti i rutine (pozivaju se eksplicitno),
- postoji tzv. referenca na vlastiti objekt čime se ostvaruje "self" funkcionalnost,
- postoje konstruktori i destruktori i inicijalizacijske rutine u vidu implicitnih metoda koje je moguće preopteretiti,
- općenito rečeno, preopterećivanje nalik na ono u jeziku C++ nije dozvoljeno, ali se ostvaruje kroz lijeve i desne implementacije funkcija i nadzor atributa; takvo opterećivanje je rudimentarna iako upotrebljiva forma iznimki, gdje se omogućava efikasniji prihvat nepoznatog podatka i njegova obrada,
- prisutan je i naprednji tip podataka - set odnosno grupa koja je implementacijski dvostruko povezana lista s glavom,
- postoje i naprednije metode na razini jezika koje omogućavaju postavljanje vremenski baziranih metoda, što je već zadiranje u prostor operacijskog sustava.

Možemo unaprijed zaključiti da je riječ o jeziku visoke razine (vrlo blizak ljudskom tj. engleskom jeziku) koji umnogome ima slične karakteristike tipičnom OO jeziku,

primjerice C++-u, ali sadrži i neke specifične osobine koje mu omogućavaju da bude primjereniji simulacijskom jeziku (rutine bazirane oko vremenskih događaja, statističke funkcije, nadzor atributa, lijeve i desne implementacije, razlikovanje rutina i funkcija). Jedna od većih razlika je nedostatak preopterećenja operatora, tehnike vrlo često rabljene u jeziku C++. Općenito rečeno, ovaj jezik zadovoljava standardne OO premise, a mogućnost lijeve i desne implementacije predstavlja programsku zanimljivost već viđenu u jezicima Eiffel i SIMSCRIPT II.5.

2.2 Izgled i opća sintaksa programa

Za razliku od SIMPSCRIPT II.5 programa koji se sastoji od zaglavlja (*preamble)* koja sadržava sve definicije globalnih varijabli, glavnog programa (*main routine*) i nekoliko opcionalnih potprograma-rutina (*subordinate routines*), struktura OO SIMSCRIPT programa potpuno modularna, proširujući osnovne strukture.
Tipični OO SIMSCRIPT program se sastoji od glavnog **modula** (*main module*) i nekoliko opcionalnih podmodula koji se zovu **podsustavi** (*subsystem*, *module*, *package*). Taj glavni modul izgleda upravo kao tipični SIMSCRIPT II.5 program. Podsustavi su, pak, nešto drukčiji: oni su imenovani moduli, a sastoje se od **javnog** zaglavlja (*public preamble*), **privatnog** (*private preamble*) opcionalnog zaglavlja i nekoliko opcionalnih potprograma-rutina. Jedan podsustav može biti naveden u nekoliko zaglavlja, npr. u zaglavlju glavnog programa, ali i javnim te privatnim zaglavljima drugih podsustava, pri čemu se u jednom zaglavlju može navesti proizvoljan broj podsustava.

Same dozvole pristupa (enkapsulacija) i pojednostavljeno nasljeđivanje su izvedeni na sljedeći način: simboli definirani u zaglavlju glavnog modula su dostupni svim potprogramima u glavnom modulu. Oni koji su pak definirani u javnim zaglavljima pojedinačnog podsustava su dostupni u privatnom zaglavlju, rutinama u podsustavu ali i svakom drugom podsustavu koji koristi dotični podsustav (tako što je naveo njegovo ime u vlastitom zaglavlju). Simboli definirani u privatnom zaglavlju podmodula su dostupni isključivo rutinama tog podmodula.

Zaglavlje glavnog modula može sadržavati i globalne varijable koje se nazivaju **sistemski atributi** (*system attributes*), no moguće je napraviti i sekundarne globalne varijable u vidu **podsistemskih atributa** (*subsystem attributes*) koje su globalne određenom podustavu i mogu se definirati ili u javnom ili u privatnom zaglavlju svakog podsustava. Lokalne varijable se definiraju korištenjem inicijalizacijskog potprograma koji se poziva samo jednom, za vrijeme inicijalizacije programa.

