Osnove programiranja

Dr Milan Popović, soba 214, e-mail: milan.popovic@bbs.edu.yu

Konsultacije: Ponedeljak 11:30 – 13:00, Utorak 11:00 – 13:00

1. UVOD

- O čemu se radi u ovom predmetu
- Šta je cilj ovog predmeta
- Kakvo je predznanje potrebno
- Šta vam je potrebno za ovaj predmet
- Šta ćete saznati u ovom predmetu
- Koliko traju predavanja
- Šta je sadržaj ovog predmeta
- Uputstvo za učenje

O čemu se radi u ovom predmetu?

Kompjuterski programi se dizajniraju (projektuju, razvijaju) sa ciljem da se pomoću njih rešavaju kompjuterski rešivi problemi. Pretraživanje velikih baza podataka, izračunavanje plata, ažuriranje medicinskih kartona, samo su neki primeri uobičajene primene kompjutera (računara). Postoje i kompjuterski nerešivi problemi, ali oni neće biti predmet našeg razmatranja. Broj kompjuterski rešivih problema je beskonačan, a broj praktičnih primena računara je ograničen samo našom maštom, znanjem i veštinom.

Za sve primene kompjutera postoji jednistven scenario kako se projektuje programski sistem kojim se kompjuterska mašinerija (hardver) stavlja u željenu funkciju.

Takav scenario se može izraziti sledećim fazama (koracima):

0. Originalna ideja o temi, zadatku ili problemu se što preciznije definiše
0. Temeljna analiza teme, zadatka, problema
0. Projektovanje rešenja (sistema)
0. Implementacija (konstrukcija) rešenja (sistema)
0. Testiranje i verifikacija rešenja (sistema)
0. Održavanje i unapređivanje rešenja (sistema)

Šta je cilj ovog predmeta?

U ovom predmetu biće prikazane neke metode, alati i tehnike koje se koriste u fazama 3 i 4 gornjeg scenarija, sa posebnim akcentom na projektovanju programa korišćenjem proceduralnih jezika treće generacije kao što su Visual Basic, C++, Java. Ovaj predmet treba da da dobru osnovu za lako savladavanje bilo kog programskog jezika treće generacije. Čak iako student nema nameru da postane profesionalni

programer, u ovom predmentu moze naći neke korisne metode i tehnike koje će poboljšati njegovu opštu sposobnost rešavanja problema.

Kakvo je predznanje potrebno?

Nije potrebno nikakvo prethodno kompjutersko znanje i iskustvo za uspešno savladavanje ovog predmeta. Svi koncepti i termini biće objašnjeni na pogodan i jasan način. Pitanja i vežbe koji su deo ovog teksta su od velike koristi za proveru i utvrđivanje usvojenih znanja.

Šta vam je potrebno za ovaj predmet?

Za uspešno savladavanje ovog predmeta potrebno je (i dovoljno) da redovno pratite nastavu, vežbe i uradite ono što se od vas zahteva na časovima. Potrebni kompjuteri su raspoloživi u laboratoriji (ERC-u), a posedovanje vlastitog kompjutera je prednost. Softverski alati koji će biti korišćeni za praktične vežbe su iz tzv. pablik domena (public domain), što znači da se mogu slobodno kopirati i/ili daunlodovati (download) sa Interneta.

Šta ćete dobiti izučavanjem ovog predmeta?

Pored očigledne koristi od još jednog položenog ispita, nakon uspešnog završetka rada na ovon predmetu student će dobiti:

- Dobru osnovu za detaljnije izučavanje programski jezika kao što su C, C++, C#, Java. Svi ovi jezici se danas intenzivno koriste za razvoj softverskih sistema.
- Dobru osnovu za studiranje drugih predmeta iz oblasti poslovne informatike

Koliko traju predavanje?

Trajanje predavanja predviđeno je kroz 15 tročasovnih blokova predavanja i dvočasovnih vežbi, što čini ukupno 75 časova nastave. To vreme je dovoljno za savladavanje ukupnog materijala. Samostalan rad u otprilike istom obimu trebalo bi da rezultira uspešnim završetkom.

Šta je sadržaj ovog predmeta?

0. **Uvod u predmet**
   Predstavljanje i načini komunikacije
   Plan i program predmeta
   Način polaganja
0. **Kompjuteri, programi, programski jezici**
   Osnovni elementi kompjuterskog sistema: hardver, softver, komunikacije
   Kompjuterski programi
   Generacije programskih jezika
   Model kompjuterskog programa
   Podaci i informacije
0. **Podaci i informacije**

Obrada podataka (ažuriranje, sortiranje i pretraživanje)
Prenos i prezentacija podataka
Vrste podataka i način kodiranja
Osnovni tipovi podataka (int, float, boolean)
Strukture podataka (nizovi, liste, stekovi, stabla, grafovi, heš tabele)

0. **Interakcija čovek-kompjuter**
   Interfejsi
   Linijski interfejsi
   Grafički interfejsi
   Kompleksni interfejsi
0. **Datoteke i baze podataka – čitanje, pisanje, ažuriranje**
   Serijske datoteke
   Sekvencijalne datoteke
   Indeks-sekvencijalne datoteke
   Datoteke sa direktnim pristupom (hešing)
   Relacione baze podataka
0. **Algoritmi i algoritamske strukture**
   Pojam algoritma i teorema o programskoj strukturi
   Sekvenca, selekcija, iteracija, rekurzija
0. **Razvoj algoritama**
   Metodologija razvoja algoritama
   Načini prikazivanje algoritama:
      Opisno – prirodnim jezicima
      Blok dijagrami
      Pseudokod
      Nassi-Schnaiderman diajgrami
      JSD (Jackson Structured Diagrams) dijagrami
   Modularnost i strukturalnost algoritama (programa)
   Testiranje algoritama (programa)
0. **Programska okruženja – platforme za razvoj i eksploataciju softvera**
   Operatini sistemi za stand-alone kompjutere i mreže
   Kompajleri, debageri, linkeri i loderi
   Razvojna i run-time okruženja
   Open source
   Internet programiranje
0. **Objektno orjentisano programiranje**
   Principi OO programiranja
   Apstrakcija, enkapsulacija, nasleđivanje, polimorfizam
   Klase, objekti, atributi i metode
   OO jezici i metodologije
0. **Metode programiranja**
   Proceduralno, funkcionalno logičko programiranje
   Strukturno i modularno programiranje
0. **Upravljanje softverskim projektima**
   Životni ciklus programa (softvera)
   Metodologije za razvoj softvera
   Savremene metode (extremno i programiranje u paru)
   Timski rad
0. **Praktična nastava**
   Laboratorijske vežbe na kompjuterima

Seminarski rad: programski projekat

LITERATURA

1. Osnove programiranje, Milan Popović, BPS 2007
2. .NET Framework Version 2.0 Redistributable Package (x86), Microsoft Corporation, free download
3. Visual C# 2008 Express Edition, Microsoft Corporation, free download
4. SQL Server 2008 Express Edition, Microsoft Corporation, free download

Uputstvo za učenje

U svakoj lekciji naći ćete pitanja kao što su:

2. definišite šta se podrazumeva pod kompjuterskim programom
2. definišite šta se podrazumeva pod kompjuterom
2. ukratko opišite istorijski razvoj kompjuterskih (programskih) jezika
2. skicirajte jednostavan model kompjuterskog programa i njegovog okruženja
2. objasnite kako se podaci razlikuju od informacija
2. definišite termin algoritam
2. napravite listu karakteristika algoritma
2. opišite šta je programske sekvenca
2. opišite if ... then i if ... then ... else konstrukcije
2. opišite razliku između objekta i klase.

U čitavom kursu ima oko 100 takvih pitanja koja su sačinjena tako da ih možete lako obraditi. Taj skup pitanja vrlo reprezentativno prikazije skup znanja koja su vam potrebna o programitranju u jezicima treće generacije. To su zapravo pitanja koja vam se mogu postavljati i prilikom intervjua za posao u oblasti razvoja softvera i informacionih sistema.

Materijal je podeljen u deset tema koje su kompaktne i detaljno obrađuju neku od oblasti programiranja. Predavanja se logično nastavljaju jedna u drugu, tako da se koncepti razvijaju u prirodnom redosledu. U većini predavanja data su pitanja na koja treba da odgovorite, kao i vežbe koje treba samostalno da uradite. Ova pitanja i vežbe imaju za cilj da testiraju vaše napredovanje u učenju, kao i da vam daju priliku da praktikujete novostečena znanja i veštine. Pitanja i vežbe su za vašu korist i ako ih korektno uradite možete biti sigurni da ste ispunili očekivanja i postigli cilj ovog predmeta.
Provera znanja se realizuje kroz pismeni ispit u trajanju od 2 sata.

Studenti su, takođe, obavezni da urade i jedan seminarski rad - programski projekat.

Na kraju semestra rezultati pismenog ispita, seminarskog rada i aktivnsoti studenata tokom nastave i vežbi rezultiraju konačnom ocenom prma sledećoj formuli:
Pismeni ispit 60 poena
Seminarski rad 20 poena
Aktivnost u nastavi 10 poena
Aktivnost na vežbama 10 poena

2. Kompjuteri, programi, programski jezici

Pošto je cilj ovog predmeta izučavanje kako se projektuju programi jezicima treće generacije (3GL), može biti od koristi podsećanje na neke opšte kompjuterske termine koji će biti korišćeni. Na primer, šta ćemo podrazumevati kad kažemo kompjuter, šta je kompjuterski program, šta su to generacije programskih jezika?

Nakon izučavanja ove lekcije bićete u stanju da:

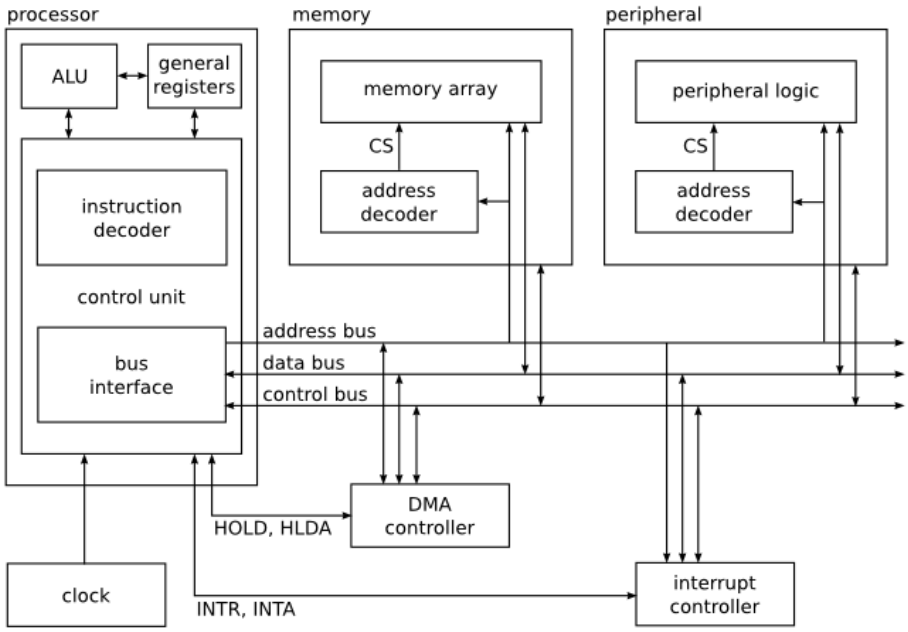
1. definišete šta je to kompjuter
2. definišete šta je to komjuterski program
3. opišete razvoj kompjuterskih (programskih) jezika
4. skicirati jednostavan model kompjuterskog programa i njegovog okruženja
5. objasniti kako se podaci razlikuju od informacija

Osnovni elementi kompjuterskog sistema: hardver, softver, komunikacije

Kompjuteri su, verovatno, najsloženije mašine koje je čovek do sada stvorio. Ako bi hteli da definišemo kompjutre mogli bi smo reći:

- Kompjuter ima ulazne i izlazne uređaje.
- Akompjuter ima centralnu procesorsku jedinicu (poznatu kao CPU) za izvršavanje aritmetičkih i logičkih operacija.
- Kompjuter ima memoriju za smeštanje programa i podataka.
- Kompjuter može da izvršava nizove instrukcija (naredbi, komandi).

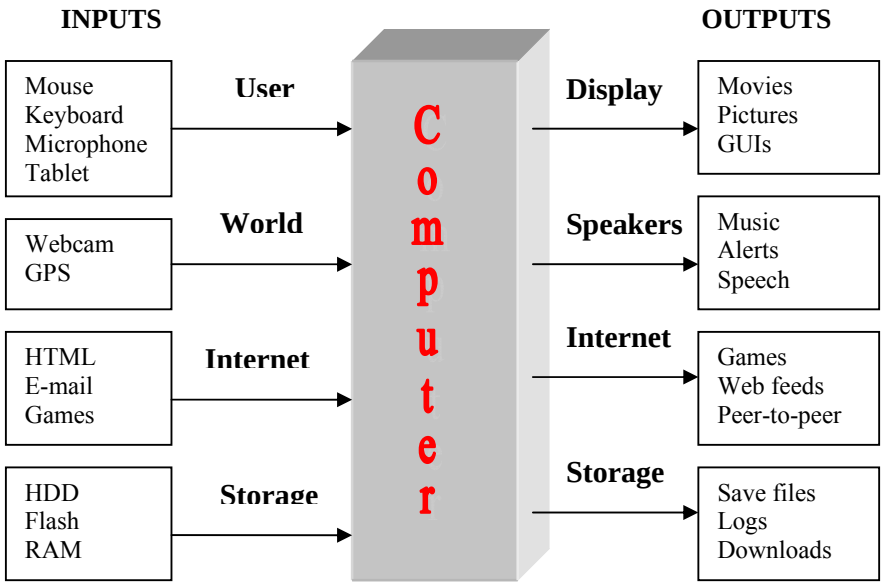
Na sledećoj slici prikazana je takozvana osnovna arhitektura kompjuterskog sistema onako kako je proučavaju hardverski profesionalci.



Slika 2.1 Osnovna arhitektura kompjtera

Na slici se uočavaju gore pomenuti delovi kompjutea – procesor, memorija, periferni uređaji kao i spojni putevi (magistrale) koji povezuju delove u jedinstvenu celinu. Preko magistrala se odvija sva komunikacija među delovima kompjutera, a magistrale (ili bus-ovi) služe za prenos podataka, adresa i kontrolnih komandi. Gornja arhitektura prikazuje takozvani stand-alone kompjuter koji nije povezan sa drugim kompjuterima. Danas je, međutim, skoro uvek slučaj da je kompjuter na kome radimo u lokalnoj ili globalnoj (Internet) mreži.

Za potrebe programera i softver inženjera pogodnija je malo uprošćenija ahitektura, koja u sebi sadrži i mrežni deo prikazana koja jeprikazana na sledećoj slici.



Slika 2.2 Programerski pogled na kompjuter

Šta je kompjuterski program?

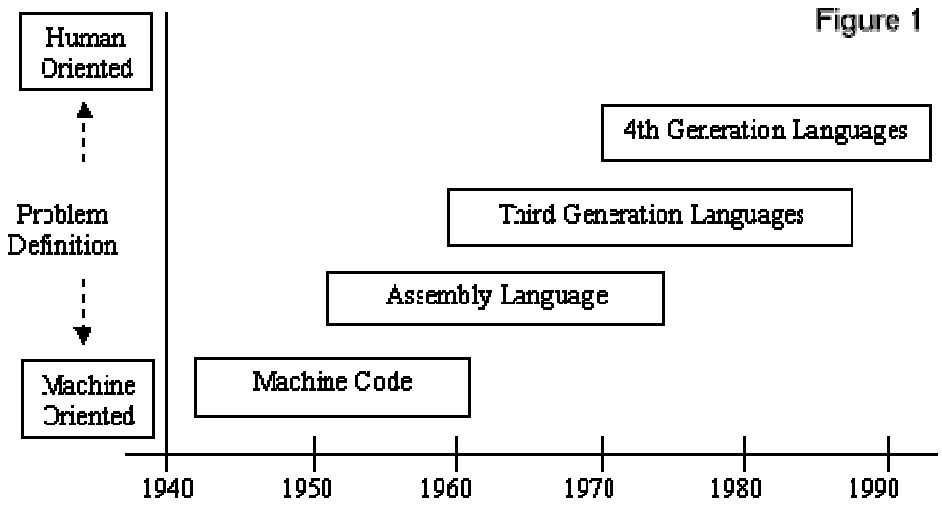
Ako pogledate definiciju koju smo dali za kompjuter, ali obrnutim redosledom počev od "Kompjuter može da izvršava nizove instrukcija" - dobićete definiciju kompjuterskog programa - to je niz (sekvenca) instrukcija koju CPU interpretira kao aritmetičke ili logičke operacije nad podacima iz memorije. Kompjuterski program je smešten u memoriji i takođe se sastoji od podataka i instrukcija. Ulazno-izlazni uređaji (I/O) omogućavaju mašini da prenosi podatke između nje i spoljnjeg sveta.

Instrukcije i programi zahtevaju neki redosled kojim će ih kompjuter izvršavati, a to je zapravo element za projektovanje. Projektovanjem programa određujemo koje su programske instrukcije i kakva struktura podataka je potrebna da bi mašina-kompjuter obavila neki željeni zadatak. Najznačajnije je to što projektant programa određuje niz (sekvencu), tj. redosled kojim će se instrukcije izvršavati u cilju uspešnog odvijanja i završetka programa.

Sada kada imamo jednu jasniju predstavu šta je kompjuter i šta je kompjuterski program, hajde da pogledamo šta su generacije jezika?

Generacije kompjuterskih jezika

Pojam generacije proistekao je iz razvoja koji je prikazan na sledećoj slici koja prikazuje razvoj kompjuterskih jezika od 1940-tih godina do danas.



Slika 2.3 Razvoj programskih jezika

Cilj ove slike nije da bude potpuno precizna, već da da ideju o vremenskim periodima i različitim generacijama koje se se pojavljivale i međupovezanost jezika raznih generacija sa složenosću problema koji se rešavaju primenom kompjutera.

Vertikalna osa predstavlja definiciju problema, tj. pokazuje kako mi projektujemo (pišemo) kompjuterske programe. Na dnu ove skale se nalazi mašinski orjentisan način definisanja programa. To znači da se kompjuterski program izražava u jeziku koji je bliži mašini nego programeru-čoveku, kao što smo vi i ja. Na vrhu ove ose je ljudski orjentisan način izražavanja programa. To znači da je definicija problema - kompjuterski program –izražen na način bliži jeziku projektanta-čoveka nego jeziku mašine. I to je naravno poželjniji i bolji način za programiranje.

Mašina razume jezik izražen pomoću 1 i 0 (binarni kod), na primer:

10101111000101000100010111110100101010101

dok čovek više voli da isti iskaz prikaže ovako:

Saberi total iz ove nedelje sa totalom iz prethodne nedelje

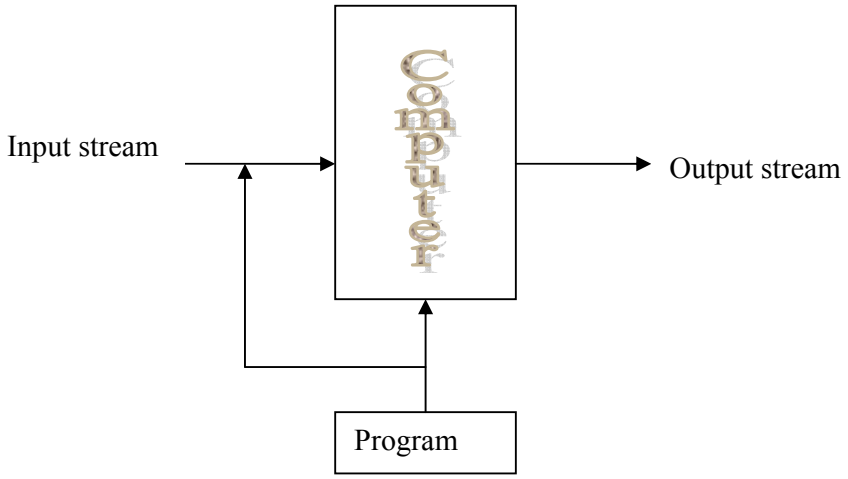
Horizontalna osa na slici prikazuje približno vreme kada se određena generacija jezika pojavila i ušla u upotrebu.

Imajte na umu da slika prikazuje samo aproksimativno, i da to što su jezici treće generacije završeni krajem 80-tih godina prošlog veka, to ne znači da oni nisu i danas u upotrebi. U praksi ne postoje jasne granice upotrebe jezika, pa nije redak slučaj da se kod nekih obimnijih projekta pojavi više generacija jezika u istom projektu. Tako se mogu naći projekti koji u sebi sadrže asemblerske programe kombinovane sa programima treće i četvrte generacije.

Poslednje generacije kompjuterskih jezika omogućavaju projektantima i programerima da pišu programe na jeziku vrlo sličnom prirodnom jeziku.

Model kompjuterskog programa

Pošto je kompjuterski program niz instrukcija koje "obrađuju" podatke, možemo to prikazati kroz sledeći prosti model:



Slika 2.4 Program transformiše ulaz i generiše izlaz

Model pokazuje da program procesira podatke koji dolaze sa ulaza – izvora podataka, a rezultati obrade idu na izlaz. Iako to nije obavezno slučaj, kod većih programa i programskih sistema ulaz i izlaz podataka su obično van samog programa. Ulaz je obično neki ulazni ili memorijski uređaj kao što je tastatura, disk, skener, traka, a izlaz, obično neki izlazni uređaj ili memorija kao sto je ekran, štampač, disk, traka, crtač itd.

Sam program je zapravo niz instrukcija zapisanih u memoriji kompjutera kojim se precizno određuje na koji način će se transformisati ulazni podaci da bi se dobio željeni izlaz. No da bi takav pogodan program dospeo u memoriju računara, potrebno je da bude najpre sačinjen u nekom programskom jeziku koji je razumljiv za čoveka-programera a potom preveden na jezik razumljiv procesoru kompjutera. Taj proces stvaranja novih programa i njihovo prevođenje ra kompjuterski jezik jeste glavni predmet našeg razmatranja.

Podaci i informacije

Poslednja tema u ovom delu je kratko objašnjenje termina podatak i informacija. Ponekad se ovi termini koriste kao sinonimi, iako ne bi trebalo da bude tako. Podatak predstavlja neku vrednost (niz simbola, znakova) van konteksta i bez značenja. Informacija je podatak sa značenjem. Na primer, recimo da imamo sledeću listu brojeva:

24.56, 21.94, 27.23, 30.61

Šta oni znače? Značenje ovih brojeva zavisi od konteksta u kojem se koriste. Oni mogu predstavljati prosečne temperature tokom nekog perioda, ali isto tako mogu predstavljati i cenu nekog proizvoda. Takvi brojevi mogu biti smešteni na disk i možemo napisati kompjuterski program koji čita te brojeve sa diska - izvora podataka – i koji ih tretira kao temperature. Neki drugi program ih moze takođe čitati sa diska i tretirati ih kao cene. Tako, zapravo sam program određuje kontekst podataka.

Siže predavanja

U ovom predavanju upoznali smo se sa osnovnim konceptima koji će nam pomoći da bolje razumemo naredna predavanja. Sada ste u stanju da definišete šta se podrazumeva pod kompjuterom i kompjuterskim programom, imate jasniju ideju kako su se razvijali programski jezici, kao i jasniju predstavu o sličnostima i razlikama pojmova podatak i informacija.

O podacima i informacijama biće više reči u narednom predavanju.

Pitanja

1. Šta je to *instrukcija* u generalnom smislu?
2. Šta je to *mašina* u generalnom smislu?
3. Dajte jedan primer mašine i način na koji dajete instrukcije toj mašini. To ne mora biti kompjuter (recimo automat za koka-kolu)
4. Navedite četiri osnovne karakteristike kompjutera. Možete li da navedete još neke?
5. Šta se podrazumeva pod izrazom - *niz* (sekvenca)?
6. Šta znači *izvršavanje* instrukcije?
7. Šta je *kompjuterski program*?
8. *Mašinski kod* koristi prirodan jezik računara – šta to zapravo znači?
9. Šta je *ulaz podataka*?
10. Šta je *izlaz podataka*?

3. Podaci i informacije

Obrada podataka

Pod obradom podataka podrazumeva se proces kojim se podaci transformišu tako da se od nekog početnog skupa (ulaznih) podataka dobije novi (izlazni) skup podataka. Tako se recimo mogu sabrati visine (ili težine) niza osoba, pa kada se takav zbir podeli sa brojem osoba čije su visine (težine) sabrane dobijamo informaciju o prosečnoj visini (težini) tog skupa osoba. Može se reći da smo obradom podataka o visinama (težinama) kao rezultat dobili jednu novu informaciju (podatak). Zapravo svaka obrada podataka i ima za cilj dobijanje nove informacije koja je "sakrivena" u sirovim podacima.

Proces obrade podataka možemo slikovito prikazati sledećim dijagramom:

Slika 3.1 Proces obrade podataka

Podrazumeva se, naravno, da se obrada podataka vrši u kompjuterima. Podaci se mogu u kompjuterima obrađivati na više načina u zavisnosti od uslova pod kojima se obrada vrši.

Ako se obrada vrši na taj način što se podaci najpre prikupljaju, prilagođavaju za kompjutersku obradu unose u kompjuter tek kada su svi podaci prikupljeni onda takvu obradu podataka nazivamo **batch** (beč) obradom. To je slučaj sa raznim statističkim obradama, kao i obradama recimo plata u nekom preduzeću, obradama računa za električnu energiju, TV pretplatu i slično. Trajanje ovakve obrade može biti i više sati rada kompjutera.

Ako se obrada vrši odmah nakon unosa podataka onda kažemo da se radi o **on-line** obradi. To je slučaj recimo sa bankarskim transakcijama kada podižete (ili ulažete) gotovinu sa vašeg računa, plaćate kreditnom karticom ili rezervišete mesto u avionu (ili pozorištu). Kod takvi obrada se ne može čekati na prispeće svih podataka, već se obrada vrši redom kako oni stižu. Trajanje obrade može biti do nekoliko sekundi (ponekad i minuta).

Postoje slučajevi kada se podaci moraju obraditi za veoma kratko vreme (ispod 1 sekunde ili čak za hiljaditi deo sekunde). Za takve obrade kašemo da se odvijaju u realnom vremenu (**real-time**). Primeri takve obrade su razne vrste upravljanja robotima, automatskim pilotima, hirurškim zahvatima i sl. Te su obrade često povezane i sa visokim rizicima po bezbednost ljudi ili gubitak njihove imovine.

A kakvi se sve podaci mogu obrađivati? Kakve sve oblike podataka znamo?

Vrste podataka i način kodiranja

Podaci i informacije imaju dve važne karakteristike: formu i sadržinu. Forma je oblik u kojem se podaci pojavljuju a sadržina je njihovo značenje. Uobičajene i najrasprostranjenije forme podataka su slova, brojevi, zvuk, slika, filmski zapis. To su forme prepoznatljive čoveku. Čovek može da prepozna i neke druge forme podataka koje prima i drugim čulima kao što su dodir, miris, ukus, temperatura. Ali u prirodi postoje i druge fizičke karakteristike materije i prostora koje su moguće forme podataka kao što su električni i magnetni signali, bio-energetski signali da pomenemo samo neke od njih.

Naša dalja pažnja biće usmerena samo na standardne forme podataka koje su najčešće u upotrebi, mada se taj skup stalno širi novim formama.

Značenje podataka predstavlja drugu važnu njihovu karakteristiku. Jedan te isti podatak može imati različita značenja, u zavisnosti od interpretacije. Kakvu ćemo interpretaciju pridružiti nekom podatku? Najčešće je interpretacija povezana sa kontekstom u kojem se neki podatak nalazi. Mi se ovde nećemo baviti značenjima podataka već samo njihovom formom i mogućnostima transformacije podataka iz jedne forme u drugu, kao i kombinovanjem podataka u cilju dobijanja novih informacija onako kako to prikazuje gornji dijagram.

Numerički podaci (brojevi)

Brojevima se izražavaju kvantifikatorska svojsta objekata i procesa. Brojevima prikazujemo cene, količine, fizičke veličine (visine, težine, temperature, itd.) kao i razne druge merljive podatke. Brojevima se mogu modelirati i veoma složene prirodne i društvene pojave i procesi. Statistički pokazatelji su dobar primer za upotrebu brojeva pri analizi prirodnih i društvenih fenomena. Aritmetika je oblast matematike koja se bavi osnovnim karakteristikama brojeva. Tokom razvoja matematike uvedeno je više vrsta brojeva, od prirodnih brojeva (0,1,2,3...), celih brojeva (...-2,-1,0,1,2,...), racionalnih brojeva (razlomaka, ½, 2/3 , ...), pa sve do kompleksnih brojeva (2 + 3i). Za nas su najinteresantniji celi i decimalni brojevi kao brojevi koji se najčešće koriste u praksi.

Dekadni brojni sistem

Brojevi se mogu zapisivati na više načina. U svakodnevnoj upotrebi je tzv. dekadni brojni sistem gde se za zapisivanje brojeva koristi dest cifara (0,1,2,3,4,5,6,7,8,9), pa otuda i ime dekadni (deka na grčkom znači deset).

U dekadnom sistemu se recimo zapis 327 tumači kao broj: tri stotine dvadest sedam. To se može i ovako prikazati:

$327 = 3x100 + 2x10 + 7$ (tri puta sto plus 2 puta deset plus 7), ili ovako:
$327 = 3x10^2 + 2x10^1 + 7x10^0$ (tri puta deset na kvadrat plus dva puta deset na jedan plus 7 puta deset na nula), pa zato kažemo da je ovo pozicioni brojni sistem jer vrednost cifre zavisi od mesta na kojem se cifra nalazi u broju.

Decimalni brojevi pored cifara koriste i decimalni zarez kojim se razdava broj celi od broja koji označava deo celog. Tako recimo broj 1835,346 označava broj koji ima 1835 celih i dodatak od 346 hiljaditih delova celog (346/1000). U komjuterskim aplikacijama decimalni zarez se često zamenjuje decimalnom tačkom tako da se gornji broj može zapisati i kao 1835.346.

Binarni brojni sistem

Brojevi se na sličan način mogu zapisivati sa različitim brojem cifara. U kompjuterskim naukama u upotrebi su pored dekadnog još i binarni, oktalni i heksadecimalni brojni sistemi.

Binarni brojni sistem koristi samo dve cifre: 0 i 1. Ovaj sistem je pogodan za komjutere zbog tehnologije njihove izrade i tzv. Bulove algebre na kojoj se zasniva kompjuterska tehnika. Slično kao i kod dekadnog sistema brojevi se zapisuju ciframa 0 i 1, a vrednsot cifre zavisi od njene pozicije u broju. Tako na primer, binarni broj 10011 zapravo ima sledeću vrednsot:

$10011 = 1x2^4 + 0x2^3 + 0x2^2 + 1x2^1 + 1x2^0$

Ili u dekadnom obliku 1x16 + 0x8 + 0x4 + 1x2 + 1x1 = 19 (devetneast).

Gornji primer pokazuje kako se vrši konverzija iz binarnog u dekadni brojni sistem. Potrebno je, zanači, samo svakoj cifri binarnog broja pridužiti njenu vrednost u zavisnosti od pozicije i sabrati.

Obrnute konverzija, iz dekadnog u binarni oblik se može izvršiti sledećim postupkom:

1. Broj koji se konvertuje podeliti sa 2.
2. Rezultat deljenja zapisati ispod broja koji se konvertuje.
3. Ostatak pri deljenju (koji može biti samo 0 ili 1) zapisati sa strane.
4. Ponoviti korake 1-3 sve dog rezultat deljenja ne bude jednak 0.
5. Ostatke pri deljenju (nule i jedinice) poredjati jedne za drugim u redosledu od poslednjeg do prvog izračunatog ostatka.

Gornji postupa ilustrujmo na primeru konverzije broja 213 i binarni oblik.

213 podeljeno sa 2 daje rezultat 106 i ostatak 1
106 podeljeno sa 2 daje rezultat 53 i ostatak 0
53 podeljeno sa 2 daje rezultat 26 i ostatak 1
26 podeljeno sa 2 daje rezultat 13 i ostatak 0
13 podeljeno sa 2 daje rezultat 6 i ostatak 1
6 podeljeno sa 2 daje rezultat 3 i ostatak 0
3 podeljeno sa 2 daje rezultat 1 i ostatak 1
1 podeljeno sa 2 daje rezultat 0 i ostatak 1

Dakle, 213 dekadno je isto što i 11010101 binarno.

Heksadecimalni brojni sistem

Heksadecimalni brojni sistem ima 16 cifara (heksa na grčkom znači šestnaest).

Pored deset cifara koje su preuzete iz dekadnog sistema heksdecimalni sistem koristi i slova A,B,C,D,E i F. To je takođe pozicionin brojnin sistem sa ciframa: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F, gde cifre 0-9 imaju uobičajene vrednosti a A,B,C,D,E i F vrednosti 10,11,12,13,14, i 15 redom.

Tako je, na primer, heksadecimalni broj 4A2 zapravo:

$4A2 = 4x16^2 + 10x16^1 + 2x16^0$ (setiete se da A ima vrednost 10, otuda $10x16^1$)

Dakle 4A2 heksadecimalno ima vrednost 1186 (1024 + 160 + 2).

Konverzija iz dekadnog u heksadecimalni oblik (i obrnuto) vrši se analogno kao kod binarnih brojeva.

Međutim, konverzija iz binarnog u heksadecimalni oblik (i obrnuto) je mnogo jednostavnija.

Recimo da želimo da heksadecimalni broj 4A2 (iz gornjrg primera) prevedemo u binarni oblik. Sve što treba da uradimo je da svaku heksadecimalnu cifu prevedemo u binarni oblik, kako sledi:

4A2 hesadecimalno je 100101010 binarno jer je 4 u binarnom obliku 100, A je 1010, a 2 je 10, pa kad ova tri binarna niza spojimo dobijamo 100101010.

Obrnuto, ako želimo da neki binarni broj prevedemo u hesadecimalni oblik, sve što treba da uradimo je da binarni broj razdelimo u grupe od po četiri cifre (počev od krajnje desne cifre) i svaku takvu grupu prevedemo u heksadecimalnu cifru.

Na primer binarni broj 100010101010111010010011 1 koji na prvi pogled izgleda zastrašujuće dugačak, lao se prevodi u heksadecimalni zapis.

Najpre broj razdelimo u grupe od po četiri cifre. Dobićemo sledeći niz:

1 0001 0101 0101 1101 0010 0111

Sada svaku grupu prevedimo u heksadecimalnu cifu:

Iamaćemo:

1 prevodimo u 1
0001 takođe u 1
0101 prevodimo u 5
0101 takođe u 5
1101 prevodimo u D (13)
0010 prevodimo u 2
0111 prevodimo u 7

Pa je tako

10001010101011101001001111 binarno jednako 1155D27 heksadecimalno. Lako, zar ne?

Operacije sa brojevima

Brojeve možemo da sabiramo, oduzimamo, delimo i možimo. To su tzv. osnovne aritmetičke operacije sa brojevima. Već se u početnom stadijumu učenja srećemo sa ovim operacijama i naučimo kako da ih izvršavamo nad dekadno zapisanim brojevima. Ovde ćemo samo napomenuti da se gornje aritmetičke operacije izvršavaju analognim postupcima i za binarne i heksadecimalno zapisane brojeve.

I najsloženiji matematički proračuni, najsloženije jednačine se na kraju svode na ove četiri operacije. Sve što se može sračunati sračunava se ovim operacijama. Naravno za složene proračune imamo postupke (algoritme) u kojima se ove osnovne operacije kombinuju na različite načine da se dođe do željenih rezultata. Ali o algoritmima ćemo kasnije. O brojevima i operacijama sa brojevima za sada ovoliko.

