PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET
METROPOLITAN
BEOGRAD

KI301 - KONSTRUISANJE SOFTVERA

Tehnike kodiranja

Lekcija 05

PRIRUČNIK ZA STUDENTE

KI301 - KONSTRUISANJE SOFTVERA

Lekcija 05

TEHNIKE KODIRANJA

L5: TEHNIKE KODIRANJA

L2: Tehnike kodiranja - Sadržaj

Sadržaj:

- Kontrola protoka u programu
- Kompleksnost podprograma
- Duboko ugneždavanje
- Neobične kontrolne strukture
- Tehnika strukturnog programiranja
- Tehnika proceduralnog programiranja

Poglavlje 1

Kompleksnost podprograma

MERA KOMPLEKSNOSTI PODPROGRAMA

Evo tehnike za merenje kompleksnosti podprograma.

Složenost nekog programa uveliko zavisi od upotrebe kontrolnih struktura, tj. od upotrebe if-instrukcija i petlji, i case-instrukcija. Evo tehnike zamerenje kompleksnosti podprograma:

1. Započeti sa brojem 1 koji označava osnovnu putanju kroz podprogram

2. dodaj 1 za svaku ključnu reč tipa if, while, for, and, or ,

3. i dodaj 1 i za svaki pojedinačni case-slučaj tj. za svaku pojedinačnu granu u okviru case-instrukcije

Ako je stepen kompleksnosti veći od 10, to je indikacija da podprogram treba razbiti na dva podprograma.

Druge mere kompleksnosti podprograma su:

- količina podataka upotrebljenih u podprogramu
- dubina ugneždavanja
- broj instrukcija podprograma
- broj ulaza i izlaza
- itd

POKAZNA VEŽBA 1

Pokazni primer.

Nivo kompleksnosti jeste broj CASE slučajeva tj. u ovom slučaju 12 i potrebno ga je razbiti na dva potprograma.

```
public class Mesec {
    public static void main(String[] args) {

        int mesec = 8;
        String mesecString;
        switch (mesec) {
            case 1:  mesecString = "Januar";
                     break;
            case 2:  mesecString = "Februar";
```

```
                    break;
            case 3:  mesecString = "Mart";
                    break;
            case 4:  mesecString = "April";
                    break;
            case 5:  mesecString = "Maj";
                    break;
            case 6:  mesecString = "Jun";
                    break;
            case 7:  mesecString = "Jul";
                    break;
            case 8:  mesecString = "Avgust";
                    break;
            case 9:  mesecString = "Septembar";
                    break;
            case 10: mesecString = "Octobar";
                    break;
            case 11: mesecString = "Novembar";
                    break;
            case 12: mesecString = "Decembar";
                    break;
            default: mesecString = "Mesec ne postoji.";
                    break;
        }
        System.out.println(mesecString);
    }
}
```

POKAZNA VEŽBA 2

Pokazni primer 2.

Primer kompleksnosti 4, zbog četiri case opcije.

```
public class Test {

   public static void main(String args[]) {
      // char kategorija = args[0].charAt(0);
      char kategorija = 'C';

      switch(kategorija) {
         case 'A' :
            System.out.println("Motocikl!");
            break;
         case 'B' :
            System.out.println("Automobil!");
            break;
         case 'C' :
            System.out.println("Teretno vozilo");
            break;
```

```
         case 'D' :
            System.out.println("Autobus");

         default :
            System.out.println("Ne postoji kategorija.");
      }
      System.out.println("Kategorija vozacke dozvole je " + kategorija);
   }
}
```

INDIVIDUALNA VEŽBA

Zadaci za samostalan rad.

1. Napisati programski kod čiji nivo kompleksnosti iznosi 5.

2. Napisati programski kod čiji nivo kompleksnosti iznosi 3.

3. Od čega zavisi nivo kompleksnosti?

Poglavlje 2

Neobične kontrolne strukture u programskom kodu

VIŠETSRUKI IZLAZ IZ PODPROGRAMA

Na osnovu iskustva preporučuje se da se minimizira broj izlaza iz podprograma, i da se koriste višestruki izlazi samo izuzetno.

Instrukcijareturn je u jeziku C kontrolna konstrukcija koja omogućuje programu da izadje iz podprograma po potrebi. Na taj način se podprogram završava kroz normalni izlazni kanal. Ponekad brzi izlazak iz podprograma se želi

- u cilju bolje čitljivosti i razumljivosti podprograma,
- ili pak da bi se u slučaju otkrivanja neke greške u procesiranju podataka, izbegla potreba za dopisivanjem programa
- ili pak u cilju pojednostavljenja kompleksnog procesiranja greške
- Znači želi se izaći iz podprograma odmah. To implicira da se ima višestruka naredba return.

U jeziku C++ i Java, instrukcija return omogućuje da izađete iz podprograma kad hoćete.Uobičajeno na kraju podprograma imamoreturn instrukciju. I ova instrukcija omogućuje povratak u podprogram iz kojeg je pozvan dotični pozvan podprogram. U nekim podprogramima, imamo višestruke izlaze iz podprograma, tj. višestruke instrukcije return, tj. instrukcije return na nekoliko mesta u podprogramu a ne samo na kraju podprograma. Medjutim, na osnovu iskustva preporučuje se da se minimizira broj izlaza iz podprograma, i da se koriste višestruki izlazi samo izuzetno, npr.

- kada se tako omogućuje bolja čitljivost tj. razumljivost programa
- za pojednostavljenje procesiranje greški

PRIMER VIŠESTRUKOG IZLAZA IZ PROGRAMA

Evo primera višestrukog izlaza iz programa.

Npr.

```
if (A < B)

{

return comparison1;
```

```
}

else if (A > B)

{

return comparison2;

}

return comparison3;

}
```

U principu, višestruki izlaz iz podprograma treba koristiti samo ako je neophodno, jer izaziva nedoumice.

GOTO-INSTRUKCIJA

Goto-instrukcija može da ponekad bude korisna.