Područje djelovanja određenog simbola je ovisno o tome gdje je simbol referenciran. Na primjer, ako je definiran u javnom zaglavlju podmodula koji je uključen u tekući podmodul, onda se može koristiti samo lokalnim nazivom, no u svakom drugom slučaju je nužno apsolutno referenciranje (u vidu PODSUSTAV:SIMBOL).

Primjer 1

Prikazat ćemo ukratko izgled tipičnog OO SIMSCRIPT programa s modulima (SHIPPING kao glavni modul te PORT kao njegov podmodul), definicijama zaglavlja, javnim i privatnim definicijama, rutinama i inicijalizacijom:

```
preamble for the SHIPPING system ''zaglavlje glavnog modula SHIPPING
importing the PORT module ''koji navodi PORT modul radi korištenja
  ...
end

routines for the SHIPPING system ''deklaracije rutina
main
  ''rutine mogu slobodno koristiti LOAD ili apsolutno PORT:LOAD
  ''jer je rijec o javnim atributima, ali ne i CAPACITY koji je
  ''je deklariran kao privatni atribut PORT modula
  ...
end

public preamble for the PORT subsystem ''javno zaglavlje za PORT
  ''sve ove definicije su dostupne svim nadređenim modulima
  define LOAD as a routine
    given an integer argument and a real argument
    yielding 2 real values
end

private preamble for the PORT subsystem ''privatno zaglavlje za PORT
  ''ove su pak definicije nedostupne nadređenim modulima
  the subsystem has a CAPACITY
  define CAPACITY as an integer variable
end

routines for the PORT subsystem ''slijede rutine PORT modula

initialize
  ''inicijalizacijska rutina, poziva se jednom prije izvršavanja
  ...
  let CAPACITY = 50
end

routine LOAD given IVAL, RVAL yielding R1, R2
  ''implementacija privatne! rutine
  ...
end
...
```

2.3 Klase i strukture podataka

SIMSCRIPT II.5 je za varijable, odnosno tzv. atribute, standardno koristio privremene objekte (*temporary entity*) u formi dinamički alociranog područja memorije. Dotični objekti su vjerojatno najsličniji C-ovom "struct" tipu varijabli. Za promjenu, OO SIMSCRIPT je jezik baziran oko klasa. **Klase** kao takve se

deklariraju unutar već spomenutih zaglavlja, a u definiciji klase se mogu navesti nove definicije pripadnih varijabli (atributa), metoda i nizova. Naravno, varijable unutar klase mogu imati tip isti kao i klasa koja se upravo definira.

Instanciranje objekta neke klase se ne dešava implicitno, već se eksplicitno navodi i provodi kroz odgovarajuće naredbe (**konstruktor** funkcija). Prilikom instanciranja, konstruktor automatski postavlja sve varijable objekta na nulu. Za razliku od prethodnika, SIMSCRIPT II.5 kod kojeg su se standardno sva u programu alocirana memorija morala eksplicitno oslobađati, OO SIMSCRIPT posjeduje automatski **mehanizam oslobođenja nekorištene memorije** (*garbage collector)* koji koristi nepoznati algoritam, budući da nije nigdje naveden u literaturi. No, moguće je i ručno dealocirati objekt, eksplicitnim pozivanjem **destruktor** funkcija. Treba primijetiti da OO SIMSCRIPT omogućava isključivo **dinamičke objekte** (*dynamic objects*) i još k tome dostupne kroz referencirajuće varijable. Svaka takva varijabla je specifičnog tipa prema odgovarajućoj klasi, a prilikom alociranja memorije se u takvu varijablu upisuje adresa memorije pridružene objektu. Za usporedbu, C++ omogućava i **statičke** i dinamičke **objekte**.