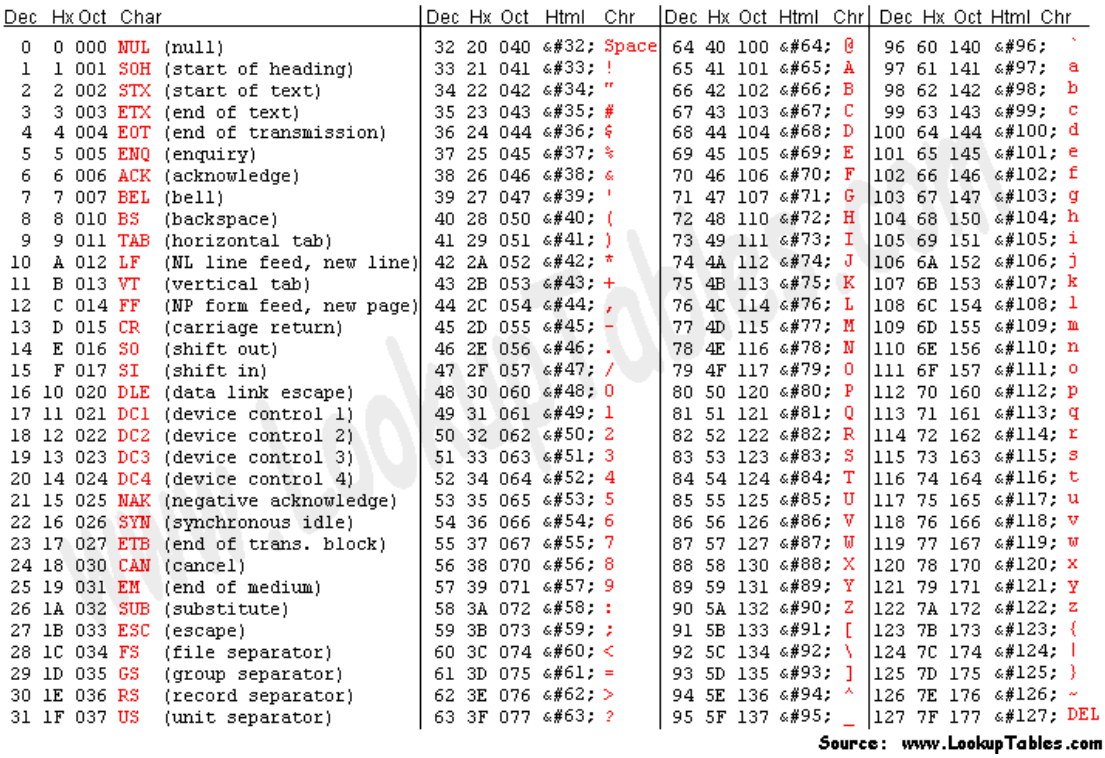
Tekstualni podaci

Naša civilizacija je dobila ubrzan razvoj od trenutka kada je čovek počeo da zapisuje svoje misli u obliku teksta. Posebno ubrzanje je dobijeno Gutenbergovim pronalsakom štamparije kada je postalo moguće tekst umnožavati u više primeraka na ekonomičan način. Pojavom kompjutera, a posebno personalnih kompjutera i njihovim povezivanjem u globalnu mrežu (Internet) dostupnost tekstualnih informacija u obliku knjiga, časopisa i drugih tekstova postala je tako velika da su nam danas potrebni posebni pretraživači (Google na primer) bez čije pomoći bi bili izgubljeni u tom okeanu informacija, među kojima tekstualne informacije imaju značajan udeo.

Pa kako su računari u stanju da pamte tekst. Za brojeve smo već videli, koriste cifre 0 i 1. A tekst? Takođe. Kako je to moguće?

Na samom početku korišćenja računara (50-tih godina prošlog veka) napravljen je jedan šifarnik kojim su slova kodirana brojevima. Pa kada slova prevedemo u brojeve onda je lako, brojeve možemo zapisivati u kompjuterima.

Napravljeni šifarnik slova poznat je kao ASCII (American Standard Code for Information Interchange) tabela i još uvek je u upotrebi (vidi sledeću tabelu).



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Slika 3.2 ASCII tabela

Kao što se može videti sa gornje slike ASCII tabela sadrži slova intrenacionalnoh alfabeta (26 slova od A do Z, mala i velika), cifre znake interpunkcije I razne druge simbole, sa ili bez grafičkog prikaza.

Tako recimo veliko slovo A ima šifru 65 u dekadnom brojnom sistemu, ili 41 u heksadecimalnom.

ASCII kod je prvobitno bio 7-bitni kod (jer ima 128 znakova, a oni se mogu kodirati sa 7 binarnih cifara). Pošto je sadržao samo internacionalnu latiničnu azbuku, ovaj kod je bilo nemoguće koristiti za druge jezike i pisma (osim engleskog).

U poslednje vreme, a posebno za potrebe interneta koristi se novi tzv. UNICODE. Ovaj kod se pojavljuje u više varijanti, a UTF-16 je 16-bitna varijanta koja pokriva 30-tak sistema za pisanje teksta na raynim jezicima.

Operacije sa tekstom

Slično kao što kod brojeva postoje osnovne operacije, tako i obrada tesktualnih podataka ima jedan skup jednostavnih operacija pomoću kojih se mogu obavljati i najsloženije operacije nad tesktovima. To su operacije nad pojedinačnim slovima u tekstu kao što su: brisanje slova, dodavanje slova, promena slova. Pomoću tih operacija moguće je izvršiti razne operacije nad celim tekstovima kao što su brisanje dela teksta, dodavanje teksta, zamena teksta drugim tekstom. Tekst procesori (MS Word, na primer) obavljaju slične operacije.

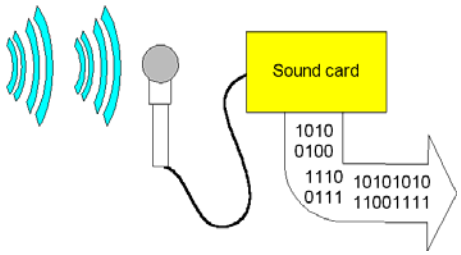
Razvojem grafike (kako na ekranima tako kod štampača) postalo je moguće prikazivati slova u raznim oblicima (fontovima), raznim veličinama i drigim tipografskim karakteristikama (boja, otisak, italik i sl.).

Danas se u kompjuterima koriste i tzv. hipertekstovi u kojima se tekst ne mora čitati liearno već se može "skakati" sa teksta na tekst u proizvoljnom redosledu sledeći takoznake hiperlinkove. A tekstualni zapisi mogu da sadrže slike, video i zvučne delove, tako da se dobija informacija obogaćena raznim vrstama i formama podataka.

Zvučni podaci

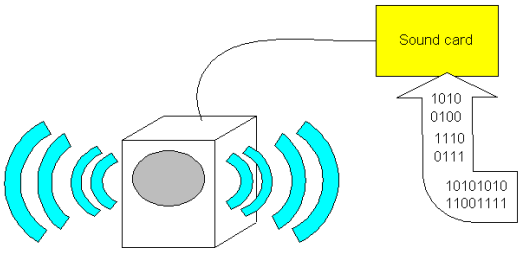
Zvuk, kao što je poznato nastaje vibracijom vazduha, pa kada se ta vibracija prenese do naše bubne opne, ona vibrira u istom ritmu i proizvodi signal koji naš mozak interpretira kao zvuk.

Kako zvuk registruju, pamte i reprodukuju kompjuteri? Znamo da su mikrofoni i zvučnici standardni deo periferije načih PC-ja. No kako se zvuk pamti u komjuteru? Opet pomoću brojeva. Posto je zvuk vibracija vazduha, ta se vibracija u mikrofonu (slično kao u našoj bubnoj opni) pretvara u električni signal kako je to ilustrovano na sledećoj slici.



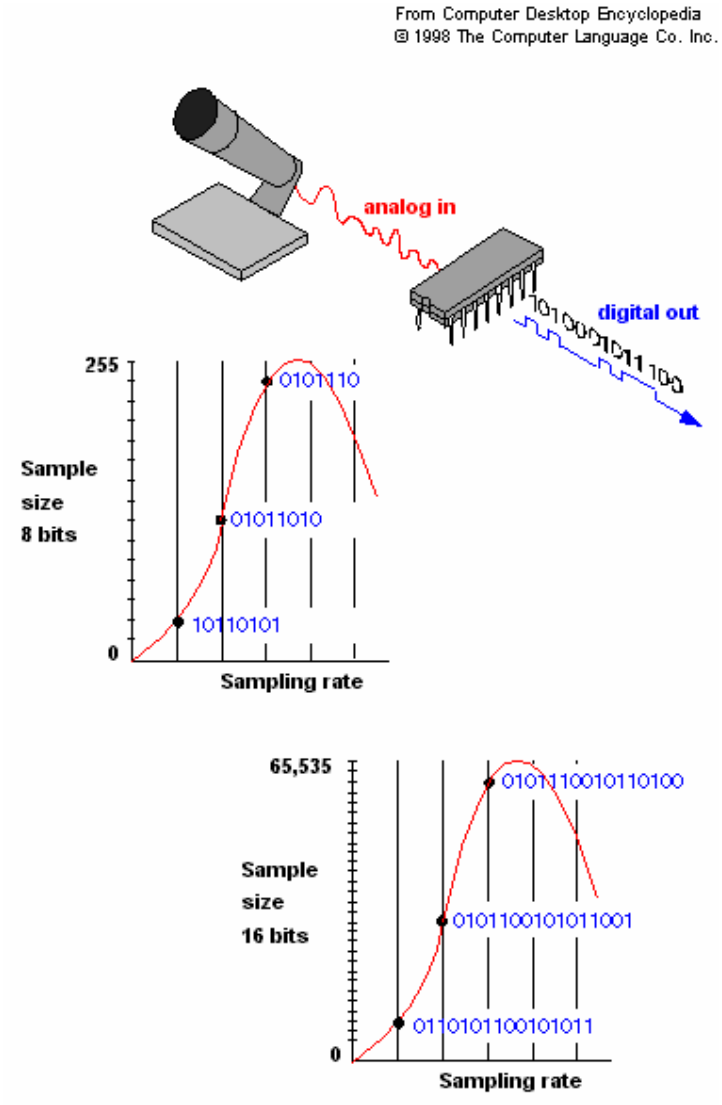
Slika 3.3 Registrovanje zvuka

S druge starne, reprodukcija zvučnog signala se odvija tako što se brojevi kojima je registrovan zvuk pretvaraju ponovo u električna signale kako to prikazuje sledeća slika.



Slika 3.4 Reprodukcija zvuka

Konverzija iz zvuka u brojeve (i obrnuto) vrši se tako što se analogni električni signali sempliraju (digitalizuju) na način ilustrovan sledećom slikom.



3.5 Digitalizacija zvuka

Zvučni podaci (govor, muzika) se mogu takođe obrađivati. I kod njih postoji jedan skup osnovnih operacija: brisnaje dela zvučnog zapisa, dodavanje zvučnog zapisa, zamena zvučnog zapisa. Pomoću takvih operacija moguće je izvršiti veoma složene transformacije zvučnih zapisa i postići razne zvučne efekte.

U praksi se koristi više načina digitalizacije i zapisivanja zvučnih podataka. Kompjuterski fajlovi u kojima se nalazi zvuk mogu imati razne ekstenzije kao što su wav, mp3, aud, midi, da pomenemo samo neke od najpoznatijih.

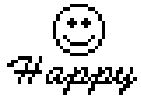
Grafika (slika, video)

Jedna slika vredi hiljadu reči kaže kineska poslovica. Vuzuelne informacije su veoma pogodne za čoveka. Zato kompjuterska tehnologija posebnu pažnju poklanja ovoj vrsti podataka. Svedoci smo stalnog razvoja u ovoj oblasti, gde postoji tendencija da

se preko video tehnologija objedine svi komunikacioni kanali (TV, komputeri, novine) u jedisntven sistem multimedijalne prezentacije podataka i informacija.

Kako se slika (nepokretna i poretna) registruje, memoriše i reprodukuje u kompjuterima. Brojevima, ponovo.

Sledeća slika ilustruje jedan proces digitalizacije crno-bele fotografije.

Original	Digitalizovano	Bit mapa
		00000000000000000011110000000000000000 00000000000000001100001100000000000000 00000000000000010000000010000000000000 00000000000000100000000001000000000000 00000000000001000100010010000000000000 00000000000010001110111001000000000000 00000000000010000100010001000000000000 00000000000010000000000001000000000000 00000000000010000000000001000000000000 00000000000010010000010010000000000000 00000000000010010000100100000000000000 00000000000010001111000100000000000000 00000000000000100000000100000000000000 00000000000000011000011000000000000000 00000000000000000111100000000000000000 00011110010000000000000000000000000000 01100010010000000000000000000000000000 11000100100000000000000000000000000000 00000100100001110001011000101100100100 00111111110010010001101000110101100100 00001001000100100111001011100101001000 00010010000101101010010101001011011010 00010010000110110111111011111101101100 00000000000000000100000010000000011000 00000000000000000110000011000000000110000 00000000000000000100000010000000000100000

Slika 3.6 Digitalizacija slike

Proces se može opisati ovako: Originalna slika se pretvara u “mozaičku” verziju, gde svaki kamen u mozaiku prestavlja crnu ili belu površinu. Onda se mozaik konveruje u nizove 0 i 1 tako da 0 odgovara belom a 1 crnom kamičku. Na kraju se te serije 0 i 1 zapisuju u memoriji kompjutera.

Reprodukcija je obrnut proces: Svaka 0 i 1 iz memorije se na ekranu monitora prikazuje kao svetla (bela) ili tamna (crna) tačka. Svaki monitor se zapravo i sastoji od određenog broja vertikalnih i horizontalnih linija koje se sastoje od određenog broja tačaka (piksela). Ovde se matematička definicija linije razlikuje od praktičene definicije. Dok se matematička linija sastoji od beskonačnog niza tačaka, linija na monitoru ima konačan broj takvih tačaka (broj verikalnih i horizontalnih tačaka na ekranu nazivamo rezolucijom ekrana).

A kako pamtimo sliku koja nije crno bela? Tu umesto što svaku tačku pamtimo pomoču 0 ili 1, pamtićemo boju tačke. Iz teorije boja pozanto je da se svaka boja može dobiti kombinacijom tri boje (crvene, zelene i plave). Na engleskom, Red, Green, Blue (RGB). Dakle ako za svaku osnovnu boju uzmemo jedan bajt memorije onda možemo svaki piksel “obojiti” sa jednom od 256x256x256 boja.

Slično, ako umesto kolora želimo sliku sa tonovima sivog možemo svakom pikselu pridužiti jedan bajt kojim se opisuje jedan od 256 nivoa sivog.

A šta ako imamo pokretnu sliku (video zapis). Pa nije problem, potrebno je samo pamtiti više uzastopnih slika u jednosj sekundi. Kod klasičnih filmova sa filmskom vrpcom uobičajeno je da se prave 24 slike u sekundi, kod kompjutera taj broj je obično 36 pa je tako kompjuterska slika stabilnija.

Naravno zapisivanje slike, a posebno videa zahteva veliku količinu kompjuterske memorije, mnogo veće nego za zvuk, a naročito u odnosu na tekst i brojeve.

I sliku (video) je moguće obrađivati. Osnovne operacije su ovde promena karaketristika piksela (boje, osvetljaja, kontrasta).

Postoji takođe, mnogo načina digitalizacije slike (i vide-a). Najpoznatiji formati zapisa su bmp, jpg, png, pic, tiff itd.

Pitanja

1. Kako se podaci razlikuju od informacija?
2. Nevedite osnovne operacije koje su prisutne u informacionim sistemima?
3. Od kojih se komponenti sastoji informacioni sistem? Šta sve IS uključuje?
4. Navedite neke od operacija koje se mogu vršiti nad podacima?
5. Navedite bar tri vrste podataka.
6. Navedite najčešće korišćene osnovne tipove podataka u programskim jezicima.
7. Definišite podatke tipa bool(ean).
8. Koliko je heksadecimalno 7F u dekadnom sistemu?
9. Koliko bajtova je potrebno za predstavljanje jednog slova u ASCII kodu? A u Unicode-u.
10. Zašto služi kompresija zvučnih signala?
11. Navedite bar dva formata zvučnog zapisa.
12. Navedite bar tri formata za memorisanje slike.

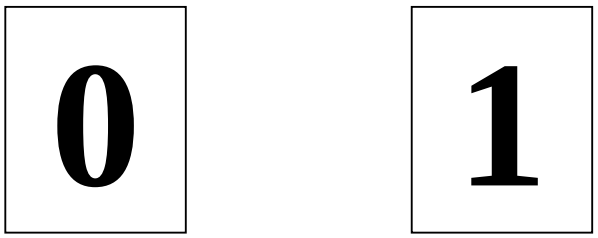
4. Memorisanje, prenos i prezentacija podataka

Memorisanje podataka u kompjuterima

U predhodnom poglavlju upoznali smo se sa različitim vrstama podataka (numeričkim, tekstualnim, zvučnim i video) i načiniom njihove digitalizacije. Sada čemo se baviti kompjuterskom memorijom koja može da sadrži takve podatke.

Ono što očekujemo od kompjuterske memorije jeste da u nju možemo smestiti podatke, da im možemo pristupati, da ih možemo menjati i transferovati iz jedne memorijske lokacije u drugu, ili čak iz jednog kompjutera u drugi udaljeni kompjuter.

Ali počnimo redom. Videli smo da nam je za memorisanje bilo koje vrste podataka potrebno zapravo da memorišemo samo nizove 0 i 1 od kojih se sastoje sve vrste podataka. Zato ćemo za momenat zamisliti da imamo neki fizički objekat u koji možemo da upišemo nulu ili jedinicu. Zamislimo ga na način koji prikazuje sledeća slika:



Slika 4.1 Memorija za pamćenje 0 i 1

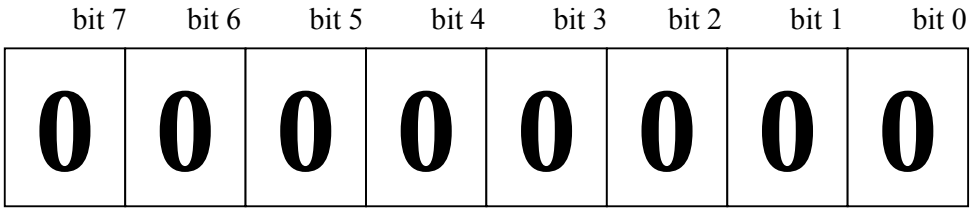
Ovakva elementana memorijska ćelija naziva se jednim bitom memorije. Naziv dolazi otuda što se u njoj može pamtiti jedna binarna cifra (**bi**nary digi**t**).

Kako se tehnološki realizuje takva memorija za nas u ovom trenutku nije važno. Napomenućermo samo da ima više načina za izradu takvih memorija – poluprovodnici, magnetni materijali, optički materijali. Od poluprovodničke memorije se najčešće prave RAM i ROM memorije, od magnetnih hard diskovi a od optičkih CD i DVD diskovi.

Zamislimo sada da ovako malih memorijskih delova, kao onih na slici 4.1 imamo u izobilju – stotine miliona, kao što je čest slučaj. Kako se snaći u tom ogromnom broju, kako gnati gde nam se nalaze podaci, gde su slike, gde su tekstovi, a gde programi?

Da bi smo to postigli, porebno je da uvedemo neki sistem u organizacuji memorije. Prvi korak je da ove male memorijske elemente povežemo u veće skupine.

Ako 8 bitova povežemo u jednu celinu dobićemo tzv. bajt kao pokazuje sledeća slika.



Slika 4.2 Bajt

U jednom bajtu možemo da smestmo jedu od 256 mogućih kombinacija 0 i 1. Recimo možemo da smestmo neko slovo iz ASCII tabele, ili jednu od RGB boja nekog piksela iz slike. Kao što se vidi sa gornje slike s obzirom da svaki bit u bajtu ima svoje mesto (adresu od 0 do 7) svakom od tih bitova možemo individualno da pristupamo i da ga menjamo.

A sada pokušajmo da više bajtova povežemo u jednu celinu kao na sledećoj slici.

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
...								
N	0	0	0	0	0	0	0	0

Slika 4.3 Niz bajtova

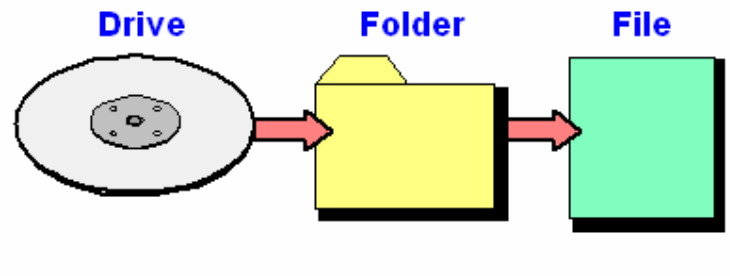
Brojevi 0,1,2,3,4,...,N predstavljaju adrese bajtova tako da sada možemo pristupati i svakom bajtu posebno i menjati njegov sadržaj. Dakle memoriju računara možemo

posmatrati kao jedan linearna niz bajtova, gde se svakom bajtu može pristupati preko njegove adrese, a unutar bajta i svakom bitu. Obično memorije sadrže veliki broj bajtova koji se meri kilo, mega, giga ili tera bajtima, a prema sledećoj tabeli vrednosti za ove mere.

Name	Abbr.	Size
Kilo	**K**	**2^10 = 1,024**
Mega	**M**	**2^20 = 1,048,576**
Giga	**G**	**2^30 = 1,073,741,824**
Tera	**T**	**2^40 = 1,099,511,627,776**
Peta	**P**	**2^50 = 1,125,899,906,842,624**
Exa	**E**	**2^60 = 1,152,921,504,606,846,976**
Zetta	**Z**	**2^70 = 1,180,591,620,717,411,303,424**
Yotta	**Y**	**2^80 = 1,208,925,819,614,629,174,706,176**

Slika 4.4 Veličina memorije (u bajtima)

Gore prikazani model memorije najpogodniji je za RAM i ROM memorije. Kod magnetnih i otičkih diskova uobičajna je jedna druga organizacija, kod koje imamo podelu memorijskog prostora na foldere (direktorijume). Folderi se uređuju hijerarhijski kako prikazuje sledeća slika.



Slika 4.5 Organizacija podataka na diskovima

Osnovni kontejner podataka na diskovima je fajl (file, ili u prevodu datoteka). Fajl možemo takođe zamisliti kao niz bajtova. S obzirom da su fajlovi, najčešće, veoma dugački nizovi bajtova, pojedinačnim bajtovima u fajlu se ne pristupa preko adrese svakog bajta iz fajla, već se fajl čita deo po deo, pa se onda u trenutno učitanom delu pronalaze podaci koje treba koristiti (ili menjati).

Da rezimiramo: Kod kompjutera razlikujemo dve osnovne vrste memorije RAM (i ROM) u kojima se podacima pristupa preko adresa bajtova u kojima se podaci nalaze, i disk memorije (magnetni i optički) u kojima su podaci smešteni u fajlove pa se pristup podacima vrži tako što se fajlovi čitaju deo po deo, podaci se ovim čitanje prebacuju u RAM a onda im se pristupa.

Prenos (transfer) podataka

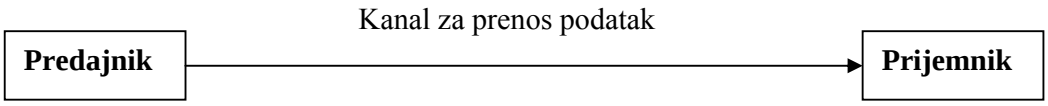
Malo pre smo videli kako se podaci iz fajlova moraju najpre preneti u RAM da bi se sa nad njima vršila obrada. U kompjuteru se odvija veoma živ i obiman prenso podataka od jedog do drugog njegovog dela, od memorije do procesora i nazad, od

memorije do perifernih uređaja i nazad. To su tzv. lokalni transferi koji se odvijaju velikom brzinom na malim rastojanjima (unutar kompjutera i njegovih čipova).

Može se slobodno reći da se tokom obrade podata najveći deo vremena troši upravo na tom transferu podataka. I sami ste svedoci kako po nekad nestrpljivo čekate da podaci (i programi) sa vašeg diska budu učitani u memoriju računara. Da ne govorimo ovremenu dok čekate da vam se sa interneta daunloduje (download) neki sadržaj. Pošto prenos podataka ima tako značajno mesto, onda je potrebno da se brzina prenosa učini što većom.

Brzina prenosa zavisi od načina ostvarivanja komunikacije između pojedinih memorijskih delova. Najbrže su, svakako, optičke komunikacije kod kojih je brzina prenosa inforamcije približna brzini svetlosti, koja je opet najveća moguća poznata brzina u prirodi. Nakon optičkih slede elektromagnetni i električni signali koji imaju slične brzine, ali manji domet bez gubitaka.

Problem prenosa podataka možemo da zamislimo kao problem prenosa niza bitova (ili bajtova) kako prikayuje sledeća slika.

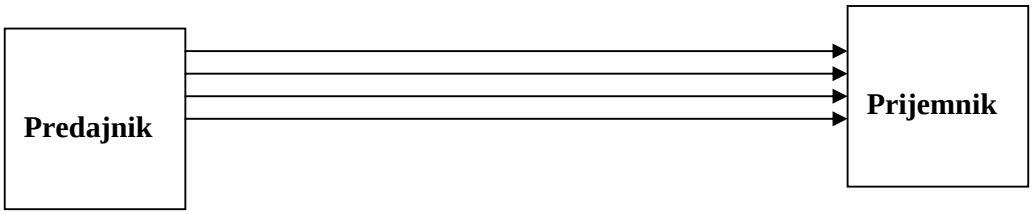


Slika 4.6 Serijski prenos podataka

Brzina kojom se podaci prenose od predajnika do prijemnika naziva se bitrate i meri se brojem bitova u sekundi. Na primer 64 kbit/s je brzina kojom se 64000 bitova prenosi u jednoj sekundi.

Ako se bitovi prenose jedan za drugim, tada govorimo o serijskom prenosu podataka.

No moguće je prenosti i više bitova istovremeno kao što je prikazano na sledećoj slici.



Slika 4.7 Paralelni prenos podataka

Naravno da je paralelni prenos brži, ali se kod njega koristi više linija za povezivanje predajnika i prijemnika. To je ekonomično samo u slučajevima kada su predajnik i prijemnik sasvim blizu jedana drugom (kau u slučaju čipova, ili štampanih kola).

Treba takođe razlikovati i dve vrste veze između predajnika i prijemnika. Ako je prenos podataka moguć samo u jednom smeru (od predajnika ka prijemniku, ali ne i

obratno) govorimo o simpleks vezi. Ako je prenos moguć u oba smera, ali ne istovremeno, onda je to semi dupleks veza, ako je prenos moguć u oba smera i istovremeno oda je to tzv. ful dupleks veza.

Danas je uobičajeno da se računari povezuju u loaklne mreže (LAN) unutra nekog ograničeng prostora, a takođe i na širem prostoru (WAN mreže), pa i globalno na Internetu. U svi tim slučajevima imamo dvosmernu vezu pa možemo podatke i da primamo (download) i da šaljemo (upload) od/do udaljenih ralunara.

Brzine prenosa podataka unutar samog računara su reda veličine terabita u sekundi, brzine u lokalnim mrežama se mere gigabitima u sekundi, dok su veze sa intenetom reda megabita u sekundi.

Prezentacija podataka

Prezentacija podataka je finalna faza svake obrade podataka. To je ono što se vidi na kraju. Korisnik najčešće nema uvid u to kako komjuter obavlja aritmetičke i logičke operacije, obrađuje sliku ili zvuk, njega interesuje samo krajnji rezutat: da vidi svoj novac na svom bankarskom računu, da čuje pesmu koju je poželeo, da vidi spot koji ga interesuje i slično. On želi da do svih tih informacija dođe na što jednostavniji način, što brže i da pri tom ima i estetski doživljaj – da ekran prikazuje neke za oko pogodne prizore.

Upravo tu komunikaciju kompjutera i korisnika nazivamo interfejsom čovek-kompjuter.

Tokom razvoja kompjutera ova komunikacija je sve pogodnija i pogodnija za čoveka. Sa prvim kompjuterima 40-tih godina prošlog veka mogli su da komuniciraju samo njihovi konstruktori, krajem prošlog veka to su mogli samo posebno obučeni operateri, a danas kompjuter koriste skoro svi. U Švedskoj recimo više od 80% populacije koristi Internet. I kod nas se taj broj rapidno uvećava iz godine u godinu.

Verovatno neće proći dugo vremena kada ćemo sa računarima komunicirati ne samo preko tastature, ekrana i miša, već i na mnogo jednostavniji način – prirodnim glasom kao i sa ljudima. Već sada su u upotrebi govorni automati koji upravo to rade ali u još uvek primitivnom obliku. Da ne govorimo o posebnim naočarima, odelima i heptičkim uređajima kojima se mogu ostvariti i druge virtuelne komunikacije čovek kompjuter.

Mi ćemo se za sada držati klasičnih oblika komunikacije i truditi se da postignemo sa njima što bolje efekte.

Pitanja

1. Šta je bit, bajt?
2. Šta je RAM i ROM?
3. Šta je to memorijska adresa?
4. Kako se određuje pozicija bita u bajtu?
5. Koliko bitova ima jedan kilobajt?

6. Šta je to fajl?
7. Šta je to folder?
8. Koliko fajlova može biti u jednom folderu?
9. Koliko foldera može biti na jednom disku?
10. Mogu li dva foldera imati isto ime?
11. Mogu li dva fajla imati isto ime?
12. Šta je serijski prenos podataka?
13. Šta je paralelni prenos podataka?
14. Šta je simpleks, semidupleks i dupleks prenos podataka?
15. Koliko bajtova je potrebno za 1 minut muzike?
16. Koliko bajtova je potrebno za jednu stranicu teksta?
17. Koliko bajtova je potrebno za jenu sliku?
18. Koliko bajtova je potrebno za 1 minut video zapisa?
19. Šta je to GUI?
20. Koje nestandardne periferne uređaje poznajete?
21. Šta je to heptika?

5. Osnovni tipovi podataka

U programskim jezicima tipovi podataka (data types) definišu skupove vrednosti I doyvoljene operacije sa tim vrednostima. Na primer u C# jeziku, tip podatka “int” predstavlja skup celih 32-bitnih celih brojeva u rasponu od -2,147,483,648 do 2,147,483,647 sa kojima možemo vršiti operacije sabiranja, oduzimanja i množenja. Skoro svi programski jezici definišu tipove podataka kao što su: celi brojevi (integers, skraćeno int), realni brojevi (float), slova (character, sktaćeno char), logičke vrednosti (boolean, skraćeno bool).

Na primeru C# jezika, pokazaćemo skupove vrednosti za osnovne tipove podataka kao i operacije sa njima.

Celi brojevi

Kao što je već pomenuto, celi brojevi (engleski integers), su brojevi koji se definišu (deklarišu) pomoću skraćenice int kao u sldećemo primeru:

int kolicina;

Ovde reč “kolicina” označava promenjivu vredost (varijablu) koja može da dobije bilo koju vrednost iz skupa celih brojeva -2,147,483,648 do 2,147,483,647.

Sa celim projevima možemo da vršimo operacije sabiranja, oduyimanja i množenja koje se označavaju u većini programskih jezika na sledeći način:

Sabiranja: koristi se znak + kao što je uobičajeno u matematici.
Oduzimanje: koristi se znak - , takođe uobičajeno u matematici.
Množenje: koristi se znak * (zvezdica), što nije uobičajeno u matematici (tamo se koristi znak x).

Celi brojevi se ne mogu deliti, jer se kao rezultat ne dobija uvek ceo broj, pa bi se tako narušio princip da rezultat operacije nad nekim tipom podatka mora bisti isti takav tip. Zato se deljenje kod celih brojeva definiše tako da se kao rezultat deljenje dobija samo celi rezultat. Na primer 9 podeljeno sa 2 je 4 (ostatak pri deljenju se odbacuje). Zato je i 2 podeljeno sa 3 jednako 0.

Deljenje se označava sa znakom / (kosa crta).

Realni brojevi

Realni brojevi su brojevi koje mi u svakodnevnoj upotrebi nazivamo decimalnim brojevima (sa decimalnim zarezom, ili tačkom). Na engleskom se decimalna tačka naziva „floating point“ pa otuda skraćenica float za takve brojeve.

U primeru:

float cena;

cena označava promenjivu vredost (varijablu) koja može da dobije bilo koju vrednost iz skupa decimalnih brojeva u opsegu $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$.

Operacije sa realnim (float) brojevima su: sabiranje, oduzimanje, množenje i deljenje, koje se označavaju simbolima +, -, *, / respektivno i imaju uobičajeno značenje kao u matematici.

Slovni podaci

Slova (characters na engleskom) je takav tip podatka koji može da sadrži jedno slovo iz ASCII ili Unicode tabele.

Tako na primer:

char slovo;

označava varijablu koja može da sadrži bilo koji slovni znak koji je zastupljen u Unicode-u. Povezivanje više slova u tzv. string, postiže se mogućnost deklarisanja teksta kao podatka. Ali o nizovima će biti reči na drugom mestu.

Od operacije sa slovima pomenućemo konverzije iz jednog sistema u drugi (npr. iz ASCII u Unicode i obrnuto), kao i operacije "sabiranja" koje imaju specifično značenje.

Sabiranje slovima:

"a" + 1 daje rezultat "b" (jer je slovo b iza slova a)
"a"+2 daje rezultat "c", itd.

Logički podaci

Logički (boolean na engleskom, bool skraćeno) je veoma poseban tip podatka koji ima samo dve vrednosti tačno (na engleskom true) i netačno (na engleskom false).

Tako na primer:

bool iskaz;

iskaz označava varijablu koja može da sadrži jednu od dve vrednosti (true ili false).

Operacije sa logičkim podacima su: "logičko I" (na engleskom AND), "logičko ILI" (na engleskom OR), "logičko NE" (na engleskom NOT).
Za operacije AND, OR i NOT koriste se redom zbaci &&, ||, i !. Za logičko "računanje" koristi se sledeća tabela logičkih operacija:

X	Y	X && Y	X || Y	! X
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Pomenućemo sada i jednan tip podataka koji je u C jeziku odomaćen iako ne predstavlja nikakavu novu vrstu podataka koja već nije uključena u gore navedene.

Radi se o tipu **byte** koji je prosto niz od 8 bitova pa može da ima vrednost od 0 do 255.

Tako recimo u primeru:

byte podatak;

varijabla podatak može da uzme bilo koju vrednost između 0 i 255. Ovaj tip podatka se najčešće koristi za tzv. byte strimove (byte streams) o kojima će biti reči na drugom mestu.

Treba na kraju napomenuti da u programskim jezicima obično postoje i razne modifikacije napred navedenih osnovnih tipova podataka.

S druge strane moguće je gore navedene tipove podataka grupisati s strukturirati na različite načine da bi se dobile takve strukture podataka koje na najbolji način reprezentuju problem koji se rešava. O strukturama podataka biće reči u natstavku.

Pitanja

1. Šta su tipovi podataka?
2. Koji su osnovni tipovi podataka?
3. Šta je int tip podataka?
4. Koje vrednsoti može da uzme int tip podatka?
5. Koje su operacije moguće sa int tipovima podataka?
6. Kakva je specifičnost deljenja kod int podataka?
7. Šta su realni podaci?
8. Koje vrednsoti može da uzme float tip podatka?
9. Koje su operacije moguće sa float tipovima podataka?
10. Koji se simboli (znaci) koriste za operacije nad int i float podacima?
11. Šta je char tip podatka?
12. Koje vrednosti može da uzme char tip podatka?
13. Koje su operacije moguće sa char podacima?
14. Šta je bool tip podatka?
15. Koje vrednosti može da uzme bool tip podatka?
16. Koje su operacije moguće sa bool podacima?
17. Šta je byte tip podataka?
18. Koje vrednsoti može da ima byte tip podataka?

6. Osnovne strukture podataka

Strukture podataka se bave organizacijom podataka i načinom njihovog memorisanja u kompjuterima.

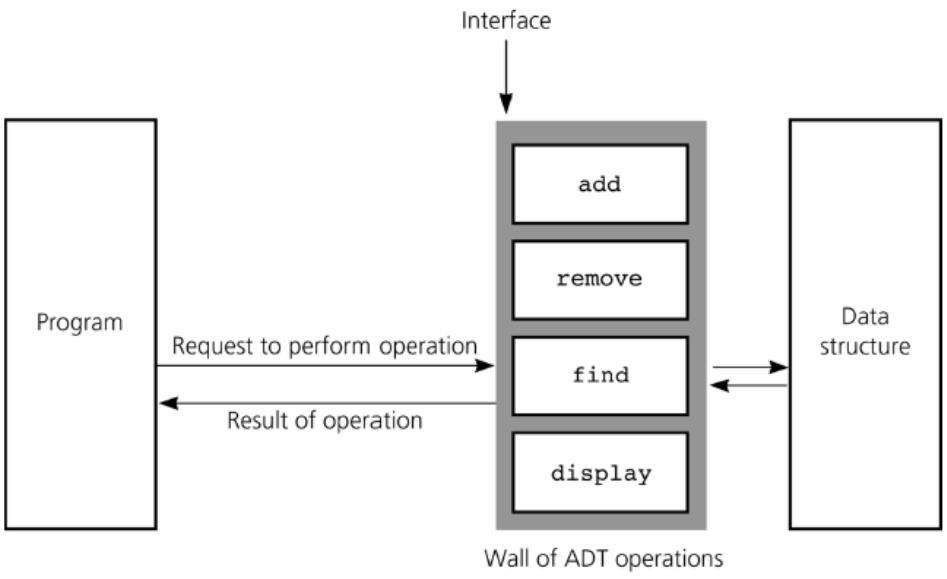
Struktura podataka je takva konstrukcija koja se može izraziti programskim jezikom za memorisanje skupa međusobno povezanih podataka - npr. niza celih brojeva, niza objekata, niza nizova, itd. Često se kompjuterske strukture podataka nazivaju i apstraktnim strukturama, jer se podaci modeliraju tako da vrše apstrakcije koje na najbolji način odgovaraju problemu koji se rešava. Kaže se da se u načinu predstavljanja problema podacima zapravo krije i njegovo rešenja.