Instrukcija goto može da bude korisna sa gledišta pisanja izvornog koda programa, da ga čini elegantnijim ponekad, ali kod kompilacije pomoću kompajlera može da izazove velike probleme jer je takav program teško analizirati s obzirom da imamo preskakanje u programu i postavlja se onda pitanje greški u logici programa. Dakle goto-instrukcija ponekad čini izvorni kod bržim i manjim tj. elegantnijim, ali posle kompilacije može doći do greški i sporosti i komplikovanosti mašinskog programa tj. mašinskog koda dobijenog posle kompilacije.

Goto-instrukcija može da ponekad bude korisna npr.

- kod alokacije resursa, npr. zatvaranje fajlova
- kod procesiranja grešaka
- ipak, u mnogim slučajevima može se izbeći upotreba goto-instrukcija pomoću upotrebe višestrukih kontrolnih struktura , naime standardna tehnika za eliminaciju goto-instrukcija je upotreba ugneždenih if-instrukcija
- upotrebom statusne varijable može se izbeći upotreba goto-instrukcije, gde statusna varijabla označava da li je podprogram u nekom stanju greške
- u Javi postoji instrukcija try-finally koja omogućuje eliminaciju goto.instrukcije u cilju ćišćenja resursa tj. fajlova (cleaning resources) u uslovima greške (error conditions)

Dakle, u cilju eliminacije goto-instrukcije, može se u 9 od 10 slučajeva koristiti:

- razbijanje podprograma na manje podprograme
- koristiti try-finally
- koristiti ugneždene if-instrukcije
- koristiti statusne varijable
- ali u onim ylučajevima gde ne možemo izbeći goto-instrukciju, treba je detaljno dokumentovati u programu i koristiti je ipak
- ako se koristi goto-instrukcija treba proveriti da li ima samo jedna goto-label u podprogramu, da li se koristi goto samo napred a ne i nazad

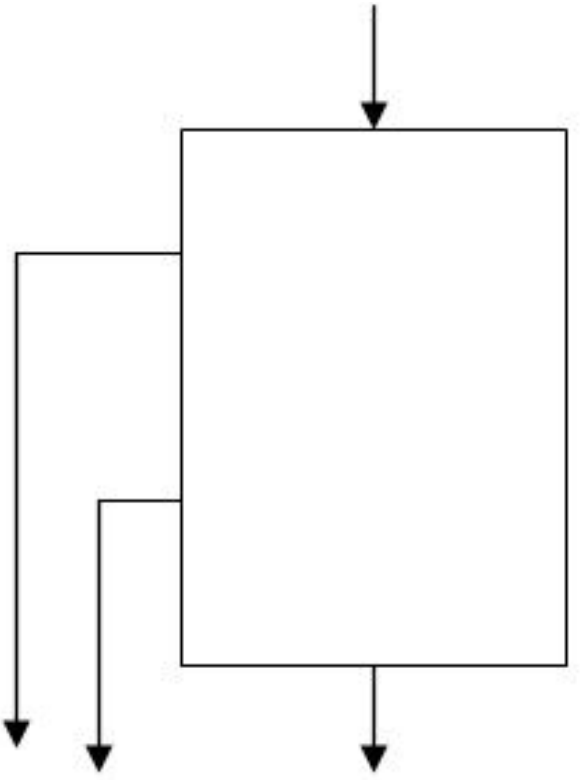
INSTRUKCIJA GO-TO :DISKUSIJA

Neko može da pomisli da je debata oko korišćenja instrukcije goto prošlost, ali nije. Naime instrukcija goto se još uvek koristi, nije izumrla.

Neko može da pomisli da je debata oko korišćenja instrukcije gotoprošlost, ali nije. Naime instrukcija goto se još uvek koristi, nije izumrla. Takodje, moderni ekvivalenti ove instrukcije, npr. višestruki izlaz iz podprograma, višestruki izlazi iz kružne tj petljaste instrukcije, npr. *do while*, takodje kod procesiranja greški. Opšti stav je da je izvorni kod bez gotokvalitetniji kod. Generalni stav je *da* je izvorni kod lošiji što ima više ove instrukcije.Naime, lakše je dokazati da je neki izvorni kod tačan ako nema instrukcije *goto*. Ova instrukcija ima vliki uiticaj na logičku strukturu koda. Podela koda na manje delove omogućuje proveru izv. koda, a kad imate instrukciju *goto,* onda je to teško ili nemoguće. Instrukcija *goto*, izaziva velike probleme kod optimizacije kompajlera. Ova instrukcija otežava analizu izv. koda, i ovo negativno utiče na optimizaciju kompajlera. Tako da, i ako upotreba *goto* daje efikasniji izvorni kod programa, kad se ima u vidu rad kompajlera, postavlja se pitanje da li izbaciti goto.

VIŠESTRUKI IZLAZ IZ PROGRAMA-DIJAGRAM

Na donjoj slici vidimo kako izgleda višestruki izlaz iz podprograma.



Medjutim, tačno je da ponekad instrukcija goto, čini izvorni kod manjim i bržim. Ipak u praksi, postoji princip da izv. kod treba da ide striktno od vrha do dna, i upotreba goto narušava taj

princip. Medjuim, svako pravilo ima izuzetaka, i ponekad, u specijalnim okolnostima, pažljiva upotreba goto može biti korisna. Ali, ne sme se koristiti bez rezerve, i bez diskriminacije. Nareba goto može da eliminiše potrebu za dupliranjem koda, a duplirajći kod može da pravi probleme kod modifikacije koda. Na kraju krajeva u mnogim modenim jezicima naredba goto nije eliminisana, npr C++. Ipak, u mnogim slučajevima, naredba goto se može zameniti i eliminisati, jer često se koristi samo da bi se skratilo programiranje.

INSTRUKCIJA GO-TO: PRIMER

Npr sledeći izv. kod u C++ sadrži nepotrebno goto.

Npr sledeći izv. kod u *C*++ sadrži nepotrebno goto:

do {getfile1 (file1, data);

if (eof(file1))

{goto exit;}

do(data); } while (data != -1);

exit;

Naime gotose može izbaciti na sledeći način:

getfile1 (file1, data);

while ((!eof(file1))&&((data != -1)))

{

do(data);

getfile1 (file1, data) }

Standardna tehnika za eliminaciju naredbe *goto* je korišćenje umreženih instrukcija *if* (***nested if statements***). Medjutim, kod procesiranja grešaka, i pisanja interaktivnog koda, nekad je korisno primeniti *goto.*

IZLAZAK IZ PETLJE:

Medjutim petlja sa puno naredbi break, ukazuje na nejasno razmišljanje, i u tom slučaju, treba razmisliti da umesto jedne petlje sa mnogo izlaza, se formira niz petlji.