Primjer 2

Pokazat ćemo definiranje klase SHIP, deklariranje pripadnog objekta, instanciranje i kasniju destrukciju:

```
begin class SHIP ''definicija klase, standardno u zaglavlju modula
  ''slijede definicije za klasu SHIP
  ...
end

define TANKER as a SHIP variable ''definicija tipa objekta
...
create TANKER ''eksplicitno stvaranje objekta
...
destroy TANKER ''i eksplicitna destrukcija, koju inače radi GC
...
```

2.4 Atributi objekata i klasa

Varijable, odnosno atributi unutar klase, imaju svoje **područje djelovanja** (*scope*) kao što se i očekuje od modernog jezika - one su lokalne vlastitoj klasi. No, simboli mogu biti definirani i u globalnom nivou ili lokalno nekoj drugoj klasi. Gdje nije jasno kako razriješiti lokalitet, nužno je koristiti i ime klase uz naziv atributa. Pristup atributu nekog objekta se obavlja kroz referencu koja se dobiva (i provjerava za ispravnim tipom za svaki pristup prilikom kompiliranja) kroz ime atributa i imenom specifičnog objekta. Jasno, atributi mogu biti i nizovi, pri čemu oni moraju biti alocirani prije upotrebe.

Klasa može imati i tzv. **atribut klase** *(class attribute)*, koji nije povezan s nekim specifičnim objektom već općenitom klasom i pristupa mu se bez dodatnih referenci. Gdje svaki objekt ima svoju kopiju svakog pojedinačnog atributa, ovdje program ima samo po jednu kopiju svakog atributa klase. Dotični tip atributa je identičan ponašanju statičkih varijabli iz jezika C++.

Primjer 3

Slijedi jednostavan primjer definicije klase, instanciranja objekata i postavljanja atributa objekta:

```
begin class SHIP ''definicija klase
  ...
  every SHIP ''popisivanje standardnih atributa
    has a NAME, a NUMBER.OF.ENGINES, a MAXIMUM.SPEED
      and a MAXIMUM.RPM
  the class has a NUMBER.OF.SHIPS ''popisivanje atributa klase

  define NAME as a text variable ''jednostavno pridjeljivanje tipova
  define NUMBER.OF.ENGINES as an integer variable
  define MAXIMUM.SPEED as a real variable
  ''za matricu je nužno kasnije postaviti veličinu
  define MAXIMUM.RPM as a 1-dim real array
  define NUMBER.OF.SHIPS as an integer variable
  ...
end

define TANKER as a SHIP variable ''deklaracija objekta
create TANKER ''i instanciranje

let NAME(TANKER) = "Emerald Glory" ''te pristupanje atributima
let NUMBER.OF.ENGINES(TANKER) = 4
let MAXIMUM.SPEED(TANKER) = 25.0
reserve MAXIMUM.RPM(TANKER) as 4 ''matrice treba rezervirati
let MAXIMUM.RPM(TANKER)(1) = 182.5
let MAXIMUM.RPM(TANKER)(2) = 172.4
let MAXIMUM.RPM(TANKER)(3) = 162.3
let MAXIMUM.RPM(TANKER)(4) = 152.2
''primjer jednostavnog formatiranja
write NUMBER.OF.SHIPS as "There are", i 3, " ships.", /
...
```

2.5 Metode objekata

Tipove metoda u OO SIMSCRIPT-u možemo podijeliti ukratko u dva tipa: **funkcijska metoda** (*function method*), koja je najsličnija kakvoj funkciji i daje rezultat izvođenja neke funkcije, te **podrutinska metoda** (*subroutine method*) koja nije u vidu funkcije, već standardne rutine (i onda se tako i eksplicitno poziva). Svaka metoda unutar objekta se mora precizno definirati s brojem argumenata (i tipom funkcijskog rezultata za funkcijske metode) - u suprotnom slučaju se pretpostavlja da je riječ o podrutinskoj metodi bez argumenata. Kao i kod atributa, imena metoda imaju lokalni doseg unutar klase, i također je moguće

koristiti apsolutno referenciranje (sa imenom klase) da se nedvosmisleno odredi koja se metoda koristi. Bitno je napomenuti da se i ovdje pozivi provjeravaju za ispravnošću tipova tijekom samog procesa kompiliranja, a metodama objekta se argumenti implicitno prenose po vrijednosti.