Apstrakna struktura podataka (Abstract Data Structure-Type) - ADT

Skup (kolekcija) povezanih podataka sa skupom operacija nad tim podacima naziva se apstraktnom strukturom podataka. Apstrakna struktura pokazuje koje su operacije definisane nad podacima, ali ne i kako se te operacije implementiraju. Programer, znači, može da koristi apstaraktnu strukturu bez znanja kako je ona implementirana.

Operacije nad strukturama podataka

Svaka (apstraktna) struktura podataka ima sa njom povezan skup operacija kojima se uspostavlja sama struktura (konstruktorske operacije), kao i niz operacija za dodavanje, brisanje, sortiranje i pretraživanje podataka memorisanih unutar strukture, kao što je ilustrovano su na sledećem dijagramu.



Slika 6.1 Upotreba apstarktnih struktura podataka

Gornja slika slikovito predstavlja poznatu “formulu”:

PROGRAM = ALGORITAM + STRUKTURA PODATAKA

Gornjom formulom se izražava povezanost strukture podataka sa rešenjem (algoritmom) problema.

Slika prikazuje kako program preko interfejsa (operacije, funkcije, procedure, metoda) koristi strukturu podataka. U objektno orjentisanoj metodologiji zapravo struktura podataka zajedno sa interfejsom predstavlja osnovni element OO pristupa. Tako se definiše klasa objekata koji su modelirani strukturom podataka i metodama njihove manipulacije (dinamičkog kreiranja objekta, modifikacije objekta, i sl). OO pristup se može smatrati generalizacijom gornje formule o programima.

O Objektno orjentisanom programiranju biće više reči kasnije.

Apstrakne strukture podataka imaju niz prednosti nad klasičnim pristupom, jer s jedne starne precizno definišu objekte, a s druge strane olakšavaju programiranje jer:

- Sakrivaju nepotrebne detalje (implementaciju)
- Olakšavaju upravljanje razvojem kompleksnog softvera
- Olakšavaju održavanje softvera
- Smanjuju potrebu za funkcionalnim promenama softvera
- Omogućavaju lokalne a ne globalne promene

Nizovi i indeksi (pointeri)

Niz predstavlja kolekciju objekata (podataka) istog tipa smeštenih u memoriji u uzastopnim memorijskim lokacijama kao što prikazuje sledeća slika.

P[0]	P[1]	...	P[n-2]	P[n-1]

U gornjem primeru prikazan je slučaj jenog niza od n elemenata (zapazite da brojanje počinje od 0), gde čitav niz ima zajedničko ime **p**.

Pojedinačnom elementu niza pristupa se tako što se pored imena niza u zagradi navede i redni broj (indeks) tog elementa u nizu. Tako je p[14] podatak koji se nalazi u 15-tom elementu niza p.

Nizovi su standardan element u programskim jezicima i obično se definišu (deklarišu) na početku programa kao u sledećem primeru iz jezika C.

int kolicina[100];

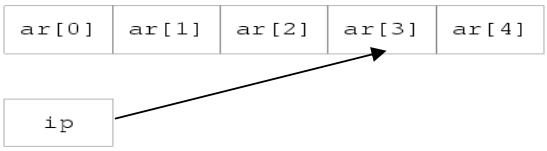
Ovde je deklarisan niz sa imenom od 100 elemenata, celobrojnog tipa.

POINTERI

Pointer predstavlja specijalnu vrstu podataka koji se interpretiraju kao adrese na podatke, a nisu podaci sa nekim drugim značenjem.

```
int ar[5], *ip;
```

Gornjom deklaracijom definisan je niz p i pointer pp



ip = &ar[3];

Ovde je ip pointer na četvrti element niza ar.

Všedimenzionalni nizovi

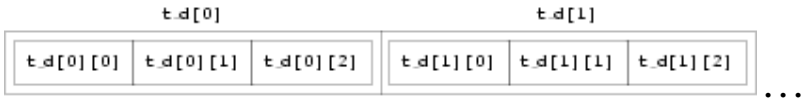
Možemo zamisliti i nizove sa više dimenzija (indeksa). Tako bi dvodimenzioni niz deklarisan sa:

```
int t_d[2][3]
```

mogao biti predstavljen kao dvodimenziona šema (matrica)

t_d[0][0]	T_d[0][1]	t_d[0][2]
t_d[1][0]	T_d[1][1]	t_d[1][2]

dok bi mogao da ima sledeće preslikavanje u memoriji računara.



...

```
Trodimenzioni niz:

int three_dee[5][4][2];
```

ima 5x4x2=40 elemenata, gde se svakom od elemenata pristupa preko tri indeksa.

Ovaj niz bi slikovito mogao da bude prikazan kao paralelopiped (trodimenzionalno telo) izdeljeno na 40 "kućica", gde svaka kućica ima svoju adresu izraženu sa tri indeksa, kao npr. three_dee[3][2][1].

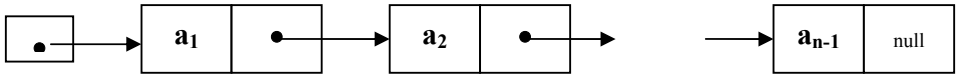
Iako je moguće zamisliti i nizove sa više od tri indeksa, njihova vizuelizacija nije moguća.

Treba ovde još istaći da elementi niza mogu biti proizvoljni objekti, pa čak i nizovi. Tako nizovi mogu sadržati kao elemente brojeve (cele i decimalne), slova, slike, itd.

Liste

Lista je konačan skup podataka $a_1,a_2,a_3,...,a_n$ sličan nizu koji smo ranije definisali. Međutim, za razliku od niza liste mogu biti realizovane kao dinamičke strukture kod kojih se unapred ne zna broj elemenata.

Sledeća slika ilustruje jednu jednostruko povezanu listu.



Slika 6.2 Lista

Liste su vrlo rasprostranjene u programiranju – lista događaja, lista poruka, lista klasa itd.

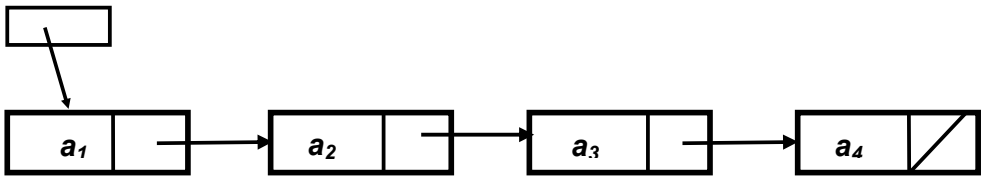
Tipične operacije sa listama:

- Kreiranje liste
- Dodavanje (Insert) elementa u listu
- Brisanje (Remove) elementa liste
- Test da li je lista prazna (bez članova)
- Traženje elmenta u listi
- Trenutni element/ sledeći/ prethodni
- Nađi k-ti element
- Štampaj celu listu
- Itd.

Implementacija lista POINTERIMA

Sada ćemo prikazati jedan efikasniji način implementacije liste pomoću pointera, dinamički.

Glava liste – pointer na prvi element

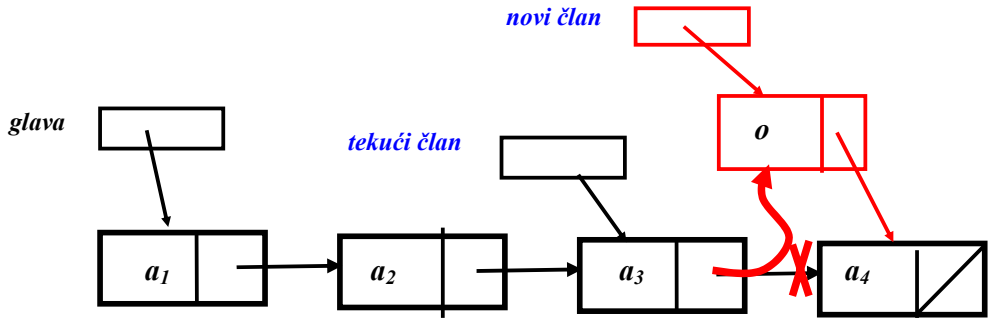


Slika 6.3 Implementacija liste pointerima

Dakle, kod implementacije pointerima lista se sastoji od jednog posebnog pointera koji ukazuje na početak liste, a svaki element u listi se sastoji iz dva dela (za podatke i pointera koji ukazuje na adresu sledećeg elementa liste), kako je to prikazano na gornjoj slici.

Sada ćemo ilustrovati kako se operacije insert i delete sa lakoćom realizuju u ovom slučaju, kada je lista implementirana pointerima.

Operacija INSERT



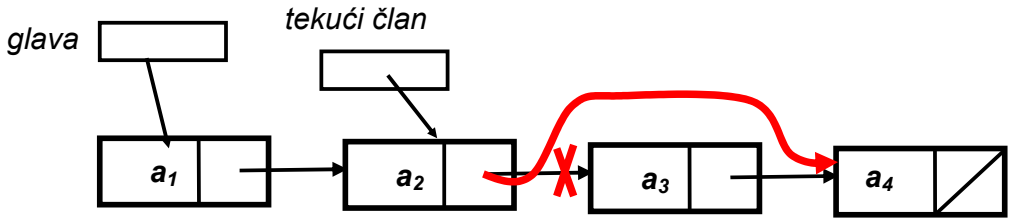
Slika 6.4 Dodavanje novog člana u listu

Ako novog člana liste treba dodati posle tekućeg člana (a_3), onda se takva operacija može opisati na sledeći način.

Jednostavno se pointer člana a_3 promeni da pokazuje na novi član, a pointer novog člana uzme predhodnu vrednost pointera člana a_3. Znači, nema nikakvog pomeranja članova niza, već se prostom zamenom pointera izvrši dodavanje novog člana.

Operacija DELETE

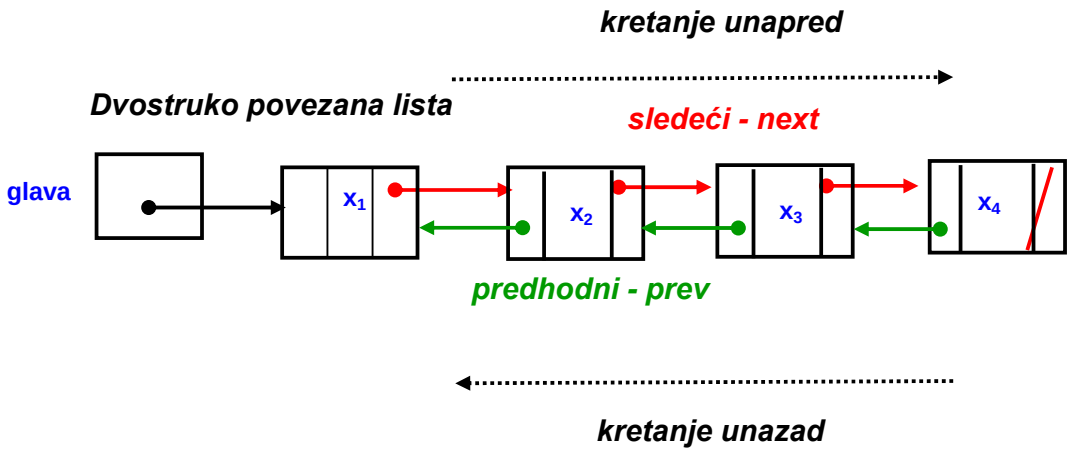
Kod brisanja je stvar još jednostavnija. Prosto se članu koji predhodi brisanom članu promeni pointer da ukazuje na član koji sledi posle brisanog člana, kako to prikazuje sledeća slika.



Slika 6.5 Brisanje člana iz liste

Dvostruko povezane liste

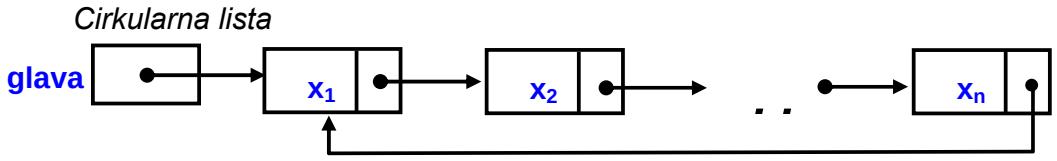
Ponekad je pogodno da se kretanje kroz listu, od elementa do elementa vrši u oba smera (napred i nazad). U takvim slučajevima organizuju se liste sa dva pointera, jedan za kretanje unapred kao kod obične liste, i dodatni pointer za kretanje unazad, što je ilustrovano na sledećoj slici.



Slika 6.6 Dvostruko povezana lista

Cirkularne liste

Cirkularna lista nastaje kada umesto da poslednji član liste ima NULL pointer (NULL pointer je pointer koji ne pokazuje ni na jedan element) on sadrži adresu prvog člana. Time se omogućava kružno-cirkularno kretanje kroz listu, kao što se možete uveriti posmatranjem sledeće slike.



Slika 6.7 Cirkularna lista

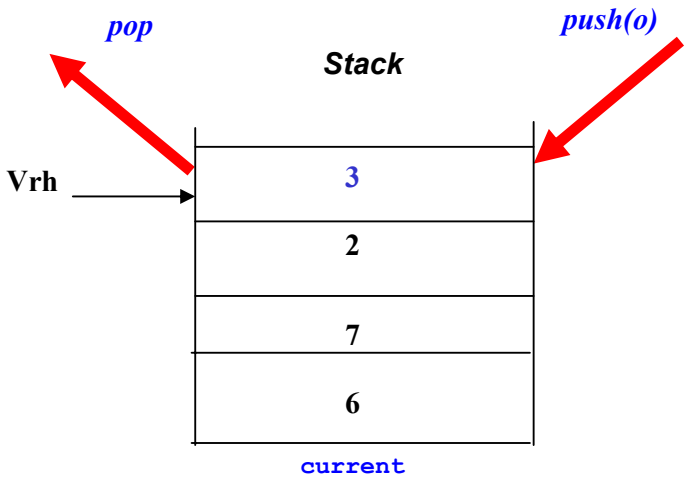
Generalno govoreći, liste su jedna od najčešće korišćenih apstraktnih struktura i služe za implementacijo mnogih drugih i složenijih struktura. Posebno je pogodna kada se implementira pointerima i dinamički. Objektno orjentisani jezici obično sadrže u biblioteci klasu koja odgovara napred izloženom konceptu liste i iz nje izvedenih drugih struktura o kojima će sada biti reči.

Stekovi (LIFO lista, Stack)

Stek je lista kod koje se dodavanje i brisanje člana uvek vrši na jednom mestu, tj. na vrhu steka (početku liste).

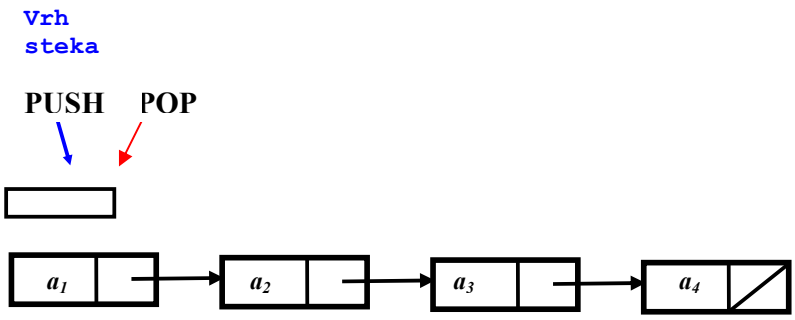
Operacije dodavanja i brisanja u steku imaju posebne nazive: push (dodaj-insert) and pop (obriši-delete).

Stek se može slikovito prikazati na sledeći način:



Slika 6.8 Primer steka

Implementacija pomoću liste



Slika 6.9 Implementacija steka pomoću liste

Kod operacije POP pointer vrha uzima vrednost koju je imao pointer prvog elementa steka. Operacijom PUSH pointer vrh pokazuje na novi element a novi element na prvi element pre izvođenja PUSH operacije.

Stek ima raznovsne primene kao što su:

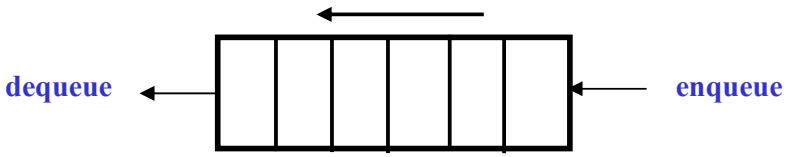
- *kod editora teksta (line editing)*
- *uparivanje zagrada (leksička analiza)*
- *izračunavanje u postfiks notaciji*
- *rekurziji i pozivu potprograma (funkcije)*

Redovi (FIFO liste, Queue)

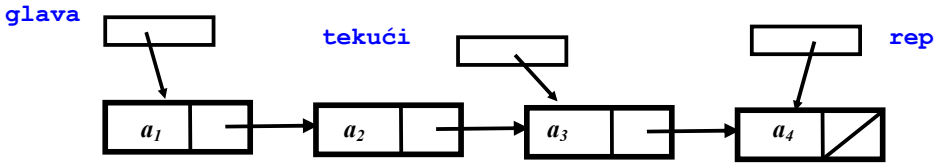
Kao i stekovi, redovi se takođe mogu implemetirati kao liste. Kod redova se, međutim, dodavanje vrši na jednom kraju a brisanje na drugom kraju liste.

Redovima se implementira FIFO (first-in first-out) princip. Npr., printer/job red.

Dve osnovne operacije sa redovima:
–dequeue: brisanje elementa sa čela reda
–enqueue: dodavanje člana na kraj reda



Implementacija pomoću liste

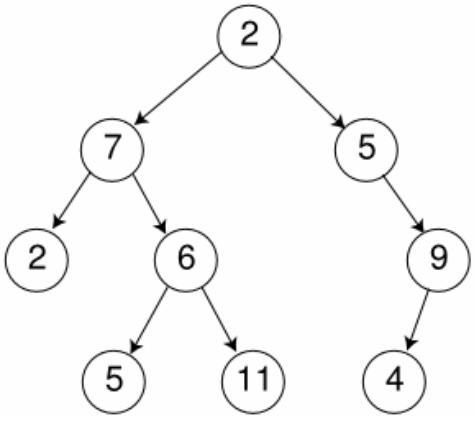


Slika 6.10 Implementacija reda pomoću liste

Primene redova
- Printer redovi za čekanje na čtampu,
- Telekomunikacioni redovi ,
- Simulacije,
- Itd.

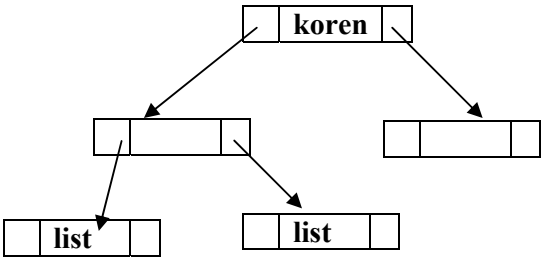
Binarno stablo (Binary Tree)

Binarno stablo je struktura podataka koji su povezani tako da svaki od podataka (čvorova stabla) može imati najviše dva sukcesora. Elementi binarnog stabla su: Koren (root), unutrašnji elementi i listovi (leaf) stabla. Koren (root) je čvor koji se nalazi na vrhu stabla



Slika 6.11 Primer binarnog stabla

Implementacija listama



Slika 6.12 Implementacija binarnog stabla listom

Svaki element u stablu ima pored podataka i dva pointera koji pokazuju na levo i desno podstablo (ako ih ima, ili NULL ako ih nema).

Operacije nad binarnim stablima

- dodavanje novog elementa
- brisanje postojećeg elementa
- prolazak: preorder, inorder, postorder

Dodavanje i brisanje elenata stabla vrši se promenom pointera slično kao kod lista (ako su stabla implementirana pomoću liste), tako da se nećemo posebno baviti ovim operacijama. Samo ćemo, nešto kasnije, pokazati jednu strategiju za dodavanje elemenata u stablo koja će rezultirati jednom specijalnom vrstom stabla – takozvanim sortnim binarnim stablom.

Malo detaljnije ćemo se baviti operacijama sistematskog obilaska svih elemenata stabla – takozvanim algoritmima za prolazak kroz stablo. Ovo iz razloga što ovi algoritmi dobro ilustruju jedan važan koncept u programiranja – takozbanu rekurziju.

Algoritmi prolaska kroz binarno stablo.

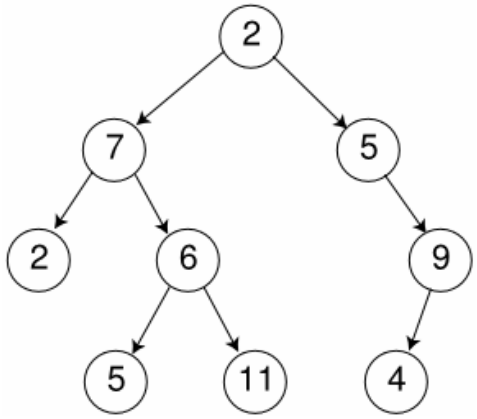
Preorder algoritam

Preorder algoritam se može iskazati na sledeći načina, sa tri koraka.

Preorder: 1. Poseti koren
2. Poseti levi deo stabla (levo podstablo)
3. Poseti desni deo stabla (desno podstablo)

Korak 1. je eksplicitan. U njemu se posećuje koren, tj jest čvor koji je na vrhu stabla (ili podstabla, kada se posećuje podstablo). Koraci 2 i 3 su implicitni (rekurzivni), jer nam ne kažu koji čvor treba posetiti, već nam daju uputstvo kako nastaviti sa prolazon – kroz levo ili desno podstablo. Zapravo poseta levom (ili desnom) podstablu će se vršiti istim ovim **preorder** algoritmom, ali sada primenjenim na podstablu. I tako redom dok god ima novih podstabala.

Za primer preorder prolaska kroz stablo uzmimo stablo sa sledeće slike.



Ako primenimo preorder algoritam imamo:

Korak 1. Posetimo koren stabla. Dakle posetimo vrh koji sadrži podatak 2 i označimo ga posećenim.

Korak 2. Sada treba da posetimo levo podstablo, tj stablo koje ima čvorove 7,2,6,5,11. Kako da ga posetimo. Pa opet istim algoritmom, ali sada primenjenim na ovo podstablo. A to znači:

Korak 1. Poseti koren. Znači poseti i označi posećenim čvor 7 (koji je koren podstabla koji upravo posećujemo).

Korak 2. Poseti levo podstablo tekućeg podstabla. To je podstablo koje se sastoji od samo jednog čvora – čvora 2. Tako imamo:

Korak 1. Poseti koren – označi 2 posećenim.

Korak 2. Nema levog podstabla.

Korak 3. Nema desnog podstabla.

Znači nastavljamo sa korakom 3 za levo podstablo originalnog stabla.

Korak 3. Poseti desno podstablo. To je podstablo sa čvorovima 6,5,11

Korak1. Poseti koren – to je čvor 6.

Korak 2. Poseti levo podstablo. To je samo jedan čvor – 5.

Korak 3. Poseti desno podstablo. To je samo jedan čvor – 11.

Sada se vraćamo na Korak 3 za pocetno stablo.

Korak 3 Sada treba da posetimo desno podtablo, tj stablo koje ima čvorove 5,9,i 4.

Korak1. Poseti koren – to je čvor 5 oynačimo ga posećenim.

Korak 2. Poseti levo podstablo – nema

Korak 3. Poseti desno podstablo – to je stablo sa čvorovima 9 i 4.

Korak 1. Poseti koren – to je čvor 9, označi ga posećenim.

Korak 2. Poseti levo podstablo – to je samo jedan čvor, 4, označi ga posećenim.

Korak3. Poseti desno podstablo – nema.

Time je algoritam završen. Redosled posete čvorova je sledeći: 2,7,2,6,5,11,5,9,4.

Sličan postupak se sprovodi i za druga dva načina prolsaka kroz stablo. Za vežbu odredite redosled prolaska kroz gornje stablo za inorder I postorder redolsede.

Inorder: 1. Poseti levi deo
2. Poseti root
3. Poseti desni deo

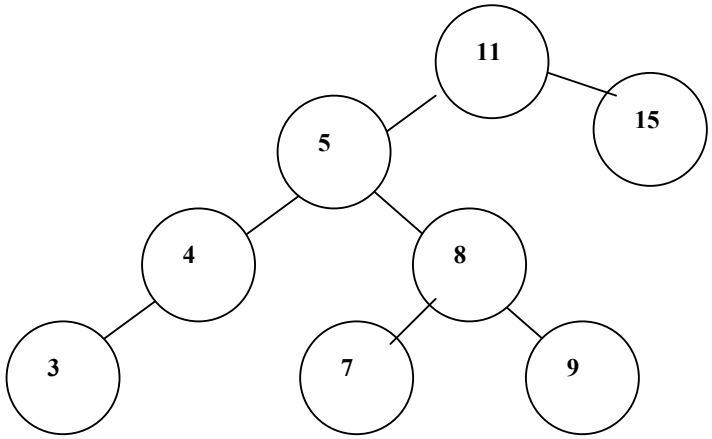
Postorder: 1. Poseti levi deo
2. Poseti desni deo
3. Poseti root

Sortno binarno stablo

Sortno binarno stablo je stablo koje se dobija kada se niz brojeva (ili nekih drugih objekata) unose u stablo po sledećem algoritmu.

1. Prvi pristigli broj (objekat) se unosi u koren stabla.
2. Svaki sledeći se rekurzivno unosi u stablo tako da ako je manji od broja (objekta) u korenu ide u levo podstablo, a ako je veću u desno.

Tako na primer niz brojeva 11,5,8,4,3,9,7,15 biće smešten u sortno binarno stablo kao što je prikazano na sledećoj slici.



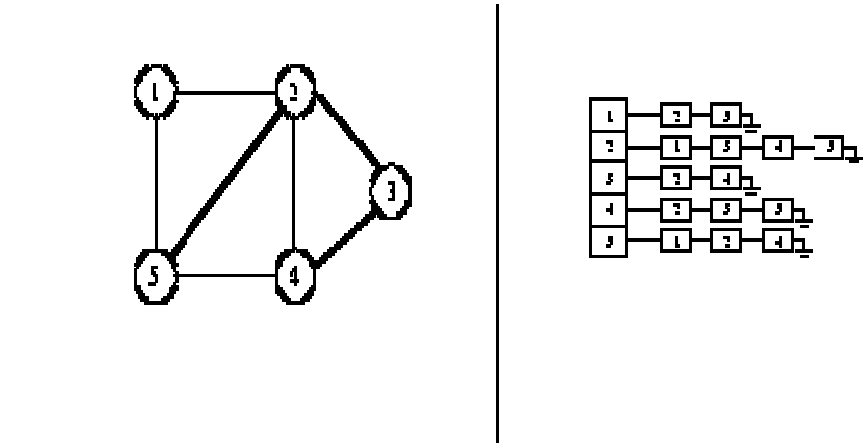
Slika 6.13 Sortno binarno stablo

Ako se sada kroz ovo stablo prođe inorder redosledom dobićemo niz: 3,4,5,7,8,9,11,15. Zapravo dobićemo sortiran ulazni (početni niz). Dakle, ovaj postupak se može koristiti za sortiranje (veoma efikasno) nizova brojeva (objekata).

Grafovi G(nodes, vertices) – čvorovi i lukovi.

Grafovi su posebne mrežne strukture koje nalaze primenu u raznim algoritmima optimizacije, teoriji igara, telekomunikacijama, itd.

Na sledećoj slici prikazan je jedan graf i njegova implementacija kombinacijom nizova i lista. Kružići u grfau predstavljaju takozvane čvorove (engleski nodes) , a linije lukove (engleski arcs).



Slika 6.14 Graf

Graf se može implementirati kao niz lkoji sadrži čvorove uz koje je pridružena lista susednih čvorova sa kojima je čvor povezan. Tako se na predhodnoj slici nalazi lista sa 5 elemenata koja sadrži čvorove 1 do 5. Svaki element iz niza ima pointer na listu. Tako čvor 1 ima pointer na čvor 2 a ovaj na čvor 5, jers su čvorovi 2 i 5 susedni čvoru 1. Slično tome, čvor 2 ima pointer na listu čvorova 1,3,4 i 5 jer je čvor 2 povezan sa svim tim čvorovima.

Algoritmi sa grafovima su složeniji i izlaze van okvira ovog predemeta. Ovde ćemo samo napomenuti da se za sistematski prolaz kroz graf mogu koristiti dve osnovne startegije: prvo u dubinu (depth first) ili prvo u širinu (breadth first). Kod prve se najpre ide po dubini grafa, tj. ide se sve dalje od početnog čvora, a kod druge se najpre posećuju susedni čvorovi.

Heš tabele

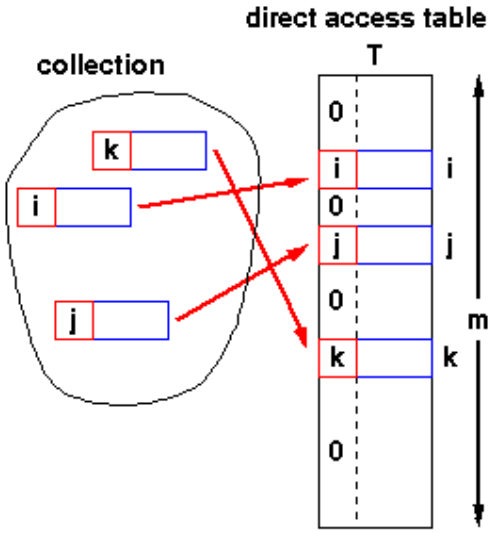
Heš tabele je posebna struktura podataka koja podseća na asocijativnu memoriju kod čoveka. U toj strukturi sam podatak koji se traži ukazuje na moguću adresu na kojoj se on nalazi u memoriji. To se postiže takozvanom transformacijom podatka (ključa) u adresu na sledeći način:

adresa= f(ključ), gde je f neka matematička funkcija.

Najčešće se primenjuju funkcije opisane metodama 1,2 i 3.
Metod 1: adresa = (ključ) MOD N + 1
Metod 2: promena baze (radix conversion)
Metod 3: kvadriranje ključa i uyimanje dela rezultata

Implementacija heš tabele nizovima

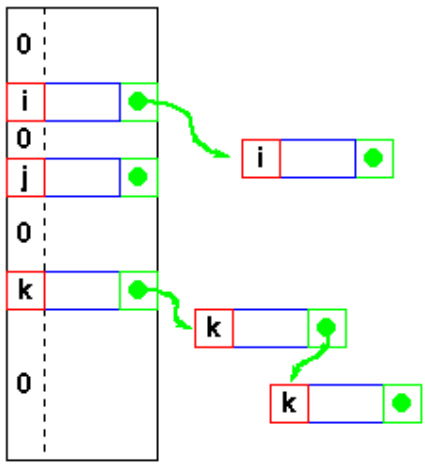
Podaci iz neke kolekcije podataka imaju poseban deo koji se zove ključ i koji služi za izračunavanje adrese na kojoj će podataka (zajedno sa ključem) biti smešten. Taj postupak je ilustrovan na sledećoj slici.



Slika 6.15 Heš tabela

Međutim, postoji verovatnoća da će transformacija dva različita ključa dati istu adresu, tako da dolazi do takozvane kolizije ključeva. Tada bi oba ključa trebalo smestiti u istu adresu, što je naravno nemoguće. U takvim slučajevima se pribegava posebnom postupku razrešavanja kolizije, što je moguće postići na više načina.

Na sledećoj slici je prikazano rešavanje kolizije olančavanjem (chaining).



Samo ćemo navesti još neke metode rešavanja kolizije bez dalje elaboracije:

1. Posebno područje memorrije za koliziju (overflow areas),
2. Primena alternativne heš funkcije (re-hashing),
3. Smeštanje na slučajno odabranu adresu (random probing)

Pitanja

1. Navedite bar tri osnovne strukture podatak.
2. Šta je ADS (ADT)?
3. Navedite neke od operacija nad strukturom podataka.
4. Koje su osnovne karakteristike ADT-a?
5. Definišite poiter (pokazivač).
6. Koliko elemenata sadrži niz **triD** definisan sa : int triD [10][3][5]? A koliko bitova, ako int zauzima 4 bajta?
7. Koliki je maksimalan i minimalan indeks niza deklarisanog sa: char niz [100]?
8. Koje podatke o listi (linked list) najčešće moramo da imamo?
9. Koje su najčešće operacije nad elementima liste?
10. Navedite način na koje se liste mogu implementirati?
11. Kakva je prednost dvostruko povezanih lista u odnosu na obične (jednostruko povezane) liste?
12. Definišite cirkularnu listu?
13. Opišite FIFO i LIFO principe.
14. Koje podatke o steku najčešće moramo da imamo?
15. Koje su najčešće operacije nad stekom?
16. Navedite bar dve primene steka.
17. Koje podatke o redovima najčešće moramo da imamo?
18. Koje su najčešće operacije nad redovima?
19. Navedite bar dve primene redova.
20. Nacrtajte primer nekog binarnog stabla i označite koren (root) i lišće (leafs) stabla.
21. Na primeru nekog stabla (nacrtati stablo i njegove čvorove označiti slovima a,b,c,d,...) ispisati redosled posete čvorova za:
    A. preorder
    B. inorder
    C. postorder

    prolaske kroz stablo.
22. Šta je sortno binarno stablo?
23. Definisati hešing postupak za memorisanje podataka.
24. Na kojoj adresi će se naći podatak 248 ako se za hešing koristi sledeća transformacija ključa u adresu:
    a) adresa = (kljuc) MOD 300 + 1
    b) kvadriranje ključa i uzimanje prve tri cifre.

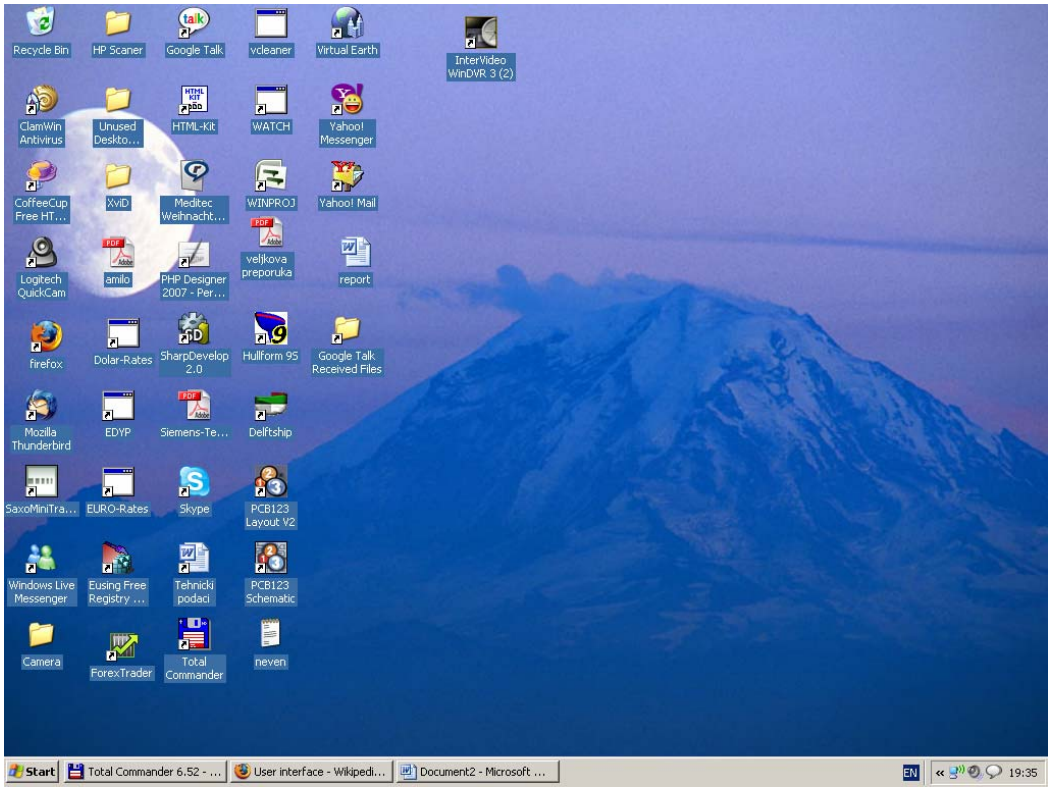
7. Interakcija čovek-kompjuter

Interakcija između čoveka i kompjutera (Human-Computer Interaction) je multidisciplinarna studija odnosa korisnika (user) i kompjutera, koja taj odnos proučava sa aspekta kompjuterskih nauka, psihologije i dizajna, sa ciljem da se što više olakša upotreba računara od strane običnih korisnika. Korisnički intefejsi koji su deo ove interakcije obuhvataju i softver I hardver kojim se postiže komunikacija između čoveka i kompjutera.