Petlja (loop)je neformalni naziv za iterirajuću tj. kružnu logičku strukturu, strukturu koja izaziva ponavljanje odredjenog bloka u izv. kodu. Tipične petlje u jeziku C++ i Java su

for

while

do-while

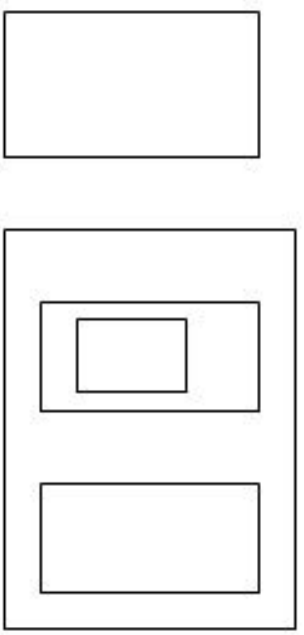
i korišćenje petlji je je jedan od najsloženijih zadataka programiranja, i važan faktor pravljenja kvalitetnog softvera. U jeziku *Visual Basic* postoji petlja sa izlazom, Loop-with-exit.

Kod petlji je važno da se izvrši inicijalizacija akumulisanih veličina, pošto petlje obično sadrže takve veličine, ali se može zaboraviti uraditi inicijalizacija. Rani izlazak iz petlje postoji, npr. u C++, naredba break.Ova naredba omogućuje da se petlja završi regularno, i nastavi rad programa na izlazu iz petlje. Medjutim petlja sa puno naredbi *break*, ukazuje na nejasno razmišljanje, i u tom slučaju, treba razmisliti da umesto jedne petlje sa mnogo izlaza, se formira niz petlji, umesto te jedne petlje sa puno izlaza. Takodje, instrukcija continuepostoji, koja omogućuje preskakanje instrukcija u petlji.

GRUPISANJE INSTRUKCIJA:

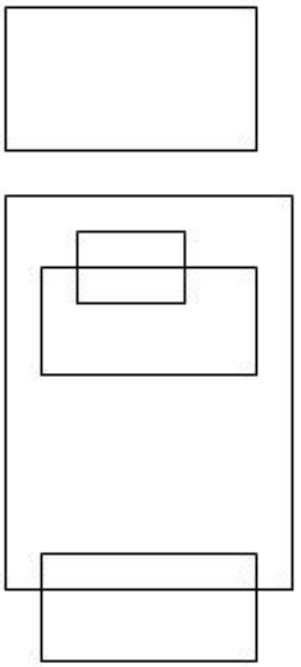
Instrukcije koje su u vezi treba grupisati, u blokove. Ako ima preklapanja na pojedinim blokovima, konstrukcija kao na drugoj slici, to je znak da je izv. kod loše organizovan.

Instrukcije koje su u vezi treba grupisati, u blokove. U vezi su instrukcije ako operišu na istim podacima, ili zavise jedni od drugih. Ako je izv. kod dobro organizovan u grupe, nema preklapanja, ali može biti umrežavanja, kao što je ilustrovano na donjoj slici. Ako ima preklapanja na pojedinim blokovima, konstrukcija kao na drugoj slici, to je znak da je izv. kod loše organizovan. Ako je to slučaj, treba reorganizovati izv. kod programa.



LOŠE ORGANIZOVAN PROGRAM

Evo kako izgleda loše organizovan program , špagetsko programiranje, koji treba reorganizovati.



POKAZNA VEŽBA 1

Pokazni primer.

Primer RETURN instrukcije:

```
public class Program {

    static int izracunajPovrsinu(int duzina, int sirina) {
        // Return izvrsava operaciju nad dva argumenta(variable).
        return duzina * sirina;
    }

    public static void main(String[] args) {

        // Upisivanje vrednosti u rezultat nakon izvrsene operacije
izracunajPovrsinu i ispisuje rezultat.
        int rezultat = izracunajPovrsinu(10, 3);
        System.out.println(rezultat);
    }
}
```

Kao primer uzet je programski kod koji računa površinu tako što se definiše metoda koja se naziva izracunajPovrsinu sa parametrima dužina i širina i preko return instrukcije računa površinu, a nakon toga se u okviru klase poziva sama metoda izracunajPovrsinu i unose se parametri za dužinu i širinu, pa se na kraju u okviru konzole ispisuje rezultat množenja ta dva parametra .

POKAZNA VEŽBA 2

Pokazni primer 2.

Primer 2 RETURN instrukcije:

```
public class Program {

    static int sabiranje(int broj1, int broj2) {
        // Return izvrsava operaciju nad dva argumenta(variable).
        return broj1 + broj2;
    }

    public static void main(String[] args) {

        // Upisivanje vrednosti u rezultat nakon izvrsene operacije sabiranje i
ispisivanje rezultata.
        int rezultat = sabiranje(5, 2);
        System.out.println(rezultat);
    }
}
```

Korišćenjem instrukcije RETURN se dobija zbir dva broja koji se definišu u okviru metode Sabiranje. Nakon što se definiše metoda, potrebno je da se u okviru klase pozove metoda sabiranje i unesu parametri potrebni za izvršavanje, nakon čega će se sam rezultat ispistati u konzoli.

ZADACI ZA INDIVIDUALNU VEŽBU

Zadaci za samostalan rad.

1, Upotrebiti return instrukciju na pravilan način u programskom kodu.

2. Prikazati deljenje dva broja korišćenjem return instrukcije.

3. Prikazati upotrebu break, continue, return, na konkretnim primerima.

4. Prikazati upotrebu throw, catch, instrukcije.

Poglavlje 3

Tehnika strukturnog programiranja

VRSTE PROGRAMIRANJA

Upotreba "goto" instrukcija (i njenih ekvivalenata) kvari koncept strukturnog programiranja, i treba ga izbegavati jer dobijamo program koji je teško razumeti, održavati, i teško kompajlirati.