Tijekom implementacije samih metoda, programeru je na raspolaganju i lokalna referenca na vlastiti objekt, a ona sadrži adresu instanciranog objekta. Dotičnu referencu je nužno implicitno definirati, a po funkcionalnosti je najsličnija C++ "this" pokazivaču.

Kao i u većini objektnih jezika, moguće je ostvariti dodatnu funkcionalnost nad ugrađenim **konstruktorom** ili **destruktorom** objekta, pri čemu se željene metode unutar nekog objekta implicitno pozivaju nakon stvaranja ili prije uništenja svakog pojedinačnog objekta neke klase. Specifično se destruktorska funkcija poziva implicitno bilo kad se objekt ručno dealocira, bilo kad to automatski napravi *garbage collector*.

Dotične metode se implementiraju kao podrutinski pozivi bez argumenata, a u slučaju da je nužno proslijediti kakav argument, koriste se zamjenske metode koje primaju argumente i koriste ih za inicijaliziranje novih objekata. Jasno, takve metode se moraju pozivati eksplicitno.

Nadalje, kao i kod atributa, moguće je koristiti **metodu klase** *(class method)* koja je specifična za pojedinačnu klasu, te se ne poziva s imenom objekta. No, implementacijski gledano, u takvoj metodi se ne može koristiti lokalna vlastita referenca. Dotična metoda je analogija C++ statičkoj članskoj funkciji (*static member function*).

Primjer 4

Slijedi standardan primjer klase s ugrađenim metodama, te tipičnim konstruktorom, destruktorom i inicijalizacijskom rutinom

```
begin class SHIP ''standardna definicija klase
  ...
  every SHIP
    has a CONSTRUCTOR method, ''očigledno konstruktorska rutina
        a DESCTRUCTOR method, ''destruktorska
        an INITIALIZE method, ''i ona za inicijalizaciju
        a DESCRIBE method ''obična rutina (i to bez argumenata)
        and a COMPARE.RPM method ''i obična funkcija (vraća vrijednost)
  define COMPARE.RPM as an integer method
    given a real argument
  ...
end

methods for the SHIP class
''da nema ovakve deklaracije metoda, bilo bi nužno koristiti
''deklaracije metoda na sljedeći način: method SHIP'DESCRIBE
''odnosno: method SHIP'COMPARE.RPM(RPM)
```

```
method CONSTRUCTOR
  write as "A ship has been created", /
  add 1 to NUMBER.OF.SHIPS
end

method DESTRUCTOR
  call DESCRIBE
  write as "This ship has been destroyed", /
  subtract 1 from NUMBER.OF.SHIPS
end

method INITIALIZE given TVAL, IVAL, RVAL
  let NAME = TVAL
  let NUMBER.OF.ENGINES = IVAL
  let MAXIMUM.SPEED = RVAL
end

method DESCRIBE ''metoda, obrađuje podatke ali ne vraća vrijednost
  write NAME, NUMBER.OF.ENGINES,
    MAXIMUM.SPEED as "The ship ",
    "named ", t *, " has ", i 2, " engines", "
    and a maximum speed of ", d(5,1), /
end

method COMPARE.RPM(RPM) ''funkcija, prima argumente i vraća vrijednost
  define I, COUNT as integer variables
  for I = 1 to NUMBER.OF.ENGINES
    with MAXIMUM.RPM(I) > RPM
      add 1 to COUNT
  return with COUNT
end
...

define TANKER as a SHIP variable ''deklaracija objekta
create TANKER
''call CONSTRUCTOR(TANKER) se dešava implicitno!
call INITIALIZE(TANKER)
  given "Emerald Glory", 4, 25.0
```