U užem smislu, čovek-komjuter intefejs se odnosi na grafičku, tekstualnu ili audio komunikaciju čovek-komjuter. Mi ćemo se ovde upoznati sa nekim elementima te komunikacije.

Grafički Interfejs (Graphic User Interface – skraćeno GUI)

To je danas najrasprostranjeniji oblik interakcije čovek-kompjuter. Komunikacija se ostvaruje tako što čovek (korisnik) putem uređaja kao što su tastatura (keyboard), ili miša (mouse) saopštava svoje komande, a kompjuter odgovara grafičkim prikazima na monitoru. Tipičan primer su Majkrosoft Windows operativni sistemi (vidi sliku).



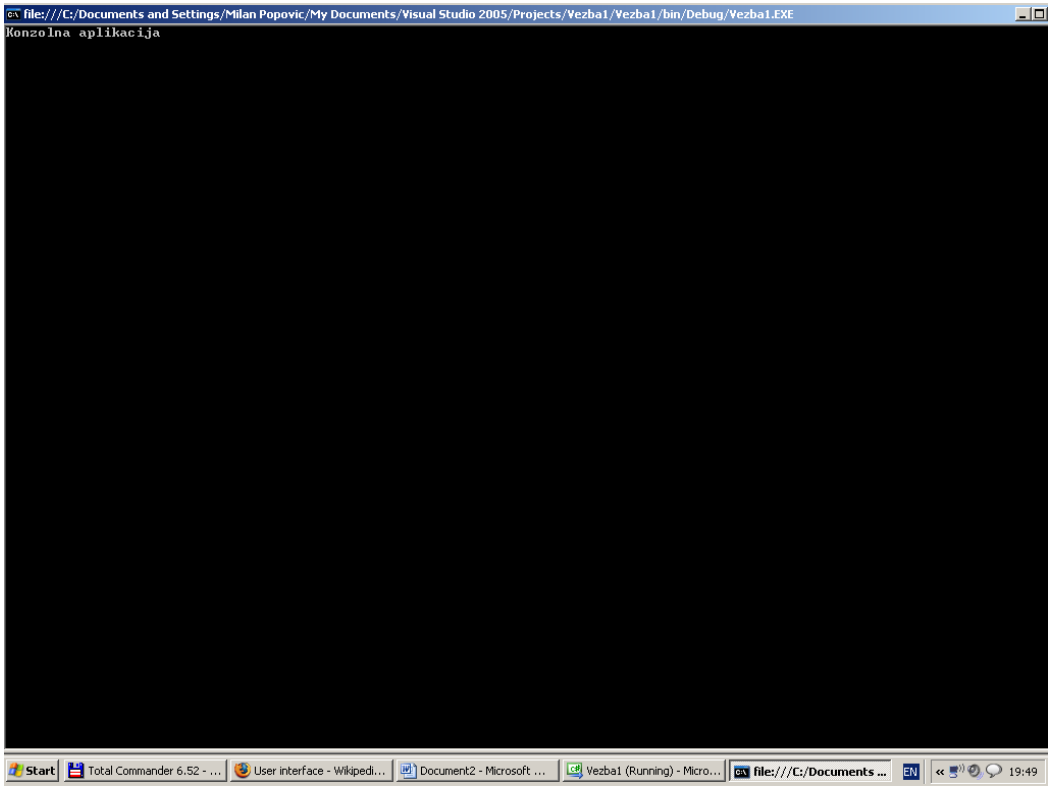
Slika 7.1 Windows XP grafički interfejs

Savremeni programski jezici, kao što je i C# pružaju mogućnost da programeri koriste isti pristup pri razvoju aplikacija, tako da programi pisani u tim jezicima imaju isti korisnički interfejs kao i operativni sistemi pod kojima aplikacije rade. To doprinosi

uniformnosti pri korišćenju i olakšava korisnicima snalaženje u radu sa aplikacijama kao i sa samim operativnim sistemima.

Linijski interfejsi (Command Line Interfaces)

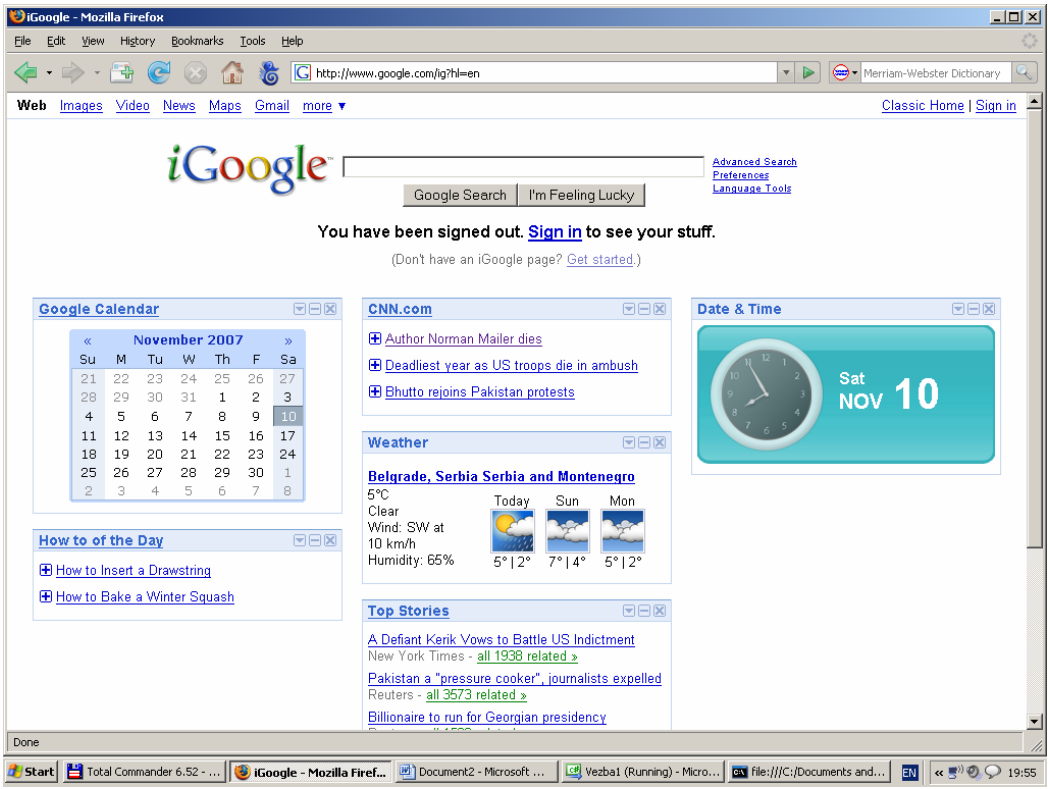
Kod linijskih interfejsa, korisnik koristi samo tastaturu za ukucavanje komandi, a i odgovor kompjutera je takođe u obliku teksta. To je nekada bio jedini način komunikacije čovek kompjuter (recimo pre pojave Windows-a, Majkrosoft je imao DOS operativni sistem sa linijskim interfejsom). Još uvek je takav interfejs u upotrebi za aplikacije koje ne zahtevaju grafičku interakciju. Čak i Windows opeartivni sistemi zadršali su takvu mogućnost, preko takozvanog „Command Prompt“ panela. U C# za aplikacije koje koriste linijski intefejs kaže se da rade u Console režimu (vidi sliku)



7.2 Linijski interfejs - konzolna aplikacija

Web orjentisani interfejs

Web interfejsi su specijalan oblik grafičkog interfejsa u kojem se korisi hipertekst za prikaz podataka i informacija. Razvoj web aplikacija vrši se jezicima kao što su HTML (Hyper Text Mark-up Laguage), Java, PHP i sl. Ovde se aplikacije nalaze na web serverima i preko interneta se aktiviraju web brauzerima (web browser) kao što je Explorer ili Firefoxm, na primer (vidi sliku).



7.3 Web interfejs

Specijalni iterfejsi

Gore navedeni intefjesi čovek-kompjuter su primeri klasičnih interfejsa koji su u najširoj upotrebi. Međutim postoji i niz specijalnih intefejsa koji omogućavaju i druge oblike komunikacije čovek računar. Nabrojaćemo ovde neke od njih:

Taktilni interfejsi
Audio intefejsi
Touch Screen interfejsi
Heptički intefejsi (specijalna odela)

Pitanja

1. Šta je to interakcija čovek-kompjuter?
2. Nabrojte nekoliko vrsta intefejsa čovek-kompjuter.
3. Šta je grafički interfejs?
4. Šta je linijski interfejs?
5. Šta je linijski interfejs?
6. Nabrojte neke specijalne interfejse?
7. Šta su heptički intefejsi?
8. Kako funkcioniše touch screen interfejs?
9. Šta su audio interfejsi (govorni automati)?

8. Datoteke

Datoteke su važna struktura podataka pa joj zbog toga posvećujemo posebno predavanje. Skoro da nema ni jedne aktivnosti koja se obavlja uz pomoć računara a da se pri tom ne koristi neka datoteka.

Štaje to datoteka?

Datoteka (engleski file – »fajl«) je logički organizovana skupina međusobno povezanih podataka.

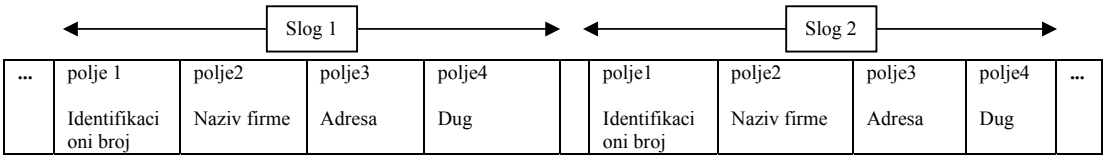
Pod logičkom organizacijom se podrazumeva da je datoteka logički izdeljena na manje delove – slogove koji čine jedinstvene celine podataka. Jedinstvenost se ogleda u tome što se slogovi međusobno razlikuju u bar nekom podatku koji sadrže.

I slogovi sami po sebi jesu strukture podataka, jer se sastoje od više podataka kojima odgovaraju polja (engleski field – »fild«) u slogu.

Pored logičke organizacije koja pokazuje logičke veze medju podacima u datoteci, za potpuno razumevanje koncepta datoteke potrebno je razmatrati i način na koji se podaci iz datoteke memorišu na nekom medijumu (traci, disku, itd...). Metode i tehnike memorisanja datoteka nazivaju se fizičkom organizacijom. Fizička organizacija zapravo pokazuje kako se podaci iz logičke datoteke preslikavaju u fizičke adrese (staze i sektore kod diskova na primer). Mi ćemo se baviti prevashodno logičkom organizacijom datoteka, to jest datoteke ćemo posmatrati sa aspekta programera.

Slogovi

Slogovi su logičke celine međusobno povezanih podataka kojima se modelira onaj entitet koji je predstavljen u datoteci. Tako će recimo datoteka KUPCI sadržati slogove koji sadrže potrebne podatke o kupcima, kao što je prikazano na sledećoj slici:



Kao što se iz predhodnog primera vidi slogovi se sastoje od jednog ili više uzastopnih polja koja sadrže podatke. Polja odgovaraju obeležjima ili atributima koje dodeljujemo entitetu koji se modelira. U predhodnom slučaju entitet su bili kupci, a taj entitet je modeliran atributima: identifikacioni broj, naziv firme, adresa i dug.

Dakle, polja sadrže elementarne podatke i obično su popunjena brojevima i slovima. Moguće su i polja koja sadrže i složenije vidove podataka, kao slike, zvučni zapisi i sl.

Datoteke obično sadrže veliki broj slogova, čiji broj se nekad meri i milionima slogova (za čuvanje biračkih spiskova na primer). Zato je veoma važno da logička i fizička organizacija podataka bude tako uređena da obezbedi efikasno korišćenje i manipulaciju podacima u datotekama.

Koje su osnovne operacije nad datotekama?

Razmotrićmo nekoliko osnovnih operacija koje se sreću pri radu sa datotekama.

Kreiranje datoteke

Kreiranje datoteke je operacija kojom se uspostavlja nova datoteka i priprema za sve naredne operacije. Kreriranjem se obično definišu osnovni podaci o datoteci kao što su: naziv datoteke, struktura slogova – nazivi polja i tipovi podataka u poljima, medijum na kojem se datoteka nalazi (traka, disk, CD, itd.), tip organizacije datoteke (o tipovima organizacije biće reči malo kasnije) itd.

Upisivanje slogova u datoteku

Datoteke se kreiraju sa namerom da u nima budu pohranjeni (memorisani) slogovi. Pohranjivanje slogova u datoetku vrši se operacijama upisa. Da bi se operacija upisa izvršila potrebno je datoteku otvoriti (operacija OPEN), zatim operacijama upisa (operacija WRITE) u datoteku smeštati slogove jedan za drugim i nakraju datoteku zatvoriti (operacija CLOSE) kojom se na kraj datoteke postavlja poseban znak-marker kraja datoteke (end-of-file marker).

Učitavanje slogova iz datoteke

Slogovi koji su predhodno upisani u datoteku mogu biti naknadno učitavani. Pod učitavanjem se ovde podrazumeva proces prenošenja podataka iz sloga na medijumu na kome se datoteka nalazi (traka, disk, CD) u centralnu (RAM) memoriju računara. Operacija učitavanja (READ) zahteva takođe da datoteka bude predhodno otvorena. Zavisno od načina logičke i fizičke organizacije datoteke moguće je slogove čitati u različitom redosledu. O tome će biti govora kasnije. Za sada ćemo smatrati da se slogovi čitaju jedan za drugim onako kako su i bili upisivani. Na kraju čitanja. datoteku treba ponovo zatvoriti.

Brisanje slogova

Ponekad je potrebno da se neki već upisani slog (ili više njih) odstrani iz datoteke. Ovu operaciju nazivamo brisanjem sloga (operacija DELETE). Za izvršavanje ove operacije potrebno je otvoriti datoteku, pronaći (locirati) slog koji se briže, izvršiti brisanje i zatvoriti datoteku. Kako se slogovi lociraju? Pa tako što se učitavaju redom slogovi i proverava identifikaciono polje da li odgovara slogu koji treba brisati.

Možda je ovo trenutak da definišemo identifikator sloga (datoteke). U gornjem primeru smo u datoteci KUPCI imali jedno polje – identifikacioni broj, koje je služilo jednistvenu identifikaciju sloga. To znači da datoteka KUPCI ne može da sadrži dva sloga sa istim identifikacionim brojem. Jedno ili više polja datoteke čiji sadržaj se ne sme ponoviti u drugim slogovima nazivaju se potencijalnim ključevima datoteke.

Ključevi su dakle jedinstveni za svaki slog i mogu služiti za njihovo pretraživanje i lociranje . O upotrebi ključeva biće više reči kada se budu razmatrale razlišite vrste organizacije datoteka.

Dodavanje slogova

Kada u već postojeću datoteku koja sadrži slogove treba dodati slog sa podacim, tada se operacija dodavanja vrši upisivanjem novog sloga, ato znači da datoteku treba otvoriti, locirati mesto gde će novi slog biti dodat, upisati slog i zatvoriti datoteku. Ponovo imamo pitanje lociranja mesta gde će slog biti upisan. Mesto zapisa novog sloga zavisi od vrste organizacije datoteke, što ćemo razmatrati uskoro.

Promena sadržaja sloga

Ponekad je potrebno izmeniti sadržaj jednog ili više polja u nekom slogu datoteke. Ta operacija je delikatna sa više aspekata. Poseban problem predstavlja promena onih polja koji sačinjavaju ključ datoteke, jer to može da izazove neželjene posledice (pojavljivanje slogova sa istim ključem, narušavanje uređenosti datoteke, i sl.). Sama operacija bi se promene sadržaja bi se mogla odvijati na sledeći način: učitati slog koji se menja, izvršiti zamenu polja, bisati slog iz datoteke, dodati sloga sa izmenjenim poljima u datoteku. Ovakav scenario se najčešće i primenjuje pri promeni (ažuriranju) slogova datoteke.

Organizcija datoteka

Postoji više različtih tipova logičke organizacije datoteka. Svaki od tih tipova ima svoje prednosti i nedostatke i svaki od njih je pogodniji od drugih za neke specifične primene. Pravilan izbor organizacije datoteka presudno utiče na ukupnu efikasnost obrade podataka.

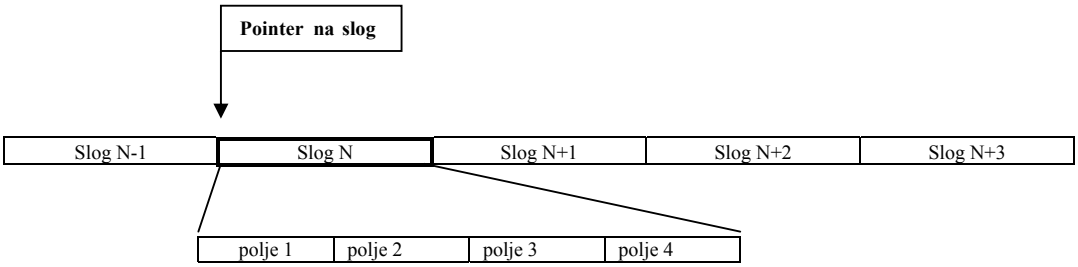
Slede četiri glavne organizacije datoteka koje će biti i ukratko opisane:

1) Datoteke sa serijskim pristupom (Serijske datoteke)
2) Datoteke sa sekvencijalnim pristupom (Sekvencijalne datoteke)
3) Datoteke sa direktnim pristupom (Direktne datoteke)
4) Datoteke sa indeks-sekvencijalnim pristupom (Indeks-sekvencijalne datoteke)

Moguće su i druge varijacije ovih osnovnih tipova datoteka, ali mi se njima nećemo baviti.

Serijske datoteke

Osnovna karakteristika serijskih datoteka je da se slogovi u datoteci upisuju jedan za drugim bez nekog logičkog redosleda (najčešće onako kako podaci pristižu u vremenskoj seriji). Takva organizacija se najčešće primenjuje kada se podaci prikupljaju sa raznih lokacija (izvora) tokom nekog vremenskog perioda. Oni se tada najčešće memorišu (upisuju u datoteku) sa namerom da budu obrađeni u nekom kasnijem trenutku (kada pristignu svi očekivani podaci). Sledeća slika ilustruje serijsku organozaciju.



Serijske datoteke se obično obrađuju tako što se slogovi čitaju redom od prvod do poslednjeg. One takođe mogu biti korišćene kao priprema za formiranje uređenih datoteka čiji opis sledi.

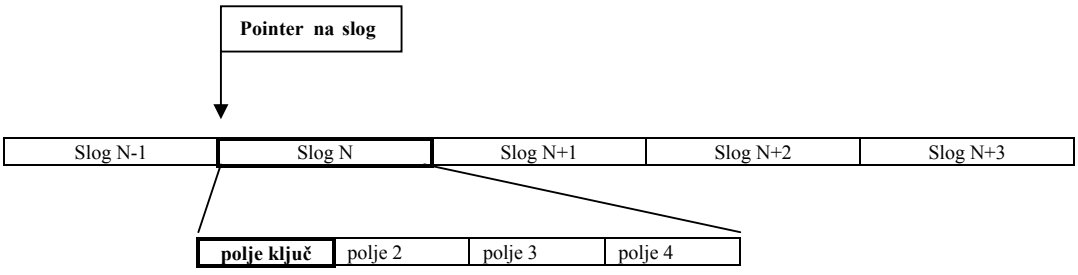
Sekvencijalne datoteke

Kada želimo da datoteka ima neki logičan redosled slogova onda nam sekvencijane datoteke pružaju jednu takvu mogućnost. Na primer, bankovne račune u datoteci možemo smeštati u rastućem redosledu broja računa, podatke o studentima možemo u slogovima koji su poređani u po azbučnom redosledu imena studenata, ili pak po rastućem redosledu broja indeksa.

Upravo na primeru studenata vidimo značaj ključa datoteke. Dok u školi možemo imati više studenata sa istim imenom i prezimenom, broj indeksa je jedinstven za svakog studenta i predstavlja ključ datoteke.

Sekvencijalne datoteke su jako efikasne za takozvanu beč (batch) obradu podataka, kao što je slučaj kod obračuna računa za komunalne usluge (struja, voda, itd.). Pri beč obradi se pristupa svim slogovima datoteke u redosledu koji je određen ključem, vrši se odgovarajuća obrada podataka iz sloga (obračun potrošnje vode ili stuje) i štampaju izveštaji (računi, fakture, statistički pokazatelji, itd.)

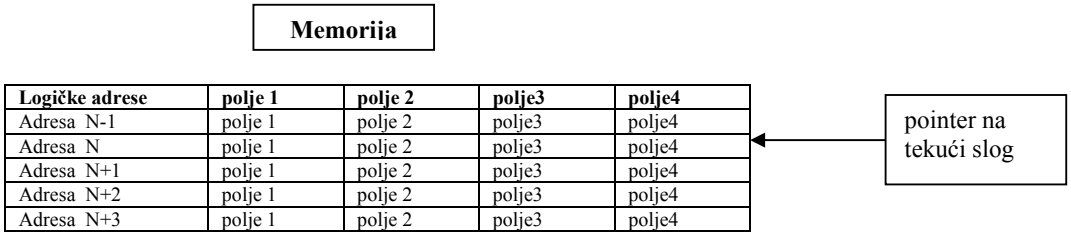
Sledeća slika ilustruje sekvencijalnu organizaciju podataka.



Sekvencijalna organizacija nije pogodna za pojedinačnu obradu slogova. Na primer, ako nam iz neke datoteke koja ima N slogova treba samo jedan slog da bi nad njim izvršili neku operaciju (brisanje, izmena podataka, obrada podataka iz sloga), onda je u proseku potrebno N/2 učitavanja slogova da bi učitali željeni slog. Za veliko N (npr. milion) to bi moglo da rezultira u veoma neefikasnom radu.

Direktne datoteke

Direktne datoteke upravo rešavaju problem individualnog pristupa slogovima. Kod direktne organizacije slogovima se pristupa direktno, bez čitanja predhodnika traženom slogu. To se postiže na taj način što se uspostavlja veza između ključa i logičke (fizičke) adrese u memoriji na kojoj se slog sa datim ključem nalazi. Ovavrsta organizacije može se ilustrovati sledecom slikom.



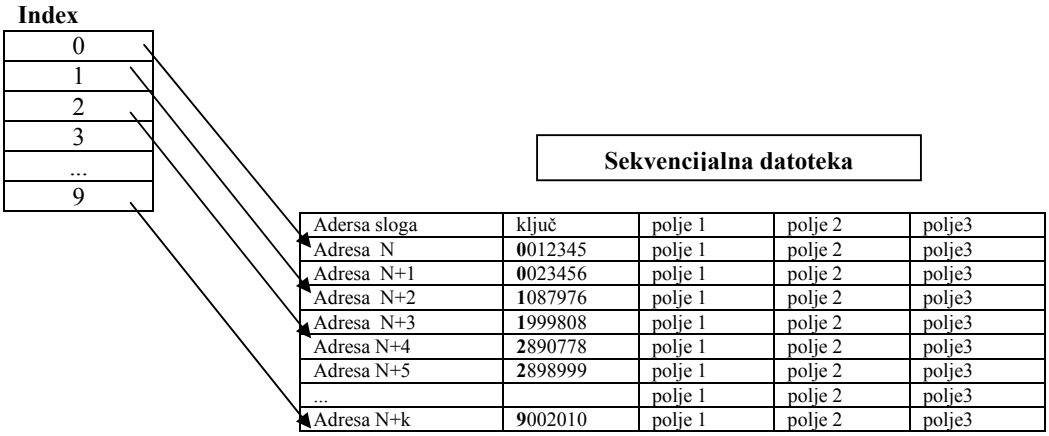
Uspostavljanje veze između ključa i adrese se može vršiti na više načina. Najčešće se koristi hešing metoda o kojoj je već bilo reči pri definisanju heš tabela u predhodnom predavanju. Direktna organizacija zahteva i da uređaj za memorisanje podataka ima mogućnost direktnog pristupa (diskovi imaju tu mogućnost, dok na trakama možemo čuvati samo serijske i sekvencijalne datoteke).

Za razliku od sekvencijalnih datoteka, gde su slogovi poređani u nekom logičkom redosledu, kod direktnih datoteka slogovi su rasuti po memoriji bez ikakvog logičkog reda (već onako kako transformacija ključa u adresu zahteva). Zato su nepogodne za beč obradu.

Indeks-sekvencijalne datoteke

Indeks sekvencijalne datoteke imaju za cilj da pomire prednosti i nedostatke sekvencijalnih i direktnih datoteka. Drugim rečima, kod indeks'sekvencijalne organizacije slogovima se može efikasno pristupati i sekvencijalno i direktno. U cilju postizanja takve organizacije uvode se pomoćne datoteke koje sadrže indekse za lakše pronalaženje slogova (setite se indeksa na kraju knjige koji olakšavaju pronalaženje karakterističnih reči koje se u klnizi pojavljuju).

Indeks-sekvancijana datoteka se zapravo sastoji od jedne sekvencijalne (po ključu uređenje datoteke) i jedne pomoćne datoteke indeksa, kao što to ilustruje sledeca slika.



Ovakvom organizacijom se obezbeđuje da se slogovima datoteke može pristupati sekvencijalno jer su slogovi upisani u sekvencijalnu datoteku u recimo rastućem reodsledu ključa (kao na slici). S druge strane, kad želimo da pristupimo jednom slogu sa određenim ključem, tada najpre pogledamo kojem indeksu tak ključ odgovara pa sledeći pointer iz indeksne datoteke pristupamo grupi slogova u kojoj se može naći slog sa traženim ključem. Unutar te grupe pristupamo slogovima redom, pa ili pronađemo traženi slog ili zaključimo da takav slog (sa zadatim ključem) nije u datoteci.

Datoteke su obično u novije vreme zapravo najčešće delovi baze podataka. Drugim rečima baze podataka se obično sastoje od više datoteka uređenih na jedan od predhodno opisanih načina. Zato i softver za podršku radu sa bazama podataka, tzv DBMS (database management software) kao što je Oracle, SQL ili neki drugi, u sebi sadrže sve potrebne elemente za kreiranje, brisanje, modifikaciju kao i obradu podataka u datotekama koje čine bazu. Programski jezici obično ne sadrže takve mogućnosti kao inherentne karakteristike jezika, već obično uz takve jezike idu bibiloteke potprograma (klasa, funkcija, procedura) koje programerima omogućavaju manipulaciju podacima u bazama podataka. To su tzv. ODBC (Open Database Connectivity) drajveri.

Kako su datoteke (u bazama podataka) osnovni resurs koji sadrži velike i značajne podatke u svakoj poslovnoj strukturi, to njihova organizacija, ažurnost i obrada značajno doprinose ukupnoj produktivnosti poslovnog sistema. Danas, kada poslovni sistemi prelaze ne samo granice zemalja već i kontinenata, nije redak slučaj da se datoteke jedne kompanije nalaze na više lokacija širom sveta i da obuhvataju velike količine informacija (reda tetra bajta).

Zbog značaja koje podaci imaju u poslovanju svake kompanije posebna pažnja se poklanja integritetu, sigurnosti i zaštiti podataka (datoteka).

Pod integritetom se podrazumeva sprečavanje grešake na unosu, grešake u proceduri, programskih grešaka, virusa, grešake u prenosu. To se postiže permanentnom validacijom i verifikacijom podataka kojom se utvrđuje integritet, kao i raznim metodama za uspostavljanje narušenog integriteta.

Sigurnost datoteka se bavi pitanjima slučajnih oštećenja, namernih oštećenja, ilegalanog pristupa i sl. Povećanje sigurnosti podataka postiže se različitim merama kao što su ograničenje pristupa podacima putem lozinki, strogo poštovanje procedura za promenu podataka, čuvanje kopija (back-up), firewall-ove, pa sve do fizičke zaštite pristupa prostoru i opremi na kojoj se čuvaju podaci.

Zaštita datoteka uključuje takođe i kriptografiju i antivirusne mere.

Detaljan opis nekih od najčešćih procedura kao što su sortiranje i merđyovanje datoteka, kao i primere obrade možete naći u knjizi »Projektovanje programa«, N.Marković, BPŠ 2001.

Pitanja

1. Definišite sledeće izraze povezane sa datotekama: Character, Field, Record, File.
2. Po čemu se datoteka razlikuje od drugi struktura podataka?
3. Šta je to slog?
4. Šta je polje?
5. Koje su četiri osnovne organizacije podataka?
6. Šta je serijska datoteka?
7. Šta je sekvencijalna datoteka?
8. Šta je ključ datoteke?
9. Koji tipovi datoteke se mogu organizovati na magnetnoj traci?
10. Šta je direktna datoteka?
11. Šta je indeks-sekvencijalna datoteka?
12. Koje se tipične operacije izvršavaju nad datotekama?
13. Kako se kreira datoteka?
14. Šta znači ažurirati datoteku?
15. Kakvi se problemi mogu javiti pri ažuriranju sekvencijalne datoteke?
16. Šta znači sortiranje datoteke?
17. Šta je merdžovanje datoteka?
18. Šta su master i transakcione datoteke?
19. Kako se može sprečavati neautorizovan pristup podacima?
20. Kako se može detektovati neautorizoban pristup podacima?
21. Kako se podaci mogu zaštititi od uništenja?
22. Kako kriptografija pomaže zaštitu podataka?
23. Kako virusi mogu uticati na podatke?

9. Algoritmi

Nakon ovog predavanja vi ćete biti u stanju da:

1. definišete termin *algoritam*
2. navedete osnovne karakteristike algoritama
3. opišete šta se podrazumeva pod *nizom* (sekvencom)
4. opišete *if ... then* i *if ... then ... else* konstrukciju
5. opišete *do...while I while*…do konstrukciju
6. navedete različite načine izražavanja algoritama
7. objasnite šta se podrazumeva pod terminom "varijabla"
8. objasnite šta se podrazumeva pod pojmom *tip varijable (data type)*
9. objasnite šta su to pravila za davanje imena varijablama
10. opišete strategiju projektovanja algoritama

Šta je to algoritam?

Poznati programerski istraživač i profesor Niklaus Wirth je dao ovakvu definiciju:

Programs = Algorithms + Data

Još u prvom predavanju vam je predstavljen pojam programa i podataka, ali šta je sad to algoritam?

Algoritam je osnovno rešenje problema na kome se zasniva program ili plan za izradu programa, ili se može reći da je algoritam:

"Efektivna procedura za rešavanje problema u konačnom broju koraka."

Efektivna zapravo znači da su koraci koje procedura definiše izvodljivi i jasno definisani. Veoma je važno da je broj tih koraka konačan tj. da se procedura odvija u konačnom vremenu. Dobro projektovan program mora uvek dati odgovor, taj odgovor nam se ne mora uvek svideti, ali on mora biti dat u konačnom vremenu. Ponekad se dešava da nama odgovora. To je u računarskoj teoriji poznato kao HALTING problem. Međutim , dobro projektovani programi moraju imati rešenje i za takve slučajeve, tj. moraju garantovati završetak u konačnom broju koraka.

Osnovne elementi (konstruktori) algoritama

Evo najpre jednog jednostavnog algoritma za spremanje čaja:

1. Ako u čajniku nema vode napunite čajnik vodom
2. Stavite čajnik na šporet i uključite odgovarajuću ringlu.
3. Ako šolja za čaj nije prazna ispraznite je.
4. Stavite lišće čaja u šolju za čaj.

5. Ako voda u čajniku nije provrela idite na korak 5, ako jeste idite na korak 6.
6. Isključite ringlu.
7. Sipajte vodu iz čajnika u šolju (pazite da ne prelijete).

Zapažamo da ovaj algoritam ima više koraka, da neki od koraka (1,3 i 5) sadrže donošenje odluka, da jedan korak (5) sadrži ponavljanje u kome se izvršava proces čekanja na vodu da provri.

Algoritam sadrži tri elementa:

1. Sekvencu – niz (proces)
2. Odlučivanje (selekcija)
3. Ponavljanje (repeticija, iteracija, ciklus, petlja)

Godine 1964 matematičari Corrado Bohm i Guiseppe Jacopini pokazali su da se svaki algoritam može izraziti pomoću sekvence, odluke i ponavljanja. To je poznato kao **teorema o programskoj strukturi** i predstavljalo je veoma važan korak ka strukturalnom programiranju koje je danas u upotrebi.

Sekvenca (niz operacija)

Sekvenca znači da se svaki korak sekvence izvršava u unapred datom redosledu – onako kako se pojavljuju u sekvenci, jedan za drugim. U predhodnom algoritmu svaki korak mora biti urađen u zadatom redosledu. Ako bi promenili redosled koraka to bi moglo da dovede do pogrešnih (ponekad i tragikomičnih) rezultata.

Odluka (Selekcija)

U algoritmima rezultat odluke je ili *tačno* ili *netačno*, nema ništa između. Rezultat odluke se bazira na nekoj tvrdnji (logičkom iskazu) koja može da ima vrednost tačno ili netačno, na primer:

if danas je sreda then uzmi platu

je odluka koja ima sledeći oblika:

if tvrdnja then proces

Tvrdnja (iskaz) je neka rečenica koja može biti tačna ili netačna, tako je tačno da danas je sreda ili je netačno da danas je sreda. To ne može biti istovremeno i tačno i netačno, niti nešto treće. Ako je tvrdnja tačna tada se izvršava proces koji sledi iza reči *then*. Ako tvrdnja nije tačna prelazi se na sledeću instrukciju bez izvršavanja procesa posle reči *then*.

Odluka može da ima i nešto složeniji oblik:

if *tvrdnja*
then *proces1*
else *proces2*

to je oblik *if ... then ... else* .... Ovo znači da ako je tvrdnja tačna tada se izvršava proces1 a ako je netačna proces2.

U onom prvom obliku odluke *if* *tvrdnja* *then* *proces* u else delu nema procesa pa zato ni else nije potrebno.

Ponavljanje (Iteracija)

Ponavljanje ima dva oblika: do…while ciklus i while..do ciklus.

Repeat ciklus se koristi za ponavljanje procesa ili niza procesa sve dok tvrdnja iz uslova ponavljanja ne postane tačna. Repeat ciklus ima sldeći oblik:

do
Proces1
Proces2
.....
ProcesN

while *tvrdnja*

Evo jednog primera

do
Sipaj vodu u čajnik
while *čajnik nije pun*

Proces je *Sipaj vodu u čajnik*, a uslov za nastavak ciklusa *čajnik nijeje pun*.

do…while ciklus obavi (bar jednom) proces pre testiranja da li je uslov završetka ciklusa ispinjen. Šta će se desiti u predhodnom primeru ako je čajnik već bio pun kada je ciklus započet? Doći će do neželjenog procesa koji će izazvati prelivanje vode iz čajnika.

Za takve slučajeve pogodnije je koristiti while…do ciklus:

while *čajnik nije pun*
Sipaj vodu u čajnik

Pošto se odluka da li je čajnik pun ili ne donosi pre sipanja vode, mogućnost prelivanja je eliminisana.

Različiti načini izražavanja algoritama

Jedan način izražavanja algoritma smo videli maločas – taj način ćemo zvati forma deskriptivnih koraka. Tokom predavanja proučićemo četiri različite forme izražavanja algoritama:

1. Koračna-forma
2. Pseudokod
3. Dijagram toka
4. Nassi-Schneiderman (NS) forma
5. Jackson-ovi strukturni dijagrami (JSD) forma

Prve dve su pisane forma. Primer pravljenja čaja je tipičan primer Koračne forme (Step-Form) gde smo za izražavanje algoritma koristili prirodan jezik. Problem sa prirodnim jezikom je u tome što ponekad može biti neprecizan. Tako se može desiti da ono što jedan čovek napiše drugi pročita sa sasvim drugim tumačenjem. Pseudokod je takođe vrlo sličan prirodnom jeziku ali mnogo precizniji i sa ograničenim rečnikom.