Kod o.o. programiranja tj. data-centred programiranja programi se sastoje od klasa, a klase od skupa podataka i metoda (tj. funkcija, tj. podprograma) koje operišu sa podacima klase, i pri tome se koristi koncept kapsulizacije tj. privatnosti (data hiding). Za razliku od o.o.p., kod statement-centred tj. proceduralnog programiranja, programi se sastoje od niza podprograma a svaki program od niza instrukcija tj. od statements.

S druge strane,strukturno programiranje je termin koji označava jednostavnu ideju, da se program sastoji od delova gde svaki deo ima jedan ulaz i jedan izlaz, i da ovi blokovi su povezani jedan ispod drugog, tako da program se izvršava idući odozgo na dole, bez skakanja gore-dole. Tri komponente strukturnog programiranja su tri vrste blokova:

- sequence, tj. skup sekvencijalnih instrukcija koje se izvršavaju jedna posle druge
- selection, tj. if-instrukcija ili case-statement (tj. switch-instrukcija)
- iteration, tj. petlja,
- i ovi blokovi omogućuju da se formira bilo koji podprogram.
- Ali, upotreba"goto" instrukcija (i njenih ekvivalenata) kvari koncept strukturnog programiranja, i treba ga izbegavati jer dobijamo program koji je teško razumeti, teško održavati, i teško kompajlirati.

Strukturno programiranje je veliki korak napred u odnosu na nestrukturno programiranje. Kod nestruktuiranog programiranja, ceo program se sastoji od jednog bloka, ponavljanje operacija i grananje u programu se obavlja pomoću GOTO i JUMP naredbi. Takav program se teško razume, teško ispravlja, a lako se greši u kodu.
.Takođe, struktuirano programiranje deli veliki program u modularne procedure. Veliki program je podeljen u niz podprocedura. Subprocedure se pišu odvojeno, i zatim se posle testiranja ovih podprocedura one integrišu stavljanjem u pravi redosled. Ovakav dizajn programa, strukturisani dizajn programa, radikalno poboljšava: čitljivost prrograma, njegovu pouzdanost, i lakoću ponovnog korišćenja.

STRUKTURNO PROGRAMIRANJE

Strukturno programiranje koristi jednostavnu ideju da jedan podprogram treba da koristi kontrolne strukture koje imaju jedan ulaz i jedan izlaz.

Termin "structured programming" je uveden još daleke 1969.g. Strukturno programiranje koristi jednostavnu ideju da jedan podprogram treba da koristi kontrolne strukture koje imaju jedan ulaz i jedan izlaz. A kontrolna struktura sa jednim ulazom i jednim izlazom dakle je blok instrukcija u podprogramu koji ima samo jednu početnu tačku i samo jednu završnu tačku.

Jedan strukturisan program se sastoji od sledećih komponenti:

- sekvencijalni blokovi instrukcija
- selektivnih instrukcija tj. if-instrukcija, if-else-instrukcija i switch-instrukcija
- petljastih instrukcija tj. for-instrukcija i while-instrukcija

Glavna teza strukturnog programiranja je da svaki podprogram mora da se sastoji od ove tri vrste instrukcija (sekvencijalne, selektivne, i petljaste instrukcije), a druge kontrolne strukture kao što su:

- break, continue, return,
- throw, catch,
- i slične instrukcije, treba izbegavati, i ako se koriste onda ih treba vrlo kritički i pažljivo prostudirati i koristiti samo izuzetno kad je neophodno.
- Tehnika strukturnog programiranja daje programeru na upotrebu veoma korisne logićke uzorke u grafičkoj formi (flow-chart dijagrami), koje je lakše analizirati i proveravati nego sam izvorni kod programa.

PREPORUKE KOD STRUKTURNOG PROGRAMIRANJA

U “špageti-programranju” se ekstenzivno koriste skokovi (jumps) tj. goto-instrukcije, što rezultuje u programe koji se teško razzumeju i održavaju.

Strukturno programiranje je predloženo da pojednostavi proceduralni detaljni dizajn i pretvori ga u jednostavnu identifikaciju jednog limitiranog broja logičkih konstrukcija. Ovo pojednostavljuje dramatično kompleksnost programa tj. softvera, i omogućuje : Laku čitljivost tj. razumevanje programa, Lako testiranje, Lako održavanje programa . Strukturno programiranje (Structured programming) je concept koji ima za cilj da se poboljša jasnoća, kvalitet, I vreme razvoja kompjuterskog programa, pomoću ekstenzivne (velike) upotrebe:

- podprograma (subroutines), blokova instrukcija, petlji, selekcionih struktura (if-then, if-then-else),
- nasuprot tzv. “špageti-programranju”, gde se ekstenzivno koriste skokovi (jumps) tj. goto-instrukcije, što rezultuje u programe koji se teško razzumeju I održavaju. Kod

strukturnog programiranja sesvi programi vide kao kompozicija tri kontrolne structure: sekvence, selekcije, iteracije.

Elementi strukturnog programiranja su: “Kontrolne strukture”: “sekvence” (instrukcije ili podprogrami izvršavaju se u nizu), “selekcije” (if-then, if-then-else instrukcije), “iteracije” (for-petlje, do-petlje) , pri čemu se preporučuje

- da svaka petlja ima samo jednu ulaznu tačku I samo jednu izlaznu tačku
- “podprogrami” tj. “subroutines” koji predstavljaju posebne celine koje se mogu pozvati za izvršavanje (procedures, functions, methods, subprograms koji omogućuju da se predstave u program preko jedne instrukcije a to je poziv podprograma)
- “blokovi” omogućuju da se niz instrukcija predstavi kao jedna instrukcija
- Strukturno programiranje je moguće primeniti u bilo kojem programskom jeziku, proceduralnom (npr. FORTRAN) ili o.o. programskom jeziku (Nnpr. C++ ili Java).

“RANI IZLAZ” (EARLY EXIT) IZ PETLJE ILI FUNKCIJE

“Rani izlaz” (early exit) iz petlje ili funkcije (podprograma) predstavlja čestu devijaciju od strukturnog programiranja?