2.6 Preopterećivanje i nadzor

SIMSCRIPT OO ne pruža niti jednu formu **preopterećivanja** (*overloading*) operatora ili metoda. Preopterećivanje je vrlo popularno u jezicima poput C++-a ili Java-e, a riječ je o omogućavanju metodi da imaju višestruke implementacije koje se razlikuju i automatski primjenjuju u ovisnosti o broju argumenata i njihovih tipova. No, kao alternativa je uvedena mogućnost da metode imaju **lijevu** i/ili **desnu implementaciju** (*left/right implementation*). Riječ je o mehanizmu gdje svaka funkcija može imati dvije implementacije, pri čemu se desna implementacija izvršava kad se funkcija poziva radi vraćanja vrijednosti, a lijeva implementacija onda kad se funkciji pridružuje vrijednost. Očito, ako postoji ova potonja, funkcija se može pojavljivati u lijevoj strani izraza pridruživanja (analogija *lvalue* u terminologiji jezika C++ i C). Nadalje, podržani su i **nadzirani**

atributi (*monitored attibutes*), čija je karakteristika da implicitno definiraju funkciju s istim imenom i tipom kao i atribut, s lijevim ili desnim implementacijama ovisno da li je atribut nadziran na lijevoj ili desnoj strani. Ako je nadziran na desnoj strani, svaki desni pristup atributu (čitanje njegove vrijednosti) poziva desnu implementaciju funkcije. Ako se nadzire na desnoj strani, svaki lijevi pristup (postavljanje vrijednosti atributu) poziva lijevu implementaciju. Metoda objekta može nadzirati atribut objekta, a metoda klase atribut klase. Nadzor kao takav predstavlja rudimentarnu formu tzv. iznimki, budući da omogućava mehanizme provjere podataka i tipova za vrijeme izvršavanja koda.

Jedna od zanimljivih primjena nadziranja atributa je mogućnost automatskog stvaranja **statistika** (broj jedinki, zbroj, srednja vrijednost, zbroj kvadrata, srednja kvadratna vrijednost, varijacija, standardna devijacija, minimum, maksimum, histogram vrijednosti, itd.) na promatranom atributu objekta ili klase, pri čemu atribut može biti bilo skalarna vrijednost bilo polje vrijednosti. Specifično, atribut biva implicitno nadziran na lijevoj strani, a statistička vrijednost je desna implementacija koja vraća rezultat za navedeni atribut.

Primjer 5

Pokazat ćemo kako se NUMBER.OF.ENGINES atribut može promatrati na lijevoj strani, tako da se spriječi pridruživanje krive vrijednosti. Također, upravo i takav nadzor omogućava da MAXIMUM.RPM polje ima po jedan element za svaki motor. Dalje, MAXIMUM.SPEED atribut može zamijeniti s funkcijskom metodom istog naziva pri čemu desna implementacija takve metode može izračunavati npr. maksimalnu brzinu broda bazirano na nekim drugim atributima (npr. broju i maksimalnoj brzini motora):

```
begin class SHIP
  ...
  every SHIP ''tipični primjer
    has a NAME, a NUMBER.OF.ENGINES,
    and a MAXIMUM.SPEED method

  define NAME as a text variable
  define NUMBER.OF.ENGINES as an integer variable
    monitored on the left ''nadzor varijable s lijeva
  define MAXIMUM.SPEED as a real method
    with no arguments
  ...
end

methods for the SHIP class ''slijede metode

left method NUMBER.OF.ENGINES ''lijeva implementacija
  define N as an integer variable
  enter with N ''saznajemo originalnu vrijednost varijable
  if N is positive
    ''pozitivna je i možemo ju pridijeliti NUMBER.OF.ENGINES
    move from N
    ''ako MAXIMUM.RPM nema N elemenata
```

```
    if DIM.F(MAXIMUM.RPM) is not equal to N
      ''odbacujemo staro polje i stvaramo novo
      release MAXIMUM.RPM
      reserve MAXIMUM.RPM as N
    always
  else
    ''N je negativan, to je greška!
    write as "Invalid number of engines", /
  always
end


right method MAXIMUM.SPEED ''desna implementacija
  define RESULT as a real variable
  ''izračunava se maksimalna brzina
  ...
  return with RESULT
end

left method MAXIMUM.SPEED ''lijeva implementacija
  ''nemoguće je MAXIMUM.SPEED ručno postaviti!
  write as "Cannot set maximum speed", /
end
```