Posldenja tri načina izražavanja algoritama su grafički, to jest u njima se koristi mešavina grafičkih simbola i pisanih reči da se predstave sekvenca, odluka i ponavljanje.

U sledećem predavanju ćemo se baviti ovim različitim načinima izražavanja algoritama, a sada ćemo najpre proučiti dve važne teme.

Šta su to varijable?

Pošto je *Programs = Algorithms + Data* vraćamo se na temu podataka. Kako smo već rekli podatak je simbolički prikazana vrednost koja u programskom kontekstu dobija i značenje – znači u programu se podatak transformiše u informaciju. Pitanje glasi: Kako se podaci predstavljaju u programima?

Skoro svaki program sadrži podatke, a podaci se obično "sadrže" u varijablama. Tako varijablu možemo shvatiti kao kontejner za vrednosti koje mogu da se menaju tokom izvršavanja programa. Na primer, u našem primeru za pravljenje čaja nivo vode u čajniku je jedna varijabla, temperatura vode je varijabla, a i količina lišća je, takođe, varijabla.

Svakoj varijabli u programu se daje posebno ime, na primer:

- Nivo_Vode
- Temperatura_Vode
- Količina_Listova_Čaja

i u svakom datom trenutku vrednost koja je predstavljena varijablom Nivo_Vode može biti različita od vrednosti iste varijable u nekom drugom trenutku. Instrukcija:

- *If čajnik ne sadrži vodu then napuni čajnik*

može biti napisana i ovako:

- *If Nivo_Vode je 0 then napuni čajnik*

ili

- *If Nivo_Vode = 0 then napuni čajnik*

U nekom trenutku Nivo_Vode će dostići dozvoljeni maksimum, ma koliki on bio, i tada je čajnik pun.

Varijable i tipovi podataka

Podaci koji se koriste u algoritmima mogu biti različitog tipa. Najprostiju tipovi podataka su:

- numerički podaci kao što su 12, 11.45, 901.
- alfabetski (slovni) podaci (karakteri) kao što su 'A', 'Z' ili 'Ovo je alfabetski niz'.
- logički podaci sa TRUE, FALSE vrednostima.

Davanje imena varijablama

Uvek treba da se trudite da varijablama u algoritmu date smislena imena što će algoritam (i program) učiniti čitljivijim i razumljivijim. To je posebno važno kod velikih i kompleksnih programa.

U algoritmu za pravljenje čaja koristili smo prirodan jezik. Pokazaćemo kako se mogu koristiti imena varijabli za varijable tog algoritma. U desnoj koloni smo izabrali imena varijabli koja iako kraća od originalnih ne umanjuju značenje. Donja crta u nazivu varijable treba da označi da sve reči predstavljaju jednu jedinstvenu celinu kojom se predstavlja data varijabla.

1. Ako u čajniku nema vode napunite čajnik vodom	1. If *čajnik_prazan* then napuni čajnik
2. Stavite čajnik na šporet i uključite odgovarajuću ringlu	2. Stavite čajnik na šporet i uključite odgovarajuću ringlu
3. Ako šolja za čaj nije prazna ispraznite je.	3. If *šolja_nije_prazna* then isprazni šolju
4. Stavite lišće čaja u šolju za čaj.	4. Stavite lišće čaja u šolju za čaj.
5. Ako voda u čajniku nije provrela idi na korak 5	5. If *voda_ne_vri* then idi na korak 5
6. Isključite ringlu	6. Isključite ringlu

7. Sipajte vodu iz čajnika u šolju	7. Sipajte vodu iz čajnika u šolju

Ne postoje neka stroga pravila kao treba davati imena varijablama, ali postoje određene konvencije i preporuke. Dobro je da se usvojite neku od tih preporuka i da je onda dosledno koristite.

Uz put slična preporuka se može dati i za davanje imena procesima, takođe. To sve čini vaš program čitljivijim i razumljivijim, a time pogodnijim za održavanje i dalje unapređivanja - što znači da pomaže produžetku životnog ciklusa programa.

Strategija projektovanja algoritama

Sada kada znamo ponešto o algoritmima u stanju smo da razmatramo i strategiju za projektovanje algoritama. Evo jedne takve strategije koja može da bude korisna:

Korak 1: Istraživački korak

1. Identiifikovati procese
2. Identifikovati glavne odluke
3. Identifikovati ponavljanja
4. Identifikovati varijable

Korak 2: Izrada preliminarnog (grubog) algoritma

1. Izrada algoritma “višeg nivoa”
2. Proći kroz algoritam misaonom simulacijom. Ako simulacija otkrije probleme ispraviti algoritam.

Korak 3: Izrada finalnog (detaljnog) algoritma

1. Do detalja razraditi “grubi” algoritam napravljen u koraku 2.
2. Grupišite procese koji se mogu grupisati
3. Grupišite varijable koje se mogu grupisati
4. Testirajte algoritam simulacijom korak po korak.

Ovu strategiju ćemo stalno koristiti, ali za momenat ćemo se baviti samo opisom navedenih koraka.

Prvi korak – Istraživanje – zahteva pažljivo proučavanje definicije problema koji se rešava, kao i detaljnu analizu termina i pojmova koji se u toj definiciji koriste. U primeru sa pravljenjem čaja uočavamo niz procesa – punjenje čajnika, stavljanje na ringlu i tako dalje. Tu su takođe odluke i varijable.

Drugi korak – Preliminarni algoritam – je prvi pokušaj rešavanja problema. taj prvi pokušaj može biti i veoma grub, ponekad i nepravilan, ali će korak 2.2 otkriti sve nedostatke i pomoći da se dođe do korektnog rešenja.

Treći korak – Finalizacija – je i najteži, zahteva iskustvo, strpljenje i preciznost. takođe je za uspešnu realizaciju ovog koraka potrebno vladanje nekim programerskim veštinama koje se stiču samo dugotrajnim praktikovanjem – vežbanjem. U pragmatičnom delu ovog predmeta, kada se izučava specifičan programski jezik (Visual Basic ili C++) takva praktična znanja i veštine biće razvijene do novoa samostalnog razvoja programa od ideje do gotovog izvršnog programa.

Sada ćemo analizirati jedan nešto složeniji problem i algoritam kojim se on rešava.

Problem stabilnih parova

Pretpostavimo da u jednom gradu na jugu Srbije imamo n mladića i n devojaka. Svi se moeđusobno poznaju, jer to je jedno malo mesto. Svaki mladić ima rang listu svih devojaka iz tog gradića, tako da se na toj listi na prvom mestu nalazi deveojka koja mu se najviše sviđa, a na poslednjem n-tom mestu devojka koja mu se najmanje sviđa. Istu takvu listu imaju i sve devojke, ali sa rang listom svih n mladića. Te liste su nam unapred poznate na neki način (Radio Milevom).

Cilj nam je da oženimo momke i devojke tako da svi budu sretni, a da brakovi budu stabilni. Videćemo uskoro šta pod tim podrazumevamo.

Definicija. Skup brakova je stabilan ako ne postoji par mladić-devojka koji se jedno drugom sviđaju više nego njihovi bračni drugovi.

Na primer, pretpostavimo da su Marko i Jovana u jednom, a Petar i Milica u drugom braku. Na nesreću Jovani se više sviđa Petar nego Marko, a Petru se sviđa više Jovana nego Milica. Ovo znači da bi Jovana i Petar bili srećniji da su zajedno. To može da bude razlog za bračnu prevaru, pa se njihovi brakovi mogu smatrati nestabilnim.

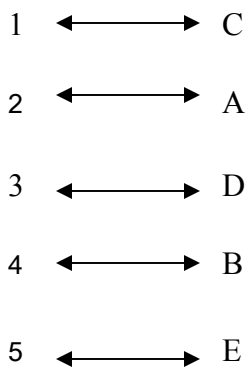
Matematički se može dokazati da uvek postoji stabilno uparivanje mladića i devojaka, takvo da ne postoji ni jedan nestabilan par. (Takođe se može matematički dokazati da ne postoji stabilno uparivanje ako bi dozvolili da se bračno uparuju osobe istog pola.).

Veoma je interesantno da ovaj algoritam u SAD-u ima jednu veoma praktičnu primenu. Tamo se svake godine svršeni studenti medicine upućuju (mladići) na praksu u bolnice (devojke) upravo korišćenjem ovog algoritma. Inače, algoritam su prvi put opisali D. Gale i L.S. Shapley 1962 godine.

Hajdemo sada da problem razmotrimo na jednom konkretnom slučaju 5 devojaka i 5 mladića. Mladiće ćemo označiti brojevima 1-5, a devojke slovim A-E. Sledeća tabela pokazule njihove rang liste.

mladići	devojke
1: CBEAD	A : 35214
2: ABECD	B : 52143
3: DCBAE	C : 43512
4: ACDBE	D : 12345
5: ABDEC	E : 23415

Pokušajmo najpre jednu prostu (grabljivu) strategiju. Krenimo redom od prvog mladića i svima dajmo devojku koja je najviše kotirana na njegovoj listi, a još uvek je "raspoloživa". To bi nam dalo sledeći rezultat uparivanja:



Da proverimo stabilnost ovako napravljenih parova. Mladići 1, 2 i 3 su dobili svoje omiljene devojke, tako da im neće pasti na pamet da jure za drugima. Mladić 4 može biti problematičan jer mu se više sviđa devojka A od one koju je dobio (B), ali je devojka A njega stavila na poslednje mesto, tako da to nije problem. Međutim mladiću 4 se više sviđa i devojka C one koju je dobio, a devojci C se on takođe više sviđa od onog kojeg je ona dobila (1). Znači mladić 4 i devojka C su potencijalni brakolomci. Uparivanje je, dakle, nestabilno. Mogli bi sada da pokušamo da nekom kombinatorikom to popravimo, ali bi pri tom mogli da formiramo druge nestabilne parove, i da vrteći se u krug ne nađemo rešenje. I to za samo 5 parova. A, šta bi bilo da je 100-tinak (ili više) parova u pitanju. Izgleda kao nemoguć zadatak. Ali, generalno rešenje postoji i to veoma elegantno.

Algoritam

Sada ćemo opisati jedan algoritam koji se odvija u nekoliko dana i koji dovodi do stabilnog uparivanja.

Svakod dana se ponavlja sledeći ritual (scenario):

Jutro: Svaka devojka izađe na svoj balkon. Svaki mladić dođe ispod balkona devojke koja je na prvom mestu njegove rang liste i peva joj serenadu. Ako mladiću nije ostala ni jedna devojka u listi, on ostaje kod kuće (i radi domaće zadatke iz programiranja).

Popodne: Svaka devojka koja ispod svog balkona ima udvarače (koji pevaju serenade), jednom od njih koji je najviši na njenoj rang listi kaže: "Možda, dođi sutra.", a svim ostalima kaže: "Nikad se neću udati za tebe. Šetaj".

Uveče: Svaki mladić koji je dobio "korpu" iz svoje liste briše devojku kod koje nema nikakve šanse.

Kada posle nekoliko dana ujutro ispod svakogbalkona bude samo po jedan mladić, devojke uzimaju te mladiće i tako se formiraju parovi. Tako formirani parovi biće stabilni u skaldu sa našom definicijom stabilnosti. Tu je i kraj našeg algoritma. Algoritam sadrži sve tri vrste koraka: sekvencu, selekciju i repeticiju.

Siže predavanja

U ovom predavanju ste saznali o:

- algoritmima i nihovim osnovnim elementima – sekvenci, odluci i ponavljanju
- različitim načinima prikaza algoritama
- varijablama, tipovima varijabli i konvencijama za davanje imena varijablama
- strategiji projektovanja algoritama

Algoritam je zapravo plan (scenario) kako će problem biti rešen, a skoro svi algoritmi imaju iste osobine i sastavljeni su od istih elemenata (teorema o strukturi). Postoji više načina za izražavanje (prikazivanje) algoritama, a neki od tih načina su ovde pomenuti. Svaki algoritam koristi podatke koji se mogu menjati tokom rada algoritma. Za projektovanje (izradu) algoritma dobro je imati strategiju. Jedna moguća strategija prikazana je u ovom predavanju.

Sada ste već spremni za projektovanje programa, ali pre toga u narednom predavanju bavićemo se jednom temom koja projektovanje programa stavlja u širi kontekst projektovanja softverskih sistema i softversko inženjerstvo uopšte.

Pitanja

1. Definišite pojam *algoritam*.
2. Šta se podrazumeva kad kažemo da je neki algoritam *efektivan*?
3. Kakvo je značenje reči *konačan* u vezi sa algoritmima?
4. Navedite tri osnovna elementa algoritama.
5. Koja su to dva konstruktora odluke?
6. Šta je to tvrdnja?

7. Koja su to dva konstruktora ponavljanja?
8. Navedite četiri načina prikazivanja algoritama.
9. Šta je to varijabla?
10. Dajte neki primer varijable iz svakodnevnog života.
11. Šta je to tip podatka?
12. Navedite tri osnovna tipa podatka koji se mogu pojaviti u algoritmima.
13. U predavanju je prikazana jedna strategija za projektovanje algoritama. Navedite osnovne korake te strategije. Možete li napraviti neku svoju strategiju - opišite.

10 . Razvoj algoritama

Prevalili smo polovinu semestra i vreme je da napravimo malu rekapitulaciju pređenog i preostalog puta.

Podaci

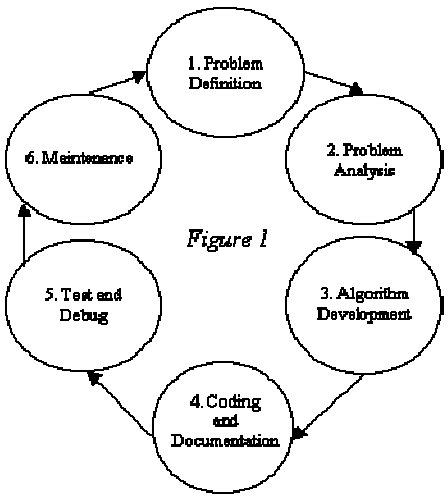
Iscrpno smo se bavili podacima, analizirajući različite vidove u kojima se oni javljaju (tekstualni, numerički, zvučni, grafički, video, itd.), analizirajući načine na koji se podaci memorišu u računarima (bitovi, bajtovi itd.), tipove podataka (int, char, float, pointer, itd...) i strukture podataka (nizovi, liste, stekovi, redovi, grafovi, heš tabele, datoteke), što sve zajedno čini osnovu za kompleksnu primene računara za rešavanje različitih problema u raznim domenima primene (tehnici, biznisu, medicini, upravi, itd...)

Životni ciklus softvera - Software Development Life Cycle (SDLC)

Razvoj i proizvodnja softvera je jedna od danas najatraktivnijih poslovnih sfera. Setite se samo da je vlasnik Microsofta dugo godina prvi na listi najbogatijih ljudi na svetu, a njegova kompanija prodaje softverske proizvode (Windows operativne sisteme i drugo) u stotinama miliona primeraka širom sveta.

Razvoj i proizvodnja softvera je složen zadatak, koji zahteva visoko kvalifikovane ljude, vrhunsku organizaciju posla, vrhunski marketing i menadžment.

Mi ćemo se detaljnije bavit fazama kroz koje svaki softverski proizvod prolazi od ideje o proizvodu do njegove realizacije, upotrebe, zastarevanja i nestajanja. Sledeća slika ilustruje taj ciklus.



Slika 10.1 Životni ciklus softvera

Svaka od faza zahteva posebnu metodologiju, posebno obučene ljude za tu fazu i posebne tehnike rada.

Naše težište biće na Fazi 3 – Razvoju algoritama jer to i jeste naša osnovna tema. Međutim moraćemo se, bar delimično, baviti i svim drugim fazama.

Pojam algoritma (programa)

Podsetićemo se na neke osnovne pojmove koje smo definisali u poglavlju 9.

Dali smo jednu intuitivnu definiciju algoritma (Efektivna procedura kojok se rešava neki problem u konačnom broju koraka). Veza između programa, algoritma i podataka najbolje se odslikava formulom (Niklaus Wirth, autor jezika Pascal):

Program = Algoritam + Podaci

gde se pod programom podrazumeva algoritam izražen u nekom konkretnom programskom jeziku (C, C++, C#, Java, Cobol, Basic, itd.).

Teorema o programskoj strukturi

Analizirajući razne algoritme Bohm i Jacopini su uočili da se svaki algoritam može izraziti sa sledeće tri vrste koraka.

1. Sekvenca - proces
2. Odluka - selekcija
3. Ponavljanje - repeticija, iteracija, ciklus, petlja

O ovim osnovnim algoritamskim strukturama bilo je reči u poglavlju 9. Ovde se samo podsećamo na njih jer one predstavljaju korake koji se mogu naći u svakom algoritmu.

Izložićemo sada jednu opštu metodologiju (strategiju) za razvoj algoritama. Naravno, ovo nije jedina metodologija, ali se pomoću nje jasno ilustruje proces razvoja algoritama.

Strategija razvoja algoritama

Korak 1: Istraživački korak
- Identiifikovati procese
- Identifikovati glavne odluke
- Identifikovati ponavljanja
- Identifikovati varijable

Korak 2: Izrada preliminarnog (grubog) algoritma
- Izrada algoritma "višeg nivoa"
- Proći kroz algoritam misaonom simulacijom. Ako simulacija otkrije probleme ispraviti algoritam.

Korak 3: Izrada finalnog (detaljnog) algoritma
- Do detalja razraditi "grubi" algoritam napravljen u koraku 2.
- Grupišite procese koji se mogu grupisati
- Grupišite varijable koje se mogu grupisati
- Testirajte algoritam simulacijom korak po korak.

Metode prikazivanja algoritama

Sada ćemo se baviti sa nekoliko najrasprostranjenijih tehnika (načina) za prikazivanje algoritama. Jedan od načina smo već koristili u primeru »Stabilni parovi«. Reč je o opisnom prikazu algoritma korišćenjem prirodnog (srpskog, engleskog ili nekog drugog) jezika.

Sledi lista tehnika koje ćemo proučiti:

- Prirodni jezik (koračna forma)
- Dijagrami toka (Flowcharts)
- Pseudo kod (Pseudo cod)
- Nasi-Šnajderman dijagrami (Nassi-Sneiderman – N/S diagrams)
- Džeksonovi Strukturalni dijagrami (Jackson Structured Diagrams – JSD)

Za svaku od navedenih tehnika, pored opisa, daćemo i kratak komentar o uzajamnim prednostima i nedostacima.

Prirodni jezik

Ovom tehnikom se algoritam prikazuje kao niz brojem označenih koraka. Svaki korak se sadrži jednu ili više rečenica prirodnog kezika (srpskog, na primer) kojim se opisuje proces (operacija) koju u tom koraku treba izvršiti.

Evo jednog jednostavnog primera:

Problem: Prikazati (na monitoru računara, na primer) dvostruku vrednost broja koji je predhodno unet u računar (pomoću tastature, na primer).

Algoritam u priprodnom jeziku:

1. Tražiti od korisnika da uz pomoć tastature unese broj.
2. Učitati broj koji korisnik ukuca na tastaturi.
3. Pomnožiti učitani broj sa brojem 2.
4. Prikazati rezultat operacije iz koraka 3 na monitoru računara.

Sekvenca se prikazuje jednostavno ninizanjem koraka jedan za drugim, pri ;emu svaki korak dobija redni broj u redsledu kojim treba da se iyvršavaju.

Odluka se pprikazuje opisom uslova i uputstvom na koji korak se ide za slučaj da je uslov ispunjen, a na koji korak kada uslov nije ispunjen. Na sličan način možemo izvršiti i selekciju iz više mogućih rezultata nekog uslova.

Repeticija – iteracija se postiže tako što se izvršenje nastavlja nekim korakom koji ima manji redni broj od onog u kojem se postavlja uslov repeticije.

Prednisti prirodnog jezika:

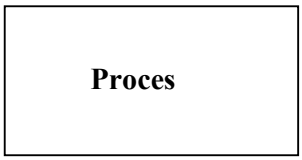
- Jednostavan za učenje, jer se ionako služimo prirodnim jezikom.

Nedostaci:

- Koraci su predugački jer se mora koristitu puno reči za njihov opis.
- Prevođenje iz prirodnog jezika u kompjuterski jezik može biti teško jer za razliku od prirodnih jezika, kompjuterski (programski) jezici imaju vrlo precizno definisanu sintaksu (gramatiku) i semantiku (značenje).

Dijagrami toka (Flowcharts)

Ova tehnika koristi niz grafičkih simbola povezanih usmerenim linijama (strelicama) kojima se pokazuje sekvenca (niz) u kojoj će koraci opisani grafičkim simbolima biti izvršavani. Unutar grafičkog simbola upisuje se prirodnim jezikom (ili pseudo kodom) proces (operacija) koju treba izvršiti. Grafički simboli koji se najčešće koriste za prikaza algoritama dijagramom toka su sledeći:

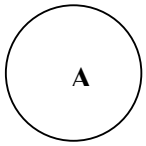


Pravouganik – sadrži opis procesa (naredbi) koje se izvršavaju jedna za drugom u redosledu kako su napisane. Nakon što se naredbe (jedna ili više) izvrše nastavlja se sa sledećim grafičkim simbolom koji sledi. Dakle, pravougaonik odgovara programskoj sekvenci.

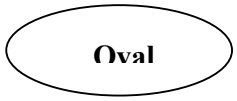
Romb – predstavlja ili ulaznu ili izlaznu naredbu kojim se podaci unose u kompjutersku memoriju (ulaz) ili iz kompjuterske memorije prikazuju na nekom spoljnjem uređaju (izlaz). Nakon što se naredba ulaza (izlaza) izvrši, algoritam se nastavlja naredbom iz sledećeg simbola.

Dijamant – predstavlja proces donošenja odluke. Odluka sadrži pitanje koje obično ima dva odgovora, DA ili NE, pa se nakon utvrđivanja koji od tih odgovora je tačan algoritam nastavlja jednom od dve moguće putanje koje slede (izlaze) iz ovog simbola.

Strelica – služi za povezivanje grafičkih simbola u smeru njihog logički sukcesivnog izvršavanja.

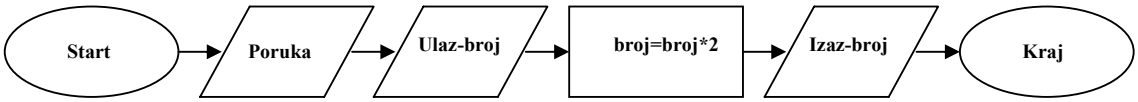


Krug – koristi se za povezivanje delova dijagranma toka. Kada dijagram toka ne može da bude prikazan na jednoj stranici (što je čest slučaj) ovaj znak se koristi za povezivanje delova dijagrama koji se nalaze na različitim stranicama.



Oval (elipsa) – se koristi za označavanje početka i kraja algoritma. Oval za start pokazuje gde algoritam započinje i obično sadrži reč “Start”, a oval za kraj pokazuje gde se algoritam završava i obično sadrći reč “Kraj”.

Evo kako bi algoritam iz predhodnog primera (prikaz dvostruke vrednosti učitanog broja) bio prikazan ovom tehnikom:



Kako se prikazuje repeticija dijagramima toka?

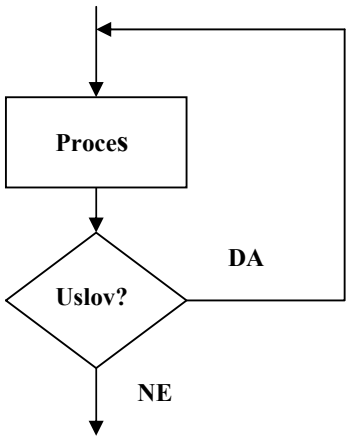
Kako smo već videli dijagrami toka imaju simbole za sekvencu (pravougaonik) i odluku (dijamant). Ali, kao prikazati repeticiju (do...until i while tipa na primer).

To se postiže kombinacijom simbola za sekvencu i odluku.

Tako se repeticija tipa:

do {
proces;
}**while**(uslov)

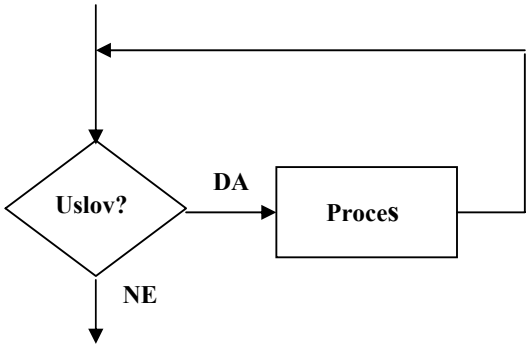
postiže sledećim dijagramom:



Slično tome repeticija:

while (uslov) {
proces;
}

može da se izrazi dijagramom toka na sledeći način:



Prednosti dijagrama toka:

- Grafička prezentacija algoritma olakšava pronalaženje logičkih grešaka u algoritmu ("slika vredi hiljadu reči" – stara kineska poslovica).
- Postojanje znaka za povezivanje (krug) omogućava veoma lako dodavanje novig delova algoritma.

- Algoritam prikazan grafički lakše se prevodi u programski kod nego što je to slučaj sa prirodnim jezikom ili pseudokodom.

Nedostaci:

- Treba upamtiti značenje grafičkih simbola.
- Kada postoji mnogo koraka odluke i ponavljanja, dijagram toka može da se pretvori u vrlo zamršenu mrežu iz koje je teško sačiniti valjan programski kod.

Pseudo kod

Ova tehnika je veoma slična prirodnom jeziku s tom razlikom što se umesto prirodnog jezika koristi neki drugi jezik (sličan prirodnom) koji ima precizniju sintaksu, koristi manji broj unapred zadatih reči, pa je time lakše definisati i semantiku (značenje) rečenica koje se formiraju u tom jeziku. Takav jezik često podseća i na programske jezike, to jest on predstavlja kompromis između prirodnog i programskog jezika. Kompromis u smislu da je dovoljno jednostavan i razumljiv za čoveka, a istovremeno pogodan za dalje transformisanje algoritma u program. Rcimo takav jezik može sadržati reči kao što su: display (za prikaz poruke na monitoru), read (za učitavanje podatka), kao i simbole +,-,*,/,= za korišćenje u matematičkim formulama itd.

Tako naš predhodni primer algoritma može pseudo kodom biti prikazan kako sledi:

1. **display** poruka
2. **read** broj
3. rezultat = broj*2
4. **display** rezultat

Prednosti:

- Jednsotavan za učenje skoro kao i kod prirodnog jezika.
- Lakši za prevođenje u programski jezik, jer kao što je rečeno jako podseća na stvarne programske jezike.

Nedostaci:

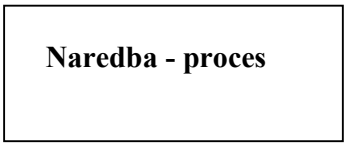
- Ova tehnika se oslanja na poznavanje takozvanih imperativnih (proceduralnih) jezika, pa za one koji se prvi put sreću sa ovom vrstom jezika može biti malo zbunjujuće – meša se prirodni i simbolički jezik.

Nasi-Šnajdermanovi dijagrami (N/S dijagrami)

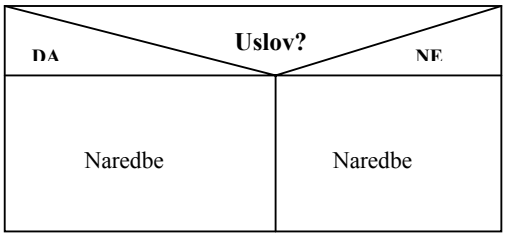
I kod Nasi-Šnajdermanovih algoritama se koriste grafički simboli kao i kod dijagrama toka, ali se ovde čitav algoritam stavlja u jedan jedini pragougaonik (boks). Simboli (koraci algoritma) se izvršavaju počev od prvog simbola na vrhu boksa i nastavljaju redom do poslednjeg na prikazanog dnu boksa. Svaki simbol sadrži ili prirodnim jezikom ili pseudokodom prikazane naredbe (procese).

Postoje tri vrste simbola: za sekvencu, za odluku i za repeticiju, u skladu sa teoremom o programskoj strukturi.

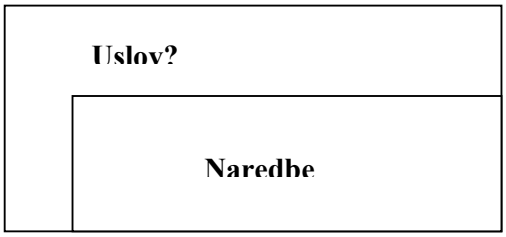
Simboli su:



Pravougaonik – koristi se za naredbe koje se izvršavaju jedna za drugom (sekvenca). Kada se sve naredbe iz pravouganika izvrše algoritam se nastavlja sledećim N/S simbolom.

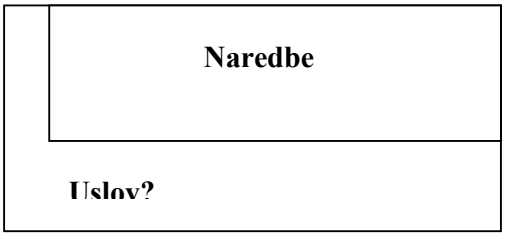


Simbol za odluku se sastoji od dela u kojem se nalazi uslov i dva prevougaonika koji sadrže alternativne naredbe ako je uslov ispunjen (DA) i ako nije (NE). Kada se izvrši odgovarajuća naredba algoritam se nastavlja sledećim N/S simbolom.



Repeticija se sastoji od ulsova i naredbe (ili niza naredbi) koje se izvršavaju sve dok je uslov ispunjen (while ciklus). Kada uslov nije ispunjen algoritam nastavlja rad sledećim N/S simbolom.

Repeticija za slučaj do...while ciklusa izgleda ovako:



U ovom slučaju, za razliku od predhodnog, Naredbe će biti izvršene pre testiranja da li je uslov ispunjen. Ako jeste ponavlhja se iyvršavanje Naredbi, a ako nije algoritam nastavlja rad sledećim N/S simbolom.

Evo kako bi izgledao naš standardni primer korišćenjem N/S dijagrama:

display poruka
read broj
rezultat=broj*2
display rezultat

Prednosti N/S dijagrama:

- Grafička prezentacija algoritma olakšava pronalaženje logičkih grešaka u algoritmu (kao kod dijagrama toka).
- Lakše se prevodi u programski kod nego dijagram toka. Tri programske strukture (sekvenca, odluka i repeticija) zastupljene su u svim programskim jezicima.
- Pošto nema sterlica kao kod dijagrama toka ne mogu se kreirati zamršene strukture, već algoritam "glatko" sledi logiku rešenja.

Nedostaci:

- Moraju se pamtiti grafički simboli koji predstavljaju sekvencu, odluku i repeticije.
- Otežano je umetanje novih koraka u već sačinjeni algoritam, a što je bilo lako kod dijagrama toka (korišćenjem simbola za konekciju – kruga). S drude strane nastavljanje algoritma na sledećoj strani je jednostavno – nacrtate novi boks a stranu obeležite brojem 2, itd.

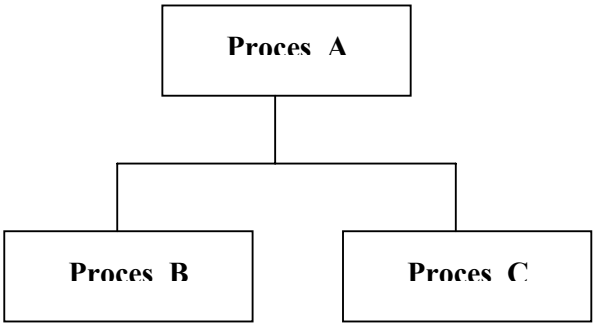
Džeksonovi strukturalni dijagrami (Jackson Structured Diagrams) – JSD dijagrami

Džeksonovi dijagrami slede ideju o podeli problema na niz podproblema manje složenosti (strategija poznata kao "devide and conquer" – podeli pa vladaj).

Ali oni takođe zadovoljavaju i teoremu o programskoj strukturi, to jest JSD dijagrami imaju grafički simbol za sve tri osnovne komponente algoritma – sekvencu, odluku i repeticiju.

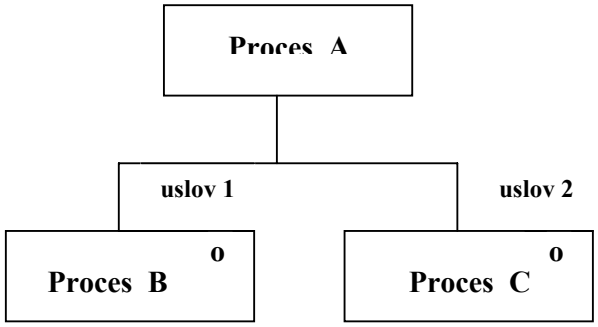
Sledi opis načina na koji se prikazuju ove komponente.

a) Sekvenca



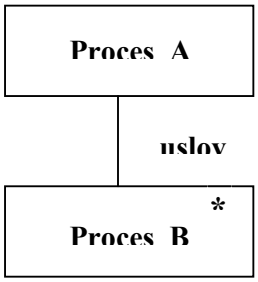
Ovde je proces A složen proces koji se sastoji od prostih procesa B i C. Da bi se obavio proces A potrebno je najpre obaviti proces B a onda proces C. Podprocesi se , dakle, izvršavaju sleva udesno jedan za drugim. Naravno proces A je mogao biti jedan jednostavan proces koji ne zahteva dalju podelu.

b) Odluka (selekcija)



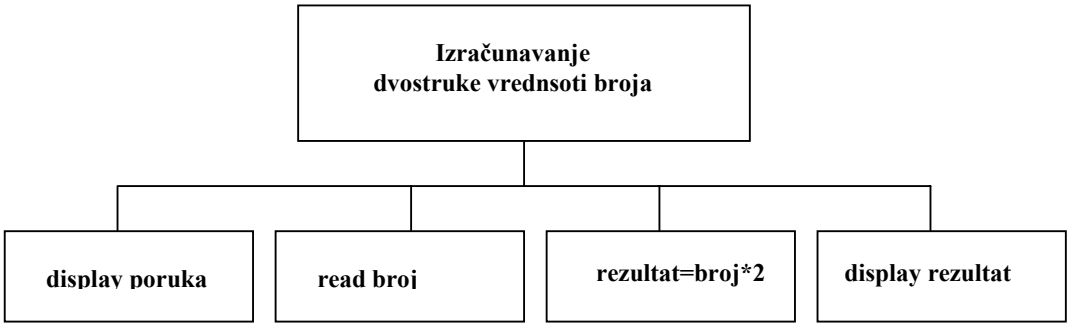
Odluka se označava tako što se proces A deli na procese B i C, ali uslovno, tako da se proces A ispunjava bilo izvršavanjem procesa B (kada je uslov 1 ispunjen) bilo izvršavanjem procesa C (kada je uslov 2 ispunjen). Uočite mali kružić u gornjem desnom uglu pravougaonika procesa B i C. Ti se kružići koriste da se pravougaonik odluke razlikuje od pravougaonika sekvence.

c) Repeticija



Repeticija – iteracija je složena komponenta koja se izvršava ponavljanjem nekog procesa nula ili više puta u zavisnosti da li je uslov ispunjen ili ne.

Evo kako bi izgledao naš primer prikazan JSD dijagramom:



Prednosti:

- Grafička prezentacija algoritma olakšava pronalaženje logičkih grešaka u algoritmu (kao kod dijagrama toka).
- Jednostavniji grafički simboli nego kod dijagrama toka i N/S dijagrama.
- Sledi logiku rešenja podelom na podprobleme.

Nedostaci:

1. Ne uočavaju se lako odluke i repeticije.
2. Prevođenje u programski kod je nešto složenije – mora se voditi računa o redosledu izvršavanja procesa.

Pitanja

1. Nabrojte faze u životnom ciklusu softvera.
2. Definišite algoritam.
3. Objasnite zašto se može smatrati da je Program=Algorita+Podaci.
4. Kako glasi teorema o programskoj strukturi?
5. Navedite korake u nekoj od strategija za razvoj algoritama.
6. Navedite nekoliko tehnika za prikaz algoritama.
7. Koje su osnovne komponente (grafički simboli) dijagrama toka?
8. Koje su osnovne komponente (grafički simboli) N/S dijagrama?
9. Koje su osnovne komponente (grafički simboli) JSD dijagrama?
10. Navedite komparativne prednosti i nedotatke algoritama izraženih: prirodnim jezikom, dijagramima toka, pseudokodom, N/S dijagramima, JSD dijagramima.