U mnogim slučajevima goto-instrukcija može se izbeći tj. zameniti primenom strukturnog programiranja tj. pomoću kontrolnih struktura “selekcije” i “iteracije” (ponavljanja). Ipak programski jezici nisu strogo strukturni, naime mnogi programski jezici dozvoljavaju npr. upotrebu return-instrukcije za rani izlaz iz podprograma, što rezultuje u višestruke izlazne tačke, umesto jedne izlazne tačke kako se zahteva strukturnim programiranjem.

“Rani izlaz” (early exit) iz petlje ili funkcije (podprograma) predstavlja čestu devijaciju od strukturnog programiranja. Tu se javlja problem da krajnje instrukcije se onda ne izvršavaju, npr. zatvaranje fajlova ili čišćenje memorije. Da bi se izbegli ovi problem mora se kod svakog izlaza vršiti pozivanje krajnjih instrukcija, što izaziva dodatne probleme. Primer ranog izlaza su npr. “izuzeci”, ili situacije kada podprogram nema vise šta daradi a nije još došao do kraja.

IZLAZ IZ UGNEŽDENIH PETLJI

U slučajevima kada je potreban izlaz iz ugneždenih petlji procesiranja, insistiranje na strukturnom programiranju, može izazvati neefikasnost u dizajnu, ili povećati mogućnost greške.

U slučajevima kada je potreban izlaz iz ugneždenih petlji procesiranja, dogmatska upotreba strukturnog programiranja tj. insistiranje na strukturnom programiranju, može izazvati neefikasnost u dizajnu, ili povećati mogućnost greške. U tom slučaju, treba

- Izvršiti poseban dizajn izlaza iz umreženih petlji, što pdrazumeva narušavanje tj. odstupanje od striktne forme strukturnog programiranja, ali na jedan kontrolisan i proveren način,
- ili pokušati modifikaciju dizajna tako da se izbegne izlaz iz umreženih petlji

Kod ugneždenih kružnih konstrukcija (nested loops), kada je potrebno definisati izlaznu opciju iz te komplikovane mreže, ako strukturno programiranje rezultira u kompleksan dizajn, onda se može odstupiti od strukturnog programiranja.

POOKAZNI PRIMER 1

Pokazni primer.

Jedan strukturisan program se sastoji od sledećih komponenti:
1.Sekvencijalni blokovi instrukcija
2.Selektivnih instrukcija tj. if-instrukcija, if-else-instrukcija i switch-instrukcija
3.Petljastih instrukcija tj. for-instrukcija i while-instrukcija

```
class IfElse {
   public static void main(String[] args) {
      int broj = 10;

      if (broj > 0) {
         System.out.println("Broj je pozitivan.");
      }
      else {
         System.out.println("Broj je negativan.");
      }
}
```

POKAZNI PRIMER 2

Primer višestrukog izlaza.

Evo primera.

```
if (A < B)
{
return comparison1;
}
else if (A > B)
{
return comparison2;
}
return comparison3;
}
```

INDIVIDUALNA VEŽBA

Zadaci za samostalni rad.

1. Prikazati upotrebu break, continue, return, na konkretnim primerima.Da li je to dobar ili loš način strukturnog programiranja. Objasniti.

2. Prikazati upotrebu throw, catch, instrukcije. Da li je to dobar ili loš način strukturnog programiranja. Objasniti.

Poglavlje 4

Tehnika proceduralnog programiranja

PSEUDOKOD

Tekstualni 'recept', tj. opis procedure, naziva se i "pseudokod", pseudocode, ili pseudo-dizaln- jezik (pseudo design language -PDL).

Neki kompjuterski program može biti posmatran jednostavno kao lista instrukcija koje trebaju da se izvrše po odredjenom redu, tj. kao jedna procedura koja treba da se izvrši u nizu koraka. Programski jezici koji to omogućavaju se zovu 'imperativni jezici'. Drugi naziv za kompjutersku proceduru je podprogram (subroutine).Na primer, neki programer može da kreira proceduru tj. podprogram da izračuna koren zbira kvadrata elemenata jedne matrice tj. niza brojeva.Sledeća formula se koristi za tu svrhu: $S = [(\sum xi^2)/(n)]^{1/2}$

Da bi se opisala procedura za obavljanje ovog proračuna, sledeći opis programa može da se napravi:

Zadati da je SUM i SUMSQUEARES jednako 0.0

Zadati n = dimenzija matrice

Startovati sa prvim x, tj x1, i nastaviti dok se svi xi ne procesiraju.

Zadati SUMSQUARES =SUMSQUARES+xi2

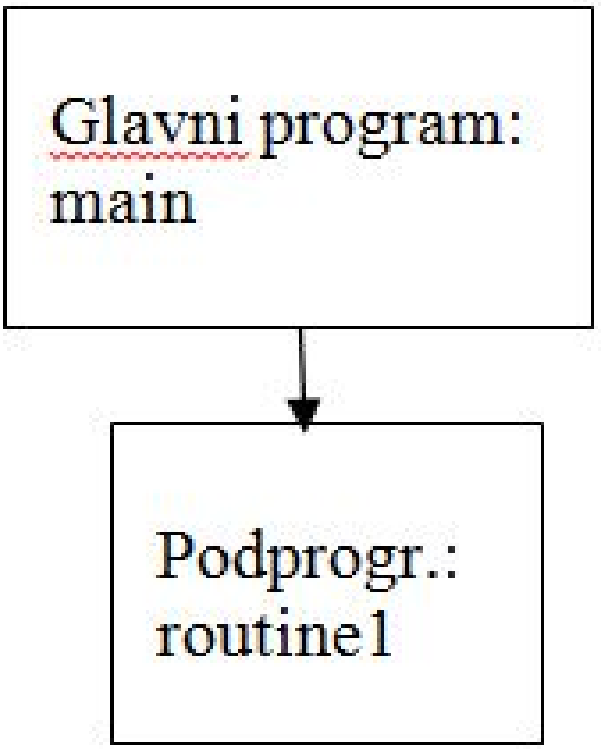
Kraj petlje, tj. kraj iteriranja

Vratiti (kao rezultat rada procedure) SquareRoot od (SUMSQUARES)/(n)

Ovo je 'recept', opis procedure, tj. tzv. "pseudokod", pseudocode, za proračun korena zbira kvadrata elemenata jednog niza tj. sekvence brojeva. Na osnovu ovog pseudokoda, može se u nekom programskom jeziku, npr. u jeziku FORTRAN ili Java ili C/C++, napisati izvorni kod koji predstavlja implementaciju ove procedure. Java je dizajnirana da služi kao objektno orijentisani jezik, i objektna orijentacija omogućuje više sofisticirane načine da se kreiraju moduli nego što to nudi proceduralni jezik FORTRAN. Bez obzira, Java se isto može koristiti i kao proceduralni jezik.