Primjer 6

Idući primjer nam pokazuje jednostavan način kako iskoristiti nadzor na lijevoj strani za izračun statistika nad određenom varijablom:

```
begin class SHIP
...
  every SHIP has a SPEED
  define SPEED as a real variable
  accumulate AVG.SPEED as the mean of SPEED
...
end
```

2.7 Grupiranje objekata

Još jedna od specifičnosti OO SIMSCRIPT-a su **setovi** (*sets*). Riječ je o sortiranoj (FIFO, LIFO ili sortirano po nekim atributima elemenata) grupi istih jedinki (objekata) koja je fizički implementirana u obliku dvostruko povezane liste. Dotične jedinke su članovi, a svaki set ima svojeg vlasnika: specifičnost vlasnika je da za razliku od standardnih pokazivača na prethodne i iduće jedinke ima i ukupni broj jedinki u setu, kao i pokazivače na prvi i zadnji element. Vlasnik nekog seta može biti objekt, klasa, sustav ili podsustav ili neki SIMSCRIPT II.5 kompatibilan entitet.

Primjer 7

Primjer pokazuje grupu SHIP objekata čije je ime FLEET. Klasa SHIP u stvari "posjeduje" dotičnu grupu. Ostatak koda prikazuje stvaranje i inicijalizacije SHIP

objekta i njegovo dodavanje u grupu FLEET, kao i jednostavno iteriranje elemenata u grupi:

```
begin class SHIP
...
  every SHIP
    belongs to a FLEET
  the class
    owns a FLEET
  ...
end

create SHIP
call INITIALIZE(SHIP)
  given "Queen Mary", 8, 35.0
file SHIP in FLEET

for each SHIP of FLEET
  call DESCRIBE(SHIP)
```

2.8 Nasljeđivanje i polimorfizam

Svaka individualna klasa u OO može biti **naslijeđena** od jedne **osnovne klase** (*single inheritance*), ili pak od dvije ili više osnovnih klasa (*multiple inheritance*). Dozvoljeno je i da takva klasa bude osnovna jednoj ili više izvedenih klasa. Izvedena klasa standardno nasljeđuje sve atribute, metode i setove iz osnovnih klasa, ali može definirati i vlastite atribute, metode i setove. Također, dozvoljeno je i standardno preklapanje naslijeđenih metoda objekta vlastitima.

Omogućen je i **polimorfizam** - ovisno o tipu objekta, pozivanje rutina će pozvati odgovarajuću (različitu) rutinu iz pojedinog objekta.