11 Testiranje algoritama (programa)

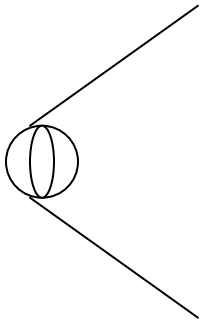
Dokaziti da algoritam obavlja zadatak ili rešava problem koji je postavljen nije nimalo lak zadatak. Iako postoje matematičke metode za takvo dokazivanje (koje se nazivaju verifikacija programa) u praksi se najčešće takve metode ne primenjuju. Umesto toga pristupa se testiranju programa. Izuzev za veoma jednostavne programe, testiranjem se ne može dokazati ispravnost programa. Testiranjem se pokušavaju pronaći greške u programu. No, bez obzira koliko se grešaka pronađe, uvek postoji mogućnost da ih ima još. I pored toga što testiranje ne garantuje odsustvo grešaka, testirani programi su bolji nego netestirani.

Kao što je već rečeno u vezi sa životnim ciklusom softvera, a i kod strategije razvoja algoritama, testiranje je veoma važna faza u razvoju svakog algoritma – programa.

Nažalost, testiranjem se ne može dokazati ispravnost programa, to jest ne može se dokazati potpuno odsustvo grešaka, ali se mogu otkriti neke od grešaka ako se primeni pravilan postupak testiranja nad pažljivo odabranim testnim podacima.

Cilj testiranja je da se pronađu okolnosti pod kojima program (algoritam) daje pogrešne rezultate ili ne da je rezultat uopšte (halting problem). Ako ne možemo da pronađemo takve okolnosti tada možemo imati razumno verovanje da program ispravno funkcioniše. Pravljenje dobrog test plana kojim se potrđuje ispravnost programa je veoma težak zadatak. Kada programi imaju hiljade linija koda (koraka) teško je testirati svaku moguću »putanju« pri njegovom izvršavanju kako bi se proverilo da program ispravno radi u svim mogućim slučajevima. Postoji nekoliko test strategija koje povećati šansu za pronalaženje grešaka (bagova) u programima.

Testiranje »bele kutije« (staklene kutije)



```
if total < 6000 then
        display('Uneti platu')
        read(plata)
        total=total+plata

else
        display('Uneti ID')
        read(id)
endif
```

Ovu vrstu testiranja može da obavlja ili programer koji je napisao program ili neko drugi kome je specijalnost testiranje softvera. Testiranje se naziva »bela kutija« zato što je onom ko testira poznat algoritam ili programski kod.

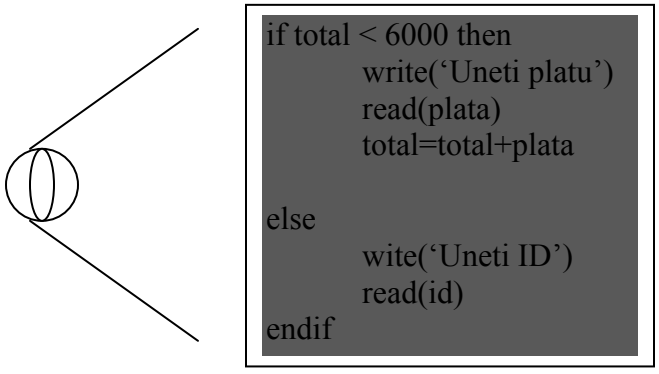
Posmatrajmo sledeći primer programa napisanog u pseudokodu:

```
if podatak < 100 then
        print('Loša ocena')
else
        if podatak <200 then
                display('Moglo je bolje')
        else
                if podatak <300 then
                        display('Dobro')
                else
                        display('Vrlo dobro')
                endif
        endif
endif
```

Pošto osoba koja testira program može da vidi program onda je u stanju da proveri saki deo programa tako što će testirati program za razne vrednosti podatka: za manje od 100, između 100 i 200, između 200 i 300, i na kraju za 300 ili više od 300.

Za testiranje programa pored uobičajenih biraju i takozvane granične vrednsoti. Granične vrednosti su one vrednosti podataka kod kojih se mogu očekivati pogrešni rezultati (nula, ekstremne vrednosti itd.)

Testiranje »crne kutije«



```
if total < 6000 then
        write('Uneti platu')
        read(plata)
        total=total+plata

else
        wite('Uneti ID')
        read(id)
endif
```

Testiranje “crne kutije” se tako naziva jer je programski kod ili algoritam nevidljiv osobi koja vrši tetsiranje. test se sprovodi tako što se programu (crnoj kutiji) daju razlišiti podaci i posmatra ponašanje algoritma. Da bi se postigao efekat testiranja, kod ove vrste testova se test podaci pripremaju pre izrade programa, imajući na umu samo programske zahteve date u psecifikaciji programa.

Za ovu vrstu testa se obično planiraju tri vrste testnih podataka :

- Ispravni podaci
- Neispravni podaci
- Granične vrednosti podataka

Za predhodni primer, ispravni podaci bi bili recimo 34, 150, 250, neispravni recimo ako umesto cifara damo slova, a graničnim slučajevima se mogu smatrati podaci 99,100,101,199,200,201,299,300 i 301.

Alfa testiranje

Softver može biti testiran bilo kojom od predhodne dve metode. Uobičajeno je da se nakon testiranje progrem pre puštanja u prodaju, da na upotrebu ograničenom broju (obično programera) u istoj kompaniji. Oni koristeći program (simulirajući krajnjeg korisnika) pronalaze eventualne greške koje su zaostale posle testiranja, a takođe mogu da daju i sugestije za unapređenje programa

Beta test testiranje

Nakon alfa testiranja uobičajena je procedura da se softver da na upotrebu ograničenom broju spolašnjih korisnika i da se od njih dobije odziv. Odziv može da sadrži pronađene greske ili takođe sugestije za funkcionalno ili neko drugo unapređenje softvera .

Tehnike testiranja

Jedna od najčešće primenjivanih tehnika testiranja kada znamo algoritam (»bela kutija«) je primena tebela za testiranje (»Trace Table«). Ovde se nad testnim podacima vrši simuliranje rada algoritma, a za svaki korak algoritma se u tabeli vrši odgovarajuća izmena podataka definisana tim korakom.

Pokazaćemo to na jednom jednostavnom primeru.

Posmatrajmo sledeći algoritam (dat u prirodnom jeziku):

1. Postaviti **pokazivač (indeks) na tekuće slovo** na vrednost 0.
2. Postaviti **pokazivač (indeks) na poslednje slovo** na vrednost koja odgovara ukupnom broju slova na stranici.
3. Postaviti **brojač** na 0.
4. Učitati slovo na koje pokazuje **pokazivač (indeks) na tekuće slovo**.
5. Ako je učitano **slovo** samoglasnik dodati 1 na **brojač.**

6. Povećati **pokazivač (indeks) na tekuće slovo** za 1.
7. Ako je **pokazivač (indeks) na tekuće slovo** <= **pokazivač (indeks) na poslednje slovo** idi na korak 4.

Ovim programom se očigledno prebrojavaju samoglasnici u tekstu ispisanom na nekoj stranici.

Kao testne podatke uzmimo da se na stranici nalazi sledeći tekst:

Taj tekst

Tekst ima 9 slova (uočite da smo i razmak (blanko znak) računali kao slovo, jer i to je jedan ASCII znak.

Sledeća tabela ilustruje kako se u svakom od koraka gornjeg algoritma menjaju vrednosti podataka (promenljivih) koje se pojavljuju u algoritmu.

Korak	**pokazivač na tekuće slovo**	**pokazivač na posledenje slovo**	**brojač**	**slovo**	**da li je slovo samoglasnik**	**da li je pokazivač na tekuće slovo <= pokazivač na poslednje slovo**
pre početka algoritma	x	x	x	x	x	x
1	1	x	x	x	x	x
2	1	9	x	x	x	x
3	1	9	0	x	x	x
4	1	9	0	T	x	x
5	1	9	0	T	ne	x
6	2	9	0	T	ne	x
7	2	9	0	T	ne	da
4	2	9	0	a	ne	da
5	2	9	1	a	da	da
6	3	9	1	a	da	da
7	3	9	1	a	da	da
4	3	9	1	j	da	da
5	3	9	1	j	ne	da
6	4	9	1	j	ne	da
7	4	9	1	j	ne	da
4	4	9	1	‘ ‘	ne	da
5	4	9	1	‘ ‘	ne	da
6	5	9	1	‘ ‘	ne	da
7	5	9	1	‘ ‘	ne	da
4	5	9	1	t	ne	da
5	5	9	1	t	ne	da
6	6	9	1	t	ne	da
7	6	9	1	t	ne	da
4	6	9	1	e	ne	da
5	6	9	2	e	da	da
6	7	9	2	e	da	da
7	7	9	2	e	da	da

4	7	9	2	k	da	da
5	7	9	2	k	ne	da
6	8	9	2	k	ne	da
7	8	9	2	k	ne	da
4	8	9	2	s	ne	da
5	8	9	2	s	ne	da
6	9	9	2	s	ne	Da
7	9	9	2	s	ne	Da
4	9	9	2	t	ne	Da
5	9	9	2	t	ne	Da
6	10	9	2	t	ne	Da
7	10	9	2	t	ne	Ne

Ovakve tabela nam pokazuju neke interesantne stvari o algoritmima. Iz tabele se jasno vidi da promenljive (podaci – varijable) ne menjaju vredosti sve dok se ne naiđe na korak u kojem se nad njima vrši neka operacija. Druga važna činjenica je da tabela postaje veoma velika kada algoritam sadrži ponavljanje, posebno nad većim skupom podataka. Veličina tabele se može smanjiti ako se kod repeticije uradi samo jedan ciklus u posebnoj tabeli a u glavnoj tabeli pokaže samo prvo i poslednje ponavljanje. Tabela takođe pokazuje i značaj inicijalizacije promenljivih na početku programa kako bi se izbegle situacije da algoritam radi sa pogrešnim podacima (pogledajte inicijalno stanje pre početka rada algoritma gde promenljive imaju nepoznatu vrednost označenu sa x).

Pitanja

1. Šta su testni podaci?
2. Šta je testiranje »bele kutije«?
3. Šta je testiranje »crne kutije«?
4. Šta je alfa test?
5. Šta je beta test?
6. Šta su testne tabele (Trace table)?
7. Kada testna tabela ima mnogo više redova nego što algoritam ima koraka?

12 Programska okruženja

Bilo ko ko želi da programira suočava se sa problemom izbora takozvanog programskog okruženja. Pod programskim okruženjem podrazumeva se skup softverskih alata pomoću kojih programer projektuje i razvija programe. Programsko okruženje uvek uključuje editor, kompajler, biblioteku predhodno razvijenih programa (run time environment), dibager (debugger). Pored ovoga, programeru mogu biti na raspolaganju i drugi softverski alati kao što su alati za analizu i projektovanje sistema (UML, na primer), alati za grafički dizajn korisničkog interfejsa itd.

Mogu se razlikovati dve vrste programskih okruženja: integrisana okruženja (integrated development environment, ili IDE skraćeno) i linijski editori komandi.

Integrisano okruženje je grafički korisnički interfejs koji sadrži sve potrebne alate za razvoj programa kao što su editor, kompajler, grafički dizajner (za forme i sl..), dibager, pojekt menadžer itd.

Okruženje sa linijskim editorom komandi je jednsotavno okruženje gde programer korišćenjem jednostavnih tekst editora unosi naredbe programskog koda, naredbe za kompilaciju i izvršavanje.

Danas se za programiranje najčešće koriste integrisana okruženja. Za programere početnike se međutim preporučuje koričćenje jednostavnih linijskih editora kako bi programeri početnici mogli da prate sve korake u izradi programa koji su u IDE ponekad sakriveni.

Naravno, IDE su znatno pogodnija za razvoj složenih programskih sistema jer uključuju i elemente upravljanja projektima.

Poznati primer koji se koristi pri izučavanju bilo kog programskog jezika je primer »Hello World« programa. To je veoma jednostavan program kojim se na ekranu štmpa poruka »Hello World«. To je dobar primer kako je ponekad jednostavnije koristiti linijke editore nego IDE za razvoj jednostavnih programa.

Evo kako taj program izgleda u C#.

```
using System;
class Hello
{
      static void Main() {
            Console.WriteLine("Hello, World");
      }
}
```

Datoteke koje sadrže C# program obično imaju ekstenziju .cs. Pod pretpostavkom da smo gore navedeni program smestili u datoteju pod imenom hello.cs, onda bismo korišćenjem linijkog editora komandi mogli koristiti komandu **csc hello.cs** kojom se poziva Microsoft-ov C# kompajler da izvrši prevođenje programa u izvršni mašinski kod koji će biti smešten u datoteci **hello.exe**.

Sada ćemo analizirati ovaj jednostavan program sa ciljem da razumevajući njegovu strukturu razumemo i čitav proces razvoja programa.

Program počinje **using System** naredbom čime se označava da želimo da u programu koristimo biblioteku klasa koju je Microsoft (.NET Framework biblioteka) razvio i stavio na raspolaganje programerima. Ova biblioteka sadrži oko 8.000 klasa podeljenih u nekoliko tematskih oblast: za rad sa podacim (System.Data) , za rad sa windows formama (System.Windows.Forms), za rad sa ulazno-izlaznim uređajima (IO) itd.

Zato što smo using System naredbu uvrstili u naš program, možemo da koristimo Console.WriteLine metodu koja je deo gornje biblioteke.

Because of the using directive, the program can use Console.WriteLine as shorthand for System.Console.WriteLine.

Dalje uočavamo da program počinje definicijom klase hello (**class hello**), koja ima samo jedan metod pod imenom **Main**. Ispred nazva Main stoje reči **static** i **void**. Reč static označava da se metoda Main ne odnosi ni na jedan poseban objekat, to jest da se static metod koristi bez reference na objekat. Reč **void** označava da metod Main ne vraća vrednost. Po konvenciji static metod pod imenom Main je ulazna tačka u svaki C# program.

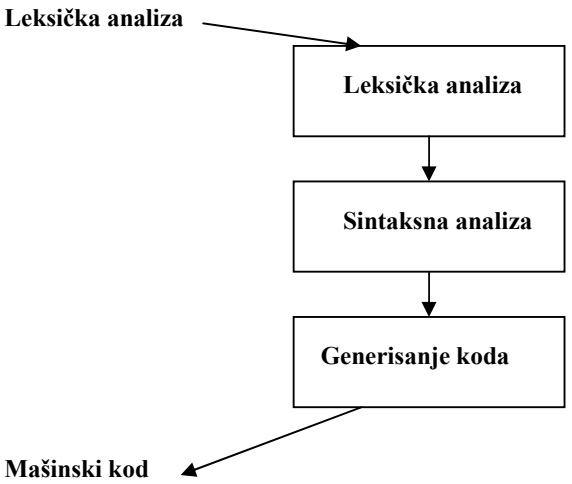
Metod Main sadrži samo jednu liniju koda, koja zapravo i nije neka C# naredba, već poziv metode **Console.WriteLine** koja je deo biblioteke System, a ta metoda će automatski biti pozvana od strane C# kompajlera.

Kompajleri i interpreteri

Postoje dva načina kako se programi nanpisani u programskijm jezicima mogu izvršavati u računarima. To su kompilacija i interpretacija.

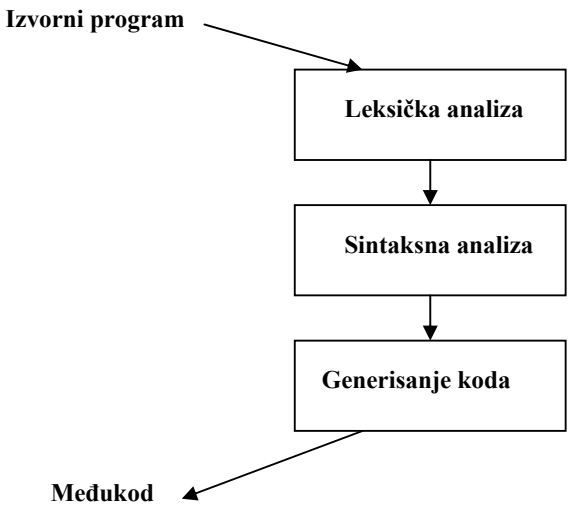
Kompilacija (koja se obavlja kompajlerima) je postupak u kome se program napisan u izvornom obliku prevodi u mašinski kod datog računara.

Sledeća slika ilustruje proces kompilacije programa od izvornog oblika to objektnog (mašinskog) koda.



Interpretacija (koja se obavlja interpreterima) je postupak u kome se program napisan u izvornom obliku prevodi u neki »međukod«, to jest neki jednostavniji jezik sličan mačinskom jeziku, a zatim računar izvršava (intrepretira) naredbe međukoda. Time se postiže da se isti međukod može izvršavati na različitim računarima jer nije zavistan od mašinskog jezika nekog specifičnog računara. Za međukod bi se moglo reći da predstavlja neki univerzalni mašinski jezik.

Sledeća slika ilustruje proces interpretacije.



Kompilirani programi obično rade brže, a prednost interpretera je što su obično iteraktivni i nije potrebno prolaziti sve faze kao kod kompilacije da bi se program izvršio.

.

Linkeri i loderi (loadres)

Linkeri i loderi su važni elementi u procesu dobijanja izvršnog (.exe) programa . Njihova uloga je da izvrše povezivanje (linkovanje) različitih internih delova programa kao i eksternih bibliotečkih potprograma u jednistvenu izvršnu celinu, kao i da tako pripremljen izvršni program prenesu (loduju) u memoriju računara kako bi počelo njegovo izvršavanje (egzekucija).

Programski editor

Programski editor je deo integrisanog okrušenja i sluši za editovanje različitih komponenti izvornog koda (programskih naredbi, formi, itd.).

Dibager (Debugger)

Dibager je softverski alat koji služi za testiranje ispravnosti rada programa. On omogućava izvršavanje programa korak-po-korak pri čemi programer može da prati promenu vredosti varijabli. Dibageri modu da na automatizovan način prave "trace table" koje smo ranije pominjali kao sredstvo za testiranje algoritama.

Dinamičko povezivanje programa (DLL)

Umesto pravljenja izvršnog programa u .exe obliku, četo se pribegava kompilaciji programa do takozvanog .dll oblika. To je skraćenica od Dynamic Link Library (dll) što znači da je ovako generisan mašinski kod moguće dinamički povezivati tokom izvršenja nekog drugog progrtama koji može da poziva .dll metode.

Povezivanje sa softverima za baze podataka (ODBC)

***O**pen **D**ata**B**ase **C**onnectivity (ODBC)*, je jedan standardni metod za povezivanje programa sa bazama podataka. Standard je razvila SQL Access grupa 1992 godine sa ciljem da se bilo kom podatku može pritupiti iy bilo koje aplikacije nezavisno od sistema za upravljanje bazama podataka (DBMS). To se postiže umetanjem drajvera za baze podataka kao interfejsa (srednjeg sloja) između aplikacije i DBMS-a. Ovi drajveri mogu se naći u u Microsoft .NET Framework biblioteci.

Korisnički interfejs

Pod korisničkim interfejsom, u užem smislu , podrazumeva se izgled ekrana koji korisnik nekog programa vidi na svom monitoru. Međutim, u širem smislu, korinsički interfejs predstavlja bilo koji način na koji čovek interaguje sa računarom. Korisničkim interfejsima se poklanja velika pažnja, jer od toga kakav je korisnički interfejs zavisi upotrebljivost programa ili lakoća sa kojom čovek-korisnik može da koristi neki softver. Dobar korisnički interfejs olakšava korisniku da obavi posao koji mu je potreban. Često se ispred reči korisnički intergejs stavlja i reč "grafički" kojom se označava da softver sadrži grafičke prikaze (forme, slike, ikone, itd.) kojima se olakšava korišćenje programa. Grafički interfejsi se označavaju sa GUI (Graphic User Interface).

Pitanja

1. Šta je programsko okruženje?
2. Koji su najčešće komponente IDE?
3. Šta je kompajler?
4. Šta je interpreter?
5. Šta je linker?
6. Šta je loder?
7. Šta je linijski editor?
8. Šta je dibager?
9. Šta je ODBC?
10. Šta je GUI?

13 Događajima upravljano, OO i GUI programiranje

Kod konvencijalnog programiranja, sekvenca operacija u nekoj aplikaciji određena je centralnim programskim modulom – glavnim programom (Main program). Kod događajima upravljanom programiranju (event-driven programming) sekvenca operacija u aplikaciji određena je korsinikovom interakcijom putem grafičkog interfejsa (formi, menija, dugmića, itd).

Program za događajima upravljanom aplikacijom ostaje u pozadini sve dok se ne desi neki događaj, kao na primer:

- Kada se promeni vrednost u nekom polju forme aktivira se program za verifikacuju ispravnosti unetog podataka.
- Kada korsinik pritisne neko dugme i time označi da je završen unos podataka koim se menja stanje u skladištu, aktivira se procedura za ažuriranje skladišta.

Događajima upravljano programiranje, grafički kosrinički intefejs (GUI) i objektno-orijentisano programiranje su međusobno povezani jer su forme i grafički interfejs objekti (boksovi, dugmići, i sl.) predstavljaju skelet čitave aplikacije.

Primer konvencionalne (konzolne) aplikacije u C#


```
using System;
using System.Collections.Generic;
using System.Text;

namespace prviprojekat
{
    class Program
    {
        static void Main(string[] args)
        {
            string naslov = "POZDRAV SVIMA";
            Console.WriteLine(naslov);
            Console.ReadKey();
        }
    }
}
```

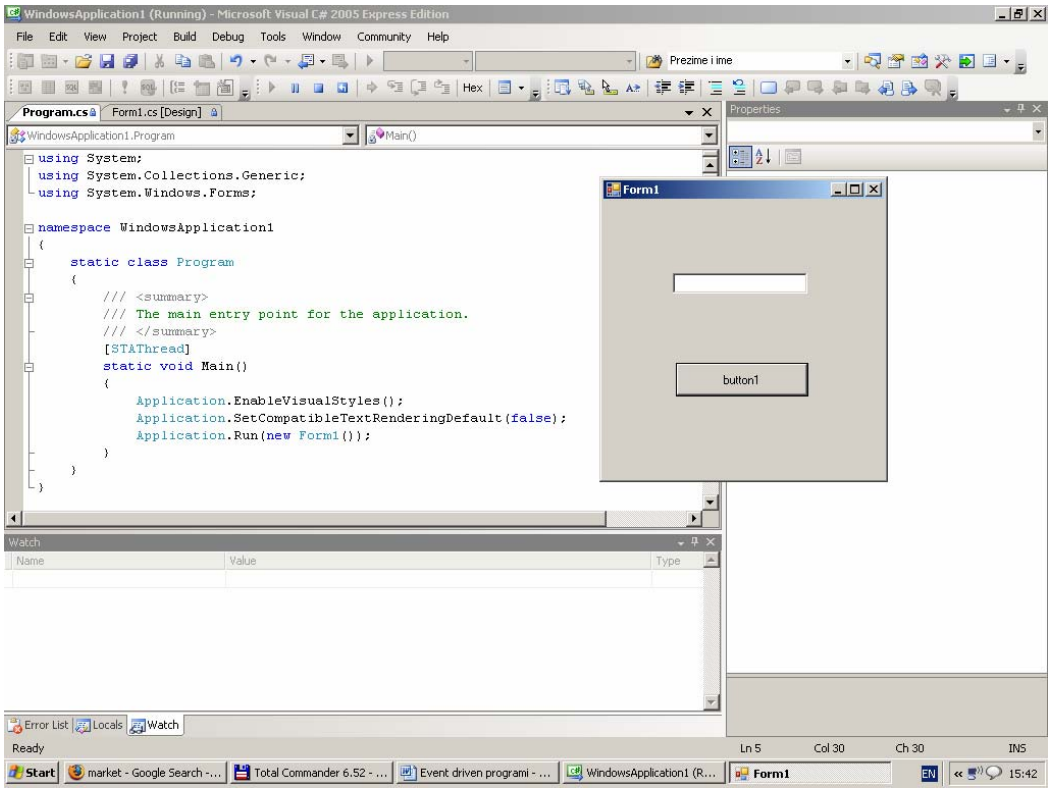
U gornjem primeru vidi se kako glavni program (Main) upravlja odvijanjem aplikacije: najpre se deklariše varijabla naslov I dodeljuje joj se vrednost "POZDRAV SVIMA", zatim se poziva procedura Console.WriteLine kojom se ispisuje na ekranu vrensot varijable naslov, I na kraju se procedurom Console.ReadKey čeka da korisnik pritisne neki taster na tastaturi pa da aplikacija završi rad.

Sledeći primer prikazuje jednu događajima vođenu aplikaciju u C#.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace WindowsApplication1
{
    static class Program
    {
```

```
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

Ovde se, za razliku od prethodnog primera, u glavnom programu samo aktivira jedan objekat tipa forme, a sva dalje događanja u aplikaciji zavisiće od interakcije korisnika i objekata prikazanih u formi. Jedan jednostavan primer forme koja sadrži samo dva objekta (button1 i tekst boks) prikazan je na sledećoj slici:



Programski segment kojim se definiše upravljanje događajima sa formom Form1 iz gornjg primera može da, na primer, glasi ovako:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
```

```
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
           Random r = new Random();
           textBox1.BackColor =
           Color.FromArgb(r.Next(0,255),r.Next(0,255),r.Next(0,255));
        }
    }
}
```

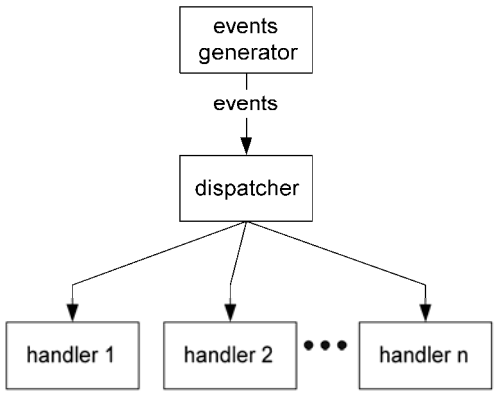
Ovde se može uočiti procedura **`button1_Click`** `kojom se definiše šta će biti urađeno kada korisnik pritisne dugme button1. U ovom slučaju svaki pritisak na dugme button1, obojiće tekst boks drugom (slučajno odabranom) bokom.`

Sada, kada smo se upoznali sa osnovnom razlikom između konvencionalnih (konzolnih) i događajima upravljanim (windows) aplikacijama, možemo se malo detaljnije pozabaviti mehanizmom izvršavanja događajima upravljanim programima (aplikacijama).

Očekivanje (listening) i opsluživanje (handling) događaja (events)

Kada kreira jednu događajima vođenu aplikaciju, programer zapravo kreira niz malih programa koje se nazivaju opslužiocima događajima (event handlers), koji se dodeljuju događajima koje pokreće neki od objekata iz aplikacije. Ovi mali programi, opslužioci, definišu se u modulu kojim se opisuju forme (vidi gornji primer).

Proces čekanja i opsluživanja događaja može biti ilustrovan sledećom slikom:



Gornji proces možemo prikazati i pseudokodom kako sledi:

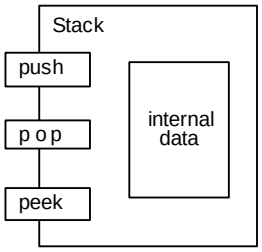
while(true) {

    if (događaj == krajAplikacije) završi;

```
        if (događaj == X) pozvatiOdgovarajućuProceduruX;

        if (događaj == Y) pozvatiOdgovarajućuProceduruY;
        …
        if (događaj == …) pozvatiOdgovarajućuProceduru…;
}
```

Objekti

Tokom 1990-tih godina objektno-orjentisana tehnologija je postepeno prevladala strukturalnu metodologiju 1970-tih i 1980-tih godina. U softversku metodologiju su uvedeni novi dijagrami kojima su opisivani objekti, različiti od dijagrama koje smo razmatrali u poglavlju 10 (blok, N/S i JSD dijagrami). U to vreme, korišćeni su takozvani ***objektni dijagrami,*** kao ovaj prikazan na sledećoj slici:



Objektni dijagram za STACK

U ovom dijagramu STACK je tip objekta. Umesto reči tip koju smo koristili kod varijabli, kod objekata se koristi reč klasa (class). Push, pop i peek su takozvane metode. Da bi koristili Stack klasu, potrebno je da kreirate stack objekat a onda da korsitite metode kojima se obavljaju operacije nad objektima. Na primer:

```
# create a stack object by instantiating the Stack class
myStack = new Stack()

myStack.push("abc")
myStack.push("xyz")

print myStack.pop()  # prints "xyz"
print myStack.peek() # prints "abc"
```

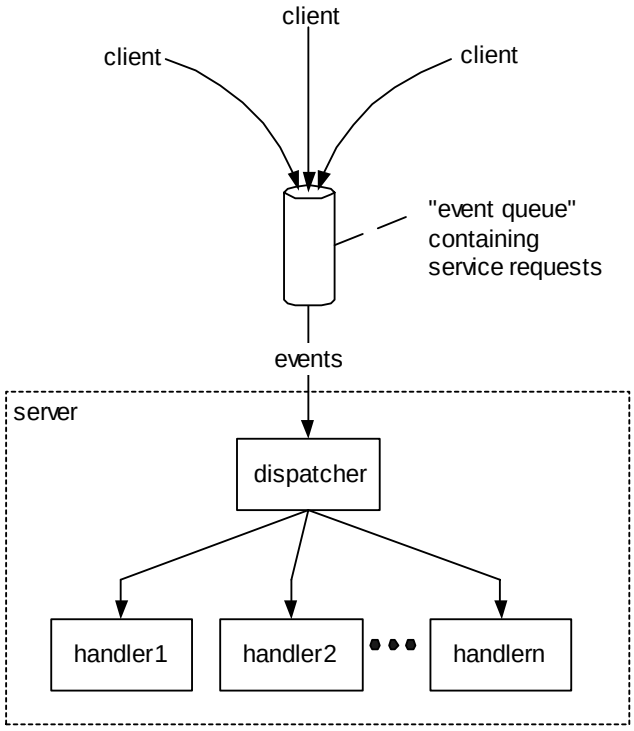
U gornjem primeru se naredbom `myStack = new Stack()` kreira jedand novi objekat sa imenom myStack, zatim se neredbama `myStack.push("abc")` i `myStack.push("xyz")` na stek postavljaju dva stringa (“abc” i “xyz”), a zatim se naredbama `print myStack.pop()` i `print myStack.peek()`štampaju isti stringovi obrnutim redom (jer je stek, setite se, LIFO lista).

Možemo da uočimo sličnost između OO programiranja i event-driven programiranja, koja se ogleda u tome da metode iz OO programa veoma podsećaju na opslužioce događaja (event handlers).

Klijent-server arhitektura

Još jedan poznati slučaj gde se pojavljuje opslušivanje događaja jeste takozvana klijent-server arhitektura. ***Server*** je neki hardverski ili softverski modul koji pruža neku uslugu (service) **klijentima**. Posao servera je da čeka na zahtev za uslugu (***service requests***) od klijenata, da odgovara na te zahteve pružanjem zahtevane usluge, a onda da nastavi čekanje na nove zahteve. Poznati primeri servera su: serveri za štampanje, fajl serveri, database serveri, aplikacioni serveri, web serveri. Na primer, kad god posetite neku novu web adresu pomoću vašeg brauzera, vaš bauzer (koji je u tom slučaju klijent) šalje zahtev web serveru na koji ovaj odgovara slanjem tražene web stranice.

Ovaj proces traženja i pružanja usluge u interakciji klijent-server može biti ilustrovan kao na sledećoj slici.

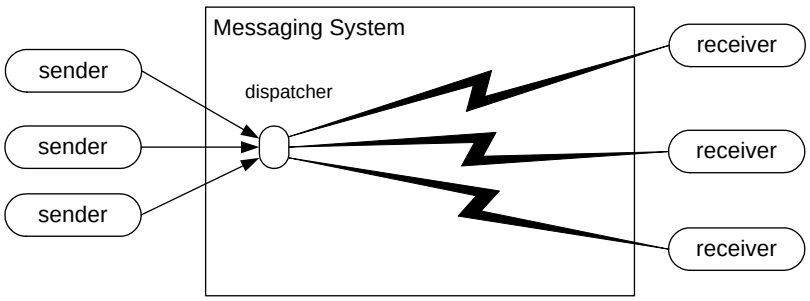


Klijent-server arhitektura

Sistemi za razmenu poruka (Messaging Systems)

Sistemi za razmenu poruka predstavljaju ekstremni slučaj opslužioca događaja (handler-a). Svrha sistema za razmenu poruka je da preuzme događaj (poruku) od generatora događaja (pošiljaoca) i prosledi ga opslužiocu (primaocu), pa je tako uloga dispečera jednostavna. Jedan poznat primer sistema za razmenu poruka su pošte. Pošiljalac predaje poruku (pismo ili paket) na poštu (sistemu za raymenu poruka). Pošta prepoznaje adresu primaoca i transportuje pošiljki na tu adresu.

Sledeća slika ilustruje ovaj proces_



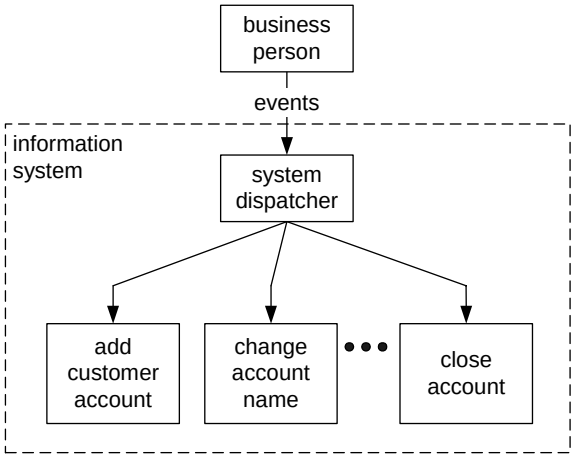
E-mail sistem vrši istu ovakvu funkciju, kao i klasična pošta, jedina razlika je što su poruke u elektronskoj a ne u fizičkoj formi.

Objektno-orjentisano događajima upravljano programiranje

Sada kada smo u stanju da vidimo potpunu sliku kako se koncept opsluživanja manifestuje u kompjuterskim sistemima, možemo da se pozabavimo I pitanjem kako to sve funkcioniše u konkretnim programima.

Posmatrajmo jedan poslovni sistem koji posluje sa mušterijama. Prirodno je da vlasnici tog biznisa žele da imaju na raspolaganju informacioni sistem u kojem mogu da memorišu, pretražuju i ažuriraju informacije o računima svojih mušterija. Oni žele sistem koji može da opsluži različite zahteve: zahtev za otvaranjem računa nove mušterije, promenu naziva nekog postojećeg računa, zatvaranje postojećeg računa itd. Takav sistem, znači mora da ima opslužioce (handler-e) za sve ove događaje.

Sledeća slika ilustruje arhitekturu jednog takvog informacionog sistema.



Pre pojave objektno orjentisanog programiranja, gornji opslužioci bi bili programski implementirani kao niz potprograma (subroutines).

Tako bi dispečerski deo programa mogao da izgleda kao što je prikazano u slegećem pseudokodu:

A

```
get eventType from the input stream

if   eventType == EndOfEventStream :
     quit # break out of event loop

if   eventType == "A" : call addCustomerAccount()
elif eventType == "U" : call setCustomerName()
elif eventType == "C" : call closeCustomerAccount()
... and so on ...
```

potprogrami kojima se opslužuju zahtevi mogli bi da izgledaju ovako:

```
subroutine addCustomerAccount():
    get customerName                  # from data-entry screen
    get next available accountNumber  # generated by the system
    accountStatus = OPEN
    # insert a new row into the account table
    SQL: insert into account
         values (accountNumber, customerName, accountStatus)
subroutine setCustomerName():
    get accountNumber       # from data-entry screen
    get customerName        # from data-entry screen
    # update row in account table
    SQL: update account
         set   customer_name = customerName
         where account_num   = accountNumber
subroutine closeCustomerAccount():
    get accountNumber       # from data-entry screen
    # update row in account table
    SQL: update account
         set   status      = CLOSED
         where account_num = accountNumber
```

Danas, korišćenjem OO tehnologije, opsluživanje događaja se implementira metodama za objekte. Tako sada, programski kod kojim se postiže isti efekat kao u predhodnom primeru može da izgleda ovako:

```
get eventType from the input stream

if   eventType == "end of event stream":
     quit # break out of event loop

if   eventType == "A" :
     get customerName                  # from data-entry screen
     get next available accountNumber  # from the system

     # create an account object
     account = new Account(accountNumber, customerName)
     account.persist()

elif eventType == "U" :
     get accountNumber       # from data-entry screen
     get customerName        # from data-entry screen