UPOTREBA KOMPJUTERSKE PROCEDURE

Drugi naziv za kompjutersku proceduru je podprogram (subroutine).



Proceduralno programiranje nudi standardna rešenja za računarske aplikacije, u blokovima izv. koda, koji se mogu pozvati pomoću imena podprograma, koji nude višestruku aplikaciju, ponovnu upotrebu, pomoću liste argumenata, tj. ulaznih podataka. Promenljive koje su deklarisane u okviru procedure tj. podprograma, kao npr. 'sum', su vidljive jedino u okviru podprograma. Jednom kada se ima gornji podprogram, on se može koristiti ponovo, i opet ponovo. Takva ponovna upotreba može se ostvariti:

- uključivanjem tog podprograma u bilo koji program koji zahteva takav podprogram.
- Ili se može taj podprogram staviti u biblioteku podprograma, odakle se mogu pozivati pomoću imena podprograma.

VIDEO VEŽBA

Video vežba - Modelovanje) pomoću pseudokoda (Program Design With Pseudocode Computer Program Language).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

INDIVIDUALNA VEŽBA

Zadaci/Pitanja za individualnu vežbu

1. Koje su prednosti upotrebe PDL?

2. Dati jedan primer primene PDL-a?

3. Koje su konvencije upotrebe pseudokoda tj. PDL-a?

3. Opisati upotrebu kompjuterskih procedura kod proceduralnog programiranja?

4. Dati primer da sej podprogram stavi u biblioteku podprograma.

DOMAĆI ZADATAK 5

Domaći zadatak 5 se dobija direktno od asistenta nakon poslatog prvog domaćeg zadatka ili na poseban zahtev studenta.

Domaći zadatak 5 je individualan za svakog studenta i dobija se po definisanim instrukcijama.
Domaći zadatak se imenuje:
SE211-DZ05-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br.Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta: nebojsa.gavrilovic@metropolitan.ac.rs sa naslovom (subject mail-a) SE211-DZ05.

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.
Napomena:
Domaći zadaci treba da budu realizovani u zadatku navedenom razvojnom okruženju i da predstavljaju jedinstveno rešenje svakog studenta. Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

Poglavlje 5

Primena flow-chart dijagrama

“FLOW-CHART” DIJAGRAM

Umesto termina “flowchart” tj. “flow-chart” koristi se i termin flow-diagram.

Flow-chart dijagram se koristi za predstavljanje algoritama i procesa, gde se pojedini koraci prikazuju kao “kutije” (kvadrati, pravougaonici, ….) , a redosled ovih koraka je prikazan preko strelica koje povezuju “kutije”. Ovakav dijagram predstavlja model rešenja za neki postavljeni problem. Flow.chart dijagrami se koriste kod analize, dizajna (modelovanja), dokumentovanja programa ili procesa u raznim oblastima. U stvari, flow-chart dijagrami se koriste kod dizajna (modelovanja) I dokumentovanja složenih programa I procesa. Pomoću vizualizacije posmatranog problema, omogućuje se bolje razumevanje problema, I otkrivanje nedostataka, uskih grla, I dugih manje očiglednih osobina problema. Postoje dve vrste “kutija” koje se koriste kod flow-chart dijagrama:

- neki procesirajući korak, obično se naziva “aktivnost”, I označava se kao pravougaonik na dijagramu
- odlučivanje, koje se prikazuje kao dijamantni znak (tj. romb)

Umesto termina “flowchart” tj. “flow-chart” koristi se i termin flow-diagram.

Jasno je da grafičko orudje kao što je karta protoka (flowchart) je vrlo korisno jer slikovito i brzo informišu o nizu detalja, npr. proceduralnih detalja. Medjutim ako se pogreši kod korišćenja ovog orudja, ovo vodi greškama u softveru, pa treba ovo orudje pažljivo koristiti. Karta protoka je jednostavno orudje. Jedan pravougaoni blok označava jedan procesirajući korak, ili uzastopni tj. sekvencijalni niz procesirajućih koraka. Dijamantni znak označava logički uslov a strelice pokazuju kontrolisani protok procesiranja podataka.

FC DIJAGRAMI - PRIMERI

Grafičko orudje kao što je karta protoka (flowchart) je vrlo korisno jer slikovito i brzo informišu o nizu detalja, npr. proceduralnih detalja.

Na Slici 1 vidimo sekvencijalnu konstrukciju od dva bloka koji jednostavno idu jedan za drugim, bez ikakve mogućnosti grananja. Medjutim, uslovna konstrukcija‘if-then-else’označena dijamantnim znakom obavlja logičku operaciju, i ako je uslov ispunjen, npr. uslov x>5?, onda jedna grana dijagrama se procesira, a ako nije ispunjen uslov onda se

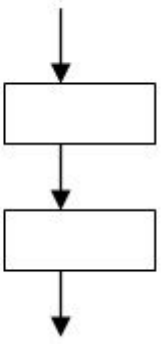
druga grana dijagrama procesira. Struktura koja označava repeticiju tj. iteracije tj. kruženje u procesiranju, ima dve slčne forme, ‘uradi-dok’ i ‘ponovi-dok’.

Na Slici 2 je prikazana logićka struktura (tj. upravljačka struktura) višestruke selekcije tj. *select-case*. Ova logička struktura koristi se za obavljanje selekcije, tako što se neki parametar testira sukcesivno, sukcesivnim odlučivanjem, dok se uslov istinitosti ne ispuni, i onda se obavlja odredjena operacija. Ova konstrukcija je u stvari varijacija strukture iff-then-else, ali je ponekad efikasnija od nje.