Primjer 8

Dotični primjer se nastavlja na SHIP seriju. Uvodimo podklasu TUGBOAT koja ima dodatni atribut TUGGED.SHIP i sadrži naslijeđene metode iz klase SHIP. Definirane su nove metode BEGIN.TUGGING i END.TUGGING, a metoda DESCRIBE naslijeđena iz klase SHIP je preklopljena novom implementacijom. Zanimljivo je da TUGBOAT nasljeđuje i sposobnost da bude dio grupe FLEET, što ćemo također iskoristiti u primjeru:

```
begin class TUGBOAT
  every TUGBOAT
  is a SHIP, ''koga nasljeđuje TUGBOAT
  has a TUGGED.SHIP, ''dodatni atribut
         a BEGIN.TUGGING method, ''dodatne metode
         and an END.TUGGING method, ''dodatne metode
  and overrides the DESCRIBE method ''i vlastita implementacija!

  define TUGGED.SHIP as a SHIP variable
  define BEGIN.TUGGING as a method
```

```
    given a SHIP argument
end ''TUGBOAT
...

methods for the TUGBOAT class

method BEGIN.TUGGING given SHIP
  if TUGGED.SHIP is not zero
    call END.TUGGING
  always
  let TUGGED.SHIP = SHIP
  write NAME, NAME(TUGGED.SHIP)
  as t *, " begins tugging ", t *, /
end

method END.TUGGING
  if TUGGED.SHIP is not zero
    write NAME, NAME(TUGGED.SHIP)
      as t *, " finishes tugging ", t *, /
    let TUGGED.SHIP = 0
  always
end

method DESCRIBE ''preklapamo postojeću implementaciju
  call SHIP'DESCRIBE ''pozivamo metodu iz originalnog SHIP objekta
  if TUGGED.SHIP is zero ''i slijedi dodatni dio koda
    write as "It is an idle tugboat", /
  else
    write NAME(TUGGED.SHIP) as "It is currently tugging ",
      "the ship named ", t *, /
  always
end
...

create TUGBOAT ''stvaranje
call INITIALIZE(TUGBOAT)
  given "Mighty Mouse", 1, 20.0 ''i tipična inicijalizacija
file TUGBOAT in FLEET ''ubacujemo ga u grupu
call BEGIN.TUGGING(TUGBOAT) given TANKER ''metoda iz TUGBOAT
...
for each SHIP of FLEET ''i tipični prikaz, gdje se koristi...
  call DESCRIBE(SHIP) ''originalni ali i overloadani DESCRIBE
```

2.9 Kontrola pristupa

Imamo dvije vrste klasa - privatne i javne. Privatne klase se definiraju u zaglavlju glavnog modula ili privatnom zaglavlju nekog podsustava. Javne se, za razliku definiraju u javnom zaglavlju nekog podsustava ili kao dodatni blok u privatnom zaglavlju tog istog podsustava, pri čemu se za taj drugi (pretpostavljamo da je postoji privatni blok) dodatni blok podrazumijeva da je javni. Samo su atributi, metode i setovi iz prvog bloka dostupni dodanim modulima, ostatak je dostupan isključivo lokalnom podsustavu. Najčešće se atributi privatne klase definiraju u privatnom zaglavlju, a metode definirane u javnom zaglavlju predstavljaju javno

sučelje toj klasi. Svakoj klasi mogu pristupiti sve druge klase koje su definirane u tom istom modulu, a takav način dozvola odgovara C++ "friends" terminologiji.

Primjer 9

Prikažimo trivijalni primjer s javnim i privatnim zaglavljima:

```
public preamble for the PORT subsystem
  ...
  begin class SHIP
    ''javni dio javne klase, odnosno sučelje same klase,
    ...
  end
  ...
end

private preamble for the PORT subsystem
  ...
  begin class SHIP
    ''privatni dio javne klase, dostupan samo u podsustavima
    ...
  end
  ...
end
```

2.10 Događajno bazirane metode

SIMSCRIPT OO definira **procesnu metodu** (*process method*), koja je implementacijski podrutina bez argumenata i istovremeno predstavlja i metodu i proces. Moguće ju je direktno pozvati kao i ostale rutine (simulacija trenutačnih događaja), ali omogućava i izvršavanje u predodređeno vrijeme tijekom izvršavanja programa (simulacija diskretnih događaja). Takvu metodu je moguće privremeno zaustaviti (dok je zaustavljena se nastavlja simulacijsko vrijeme), ali i kasnije nastaviti. Naravno, moguće je da ju zaustavi neka druga rutina, a moguće je da i sama sebe zaustavi čekajući na kakav događaj, odnosno buđenje od kakve druge rutine. Implementacijski, za metode koje se trebaju izvršiti u nekom vremenu se implicitno stvaraju privremeni objekti sa specijalnim atributima, popunjeni s podacima o budućim događajima.