     # create an account object & load it from the database
     account = new Account(accountNumber)
     account.load()

     # update the customer name and persist the account
     account.setCustomerName(customerName)
     account.persist()

elif eventType == "C" :
     get accountNumber       # from data-entry screen

     # create an account object & load it from the database
     account = new Account(accountNumber)
     account.load()

     # close the account and persist it to the database
     account.close()
     account.persist()

... and so on ...
```

A ***Account*** klasa, sa metodama koje fukncionišu kao handler-i, može da izgleda ovako:

```
class Account:

    # the class's constructor method
    def initialize(argAccountNumber, argCustomerName):
        self.accountNumber = argAccountNumber
        self.customerName  = argCustomerName
        self.status = OPEN

    def setCustomerName(argCustomerName):
        self.customerName = argCustomerName

    def close():
        self.status = CLOSED

    def persist():
```

```
        ... code to persist an account object

    def load():
        ... code to retrieve an account object from persistent storage
```

Grafički interfejsi (GUI programiranje)

Grafički interfejsi su, danas, najčešće korišćeni okviri (frameworks) za razvoj objektno-orjentisanih i događajima upravljanih aplikacija (programa).

GUI programiranje je teško iz više razloga.

Kao prvo, potrebno je uložiti dosta truda da se specificira iygled GUI-a na ekranu kompjutera. Sve te stvarčice koje se pojavljuju na ekranu – svako dugme, labela, meni, tekst boks, list boks, itd. – moraju biti definisane preko oblika, veličine, boje pozadine, fontova itd., mora se definisati njihova početna pozicija unutar forme, kako će se ponašati ako im se promeni veličina, kao i kao se uklapaju sa ostalim objektima definisanim na ekranu. Zato se kao pomoć pri dizajnu koriste IDE alati kojima se olakšava definisanje izgleda.

Drugo, GUI programitanje je teško jer postoji jako mnogo vrsta različitih događaja koje treba opslužiti. Skoro svaki element GUI-a – svako dugme, ček-boks, radio dugme, polje za unos podataka, list boks, tekst boks, meni, itd. – predstavljaju generatore jednog ili više događaja. Osim toga i hardverski uređaji mogu da generišu događaje. Recimo, miš moše da generiše pritisak (klik) na levo dugne, desno dugme, dvsotruki klik na levo i desno dugme, itd. Slično svaki pritisak na tastaturi slova, broja ili funkcijske tipke može generisati događaje. Za sve takve događaje moraju biti programirani opslužioci (handleri).

Integrisana razvojna okruženja kao što je Microsoft Visual Studio značajno olakšavaju razvoj GUI aplikacija jer u sebi sadrže takozvane form dizajnere kojima se olakšava definisanje svih karakteristika (properties) objekata koji se koriste u interfejsu. Međutim, hendleri moraju i dalje biti programirani na način sličan konvencionalnim programima.

Pitanja

1) Kakva je razlika između konvencionalnog (proceduralnog) i događajima upravljanog programiranja?
2) Kakvu ulogu ima glavni program (Main) u konvencionalnom, a kakvu u event-driven programiranju.
3) Kako se u Microsoft Visual Studio razvojnom okruženju označavaju konvencionalne, a kako even-driven aplikacije?
4) Koje sve komponente kompjuterskog sistema mogu da generišu događaje?
5) Navedite primer jednog softverski generisanog događaja.
6) Navedite jedan primer hardverski generisanog događaja?

7) Šta je to opsluživanje događaja?
8) Šta je server?
9) Šta je klijent?
10) Šta je dispečer događaja?
11) Šta su objekti (klase)?
12) Od čega se sastoje objekti (klase)?
13) Šta je OO programiranje?
14) Šta je GUI programiranje?
15) Kakva veza postoji između događajima upravljanom, OO i GUI programiranja?

14 Objektno orjentisano (OO) programiranje

Šta je OO programiranje?

Objektno orjentisano programiranje je metodologija programiranja kojom se modelira realan svet kao skup objekata i odnosa među objektima. Osnovni element u ovoj metodologiji su, znači, objekti koji se koriste za razvoj programa (softvera).

Pomislite o objektima koji nas okužuju – automobili, ptice, drveće, ljudi itd. Ima ih bezbroj svuda oko nas. Uobičajeno je da sve objekte koji nas okružuju klasifikujemo pa otuda i jedan od osnovnih pojmova kod OO metodologije – pojam klase (Class).

Klasa (Class)

Klasa je sastavljena od osobina objekta koje se izražavaju podacima koje ih opisuju (atributa) i akcija koje objekti mogu sprovoditi (metoda).

Uzmimo na primer automobil: podaci (atributi) koji karakterišu automobile su brzina, boja, broj sedišta, itd., a aktivnsoti (metode) koje ovaj objekat može da obavlja su ubrzavanje, kočenje, promena stepena brzine (menjačem) itd.

Ili, uzmimo za primer objekte kao što su studenti. Atributi studenta su ime, prezime, broj indeksa, godina rođenja, godina studija, itd. Aktivnosti koje studenti izvršavaju su upis semestra, prijava ispita, polaganje ispita, itd.

Ili, recimo posmatrajmo automat za kafu. Kao atribute automata mozemo da uzmemo boju, listu artikala koji se nude, iznos novca u kasi, trenutnu upatu, itd. Metode kod automata mogu biti izbor artikla, ubacivanje novca, preuzimanje artikla, preuzimanje kusura, odustajanje od kupovine, itd.

Klasa opisuje sve objekte datog tipa, na taj način što definiše koji će podaci biti korišćeni za opis svakog pojedinačnog objekta. Zato je klasa apstraktna struktura bez pojedinačnih vredosti za podatke kojima se opisuju objekti.

Objekti (Object)

Objekta je jedna konkretizacija klase, odnosno objekat je jedan konkretan primerak sa konkretnim atributima koji ga razlikuju od drugih objekata iz iste klase. Na primer u klasi student, koju smo ranije opisali, može da sadrži objekat “Jovana Mirković” koja prestavlja jedan poseban slučaj objekta iz klase “Student”.

Razmena poruka (Message Passing)

Objekti ne postoje izolovano. Oni interaguju sa drugim objektima. U OO programiranju ova interakcija se ostvaruje preko poruka. Prenos poruka je proces u kome jedan objekat (pošiljalac-sender) šalje podatke drugom objektu (primaocu-receiver) ili traži od drugof objekta da aktivira neki svoj metod. Tako svaka poruka ima pošiljaoca i primaoca.

Ponašanje objekata (Behavior)

Ponašanje se odnosi na to kako objekti reaguju na poruke, to jest kako menjaju sopstveno stanje (atribute) i/ili aktiviraju svoje metode.

Pogledajmo sada kako se OOP koristi u C#.

Deklarisanje nove klase

Kada želimo da deklarišemo (definišemo, opišemo) novu klasu, u C# to činimo uz pomoć sledeće sintakse:

class < Ime Klase >
{

atributi…// podaci kojima se opisuju objekti iz klase

metode…// kojima se opisuju akcije (funkcije) koje objekti izvršavaju…

}

Ovde je *class* ključna reč. Ime klase može biti bilo koje ime koje želimo da damo klasi, ali u skaldu sa pravilima imenovanje klese

Pravila za davanje imena klasama

- Ime klase mora početi slovom iza koga može da sledi niz slova (A-Z, velika i/ili mala), cifre (0-9) i znak "donja povlaka" (_)
- Specijalna slova kao što su ? - + * / \ ! @ # $ % ^ ( ) [ ] { } , ; : . ne smeju se koristiti u imenu klase.
- Ime klase ne sme da bude neka od službenih reči C#, kao što su reči **using, public,** etc.

Konvencija za imenovanje klase

Sledeća konvencija nije obavezujuća, već predstavlja preporuku kako treba davati imena klasama.

- Ime klase treba da bude sa značenjem koje opisuje klasu, a najbolje je da bude neka imenica (na primer Student za klasu studenti, Automat za automat za kafu, i sl).
- Kod imenovanja uobičajene su dva slučaja: takozvani Pascal označavanje gde je prvo slovo u imenu veliko a sva ostala mala, ili Kamilje (Camel) označavanje gde je u imenu klase prvo slovo malo a samo prva slova iz svake naredne reči su velika Na primer:-
    - Pascal Case: Klasajedan
    - Camel Case: klasaJedan

Primer deklaracije klase

```
class Student
{
   //  Podaci - Atributi

   public string Ime;
   public int Starost;

   // Medode
   public void Prikazinaekranu()
   {
      Console.WriteLine("Name {0}\n Age {1}", Name, Age);
   }
}
```

Reč *public* označava da će atributi i metode koje imaju ovu reč biti dostupne I van same klase Student.

Deklarisanje i inicijalizacija objekta

Kada želimo da definišemo objekat iz neke klase, to radimo na način sličan deklarisanju varijabli, sa sledećom sintaksom:

<Ime klase> <Ime objekta>;

Primer:

Student S1;

Ali, pre nego što počnemo sa korišćenjem objekta, potrebno je da mu zadamo početne atribute (podatke), kroz proceduru koja se naziva inicijalizacija. Ta procedura uključuje dodelu memorije i druge osnovne procese za inicijalizaciju (konstrukciju) objekta. U C# koristimo reč *new* da označimo ovaj proces

Sintaksa je sledeća:-

Napomena: Konstruktori su posebne metode iz klase koje služe za inicijalizaciju objekta.

<Ime objekta> = new <Ime klase>;

Primer:

S1 = new Student();

Ova dva koraka (deklaracija i inicijalizacija) mogu da se obave u jednoj liniji koda kao što sledi:

Student S1 = new Student(); // Konstruktor Student()

Korišćenje objekata u OO programima

Za pistup atributima i metodama deklarisanog i inicijalizovanog objekta koristi se . (Dot) operator.

Na primer ako želimo da aktiviramo metod za prikaz na ekranu podataka objekta S1 iz klase student pisaćemo:

S1.Prikazinaekranu();

Atributima se može pristupati na sličan način, na primer:

S1.Ime = "Teofilovic Danijel";

promeniće ime objekta student S1.

Oslobađanje memorije

Da oslobodimo memoriju koju zauzima objekat S1 (koji nam više nije potreban) koristomo izraz:

S1 = null;

Evo kako bi sada mogao da izgleda jedan kompletan program koji koristi klasu Student.

```
using System;

namespace PrimerStudent
{
   class Student
{
   //  Podaci - Atributi
      public string Ime;
   public int Starost;

   // Medode
   public void Prikazinaekranu()
   {
      Console.WriteLine("Ime i prezime {0}\n Godine starosti {1}", Ime, Starost);
   }
}
```

```
class Program
   {
      static void Main(string[] args)
      {
         Student S1;                    // Deklaracija
         S1 = new Student();            // Inicijalizacija

         Student S2 = new Student();    // Deklaracija & Instantiation

         // Davanje vrednsoti atributima
         S1.Name = "Mihailo Petrovic";
         S1.Starost = 20;

         S1.Name = "Jelica Milic";
         S2.Starost = 21;

         // Aktiviranje metoda
         S1.Prikazinaekranu();
         S2. Prikazinaekranu ();
         Console.ReadLine();
      }
   }
}
```

Prednosti korišćenja OO programiranja

- **Realistično modeliranje realnosti**: Korišćenje OO metodologije daje mogućnost boljeg I verodostojnijeg modeliranja procesa iz naše realnosti.
- **Višestruko korišćenje klasa**: Jedanput dizajnirana klasa može biti iskorišćena u raznim aplikacijama.
- **Olakšano održavanje softvera**: Korišćenjem OO metodologije značajno se olakšava održavanje softvera, jer je jednostvnija identifikacija modulča koje treba menjati.

Osnovna svojstva OO metodologije

Sada ćemo se ukratko baviti nekim osnovnim svojstvima OO metodologije.

Učaurivanje, enkapsulacija (*Encapsulation*, *data hiding*)

Enkapsulacija je jedan važan koncept u OO programiranju. Spoljni svet, kao je već rečeno, pristupa samo onim atributima i metodama koje su označene kao public. Programer koji koristi neku klasu ne mora da zna kako su atributi i metode u samoj klasi kodirane. Sve što je njemu potrebno je da poznaje intefejs preko kojeg pristupa metodama i atributima klase. Naravno, potrebno je da razume i značenje atributa i metoda kojima pristupa, ali ne mora da zna kako su same metode programirane. Na taj način je obezbeđeno da se greške koje se pojave u klasi otklanjaju u samoj klasi, bez potrebe za promenom onog dela programa koji tu klasu koristi.

Apstrakcija (*Abstraction*)

Apstrakcija je proces kojim se objekti iz realnog sveta modeliraju uprošćenom "slikom" napravljenom u klasi pomoću atributa i metoda. Recimo klasa automobil koju smo napred pomenuli ne opisuje sve moguće karakteristike automobola, već neke osobine zanemaruje (apstrahuje) i ne prikazuje ih kroz atribute i metode. Koji atributi i metodi će biti odabrani za prikaz u klasi zavisi od namera koje imamo u pogledu budućeg korišćenja klase u raznim aplikacijama. Pravilno odabran model realnog sveta može da pomogne da se razumeju rešenja problema koje treba rešiti kroz softversku aplikaciju.

Kompozicija (*Composition*)

Objekti mogu da interaguju na različite načine u nekom sistemu. U nekim slučajevima klase i objekti mogu biti tako međusobno povezani da svi zajedno čine kompleksan sistem. U primeru sa automobilom točkovi, paneli, motor, menjač itd. Mogu biti posmatrani kao posebne klase. Klasa automobil, u tom slučaju, predstavlja kompoziciju ovih posebnih klasa

Nasleđivanje (*Inheritance*)

Nasleđivanje Inheritance is an interesting object-oriented programming conceptomogućava da neka klasa (sub klasa) bude bazirana na drugoj klasi (super klasa) i da od nje nasledi svu funkcionalnost. Kroz dodatni kod (atribute metode) sub klasa može biti specijalizovana za posebne potrebe. Na primer, kod klase vozila (kao super klase) možemo kreirati sub klase automobili i motorcikli. Obe ove klase nasledile bi sve metode klase vozila, ali bi takođe mogle da imaju I svoje specijalizovane metode i atribute, kao što su metoda Nagnise() i atribut Ugaonaginjanja za motorcikle.

Polimorfizam (*Polymorphism*)

Polimorfizam je sposobnost objekta da menja ponašanje u zavisnosti od načina na koji se koristi. Polimorfizam u najjednostavnijem obliku se pojavljuje kao niz metoda sa istim imenom ali sa različitim parametrima. Ovaj oblik polimorfizma naziva se "overloding" (overloading), a sreće se i kod operatora . U zavisnosti od parametara koje koristimo biće pozvan onaj metod koji odgovara korišćenim parametrima. Polimorfizam ima i druge oblike o kojima ovde neće reči.

Modularnost (*Modularity*)

IPored gore navedenih koncepata OO programiranje donosi i povećanje modularnosti programa. Individualne klase ili grupe povezanih klasa (biblioteke klasa) mogu se posmatrati kao moduli koji se mogu koristiti u raznim softverskim projektima. Na taj način se smanjuje potreba za ponovnim programiranjem sličnih zadataka, pa se tako redukuje ukupan napor potreban za razvoj sofvera.

Pitanja

1. Šta je klasa?
2. Šta je objekat?
3. Koji su osnovne komponente klase?
4. Šta je apstarkcija?
5. Šta je učaurivanje?
6. Šta je nasleđivanje? Primer.
7. Šta je polimorfizam? Primer.
8. Šta je overloding metoda? Primer.
9. Šta je overloding operatora? Primer.
10. Koje su prednosti korišćenja OO metoda?
11. Kako se postiže modularsnot u OO programiranju?

15 OOP i UML

Od proceduralnog ka OO modelu programiranja

Svi programski jezici podržavaju četiri sledeća koncepta:

- Sekvencu – kojom se izvršava niz instrukcija (naredbi, komandi)
- Selekciju – donošenje odluke (if..then...else, switch....case)
- Iteracije – ponavljanja, repeticije, petlje, ciklusa
- Apstrakcije – procesa kreiranja softvera kojim se omogućava parametrizacija rešenja čime se postiže da softver dobija generalan karakter

Upravo se na konceptu apstrakcije razlikuju proceduralni (kao što je C) i OO jezici (kao što su C++,C#,Java).

Apstrakcija kod proceduralnih jezika

Apstrakcija kod proceduralnih jezika se postiže korišćenjem potprograma (funkcija i procedura)**.**

Funkcije služe za transformaciju eksternih podataka u neki finalni rezultat, kao na primer funkcija koja određuje srednju vrednost između dva broja:

```
float SrednjaVrednost(float a, float b) {
        float result;
        result = (a+b)/2;
        return result;
}
```

Na taj način se problem top-daun (top-down) pristupom može podeliti u potprobleme (metoda poznata kao »podeli pa vladaj« - »divide and conquer«) koji se rešavaju kroz posebne procedure ili funkcije što se može izraziti na sledeći način:

$F(G(x_1,x_2,...,x_n), H(y_1,y_2,...,y_m))$

gde se funkcijom F rešava problem koji je podeljen na dva potproblema koji se rešavaju funkcijama G i H. Vrednosti $x_1,x_2,...,x_n$ i $y_1,y_2,...,y_m$ nazivaju se parametrima funkcija.

Prenos parametara

U jeziku C#, na primer, parametri mogu biti prenošeni na tri načina: po vrednosti, po referenci i kao izlazni parametri. Razmotrićemo samo prva dva načina (po vrednosti i po referenci)

Porenos parametara „po vrednosti“

Kod prenosa parametara kao vrednosti u potprogram (metodu) se prenosi vrednost koja se dalje u metodi koristi, ali se ne menja vrednost varijable koja je bila korišćena u pozivu potprograma (metode).

To je ilustrovano u sledećem primeru:

```
// PassingParams1.cs
using System;
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of myInt.
    {
        x *= x;
        Console.WriteLine("The value inside the method: {0}", x);
    }
    public static void Main()
    {
        int myInt = 5;
        Console.WriteLine("The value before calling the method: {0}",
           myInt);
        SquareIt(myInt);   // Passing myInt by value.
        Console.WriteLine("The value after calling the method: {0}",
           myInt);
    }
}
```

Output

```
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 5
```

Prenos parametara »po referenci«

Kod prenosa parametara »po referenci« u potprogram (metodu) se prenosi referenca (adresa) varijable koja se dalje u metodi koristi, pa se menja i vrednost varijable koja je bila korišćena u pozivu potprograma (metode).

To je ilustrovano u sledećem primeru:

```
// // PassingParams2.cs
using System;
class PassingValByRef
{
```

```
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of myInt.
    {
        x *= x;
        Console.WriteLine("The value inside the method: {0}", x);
    }
    public static void Main()
    {
        int myInt = 5;
        Console.WriteLine("The value before calling the method: {0}",
           myInt);
        SquareIt(ref myInt);   // Passing myInt by reference.
        Console.WriteLine("The value after calling the method: {0}",
           myInt);
    }
}
```

Output

```
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 25
```

Apstrakcija u OO jezicima

Apstrakcija u OO jezicima se postiže kroz takozvane apstraktne tipove podataka (ADT – Abstract Data Type), koji sadrže istovremeno i podatke i operacije (metode) koji se nad njima mogu vršiti.

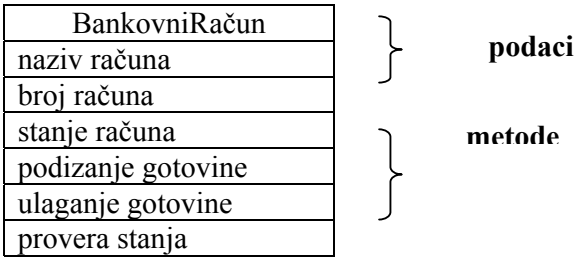
U OO jezicima apstrakcija se postiže preko programske konstrukcije koja se naziva *klasa*.

U OO terminologiji podaci se nazivaju *atributima* a operacije koje se nad njiva izvršavaju nazivaju se *metodama*.

Dakle klasa sadrži atribute i metode.

Klasa predstavlja generalizaciju (model) nekog entiteta (objekta) iz realnog sveta. Klasom se modeliraju podaci (atributi) i procedure (metode). Primeri entiteta koji se mogu modelirati klasama su: student, automobil, bankarski račun i sl.

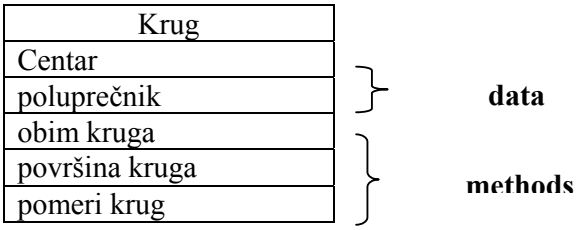
Primer klase – Bankovni račun




```
// Pseudokod za klasu Bankovni račun
class BankovniRacun {
        string nazivRacuna;
        long brojRacuna;
        float stanjeRacuna;

        podizanjeGotovine();
        ulaganjeGotovine();
        proveraStanja();
} // Klasa Bankovni račun
```

Evo još jednog primera:

Klasa Krug




```
// Pseudokod za klasu Krug
class Krug {
        double cetarX, centarY;
        double poluprecnik;

        povrsinaKruga();
        obimKruga();
        pomeriKrug();
} // Klasa Krug
```

Objekat je jedan konkretan slučaj (instanca) klase – na primer Petrov račun u banci je jedan objekat tipa bankovni račun.

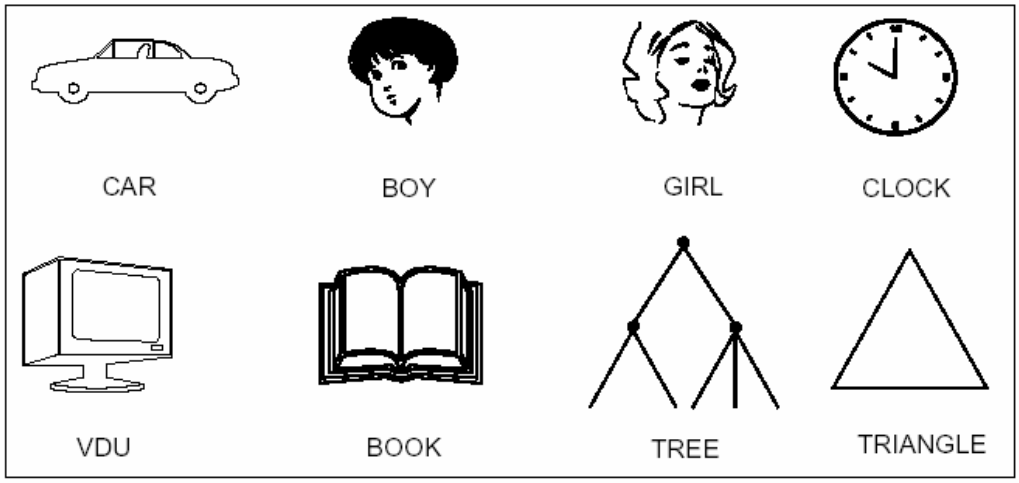
Objekat Jelenin račun je drugi objekat iz iste klase. Dakle objekti sadrže konkretne podatke dok klasa ne sadrži podatke. Klasa je samo okvir (templejt) za podatke – pokazuje koje će podatke objekat iz date klase imati.

Na primer Petrov račun može izgledati ovako:
nazivRacuna = Petar Petrovic
brojRacuna = 423838634543
stanjeRacuna = 150.850,00

Obično aplikacija koju želimo da razvijemo OO metodologijom sadrži veći broj objekata koji su međusobno povezani i u nekoj interakciji. Recimo, ako želimo da izgradimo informacioni sistem neke banke možemo imati sledeće klase: bankovni račun, korisnik usluga (mušterija), bankovni automat, krediti, itd.

Ili u slučaju trgovine, moguće klase su: skladišta, fakture, kupci, dobavljaci, itd.

Objektno orjentisani sistem je skup međusobno interagujućih objekata kao što ilustruje sledeća slika.



Primer objekata

Klase su zapravo objekti sa istim atributima i ponašanjem

Na sledećoj slici su muskarci i žene predstavljeni jedinstvenom klasom osobe, putnički i teretni automobil klasom automobili, a trouglovi, šestouganici i rombovi klasom poligoni.

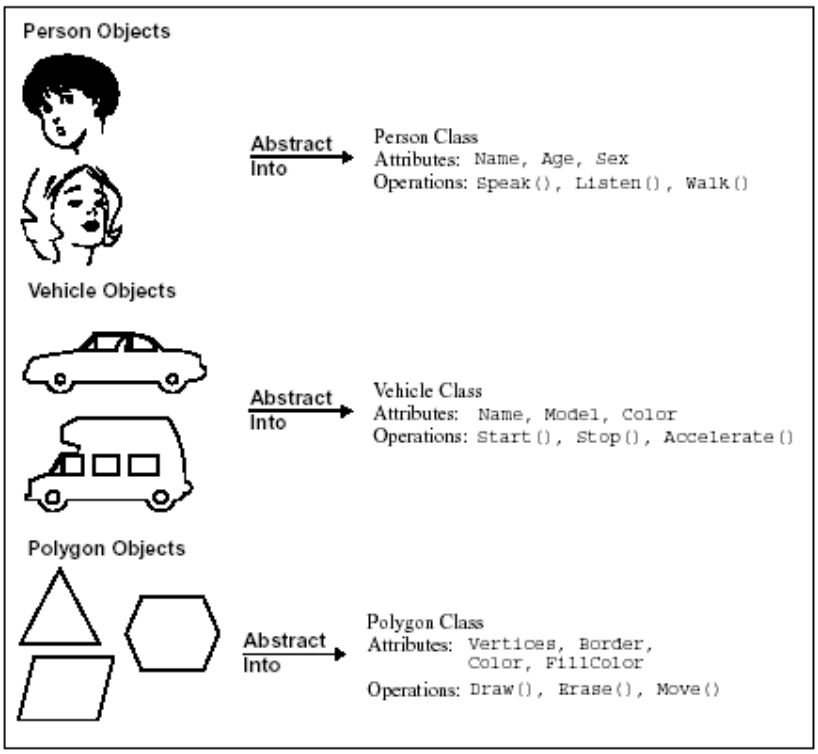


Figure 1.12: Objects and classes

Uvod u UML

Kao što smo kod proceduralnih metoda koristili grafičke simbole za prikazivanje algoritama, tako se i OO metodologiji nametnuo jedan standardni grafički način za prikazivanje klasa.

UML (Unified Modeling Language) je grafički jezik za modeliranje koji se koristi za prikaz različitih OO komponenti pri analizi, projektovanju i implementaciji OO sistema. UML je postao međunarodni standard. Njegovi autori su Grady Booch, Jim Rumbaugh and Ivar Jacobson.

Kratak istorijat razvoja UML-a

Istorijat razvoja UML je interesantan i sa aspekta ubrzanja širenja metodologija, a ne samo novih tehnoloških rešenja.
1994 – Grady Booch i Jim Rumbaugh su osnovali kompaniju Rational Rose sa novom idejama:

Grandy Booch – sa Booch Metodom
Jim Rumbaugh – Tehnikom Objektnog Modeliranja

1995 – Ivar Jacobson se pridružio timu sa idejom:

Objekt Orjentisanog Softverskog Inženjerstva

Tako je nastala prva verzija UML – 0.8

1995 Object Management Group (OMG) – proglasila je UML standardom.

1996 – U proces su se uključile i druge kompanije.

Tekuća verzija je UML 2.0

Skraćeni opis UML-u i dobar tutorial možete pronaći na adresi

http://www.smartdraw.com/tutorials/software-uml/uml.htm

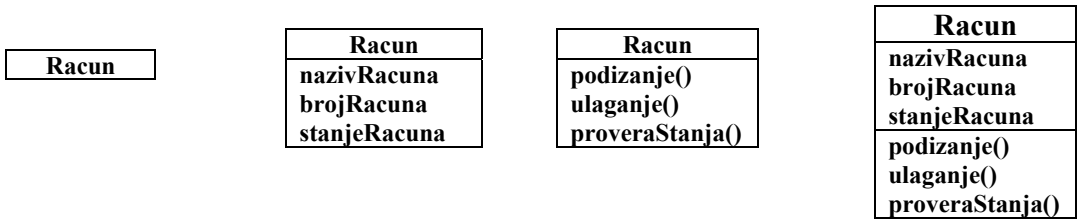
a detalje na: http://www.uml.org

Predstavljanje klasa pomoću UML-a

Klase se mogu predstaviti na jedan od sledećih načina u zavisnosti od nivoa detalja koji se želi prikazati.

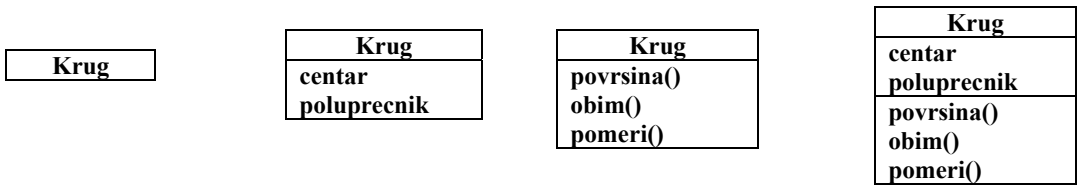
Posmatrajmo ponovo naš primer bankovnog računa. Radi skracenog pisanja malo smo promenili nazive podataka i metoda.

Klasa Racun



(a) Klasa Samo ime klase	(b) Klasa Ime klase i atributi	(c) Klasa Ime klase i metode	(d) Klasa Ime klase atributi i metode

Klasa Krug



Još o karakteristikama klasa

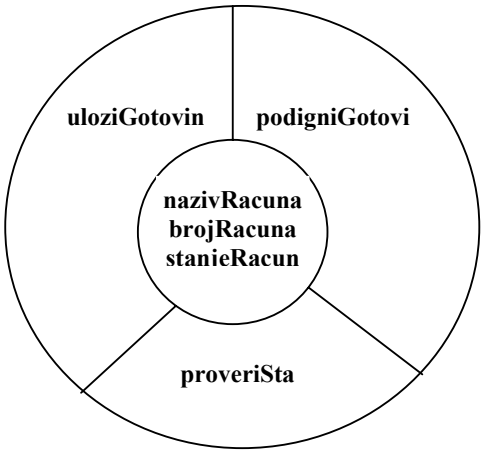
Učaurivanje (Encapsulation)

Sve informacije (atributi i metode) u OO sistemu su smeštene (sakrivene) unutar same klase (objekta).

Ovim informacija se može pristupati samo preko metoda iz klase koje su stavljene na korišćenje spoljnjem svetu kao interfejs. Sama implementacija je sakrivena od korisnika.

Mogućnost da se kod klasa (objekata) od korisnika mogu sakriti detalji naziva se učaurivanjem (enkapsulacijom). Kao da je objekat (klasa) zatvoren u kapsulu, ne može se prići njegovim unutrašnjim delovima, već se objekat koristi samo pomoću javnih metoda koje jedino imaju "pravo" da pristupaju i menjaju podatke (atribute) objekta.

Primer učaurivanja



Samo metode uloziGotovinu(), podigniGotovinu() i proveriStanje() mogu pristupati i modifikovati podatke nazivRacuna, brojRacuna i stanjeRacuna.

Pogledajmo sada kako pseudokod i UML dijagram međusobno korespondiraju.

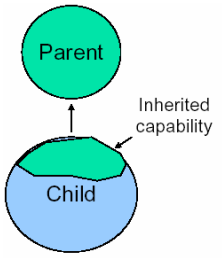
```
class Racun {
        private String nazivRacuna;
        private brojRacuna;
        private float stanjeRacuna;