Naravno, logičke strukture se mogu kombinovati, umrežavati, kao što je ilustrovano na slici 3. Na ovaj način, vrlo komplikovane logičke šeme mogu biti modelovane. Na Slici 3 imamo jednu repeticionu strukturu ‘repeat-until’ (kružnu konstrukciju) koja je deo jedne uslovne konstrukcije. Takodje, još jedna uslovna konstrukcija je deo prethodne uslovne konstrukcije. Na kraju, uočava se da prvi blok i ostatak programa predstavljaju dva bloka u nizu. Ponekad su karte protoka nepregledne i nepraktične, mogu zauzimati veliki prostor i izazvati konfuziju. Zato se ovi dijagrami mogu koristiti u saradnji sa drugim orudjima detaljnog dizajna. Karte protoka se mogu prikazati u više slojeva na više strana, gde se krugovima npr. označavaju ‘pozivi’ pojedinih delova.

SEKVENCIJALNE INSTRUKCIJE

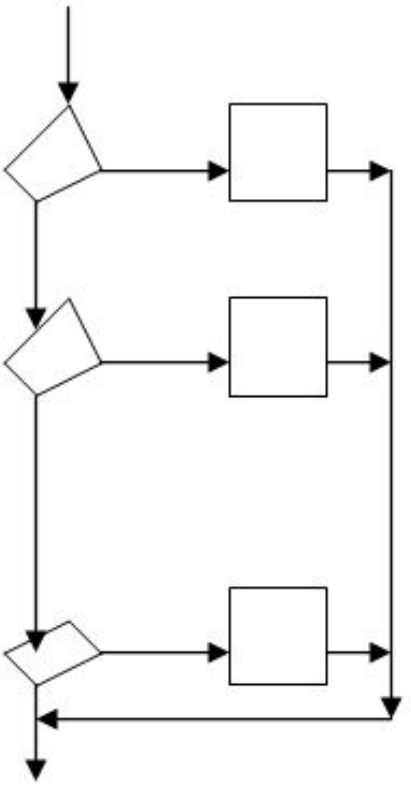
Slika daje prikaz sekvencijalnih instrukcija.



Slika 5.1

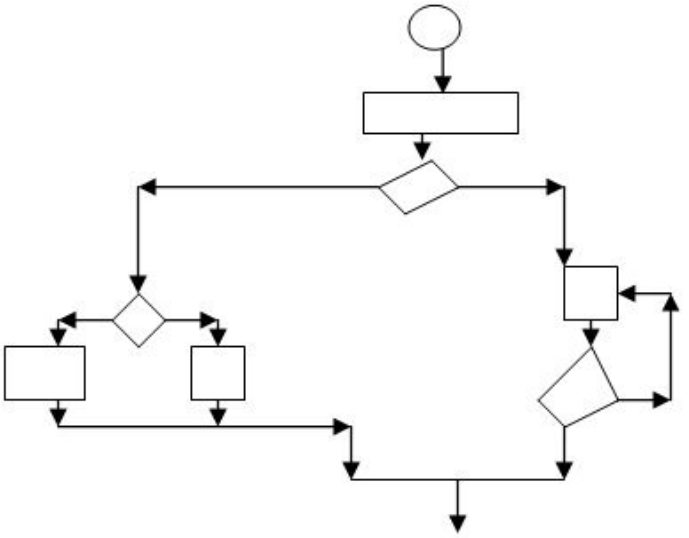
LOGIČKA STRUKTURA SELECT-CASE

Slika ilustruje flow-chart dijagram select-case instrukcije.



UMREŽENE LOGIČKE STRUKTURE

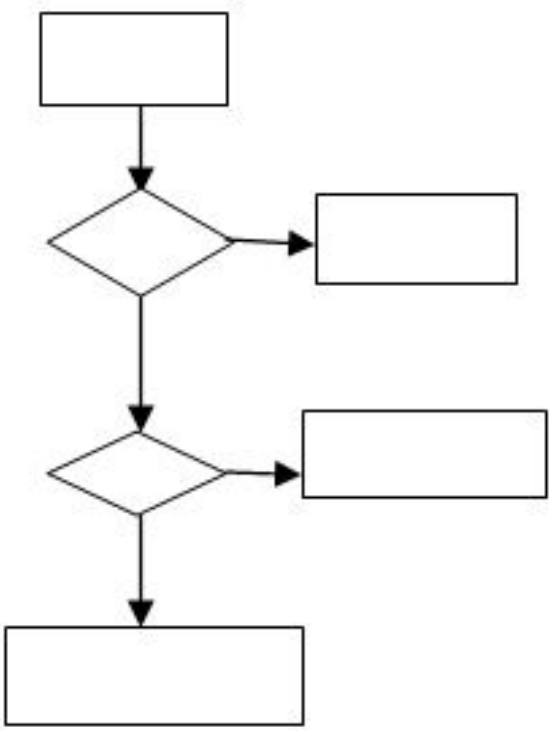
Na slici su složeni flow-chart dijagrami. Ovde se vide i selekcione instrukcije (rombovi) i petljaste instrukcije (for-instrukcije, while-instrukcije) koje omogućuju kruženje u programu.



Slika 5.2

PRIMER FLOWCHART DIJAGRAMA SA VIŠESTRUKIM IZLAZOM

Slika prikazuje višestruki izlaz iz programa.



Slika 5.3

NOTACIJA FLOW-CHART DIJAGRAMA

Ovde se bavimo opisom simbola flow-chart dijagrama.

Flow-chartdijagrami su popularno sredstvo za opis kompjuterskih algoritama. Modernije tehnike kao što je npr. UML dijagrami aktivnosti se smatraju ekstenzijom flow-chart dijagrama. Kod flow-chart dijagrama je problem predstavljati skokove u programu (*goto*-instrukcije). Alternativa flow-chart dijagramu je npr. pseudo-code jezik (pseudo-code language- PDL). Postoji vise vrsta flow-chart dijagrama: *program flowchart* koji pokazuje upravljanje procesiranjem u nekom programu, zatim data flowchart koji prikazuje upravljanje tokom podataka.