Procesna metoda se može ostvariti kao metoda objekta ili metoda klase. Ako je riječ o prvoj, poziva se kao dio objekta u smislu aktivnosti objekta. Paralelne aktivnosti objekta se modeliraju kao konkurentne procesne metode koje se pozivaju u ime dotičnog objekta.

Primjer 10

Kao primjer događajno baziranih metoda, pokazat ćemo korištenje procesne metode VOYAGE u klasi SHIP i dodat ćemo atribut AT.SEA.FLAG koji će ukazivati da li je brod npr. na putovanju koje se može vremenski određivati. Pri tome koristi se više načina postavljanja procesa:

```
begin class SHIP
  ...
  every SHIP
    has a VOYAGE process method
           and an AT.SEA.FLAG
  define VOYAGE as a process method ''istovremeno i metoda i proces
    given a real argument
  define AT.SEA.FLAG as an integer variable ''korisna zastavica
  ...
end
...

methods for the SHIP class

process method VOYAGE given DURATION ''definicija procesne metode
  if AT.SEA.FLAG = 1
    ''već je metoda u tijeku, trivijalno zaključavanje
    write NAME as "The ship named ", t *,
    " is already travelling", /
    return
  otherwise
    write NAME as "The ship named ", t *,
      " begins a voyage", /
    let AT.SEA.FLAG = 1
    wait DURATION days ''ovime koristimo suspendiranje rutine
    ''... prošlo je definirano vrijeme
    let AT.SEA.FLAG = 0
    write NAME as "The ship named ", t *,
      " ends a voyage", /
end

''primjer direktnog iniciranja trodnevnog puta, pri čemu pozivatelj
''čeka da metoda završi prije nego se počne izvršavati iduća linija
''u kodu
call VOYAGE(TANKER) given 3.0
...

''primjer koji će pokrenuti izvršavanje puta, ali će odmah
''nastaviti s izvršavanjem iduće linije koda, u praksi
''prije nego se "putovanje" stigne dešavati
schedule a VOYAGE(TANKER) given 3.0 now

''postavljanje "putovanja" u budućnosti koje implicitno stvara
''dodatne strukture u implicitno definiranom VOYAGE atributu objekta,
''što nam omogućava kasnije prekide, definiranje novog događaja, itd.
schedule a VOYAGE(TANKER) given 3.0 in 7.0 days
''što nam omogućava da vršimo npr. sljedeće operacije:
cancel the VOYAGE(TANKER) ''odustajanje
reschedule the VOYAGE(TANKER) in 8.0 days ''seljenje događaja
...
interrupt the VOYAGE(TANKER) ''privremeni prekid
wait 2.0 days ''proizvoljno čekanje (suspendirana je rutina)
resume the VOYAGE(TANKER) ''.. te nastavak
```

3 Literatura

1. S. Vidović, D. Radošević: **Programiranje 2**, FOI skripta sa predavanja
2. M. Vedriš: **Programski jezik Java**, PMF skripta sa predavanja
3. J. Šribar, B. Motik: **Demistificirani C++**, Tisak Trebotić, Zagreb, 2001.
4. CACI Products Company: **Simscript II.5 Reference Handbook**, 1997.
5. CACI Products Company: **Simscript II.5 Programming Language**, 1997.
6. J. Joines, S. Roberts: Fundamentals of Object-Oriented Simulation, Proceedings of the 1998 Winter Simulation Conference, URL: http://www.informs-cs.org/wsc98papers/018.PDF
7. V. Rice, A. Marjanski, M. Markowitz, M. Bailkey: Object-Oriented SIMSCRIPT, Proceedings of the 37th Annual Simulation Symposium, April 18-22, 2004, Arlington, Virginia, pp. 178-186, URL: http://www.simprocess.com/docs/O-O_Simscrip_IEEE.pdf
8. C. Hostetter: Survey of Object Oriented Programming Languages, URL: http://www.rescomp.berkeley.edu/~hossman/cs263/paper.html (23/5/98)