        public podigniGotovinu();
        public uloziGotovinu();
        public proveriStanje();
} // Klasa Racun
```

Racun
nazivRacuna brojRacuna stanjeRacuna
podigniGotovinu() uloziGotovinu() proveriStanje()

Nasleđivanje (Inheritance)

Nove klase mogu biti kreirane korišćenjem postojećih klasa. Tako se stvara odnos Roditelj-Dete (Parent-Child), gde roditelj predstavlja postojeću klasu (super klasu) a dete novu (izvedenu) klasu (pod klasu).

Podklasa (dete) nasleđuje svojstva od superkalse (roditelja), što ilustruje sledeća slika:



Ponovo ćemo iskoristiti primer bankovnog računa da ilustrujemo nasleđivanje

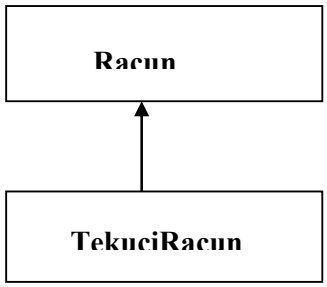
Klasa Racun – je klasa roditelj (superklasa) koja ima sledeće atribute:
nazivRacuna, brojRacuna i stanjeRacuna
No, moguće je imati raličite vrste računa u bankama – tekuće, žiro, itd.

Tako recimo, klasa TekuciRacun je nova klasa – dete (podklasa) koja pored gore navedenih atributa iz klase Racun (superklase) može imati i dodatne atribute:
brojIzdatihCekova i brojRealizovanihCekova

I sada, umesto da kreiramo novu klasu TekuciRacuni koja bi imala svih pet atributa, možemo da klasu TekuciRacuni definišemo kao podklasu klase Racun koja nasleđuje atribute nazivRacuna, brojRacuna i stanjeRacuna, kao i sve metode klase Racun.

Prikaz nasleđivanja UML dijagramima

TekuciRacun je podklasa superklase Racun. To se UML dijagramom iskazuje na sledeći način:



Drugim rečima, klasa TekuciRacun je izvedena iz klase Racun.

Nasleđivanje –Primer 2

Klasa Krug

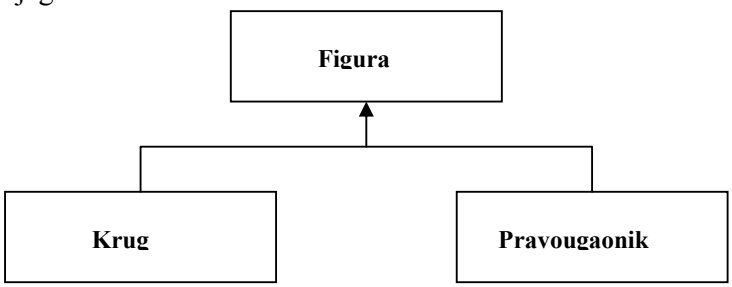
Kasa krug u našem primeru imala je atribute centar i poluprecnik.

Posmatrajmo sada klasu Pravougaonik. Ta klasa može imati atribut centar, ali poluprečnik nije prikladan atribut za ovu klasu.

Pravougaonik treba da ima atribute dužina i širina, ali oni nisu prikladni za krug. Metode površina, obim i pomeranje mogu biti primenjene i na krug i na pravougaonik (doduše sa različitim načinom izračunavanja površine i obima, ali o tome će biti reči kasnije).

Ali umesto da definišemo novu klasu Pravougaonik nezavisnu od klase Krug, mi ćemo definisati novu klasu Figura, tako što ćemo sve zajedničke atribute i metode kruga i pravougaonika smestiti u novu klasu. Onda ćemo Krug i Pravougaonik definisati kao podklase klase Figura.

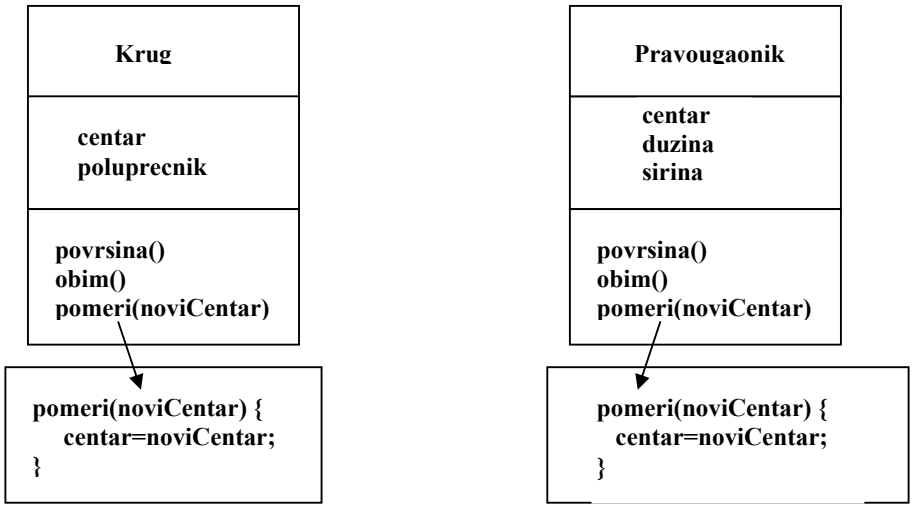
Prikažimo to UML dijagramom:



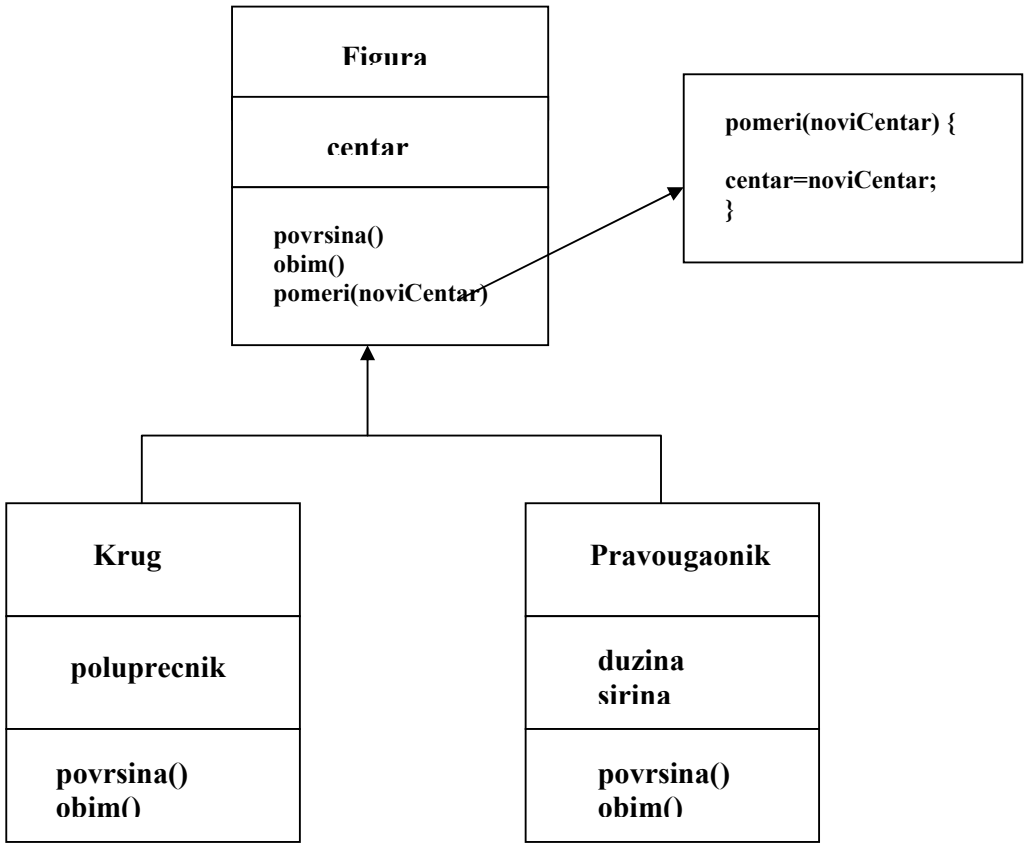
Ponovno korišćenje nasleđivanja - Reuse

Ako više klasa imaju zajedničke atribute i metode onda takve klase možemo sakupitu u jednu zajedničku klasu – superklasu (klasu roditelj). Time se postiže brži razvoj aplikacija jer se klase ne moraju nanovo programirati već se koriste već postojeće koje mogu biti sakupljene iz različitih izvora.

Primer: Pravougaonik i Krug imaju zajednički metod pomeranje jer je tada potrebno samo promeniti koordinate centra (to je atribut koji imaju i krug i pravougaonik).



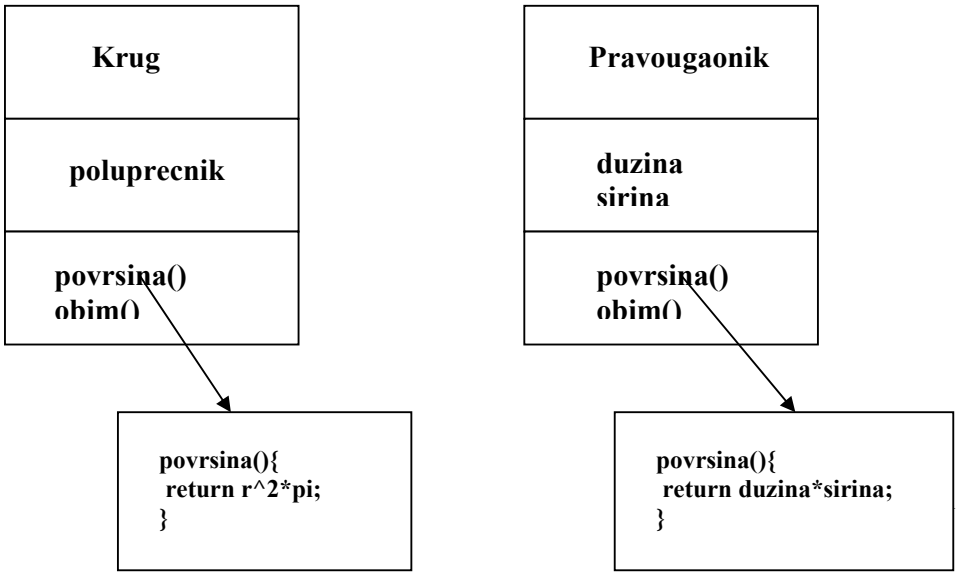
Sada bi mogli da definišemo novu klasu Figura koja će sadržati zajedničke atribute i zajedničke metode za klase Krug i Pravougaoonik, što je ilustrovano na sledećoj slici:



Korišćenje nasleđivanje za specijalizacija

Subklasi (detetu) tokom nasleđivanja mogu biti pridodata nova svojstva. Taj postupak se naziva specijalizacijom klase.

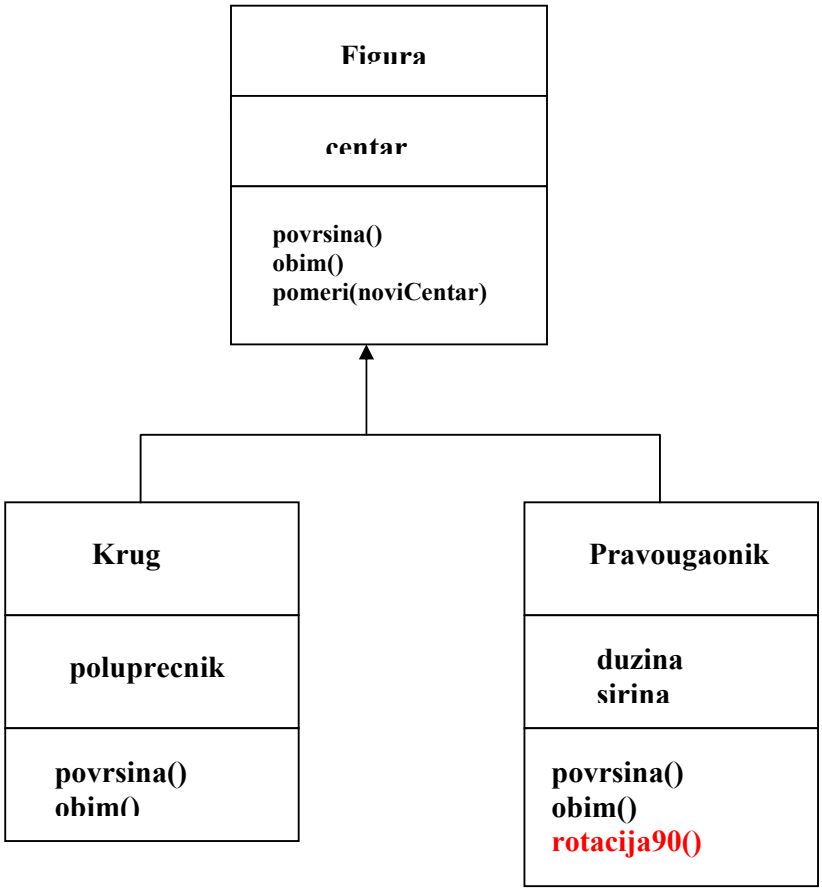
Na primer, metod povrsina je različit kod kruga i pravougaonika. Isto važi i za obim. Zato se površina i obim ponovo definišu (specijalizuju) u izvedenim klasama - Krug i Pravougaonik, kako je ilustrovano na sledećoj slici:



Korišćnje klase za proširenje funkcionalnosti postojeće klase

Klasa dete može imati i posebne metode koje nema klasa roditelj.

Na primer pravougaonimu možemo dodati novi metod koji kod kruga može biti besmislen – rotaciju za 90 stepeni, kako je to ilustrovano na sledećoj slici:



Dakle u ovom slučaju samo je dodata još jedna metoda klasi Pravougaonik, ali se ništa drugo nije promenilo.

Polimorfizam (Polymorphism)

Polimorfizam znači "više oblika"

U OOP polimorfizam i sam ima više značenja.

Polimorfizam omogućava da objekti, metode, operatori imaju rauzličita značenja zavisno od načina kako im se prenose parametri.

Na primer u sledećem pseudokodu:

krugA = new Krug(); // Kreiranje novog objekta iy klase Krug
Figura figura = krugA; // kreiranje objekta figura iz klase figura, podklase Krug
Figura.povrsina(); // povrsina ce biti izracunata prema metodi za krug

pravougaonikA = new Pravougaonik(); // Kreiranje novog objekta pravougaonika
figura= pravougaonikA;
Figura.povrsina(); // povrsina za metod pravougaonik ce se koristiti

Overloding metoda(Method Overloading)

Jedan drugi slu;aj korišćenja polimorfizma je overloding metoda, u kojem se istim imenom definiše više različitih metoda. Metode se međusobno razlikuju samo po parametrima koji im se pri pozivu prenose.

na primer neka imamo metodu inicijalizacija sa dve varijante prenosa parametara:

Metod 1 - initializacija(int a)
Metod 2 - inicijalizacija(int a, int b)

Kada u programu pozivamo metodu, od toga koje parametre stavimo u pozivu zavisiće koja od metoda (1 ili 2) će biti zapravo pozvan.

initializacija(2) // Biće pozvana metoda 1
initializacija(2,4) // Biće pozvana metoda 2

Overloding operatora (Operator Overloading)

Overloding-om operatora omogućava se davanje novog značenja operatorima kao što su +, -, *, /.

Na primer operator. + operator for Krug može biti definisan pa je izraz tipa

Krug c = c + 2;

postaje legitiman.

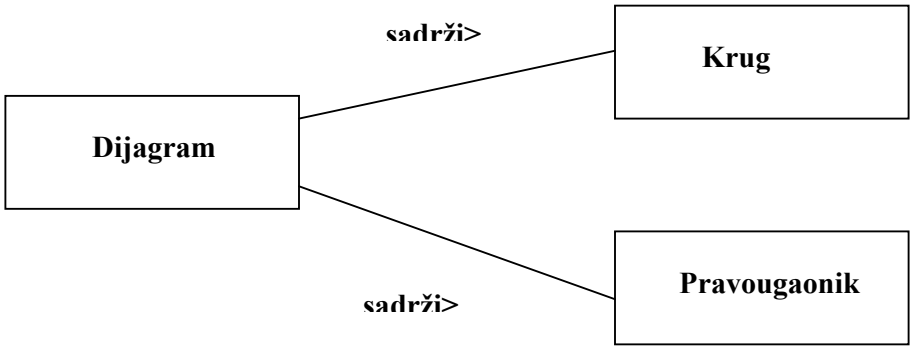
Ovo nije podržano u Java i C++ jezicima.

Asocijacija (Association)

Neka klasa može biti povezana sa drugim klasama pa je potrebno omogućiti komunikaciju među objektima iz različitih klasa.

U svhu povezivanja (asocijacije) klasa koriste se takozvani dijagrami klasa koji pokazuju međusobnu povezanost klasa i tip veze među njima.

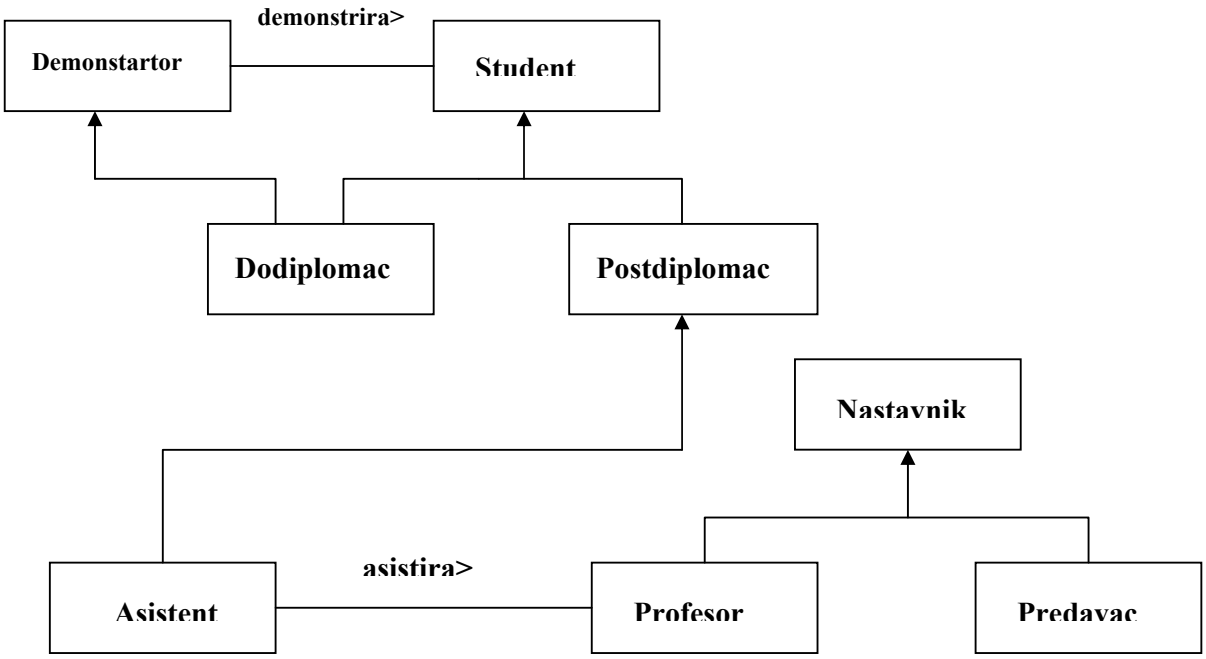
Na primer, Krug i Pravougaonik mogu biti deo klase Dijagram. Ovim se uspostavlja veza između klase Dijagram i klasa Krug i Pravougaonik kao što prikazuje sledeća alika:



Primer dijagrama klase

Nacrtati dijagram klase prema sledeém opisu:

Student može biti dodiplomac ili postdiplomac.
Dodiplomac može biti demonstrator. Demonstrator vrši demonstarcije studentima.
Predavač i profesor su dva tipa nastavnika.
Asistent je postdiplomac koji pomaže profesoru.



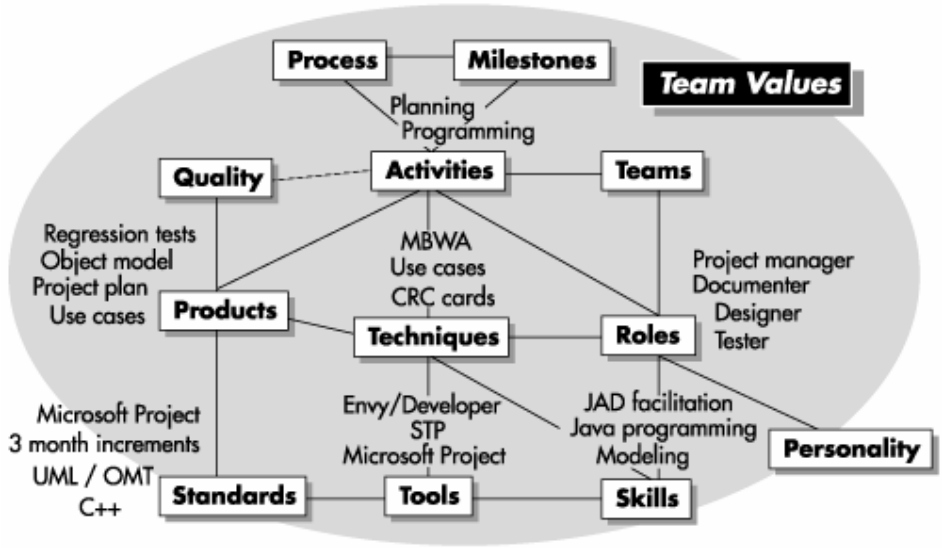
Pitanja

1. Koja su četiri osnovna koncepta programiranja?
2. Kako se princip apstrakcija izražava u proceduralnim jezicima?
3. Koji načini prenosa parametara se najčešće primenjuju u OO jezicima?
4. Koja je razlika izmešu prenosa po vrednosti i prenosa po referenci?
5. Šta je UML? Primer.
6. Šta su dijagrami klasa? Primer.

16 Metodologije za programiranje i razvoj softvera

Metodologija se može definisati kao "skup međusobno povezanih metoda i tehnika", gde se pod metodama podrazumevaju „sistematične procedure" slične tehnikama.

Metodologije za razvoj softvera bave se „metodama i tehnikama" za razne elemente koji se pojavljuju u procesu razvoja softvera. Na sledećoj slici prikazani su najvažniji elemenati razvoja softvera od kojih zavisi uspešnost svakog softverskog poduhvata.



Slika 1. Elementi metodologije razvoja softvera

Gornja slika može biti interpretirana na sledeći način.

Softverski timovi (Teams) koji se sastoje od menadžera, dizajnera, testera, dokumentalista i sl. (Roles) koji poseduju potrebna znanja i veštine (Skills) , svojim svakodnevnim aktivnostima (Activities) kao što su planiranje i programiranje kroz procese (Process) i ključne rezultate (Milestones), a korišćenjem razlišitih tehnika (Techniques) i alata (Tools) stvaraju nove oftverske proizvode (Products) određenog kavliteta (Quality) i po „de fakto" ili „de jure" standardima (Standards). Naravno, ne treba zaboraviti da su timovi sastavljeni od ljudi koji imaju svoje osobenosti (Personality) o kojima takođe treba voditi računa.

Kada se proces razvoja (proizvodnje) softvera shvati na prikazani način, onda postaje jasnije zašto je potrebno da se taj razvoj odvija prema nekoj unapred usvojenoj metodologiji. Odsustvo metodologije bi za slučaj svakog malo složenijeg softverskog poduhvata vodilo ka haosu sa negativnim posledicama.

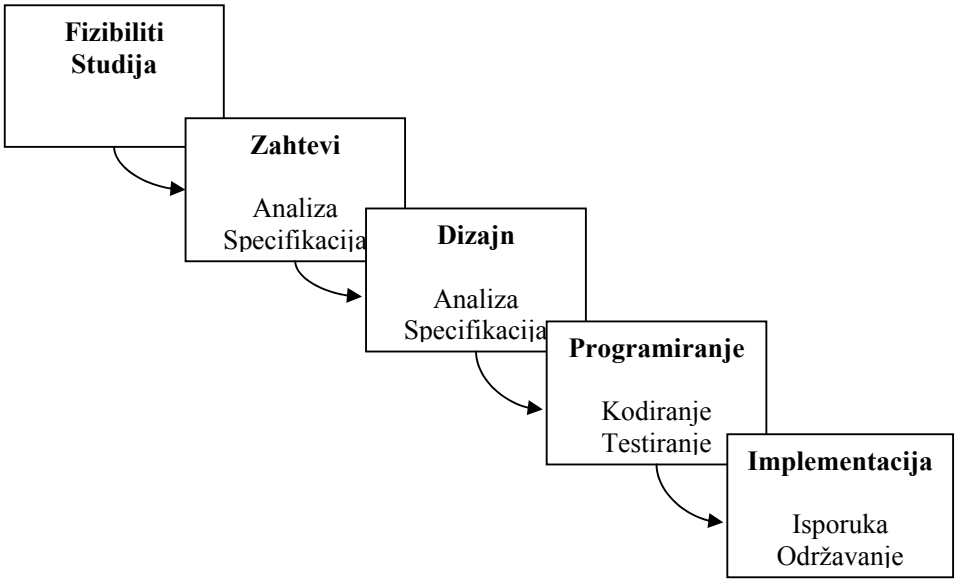
Životni ciklus razvoja softvera

Tokom vremena razvijeno je više modela za razvoj softverskih sistema poznatim pod nazivom SDLC (System Development Life Cycle) modeli.

SDLC modeli imaju za cilj da daju jedan metodičan pristup za razvoj informacionih sistema i to za sve faze razvoja od izrade studije izvodljivosti (feasibility study) , sve do održavanja gotovih aplikacija.

Ukratko ćemo navesti neke od najpoznatijih SDLC modela za razvoj softvera.

Vodopad (Waterfall) model: To je klasičan SDLC model kod kojeg se faze u razvoju softvera linearno i sekvencijalno odvijaju jedna za drugom bez preklapanja faza i/ili vraćanja (iteracije) na prethodno završenu fazu. To se može islustrovati sledećom slikom:



Slika 2 Vodopad model

Fizibiliti

Fizibiliti stufijom se određuje da li vredi uopšte započeti rad na nekom softverskom projektu. Ako je odluka da se projekat realizuje onda se već u samoj fizibiliti studiji daju osnove plana izvođenja projekta i procena budžeta (i drugih resursa) potrebnih za realizaciju.

Zahtevi

Zahtevi za izradu novog ili modifikaciju postojećeg sistema predstavljaju detaljan opis željene funkcionalnosti i drugih karakteristika softverskog sistema koji će biti razvijen.Zahtevi moraju biti što precizniji I u pisanoj formi, kako bi se na kraju projekta moglo ustanoviti da li realizovani sistem odgovara zahtevima.

Dizajn

Dizajn se može podeliti na tri dela:
Design focuses on:

- Viši nivo kojim se određuje koji su sve programski moduli potrebni, šta su njihovi ulazi/izlazi i kako oni interaguju međusobno i sa drugim softverom i operativnim sistemom.
- Niži nivo na kojem se definiše način rada programskih modula, koji algoritmi i modeli će biti korišćeni, koje su programske biblioteke potrebne i sl.
- Dizajn podataka kao što je interfejs za ulaz/izlaz podataka, strukture podataka koje će se koristiti i sl.

Do kojih će se detalja ići pri dizajnu je stvar izbora. Ako iamo veoma detaljan dizaj, pisanje programskog koda će biti mnogo lakše ali će naknadne promene ići mnogo teže, dok je za slučaj grubog (manje detaljnog) dizajna mnogo više posla ostavljeno za pisanje koda, ali je i jednostavnija njegova naknadna promena. Iynad svega je važno da dizajn bude dobro dokumentovan i da u dokumentaciji jasno piše zašto su napravljene one odluke kod kojih je bilo više opcija. Takav pristup olakšava uključivanje novih programera na projekat, a takođe olakšava dalji razvoj sistema dodavanjem novih funkcija I karakteristika.

Programiranje i testiranje

U ovoj fazi se dizajn pretvara u programski kod. Programski alati kao što su kompajleri i dibageri se pri tome koriste za generisanje izvornog koda dobrog kvaliteta a time i celokupne softverske aplikacije. Testiranje manjih delova (modula) je pogodan način za kontrolu kvaliteta i pronalaženje grešaka što je ranije moguće.
Testiranje sistema kao celine vrši se da se proveri da li sistem radi na ciljnim platformama, i da li njegovo ponašanje odgovara zathevima koji su postavljeni na početku projekta.

Održavanje

Od trenutka kada je softverski sistem isporučen korisnicima počinje i potreba za njegovim održavanjem. Mogu se pojaviti grešeke prouzrokovane pogrešno unetim podacima od strane korisnika (takve podatke uvrstiti u plan testiranja), ili zbog neočekivanog i/ili nepravilnog korišćenja softvera (takve slučajeve uvrstiti u dokumentaciju). korisnici, takođe, mogu zahtevati i dodatne funkcije koje nisu uključene u tekuću reviziju softvera, mogu tražiti da softver radi brže, ili čak postaviti pred razvojni tim i veće probleme. Proces rzvoja softvera mora biti prilagođen za promene koje takđe moraju proći kroz sve gore navedene faze.

Iterativni Razvoj (Iterative Development)

Ovde se propisuje inicijalna konstrukcija malog (pa sve većeg) dela softverskog projekta kojom se pomaže da svi oni koji su uključeni u razvoj otkriju probleme i pitanja pre nego što oni postanu suviše ozbiljni. Iterativni procesi su pogodni za komercijalni pristup razvoju jer omogućavaju da se zadovolje potrebe budućih korsnika softvera koji ne znaju da definišu šta im je potrebno. Iterativne metode

uključuju i druge ideje kao što su agilni softver I ekstremno programiranje, o čemu će biti reči kasnije.

Brzi razvoj aplikacija (Rapid Application Development - RAD): Ovaj metod baziran je na konceptu da se softverski proizvod može dobiti brže i bolje organizovanjem radionica (workshops) interesnih grupa i na taj način generišu softverski zahtevi.

Zajednički razvoj (Joint application development - JAD): Ovaj model podrazumeva uključivanje korisnika u dizajn I razvoj aplikacije kroz seriju kolaborativnih vorkšopova koji se nayivaju JAD sesijama.

Prototipski razvoj (Prototyping Model): U ovom metodu se najpre uradi prototip sistema koji predstavlja samo jednu ranu aproksimaciju finalnog sistema. Zatim se prototip testira I unapređuje sve do nivoa konačno prihvatljivog prototipa na osnovu kojeg se onda razvija željeni softverski proizvod.

Sinhronizuj i stabilizuj (Synchronize-and-Stabilize): Ovaj model se zasniva na timovima koji rade paralelno na individualnim modulima zajedničke aplikacije, povremeno (često) sinhronizuju svoj kod sa kodovima ostalih timova do postizanja stabilnog sitema, a zatim nastavljaju dalji razvoj. Ovaj metod se generalno koristi u Majkrosoftvu i predstavlja osnovu Majkrosoftvove metodologije poznate kao "Microsoft Solution Framework" (MSF).

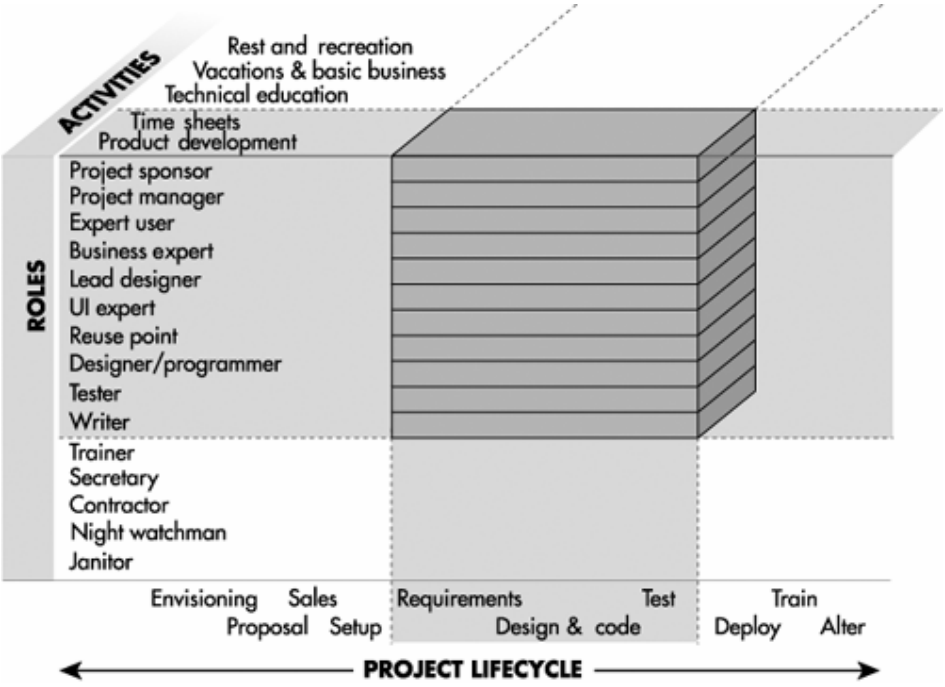
Spiralni razvoj (Spiral Model): Ovaj model razvoja softvera kombinuje vodopad i prototip modele. On je pogodan za velike, skupe i komplikovane projekte.

Organizija rada na softverskih projektima

Razvoj i proizvodnja softvera zahteva visko obrazovane stručnjake raznih profile, najsavremenija sredstva rada kao i dobru organizaciju rada. Analiziraćemo aktivnsoti i uloge članova softverskog tima, organizaciju timova i potrebne uslove rada.

Aktivnosti i uloge softverskih timova

Sledeća slika pokazuje kako se aktivnosti i uloge softverskih timova dovode u vezu sa životnim ciklusom razvoja softvera. Liste aktivnosti i uloga su samo ilustrativne i naravno ne iscrpljuju sve slučajeve.



Slika . Tri dimenzije projekta. Metodologije se bave samo delom projektnih aktivnosti.

Opisaćemo sada neke od standardnih aktivnsoti i uloga prikazanih na gornjoj slici.

Ativnosti

Razvoj proizvoda (Product Development): Ovo je centralna aktivnsot u softverskim projektima. Obuhvata već pomenute procese analize, dizajna, kodiranja, testiranja, održavanja, pisanja tehničke dokumentacije i sl.

Tehnička edukacija (Technical Education): Informacione tehnologije se veoma brzo razvijaju. Praćenje tako brzog razvoja zahteva poseban tretman u obuci i treningu članova softverskog tima. Specijalistički kursevi, seminari, konferencije, vorkšopovi su samo neki od oblika permanentnog obrazovanja neophodnog da softverski stručnjaci održe visok nivo svoje profesionalnosti.

Odmor i rekreacija (Rest and Recreation) Rad na softveru zahteva izuzetno naporno mentalno angažovanje. Borba sa vremenom za što raniji izlazak na tržište sa proizvodom koji se razvija može da bude veoma stresna. Zato softverske kuće pokušavaju da ove izuzetne napore svojih zaposlenih kompenziraju raznim oblicima komfornih radnih okruženja, sportskih i rekreacionih aktivnosti. Nije čudo što je u Kaliforniji sedište najvećih softverskih kompanija (Google, na primer) jer je Kalifornija poznata kao deo USA sa izuzetnom ponudom za rekreaciju i razonodu.

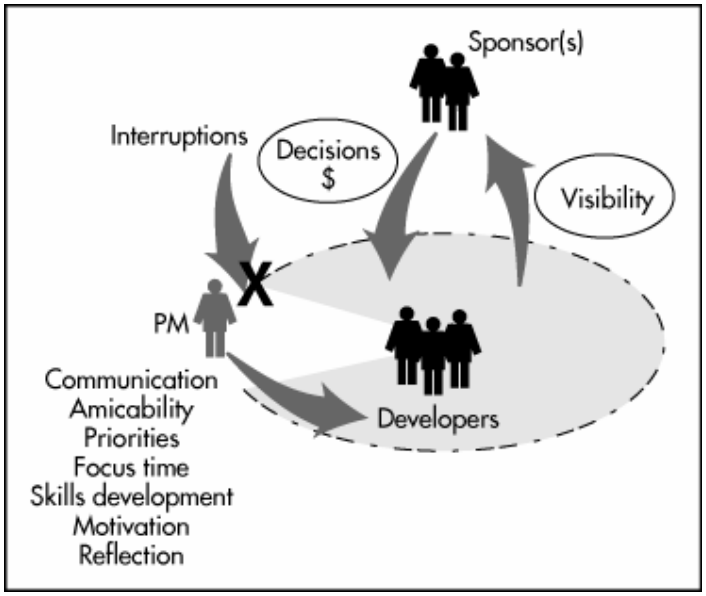
Uloge

Dizajneri/Programeri: Ovo je suštinska uloga svakog razvoja softvera. To su kreatori sistema. Oni definišu arhitekturu sistema, vrše izbor platformi, alata, metoda, algoritama, proizvode kod (source code). Može postojati više nivoa prema iskustvu (seniori, juniori, početnici, itd.) ili prema oblastima (GUI, biznis logika, baze podataka, itd).

Testeri: Testeri su veoma važna uloga kojom se postiže više efekata. Pored kontrole kvaliteta softvera koji se razvija, testeri mogu da pomognu i u podizanju ukupne produktivnsoti tima, da ukažu ne samo na reške u funkcionisanju već i da, svojim primedbama, pomognu dizajn korisničkog intefejsa, i/ili ukupne funkcionalnosti sistema.

Pisci: Pisci su oni koji proizvode hiljade stranica teksta kojim se opisuje proizvod., od prvog dokumeta kojim se proizvod najavljuje ("white paper"), pa sve do opisa sistema, uputstava za korišćenje, marketinških prezentacija i reklamiranja.

Menadžer projekta (Project Manager): To je osoba koja je najviše odgovorna za uspeh projekta. On planira aktivnosti, prati izvršenje plana, komunicira sa svim zaintersovanim stranama u projektu (članovima softverskog tima, menadžementu firme, korsnicima, itd.). Sldeća slika ilustruje rad menadžera projekta.

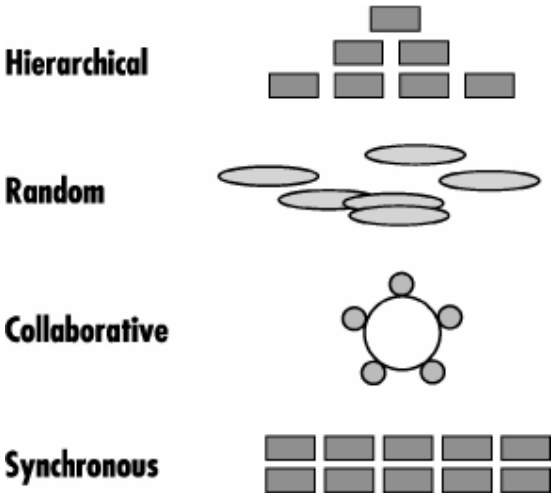


Slika Uloga menadžera projekta u agilnim SW projektima

Sponzor projekta (Project Sponsor) To je osoba koja predstavlja "spiritus movens" projekta. Projekat je "živ" sve dok su članovima tima na raspolaganju resursi (novac, ljudi i oprema) potrebni za njegovu realizaciju. Sponzor predstavlja sponu projektnog tima i finansijera projekta.

Timovi

Softverski timovi mogu biti organiyovani na više načina. Sledeća slika prikazuje tipične organizacije.



Slika Četiri principa organizacije

Kod hijerhijske organizacije uočava se jasna struktura tima sa "šefom" na vrhu i jednim ili više podređenih nivoa. (Primer: Vojna jednicia)

Random (razasuta) organizacija je organizacija u kojoj "svako radi svoj posao" bez subordinacije i definisane kolaboracije. (Primer: Fudbalska reprezentacije Srbije)

Kolaborativna organizacija zahteva povećanu komunikaciju između članova tima, nema hijerarhiju, ali deluje povezano. (Primer: Vaterpolo rprezentacija Srbije)

Sinhrona organizacija pokušava da obezbedi sinhronizovano delovanje svojih delova. Nema izraženu hijerahiju. (Primer: Sinhrono plivanje).

U softverskih firmama prisutne su sva četiri tipa organizacije, jer imaju međusobne prednosti i nedostatke. Bez obzira o kojoj se vrsti organizacije radi za timove važi pravilo da je potrebno vreme za njihovo uspostavljanje. Proces "uigravanja" se odvija kroz više faza, od formiranja tima (forming), kroz burnu fazu upoznavanja (storming), do uspostavljanja normalnog režima funkcionisanja (norming) i skladnog izvršavanja projektnih zadataka (performing).

Jedan primer moderne organizacije je takozvano "programiranje u parovima" (pair programming), gde dva programera rade simultano, kako prikazuje sledeća slika.

Slika. Programiranje u parovima

Na kraju, pominjemo i najnoviju inicijativu za metodologiju izrade softvera poznatu kao "Agilno programiranje" (Agile initiative). Ova metodologija se zasniva na principima datim u manifestu koji proklamuje ovu ideju (Agile Manifesto).

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Pitanja

1. Navedite osnovne elemente razvoja softvera za koje je potrebna metodologija?
2. Zašto je važno korišćenje metodologija?
3. Šta je životni ciklus razvoja softvera (SDLC)?
4. Koje su glavne faze SDLC-a? Objasnite svaku fazu.
5. Navedite nekoliko poznatih modela za razvoj softvera?
6. Koje su četiri poznata načina organizacije?
7. Navedite neke aktivnosti pri razvoju softvera.
8. Opišite ulogu dizajnera-programera.
9. Opišite ulogu SW testera?
10. Šta su glavne odgovornosti softverskog menadžera projekta?
11. Kakva je uloga sponzora SW projekta?
12. Navedite nekoliko uloga koje mogu imati članovi softverskog tima.
13. Šta je agilni softver?
14. Koje su prednosti (nedostaci) programiranja u parovima?