Simboli koji se koriste kod flowchart-dijagrama su sledeći:

- Simbol početka ili kraja: krug,
- Strelica: pokazuje da upravljanje procesom/programom prelazi sa jednog elementa na drugi,

- Procesirajući korak: symbol je pravougaonik,
- Podprogrami (“subroutines”) koji se koriste da prikažu složene procesirajuće korake koji se mogu detaljno prikazati u odvojenom flow-chart dijagramu: prikazani sa duplim vertikalnim linijama na pravougaoniku,,
- Ulaz/izlaz (npr. učitaj ulazni podatak, ili prikaži/štampaj izlazni podatak) se prikazuje pomoću simbola paralelograma,
- Odluke/uslovi se prikazuju pomoću romba (“dijamantni” znak),
- Simbol prikupljanja vise strelica gde samo jedna strelica izlazi a vise strelica ulazi (“junction symbol”): crni krug.

POREDJENJE UML DIJAGRAMA AKTIVNOSTI (UML DA) I FC

Glavna razlika izmedju AD i FC je da AD podržavaju paralelne procese (paralelne poslovne procese i paralelne algoritme).

Dijagrami aktivnosti (Activity diagrams-AD) igraju sličnu ulogu kao i FC (Flow Charts). Glavna razlika izmedju AD i FC je da AD podržavaju paralelne procese (paralelne poslovne procese i paralelne algoritme). AD se mogu koristiti kao FC koji su prilagodjeni za UML. Kada koristiti AD?:

-AD se često koriste da opišu use-case-ove. Uobičajena tekstualna forma use-case-a može se iskoristiti za crtanje AD.

-AD se takodje koriste za opis algoritama metoda tj. metoda klasa, obično ne za sve metode već samo za komplikovanije metode tj. algoritme metoda klasa

AD se koriste da opišu kompleksne procese. AD se sastoje od niza aktivnosti i tranzicija od jedne aktivnosti do neke druge aktivnosti. AD opisuju sekvencijalne, selekcione i iterativne aktivnosti, kao i paralelne aktivnosti. AD se koriste da grafički prikažu procesiranje nekog use-case-a ili procesiranje u okviru neke klasne metode tj. klasne operacije. AD se može koristiti kao alternativa za use-case opis.Use-case-ovi opisuju glavne aktivnosti sistema sa gledišta korisnika, i na taj način se opisuje funkcionalnost sistema. Funkcionalnost sistema je takođe opisana preko operacija tj. metoda klasa.

VIDEO TUTORIJAL O FC DIJAGRAMIMA

Problem Solving Techniques #8: Flow Charts

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO TUTORIJAL

Make a FlowChart in Microsoft Word 2013

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO VEŽBA O UML DA

Activity Diagrams Basic Symbols

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

IMPLEMENTACIJA FC DIJAGRAMA

Transformacija FC u programski kod je jednostavna, naime svaki blok na FC se može direktno prevesti programski kod.

FC je praktično ekvivalentan sa pseudokodom. Naime, svaki FC se može jednostavno transformisati u pseudokod. A pseudokod omogućuje direktno pisanje programskog koda na osnovu pseudokoda. Ovo znači, da se praktično direktno na osnovu FC može pisati programski kod, naime, svaki blok na FC se može jednostavno transformisati u programski kod. Svaki FC se sastoji od sekvencijalnih i petljastih i kondicionalnih blokova, a svaki takav blok se jednostavno prevodi u programski kod, pomoću if-instrukcija i sekvencijalnih instrukcija i petljastih instrukcija. Npr. if-blok se prevodi u if-instrukciju, while-blok u petljastu instrukciju, a običan pravougaoni blok u niz sekvencijalnih instrukcija. Pri tome, pojedini blokovi ako treba mogu se detaljnije opisati pomoću pseudokoda, i zatim na osnovu pseudokoda se jednostavno tj. direktno može napisati napisati detaljni programski kod. U suštini, FC je samo drugi oblik predstavljanja pseudokoda, što znači da se na osnovu FC jednostavno i direktno pisati programski kod.

ZADACI

Zadaci za vežbu

Zadaci:

- Nacrtati FC dijagram koji sadrži for- petlju i switch-instrukciju.
- Nacrtati FC dijagram koji sadrži while-petlju i if-then-else instrukciju.

Poglavlje 6

Zaključak

ZAKLJUČAK

- Tri komponente strukturnog programiranja su tri vrste blokova:
  sequence, tj. skup sekvencijalnih instrukcija koje se izvršavaju jedna posle druge
  selection, tj. if-instrukcija ili case-statement (tj. switch-instrukcija)
  iteration, tj. petlja,
  i ovi blokovi omogućuju da se formira bilo koji podprogram.
- Ali, upotreba "goto" instrukcija (i njenih ekvivalenata) kvari koncept strukturnog programiranja, i treba ga izbegavati jer dobijamo program koji je teško razumeti, teško održavati, i teško kompajlirati.
- U jeziku C++ i Java, instrukcija return omogućuje da izađete iz podprograma kad hoćete.Uobičajeno na kraju podprograma imamo return instrukciju. I ova instrukcija omogućuje povratak u podprogram iz kojeg je pozvan dotični pozvan podprogram. U nekim podprogramima, imamo višestruke izlaze iz podprograma, tj. višestruke instrukcije return, tj. instrukcije return na nekoliko mesta u podprogramu a ne samo na kraju podprograma.
- Medjutim, na osnovu iskustva preporučuje se da se minimizira broj izlaza iz podprograma, i da se koriste višestruki izlazi samo izuzetno, npr.
  kada se tako omogućuje bolja čitljivost tj. razumljivost programa ili za pojednostavljenje procesiranje greški.
- Glavna teza strukturnog programiranja je da svaki podprogram mora da se sastoji od ove tri vrste instrukcija (sekvencijalne, selektivne, i petljaste instrukcije), a druge kontrolne strukture kao što su break, continue, return, throw, catch, i slične instrukcije, treba izbegavati, i ako se koriste onda ih treba vrlo kritički i pažljivo prostudirati i koristiti samo izuzetno kad je neophodno.

REFERENCE

1. Code Complete: A practical handbook of software construction, by S. McConnell, Microsoft Press, ISBN 0-7356-1967-0, https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670
2. SWEBOK-V3, https://www.computer.org/web/swebok/v3
3. Theory and Problems of Software Engineering - Schaum’s Outline Series, by David Gustafson, ISBN 0-07-137794-8, http://www.mhebooklibrary.com/doi/abs/10.1036/0071377948

Linkovi:

https://www.youtube.com/watch?v=D0qfR606tVo , Writing Good Beginner Pseudocode